Dispense del corso di Algoritmi e Strutture Dati
Corso di laurea in Informatica

PROGETTO E ANALISI DI ALGORITMI

Rapporto Interno n. 230-98
Dipartimento di Scienze dell'Informazione

Ottobre 2011

Alberto Bertoni
Massimiliano Goldwurm

Indice

Capitolo 1

Introduzione

L'attività di programmazione può grossolanamente essere divisa in due aree distinte. La prima è chiamata Programmazione in grande e riguarda la soluzione informatica di problemi di grande dimensione (si pensi allo sviluppo di un sistema informativo di una azienda multinazionale). La seconda invece può essere chiamata Programmazione in piccolo e consiste nel trovare una buona soluzione algoritmica a specifici problemi ben formalizzati (si pensi agli algoritmi di ordinamento).

Obbiettivo di questo corso è quello di fornire una introduzione alle nozioni di base e ai metodi che sovrintendono questo secondo tipo di problematica, dedicata allo studio della rappresentazione e manipolazione dell'informazione con l'ausilio della teoria degli algoritmi e della organizzazione dei dati. Si tratta di una tipica attività trasversale che trova applicazione in tutte le aree disciplinari dell'informatica, pur essendo dotata di propri metodi e di una propria autonomia a tal punto da essere inclusa in una delle nove branche nelle quali la ACM (Association for Computing Machinery) suddivide la Computer Science: Algoritmi e strutture dati, Linguaggi di programmazione, Architetture dei calcolatori, Sistemi operativi, Ingegneria del software, Calcolo numerico e simbolico, Basi di dati e sistemi per il reperimento dell'informazione, Intelligenza artificiale, Visione e robotica.

1.1 La nozione di algoritmo

Informalmente, un algoritmo è un procedimento formato da una sequenza finita di operazioni elementari che trasforma uno o più valori di ingresso (che chiameremo anche input) in uno o più valori di uscita (rispettivamente, output). Un algoritmo definisce quindi implicitamente una funzione dall'insieme degli input a quello degli output e nel contempo descrive un procedimento effettivo che permette di determinare per ogni possibile ingresso i corrispondenti valori di uscita. Dato un algoritmo A, denoteremo con f_A la funzione che associa a ogni ingresso x di A la corrispondente uscita $f_A(x)$.

Questa corrispondenza tra input e output rappresenta il problema risolto dall'algoritmo. Formalmente, un problema è una funzione $f : D_I \longrightarrow D_S$, definita su insieme D_I di elementi che chiameremo istanze, a valori su un insieme D_S di soluzioni. Per mettere in evidenza i due insiemi e la relativa corrispondenza, un problema verrà in generale descritto usando la seguente rappresentazione:

Problema NOME
Istanza : $x \in D_I$
Soluzione : $f(x) \in D_S$

Diremo che un algoritmo A risolve un problema f se $f(x) = f_A(x)$ per ogni istanza x.

L'esecuzione di un algoritmo su un dato input richiede il consumo di una certa quantità di risorse; queste possono essere rappresentate dal tempo di computazione impiegato, dallo spazio di memoria usato, oppure dal numero e dalla varietà dei dispositivi di calcolo utilizzati. È in generale importante saper valutare la quantità di risorse consumate proprio perché un consumo eccessivo può pregiudicare le stesse possibilità di utilizzo di un algoritmo. Un metodo tradizionale per compiere questa valutazione è quello di fissare un modello di calcolo preciso e definire in base a questo la stessa nozione di algoritmo e le relative risorse consumate. In questo corso faremo riferimento a modelli di calcolo formati da un solo processore e in particolare introdurremo il modello RAM, trattando quindi unicamente la teoria degli algoritmi sequenziali. In questo contesto le risorse principali che prenderemo in considerazione sono il tempo di calcolo e lo spazio di memoria.

Possiamo così raggruppare le problematiche riguardanti lo studio degli algoritmi in tre ambiti principali:

1. *Sintesi* (detta anche disegno o progetto): dato un problema f, costruire un algoritmo A per risolvere f, cioè tale che che $f = f_A$. In questo corso studieremo alcuni metodi di sintesi, come la ricorsione, la tecnica "divide et impera", la programmazione dinamica e le tecniche "greedy".

2. *Analisi*: dato un algoritmo A ed un problema f, dimostrare che A risolve f, cioè che $f = f_A$ (correttezza) e valutare la quantità di risorse usate da A (complessità concreta). Gli algoritmi presentati nel corso saranno supportati da cenni di dimostrazione di correttezza, e saranno sviluppate tecniche matematiche per permettere l'analisi della complessità concreta. Tra queste ricordiamo in particolare lo studio di relazioni di ricorrenza mediante funzioni generatrici.

3. *Classificazione* (o complessità strutturale): data una quantità T di risorse, individuare la classe di problemi risolubili da algoritmi che usano al più tale quantità. In questo corso verranno considerate le classi **P** e **NP**, con qualche dettaglio sulla teoria della **NP-completezza**.

1.2 La complessità di un algoritmo

Due misure ragionevoli per sistemi di calcolo sequenziali sono i valori $T_A(x)$ e $S_A(x)$ che rappresentano rispettivamente il tempo di calcolo e lo spazio di memoria richiesti da un algoritmo A su input x. Possiamo considerare $T_A(x)$ e $S_A(x)$ come interi positivi dati rispettivamente dal numero di operazioni elementari eseguite e dal numero di celle di memoria utilizzate durante l'esecuzione di A sull'istanza x.

Descrivere le funzioni $T_A(x)$ e $S_A(x)$ può essere molto complicato poiché la variabile x assume valori sull'insieme di tutti gli input. Una soluzione che fornisce buone informazioni su T_A e su S_A consiste nell'introdurre il concetto di "dimensione" di una istanza, raggruppando in tal modo tutti gli input che hanno la stessa dimensione: la funzione dimensione (o lunghezza) associa a ogni ingresso un numero naturale che rappresenta intuitivamente la quantità di informazione contenuta nel dato considerato. Per esempio la dimensione naturale di un intero positivo n è $1 + \lfloor \log_2 n \rfloor$, cioè il numero di cifre necessarie per rappresentare n in notazione binaria. Analogamente, la dimensione di un vettore di elementi è solitamente costituita dal numero delle sue componenti, mentre la dimensione di un grafo è data dal numero dei suoi nodi. Nel seguito, per ogni istanza x denotiamo con $|x|$ la sua dimensione.

Si pone ora il seguente problema: dato un algoritmo A su un insieme di input I, può accadere che due istanze $x, x' \in I$ di ugual dimensione, cioè tali che $|x| = |x'|$, diano luogo a tempi di esecuzione diversi, ovvero $T_A(x) \neq T_A(x')$; come definire allora il tempo di calcolo di A in funzione della sola dimensione? Una possibile soluzione è quella di considerare il tempo peggiore su tutti gli input di dimensione n fissata; una seconda è quella di considerare il tempo medio. Possiamo allora dare le seguenti definizioni:

1. chiamiamo complessità "in caso peggiore" la funzione $T_A^p : \mathbb{N} \longrightarrow \mathbb{N}$ tale che, per ogni $n \in \mathbb{N}$,

$$T_A^p(n) = \max\{T_A(x) \mid |x| = n\};$$

2. chiamiamo invece complessità "in caso medio" la funzione $T_A^m : \mathbb{N} \longrightarrow \mathbb{R}$ tale che, per ogni $n \in \mathbb{N}$,

$$T_A^m(n) = \frac{\sum_{|x|=n} T_A(x)}{I_n}$$

dove I_n è il numero di istanze $x \in I$ di dimensione n.

In modo del tutto analogo possiamo definire la complessità in spazio nel caso peggiore $S_A^p(n)$ e nel caso medio $S_A^m(n)$.

In questo modo le complessità in tempo o in spazio diventano una funzione $T(n)$ definita sugli interi positivi, con tutti i vantaggi che la semplicità di questa nozione comporta. In particolare risulta significativa la cosiddetta "complessità asintotica", cioè il comportamento della funzione $T(n)$ per grandi valori di n; a tal riguardo, di grande aiuto sono le "notazioni asintotiche" e le tecniche matematiche che consentono queste valutazioni.

È naturale chiedersi se fornisce più informazione la complessità "in caso peggiore" o quella "in caso medio". Si può ragionevolmente osservare che le valutazioni ottenute nei due casi vanno opportunamente integrate poiché entrambe le misure hanno vantaggi e svantaggi. Ad esempio la complessità "in caso peggiore" fornisce spesso una valutazione troppo pessimistica; viceversa, la complessità "in caso medio" assume una distribuzione uniforme sulle istanze, ipotesi discutibile in molte applicazioni.

1.3 Ordini di grandezza della complessità in tempo

Il criterio principale solitamente usato per valutare il comportamento di un algoritmo è basato sull'analisi asintotica della sua complessità in tempo (nel caso peggiore o in quello medio). In particolare l'ordine di grandezza di tale quantità, al tendere del parametro n a $+\infty$, fornisce una valutazione della rapidità di incremento del tempo di calcolo al crescere delle dimensioni del problema. Tale valutazione è solitamente sufficiente per stabilire se un algoritmo è utilizzabile e per confrontare le prestazioni di procedure diverse. Questo criterio è ovviamente significativo per determinare il comportamento di un algoritmo su ingressi di grandi dimensioni mentre è poco rilevante se ci interessa conoscerne le prestazioni su input di piccola taglia. Tuttavia è bene tenere presente che una differenza anche piccola nell'ordine di grandezza della complessità di due procedure può comportare enormi differenze nelle prestazioni dei due algoritmi. Un ordine di grandezza troppo elevato può addirittura rendere una procedura assolutamente inutilizzabile anche su input di dimensione piccola rispetto allo standard usuale.

Le due seguenti tabelle danno un'idea più precisa del tempo effettivo corrispondente a funzioni di complessità tipiche che vengono spesso riscontrate nell'analisi di algoritmi. Nella prima confrontiamo i tempi di calcolo richiesti su istanze di varia dimensione da sei algoritmi che hanno una complessità in tempo rispettivamente di n, $n \log_2 n$, n^2, n^3, 2^n e 3^n, supponendo di poter eseguire una operazione elementare in un microsecondo, ovvero 10^{-6} secondi. Inoltre, usiamo la seguente notazione per rappresentare le varie unità di tempo: μs =microsecondi, ms =millisecondi, s =secondi, mn =minuti, h =ore, g =giorni, a =anni e c =secoli; quando il tempo impiegato diviene troppo lungo per essere di qualche significato, usiamo il simbolo ∞ per indicare un periodo comunque superiore al millennio.

Complessità	$n = 10$	$n = 20$	$n = 50$	$n = 100$	$n = 10^3$	$n = 10^4$	$n = 10^5$	$n = 10^6$
n	$10\mu s$	$20\mu s$	$50\mu s$	$0,1ms$	$1ms$	$10ms$	$0,1s$	$1s$
$n \log_2 n$	$33,2\mu s$	$86,4\mu s$	$0,28ms$	$0,6ms$	$9,9ms$	$0,1s$	$1,6s$	$19,9s$
n^2	$0,1ms$	$0,4ms$	$2,5ms$	$10ms$	$1s$	$100s$	$2,7h$	$11,5g$
n^3	$1ms$	$8ms$	$125ms$	$1s$	$16,6mn$	$11,5g$	$31,7a$	$\approx 300c$
2^n	$1ms$	$1s$	$35,7a$	$\approx 10^{14}c$	∞	∞	∞	∞
3^n	$59ms$	$58mn$	$\approx 10^8c$	∞	∞	∞	∞	∞

Nella seconda tabella riportiamo le dimensioni massime di ingressi che possono essere processati in un minuto dagli stessi algoritmi.

Complessità in tempo	*Max Dimensione*
n	6×10^7
$n \log_2 n$	28×10^5
n^2	77×10^2
n^3	390
2^n	25

Dall'esame delle due tabelle si verifica subito come gli algoritmi dotati di una complessità in tempo lineare o di poco superiore ($n \log n$) siano utilizzabili in maniera efficiente anche per elevate dimensioni dell'input. Per questo uno dei primi obiettivi, generalmente perseguiti nella progettazione di un algoritmo per un problema dato, è proprio quello di trovare una procedura che abbia una complessità di ordine lineare o al più $n \log n$.

Algoritmi che hanno invece una complessità dell'ordine di n^k, per $k \geq 2$, risultano applicabili solo quando la dimensione dell'ingresso non è troppo elevata. In particolare, se $2 \leq k < 3$, si possono processare in tempi ragionevoli istanze di dimensione media; mentre per $k \geq 3$ tale dimensione si riduce drasticamente e i tempi necessari per processare input di lunghezza elevata risultano inaccettabili.

Infine notiamo come algoritmi che hanno una complessità esponenziale (per esempio 2^n o 3^n) presentino tempi di calcolo proibitivi anche per dimensioni di input limitate. Per questo motivo sono considerati generalmente inefficienti gli algoritmi che hanno una complessità in tempo dell'ordine di a^n per qualche $a > 1$. Questi vengono solitamente usati solo per input particolarmente piccoli, in assenza di algoritmi più efficienti, oppure quando le costanti principali, trascurate nell'analisi asintotica, sono così limitate da permettere una applicazione su ingressi di dimensione opportuna.

Si potrebbe pensare che le valutazioni generali sopra riportate dipendano dall'attuale livello tecnologico e siano destinate ad essere superate con l'avvento di una tecnologia più sofisticata che permetta di produrre strumenti di calcolo sensibilmente più veloci. Questa opinione può essere confutata facilmente considerando l'incremento, dovuto a una maggiore rapidità nell'esecuzione delle operazioni fondamentali, delle dimensioni massime di input trattabili in un tempo fissato. Supponiamo di disporre di due calcolatori che chiamiamo $C1$ e $C2$ rispettivamente e assumiamo che $C2$ sia M volte più veloce di $C1$, dove M è un parametro maggiore di 1. Quindi se $C1$ esegue un certo calcolo in un tempo t, $C2$ esegue lo stesso procedimento in un tempo t/M. Nella seguente tabella si mostra come cresce, passando da $C1$ a $C2$, la massima dimensione di ingresso trattabile in un tempo fissato da algoritmi dotati di diverse complessità in tempo.

Complessità in tempo	*Max dim. su* $C1$	*Max dim. su* $C2$
n	d_1	$M \cdot d_1$
$n \lg n$	d_2	$\approx M \cdot d_2$ (per $d_2 \gg 0$)
n^2	d_3	$\sqrt{M} \cdot d_3$
2^n	d_4	$d_4 + \lg M$

Come si evince dalla tabella, algoritmi lineari (n) o quasi lineari ($n \lg n$) traggono pieno vantaggio dal passaggio alla tecnologia più potente; negli algoritmi polinomiali (n^2) il vantaggio è evidente ma smorzato, mentre negli algoritmi esponenziali (2^n) il cambiamento tecnologico è quasi ininfluente.

Capitolo 2

Nozioni preliminari

In questo capitolo ricordiamo i concetti matematici di base e le relative notazioni che sono di uso corrente nella progettazione e nell'analisi di algoritmi. Vengono richiamate le nozioni elementari di calcolo combinatorio e i concetti fondamentali per studiare il comportamento asintotico di sequenze numeriche.

2.1 Notazioni di base

Presentiamo innanzitutto la notazione usata in questo capitolo e nei successivi per rappresentare i tradizionali insiemi numerici:

$\mathbb{N}$ denota l'insieme dei numeri naturali;
$\mathbb{Z}$ denota l'insieme degli interi relativi;
$\mathbb{Q}$ denota l'insieme dei numeri razionali;
$\mathbb{R}$ denota l'insieme dei numeri reali;
$\mathbb{R}^+$ denota l'insieme dei numeri reali maggiori o uguali a 0;
$\mathbb{C}$ denota l'insieme dei numeri complessi.

Come è noto, dal punto di vista algebrico, $\mathbb{N}$ forma un semianello commutativo rispetto alle tradizionali operazioni di somma e prodotto; analogamente, $\mathbb{Z}$ forma un anello commutativo mentre $\mathbb{Q}$, $\mathbb{R}$ e $\mathbb{C}$ formano dei campi.

Altri simboli che utilizziamo nel seguito sono i seguenti:

per ogni $x \in \mathbb{R}$, $|x|$ denota il modulo di x;
$\lfloor x \rfloor$ rappresenta la *parte intera inferiore* di x, cioè il massimo intero minore o uguale a x;
$\lceil x \rceil$ rappresenta la *parte intera superiore* di x cioè il minimo intero maggiore o uguale a x;
$\log x$ denota il logaritmo in base e di un numero reale $x > 0$.

Le seguenti proprietà si possono dedurre dalle definizioni appena date:

per ogni $x \in \mathbb{R}$

$$x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1;$$

per ogni intero n

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

per ogni $n, a, b \in \mathbb{N}$, diversi da 0,

$$\lfloor \lfloor n/a \rfloor / b \rfloor = \lfloor n/ab \rfloor,$$

$$\lceil \lceil n/a \rceil / b \rceil = \lceil n/ab \rceil;$$

per ogni numero reale $x > 1$ e ogni intero $a > 1$

$$\lfloor \log_a(\lfloor x \rfloor) \rfloor = \lfloor \log_a x \rfloor,$$

$$\lceil \log_a(\lceil x \rceil) \rceil = \lceil \log_a x \rceil.$$

2.2 Elementi di calcolo combinatorio

Le nozioni di permutazione e combinazione di un insieme finito sono strumenti fondamentali per la soluzione di molti problemi di enumerazione e manipolazione di oggetti combinatori su cui si basa l'analisi di classici algoritmi. Per questo riprendiamo questi concetti, solitamente studiati in un corso di matematica discreta, presentando solo le definizioni e le proprietà fondamentali.

Dato un intero positivo n e un insieme finito S di k elementi, $S = \{e_1, e_2, \ldots, e_k\}$, chiamiamo *$n$-disposizione* di S una qualsiasi funzione $f : \{1, 2, \ldots, n\} \longrightarrow S$. Se tale funzione è iniettiva, f sarà detta n-disposizione *senza ripetizioni*. Se f è biunivoca una n-disposizione senza ripetizioni sarà chiamata *permutazione* dell'insieme S (in tal caso $n = k$). Nel seguito una n-disposizione sarà anche chiamata disposizione di dimensione n.

Una n-disposizione f è solitamente rappresentata dall'allineamento dei suoi elementi

$$f(1)f(2)\cdots f(n)$$

Per questo motivo una n-disposizione di un insieme S è talvolta chiamata anche *parola* (o stringa) di lunghezza n sull'alfabeto S, oppure *vettore* di dimensione n a componenti in S.

Per esempio una 4-disposizione di $\{a, b, c, d\}$ è la funzione $f : \{1, 2, 3, 4\} \longrightarrow \{a, b, c, d\}$, con $f(1) = b$, $f(2) = d$, $f(3) = c$, $f(4) = a$. Essa è rappresentata dall'allineamento, o parola, $bdca$; questa disposizione è anche una permutazione.

Se la funzione $f : \{1, 2, \ldots, n\} \longrightarrow S$ è iniettiva, allora l'allineamento corrispondente

$$f(1)f(2)\cdots f(n)$$

non contiene ripetizioni. Chiaramente in questo caso si deve verificare $n \leq k$.

Esempio 2.1
Le 2-disposizioni senza ripetizioni dell'insieme $\{a, b, c\}$ sono rappresentate dalle seguenti parole: ab, ac, ba, bc, ca, cb. ∎

Indichiamo ora con $D_{n,k}$ il numero di n-disposizioni di un insieme di k elementi; analogamente, sia $R_{n,k}$ il numero di n-disposizioni senza ripetizione di un insieme di k elementi. Si verificano allora le seguenti uguaglianze:

1. $D_{1,k} = R_{1,k} = k$;
2. poiché una n-disposizione è un elemento di S seguito da una qualsiasi disposizione di S di dimensione $n - 1$, abbiamo $D_{n,k} = k \cdot D_{n-1,k}$;
3. poiché una n-disposizione senza ripetizioni di un insieme S di k elementi è un elemento di S seguito da una disposizione di dimensione $n - 1$ di un insieme di $k - 1$ elementi, abbiamo $R_{n,k} = k \cdot R_{n-1,k-1}$.

Tali relazioni provano la seguente proprietà:

Proposizione 2.1 *Per ogni coppia di interi positivi n, k si verifica*

$$D_{n,k} = k^n, \quad R_{n,k} = k(k-1)\cdots(k-n+1) \text{ (se } n \leq k).$$

Osserviamo in particolare che il numero di permutazioni di un insieme di n elementi è $R_{n,n} = n(n-1)\cdots 2 \cdot 1$. Esso è quindi dato dalla cosiddetta funzione fattoriale, indicata con

$$n! = 1 \cdot 2 \cdot \ldots \cdot (n-1) \cdot n.$$

Ricordiamo che la nozione di fattoriale viene solitamente estesa ponendo $0! = 1$.

Siamo ora interessati a calcolare il numero di n-disposizioni di un insieme $S = \{e_1, e_2, \ldots, e_k\}$ contenenti q_1 ripetizioni di e_1, q_2 ripetizioni di e_2,..., q_k ripetizioni di e_k (quindi $q_1 + q_2 + \cdots + q_k = n$). Vale a tal riguardo la seguente proposizione:

Proposizione 2.2 *Il numero $N(n; q_1, \ldots, q_k)$ di n-disposizioni di un insieme $S = \{e_1, e_2, \ldots, e_k\}$ che contengono q_1 ripetizioni di e_1, q_2 ripetizioni di e_2,..., q_k ripetizioni di e_k è*

$$\frac{n!}{q_1! q_2! \cdots q_k!}$$

Dimostrazione. Etichettiamo in modo diverso le q_1 ripetizioni di e_1 aggiungendo a ciascun elemento un indice distinto; facciamo la stessa cosa con le q_2 ripetizioni di e_2, con le q_3 ripetizioni di e_3,..., con le q_k ripetizioni di e_k. In questo modo otteniamo n oggetti distinti. Facendo tutte le possibili permutazioni di questi n elementi otteniamo $n!$ permutazioni. Ognuna di queste individua una n-disposizione originaria che si ottiene cancellando gli indici appena aggiunti; ogni disposizione così ottenuta è individuata allora da $q_1!q_2! \cdots q_k!$ distinte permutazioni. Di conseguenza possiamo scrivere $N(n; q_1, \ldots, q_k) \cdot q_1!q_2! \cdots q_k! = n!$, da cui l'asserto. ∎

Per ogni $n \in \mathbb{N}$ e ogni k-pla di interi $q_1, q_2, \ldots, q_k \in \mathbb{N}$ tali che $n = q_1 + q_2 + \ldots + q_k$, chiamiamo *coefficiente multinomiale* di grado n l'espressione

$$\binom{n}{q_1 q_2 \cdots q_k} = \frac{n!}{q_1! q_2! \cdots q_k!}.$$

Nel caso particolare $k = 2$ otteniamo il tradizionale *coefficiente binomiale*, solitamente rappresentato nella forma

$$\binom{n}{j} = \frac{n!}{j!(n-j)!},$$

dove $n, j \in \mathbb{N}$ e $0 \leq j \leq n$. Osserva che $\binom{n}{j}$ può anche essere visto come il numero di parole di lunghezza n, definite su un alfabeto di due simboli, nelle quali compaiono j occorrenza del primo simbolo e $n - j$ del secondo.

La proprietà fondamentale di questi coefficienti, dalla quale deriva il loro nome, riguarda il calcolo delle potenze di polinomi:

per ogni $n \in \mathbb{N}$ e ogni coppia di numeri u, v,

$$(u+v)^n = \sum_{k=0}^{n} \binom{n}{k} u^k v^{n-k};$$

Inoltre, per ogni k-pla di numeri $u_1, u_2, \ldots, u_k$,

$$(u_1 + u_2 + \ldots + u_k)^n = \sum_{q_1+q_2+\ldots+q_k=n} \binom{n}{q_1 q_2 \cdots q_k} u_1^{q_1} u_2^{q_2} \cdots u_k^{q_k},$$

dove l'ultima sommatoria si intende estesa a tutte le k-ple $q_1, q_2, \ldots, q_k \in \mathbb{N}$ tali che $q_1 + q_2 + \ldots + q_k = n$.

Dati due interi k, n tali che $0 \leq k \leq n$, chiamiamo *combinazione semplice* di classe k, o k-combinazione, di n oggetti distinti un sottoinsieme di k elementi scelti fra gli n fissati. Qui assumiamo la convenzione che ogni elemento possa essere scelto al più una volta.

È bene osservare che una combinazione è un insieme di oggetti e non un allineamento; quindi l'ordine con il quale gli elementi vengono estratti dall'insieme prefissato non ha importanza e due combinazioni risultano distinte solo quando differiscono almeno per un oggetto contenuto. Per esempio le combinazioni di classe 3 dell'insieme $\{a, b, c, d\}$ sono date da

$$\{a, b, c\}, \{a, b, d\}, \{a, c, d\}, \{b, c, d\}.$$

Proposizione 2.3 *Per ogni coppia di interi $n, k \in \mathbb{N}$ tali che $0 \leq k \leq n$, il numero di combinazioni semplici di classe k di un insieme di n elementi è dato dal coefficiente binomiale*

$$\binom{n}{k} = \frac{n!}{k!(n-k!)}.$$

Dimostrazione. È sufficiente dimostrare che i sottoinsiemi di $\{1, 2, \ldots, n\}$ contenenti k elementi sono $\binom{n}{k}$. A tal riguardo osserva che ogni sottoinsieme A di $\{1, 2, \ldots, n\}$ è individuato dalla sua funzione caratteristica:

$$\chi_A(x) = \begin{cases} 1 & \text{se } x \in A \\ 0 & \text{se } x \notin A \end{cases}$$

Tale funzione caratteristica è rappresentata da una n-disposizione dell'insieme $\{0, 1\}$ contenente k ripetizioni di 1 e $n - k$ di 0. Per la proposizione precedente il numero di tali disposizioni è $\frac{n!}{k!(n-k)!} = \binom{n}{k}$. ∎

Dati due interi positivi n, k, consideriamo un insieme S di n oggetti distinti; chiamiamo *combinazione con ripetizione* di classe k un insieme di k elementi scelti in S con la convenzione che ogni elemento possa essere scelto più di una volta. Nota che anche in questo caso non teniamo conto dell'ordine con il quale gli oggetti vengono scelti. Inoltre, poiché ogni elemento può essere scelto più volte, si può verificare $k > n$. Per esempio, se consideriamo l'insieme $S = \{a, b\}$, le combinazioni con ripetizione di classe 3 sono date da:

$$\{a, a, a\}, \{a, a, b\}, \{a, b, b\}, \{b, b, b\}$$

Proposizione 2.4 *Il numero di combinazioni con ripetizione di classe k estratte da un insieme di n elementi è dato dal coefficiente binomiale*

$$\binom{n+k-1}{k}.$$

Dimostrazione. Dato un insieme di n elementi, siano $s_1, s_2, \ldots, s_n$ i suoi oggetti allineati secondo un ordine qualsiasi. Sia C una combinazione con ripetizione di classe k di tale insieme. Essa può essere rappresentata da una stringa di simboli ottenuta nel modo seguente a partire dalla sequenza $s_1 s_2 \cdots s_n$:

- per ogni $i = 1, 2, \ldots, n$, affianca a s_i tanti simboli $*$ quanti sono gli elementi s_i che compaiono nella combinazione considerata;

- togli dalla sequenza ottenuta il primo simbolo s_1 e tutti gli indici dai simboli rimanenti.

In questo modo abbiamo costruito una disposizione di dimensione $n + k - 1$ nella quale compaiono k occorrenze di $*$ e $n - 1$ occorrenze di s. Per esempio se $n = 5$, $k = 7$ e $C = \{s_1, s_1, s_2, s_5, s_5, s_4, s_5\}$, la stringa ottenuta è $**s*ss*s**$.

Viceversa, è facile verificare che ogni parola di questo tipo corrisponde a una combinazione con ripetizione di n oggetti di classe k. Esiste quindi una corrispondenza biunivoca tra questi due insiemi di strutture combinatorie. Poiché $\binom{n+k-1}{k}$ è il numero di disposizioni di dimensione $n+k-1$ contenenti k occorrenze di un dato elemento e $n-1$ di un altro diverso dal precedente, la proposizione è dimostrata. ∎

Esercizi

1) Dimostrare che per ogni $n \in \mathbb{N}$ vale l'uguaglianza

$$\sum_{k=0}^{n} \binom{n}{k} = 2^n$$

2) Consideriamo un'urna contenente N palline di cui H bianche e le altre nere ($H \leq N$). Supponiamo di eseguire n estrazioni con sostituzione (ovvero, ad ogni estrazione la palline scelta viene reinserita nell'urna). Qual è la probabilità di estrarre esattamente k palline bianche?

3) Consideriamo un mazzo di carte tradizionale formato da 13 carte per ognuno dei quattro semi. Scegliamo nel mazzo 10 carte a caso (estrazione senza sostituzione). Qual è la probabilità che tra le carte scelte ve ne siano 5 di cuori?

2.3 Espressioni asintotiche

Come abbiamo visto nel capitolo precedente, l'analisi asintotica di un algoritmo può essere ridotta alla valutazione del comportamento asintotico di una sequenza di interi $\{T(n)\}$ dove, per ogni $n \in \mathbb{N}$, $T(n)$ rappresenta la quantità di una certa risorsa consumata su un input di dimensione n (nel caso peggiore o in quello medio).

Lo studio del comportamento asintotico può essere fortemente agevolato introducendo alcune relazioni tra sequenze numeriche che sono divenute di uso corrente in questo ambito.

Siano f e g due funzioni definite su $\mathbb{N}$ a valori in $\mathbb{R}^+$.

1. Diciamo che $f(n)$ è "o grande" di $g(n)$, in simboli

$$f(n) = O(g(n)),$$

se esistono $c > 0$, $n_0 \in \mathbb{N}$ tali che, per ogni $n > n_0$, $f(n) \leq c \cdot g(n)$; si dice anche che $f(n)$ ha ordine di grandezza minore o uguale a quello di $g(n)$.

Per esempio, applicando la definizione e le tradizionali proprietà dei limiti, si verificano facilmente le seguenti relazioni:

$$5n^2 + n = O(n^2), \;\; 3n^4 = O(n^5), \;\; n \log n = O(n^2),$$

$$log^k n = O(n) \;\text{ e }\; n^k = O(e^n) \;\text{ per ogni } k \in \mathbb{N}.$$

2. Diciamo che $f(n)$ è "omega grande" di $g(n)$, in simboli

$$f(n) = \Omega(g(n)),$$

se esistono $c > 0$, $n_0 \in \mathbb{N}$ tali che, per ogni $n > n_0$, $f(n) \geq c \cdot g(n)$; si dice anche che $f(n)$ ha ordine di grandezza maggiore o uguale a quello di $g(n)$.

Per esempio, si verificano facilmente le seguenti relazioni:

$$10n^2 \log n = \Omega(n^2), \quad n^{1/k} = \Omega(\log n) \text{ e } e^{n^{1/k}} = \Omega(n) \text{ per ogni intero } k > 0.$$

3. Diciamo infine che $f(n)$ e $g(n)$ hanno lo stesso ordine di grandezza, e poniamo

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

se esistono due costanti $c, d > 0$ e un intero $n_0 \in \mathbb{N}$ tali che, per ogni $n > n_0$,

$$c \cdot g(n) \leq f(n) \leq d \cdot g(n).$$

Per esempio, è facile verificare le seguenti relazioni:

$$5n^2 + n = \Theta(n^2), \quad 100n \log^2 n = \Theta(n \log^2 n), \quad e^{n+50} = \Theta(e^n), \quad \log(1 + \frac{2}{n}) = \Theta(\frac{1}{n}),$$

$$\lfloor \log n \rfloor = \Theta(\log n), \quad \lceil n^2 \rceil = \Theta(n^2), \quad n(2 + \sin n) = \Theta(n), \quad \sqrt{n+5} = \Theta(\sqrt{n}).$$

Dalle definizioni si deducono subito le seguenti proprietà:

- $f(n) = O(g(n))$ se e solo se $g(n) = \Omega(f(n))$;
- $f(n) = \Theta(g(n))$ se e solo se $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$.

Inoltre, f è definitivamente minore o uguale a una costante $p \in \mathbb{R}^+$ se e solo se $f(n) = O(1)$. Analogamente, f è definitivamente maggiore o uguale a una costante $p \in \mathbb{R}^+$ se e solo se $f(n) = \Omega(1)$.

Ovviamente le relazioni O e Ω godono della proprietà riflessiva e transitiva, ma non di quella simmetrica.

Invece Θ gode delle proprietà riflessiva, simmetrica e transitiva e quindi definisce una relazione di equivalenza sull'insieme delle funzioni che abbiamo considerato. Questo significa che Θ ripartisce tale insieme in classi di equivalenza, ciascuna delle quali è costituita da tutte e sole le funzioni che hanno lo stesso ordine di grandezza.

È possibile inoltre definire una modesta aritmetica per le notazioni sopra introdotte:

- se $f(n) = O(g(n))$ allora $c \cdot f(n) = O(g(n))$ per ogni $c > 0$;
- se $f_1(n) = O(g_1(n))$ e $f_2(n) = O(g_2(n))$ allora

$$f_1(n) + f_2(n) = O(g_1(n) + g_2(n)),$$
$$f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n)),$$

mentre non vale $f_1(n) - f_2(n) = O(g_1(n) - g_2(n))$.

Le stesse proprietà valgono per Ω e Θ.

Si possono introdurre ulteriori relazioni basate sulla nozione di limite. Consideriamo due funzioni f e g definite come sopra e supponiamo che $g(n)$ sia maggiore di 0 definitivamente.

4) Diciamo che $f(n)$ è asintotica a $g(n)$, in simboli $f(n) \sim g(n)$, se

$$\lim_{n\to+\infty} \frac{f(n)}{g(n)} = 1.$$

Per esempio:

$$3n^2 + \sqrt{n} \sim 3n^2, \;\; 2n\log n - 4n \sim 2n\log n, \;\; \log(1+\frac{3}{n}) \sim \frac{3}{n}.$$

5) Diciamo che $f(n)$ è "o piccolo" di $g(n)$, in simboli $f(n) = o(g(n))$, se

$$\lim_{n\to+\infty} \frac{f(n)}{g(n)} = 0;$$

diremo anche che $f(n)$ ha un ordine di grandezza inferiore a quello di $g(n)$. Per esempio:

$$10n\log n = o(n^2), \;\; \lceil \frac{n^2}{\log n} \rceil = o(n^2), \;\; \log^k n = o(n^\epsilon) \;\text{ per ogni }\; k, \epsilon > 0.$$

Le seguenti proprietà si deducono facilmente dalle definizioni:

- $f(n) \sim g(n)$ se e solo se $|f(n) - g(n)| = o(g(n))$;
- $f(n) \sim g(n)$ implica $f(n) = \Theta(g(n))$, ma il viceversa non è vero;
- $f(n) = o(g(n))$ implica $f(n) = O(g(n))$, ma il viceversa non è vero.

Inoltre osserviamo che anche $\sim$ definisce una relazione di equivalenza sull'insieme delle funzioni considerate; questa suddivide l'insieme in classi ciascuna delle quali contiene esattemente tutte le funzioni asintotiche a una funzione data.

Come ultimo esempio ricordiamo la nota formula di Stirling che fornisce l'espressione asintotica del fattoriale di un intero naturale:

$$n! = \sqrt{2\pi}\, n^{n+\frac{1}{2}}\, e^{-n} \left(1 + O(\frac{1}{n})\right)$$

e quindi

$$\log n! = n\log n - n + \frac{1}{2}\log n + \log\sqrt{2\pi} + O(\frac{1}{n}).$$

Esercizi

1) Mostrare mediante controesempi che le relazioni O e Ω non sono simmetriche.
2) Determinare due funzioni $f(n)$ e $g(n)$ tali che $f(n) = \Theta(g(n))$ e il limite $\lim_{n\to+\infty} \frac{f(n)}{g(n)}$ non esiste.
3) Mostrare che, se $f(n) \sim c \cdot g(n)$ per qualche $c > 0$, allora $f(n) = \Theta(g(n))$.

2.4 Stima di somme

Data una funzione $f : \mathbb{N} \to \mathbb{R}^+$, l'espressione $\sum_{k=0}^{n} f(k)$ rappresenta la somma

$$\sum_{k=0}^{n} f(k) = f(0) + f(1) + \cdots + f(n).$$

Essa definisce chiaramente una nuova funzione $S : \mathbb{N} \to \mathbb{R}^+$ che associa a ogni $n \in \mathbb{N}$ il valore $S(n) = \sum_{k=0}^{n} f(k)$.

L'analisi di semplici algoritmi richiede spesso la valutazione di somme di questo tipo; ad esempio, una stima del tempo di calcolo richiesto dall'istruzione

$$\texttt{for } i = 0 \texttt{ to } n \texttt{ do } C$$

per un comando C qualsiasi (che non modifica il valore di n), è data da

$$\sum_{k=0}^{n} c(k)$$

dove $c(k)$ è il tempo di calcolo richiesto per l'esecuzione di C quando la variabile i assume il valore k.

Osserviamo subito che l'ordine di grandezza di una somma può essere dedotto dall'ordine di grandezza dei suoi addendi.

Proposizione 2.5 *Siano f e g due funzioni definite su $\mathbb{N}$ a valori in $\{x \in \mathbb{R} \mid x > 0\}$ e siano F e G le loro funzioni somma, cioè $F(n) = \sum_{i=0}^{n} f(i)$ e $G(n) = \sum_{i=0}^{n} g(i)$ per ogni $n \in \mathbb{N}$. Allora $f(n) = \Theta(g(n))$ implica $F(n) = \Theta(G(n))$.*

Dimostrazione. La proprietà è una conseguenza della definizione di Θ. Per l'ipotesi, esistono due costanti positive c, d e un intero $k > 0$ tali che $c \cdot g(i) \leq f(i) \leq d \cdot g(i)$ per ogni $i > k$. Definiamo $A = \sum_{i=0}^{k} f(i)$ e $B = \sum_{i=0}^{k} g(i)$; nota che A e B sono maggiori di 0. Mostriamo ora che esiste una costante $D > 0$ tale che $F(n) \leq D \cdot G(n)$ per ogni $n > k$. Infatti, per tali n abbiamo

$$F(n) = A + \sum_{i=k+1}^{n} f(i) \leq A + d \sum_{i=k+1}^{n} g(i) = \frac{A \cdot B}{B} + d \sum_{i=k+1}^{n} g(i)$$

Definendo ora la costante $D = \max\{A/B, d\}$ otteniamo

$$F(n) \leq D \cdot B + D \sum_{i=k+1}^{n} g(i) = D \cdot G(n)$$

Ragionando in maniera analoga possiamo determinare una costante $C > 0$ tale che $F(n) \geq C \cdot G(n)$ per ogni $n > k$; infatti per tali n vale l'uguaglianza

$$F(n) \geq A + c \sum_{i=k+1}^{n} g(i) = \frac{A \cdot B}{B} + c \sum_{i=k+1}^{n} g(i)$$

e scegliendo $C = \min\{A/B, c\}$ otteniamo

$$F(n) \geq C \cdot B + C \sum_{i=k+1}^{n} g(i) = C \cdot G(n)$$

Abbiamo quindi provato che per due costanti positive C, D e per ogni $n > k$

$$C \cdot G(n) \leq F(n) \leq D \cdot G(n)$$

e quindi $F(n) = \Theta(G(n))$. ∎

Esempio 2.2
Vogliamo valutare l'ordine di grandezza della somma

$$\sum_{k=1}^{n} k \log\left(1+\frac{3}{k}\right).$$

Poiché $k \log\left(1+\frac{3}{k}\right) = \Theta(1)$, applicando la proposizione precedente otteniamo

$$\sum_{k=1}^{n} k \log\left(1+\frac{3}{k}\right) = \Theta(\sum_{k=1}^{n} 1) = \Theta(n).$$

∎

2.4.1 Serie geometrica

Alcune sommatorie ricorrono con particolare frequenza nell'analisi di algoritmi; in molti casi il loro valore può essere calcolato direttamente. Una delle espressioni più comuni è proprio la somma parziale della nota serie geometrica che qui consideriamo nel campo dei numeri reali. Osserva che la proprietà seguente vale per un campo qualsiasi.

Proposizione 2.6 *Per ogni numero reale ρ*

$$\sum_{k=0}^{n} \rho^k = \begin{cases} n+1 & se\ \rho = 1 \\ \frac{\rho^{n+1}-1}{\rho-1} & se\ \rho \neq 1 \end{cases}$$

Dimostrazione. Se $\rho = 1$ la proprietà è ovvia. Altrimenti, basta osservare che per ogni $n \in \mathbb{N}$,

$$(\rho-1)(\rho^n + \rho^{n-1} + \cdots + \rho + 1) = \rho^{n+1} - 1.$$

∎

La proposizione implica che la serie geometrica $\sum_{k=0}^{+\infty} \rho^k$ è convergente se e solo se $-1 < \rho < 1$; essa consente inoltre di derivare il valore esatto di altre somme di uso frequente.

Esempio 2.3
Supponiamo di voler valutare la sommatoria

$$\sum_{k=0}^{n} k2^k.$$

Consideriamo allora la funzione

$$t_n(x) = \sum_{k=0}^{n} x^k$$

e osserviamo che la sua derivata è data da

$$t'_n(x) = \sum_{k=0}^{n} kx^{k-1}.$$

Questo significa che

$$2t'_n(2) = \sum_{k=0}^{n} k2^k$$

e quindi il nostro problema si riduce a valutare la derivata di $t_n(x)$ in 2. Poiché $t_n(x) = \frac{x^{n+1}-1}{x-1}$ per ogni $x \neq 1$, otteniamo

$$t'_n(x) = \frac{(n+1)x^n(x-1) - x^{n+1} + 1}{(x-1)^2}$$

e di conseguenza

$$\sum_{k=0}^{n} k2^k = (n-1)2^{n+1} + 2.$$

∎

Esercizi

1) Determinare, per $n \to +\infty$, l'espressione asintotica di

$$\sum_{k=0}^{n} kx^k$$

per ogni $x > 1$.

2) Determinare il valore esatto della sommatoria:

$$\sum_{k=0}^{n} k^2 3^k.$$

2.4.2 Somme di potenze di interi

Un'altra somma che occorre frequentemente è data da

$$\sum_{k=0}^{n} k^i$$

dove $i \in \mathbb{N}$. Nel caso $i = 1$ si ottiene facilmente l'espressione esplicita della somma.

Proposizione 2.7 *Per ogni* $n \in \mathbb{N}$

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

Dimostrazione. Ragioniamo per induzione su n. Se $n = 0$ la proprietà è banalmente verificata. Supponiamola vera per $n \in \mathbb{N}$ fissato; allora otteniamo

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

L'uguaglianza è quindi vera anche per $n + 1$ e la proposizione risulta pertanto dimostrata. ∎

Esempio 2.4 Somme di quadrati

Possiamo ottenere un risultato analogo per la somma dei primi n quadrati, cioè $\sum_{k=0}^{n} k^2$. A tale scopo presentiamo una dimostrazione basata sul metodo di "perturbazione" della somma che consente di ricavare l'espressione esatta di $\sum_{k=0}^{n} k^i$ per ogni intero $i > 1$.

Definiamo $g(n) = \sum_{k=0}^{n} k^3$. È chiaro che $g(n+1)$ può essere espresso nelle due forme seguenti:

$$g(n+1) = \sum_{k=0}^{n} k^3 + (n+1)^3$$

$$g(n+1) = \sum_{k=0}^{n} (k+1)^3 = \sum_{k=0}^{n} (k^3 + 3k^2 + 3k + 1).$$

Uguagliando la parte destra delle due relazioni si ottiene

$$\sum_{k=0}^{n} k^3 + (n+1)^3 = \sum_{k=0}^{n} k^3 + 3\sum_{k=0}^{n} k^2 + 3\sum_{k=0}^{n} k + n + 1.$$

Possiamo ora semplificare e applicare la proposizione precedente ottenendo

$$3\sum_{k=0}^{n} k^2 = (n+1)^3 - 3\frac{n(n+1)}{2} - n - 1$$

da cui, svolgendo semplici calcoli, si ricava

$$\sum_{k=0}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}.$$

Questa uguaglianza consente di ottenere la seguente espressione asintotica

$$\sum_{k=0}^{n} k^2 = \frac{n^3}{3} + \Theta(n^2).$$

∎

Esercizi

1) Applicando lo stesso metodo usato nella dimostrazione precedente, provare che, per ogni $i \in \mathbb{N}$,

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

2) Per ogni $i, n \in \mathbb{N}$, sia $g_i(n) = \sum_{k=0}^{n} k^i$. Esprimere il valore esatto di $g_i(n)$ come funzione di i, n e di tutti i $g_j(n)$ tali che $0 \leq j \leq i-1$. Dedurre quindi una procedura generale per calcolare $g_i(n)$ su input i e n.

2.4.3 Stima mediante integrali

Nelle sezioni precedenti abbiamo presentato alcune tecniche per ottenere il valore esatto delle somme più comuni. Descriviamo ora un semplice metodo, più generale dei precedenti, che in molti casi permette di ottenere una buona stima asintotica di una somma senza cacolarne il valore esatto. Si tratta sostanzialmente di approssimare la sommatoria mediante un integrale definito.

Proposizione 2.8 *Sia $f : \mathbb{R}^+ \to \mathbb{R}^+$ una funzione monotona non decrescente. Allora, per ogni $a \in \mathbb{N}$ e ogni intero $n \geq a$, abbiamo*

$$f(a) + \int_a^n f(x)dx \leq \sum_{k=a}^{n} f(k) \leq \int_a^n f(x)dx + f(n)$$

Dimostrazione. Se $n = a$ la proprietà è banale. Supponiamo allora $n > a$. Osserviamo che la funzione è integrabile in ogni intervallo chiuso e limitato di $\mathbb{R}^+$ e inoltre, per ogni $k \in \mathbb{N}$,

$$f(k) \leq \int_k^{k+1} f(x)dx \leq f(k+1).$$

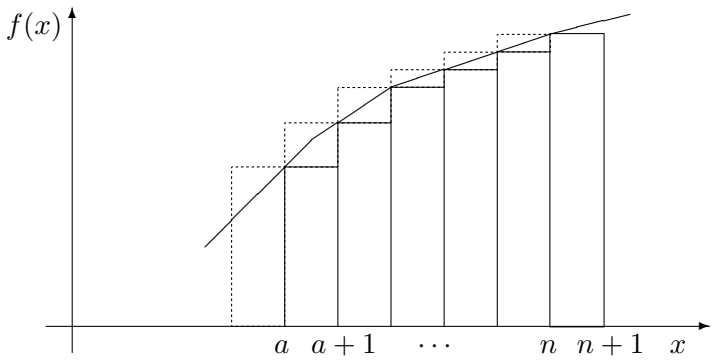
Sommando per $k = a, a+1, \ldots, n-1$, otteniamo dalla prima disuguaglianza

$$\sum_{k=a}^{n-1} f(k) \leq \sum_{k=a}^{n-1} \int_k^{k+1} f(x)dx = \int_a^n f(x)dx,$$

mentre dalla seconda

$$\int_a^n f(x)dx = \sum_{k=a}^{n-1} \int_k^{k+1} f(x)dx \leq \sum_{k=a}^{n-1} f(k+1).$$

Aggiungendo ora $f(a)$ e $f(n)$ alle due somme precedenti si ottiene l'enunciato. ∎



È di particolare utilità la seguente semplice conseguenza:

Corollario 2.9 *Assumendo le stesse ipotesi della proposizione precedente, se* $f(n) = o(\int_a^n f(x)dx)$, *allora*

$$\sum_{k=a}^{n} f(k) \sim \int_a^n f(x)dx.$$

Esempio 2.5
Applicando il metodo appena illustrato è facile verificare che, per ogni numero reale $p > 0$,

$$\sum_{k=1}^{n} k^p \sim \int_0^n x^p dx = \frac{n^{p+1}}{p+1}.$$

∎

In maniera del tutto analoga si dimostra un risultato equivalente per le funzioni monotone non crescenti.

Proposizione 2.10 *Sia* $f : \mathbb{R}^+ \to \mathbb{R}^+$ *una funzione monotona non crescente. Allora, per ogni* $a \in \mathbb{N}$ *e ogni intero* $n \geq a$, *abbiamo*

$$\int_a^n f(x)dx + f(n) \leq \sum_{k=a}^{n} f(k) \leq f(a) + \int_a^n f(x)dx$$

Esempio 2.6
Consideriamo la sommatoria $H_n = \sum_{k=1}^{n} \frac{1}{k}$ e applichiamo la proposizione precedente; si ricava

$$\log_e n + \frac{1}{n} \leq H_n \leq \log_e n + 1$$

e quindi $H_n \sim \log_e n$.

I valori H_n, per $n > 0$, sono chiamati "numeri armonici" e la loro valutazione compare nell'analisi di classici algoritmi. Ricordiamo che usando metodi più complicati si può ottenere la seguente espressione

$$H_n = \log_e n + \gamma + \frac{1}{2n} + o(\frac{1}{n})$$

dove $\gamma = 0,57721..$ è una costante nota chiamata "costante di Eulero". ∎

Concludiamo osservando che la tecnica appena presentata non permette in generale di ottenere approssimazioni asintotiche per funzioni a crescita esponenziale. Per esempio, consideriamo la sommatoria valutata nell'Esempio 2.3. La crescita della funzione $x2^x$ è esponenziale e il metodo di approssimazione mediante integrali non consente di ottenere l'espressione asintotica della somma. Infatti, integrando per parti si verifica facilmente che

$$\int_0^n x2^x dx = \frac{2^n(n\log 2 - 1) + 1}{\log^2 2} = \Theta(n2^n),$$

quindi applicando la proposizione precedente riusciamo solo a determinare l'ordine di grandezza dell'espressione considerata

$$\sum_{k=0}^{n} k2^k = \Theta(n2^n).$$

Esercizio

Determinare l'espressione asintotica delle seguenti sommatorie al crescere di n a $+\infty$:

$$\sum_{k=0}^{n} k^{3/2}, \quad \sum_{k=1}^{n} \log_2 k, \quad \sum_{k=1}^{n} k\log_2 k.$$

Capitolo 3

Modelli di calcolo

Obiettivo di questo corso è lo studio di algoritmi eseguibili su macchine: il significato di un algoritmo (detto anche semantica operazionale) e la valutazione del suo costo computazionale non possono prescindere da una descrizione (implicita o esplicita) del modello su cui l'algoritmo viene eseguito.

Il modello RAM che presentiamo in questo capitolo è uno strumento classico, ampiamente discusso in vari testi (vedi [1, 12, 15]) e generalmente accettato (spesso sottointeso) quale modello di base per l'analisi delle procedure sequenziali. L'analisi degli algoritmi che presenteremo nei capitoli successivi sarà sempre riferita a questo modello a meno di esplicito avvertimento.

Il modello qui presentato è caratterizzato da una memoria ad accesso casuale formata da celle che possono contenere un intero qualsiasi; le istruzioni sono quelle di un elementare linguaggio macchina che consente di eseguire istruzioni di input e output, svolgere operazioni aritmetiche, accedere e modificare il contenuto della memoria, eseguire semplici comandi di salto condizionato.

La richiesta che ogni registro possa contenere un intero arbitrario è ovviamente irrealistica. Per quanto riguarda l'analisi di complessità è però possibile ovviare a tale inconveniente introducendo un criterio di costo logaritmico nel quale il tempo e lo spazio richiesti dalle varie istruzioni dipendono dalle dimensioni degli operandi coinvolti.

La semplicità e trasparenza del modello consentono di comprendere rapidamente come procedure scritte mediante linguaggi ad alto livello possono essere implementati ed eseguiti su macchina RAM. Questo permette di valutare direttamente il tempo e lo spazio richiesti dall'esecuzione di procedure scritte ad alto livello senza farne una esplicita traduzione in linguaggio RAM.

Fra i limiti del modello segnaliamo che non è presente una gerarchia di memoria (memoria tampone, memoria di massa) e le istruzioni sono eseguite una alla volta da un unico processore. Questo modello si presta quindi all'analisi solo di algoritmi sequenziali processati in memoria centrale.

3.1 Macchina ad accesso casuale (RAM)

Il modello di calcolo che descriviamo in questa sezione si chiama "Macchina ad accesso casuale" (detto anche RAM, acronimo di Random Access Machine) ed è costituito da un nastro di ingresso, un nastro di uscita, un programma rappresentato da una sequenza finita di istruzioni, un contatore *lc* che indica l'istruzione corrente da eseguire, e una memoria formata da infiniti registri $R_0, R_1, \ldots, R_k, \ldots$. In questo modello si assumono inoltre le seguenti ipotesi:

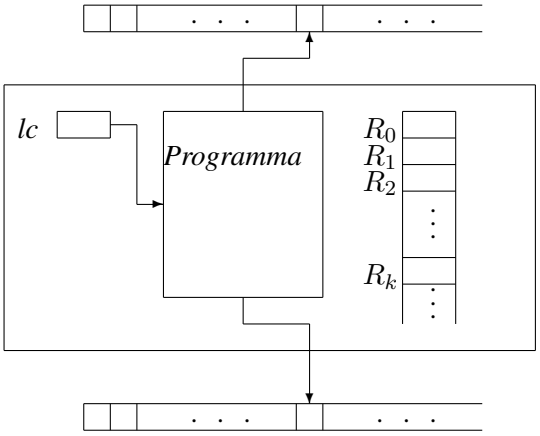
1. Ciascuno dei due nastri è rappresentato da infinite celle, numerate a partire dalla prima, ognuna delle quali può contenere un numero intero. Il nastro di ingresso è dotato di una testina di sola

lettura mentre quello di uscita dispone di una testina di sola scrittura. Le due testine si muovono sempre verso destra e all'inizio del calcolo sono posizionate sulla prima cella. Inizialmente tutte le celle del nastro di uscita sono vuote mentre il nastro di ingresso contiene l'input della macchina; questo è formato da un vettore di n interi $x_1, x_2, \ldots, x_n$, disposti ordinatamente nelle prime n celle del nastro.

2. Il programma è fissato e non può essere modificato durante l'esecuzione. Ciascuna istruzione è etichettata e il registro lc (location counter) contiene l'etichetta dell'istruzione da eseguire. Le istruzioni sono molto semplici e ricordano quelle di un linguaggio assembler: si possono eseguire operazioni di lettura e scrittura sui due nastri, caricamento dei dati in memoria e salto condizionato, oltre alle tradizionali operazioni aritmetiche sugli interi.

3. Ogni registro R_k, $k \in \mathbb{N}$, può contenere un arbitrario intero relativo (il modello è realistico solo quando gli interi usati nel calcolo hanno dimensione inferiore a quella della parola). L'indirizzo del registro R_k è l'intero k. Il registro R_0 è chiamato accumulatore ed è l'unico sul quale si possono svolgere operazioni aritmetiche.

Il modello è rappresentato graficamente dalla seguente figura:



3.1.1 Linguaggio di programmazione della macchina RAM

Il *programma* di una macchina RAM è una sequenza finita di istruzioni

$$P \;=\; ist_1; ist_2; \ldots; ist_m$$

ciascuna delle quali è una coppia formata da un *codice di operazione* e da un *indirizzo*. Un indirizzo a sua volta può essere un *operando* oppure una *etichetta*. Nella tabella seguente elenchiamo i 13 codici di operazione previsti nel nostro modello e specifichiamo per ciascuno di questi il tipo di indirizzo corrispondente.

Codice di operazione	*Indirizzo*
LOAD	operando
STORE	
ADD	
SUB	
MULT	
DIV	
READ	
WRITE	
JUMP	etichetta
JGTZ	
JZERO	
JBLANK	
HALT	

Come definiremo meglio in seguito, le prime due istruzioni LOAD e STORE servono per spostare i dati fra i registri della memoria; le istruzioni ADD, SUB, MULT e DIV eseguono invece operazioni aritmetiche; vi sono poi due istruzioni di lettura e scrittura (READ e WRITE) e quattro di salto condizionato (JUMP, JGTZ, JZERO e JBLANK); infine l'istruzione HALT, che non possiede indirizzo, serve per arrestare la computazione.

Le etichette sono associate solo a comandi di salto e servono per indicare le istruzioni del programma cui passare eventualmente il controllo; quindi ogni istruzione può anche essere dotata di etichetta iniziale (solitamente un numero intero).

Un operando invece può assumere tre forme diverse:

$=i$	indica l'intero $i \in \mathbb{Z}$,
i	indica il contenuto del registro R_i e in questo caso $i \in \mathbb{N}$,
$*i$	indica il contenuto del registro R_j dove j é il contenuto del registro R_i (e qui entrambi i, j appartengono a $\mathbb{N}$).

Osserva che $*i$ rappresenta l'usuale modalità di indirizzamento indiretto.

Il valore di un operando dipende dal contenuto dei registri. Chiamiamo quindi *stato* della macchina la legge che associa ad ogni registro il proprio contenuto e alle testine di lettura/scrittura le loro posizioni sul nastro. Formalmente uno stato è una funzione

$$S : \{r, w, lc, 0, 1, \ldots, k, \ldots\} \to \mathbb{Z},$$

che interpretiamo nel modo seguente:

$S(r)$ indica (eventualmente) la posizione della testina sul nastro di ingresso, nel senso che se $S(r) = j$ e $j > 0$ allora la testina legge la j-esima cella del nastro;

$S(w)$ indica in modo analogo la posizione della testina sul nastro di uscita;

$S(lc)$ è il contenuto del registro lc;

$S(k)$ è il contenuto del registro R_k per ogni $k \in \mathbb{N}$.

Uno stato particolare è lo stato *iniziale* S_0, nel quale $S_0(r) = S_0(w) = S_0(lc) = 1$ e $S_0(k) = 0$ per ogni $k \in \mathbb{N}$. Nello stato iniziale quindi tutti i registri della memoria sono azzerati, le testine di lettura e scrittura sono posizionate sulla prima cella e il contatore lc indica la prima istruzione del programma.

Il valore di un operando op in uno stato S, denotato da $V_S(op)$, è così definito:

$$V_S(op) = \begin{cases} i & \text{if } op \text{ è } = i, \text{ dove } i \in \mathbb{Z} \\ S(i) & \text{if } op \text{ è } i, \text{ dove } i \in \mathbb{N} \\ S(S(i)) & \text{if } op \text{ è } *i, \text{ dove } i \in \mathbb{N} \text{ e } S(i) \geq 0 \\ \perp & \text{altrimenti} \end{cases}$$

Possiamo allora descrivere l'esecuzione di un programma P su un input $x_1, x_2, \ldots, x_n$, dove $x_i \in \mathbb{Z}$ per ogni i, nel modo seguente:

1. Configura la macchina nello stato iniziale e inserisci i dati di ingresso nel nastro di lettura, collocando ciascun intero x_i nella i-esima cella, per ogni $i = 1, 2, \ldots, n$, e inserendo nella $n+1$-esima un simbolo speciale $\flat$ che chiamiamo blank.

2. *Finché* il contatore lc non indica l'istruzione `HALT` *esegui*

   (a) Individua l'istruzione da eseguire mediante il contenuto di lc.

   (b) Esegui l'istruzione cambiando lo stato secondo le regole elencate nella tabella seguente.

Le regole di cambiamento di stato sono elencate con ovvio significato nella seguente tabella nella quale S indica lo stato corrente, e $:=$ denota l'assegnamento di nuovi valori alla funzione S. Si suppone inoltre che il contatore lc venga incrementato di 1 nell'esecuzione di tutte le istruzioni salvo quelle di salto `JUMP, JGTZ, JZERO, JBLANK`.

Istruzione		*Significato*
`LOAD`	a	$S(0) := V_S(a)$
`STORE`	i	$S(i) := S(0)$
`STORE`	$*i$	$S(S(i)) := S(0)$
`ADD`	a	$S(0) := S(0) + V_S(a)$
`SUB`	a	$S(0) := S(0) - V_S(a)$
`MULT`	a	$S(0) := S(0) * V_S(a)$
`DIV`	a	$S(0) := S(0) \div V_S(a)$
`READ`	i	$S(i) := x_{S(r)}$ e $S(r) := S(r) + 1$
`READ`	$*i$	$S(S(i)) := x_{S(r)}$ e $S(r) := S(r) + 1$
`WRITE`	a	Stampa $V_S(a)$ nella cella $S(w)$ del nastro di scrittura e poni $S(w) := S(w) + 1$
`JUMP`	b	$S(lc) := b$
`JGTZ`	b	se $S(0) > 0$ allora $S(lc) := b$ altrimenti $S(lc) := S(lc) + 1$
`JZERO`	b	se $S(0) = 0$ allora $S(lc) := b$ altrimenti $S(lc) := S(lc) + 1$
`JBLANK`	b	se la cella $S(r)$ contiene $\flat$ allora $S(lc) := b$ altrimenti $S(lc) := S(lc) + 1$
`HALT`		arresta la computazione

Per semplicità supponiamo che una istruzione non venga eseguita se i parametri sono mal definiti (ad esempio quando $S(lc) \leq 0$, oppure $V_S(a) = \perp$). In questo caso la macchina si arresta nello stato corrente.

Possiamo così considerare la computazione di un programma P su un dato input come una sequenza (finita o infinita) di stati

$$S_0, S_1, \ldots, S_i, \ldots$$

nella quale S_0 è lo stato iniziale e, per ogni i, S_{i+1} si ottiene eseguendo nello stato S_i l'istruzione di indice $S_i(lc)$ del programma P (ammettendo l'ingresso dato). Se la sequenza è finita e S_m è l'ultimo suo elemento, allora $S_m(lc)$ indica l'istruzione `HALT` oppure un'istruzione che non può essere eseguita. Se invece la sequenza è infinita diciamo che il programma P sull'input dato non si ferma, o anche che la computazione non si arresta.

A questo punto possiamo definire la semantica del linguaggio RAM associando ad ogni programma P la funzione parziale F_P *calcolata* da P. Formalmente tale funzione è della forma

$$F_P : \bigcup_{n=0}^{+\infty} \mathbb{Z}^n \to \bigcup_{n=0}^{+\infty} \mathbb{Z}^n \cup \{\perp\}$$

dove denotiamo con $\mathbb{Z}^n$, $n > 0$, l'insieme dei vettori di interi a n componenti, con $\mathbb{Z}^0$ l'insieme contenente il vettore vuoto e con $\perp$ il simbolo di indefinito. Per ogni $n \in \mathbb{N}$ e ogni $\underline{x} \in \mathbb{Z}^n$, se il programma P su input $\underline{x}$ si arresta, allora $F_P(\underline{x})$ è il vettore di interi che si trova stampato sul nastro di uscita al temine della computazione; viceversa, se la computazione non si arresta, allora $F_P(\underline{x}) = \perp$.

Esempio 3.1
Il seguente programma RAM riceve in input n interi, $n \in \mathbb{N}$ qualsiasi, e calcola il massimo tra questi valori; la procedura confronta ciascun intero con il contenuto del registro R_2 nel quale viene mantenuto il massimo dei valori precedenti.

```
          READ   1
2         JBLANK 10
          LOAD   1
          READ   2
          SUB    2
          JGTZ   2
          LOAD   2
          STORE  1
          JUMP   2
10        WRITE  1
          HALT
```

∎

Esercizi

1) Definire un programma RAM per il calcolo della somma di n interi.
2) Definire un programma RAM per memorizzare una sequenza di n interi nei registri $R_1, R_2, \ldots, R_n$, assumendo $n \geq 1$ variabile.

3.1.2 Complessità computazionale di programmi RAM

In questa sezione vogliamo definire la quantità di tempo e di spazio consumate dall'esecuzione di un programma RAM su un dato input. Vi sono essenzialmente due criteri usati per determinare tali quantità. Il primo è il *criterio di costo uniforme* secondo il quale l'esecuzione di ogni istruzione del programma richiede una unità di tempo indipendentemente dalla grandezza degli operandi. Analogamente, lo spazio richiesto per l'utilizzo di un registro della memoria è di una unità, indipendentemente dalla dimensione dell'intero contenuto.

Definizione 3.1 *Un programma RAM P su input $\underline{x}$ richiede tempo di calcolo t e spazio di memoria s, secondo il criterio uniforme, se la computazione di P su $\underline{x}$ esegue t istruzioni e utilizza s registri della macchina RAM, con la convenzione che $t = +\infty$ se la computazione non termina e $s = +\infty$ se si utilizza un numero illimitato di registri.*

Nel seguito denotiamo con $T_P(\underline{x})$ e con $S_P(\underline{x})$ rispettivamente il tempo di calcolo e lo spazio di memoria richiesti dal programma P su input $\underline{x}$ secondo il criterio di costo uniforme.

Esempio 3.2
Consideriamo il programma P per il calcolo del massimo tra n interi, definito nell'esempio 3.1. È facile verificare che, per ogni input $\underline{x}$ di dimensione non nulla, $S_P(\underline{x}) = 3$. Se invece $\underline{x}$ forma una sequenza strettamente decrescente di n interi, allora $T_P(\underline{x}) = 5(n-1)+4$. ∎

Osserviamo che se un programma RAM P non utilizza l'indirizzamento indiretto (cioè non contiene istruzioni con operandi della forma $*k$) allora, per ogni input $\underline{x}$, $S_P(\underline{x})$ è minore o uguale a una costante prefissata, dipendente solo dal programma.

Poiché i registri nel nostro modello possono contenere interi arbitrariamente grandi, la precedente misura spesso risulta poco significativa rispetto a modelli di calcolo reali. È evidente che se le dimensioni degli interi contenuti nei registri diventano molto grandi rispetto alle dimensioni dell'ingresso, risulta arbitrario considerare costante il costo di ciascuna istruzione. Per questo motivo il criterio di costo uniforme è considerato un metodo di valutazione realistico solo per quegli algoritmi che non incrementano troppo la dimensione degli interi calcolati. Questo vale ad esempio per gli algoritmi di ordinamento e per quelli di ricerca.

Una misura più realistica di valutazione del tempo e dello spazio consumati da un programma RAM può essere ottenuta attribuendo ad ogni istruzione un costo di esecuzione che dipende dalla dimensione dell'operando. Considereremo qui il *criterio di costo logaritmico*, così chiamato perchè il tempo di calcolo richiesto da ogni istruzione dipende dal numero di bit necessari per rappresentare gli operandi.

Per ogni intero $k > 0$, denotiamo con $l(k)$ la lunghezza della sua rappresentazione binaria, ovvero $l(k) = \lfloor \log_2 k \rfloor + 1$. Estendiamo inoltre questa definizione a tutti gli interi, ponendo $l(0) = 1$ e $l(k) = \lfloor \log_2 |k| \rfloor + 1$ per ogni $k < 0$. Chiameremo il valore $l(k)$ "lunghezza" dell'intero k; questa funzione è una buona approssimazione intera del logaritmo in base 2: per n abbastanza grande, $l(k) \approx \log_2 k$.

Definiamo allora mediante la seguente tabella il costo logaritmico di un operando a quando la macchina si trova in uno stato S e lo denotiamo con $t_S(a)$.

Operando a	*Costo* $t_S(a)$
$= k$	$l(k)$
k	$l(k) + l(S(k))$
$*k$	$l(k) + l(S(k)) + l(S(S(k)))$

La seguente tabella definisce invece il costo logaritmico delle varie istruzioni RAM, quando la macchina si trova nello stato S. Nota che il costo di ogni operazione è dato dalla somma delle lunghezze degli interi necessari per eseguire l'istruzione.

Istruzione		*Costo*
LOAD	a	$t_S(a)$
STORE	k	$l(S(0)) + l(k)$
STORE	$*k$	$l(S(0)) + l(k) + l(S(k))$
ADD	a	$l(S(0)) + t_S(a)$
SUB	a	$l(S(0)) + t_S(a)$
MULT	a	$l(S(0)) + t_S(a)$
DIV	a	$l(S(0)) + t_S(a)$
READ	k	$l(x_{S(r)}) + l(k)$
READ	$*k$	$l(x_{S(r)}) + l(k) + l(S(k))$
WRITE	a	$t_S(a)$
JUMP	b	1
JGTZ	b	$l(S(0))$
JZERO	b	$l(S(0))$
JBLANK	b	1
HALT		1

Per esempio, il tempo di esecuzione (con costo logaritmico) di `STORE` $*k$ è dato dalla lunghezza dei tre interi coinvolti nell'istruzione: il contenuto dell'accumulatore ($S(0)$), l'indirizzo del registro (k) e il suo contenuto ($S(k)$).

Definizione 3.2 *Il tempo di calcolo $T^l_P(\underline{x})$ richiesto dal programma P su ingresso $\underline{x}$ secondo il criterio di costo logaritmico è la somma dei costi logaritmici delle istruzioni eseguite nella computazione di P su input $\underline{x}$.*

È evidente che, per ogni programma P, $T_P(\underline{x}) \leq T^l_P(\underline{x})$, per ogni input $\underline{x}$. Per certi programmi tuttavia i valori $T_P(\underline{x})$ e $T^l_P(\underline{x})$ possono differire drasticamente portando a valutazioni diverse sull'efficienza di un algoritmo.

Esempio 3.3
Consideriamo per esempio la seguente procedura Algol-like che calcola la funzione $z = 3^{2^n}$, su input $n \in \mathbb{N}$, per quadrati successivi:

```
read x
y := 3
while x > 0 do
        { y := y * y
        { x := x - 1
write y
```

La correttezza è provata osservando che dopo la k-esima esecuzione del ciclo `while` la variabile y assume il valore 3^{2^k}. Il programma RAM corrispondente, che denotiamo con Ψ, è definito dalla seguente procedura:

```
                READ    1
                LOAD    =3
                STORE   2
                LOAD    1
while           JZERO   endwhile
                LOAD    2
                MULT    2
                STORE   2
```

```
                LOAD    1
                SUB     = 1
                STORE   1
                JUMP    while
endwhile        WRITE   2
                HALT
```

Si verifica immediatamente che il ciclo `while` viene percorso n volte, e che quindi $T_\Psi(n) = 8n + 7$.

Poiché dopo la k-esima iterazione del ciclo `while` R_2 contiene l'intero 3^{2^k}, il costo logaritmico di `LOAD 2, MULT 2, STORE 2` sarà dell'ordine di $l(3^{2^k}) \sim 2^k \cdot \log_2 3$. Di conseguenza:

$$T^l_\Psi(n) = \Theta\left(\sum_{k=0}^{n-1} 2^k\right) = \Theta(2^n)$$

Quindi, mentre $T_\Psi(n) = \Theta(n)$, il valore di $T^l_\Psi(n)$ è una funzione esponenziale in n. In questo caso la misura T_Ψ risulta (per macchine sequenziali) assolutamente irrealistica. ∎

In modo analogo possiamo definire, secondo il criterio logaritmico, la quantità di spazio di memoria consumata da un certo programma su un dato input. Infatti, consideriamo la computazione di un programma P su un input $\underline{x}$; questa può essere vista come una sequenza di stati, quelli raggiunti dalla macchina dopo l'esecuzione di ogni istruzione a partire dallo stato iniziale. Lo spazio occupato in un certo stato della computazione è la somma delle lunghezze degli interi contenuti nei registri utilizzati dal programma in quell'istante. Lo spazio complessivo richiesto, secondo il criterio logaritmico, è quindi il massimo di questi valori al variare degli stati raggiunti dalla macchina durante la computazione. Denoteremo questa quantità con $S^l_P(\underline{x})$.

Esercizi

1) Sia P il programma definito nell'esempio 3.1. Qual è nel caso peggiore il valore di $T_P(\underline{x})$ tra tutti i vettori $\underline{x}$ di n interi?

2) Supponiamo che il programma definito nell'esempio 3.1 riceva in ingresso un vettore di n interi compresi tra 1 e k. Determinare l'ordine di grandezza del suo tempo di calcolo secondo il criterio logaritmico al crescere di n e k.

3) Scrivere un programma RAM per il calcolo della somma di n interi. Assumendo il criterio di costo uniforme, determinare l'ordine di grandezza, al crescere di n, del tempo di calcolo e dello spazio di memoria richiesti.

4) Eseguire l'esercizio precedente assumendo il criterio di costo logaritmico e supponendo che gli n interi di ingresso abbiano lunghezza n.

3.2 La macchina RASP

Le istruzioni di un programma RAM non sono memorizzate nei registri della macchina e di conseguenza non possono essere modificate nel corso dell'esecuzione. In questa sezione presentiamo invece il modello di calcolo RASP (Random Access Stored Program) che mantiene il programma in una parte della memoria e consente quindi di cambiare le istruzioni durante l'esecuzione.

L'insieme di istruzioni di una macchina RASP è identico a quello della macchina RAM, con l'unica eccezione che non è permesso l'indirizzamento indiretto (che denotavamo mediante $*k$).

Il programma di una macchina RASP viene caricato in memoria assegnando ad ogni istruzione due registri consecutivi: il primo contiene un intero che codifica il codice di operazione dell'istruzione; il

secondo invece conserva l'indirizzo. Inoltre il contenuto del primo registro specifica anche il tipo di indirizzo mantenuto nel secondo, cioè rivela se l'indirizzo successivo è della forma $= k$ oppure k; in questo modo il secondo registro conserva solo il valore k. Per eseguire l'istruzione, il contatore di locazione dovrà puntare al primo dei due registri.

Una possibile codifica delle istruzioni è data dalla seguente tabella:

Istruzione	*Codifica*	*Istruzione*	*Codifica*	*Istruzione*	*Codifica*
`LOAD ( )`	1	`SUB =( )`	7	`WRITE ( )`	13
`LOAD =( )`	2	`MULT ( )`	8	`WRITE =( )`	14
`STORE ( )`	3	`MULT =( )`	9	`JUMP ( )`	15
`ADD ( )`	4	`DIV ( )`	10	`JGTZ ( )`	16
`ADD =( )`	5	`DIV =( )`	11	`JZERO ( )`	17
`SUB ( )`	6	`READ ( )`	12	`JBLANK ( )`	18
				`HALT`	19

Esempio 3.4

Per "`STORE` 17" il primo registro contiene 3, il secondo 17.
Per "`ADD` $= 8$" il primo registro contiene 5, il secondo 8.
Per "`ADD` 8" il primo registro contiene 4, il secondo 8. ∎

I concetti di stato, computazione, funzione calcolata da un programma, tempo e spazio (uniforme o logaritmico) si definiscono come per le macchine RAM, con qualche piccola variazione: per esempio, salvo che per le istruzioni di salto (`JUMP, JGTZ, JZERO, JBLANK`), il registro lc viene incrementato di 2, tenendo conto che ogni istruzione occupa due registri consecutivi.

Rimarchiamo qui che nello stato iniziale i registri non sono posti tutti a 0 come avveniva nel modello RAM, dovendo il programma essere memorizzato; sottolineiamo, inoltre, che il programma può automodificasi nel corso della propria esecuzione.

Come per le macchine RAM, dato un programma RASP Ψ e un input I, denoteremo con $F_\Psi(I)$ la funzione calcolata da Ψ e con $T_\Psi(I)$, $T^l_\Psi(I)$, $S_\Psi(I)$, $S^l_\Psi(I)$ rispettivamente il tempo uniforme, il tempo logaritmico, lo spazio uniforme, lo spazio logaritmico consumato dal programma Ψ sull'ingresso assegnato.

Affrontiamo ora il problema della simulazione di macchine RAM con macchine RASP e viceversa. Un primo risultato è il seguente:

Teorema 3.1 *Per ogni programma RAM Φ, esiste un programma RASP Ψ che calcola la stessa funzione (cioè , $F_\Phi = F_\Psi$) e tale che $T_\Psi(I) \leq 6 \cdot T_\Phi(I)$.*

Dimostrazione. Detto $|\Phi|$ il numero di istruzioni del programma RAM Φ, il programma RASP Ψ che costruiremo sarà contenuto nei registri compresi tra R_2 e R_r, dove $r = 12 \cdot |\Phi| + 1$; il registro R_1 sarà usato dalla RASP come accumulatore temporaneo e nella simulazione il contenuto di indirizzo k nella macchina RAM ($k \geq 1$) sarà memorizzato nell'indirizzo $r + k$ sulla macchina RASP. Il programma Ψ sarà ottenuto dal programma Φ sostituendo ogni istruzione RAM in Φ con una sequenza di istruzioni RASP che eseguono lo stesso calcolo.

Ogni istruzione RAM che non richiede indirizzamento indiretto è sostituita facilmente dalle corrispondenti istruzioni RASP (con gli indirizzi opportunamente incrementati).

Mostriamo ora che ogni istruzione RAM che richiede indirizzamento indiretto può essere sostituita da 6 istruzioni RASP; così, il tempo di calcolo di Ψ sarà al più 6 volte quello richiesto da Φ, ed il programma Ψ occuperà al più $12 \cdot |\Phi|$ registri, giustificando la scelta di r. Dimostriamo la proprietà solo

per `MULT` $*k$ poiché il ragionamento si applica facilmente alle altre istruzioni. La simulazione di MULT $*k$ è data dalla seguente sequenza di istruzioni RASP, che supponiamo vadano inserite tra i registri R_M e R_{M+11}:

Indirizzo	*Contenuto*	*Significato*	*Commento*
M	3	`STORE` 1	Memorizza il contenuto
M+1	1		dell'accumulatore nel registro R_1
M+2	1	`LOAD` $r+k$	Carica nell'accumulatore il contenuto
M+3	$r+k$		Y del registro di indirizzo $r+k$
M+4	5	`ADD` $= r$	Calcola $r+Y$ nell'accumulatore
M+5	r		
M+6	3	`STORE` $M+11$	Memorizza $r+Y$ nel registro di
M+7	M+11		indirizzo M+11
M+8	1	`LOAD` 1	Carica nell'accumulatore il
M+9	1		vecchio contenuto
M+10	8	`MULT` $r+Y$	Esegui il prodotto tra il contenuto
M+11	-		dell'accumulatore e quello del registro $r+Y$

Per quanto riguarda il criterio di costo logaritmico, con la stessa tecnica e una attenta analisi dei costi si ottiene una proprietà analoga alla precedente.

Teorema 3.2 *Per ogni programma RAM Φ esistono un programma RASP Ψ e una costante intera $C > 0$ tali che, per ogni input I,*

$$F_\Phi = F_\Psi \ \ e \ \ T^l_\Psi(I) \leq C \cdot T^l_\Phi(I)$$

L'indirizzamento indiretto rende possibile la simulazione di programmi RASP con macchine RAM. Qui presentiamo il seguente risultato senza dimostrazione:

Teorema 3.3 *Per ogni programma RASP Ψ, esiste un programma RAM Φ che calcola la stessa funzione (cioè $F_\Psi = F_\Phi$) e due costanti positive C_1, C_2 tali che*

$$T_\Phi(I) \leq C_1 \cdot T_\Psi(I) \ \ e \ \ T^l_\Phi(I) \leq C_2 \cdot T^l_\Psi(I)$$

per ogni input I.

3.3 Calcolabilità e calcolabilità effettiva

Una conseguenza dei precedenti risultati è che la classe di funzioni calcolabili con programmi RAM coincide con la classe di funzioni calcolabile con programmi RASP. Sempre con tecniche di simulazione si potrebbe mostrare che tale classe coincide con la classe di funzioni calcolabili da vari formalismi (PASCAL, C, Macchine di Turing, λ-calcolo, PROLOG, ecc.); l'indipendenza dai formalismi rende questa classe di funzioni, dette *funzioni ricorsive parziali*, estremamente robusta, così che alcuni autori

propongono di identificare il concetto (intuitivo) di "problema risolubile per via automatica" con la classe (tecnicamente ben definita) delle funzioni ricorsive parziali (**Tesi di Church-Turing**).

Una seconda conclusione è che la classe delle funzioni calcolabili in tempo (caso peggiore) $O(f(n))$ con macchine RAM coincide con la classe di funzioni calcolabili in tempo (caso peggiore) $O(f(n))$ con macchine RASP.

Questo risultato non può essere esteso tuttavia agli altri formalismi prima citati. Se però chiamiamo **P** la classe di problemi risolubili da macchine RAM con criterio logaritmico in un tempo limitato da un polinomio (ciò succede se il tempo su ingressi di dimensione n è $O(n^k)$ per un opportuno k), tale classe resta invariata passando ad altri formalismi, sempre con costo logaritmico. Questa rimarchevole proprietà di invarianza rende la classe **P** particolarmente interessante, così che alcuni autori hanno proposto di identificarla con la classe dei "problemi praticamente risolubili per via automatica" (**Tesi di Church estesa**).

3.4 Un linguaggio ad alto livello: AG

Come abbiamo visto, un qualsiasi algoritmo può essere descritto da un programma per macchine RAM e questo permette di definire il tempo e lo spazio richiesti dalla sua esecuzione. Per contro, programmi per macchine RAM sono di difficile comprensione; risulta pertanto rilevante descrivere gli algoritmi in un linguaggio che da un lato sia sufficientemente sintetico, così da renderne semplice la comprensione, dall'altro sia sufficientemente preciso così che ogni programma possa essere trasparentemente tradotto in un programma RAM. In realtà vogliamo poter scrivere programmi "comprensibili" e contemporaneamente essere in grado di valutarne la complessità, intesa come complessità del corrispondente programma RAM tradotto, senza farne una esplicita traduzione.

Diamo qui di seguito la descrizione informale di un linguaggio di tipo procedurale che chiamiamo AG. Dichiarazione di tipi saranno evitate, almeno quando i tipi risultano chiari dal contesto.

Ogni programma AG fa uso di *variabili*; una variabile è un identificatore X associato a un insieme prefissato $\mathcal{U}$ di possibili valori (che intuitivamente definiscono il "tipo" della variabile). L'insieme $\mathcal{U}$ può essere costituito ad esempio da numeri, parole o strutture dati quali vettori, pile, liste ecc. (che saranno considerate nei capitoli 4 e 9). Esso definisce l'insieme dei valori che X può assumere durante l'esecuzione di un programma. Infatti, come vedremo in seguito, il linguaggio prevede opportuni comandi di assegnamento che consentono di attribuire a una variabile un valore dato. Così, durante l'esecuzione di un programma, ciascuna variabile assume sempre un *valore corrente*.

Sulla macchina RAM la variabile X è invece rappresentata da uno o più registri il cui contenuto, in un certo stato, rappresenta il valore corrente di X. Modificare il valore di X significa quindi sulla RAM cambiare il contenuto dei corrispondenti registri.

Una *espressione* è un termine che denota l'applicazione di simboli di operazioni a variabili o a valori costanti. Per esempio, se X e Y sono variabili a valori interi, $(X+Y)*2$ è una espressione nella quale $+$ e $*$ sono simboli che denotano le usuali operazioni di somma e prodotto. Nel prossimo capitolo introdurremo le strutture dati con le relative operazioni e sarà così possibile definire le espressioni corrispondenti (ad esempio, $\texttt{PUSH}(PILA, X)$ è una espressione nella quale X e $PILA$ sono variabili, la prima a valori su un insieme $\mathcal{U}$ e la seconda sulle pile definite su $\mathcal{U}$). Durante l'esecuzione di una procedura anche le espressioni assumono un valore corrente. Il valore di una espressione in uno stato di calcolo è il risultato dell'applicazione delle operazioni corrispondenti ai valori delle variabili.

Una *condizione* è un simbolo di predicato applicato a una o più espressioni; per esempio, $X * Y > Z$, $A = B$ sono condizioni nelle quali compaiono i simboli $>$ e $=$ dall'ovvio significato. Il valore di una condizione in un certo stato è "vero" se il predicato applicato ai valori delle espressioni è vero, "falso" altrimenti. In seguito denoteremo spesso "vero" con 1 e "falso" con 0.

Descriviamo ora in modo sintetico e informale la sintassi e la semantica dei programmi AG fornendo anche la valutazione dei relativi tempi di esecuzione. Cominciamo definendo i comandi del linguaggio che sono riportati nel seguente elenco:

1. Comando di *assegnamento*, della forma "$V := E$", dove V è una variabile ed E è una espressione. Tale comando assegna alla variabile il valore dell'espressione; il tempo di calcolo richiesto per la sua esecuzione tempo è dato dalla la somma dei tempi necessari per valutare l'espressione e per assegnare il nuovo valore alla variabile.

2. Comando `if then else`, della forma "`if` P `then` C_1 `else` C_2", dove P è una condizione, C_1 e C_2 sono comandi. L'effetto è quello di eseguire C_1 se nello stato di calcolo P è vera, altrimenti quello di eseguire C_2.
   Il suo tempo di calcolo è dato dalla somma del tempo necessario per valutare P e del tempo richiesto da C_1 o da C_2 a seconda se P è vera o falsa.

3. Comando *composto*, della forma "`begin` C_1 $C_2 \ldots C_m$ `end`" dove $C_1, C_2, \ldots, C_m$ sono comandi, $m \geq 1$. L'effetto è quello di applicare i comandi $C_1, C_2, \ldots, C_m$ nell'ordine, e il tempo richiesto è la somma dei tempi di esecuzione di $C_1, C_2, \ldots, C_m$.

4. Comando `for`, della forma "`for` i=1 `to` n `do` C", dove i è una variabile intera e C è un comando. Invece n è una variabile intera il cui valore è positivo e non viene modificato da C. L'effetto è quello di eseguire C ripetutamente per n volte, una per ciascun valore $1, 2, \ldots, n$ di i. Il tempo di calcolo è la somma dei tempi richiesti dalle n esecuzioni di C.

5. Comando `while`, della forma "`while` P `do` C", dove P è una condizione e C un comando. Se la condizione P è vera, il comando C viene eseguito; questo viene ripetuto finché la condizione diventa falsa. Il tempo di calcolo è la somma dei tempi necessari a valutare la condizione e di quelli di esecuzione di C nei vari cicli. Nota che la condizione viene sempre valutata una volta in più rispetto al numero di esecuzioni di C.

6. Comando `repeat`, della forma "`repeat` C_1 $C_2 \ldots C_m$ `until` P", dove P è una condizione e $C_1, C_2, \ldots, C_m$ sono m comandi, $m \geq 1$. La sequenza di comandi $C_1, C_2, \ldots, C_m$ viene eseguita ripetutamente fino a quando la condizione P risulta vera; la condizione P viene verificata sempre al termine di ogni esecuzione del ciclo. Nota che la sequenza di comandi viene eseguita almeno una volta e che P è una condizione di uscita dal ciclo. Il tempo di calcolo è valutato come nel caso del comando `while`.

7. *Comando "con etichetta"*, della forma "`e` :C" dove `e` è una etichetta e C un comando. Permette di rappresentare univocamente un comando specifico all'interno di un programma.

8. Comando `goto`, del tipo "`goto e`", dove `e` è una etichetta. Il suo effetto è quello di rimandare all'esecuzione del comando con etichetta `e`.

Un programma nel linguaggio AG è semplicemente costituito da un comando. Inoltre è possibile dichiarare sottoprogrammi, richiamandoli da un programma principale. L'uso che se ne può fare è duplice:

a) Il sottoprogramma serve a calcolare una funzione esplicitamente utilizzata dal programma principale.

b) Il sottoprogramma serve a modificare lo stato, cioè il contenuto delle variabili, nel programma principale.

Nel primo caso il sottoprogramma è descritto nella forma `Procedura Nome` $\underline{\lambda} \cdot C$; `Nome` è un identificatore del sottoprogramma, $\underline{\lambda} = (\lambda_1, \ldots, \lambda_m)$ è una sequenza di parametri detti *parametri formali* e C è un comando che contiene *istruzioni di ritorno* del tipo "`return` E", con E espressione.

Il programma principale contiene comandi del tipo A:=`Nome`($\underline{B}$), dove A è una variabile e $\underline{B} = [B_1, \ldots, B_m]$ è una lista di variabili poste in corrispondenza biunivoca coi parametri formali; esse sono dette *parametri reali*. L'esecuzione del comando A:=`Nome`($\underline{B}$) nel programma principale richiede di inizializzare il sottoprogramma attribuendo ai parametri formali il valore dei parametri attuali (chiamata per valore) o il loro indirizzo (chiamata per indirizzo). Il controllo viene passato al sottoprogramma: quando viene eseguito un comando del tipo "`return` E", il valore di E viene attribuito ad A nel programma principale che riprende il controllo.

Consideriamo ad esempio il sottoprogramma:

```
Procedura MAX(x, y)
          if x > y  then return x
                    else return y
```

L'esecuzione di $A :=$ `MAX`$(V[I], V[J])$ in un programma principale in uno stato in cui $V[I]$ ha valore 4 e $V[J]$ ha valore 7 attribuisce ad A il valore 7.

Anche nel secondo caso il sottoprogramma è descritto nella forma `Procedura Nome` $\underline{\lambda} \cdot C$, dove `Nome` è un identificatore del sottoprogramma e $\underline{\lambda} = (\lambda_1, \ldots, \lambda_m)$ è la sequenza di cosiddetti parametri formali; in questo caso però il comando C non contiene istruzioni del tipo "`return` E".

La procedura può essere chiamata dal programma principale con un *comando di chiamata-procedura* del tipo "`Nome`($\underline{B}$)" dove $\underline{B}$ è una lista di parametri attuali in corrispondenza biunivoca coi parametri formali. Anche in questo caso la procedura chiamata viene inizializzata attribuendo ai parametri formali il valore dei parametri attuali (chiamata per valore) o il loro indirizzo (chiamata per indirizzo).

Un esempio è dato dalla seguente procedura:

```
Procedura SCAMBIA(x, y)
        begin t := x; x := y; y := t end
```

Se le chiamate sono per indirizzo, la chiamata-procedura

`SCAMBIA`$(A[k], A[s])$

nel programma principale ha l'effetto di scambiare le componenti k e s nel vettore A.

Per quanto riguarda la chiamata per valore, si osservi che eventuali modifiche del valore di un parametro formale nel corso dell'esecuzione di un sottoprogramma non si riflette in analoghe modifiche del corrispondente parametro attuale; viceversa, se la chiamata è per indirizzo, ogni modifica del parametro formale si traduce nella modifica analoga del corrispondente parametro attuale nel programma chiamante.

Il costo della chiamata di un sottoprogramma (chiamata per indirizzo) è il costo della esecuzione del comando associato al sottoprogramma.

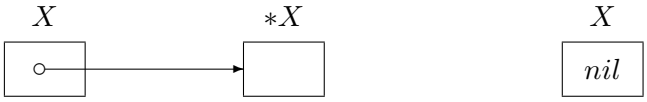
Una procedura può chiamare altre procedure, ed eventualmente se stessa. Discuteremo in seguito il costo della implementazione in RAM in questo importante caso.

Concludiamo la descrizione del linguaggio richiamando la nozione di puntatore.

Un *puntatore* è una variabile X che assume come valore corrente l'indirizzo sulla macchina RAM di un'altra variabile che nel nostro linguaggio viene denotata da $*X$. La variabile $*X$ è anche chiamata variabile *puntata* da X; durante l'esecuzione di un programma essa può non essere definita e in questo caso X assume il valore convenzionale nil (così l'insieme dei possibili valori di X è dato dalla espressione nil e dagli indirizzi di memoria della RAM).

Un puntatore è quindi un oggetto definito nella sintassi del linguaggio AG il cui significato è però strettamente legato alla macchina RAM; in particolare un puntatore può essere facilmente simulato in un programma RAM usando semplicemente l'indirizzamento indiretto. Osserva che la variabile $*X$ potrebbe rappresentare vettori, matrici o strutture dati più complesse; in questo caso, sulla macchina RAM, X sarà rappresentato da un registro che contiene l'indirizzo della prima cella che rappresenta $*X$.

Nel seguito useremo spesso l'usuale rappresentazione grafica di un puntatore descritta nella seguente figura.



Durante l'esecuzione di un programma AG opportune istruzioni di assegnamento possono modificare i valori di un puntatore e della relativa variabile puntata. Nella seguente tabella descriviamo il significato dei comandi di assegnamento che coinvolgono due puntatori X e Y; essi faranno parte a tutti gli effetti dei possibili comandi di assegnamento del linguaggio AG.

Comando	*Significato*
$*X := *Y$	Assegna alla variabile puntata da X il valore della variabile puntata da Y
$*X := Z$	Assegna il valore della variabile Z alla variabile puntata da X.
$Z := *X$	Assegna il valore della variabile puntata da X alla variabile Z.
$X := Y$	Assegna il valore di Y (cioè l'indirizzo di $*Y$) a X (dopo l'esecuzione X e Y puntano alla stessa variabile $*Y$).

Per esempio, se $*X$ e $*Y$ contengono matrici $n \times n$, il comando $*X := *Y$ trasferisce in $*X$ la matrice $n \times n$ contenuta in $*Y$; ogni ragionevole implementazione su macchina RAM richiederà per questo trasferimento $\Omega(n^2)$ passi. Viceversa, il comando $X := Y$ fa puntare la variabile X alla variabile $*Y$ (cioè alla variabile a cui punta Y) in $O(1)$ passi (secondo il criterio uniforme); naturalmente in questo caso sarà memorizzata una sola matrice nell'indirizzo comune contenuto in X e in Y.

Esercizi

1) Usando il linguaggio AG, descrivere un algoritmo per calcolare il prodotto di n interi (con $n \in \mathbf{N}$ qualsiasi), e uno per determinare il loro valore massimo e quello minimo.

2) Assumendo il criterio di costo uniforme, determinare l'ordine di grandezza del tempo di calcolo e dello spazio di memoria richiesto dagli algoritmi considerati nell'esercizio precedente.

3) Eseguire l'esercizio precedente assumendo il criterio di costo logaritmico e supponendo che gli n interi di ingresso abbiano al più n bit.

Capitolo 4

Strutture dati elementari

Per mantenere in memoria un insieme di informazioni e permettere una loro efficiente manipolazione è indispensabile organizzare i dati in maniera precisa evidenziando le relazioni e le dipendenze esistenti tra questi e definendo le funzioni che permettono la modifica delle informazioni. In questa sede non vogliamo dare una definizione generale di struttura dati (che viene rimandata a un capitolo successivo), ma più semplicemente descriverne il significato intuitivo e presentare alcuni esempi che saranno utilizzati in seguito.

Informalmente, una struttura dati è costituita da uno o più insiemi e da operazioni definite sui loro elementi. Questa nozione è quindi astratta, svincolata dalla concreta rappresentazione della struttura sul modello di calcolo RAM. Le tradizionali strutture di vettore, record, lista, già incontrate nei corsi di programmazione, possono essere così definite a un livello astratto come insiemi di elementi dotati di opportune operazioni.

L'implementazione di una data struttura descrive invece il criterio con il quale i vari elementi sono memorizzati nei registri della macchina e definisce i programmi che eseguono le operazioni. È evidente che ogni struttura dati ammette in generale più implementazioni a ciascuna delle quali corrisponde un costo in termini di spazio, per il mantenimento dei dati in memoria, e uno in termini di tempo, per l'esecuzione dei programmi associati alle operazioni.

Nella progettazione e nell'analisi di un algoritmo è tuttavia importante considerare le strutture dati svincolate dalla loro implementazione. Questo consente spesso di rappresentare una parte delle istruzioni di un algoritmo come operazioni su strutture, mettendo in evidenza il metodo adottato dalla procedura per risolvere il problema e tralasciando gli aspetti implementativi. Tutto ciò facilita notevolmente la comprensione del funzionamento dell'algoritmo e l'analisi dei suoi costi. Il passo successivo può essere quello di studiare un algoritmo a livelli diversi di astrazione a seconda del dettaglio con cui sono specificate le implementazioni delle strutture dati su cui si opera. Nel nostro ambito possiamo così individuare almeno tre livelli di astrazione: il primo è quello relativo alla macchina RAM introdotta nel capitolo precedente; il secondo è quello definito dal linguaggio AG e il terzo (descritto di fatto in questo capitolo) è quello nel quale si usano esplicitamente le strutture dati e le loro operazioni nella descrizione degli algoritmi.

4.1 Vettori e record

Sia n un intero positivo e sia $\mathcal{U}$ un insieme di valori (per esempio numeri interi, numeri reali o parole definite su un dato alfabeto). Un *vettore* di dimensione n su $\mathcal{U}$ è una n-pla $(a_1, a_2, \ldots, a_n)$ tale che

$a_i \in \mathcal{U}$ per ogni $i \in \{1, 2, \ldots, n\}$; diciamo anche a_i è la componente i-esima del vettore. L'insieme di tutti i vettori di dimensione n su $\mathcal{U}$ è quindi il prodotto cartesiano

$$\mathcal{U}^n = \underbrace{\mathcal{U} \times \mathcal{U} \times \cdots \times \mathcal{U}}_{n \text{ volte}}$$

Le operazioni definite sui vettori sono quelle di proiezione e sostituzione delle componenti, definite nel modo seguente.

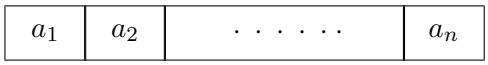
Per ogni intero i, $1 \leq i \leq n$, la proiezione i-esima è la funzione $\pi_i : \mathcal{U}^n \longrightarrow \mathcal{U}$ tale che, per ogni $A = (a_1, a_2, \ldots, a_n) \in \mathcal{U}^n$,

$$\pi_i(A) = a_i.$$

La sostituzione della componente i-esima è invece definita dalla funzione $\sigma_i : (\mathcal{U}^n \times \mathcal{U}) \longrightarrow \mathcal{U}^n$ che associa a ogni $A = (a_1, a_2, \ldots, a_n) \in \mathcal{U}^n$ e a ogni valore $u \in \mathcal{U}$ il vettore $B = (b_1, b_2, \ldots, b_n) \in \mathcal{U}^n$ tale che

$$b_j = \begin{cases} u & \text{se } j = i \\ a_j & \text{altrimenti} \end{cases}$$

Consideriamo ora l'implementazione dei vettori su macchina RAM. È evidente che se ogni elemento in $\mathcal{U}$ è rappresentabile dal contenuto di una cella di memoria, un vettore $A = (a_1, a_2, \ldots, a_n) \in \mathcal{U}^n$ è rappresentabile dal contenuto di n celle consecutive: la componente a_i è rappresentata dal contenuto della i-esima cella. La seguente figura descrive chiaramente tale implementazione.



Ovviamente sono possibili implementazioni più complesse di un vettore a seconda della rappresentazione degli elementi di $\mathcal{U}$ sulla macchina RAM. Se per esempio occorrono k celle per rappresentare un valore in $\mathcal{U}$ possiamo implementare un vettore $A \in \mathcal{U}^n$ mediante n blocchi consecutivi ciascuno dei quali composto di k registri.

Per quanto riguarda l'implementazione delle operazioni di proiezione e sostituzione osserviamo che un vettore può rappresentare il valore di una variabile e lo stesso vale per le sue componenti. In particolare, se X rappresenta una variabile su $\mathcal{U}^n$ e $i \in \{1, 2, \ldots, n\}$, possiamo denotare con $X[i]$ la variabile che rappresenta la i-esima componente di X. Questo consente di definire l'implementazione delle operazioni π_i e σ_i direttamente come assegnamento di valori alle variabili. Così nel nostro linguaggio AG potremo usare le istruzioni $X[i] := e$ o $Y := X[j]$ dove X e Y sono variabili, la prima a valori in $\mathcal{U}^n$ e la seconda in $\mathcal{U}$, mentre e è una espressione a valori in $\mathcal{U}$. Il loro significato (inteso come implementazione dell'operazione su macchina RAM) risulta ovvio.

In modo del tutto analogo possiamo definire le matrici, viste come vettori bidimensionali. Dati due interi positivi p e q, una *matrice* di dimensione $p \times q$ sull'insieme $\mathcal{U}$ è una collezione di elementi m_{ij}, dove $i \in \{1, 2, \ldots, p\}$ e $j \in \{1, 2, \ldots, q\}$, ciascuno dei quali è contenuto in $\mathcal{U}$. Tale matrice viene solitamente rappresentata nella forma $[m_{ij}]$ e gli elementi m_{ij} sono anche chiamati componenti della matrice. L'insieme delle matrici di questa forma è spesso denotato da $\mathcal{U}^{p \times q}$.

Anche in questo caso le operazioni associate sono quelle di proiezione e sostituzione, caratterizzate questa volta da coppie di indici: per ogni $M = [m_{ij}] \in \mathcal{U}^{p \times q}$ e ogni $t \in \{1, 2, \ldots, p\}$, $s \in \{1, 2, \ldots, q\}$, definiamo $\pi_{ts}(M) = m_{ts}$ e $\sigma_{ts}(M, u) = [r_{ij}]$, dove

$$r_{ij} = \begin{cases} u & \text{se } i = t \text{ e } j = s \\ m_{ij} & \text{altrimenti} \end{cases}$$

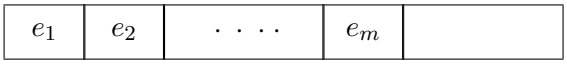
per ogni $u \in \mathcal{U}$.

Supponi che ogni elemento di $\mathcal{U}$ sia rappresentabile sulla macchina RAM mediante un solo registro. Allora, l'implementazione naturale di una matrice $M = [m_{ij}] \in \mathcal{U}^{p \times q}$ consiste nell'associare le sue componenti al contenuto di $p \cdot q$ registri consecutivi a partire da un registro fissato R_k; in questo modo la componente m_{ij} è rappresentata dal registro $R_{k+(i-1)q+j-1}$. Nota che per prelevare il valore di questa cella occorre eseguire un certo numero di operazioni aritmetiche solo per determinarne l'indirizzo. Chiaramente, se k è un intero fissato, indipendente da p e da q, il costo di accesso è $O(\log(pq))$ nel caso del criterio logaritmico. Implementazioni più complesse sono necessarie quando gli elementi di $\mathcal{U}$ richiedono più registri. Infine, le operazioni definite sulle matrici possono essere implementate in maniera del tutto simile a quelle definite sui vettori.

Una struttura del tutto analoga a quella di vettore è data dalla nozione di record. Per definire i record, consideriamo un alfabeto A (cioé un insieme finito di simboli) e per ogni $a \in A$ sia U_a un insieme di elementi. In questo modo $\{U_a \mid a \in A\}$ rappresenta una collezione finita di insiemi. Chiamiamo *record* una famiglia di elementi $\{x_a \mid a \in A\}$ tale che $x_a \in U_a$ per ogni $a \in A$. Rappresentiamo tale record nella forma $\{x_a\}_{a \in A}$. Gli insiemi U_a sono anche chiamati campi del record. La differenza rispetto ai vettori è duplice: nel record $R = \{x_a\}_{a \in A}$ gli elementi non sono ordinati, quindi R è un insieme e non una sequenza; inoltre, i campi U_a sono in generale diversi tra loro mentre nei vettori tutte le componenti appartengono allo stesso insieme di base U. Infine, per distinguere gli elementi di $R = \{x_a\}_{a \in A}$ rappresentiamo con $R \cdot a$ l'elemento x_a corrispondente al campo U_a.

Le operazioni associate ai record sono di nuovo le operazioni di proiezione e sostituzione che questa volta vengono rappresentate dai campi: per ogni record R e ogni $a \in A$ definiamo $\Pi_a(R) = R \cdot a$, mentre per $u \in U_a$ poniamo $\sigma_a(R, u) = S$, dove S è il record ottenuto da R sostituendo $R \cdot a$ con u. L'implementazione di un record è del tutto simile a quella di un vettore.

Si noti che il numero di componenti di un vettore o di un record è fissato dalla sua dimensione, mentre risulta spesso utile considerare vettori o record di dimensione variabile. A questo scopo introduciamo la nozione di tabella. Una *tabella* T di dimensione $k \in \mathbb{N}$ è un record formato da due campi: il primo contiene un intero m tale che $1 \leq m \leq k$, mentre il secondo contiene un vettore di m componenti. Essa può essere implementata mediante k celle consecutive, mantenendo nelle prime m i valori del vettore e segnalando in maniera opportuna la sua dimensione m (per esempio mediante un puntatore o memorizzando m in un dato registro). Tale implementazione, nel caso di una tabella T contenente un vettore $(e_1, e_2, \ldots, e_m)$, sarà rappresentata in seguito dalla seguente figura.



4.2 Liste

Come abbiamo visto nella sezione precedente, vettori e record sono caratterizzati da operazioni che permettono un accesso diretto alle singole componenti ma non consentono di modificare la dimensione della struttura. In questa sezione e nelle successive definiamo invece strutture nelle quali è possibile modificare le dimensioni aggiungendo o togliendo elementi, ma nelle quali l'accesso alle componenti non sempre è il risultato di una sola operazione e può richiedere, durante l'implementazione, l'esecuzione di numero di passi proporzionale alla dimensione della struttura stessa.

Una struttura dati tipica nella quale si possono agevolmente introdurre o togliere elementi è quella di lista. Dato un insieme di valori $\mathcal{U}$, chiamiamo *lista* una sequenza finita L di elementi di $\mathcal{U}$. In particolare

L può essere la lista vuota, che non contiene alcun elemento, denotata con il simbolo Λ; altrimenti L è una sequenza finita della forma

$$L = (a_1, a_2, \ldots, a_n),$$

dove $n \geq 1$ e $a_i \in \mathcal{U}$ per ogni $i = 1, 2, \ldots, n$. In questo caso diremo anche che a_i è l'i-esimo elemento di L.

Formalmente, la struttura dati è composta dall'insieme dei valori di base $\mathcal{U}$, dalla famiglia di tutte le liste su $\mathcal{U}$ e dall'insieme dei valori Booleani $\{0, 1\}$. Le operazioni associate sono invece date dalle funzioni `IS_EMPTY`, `ELEMENTO`, `INSERISCI` e `TOGLI` che definiamo nel seguito. Queste consentono di verificare se una lista è vuota, di determinare i suoi elementi oppure di modificarla introducento un nuovo oggetto o togliendone uno in una posizione qualsiasi. È importante sottolineare che le operazioni di inserimento e cancellazione dipendono dalla posizione degli elementi nella lista e non dal loro valore.

Per ogni lista L, per ogni $k \in \mathbb{N}$ e ogni $u \in \mathcal{U}$, le operazioni citate sono così definite:

$$\texttt{IS\_EMPTY}(L) = \begin{cases} 1 & \text{se } L = \Lambda, \\ 0 & \text{altrimenti;} \end{cases}$$

$$\texttt{ELEMENTO}(L, k) = \begin{cases} a_k & \text{se } L = (a_1, a_2, \ldots, a_n) \text{ e } 1 \leq k \leq n, \\ \perp & \text{altrimenti;} \end{cases}$$

$$\texttt{INSERISCI}(L, k, u) = \begin{cases} (u) & \text{se } L = \Lambda \text{ e } k = 1, \\ (a_1, \ldots, a_{k-1}, u, a_k, \ldots, a_n) & \text{se } L = (a_1, a_2, \ldots, a_n) \text{ e } 1 \leq k \leq n+1, \\ \perp & \text{altrimenti;} \end{cases}$$

$$\texttt{TOGLI}(L, k) = \begin{cases} (a_1, \ldots, a_{k-1}, a_{k+1}, \ldots, a_n) & \text{se } L = (a_1, a_2, \ldots, a_n) \text{ e } 1 \leq k \leq n, \\ \perp & \text{altrimenti;} \end{cases}$$

Mediante la composizione di queste operazioni possiamo calcolare altre funzioni che permettono di manipolare gli elementi di una lista; tra queste ricordiamo quelle che determinano o modificano il primo elemento di una lista:

$$\texttt{TESTA}(L) = \begin{cases} a_1 & \text{se } L = (a_1, a_2, \ldots, a_n), \\ \perp & \text{se } L = \Lambda; \end{cases}$$

$$\texttt{INSERISCI\_IN\_TESTA}(L, u) = \begin{cases} (u, a_1, a_2, \ldots, a_n) & \text{se } L = (a_1, a_2, \ldots, a_n), \\ (u) & \text{se } L = \Lambda; \end{cases}$$

$$\texttt{TOGLI\_IN\_TESTA}(L) = \begin{cases} (a_2, \ldots, a_n) & \text{se } L = (a_1, a_2, \ldots, a_n), \\ \perp & \text{se } L = \Lambda; \end{cases}$$

infatti è evidente che $\texttt{TESTA}(L)$=$\texttt{ELEMENTO}(L, 1)$, $\texttt{INSERISCI\_IN\_TESTA}(L, u)$=$\texttt{INSERISCI}(L, 1, u)$, $\texttt{TOGLI\_IN\_TESTA}(L)$=$\texttt{TOGLI}(L, 1)$. Usando lo stesso criterio possiamo definire l'operazione che calcola la lunghezza di una stringa e quella per sostituire il suo elemento k-esimo.

$$\texttt{LUNGHEZZA}(L) = \begin{cases} 0 & \text{se } L = \Lambda \\ 1 + \texttt{LUNGHEZZA}(\texttt{TOGLI\_IN\_TESTA}(L)) & \text{altrimenti} \end{cases}$$

$$\texttt{CAMBIA}(L, k, u) = \texttt{TOGLI}(\texttt{INSERISCI}(L, k, u), k + 1)$$

Analogamente si possono definire altre operazioni di uso frequente. Elenchiamo nel seguito alcune di queste tralasciando la definizione formale e riportando solo il significato intuitivo:

$$\texttt{APPARTIENE}(L, u) = \begin{cases} 1 & \text{se } u \text{ compare in } L, \\ 0 & \text{altrimenti} \end{cases}$$

$$\texttt{CODA}((a_1, \ldots, a_m)) = a_m$$

$$\texttt{INSERISCI\_IN\_CODA}((a_1, \ldots, a_m), x) = (a_1, \ldots, a_m, x)$$

$$\texttt{TOGLI\_IN\_CODA}((a_1, \ldots, a_m)) = (a_1, \ldots, a_{m-1})$$

$$\texttt{CONCATENA}((a_1, \ldots, a_m), (b_1, \ldots, b_s)) = (a_1, \ldots, a_m, b_1, \ldots, b_s)$$

Un calcolo eseguito spesso da algoritmi che manipolano una lista consiste nello scorrere i suoi elementi dal primo all'ultimo, compiendo certe operazioni su ciascuno di questi. Un procedimento di questo tipo può essere eseguito applicando opportunamente le operazioni definite sulle liste. Per brevità, e con un certo abuso di notazione, nel seguito indicheremo con l'istruzione

$$\texttt{for } b \in L \texttt{ do } \mathit{Op}(b)$$

la procedura che esegue l'operazione predefinita *Op* su ciascun elemento b della lista L nell'ordine in cui tali oggetti compaiono in L.

Per esempio, per determinare quante volte un elemento a compare in una lista L, possiamo eseguire la seguente procedura:

```
begin
      n:=0
      for b ∈ L do  if b = a then n:=n+1
      return n
end
```

Tale procedura può essere considerata come l'implementazione della seguente operazione definita utilizzando le operazioni fondamentali:

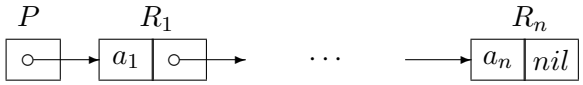
$$\texttt{NUMERO}(a, L) = \begin{cases} 0 & \text{se } L = \Lambda \\ 1 + \texttt{NUMERO}(a, \texttt{TOGLI\_IN\_TESTA}(L)) & \text{se } L \neq \Lambda \text{ e } a = \texttt{TESTA}(L)) \\ \texttt{NUMERO}(a, \texttt{TOGLI\_IN\_TESTA}(L)) & \text{se } L \neq \Lambda \text{ e } a \neq \texttt{TESTA}(L)) \end{cases}$$

4.2.1 Implementazioni

In questa sezione consideriamo tre possibili implementazioni della struttura dati appena definita. La più semplice consiste nel rappresentare una lista $L = (a_1, a_2, \ldots, a_n)$ mediante una tabella T di dimensione $m > n$ che mantiene nelle prime n componenti i valori $a_1, a_2, \ldots, a_n$ nel loro ordine. In questo modo il calcolo del k-esimo elemento di L (ovvero l'esecuzione dell'operazione $\texttt{ELEMENTO}(L, k)$) è abbastanza semplice poiché è sufficiente eseguire una proiezione sulla k-esima componente di T; possiamo assumere che nella maggior parte dei casi questo richieda un tempo $O(1)$ secondo il criterio uniforme. Viceversa l'inserimento o la cancellazione di un elemento in posizione k-esima richiede lo spostamento di tutti gli elementi successivi. Inoltre, prima di inserire un elemento, bisogna assicurarsi che la dimensione della lista non diventi superiore a quella della tabella, nel qual caso occorre incrementare opportunamente la dimensione di quest'ultima ed eventualmente riposizionarla in un'altra area di memoria. L'esecuzione di queste operazioni può quindi essere costosa in termini di tempo e spazio e questo rende poco adatta tale implementazione quando occorre eseguire di frequente l'inserimento o la cancellazione di elementi.

Una seconda implementazione, che per molte applicazioni appare più naturale della precedente, è basata sull'uso dei puntatori ed è chiamata *lista concatenata*. In questo caso una lista $L = (a_1, a_2, \ldots, a_n)$

viene rappresentata mediante n record $R_1, R_2, \ldots, R_n$, uno per ogni posizione, formati da due campi ciascuno che chiameremo el e $punt$. Ogni R_i contiene nel campo el il valore a_i e nel campo $punt$ un puntatore al record successivo. Inoltre un puntatore particolare (P) indica il primo record della lista.



Questa rappresentazione permette inoltre di implementare facilmente (in pseudocodice e quindi su macchina RAM) le operazioni definite sopra. A titolo d'esempio, riportiamo alcuni casi particolari.

```
Procedura IS_EMPTY(P)
  if P = nil then  return 1
           else  return 0
```

```
Procedura INSERISCI_IN_TESTA(P, a)
  Crea un puntatore X a una variabile di tipo record
  *X · el := a
  *X · punt := P
  P := X
```

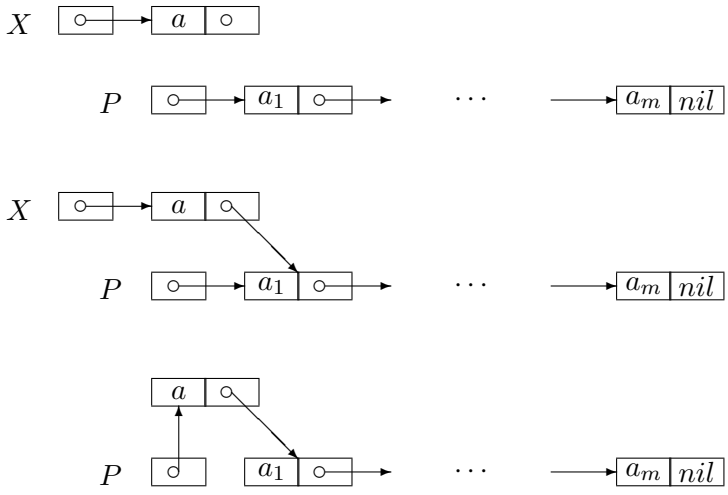


Figura 4.1: Passi esecutivi della procedura `INSERISCI_IN_TESTA`.

```
Procedura APPARTIENE(P, a)
if P = nil then   return 0
          else   begin
                   R = *P
                   while R · el ≠ a ∧ R · punt ≠ nil  do R := *(R · punt)
                   if R · el = a then return 1
                                 else return 0
                 end
```

Nota la differenza tra le implementazioni di IS_EMPTY e APPARTIENE e quella di INSERISCI_IN_TESTA: nei primi due casi la procedura restituisce un valore, nel terzo invece modifica direttamente la lista ricevuta in input. In molti casi quest'ultimo tipo di implementazione è quella effettivamente richiesta nella manipolazione di una struttura dati.

Per quanto riguarda la complessità di queste procedure, se supponiamo che l'uguaglianza tra elementi richieda tempo $O(1)$, possiamo facilmente osservare che il tempo di calcolo per IS_EMPTY e INSERISCI_IN_TESTA è $O(1)$, mentre APPARTIENE richiede nel caso peggiore $O(n)$ passi, dove n è il numero di elementi nella lista. Un aspetto molto interessante è che la complessità in spazio risulta essere $\Theta(n)$, cosa che permette una efficiente gestione dinamica delle liste.

La lista concatenata può essere "percorsa" in una sola direzione; questa implementazione si rivela inadeguata quando sia utile percorrere la lista dalla coda alla testa. Una rappresentazione della lista simmetrica rispetto al verso di percorrenza è la cosiddetta *lista bidirezionale*. Essa è ottenuta da una sequenza di records di 3 campi. In ogni record il primo campo ($prec$) contiene un puntatore al record precedente, il terzo campo ($succ$) contiene un puntatore al record successivo mentre l'elemento da memorizzare è contenuto nel secondo (el), come evidenziato nella figura 4.2.

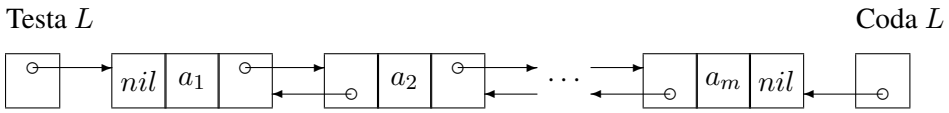


Figura 4.2: Lista bidirezionale.

Un'altra possibile implementazione di una lista può essere ottenuta mediante due tabelle dette *Nome* e *Succ*. Se I è un indice della tabella, *Nome*[I] è un elemento della lista L mentre *Succ*[I] è l'indice dell'elemento che segue *Nome*[I] in L. Un indice $I \neq 0$ identifica la lista se *Succ*[I] è definito ma *Nome*[I] non lo è , mentre conveniamo che il "fine lista" sia identificato da 0. Per esempio, la tabella seguente memorizza in posizione 3 la lista (F, O, C, A).

	Nome	*Succ*
1	C	4
2	O	1
3		5
4	A	0
5	F	2

Va osservato che l'ordine di inserimento degli elementi nella lista non necessariamente coincide con l'ordine di inserimento nella tabella.

Questa rappresentazione è flessibile e interessante, permettendo di memorizzare più liste in celle di memoria consecutive. Per esempio, la tabella successiva memorizza in posizione 3 la lista (B, A, S, S, A), in posizione 8 la lista (A, L, T, A) e in posizione 10 la lista (D, U, E).

	Nome	*Succ*
1	S	11
2	T	12
3		4
4	B	5
5	A	1
6	A	0
7	A	9

8		7
9	L	2
10		14
11	S	6
12	A	0
13	E	0
14	D	15
15	U	13

Anche in questa rappresentazione l'implementazione delle varie operazioni è lineare. Vediamo un esempio.

```
Procedura APPARTIENE(I, a)
   S:= Succ[I]
   if  S = 0 then return 0
           else begin
                    while Nome[S] ≠ a ∧ Succ[S] ≠ 0 do S:=Succ[S]
                    if  Nome[S] = a then return 1
                                    else return 0
                end
```

Le liste bidirezionali ammettono una rappresentazione analoga, a patto di aggiungere una terza tabella *Prec*, dove $Prec[I]$ è l'indice dell'elemento che precede $Nome[I]$ nella lista considerata.

4.3 Pile

Le pile possono essere interpretate come liste nelle quali le operazioni di proiezione, inserimento e cancellazione si possono applicare solo al primo elemento. Si tratta quindi di una sorta di restrizione della nozione di lista e la differenza tra le due strutture è limitata alle loro operazioni.

Formalmente, dato un insieme $\mathcal{U}$ di valori, una pila è una sequenza finita S di elementi di $\mathcal{U}$; di nuovo denotiamo con Λ la pila vuota, mentre una pila S non vuota è quindi della forma

$$S = (a_1, a_2, \ldots, a_n)$$

dove $n \geq 1$ e $a_i \in \mathcal{U}$ per ogni $i = 1, 2, \ldots, n$.

Sulle pile si possono eseguire le operazioni `IS_EMPTY, TOP, POP, PUSH` definite nel modo seguente:

$$\texttt{IS\_EMPTY}(S) = \begin{cases} 1 & \text{se } S = \Lambda, \\ 0 & \text{altrimenti} \end{cases}$$

$$\texttt{TOP}(S) = \begin{cases} a_1 & \text{se } S = (a_1, a_2, \ldots, a_n) \\ \perp & \text{se } S = \Lambda, \end{cases}$$

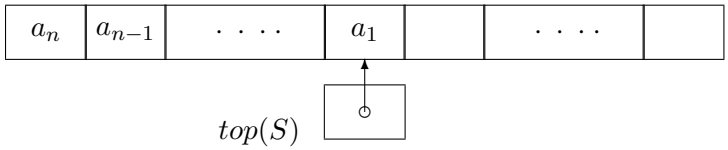
$$\texttt{POP}(S) = \begin{cases} \Lambda & \text{se } S = (a_1) \\ (a_2, a_3, \ldots, a_n) & \text{se } S = (a_1, a_2, \ldots, a_n) \text{ e } n > 1 \\ \perp & \text{se } S = \Lambda. \end{cases}$$

Inoltre, per ogni $u \in \mathcal{U}$:

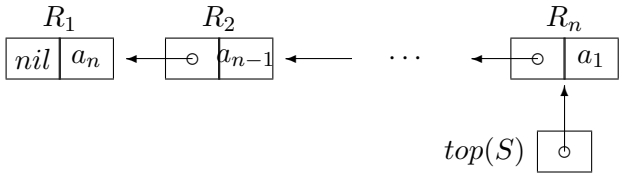
$$\texttt{PUSH}(S,u) = \begin{cases} (u, a_1, a_2, \ldots, a_n) & \text{se } S = (a_1, a_2, \ldots, a_n) \\ (u) & \text{se } S = \Lambda. \end{cases}$$

Le operazioni appena definite realizzano un criterio di mantenimento e prelievo delle informazioni chiamato LIFO (Last In First Out): il primo termine che può essere tolto dalla pila coincide con l'ultimo elemento introdotto. Nota che in questa struttura, per accedere all'elemento i-esimo, devo togliere dalla pila tutti i successivi.

L'implementazione naturale di una pila $S = (a_1, a_2, \ldots, a_n)$ consiste in una tabella di dimensione $k \geq n$ che contiene nelle prime n componenti gli elementi $a_1, a_2, \ldots, a_n$ ordinati in senso opposto e in un puntatore $top(S)$ alla componente a_1.



In questo caso, per implementare correttamente le operazioni definite sulla pila devo garantire che la dimensione n di S sia sempre minore o uguale alla dimensione k della tabella. In molte applicazioni questa condizione viene soddisfatta; altrimenti conviene implementare la pila come una lista concatenata In questo caso ogni elemento a_i di S è rappresentato da un record di due campi: il primo contiene un puntatore al record relativo alla componente a_{i+1}, mentre il secondo contiene il valore a_i. Questa rappresentazione è descritta graficamente nel modo seguente:



È facile definire i programmi che eseguono le operazioni `TOP, POP` e `PUSH` sulle implementazioni appena considerate. Nel seguito presentiamo le procedure relative alla implementazione di una pila mediante tabella che supponiamo memorizzata a partire da una cella di indirizzo k.

```
Procedura  IS_EMPTY(S)
  if  top(S) < k then return 1
              else return 0

Procedura TOP(S)
  if  top(S) ≥ k then return *top(S)

Procedura PUSH(S, x)
  top(S) := top(S) + 1
  *top(S) := x

Procedura POP(S)
  if  top(S) ≥ k then top(S) := top(S) − 1
```

4.4 Code

Una coda è una struttura che realizza un criterio di inserimento e cancellazione dei dati chiamato FIFO (First in First Out). In questo caso il primo elemento che può essere cancellato coincide con il primo elemento introdotto, cioè il più vecchio tra quelli presenti nella struttura. Anche una coda può essere interpretata come restrizione della nozione di lista.

Possiamo allora definire una *coda* Q su un insieme di valori $\mathcal{U}$ come una sequenza finita di elementi di $\mathcal{U}$. Di nuovo, Q può essere la coda vuota (Λ), oppure essa sarà rappresentata da una sequenza $(a_1, a_2, \ldots, a_n)$, dove $n \geq 1$ e $a_i \in \mathcal{U}$ per ogni i. Su una coda possiamo eseguire le operazioni `IS_EMPTY, FRONT, DEQUEUE` e `ENQUEUE` definite come segue:

$$\texttt{IS\_EMPTY}(Q) = \begin{cases} 1 & \text{se } Q = \Lambda, \\ 0 & \text{altrimenti} \end{cases}$$

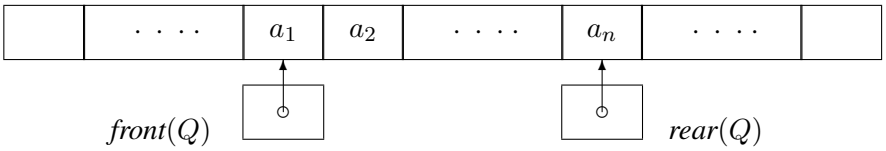
$$\texttt{FRONT}(Q) = \begin{cases} a_1 & \text{se } Q = (a_1, a_2, \ldots, a_n) \\ \perp & \text{se } Q = \Lambda, \end{cases}$$

$$\texttt{DEQUEUE}(Q) = \begin{cases} \Lambda & \text{se } Q = (a_1) \\ (a_2, \ldots, a_n) & \text{se } Q = (a_1, a_2, \ldots, a_n) \\ \perp & \text{se } Q = \Lambda. \end{cases}$$

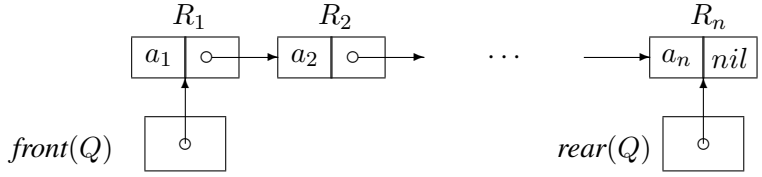
Inoltre, per ogni $b \in \mathcal{U}$,

$$\texttt{ENQUEUE}(Q, b) = \begin{cases} (a_1, a_2, \ldots, a_n, b) & \text{se } Q = (a_1, a_2, \ldots, a_n) \\ (b) & \text{se } Q = \Lambda. \end{cases}$$

Anche una coda $Q = (a_1, a_2, \ldots, a_n)$ può essere implementata usando una tabella di $k \geq n$ elementi e da due puntatori (*front* e *rear*) che indicano il primo e l'ultimo elemento della coda.



In modo analogo possiamo definire l'implementazione mediante una lista concatenata; questa rappresentazione può essere descritta rapidamente dalla figura seguente:



Esercizi

1) Dato un alfabeto finito Σ, ricordiamo che una parola su Σ è una sequenza $a_1 a_2 \cdots a_n$ tale che $a_i \in \Sigma$ per ogni i, mentre la sua inversa è la parola $a_n a_{n-1} \cdots a_1$. Usando la struttura dati appropriata e le corrispondenti operazioni, definire una procedura per ciascuno dei seguenti problemi:

- calcolare l'inversa di una parola data;
- verificare se una parola è palindroma, cioè se è uguale alla sua inversa.

2) Simulare il funzionamento di una pila mediante due code definendo anche le procedure che eseguono le corrispondenti operazioni. Quanti passi sono necessari nel caso peggiore per eseguire una operazione su una pila di n elementi?
Definire in modo analogo il funzionamento di una coda mediante due pile.

3) Usando le operazioni definite sulle liste descrivere una procedura per eliminare da una lista data tutti gli elementi che compaiono più di una volta e un'altra per cancellare tutti quelli che compaiono una volta sola. Determinare inoltre, in entrambi i casi, il numero di operazioni compiute su una lista di n elementi nel caso peggiore.

4.5 Grafi

Una delle strutture più flessibili in cui organizzare i dati per favorire la gestione delle informazioni è quella di grafo. Come le strutture precedenti, anche questa può essere definita mediante insiemi di elementi e varie operazioni di manipolazione. Preferiamo tuttavia fornire la definizione classica di grafo, qui inteso come oggetto combinatorio. Questo permette di introdurre le nozioni e i concetti principali in maniera semplice e diretta. Dalle varie implementazioni sarà quindi possibile definire facilmente le operazioni di manipolazione principali.

Dato un insieme V, denoteremo con V^s l'insieme di s-ple ordinate di elementi di V e con $V^{(s)}$ la famiglia dei sottoinsiemi di V contenenti s elementi (cioè delle s-combinazioni di V)

Per esempio, se consideriamo l'insieme $V = \{1, 2, 3\}$, allora

$$\{1,2,3\}^2 = \{(1,1),(1,2),(1,3),(2,1),(2,2),(2,3),(3,1),(3,2),(3,3)\},$$
$$\{1,2,3\}^{(2)} = \{\{1,2\},\{1,3\},\{2,3\}\}$$

Come sappiamo dalla sezione 2.2, se V contiene n elementi, V^s ne contiene n^s, mentre $V^{(s)}$ ne contiene $\binom{n}{s} = \frac{n!}{s!\cdot(n-s)!}$.

Definizione 4.1 *Un grafo orientato (o diretto) G è una coppia $G = \langle V, E\rangle$, dove V è un insieme (finito) i cui elementi sono detti vertici (o nodi) ed E è un sottoinsieme di V^2 i cui elementi sono detti archi; archi del tipo (x, x) sono detti cappi.*

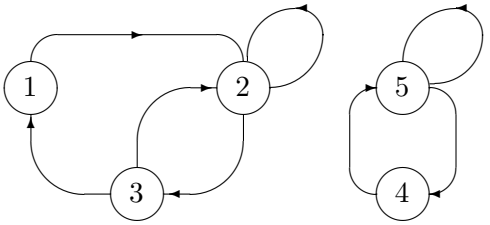
Un grafo non orientato G è una coppia $\langle V, E\rangle$, dove V è un insieme (finito) i cui elementi sono detti nodi (o vertici) ed E un sottoinsieme di $V^{(2)}$, i cui elementi sono detti lati (o archi).

La figura seguente dà una rappresentazione grafica del grafo orientato

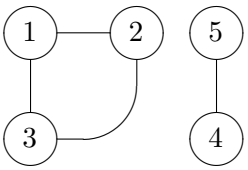
$$G_1 = \langle\{1,2,3,4,5\},\{(1,2),(2,2),(2,3),(3,2),(3,1),(4,5),(5,4),(5,5)\}\rangle$$

e del grafo non orientato

$$G_2 = \langle\{1,2,3,4,5\},\{\{1,2\},\{2,3\},\{1,3\},\{4,5\}\}\rangle.$$



G_1



G_2

Nota che un grafo non orientato non possiede cappi.

Un *sottografo* di un grafo $G = \langle V, E\rangle$ è un grafo $G' = \langle V', E'\rangle$ tale che $V' \subseteq V$ e $E' \subseteq E$.

Dato un lato (x, y) di un grafo orientato, diremo che x è la coda dell'arco e y la testa; si dirà anche che y è adiacente ad x. L'adiacenza di x in un grafo $G = \langle V, E\rangle$ è l'insieme dei vertici y tali che $(x, y) \in E$, cioè :

$$\text{Adiacenza}(x) = \{y | (x, y) \in E\}$$

Un grafo $G = \langle V, E\rangle$ può alternativamente essere rappresentato dalla famiglia delle adiacenze di tutti i suoi vertici, cioè :

$$G = \{(x, \text{Adiacenza}(x)) \, | x \in V\}$$

Per esempio, il grafo G_1 può essere rappresentato mediante le adiacenze nel modo seguente:

$$G_1 = \{(1, \{2\}), (2, \{2, 3\}), (3, \{2, 1\}), (4, \{5\}), (5, \{5, 4\})\}$$

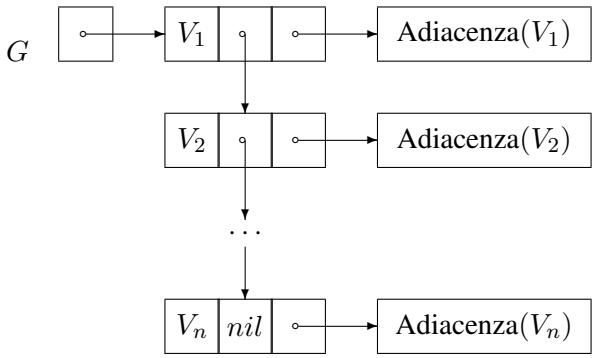
Un grafo G può essere descritto anche dalla sua matrice di adiacenza. Questa è la matrice A_G, indiciata con vertici del grafo e a componenti in $\{0, 1\}$, tale che:

$$A_G[x, y] = \begin{cases} 1 & \text{se } (x, y) \in E \\ 0 & \text{altrimenti} \end{cases}$$

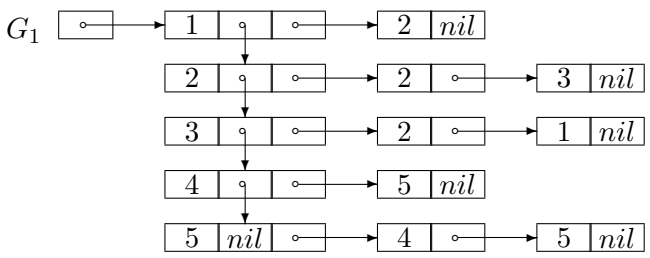
A fianco diamo la matrice di adiacenza del grafo G_1:

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

Se un grafo G ha n vertici, ogni ragionevole implementazione della matrice di adiacenza richiederà uno spazio di memoria $O(n^2)$; se G possiede un numero di lati notevolmente minore di n^2, può essere conveniente rappresentare il grafo come una lista di vertici, ognuno dei quali punta alla lista della sua adiacenza:



Qui riportiamo una possibile rappresentazione di G_1:



Se il grafo G ha n nodi ed e lati, la precedente rappresentazione occuperà una memoria $O(n+e)$.

Naturalmente le liste potranno essere descritte mediante tabelle. In questo caso possiamo considerare una coppia di tabelle di $n+e$ componenti ciascuna: le prime n componenti indicano la testa della adiacenza di ciascun nodo, mentre le altre definiscono tutte le liste. Un esempio è fornito dalla seguente figura che rappresenta le tabelle del grafo G_1.

	Testa	*Succ*
1		7
2		6
3		12
4		11
5		13
6	2	9
7	2	0
8	1	0
9	3	0
10	5	0
11	5	0
12	2	8
13	4	10

Un *cammino* da x a y in un grafo $G = \langle V, E\rangle$ è una sequenza $(v_1, v_2, \ldots, v_m)$ di vertici tale che $v_1 = x, v_m = y$ e (v_k, v_{k+1}) è un arco di G, per tutti i $k = 1, 2, \ldots, m-1$. La lunghezza del precedente cammino è $m-1$.

Diremo che un cammino è *semplice* se tutti i suoi vertici sono distinti, eccetto al più il primo e l'ultimo. Un cammino semplice in cui il primo e l'ultimo vertice coincidano è detto *ciclo*.

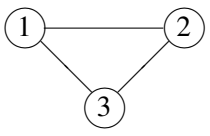
Le precedenti nozioni si estendono facilmente ai grafi non orientati; in tal caso si assume però che i cicli abbiano lunghezza almeno 3. Un grafo è detto *connesso* quando per ogni coppia x e y di vertici esiste un cammino da x a y.

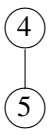
Un grafo non orientato $G = \langle V, E\rangle$ può non essere connesso; esso è però sempre esprimibile come unione di componenti connesse. Basta infatti osservare che la relazione R su V definita da

$$xRy \text{ se esiste un cammino da } x \text{ a } y \text{ oppure } x = y,$$

è una relazione di equivalenza. Le sue classi di equivalenza definiscono una partizione $\{V_1, \ldots, V_m\}$ dell'insieme V dei vertici. Per ogni $i = 1, 2, \ldots, m$ il grafo $\langle V_i, E_i\rangle$, dove $E_i = \{\{u, v\} \mid u, v \in V_i, \{u, v\} \in E\}$, è connesso ed è chiamato *componente connessa* del grafo G; vertici appartenenti a due distinte classi di equivalenza non sono invece congiungibili da alcun cammino.

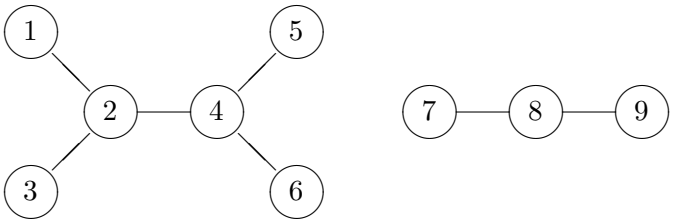
Per esempio, il grafo G_2 è formato da due componenti connesse,

la prima è 

mentre la seconda è 

4.6 Alberi

Un grafo non orientato e senza cicli è detto *foresta*; se per di più esso è connesso allora viene detto *albero*. È evidente che ogni foresta è unione di alberi, in quanto ogni componente connessa di una foresta è un albero. Il grafo sotto riportato è una foresta composta da 2 alberi:

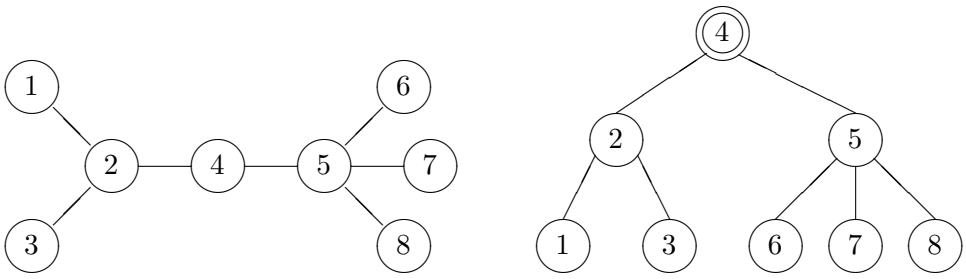


Un albero è dunque un grafo (non orientato) connesso e aciclico, cioè privo di cicli.
Osserviamo ora che, per ogni coppia di vertici x e y di un albero qualsiasi, esiste un unico cammino semplice che li collega. Infatti, poiché un albero è connesso, esiste almeno un cammino semplice che congiunge i due nodi; se ve ne fossero due distinti allora si formerebbe un ciclo (v. figura seguente).

Dato un grafo non orientato $G = \langle V, E \rangle$, chiamiamo *albero di copertura* di G un albero $T = \langle V', E' \rangle$ tale che $V' = V$ e $E' \subseteq E$. Se G non è connesso tale albero non esiste. In questo caso chiamiamo *foresta di copertura* il sottografo di G formato da una famiglia di alberi, uno per ogni componente connessa di G, ciascuno dei quali costituisce un albero di copertura della corrispondente componente connessa.

4.6.1 Alberi con radice

Un *albero con radice* è una coppia $\langle T, r \rangle$, dove T è un albero e r un suo vertice, che viene detto *radice*.



Albero Albero con radice

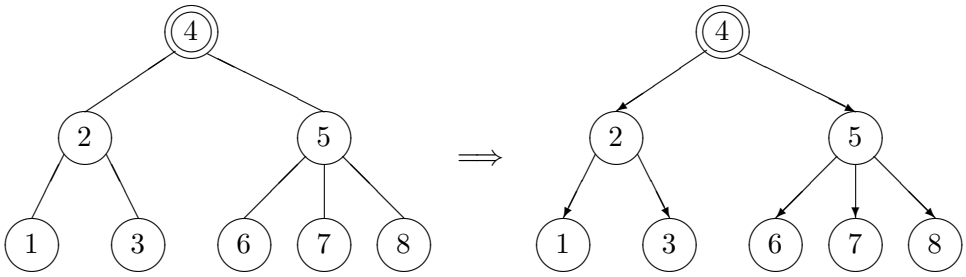
Un albero con radice è quindi un albero con cui viene evidenziato un vertice (la radice); esso viene anche detto "albero radicato". Ora, dato un vertice x in un albero con radice r, c'è un unico cammino semplice $(r, V_2, \ldots, V_m, x)$ da r a x (se $r \neq x$); il vertice V_m che precede x nel cammino è detto *padre* di x: ogni vertice di un albero radicato, escluso la radice, ha quindi un unico padre.

Un albero radicato può essere allora agevolmente rappresentato dalla tabella che realizza la funzione "padre":

Figlio	*Padre*
⋮	⋮
x	padre di x
⋮	⋮

Continuando l'analogia "famigliare", se y è padre di x diremo anche che x è *figlio* di y; diremo inoltre che z_1 e z_2 sono *fratelli* quando sono figli dello stesso padre. Continuando l'analogia "botanica", diremo che un vertice V senza figli è una *foglia*. Invece, un nodo che possiede almeno un figlio è chiamato *nodo interno*. Infine, diremo che x è *discendente* o *successore* di y se y appartiene al cammino semplice che va dalla radice a x; in tal caso diremo anche che y è un *antenato* o *predecessore* di x.

In un albero con radice, orientando ogni lato $\{x, y\}$ dal padre verso il figlio, si ottiene una struttura di grafo orientato.

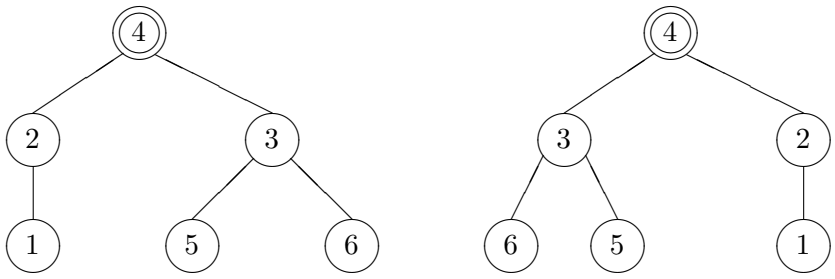


Rispetto a tale struttura, chiameremo *altezza* di un vertice x la lunghezza del più lungo cammino da x ad una foglia, e *profondità* di x la lunghezza del cammino dalla radice a x. L'altezza di un albero è l'altezza della radice.

Gli alberi con radice sono utilizzati in molte applicazioni per distribuire informazioni fra i vari nodi; gli algoritmi associati eseguono spesso processi di calcolo nei quali si percorre il cammino che va dalla radice a un dato nodo o, viceversa, che risale da un nodo fissato sino alla radice. L'altezza dell'albero diviene quindi un parametro importante nella valutazione dei tempi di calcolo di tali procedure poiché in molti casi questi ultimi risultano proporzionali (nel caso peggiore) proprio alla massima distanza dei nodi dalla radice. Per questo motivo si cerca spesso di utilizzare alberi che abbiano un'altezza piccola rispetto al numero di nodi. Si usa il termine "bilanciato" proprio per indicare intuitivamente un albero nel quale i nodi sono abbastanza vicini alla radice. In molti casi una altezza accettabile è una altezza minore o uguale al logaritmo del numero di nodi (eventualmente moltiplicato per una costante). Più formalmente possiamo dare la seguente definizione: una sequenza di alberi con radice $\{T_n\}_{n\in\mathbb{N}}$, dove ogni T_n possiede esattamente n nodi, si dice *bilanciata* se, denotando con h_n l'altezza di T_n, abbiamo $h_n = O(\log n)$.

4.6.2 Alberi ordinati

Un *albero ordinato* (detto anche *piano*) è un albero radicato in cui i figli di ogni vertice sono totalmente ordinati. I seguenti due alberi sono coincidenti come alberi radicati, ma distinti come alberi ordinati:



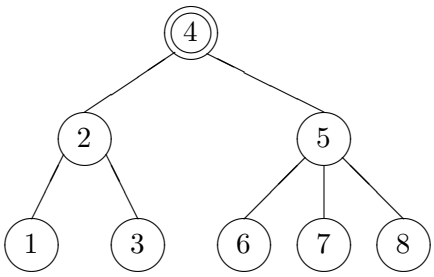
Una classica rappresentazione in memoria di un albero ordinato consiste nell'uso di tre vettori: Se l'albero è formato da n nodi, rappresentati dai primi n interi, i tre vettori P, F, S hanno dimensione n e sono definiti nel modo seguente:

$$P[i] = \begin{cases} j & \text{se } j \text{ è il padre di } i, \\ 0 & \text{se } i \text{ è la radice;} \end{cases}$$

$$F[i] = \begin{cases} j & \text{se } j \text{ è il primo figlio di } i, \\ 0 & \text{se } i \text{ è una foglia;} \end{cases}$$

$$S[i] = \begin{cases} j & \text{se } j \text{ è il fratello successivo di } i, \\ 0 & \text{se } i \text{ non possiede un fratello successivo,} \end{cases}$$

per ogni nodo i. Per esempio nell'albero ordinato



i tre vettori sono definiti da

	P	F	S
1	2	0	3
2	4	1	5
3	2	0	0
4	0	2	0
5	4	6	0
6	5	0	7
7	5	0	8
8	5	0	0

Osserviamo che in un albero ordinato un figlio della radice coi suoi discendenti forma a sua volta un albero ordinato. Questo fatto permette di dare la seguente definizione induttiva di albero ordinato, di grande interesse per il progetto e la dimostrazione di correttezza di algoritmi su alberi:

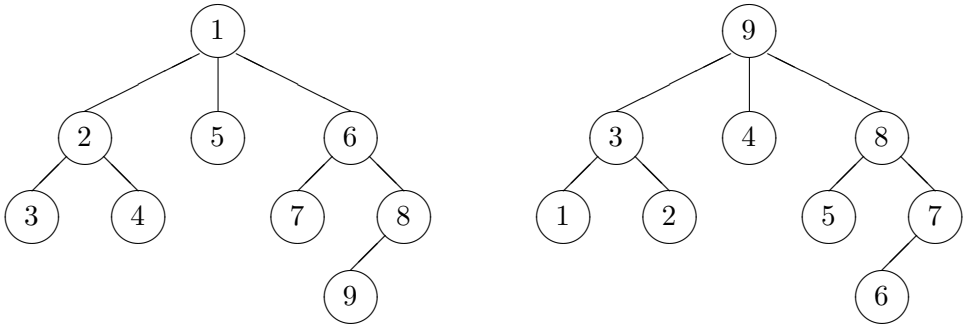
1) l'albero costituito da un solo vertice r è un albero ordinato;
2) se $T_1, T_2, \ldots, T_m$ sono alberi ordinati (definiti su insiemi di vertici disgiunti) e r è un nodo diverso dai nodi di $T_1, T_2, \ldots, T_m$, allora la sequenza $\langle r, T_1, \ldots, T_m \rangle$ è un albero ordinato.
In entrambi i casi diciamo che r è la radice dell'albero.

Consideriamo ad esempio il problema di attraversamento di alberi, cioè la visita dei vertici di un albero in qualche ordine. Due classici metodi di attraversamento sono quello in ordine anticipato e quello in ordine posticipato (pre-ordine e post-ordine), descritti dai due seguenti schemi.

Attraversamento in pre-ordine dell'albero ordinato T
Se T e' costituito da un solo nodo r allora visita r;
altrimenti, se $T = \langle r, T_1, \ldots, T_m \rangle$ allora: 1) visita la radice r di T;
2) attraversa in pre-ordine gli alberi $T_1, T_2, \ldots, T_m$.

Attraversamento in post-ordine dell'albero ordinato T
Se T e' costituito da un solo nodo r allora visita r;
altrimenti, se $T = \langle r, T_1, \ldots, T_m \rangle$ allora: 1) attraversa in post-ordine gli alberi $T_1, T_2, \ldots, T_m$;
2) visita la radice r di T.

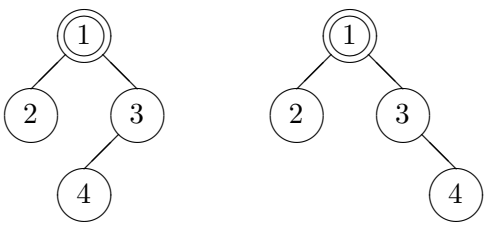
La correttezza dei metodi è immediatamente dimostrabile per induzione. Nella seguente figura mettiamo in evidenza l'ordine di visita dei nodi di un albero ordinato a seconda dei due criteri di attraversamento.



pre-ordine post-ordine

4.6.3 Alberi binari

Un *albero binario* è un albero radicato in cui ogni nodo interno ha al più due figli; ogni figlio è distinto come figlio *sinistro* oppure figlio *destro*. I seguenti due alberi sono coincidenti come alberi ordinati, ma distinti come alberi binari:



Dato un vertice x di un albero T binario, il sottoalbero che ha come radice il figlio sinistro di x (risp. al figlio destro di x), se esiste, sarà detto "sottoalbero sinistro di x" (risp. "sottoalbero destro di x").

Un albero binario può essere agevolmente rappresentato attraverso le due tabelle *sin* e *des*, che associano ad ogni vertice x rispettivamente il suo figlio sinistro (o "O" se non esiste) e il suo figlio destro (o "O" se non esiste).

Un *albero completo* di altezza h è un albero in cui tutte le foglie hanno profondità h ed ogni altro vertice ha due figli. Il numero di vertici di un albero binario completo è allora

$$n = \sum_{j=0}^{h} 2^j = 2^{h+1} - 1$$

e quindi $h \sim \lg_2 n$ (per $n \to +\infty$). Alberi completi (o "quasi" completi) contengono quindi un "gran numero" di nodi con una "bassa" altezza.

Viceversa, l'albero binario con n nodi che ha altezza massima è quello nel quale ogni nodo interno possiede un solo figlio. In questo caso l'altezza dell'albero è chiaramente $h = n - 1$.

Anche gli alberi binari possono essere attraversati in ordine anticipato o posticipato. Tuttavia un metodo tipico di visita dei nodi di un albero binario è quello di attraversamento in ordine simmetrico: prima visita il sottoalbero sinistro, poi la radice, poi il sottoalbero destro. Lo schema corrispondente può essere descritto nel modo seguente.

Attraversamento in ordine simmetrico dell'albero binario B

Sia r la radice di B;

1) se r ha figlio sinistro allora attraversa il sottoalbero sinistro di r;
2) visita r;
3) se r ha figlio destro allora attraversa il sottoalbero destro di r.

Anticipando l'argomento del prossimo capitolo presentiamo una procedura ricorsiva che visita i nodi di un albero binario assegnando a ciascun vertice il corrispondente numero d'ordine secondo l'ordinamento simmetrico. L'input dell'algoritmo è un albero binario di n vertici, rappresentato da una coppia di vettori *sin* e *des* di n componenti, definiti come sopra, e da un intero r che rappresenta la radice. L'uscita è data dal vettore N di n componenti nel quale $N[v]$ è il numero d'ordine del nodo v. L'algoritmo è definito da un programma principale e dalla procedura IN_ORDINE nella quale i parametri c, sin, des e N rappresentano variabili globali.

```
begin
      c:=1
      IN_ORDINE(r)
end

Procedura IN_ORDINE(x)
 if  sin[x] ≠ 0   then   IN_ORDINE(sin[x])
     N[x] := c
     c := c + 1
 if  des[x] ≠ 0   then   IN_ORDINE(des[x])
```

Esercizi

1) Dimostrare che ogni albero con n nodi possiede $n-1$ lati.
2) Mostrare che l'altezza h di ogni albero binario di n nodi soddisfa la relazione

$$\lfloor \log_2 n \rfloor \leq h \leq n-1.$$

Determinare un albero binario di n nodi nel quale $h = \lfloor \log_2 n \rfloor$ e uno nel quale $h = n-1$.
3) Si definisce *localmente completo* un albero binario nel quale ogni nodo interno possiede due figli. Provare che ogni albero binario localmente completo possiede un numero dispari di nodi. Un tale albero può avere un numero pari di foglie?
4) Provare che se un albero binario di altezza h possiede m foglie ed è localmente completo, allora

$$h+1 \leq m \leq 2^h.$$

In quale caso abbiamo $m = h+1$ e in quale $m = 2^h$?
5) In un albero con radice la *lunghezza di cammino* è definita come la somma delle profondità dei nodi. Mostrare che in ogni albero binario con n nodi la lunghezza di cammino L soddisfa la relazione

$$\sum_{k=1}^{n} \lfloor \log_2 k \rfloor \leq L \leq \frac{n(n-1)}{2}.$$

6) Considera la numerazione preordine di un albero binario completo e supponi che un nodo interno v abbia numero d'ordine i. Qual è il numero d'ordine del figlio sinistro di v e quale di quello destro?
7) Abbiamo visto come un albero ordinato possa essere rappresentato da una famiglia di liste di adiacenza $L(v)$, una per ogni nodo v, oppure mediante i vettori P, F, S che definiscono rispettivamente il padre, il primo figlio e il fratello successivo di ciascun vertice. Utilizzando le operazioni definite sulle liste e sui vettori, descrivere una procedura che permette di passare dalla prima rappresentazione alla seconda e viceversa.
8) Assumendo il criterio di costo uniforme, determina l'ordine di grandezza del tempo di calcolo richiesto dall'algoritmo descritto nell'esercizio precedente al crescere del numero n di nodi.
9) Diciamo che due alberi ordinati $T_1 = \langle V_1, E_1 \rangle$, $T_2 = \langle V_2, E_2 \rangle$ sono *isomorfi* e scriviamo $T_1 \equiv T_2$, se esiste una funzione biunivoca $f : V_1 \to V_2$ tale che, per ogni coppia di nodi $v, w \in V_1$:
- $(v,w) \in E_1 \Leftrightarrow (f(v), f(w)) \in E_2$;
- w è il j-esimo figlio di v in $T_1 \Leftrightarrow f(w)$ è il j-esimo figlio di $f(v)$ in T_2.

Dimostrare che $\equiv$ è una relazione di equivalenza sull'insieme degli alberi ordinati.
10) Continuando l'esercizio precedente, chiamiamo albero ordinato *non etichettato* una classe di equivalenza della relazione di isomorfismo $\equiv$ definita sopra. Graficamente esso può essere rappresentato come un albero ordinato privo del nome dei nodi (due alberi ordinati non etichettati si distinguono pertanto solo per la "forma" dell'albero). Rappresentare graficamente tutti gli alberi ordinati non etichettati che hanno al più 4 nodi. Dare una definizione analoga per gli alberi *binari* non etichettati e rappresentare tutti gli alberi di questo tipo che hanno al più 3 nodi.
11) Dimostrare che per ogni $n \in \mathbb{N}$ il numero di alberi binari non etichettati con n nodi equivale al numero di alberi ordinati non etichettati che hanno $n+1$ nodi.

4.7 Esempio: attraversamento di grafi in ampiezza

Mostriamo ora con un esempio come si può descrivere un algoritmo utilizzando alcune strutture dati introdotte nelle sezioni precedenti insieme alle relative operazioni. L'uso di questi strumenti permette di progettare e descrivere una procedura in maniera semplice e concisa, mettendo in luce l'organizzazione dei dati e il metodo adottato per ottenere la soluzione del problema. In questo modo un algoritmo può essere descritto ad alto livello, trascurando i dettagli implementativi e ponendo in evidenza il suo schema generale che spesso si riduce all'esecuzione di una sequenza di operazioni su strutture dati. Una volta fissato tale schema si potrà poi scegliere come implementare le strutture dati usate e le procedure che eseguono le corrispondenti operazioni.

Come esempio presentiamo un classico algoritmo di esplorazione di grafi basato sull'uso di una coda. Il problema è definito nel modo seguente. Dato un grafo non orientato e connesso, vogliamo visitare i suoi nodi, uno dopo l'altro, a partire da un nodo assegnato che chiamiamo sorgente. L'ordine di visita dipende dalla distanza dalla sorgente: una volta visitata quest'ultima, prima visitiamo tutti i nodi adiacenti, poi i nodi che si trovano a distanza 2 e così via, fino a quando tutti i vertici del grafo sono stati considerati. Questo tipo di visita si chiama *attraversamento in ampiezza* e rappresenta uno dei due metodi fondamentali per esplorare un grafo (l'altro, detto attraversamento in profondità, sarà presentato nel capitolo successivo). Esso può essere esteso facilmente ai grafi diretti e con ovvie modifiche anche a quelli non connessi.

Un metodo naturale per compiere un attraversamento in ampiezza è quello di mantenere in una coda una sequenza di nodi, visitando di volta in volta il vertice che si trova in testa e introducendo all'estremità opposta i nodi adiacenti che non sono ancora stati raggiunti. Sarà quindi necessario marcare i nodi del grafo per segnalare quelli che sono già stati introdotti nella coda; inizialmente il nodo sorgente è l'unico vertice che si trova in coda e quindi il solo marcato.

Durante la visita l'algoritmo determina naturalmente un albero di copertura del grafo che si ottiene considerando per ogni nodo v (diverso dalla sorgente) il lato che ha permesso di considerare v per la prima volta. Si tratta quindi di un albero che ha per radice la sorgente, formato dai lati che, uscendo da un nodo appena visitato, consentono di determinare i nuovi vertici da introdurre in coda.

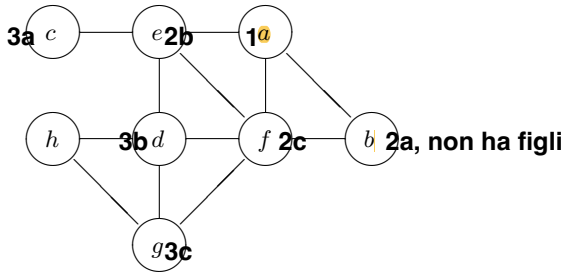
Formalmente, sia $G = \langle V, E\rangle$ un grafo non orientato e connesso e sia $s \in V$ un nodo qualsiasi (sorgente). Per semplicità rappresentiamo G mediante una famiglia di liste di adiacenza; per ogni $v \in V$, denotiamo con $L(v)$ la lista dei vertici adiacenti a v. L'algoritmo visita in ampiezza i nodi di G e fornisce in uscita la lista U contenente i lati dell'albero di copertura prodotto. Denotiamo con Q la coda mantenuta dall'algoritmo; i vertici vengono inizialmente marcati come "nuovi" e la marca viene cambiata non appena questi vengono inseriti in Q. La marcatura può essere facilmente implementata mediante un vettore. L'algoritmo è descritto dalla seguente procedura:

```
Procedure Ampiezza(G, s)
begin
      U := Λ
      for v ∈ V do  marca v come "nuovo"
      Q := ENQUEUE(Λ, s)
      marca s come "vecchio"
      while IS_EMPTY(Q) = 0 do
               begin
                     v := FRONT(Q)
                     Q := DEQUEUE(Q)
                     visita v
                     for w ∈ L(v) do
                           if w marcato "nuovo" then { marca w come "vecchio"
                                                     { U := INSERISCI_IN_TESTA({v, w}, U)
                                                     { Q := ENQUEUE(Q, w)
               end
end
```

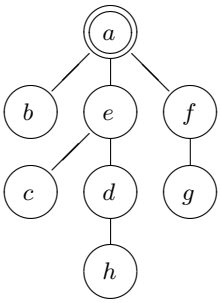
Nota che durante l'esecuzione dell'algoritmo ogni nodo $v \in V$ si può trovare in una delle seguenti condizioni:

- v è già stato visitato e in questo caso non appartiene a Q ed è marcato "vecchio";
- v non è stato visitato ma è adiacente ad un nodo visitato. In questo caso appartiene a Q ed è marcato "vecchio";
- v non soddisfa alcuna delle condizioni precedenti e quindi è ancora marcato "nuovo".

Esempio 4.1
Applichiamo l'algoritmo appena descritto al grafo definito dalla seguente figura, supponendo che a sia il nodo sorgente:



Supponendo che le liste dei nodi adiacenti siano ordinate in ordine alfabetico, l'albero di copertura in ampiezza ottenuto è dato dalla seguente immagine. Nota che l'ordine di visita dei nodi procede dall'alto verso il basso e da sinistra verso destra.



∎

Valutiamo ora il tempo di calcolo e lo spazio di memoria richiesti dall'algoritmo su un grafo di n nodi e m lati, assumendo il criterio di costo uniforme. Osserviamo innazitutto che il primo ciclo `for` (nella seconda riga) richiede un tempo dell'ordine di $\Theta(n)$ poiché il ciclo è ripetuto proprio n volte e la marcatura di un nodo richiede un tempo costante. Anche il ciclo `while` viene ripetuto una volta per ogni nodo v del grafo; il tempo richiesto ad ogni iterazione è dell'ordine $\Theta(1) + \Theta(\ell(v))$, dove $\ell(v)$ è la lunghezza della lista $L(v)$. Di conseguenza il tempo di calcolo è dato da

$$\Theta(n) + \sum_{v \in V} \Theta(\ell(v)) = \Theta(n + m)$$

Lo spazio di memoria richiesto è invece determinato dalle celle necessarie per mantenere il grafo di ingresso e da quelle utilizzate per conservare la coda Q. La prima quantità è di ordine $\Theta(n + m)$ mentre la seconda è di ordine $\Theta(n)$ nel caso peggiore. Ne segue che anche lo spazio di memoria richiesto è di ordine $\Theta(n + m)$.

Esercizi

1) Per quali grafi di n nodi l'algoritmo di visita in ampiezza richiede il massimo numero di celle di memoria per mantenere la coda Q e per quali il minimo?

2) Descrivere un algoritmo di visita in ampiezza per grafi orientati. Su quali grafi orientati l'algoritmo produce un albero di copertura in ampiezza?

3) Descrivere un algoritmo per determinare le distanze di tutti i nodi da una sorgente in un grafo non orientato connesso. Svolgere l'analisi dei tempi di calcolo e dello spazio di memoria utilizzati.

Capitolo 5

Procedure ricorsive

L'uso di procedure ricorsive permette spesso di descrivere un algoritmo in maniera semplice e concisa, mettendo in rilievo la tecnica adottata per la soluzione del problema e facilitando quindi la fase di progettazione. Inoltre l'analisi risulta in molti casi semplificata poiché la valutazione del tempo di calcolo si riduce alla soluzione di equazioni di ricorrenza.

Questo capitolo è dedicato all'analisi delle procedure di questo tipo; vogliamo innanzitutto mostrare come in generale queste possano essere trasformate in programmi iterativi, che non prevedono cioè l'esecuzione di chiamate ricorsive, direttamente implementabili su macchine RASP (o RAM). Un pregio particolare di questa traduzione è quello di mettere in evidenza la quantità di spazio di memoria richiesto dalla ricorsione.

Tra gli algoritmi che si possono facilmente descrivere mediante procedure ricorsive risultano di particolare interesse la ricerca binaria e gli algoritmi di esplorazione di alberi e grafi.

5.1 Analisi della ricorsione

Una procedura che chiama se stessa, direttamente o indirettamente, viene detta *ricorsiva*. Consideriamo ad esempio il seguente problema: determinare il numero massimo di parti in cui n rette dividono il piano. Detto $p(n)$ tale numero, il disegno di un algoritmo ricorsivo per calcolare $p(n)$ è basato sulla seguente proprietà:

1. Una retta divide il piano in due parti, cioè $p(1) = 2$;
2. Sia $p(n)$ il numero di parti in cui n rette dividono il piano. Aggiungendo una nuova retta, è facile osservare che essa è intersecata in al più n punti dalle precedenti, creando al più $n+1$ nuove parti, come si può osservare dalla figura 5.1.
   Vale quindi: $p(n+1) = p(n) + n + 1$

Otteniamo dunque la seguente procedura ricorsiva:

```
Procedura P(n)
  if n = 1 then  return 2
           else  x := P(n − 1)
                 return (x + n)
```

Questa tecnica di disegno cerca quindi di esprimere il valore di una funzione su un dato in dipendenza di valori della stessa funzione sui dati possibilmente "più piccoli". Molti problemi si prestano in modo

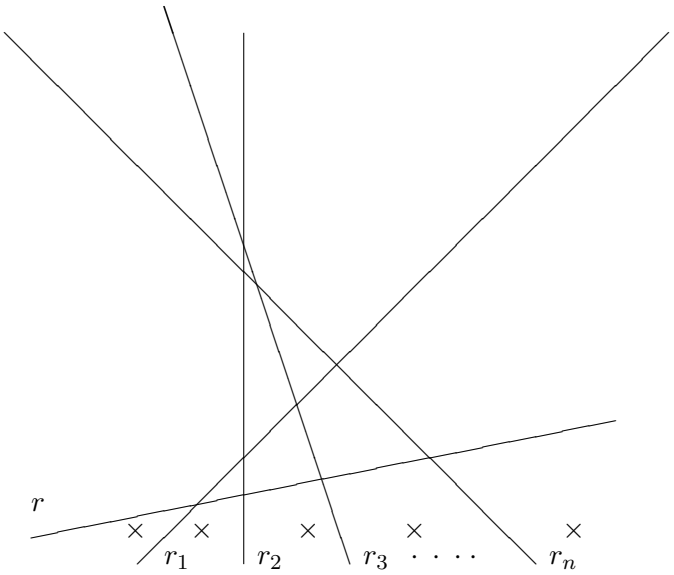


Figura 5.1: Nuove parti di piano create da una retta r.

naturale ad essere risolti con procedure ricorsive, ottenendo algoritmi risolutivi in generale semplici e chiari.

Una difficoltà che sorge è legata al fatto che il modello di macchina RAM (o RASP) non è in grado di eseguire direttamente algoritmi ricorsivi. Ne consegue che è di estremo interesse trovare tecniche di traduzione di procedure ricorsive in codice RAM e sviluppare metodi che permettano di valutare le misure di complessità dell'esecuzione del codice tradotto semplicemente analizzando le procedure ricorsive stesse. Qui delineiamo una tecnica per implementare la ricorsione su macchine RASP semplice e diretta, mostrando nel contempo che il problema di stimare il tempo di calcolo del programma ottenuto può essere ridotto alla soluzione di equazioni di ricorrenza. Questo metodo di traduzione delle procedure ricorsive in programmi puramente iterativi (nei quali cioè la ricorsione non è permessa) è del tutto generale e a grandi linee è lo stesso procedimento applicato dai compilatori che di fatto traducono in linguaggio macchina programmi scritti in un linguaggio ad alto livello.

Cominciamo osservando che la chiamata di una procedura `B` da parte di `A` (qui `A` può essere il programma principale, oppure una procedura, oppure la procedura `B` stessa) consiste essenzialmente di due operazioni:

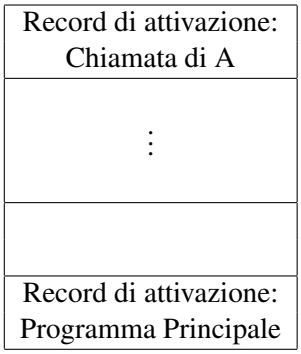
1. passaggio dei parametri da `A` a `B`;
2. cessione del controllo da `A` a `B` così da permettere l'inizio della esecuzione di `B` conservando in memoria il "punto di ritorno", cioè l'indirizzo dell'istruzione che deve essere eseguita nella procedura `A` una volta terminata l'esecuzione di `B`.

Si hanno ora due possibilità:

1. l'esecuzione di `B` può terminare; a questo punto viene passato ad `A` il valore della funzione calcolata da `B` (se `B` è una procedura di calcolo di una funzione) e l'esecuzione di `A` riprende dal "punto di ritorno";
2. `B` chiama a sua volta una nuova procedura.

Come conseguenza si ha che l'ultima procedura chiamata è la prima a terminare l'esecuzione: questo giustifica l'uso di una pila per memorizzare i dati di tutte le chiamate di procedura che non hanno ancora terminato l'esecuzione.

Gli elementi della pila sono chiamati "record di attivazione" e sono identificati da blocchi di registri consecutivi; ogni chiamata di procedura usa un record di attivazione per memorizzare i dati non globali utili.



Supponiamo che, come nello schema precedente, la procedura `A` sia correntemente in esecuzione e chiami la procedura `B`. In questo caso l'esecuzione di `B` prevede le seguenti fasi:

1. Viene posto al top nella pila un nuovo "record di attivazione" per la chiamata di `B` di opportuna dimensione; tale record contiene:
   (a) puntatori ai parametri attuali che si trovano in `A`,
   (b) spazio per le variabili locali di `B`,
   (c) l'indirizzo dell'istruzione di `A` che deve essere eseguita quando `B` termina l'esecuzione (punto di ritorno);
   (d) se `B` calcola una funzione, in `B` viene posto un puntatore a una variabile di `A` in cui sarà memorizzato il valore della funzione calcolata da `B`.
2. Il controllo viene passato alla prima istruzione di `B`, iniziando così l'esecuzione di `B`.
3. Quando `B` termina l'esecuzione, il controllo ritorna ad `A` mediante i seguenti passi:
   (a) se `B` è una procedura che calcola una funzione, l'esecuzione ha termine con un comando "`return` E"; il valore di E viene calcolato e passato all'opportuna variabile nel record di attivazione della chiamata di `A`;
   (b) il "punto di ritorno" è ottenuto dal record di attivazione di `B`;
   (c) il record di attivazione di `B` viene tolto dal top della pila; l'esecuzione di `A` può continuare.

È quindi possibile associare ad un algoritmo ricorsivo un algoritmo eseguibile su RASP (o RAM) che calcola la stessa funzione. Tale algoritmo rappresenterà la semantica operazionale RASP dell'algoritmo ricorsivo.

Allo scopo di disegnare schematicamente l'algoritmo, diamo per prima cosa la descrizione del record di attivazione di una procedura `A` del tipo:

```
Procedura A
      ⋮
   z := B(a)
  (1) Istruzione
      ⋮
```

Se nel corso della esecuzione A chiama B, il record di attivazione di B viene mostrato nella figura 5.2.

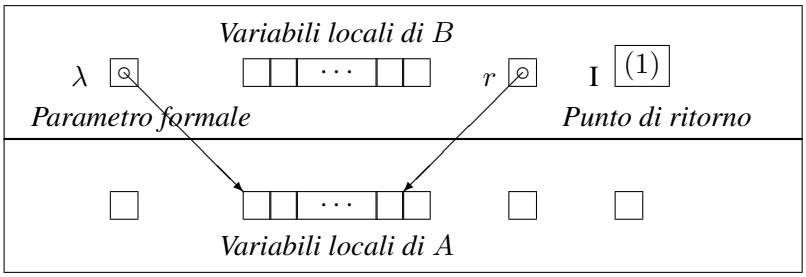


Figura 5.2: Esempio di record di attivazione.

Il cuore della simulazione iterativa di una procedura ricorsiva è allora delineato nel seguente schema:

R := Record di attivazione del programma principale (Main)
I := Indirizzo della prima istruzione del programma principale
$Pila$:= PUSH(Λ, R)
repeat
  N := Nome della procedura con registro di attivazione in TOP$(Pila)$
  ist := Istruzione di N di indirizzo I
  while ist non e' di arresto ne' una chiamata di una procedura do
    begin
      Esegui ist
      ist := Istruzione successiva
    end
  if ist e' una chiamata di A then R := Record di attivazione della chiamata di A
    I := Indirizzo della prima istruzione di A
    $Pila$:=PUSH$(Pila, R)$
  if ist e' di arresto then Valuta il risultato inviandolo al programma chiamante
    I := Indirizzo di ritorno
    $Pila$:= POP$(Pila)$
until $Pila = \Lambda$

A scopo esemplificativo, consideriamo il programma:

```
      read(m)
      z:= FIB(m)
  (W) write(z)
```

Questo programma chiama la procedura FIB:

```
Procedura FIB (n)
     if n ≤ 1 then return n
            else     a := n − 1
                     x := FIB(a)
                (A)  b := n − 2
                     y := FIB(b)
                (B)  return(x + y)
```

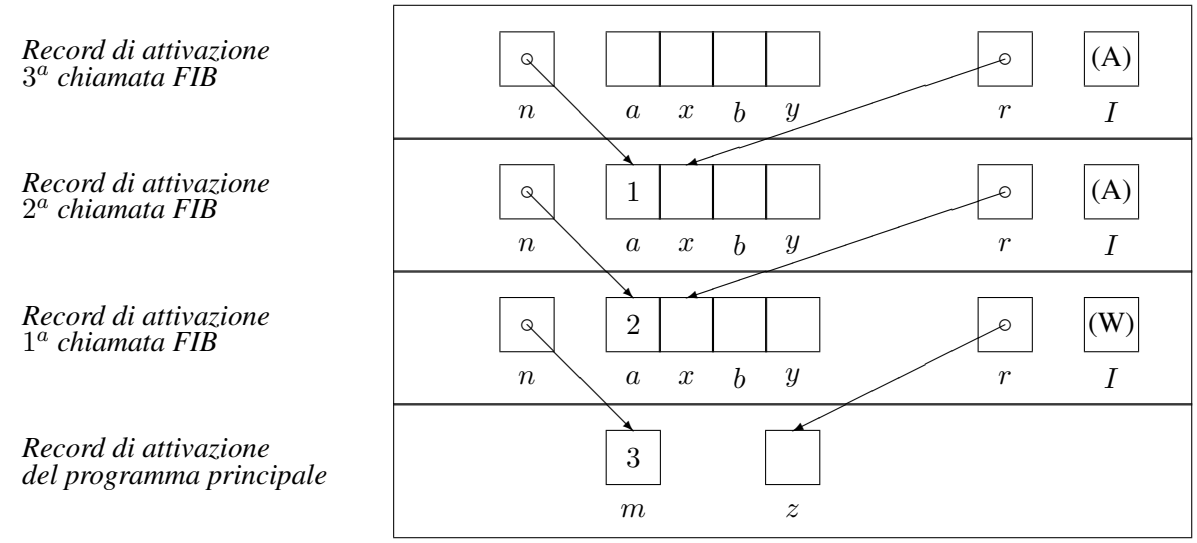


Figura 5.3: Contenuto della pila per la procedura FIB su ingresso 3 nei primi 4 passi di esecuzione.

La procedura FIB contiene due possibili chiamate a se stessa ($x :=$ FIB(a), $y :=$ FIB(b)). Le etichette (W), (A), (B) sono gli eventuali punti di ritorno.

È facile osservare che su ingresso m il programma principale stampa l' m-esimo numero di Fibonacci $f(m)$, dove $f(0) = 0$, $f(1) = 1$ e $f(n) = f(n-1) + f(n-2)$ per $n > 1$.

La figura 5.3 illustra il contenuto della pila dopo 4 chiamate su ingresso 3.

Affrontiamo ora il problema di analisi degli algoritmi ricorsivi: dato un algoritmo ricorsivo, stimare il tempo di calcolo della sua esecuzione su macchina RASP (o RAM). Tale stima può essere fatta agevolmente senza entrare nei dettagli della esecuzione iterativa, ma semplicemente attraverso l'analisi della procedura ricorsiva stessa.

Si procede come segue, supponendo che l'algoritmo sia descritto da M procedure $P_1, P_2, \ldots, P_M$, comprendenti il programma principale:

1. Si associa ad ogni indice k, $1 \leq k \leq M$, la funzione (incognita) $T_k(n)$, che denota il tempo di calcolo di P_k in funzione di qualche parametro n dell'ingresso.

2. Si esprime $T_k(n)$ in funzione dei tempi delle procedure chiamate da P_k, valutati negli opportuni valori dei parametri; a tal riguardo osserviamo che il tempo di esecuzione della istruzione $z := P_j(a)$ è la somma del tempo di esecuzione della procedura P_j su ingresso a, del tempo di chiamata di P_j (necessario a predisporre il record di attivazione) e di quello di ritorno.

Si ottiene in tal modo un sistema di M equazioni di ricorrenza che, risolto, permette di stimare $T_k(n)$ per tutti i k ($1 \leq k \leq M$), ed in particolare per il programma principale. Allo studio delle equazioni di ricorrenza sarà dedicato il prossimo capitolo.

A scopo esemplificativo, effettuiamo una stima del tempo di calcolo della procedura `FIB`(n) col criterio di costo uniforme. Per semplicità, supponiamo che la macchina sia in grado di effettuare una chiamata in una unità di tempo (più un'altra unità per ricevere il risultato). Sia $T_{FIB}(n)$ il tempo di calcolo (su RAM !) della procedura ricorsiva `FIB` su input n; tale valore può essere ottenuto attraverso l'analisi dei tempi richiesti dalle singole istruzioni:

```
Procedura FIB (n)
     if n ≤ 1 (Tempo : 2) then    return n              (Tempo : 1)

                          else    a := n − 1            (Tempo : 3)
                                  x := FIB(a)           (Tempo : 2 + T_FIB(n − 1) )
                             (A)  b := n − 2            (Tempo : 3)
                                  y := FIB(b)           (Tempo : 2 + T_FIB(n − 2) )
                             (B)  return(x + y)         (Tempo : 4 )
```

Vale allora la seguente equazione di ricorrenza:
$T_{FIB}(0) = T_{FIB}(1) = 3$,
$T_{FIB}(n) = 16 + T_{FIB}(n-1) + T_{FIB}(n-2)$, per ogni $n \geq 2$.

Esercizi

1) Valutare l'ordine di grandezza dello spazio di memoria richiesto dalla procedura FIB su input n assumendo il criterio di costo uniforme.

2) Svolgere l'esercizio precedente assumendo il criterio di costo logaritmico.

5.2 Ricorsione terminale

Il metodo generale per la traduzione iterativa della ricorsione descritto nella sezione precedente può essere semplificato e reso più efficiente quando la chiamata a una procedura è l'ultima istruzione eseguita dal programma chiamante. In questo caso infatti, una volta terminata l'esecuzione della procedura chiamata, non occorre restituire il controllo a quella chiamante.

Per descrivere la traduzione iterativa di questa ricorsione, denotiamo rispettivamente con A e B la procedura chiamante e quella chiamata e supponiamo che la chiamata di B sia l'ultima istruzione del programma A. Possiamo allora eseguire la chiamata a B semplicemente sostituendo il record di attivazione di A con quello di B nella pila e aggiornando opportunamente l'indirizzo di ritorno alla procedura che ha chiamato A; il controllo passerà così a quest'ultima una volta terminata l'esecuzione di B. In questo modo si riduce il numero di record di attivazione mantenuti nella pila, si risparmia tempo di calcolo e spazio di memoria rendendo quindi più efficiente l'implementazione.

Questo tipo di ricorsione viene chiamata *ricorsione terminale*. Un caso particolarmente semplice si verifica quando l'algoritmo è formato da un'unica procedura che richiama se stessa all'ultima istruzione. In tale situazione non occorre neppure mantenere una pila per implementare la ricorsione perchè non è necessario riattivare il programma chiamante una volta terminato quello chiamato.

Il seguente schema di procedura rappresenta un esempio tipico di questo caso. Consideriamo una procedura F, dipendente da un parametro x, definita dal seguente programma:

```
Procedura F(x)
 if  C(x) then  D
         else  begin
                     E
                     y := g(x)
                     F(y)
               end
```

Qui $C(x)$ è una condizione che dipende dal valore di x, mentre E e D sono opportuni blocchi di istruzioni. La funzione $g(x)$ invece determina un nuovo valore del parametro di input per la F di dimensione ridotta rispetto a quello di x.

Allora, se a è un qualunque valore per x, la chiamata $F(a)$ è equivalente alla seguente procedura:

```
begin
       x := a
       while ¬C(x) do
             begin
                   E
                   x := g(x)
             end
       D
end
```

5.2.1 Ricerca binaria

Consideriamo il seguente problema di ricerca:

> Istanza: un vettore ordinato B di n interi e un numero intero a;
>
> Soluzione: un intero $k \in \{1, 2, \ldots, n\}$ tale che $B[k] = a$, se tale intero esiste, 0 altrimenti.

Il problema può essere risolto applicando il noto procedimento di ricerca binaria. Questo consiste nel confrontare a con l'elemento del vettore B di indice $k = \lfloor \frac{n+1}{2} \rfloor$. Se i due elementi sono uguali si restituisce l'intero k; altrimenti, si prosegue la ricerca nel sottovettore di sinistra, $(B[1], \ldots, B[k-1])$, o in quello di destra, $(B[k+1], \ldots, B[n])$, a seconda se $a < B[k]$ oppure $a > B[k]$.

Il procedimento è definito in maniera naturale mediante la procedura ricorsiva Ricercabin(i, j) che descriviamo nel seguito. Questa svolge la ricerca nel sottovettore delle componenti di B comprese tra gli indici i e j, $1 \leq i \leq j \leq n$, supponendo i parametri a e B come variabili globali.

```
Procedura Ricercabin(i, j)
if  j < i then  return 0
        else  begin
                    k := ⌊(i+j)/2⌋
                    if a = B[k] then return k
                              else if a < B[k] then return Ricercabin(i, k − 1)
                                                else return Ricercabin(k + 1, j)
              end
```

L'algoritmo che risolve il problema è quindi limitato alla chiamata Ricercabin$(1, n)$. La sua analisi è molto semplice poiché , nel caso peggiore, a ogni chiamata ricorsiva la dimensione del vettore su cui si svolge la ricerca viene dimezzata. Assumendo quindi il criterio di costo uniforme l'algoritmo termina in $O(\log n)$ passi.

Cerchiamo ora di descrivere una versione iterativa dello stesso procedimento. Osserviamo che il programma appena descritto esegue una ricorsione terminale e questa è l'unica chiamata ricorsiva della procedura. Applicando allora il procedimento definito sopra otteniamo il seguente programma iterativo nel quale non viene utilizzata alcuna pila.

```
begin
      i := 1
      j := n
      out := 0
      while i ≤ j ∧ out = 0 do
            begin
                  k := ⌊(i+j)/2⌋
                  if a = B[k] then out := k
                            else if a < B[k] then j := k − 1
                                             else i := k + 1
            end
      return out
end
```

Osserva che lo spazio di memoria richiesto da quest'ultima procedura, escludendo quello necessario per mantenere il vettore B di ingresso, è $O(1)$ secondo il criterio uniforme.

5.3 Attraversamento di alberi

I procedimenti di visita dei nodi di un albero ordinato descritti nel capitolo precedente sono facilmente definiti mediante procedure ricorsive che ammettono semplici traduzioni iterative. Si tratta di algoritmi che hanno una loro importanza intrinseca perchè sono utilizzati in numerose applicazioni dato il largo uso degli alberi per rappresentare insiemi di dati organizzati gerarchicamente.

Supponiamo di voler visitare secondo l'ordine anticipato (preordine) i nodi di un albero ordinato T, di radice r, definito mediante una famiglia di liste di adiacenza $L(v)$, una per ogni vertice v di T. Supponiamo inoltre che T abbia n nodi rappresentati mediante i primi n interi positivi. Vogliamo associare, ad ogni nodo v, il numero d'ordine di v secondo la numerazione anticipata (cioè il numero di nodi visitati prima di v incrementato di 1). L'algoritmo nella sua versione ricorsiva è definito dalla

inizializzazione di una variabile globale c che indica il numero d'ordine del nodo corrente da visitare e dalla chiamata della procedura Visita(r).

```
begin
      c := 1
      Visita(r)
end
```

La procedura Visita è data dal seguente programma ricorsivo che utilizza, come variabile globale, il vettore N di n componenti e, per ogni nodo v, calcola in $N[v]$ il numero d'ordine di v.

```
Procedura Visita(v)
begin
      N[v] := c
      c := c + 1
      for w ∈ L(v) do Visita(w)
end
```

Descriviamo ora la traduzione iterativa dell'algoritmo applicando il metodo illustrato nella sezione precedente. Il procedimento è basato sulla gestione della pila S che conserva semplicemente una lista di nodi per mantenere la traccia delle chiamate ricorsive. In questo caso infatti l'unica informazione contenuta in ogni record di attivazione è costituita dal nome del nodo visitato.

```
begin
      v := r
      c := 1
      S := Λ
(1)   N[v] := c
      c := c + 1
(2)   if IS_EMPTY(L(v)) = 0 then
        begin
              w :=TESTA(L(v))
              L(v) :=TOGLI_IN_TESTA(L(v))
              S :=PUSH(S, v)
              v := w
              go to (1)
        end
      else if IS_EMPTY(S) = 0 then
        begin
              v :=TOP(S)
              S :=POP(S)
              go to (2)
        end
end
```

Nota che l'istruzione di etichetta (2) rappresenta il punto di ritorno di ogni chiamata ricorsiva mentre quella di etichetta (1) corrisponde all'inizio della procedura ricorsiva Visita(v).

Questa versione iterativa può tuttavia essere migliorata tenendo conto delle chiamate terminali. Infatti nella procedura ricorsiva la chiamata Visita(w), quando w rappresenta l'ultimo figlio del nodo v, è l'ultima istruzione del programma. Possiamo così modificare la versione iterativa dell'algoritmo ottenendo il seguente programma (nel quale si sono anche eliminati i comandi `go to`).

```
begin
      v := r
      N[v] := 1
      c := 2
      S := Λ
      out := 0
      repeat
            while IS_EMPTY(L(v)) = 0 then
                  begin
                        w :=TESTA(L(v))
                        L(v) :=TOGLI_IN_TESTA(L(v))
                        if IS_EMPTY(L(v)) = 0 then S :=PUSH(S, v)
                        v := w
                        N[v] := c
                        c := c + 1
                  end
            if IS_EMPTY(S) = 0  then { v :=TOP(S)
                                     { S :=POP(S)
                                else out := 1
      until out = 1
end
```

È facile verificare che gli algoritmi precedenti permettono di visitare un albero ordinato di n nodi in $\Theta(n)$ passi, assumendo il criterio uniforme. Nel caso peggiore anche lo spazio richiesto dalla pila S è $\Theta(n)$. Invece nel caso migliore, applicando la procedura che implementa la ricorsione terminale, la pila S rimane vuota e quindi non richiede alcuna quantità di spazio. Questo si verifica quando l'albero di ingresso è formato da un cammino semplice.

Esercizi

1) Applicando opportune procedure di attraversamento, definire un algoritmo per ciascuno dei seguenti problemi aventi per istanza un albero ordinato T:

- calcolare il numero di discendenti di ogni nodo di T;
- calcolare l'altezza di ciascun nodo di T;
- calcolare la profondità di ciascun nodo di T.

2) Considera il seguente problema:

Istanza: un albero ordinato T di n nodi e un intero k, $1 \leq k \leq n$;
Soluzione: il nodo v di T che rappresenta il k-esimo vertice di T secondo la numerazione posticipata (post-ordine).

Definire una procedura ricorsiva per la sua soluzione senza applicare un attraversamento completo dell'albero in input.

3) Definire una procedura *non* ricorsiva per risolvere il problema definito nell'esercizio precedente.

4) Definire una procedura ricorsiva che attraversa un albero binario visitando ogni nodo interno prima e dopo aver visitato i suoi eventuali figli.

5) Descrivere una procedura *non* ricorsiva per risolvere il problema posto nell'esercizio precedente.

6) Tenendo conto dell'algoritmo definito nella sezione 4.6.3, definire una procedura *non* ricorsiva per risolvere il seguente problema:

> Istanza: un albero binario T di n nodi, rappresentato da due vettori *sin* e *des* di dimensione n;
>
> Soluzione: per ogni nodo v di T il numero d'ordine di v secondo la numerazione simmetrica (inorder).

Su quali input la pila gestita dall'algoritmo rimane vuota?

7) Supponendo che i nodi di T siano rappresentati dai primi n interi positivi, determinare l'ordine di grandezza del tempo di calcolo e dello spazio di memoria richiesti dalla procedura precedente assumendo il criterio di costo logaritmico.

8) Descrivere una procedura per verificare se due alberi ordinati sono isomorfi.

5.4 Attraversamento di grafi

Molti classici algoritmi su grafi sono basati su procedimenti che permettono di visitare tutti i nodi uno dopo l'altro. Per compiere questa visita esistono due strategie principali, chiamate rispettivamente attraversamento *in profondità* (depth-first search) e attraversamento *in ampiezza* (breadth-first search). Esse danno luogo a procedure di base molto comuni che hanno importanza notevole in svariate applicazioni. Abbiamo già descritto nel capitolo precedente una procedura per l'attraversamento in ampiezza di un grafo. Descriviamo ora una procedura per l'attraversamento in profondità. Dal punto di vista metodologico questo procedimento può essere espresso in maniera naturale mediante una procedura ricorsiva, che possiamo quindi analizzare applicando i metodi presentati in questo capitolo.

5.4.1 Visita in profondità

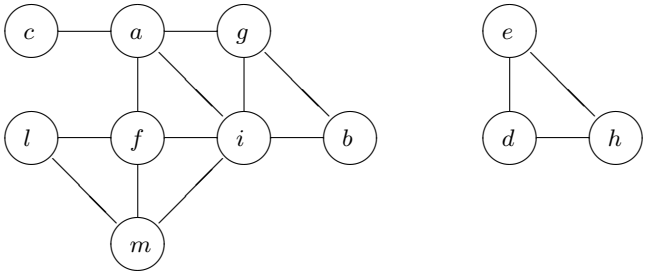
Intuitivamente nell'attraversamento in profondità si visita ciascuna componente connessa del grafo partendo da un nodo s e percorrendo un cammino, il più lungo possibile, fino a quando si giunge in un vertice nel quale tutti i nodi adiacenti sono gia' stati visitati; a questo punto si risale il cammino fino al primo nodo che ammette un vertice adiacente non ancora visitato e si ricomincia il procedimento di visita seguendo un nuovo cammino con lo stesso criterio. L'attraversamento termina quando tutti i percorsi ignorati nelle varie fasi della visita sono stati considerati. Nota che l'unione dei lati percorsi in questo modo forma un albero con radice che connette tutti i nodi della componente connessa considerata e i cui lati sono anche lati del grafo di partenza. Così l'algoritmo costruisce automaticamente una foresta di copertura del grafo di ingresso che chiamiamo foresta di copertura *in profondità* . Se il grafo è connesso questa si riduce ad un albero e parleremo allora di *albero* di copertura in profondità (depth first spanning tree).

L'algoritmo può essere formalmente descritto nel modo seguente. Consideriamo un grafo non orientato $G = \langle V, E \rangle$, rappresentato da liste di adiacenza. Usando la solita notazione, denotiamo con $L(v)$ la lista di adiacenza del nodo v, per ogni $v \in V$. L'algoritmo visita i nodi del grafo secondo il criterio sopra descritto e costruisce la relativa foresta di copertura fornendo in uscita la lista U dei suoi lati. Il procedimento può essere descritto mediante un programma principale che richiama una procedura ricorsiva per visitare i nodi di ogni componente connessa e determinare i lati del corrispondente albero di copertura. Inizialmente tutti i nodi sono marcati opportunamente in modo da riconoscere successivamente i vertici che non sono ancora stati visitati.

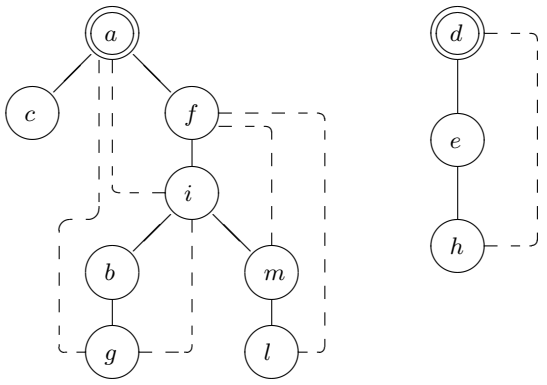
```
begin
        U := Λ
        for v ∈ V do marca v come "nuovo"
        for v ∈ V do  if v marcato "nuovo"then Profondita'(v)
        return U
end


Procedure Profondita'(v)
begin
        visita il nodo v
        marca v come "vecchio"
        for w ∈ L(v) do
            if w marcato "nuovo" then
                        begin
                              U := INSERISCI_IN_TESTA({v, w}, U)
                              Profondita'(w)
                        end
end
```

L'algoritmo quindi partiziona l'insieme dei lati E del grafo G in due sottoinsiemi: quelli che si trovano nella lista U in uscita, e quindi appartengono alla foresta di copertura costruita, e quelli che non vi appartengono. È facile verificare che ogni lato che non si trova in U al termine della procedura deve congiungere due nodi che sono uno successore dell'altro in un qualche albero della foresta (i due vertici devono cioè trovarsi sullo stesso cammino dalla radice a uno dei due nodi).

Esempio 5.1
Applichiamo l'algoritmo al grafo G descritto nella seguente figura.



Supponiamo che nel programma principale i nodi vengano considerati in ordine alfabetico e che nello stesso ordine siano disposti i vertici in ciascuna lista di adiacenza. Allora la foresta di copertura ottenuta ha per radici i nodi a e d ed è formata dai seguenti alberi ai quali abbiamo aggiunto (tratteggiati) i lati di G che non fanno parte della foresta calcolata.



∎

L'analisi dell'algoritmo è semplice. Assumiamo nuovamente il criterio uniforme e supponiamo che la visita di ogni nodo richieda tempo costante. Se il grafo di ingresso possiede n nodi e m lati, allora si eseguono $\Theta(n + m)$ passi. Infatti il costo dell'algoritmo è dato dalle marcature iniziali dei nodi, dalle chiamate alla procedura *Profondita'* effettuate dal programma principale, e dal costo complessivo di ciascuna di queste. Chiaramente le prime due quantità sono $\Theta(n)$. La terza invece è determinata dalla somma delle lunghezza delle liste $L(v)$ poiché ogni chiamata *Profondita'*(v) esegue un numero costante di operazioni per ogni elemento di $L(v)$; essendo tale somma pari a due volte il numero dei lati, otteniamo un costo $\Theta(m)$. Osserviamo che per grafi sparsi, cioè con un piccolo numero di lati, nei quali possiamo assumere $m = O(n)$, il tempo di calcolo risulta lineare rispetto al numero di nodi.

Per quanto riguarda lo spazio di memoria osserviamo che, oltre allo spazio $O(n + m)$ necessario per mantenere il grafo di input, occorre riservare un certo numero di celle per mantenere la pila che implementa la ricorsione. Quest'ultima, nel caso peggiore, può raggiungere una lunghezza proporzionale al numero di nodi e quindi una quantità $O(n)$.

Descriviamo ora la versione iterativa dell'algoritmo precedente, nella quale compare esplicitamente la pila S che implementa la ricorsione e che di fatto mantiene (nell'ordine appropriato) i nodi già visitati ma i cui vertici adiacenti non sono ancora stati tutti considerati. Il programma principale è del tutto simile al precedente e si ottiene semplicemente sostituendo la chiamata alla procedura *Profondita'*(v) con quella alla nuova procedura che chiameremo *Profondita'_it*(v).

```
Procedure Profondita'_it(v)
begin
        visita il nodo v
        marca v come "vecchio"
        S := PUSH(Λ, v)
        u := v
        repeat
                while L(u) ≠ Λ do
                        begin
                                w := TESTA(L(u))
                                L(u) := TOGLI_IN_TESTA(L(u))
                                if w e' marcato "nuovo" then
                                begin
                                        visita il nodo w
```

```
                                    marca w come "vecchio"
                                    U := INSERISCI_IN_TESTA({u, w}, U)
                                    S := PUSH(S, w)
                                    u := w
                            end
                    end
            S := POP(S)
            if S ≠ Λ    then u := TOP(S)
    until S = Λ
end
```

Esercizi

1) Definire un algoritmo per l'attraversamento in profondità dei grafi orientati. In quale caso l'algoritmo produce un albero di copertura?

2) Definire una versione iterativa dell'algoritmo di attraversamento in profondità che tenga conto della ricorsione terminale. Per quali input (grafi di n nodi) lo spazio occupato dalla pila è $O(1)$?

Capitolo 6

Equazioni di ricorrenza

Molti classici algoritmi possono essere descritti mediante procedure ricorsive. Di conseguenza l'analisi dei relativi tempi di calcolo è ridotta alla soluzione di una o più equazioni di ricorrenza nelle quali si esprime il termine n-esimo di una sequenza in funzione dei precedenti. Questo capitolo e il successivo sono dedicati alla presentazione delle principali tecniche utilizzate per risolvere equazioni di questo tipo o almeno per ottenere una soluzione approssimata.

6.1 Analisi di procedure ricorsive

Supponiamo di dover analizzare dal punto di vista della complessità un algoritmo definito mediante un insieme di procedure $P_1, P_2, \cdots, P_m$, che si richiamano ricorsivamente fra loro. L'obiettivo dell'analisi è quello di stimare, per ogni $i = 1, 2, \ldots, m$, la funzione $T_i(n)$ che rappresenta il tempo di calcolo impiegato dalla i-ma procedura su dati di dimensione n. Se ogni procedura richiama le altre su dati di dimensione minore, sarà possibile esprimere $T_i(n)$ come funzione dei valori $T_j(k)$ tali che $j \in \{1, 2, \ldots, m\}$ e $k < n$.

Per fissare le idee, supponiamo di avere una sola procedura P che chiama se stessa su dati di dimensione minore. Sia $T(n)$ il tempo di calcolo richiesto da P su dati di dimensione n (nell'ipotesi "caso peggiore" oppure nell'ipotesi "caso medio"). Sarà in generale possibile determinare opportune funzioni $f_1, f_2, \ldots, f_n, \ldots$, in $1, 2, \ldots, n, \ldots$ variabili rispettivamente, tali che:

$$T(n) = f_n(T(n-1), \cdots, T(2), T(1), T(0)) \qquad (n > 1) \tag{1}$$

o almeno tale che

$$T(n) \leq f_n(T(n-1), \cdots, T(2), T(1), T(0)) \tag{2}$$

Relazioni del precedente tipo sono dette *relazioni di ricorrenza* e in particolare quelle di tipo (1) sono dette *equazioni di ricorrenza*. Si osservi che data la condizione al contorno $T(0) = a$, esiste un'unica funzione $T(n)$ che soddisfa (1).

L'analisi di un algoritmo ricorsivo prevede quindi due fasi:

1. Deduzione di relazioni di ricorrenza contenenti come incognita la funzione $T(n)$ da stimare.
2. Soluzione delle relazioni di ricorsività stesse.

Consideriamo per esempio il problema di valutare il tempo di calcolo delle seguenti procedure assumendo il criterio di costo uniforme:

```
Procedure B(n)
begin
      S := 0;
      for i = 0, 1, ..., n do S := S + A(i);
      return S;
end

Procedure A(n)
if n = 0 then return 0 ;
         else { u := n − 1;
                b := n + A(u);
                return b ;
```

Osserviamo innanzitutto che la procedura B richiama A, mentre A richiama se stessa. Per semplicità, assumiamo uguale a c il tempo di esecuzione di ogni istruzione ad alto livello e denotiamo con $T_B(n)$ e $T_A(n)$ rispettivamente il tempo di calcolo dell'esecuzione di B e A su input n. Allora si ottengono le seguenti equazioni:

$$T_B(n) = c + \sum_{i=0}^{n}(c + T_A(i)) + c$$

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

Il problema di analisi è allora ridotto alla soluzione dell'equazione di ricorrenza relativa ai valori $T_A(n)$, $n \in \mathbf{N}$.

Lo sviluppo di tecniche per poter risolvere equazioni o relazioni di ricorrenza è quindi un importante e preliminare strumento per l'analisi di algoritmi e le prossime sezioni sono dedicate alla presentazione dei principali metodi utilizzati.

Esercizio
Scrivere le equazioni di ricorrenza dei tempi di calcolo delle procedure A e B definite sopra assumendo il criterio di costo logaritmico.

6.2 Maggiorazioni

Cominciamo presentando una semplice tecnica per affrontare il seguente problema: data una relazione di ricorrenza

$$\begin{cases} T(n) \leq f_n(T(n-1), \cdots, T(2), T(1), T(0)) & (n \geq 1) \\ T(0) = a \end{cases}$$

e data una funzione $g : \mathbf{N} \to \mathbf{R}^+$, decidere se $T(n) \leq g(n)$ per ogni $n \in \mathbf{N}$.

Una parziale risposta può essere ottenuta dalla seguente proprietà.

Proposizione 6.1 *Consideriamo una funzione $T(n)$ che soddisfi la seguente relazione:*

$$\begin{cases} T(n) \leq f_n(T(n-1), \cdots, T(2), T(1), T(0)) & (n \geq 1) \\ T(0) = a \end{cases}$$

dove, per ogni $n \geq 1$, la funzione $f_n(x_1, x_2, \ldots, x_n)$ sia monotona non decrescente in ogni variabile. Supponiamo inoltre che, per una opportuna funzione $g : \mathbb{N} \to \mathbb{R}^+$, sia $f_n(g(n-1), \ldots, g(0)) \leq g(n)$ per ogni $n \geq 1$ e $g(0) = a$. Allora $T(n) \leq g(n)$.

Dimostrazione. Ragioniamo per induzione su $n \in \mathbb{N}$. Per $n = 0$ la proprietà è verificata per ipotesi. Supponiamo che $g(k) \geq T(k)$ per $k < n$, dimostriamo che $g(n) \geq T(n)$. Infatti, a causa della monotonia di f_n:

$$T(n) \leq f_n(T(n-1), \cdots, T(0)) \leq f_n(g(n-1), \cdots, g(0)) \leq g(n)$$

∎

Osserviamo che se $T(n)$ è una funzione che verifica la relazione di ricorrenza definita nella proposizione precedente allora, per ogni $n \in \mathbb{N}$, $T(n) \leq X(n)$, dove $X(n)$ è definita dall'equazione di ricorrenza associata:

$$X(n) = \begin{cases} a & \text{se } n = 0 \\ f_n(X(n-1), ..., X(0)) & \text{se } n \geq 1 \end{cases}$$

In questo modo possiamo ridurre lo studio delle relazioni di ricorrenza a quello delle equazioni di ricorrenza.

Un'ulteriore applicazione della proposizione 6.1 è data dal seguente corollario che sarà utilizzato nell'analisi di procedure di calcolo delle mediane (sez. 8.6).

Corollario 6.2 *Date due costanti α e β, tali che $0 < \alpha + \beta < 1$, sia $T(n)$ una funzione che soddisfa la relazione*

$$\begin{cases} T(n) \leq T(\lfloor \alpha n \rfloor) + T(\lfloor \beta n \rfloor) + n & \textit{se } n \geq 1 \\ T(0) = 0 & \textit{se } n = 0 \end{cases}$$

Allora esiste una costante c tale che $T(n) \leq c \cdot n$.

Dimostrazione. Infatti $c \cdot 0 = 0$ per ogni c; quindi, per applicare la proposizione precedente, basta determinare una costante c tale che

$$c \cdot (\lfloor \alpha \cdot n \rfloor) + c \cdot (\lfloor \beta \cdot n \rfloor) + n \leq c \cdot n \qquad \text{per ogni } n \geq 1$$

Questa disuguaglianza è verificata se

$$c \geq \frac{1}{1 - \alpha - \beta}$$

Nota che in questo modo abbiamo provato che $T(n) = O(n)$. ∎

6.3 Metodo dei fattori sommanti

Con questa sezione iniziamo lo studio delle equazioni di ricorrenza più comuni nell'analisi degli algoritmi. In generale il nostro obiettivo è quello di ottenere una valutazione asintotica della soluzione oppure, più semplicemente, una stima dell'ordine di grandezza. Tuttavia sono stati sviluppati in letteratura vari metodi che permettono di ricavare la soluzione esatta di una ricorrenza. In alcuni casi poi l'equazione di ricorrenza è particolarmente semplice e si può ottenere la soluzione esatta iterando direttamente l'uguaglianza considerata.

Consideriamo per esempio la seguente equazione di ricorrenza:

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

Poiché , per ogni $n \geq 2$, $T(n-1) = T(n-2) + 2(n-1)$, sostituendo questa espressione nell'equazione precedente si ricava:

$$T(n) = 2n + 2(n-1) + T(n-2).$$

Iterando questa sostituzione per $T(n-2)$, per $T(n-3)$ e così via, si ricava

$$\begin{aligned} T(n) &= 2n + 2(n-1) + \ldots + 2(1) + T(0) = \\ &= 2\sum_{i=1}^{n} i = n(n+1) \end{aligned}$$

Esempio 6.1
Vogliamo determinare il numero esatto di confronti eseguito dalla procedura di ricerca binaria descritta nella sezione 5.2.1. Per semplicità supponiamo che un confronto tra due numeri possa fornire tre risultati a seconda se i due elementi siano uguali, il primo minore del secondo, oppure viceversa il secondo minore del primo. Sia n la dimensione del vettore su cui si svolge la ricerca e sia $T(n)$ il numero di confronti eseguiti nel caso peggiore. Quest'ultimo si verifica quando la procedura esegue un confronto e richiama se stessa su un sottovettore di lunghezza $\lfloor \frac{n}{2} \rfloor$ e questo evento si ripete per tutte le chiamate successive. Allora $T(n)$ soddisfa la seguente equazione:

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

Ricordando le proprietà delle parti intere definite nel capitolo 2 e applicando il procedimento iterativo descritto sopra, per ogni intero n tale che $2^k \leq n < 2^{k+1}$, si ricava

$$T(n) \;=\; 1 + T(\lfloor n/2 \rfloor) \;=\; 2 + T(\lfloor n/2^2 \rfloor) \;=\; \ldots \;=\; k + T(\lfloor n/2^k \rfloor) \;=\; k+1$$

Poiché per definizione $k = \lfloor \log_2 n \rfloor$ si ottiene $T(n) = \lfloor \log_2 n \rfloor + 1$. ∎

Esempio 6.2
Sia

$$T(n) = \begin{cases} 1 & \text{se } n = 0 \\ \frac{n+k-1}{n}\, T(n-1) & \text{se } n \geq 1 \end{cases}$$

Sviluppando l'equazione secondo il metodo descritto otteniamo

$$\begin{aligned} T(n) &= \frac{n+k-1}{n}\, T(n-1) \;=\; \frac{n+k-1}{n}\, \frac{n+k-2}{n-1}\, T(n-2) \;=\; \ldots \;= \\ &= \frac{n+k-1}{n}\, \frac{n+k-2}{n-1} \;\ldots\; \frac{k}{1}\, T(0). \end{aligned}$$

Quindi la soluzione ottenuta è

$$T(n) = \binom{n+k-1}{n}$$

∎

Negli esempi precedenti abbiamo di fatto applicato un metodo generale per risolvere una classe di equazioni (lineari del primo ordine) chiamato *metodo dei fattori sommanti*. Questo può essere descritto in generale nel modo seguente. Date due sequenze $\{a_n\}$ e $\{b_n\}$, consideriamo l'equazione

$$T(n) = \begin{cases} b_0 & \text{se } n = 0 \\ a_n T(n-1) + b_n & \text{se } n \geq 1 \end{cases}$$

Sviluppando l'equazione otteniamo

$$\begin{aligned} T(n) &= b_n + a_n(b_{n-1} + a_{n-1}T(n-2)) = \\ &= b_n + a_n b_{n-1} + a_n a_{n-1}(b_{n-2} + a_{n-2}T(n-3)) \;=\; \ldots \;= \\ &= b_n + a_n b_{n-1} + a_n a_{n-1} b_{n-2} + \cdots + (\prod_{j=2}^{n} a_j)b_1 + (\prod_{j=1}^{n} a_j)b_0 \;= \\ &= b_n + \sum_{i=0}^{n-1} \left(b_i \prod_{j=i+1}^{n} a_j \right). \end{aligned}$$

L'ultima uguaglianza fornisce allora l'espressione esplicita della soluzione.

Esercizi

1) Ricordando l'Esempio 2.3 determinare la soluzione dell'equazione

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

2) Sia $T(n)$ il numero di nodi di un albero binario completo di altezza $n \in \mathbb{N}$. Esprimere $T(n)$ mediante una equazione di ricorrenza e risolverla applicando il metodo illustrato.

6.4 Equazioni "divide et impera"

Un'importante classe di equazioni di ricorrenza è legata all'analisi di algoritmi del tipo "divide et impera" trattati nel capitolo 10. Ricordiamo che un algoritmo di questo tipo suddivide il generico input di dimensione n in un certo numero (m) di sottoistanze del medesimo problema, ciascuna di dimensione n/a (circa) per qualche $a > 1$; quindi richiama ricorsivamente se stesso su tali istanze ridotte e poi ricompone i risultati parziali ottenuti per determinare la soluzione cercata.

Il tempo di calcolo di un algoritmo di questo tipo è quindi soluzione di una equazione di ricorrenza della forma

$$T(n) = mT\left(\frac{n}{a}\right) + g(n)$$

dove $g(n)$ è il tempo necessario per ricomporre i risultati parziali in un'unica soluzione. Questa sezione è dedicata alla soluzione di equazioni di ricorrenza di questa forma per le funzioni $g(n)$ più comuni.

Teorema 6.3 *Siano m, a, b e c numeri reali positivi e supponiamo $a > 1$. Per ogni n potenza di a, sia $T(n)$ definita dalla seguente equazione:*

$$T(n) = \begin{cases} b & \textit{se } n = 1 \\ mT(\frac{n}{a}) + bn^c & \textit{se } n > 1. \end{cases}$$

Allora $T(n)$ soddisfa le seguenti relazioni:

$$T(n) = \begin{cases} \Theta(n^c) & \textit{se } m < a^c \\ \Theta(n^c \log n) & \textit{se } m = a^c \\ \Theta(n^{\log_a m}) & \textit{se } m > a^c \end{cases}$$

Dimostrazione. Sia $n = a^k$ per un opportuno $k \in \mathbb{N}$. Allora, sviluppando l'equazione di ricorrenza, otteniamo

$$\begin{aligned} T(n) &= bn^c + mT(\frac{n}{a}) = \\ &= bn^c + mb\frac{n^c}{a^c} + m^2T(\frac{n}{a^2}) = \\ &= bn + bn^c\frac{m}{a^c} + bn^c\frac{m^2}{a^{2c}} + m^3T(\frac{n}{a^3}) = \cdots = \\ &= bn^c(1 + \frac{m}{a^c} + \frac{m^2}{a^{2c}} + \cdots + \frac{m^{k-1}}{a^{(k-1)c}}) + m^kT(1) = \\ &= bn^c \sum_{j=0}^{\log_a n} (\frac{m}{a^c})^j \end{aligned}$$

Chiaramente se $m < a^c$ la serie $\sum_{j=0}^{+\infty}(\frac{m}{a^c})^j$ è convergente e quindi $T(n) = \Theta(n^c)$.
Se invece $m = a^c$, la sommatoria precedente si riduce a $\log_a n + 1$ e quindi $T(n) = \Theta(n^c \log n)$.
Se infine $m > a^c$, abbiamo

$$\sum_{j=0}^{\log_a n} \left(\frac{m}{a^c}\right)^j = \frac{(\frac{m}{a^c})^{\log_a n+1} - 1}{\frac{m}{a^c} - 1} = \Theta(n^{\log_a m - c})$$

e quindi otteniamo $T(n) = \Theta(n^{\log_a m})$. ∎

Notiamo che in questa dimostrazione non abbiamo sfruttato l'ipotesi che n sia un numero intero. Il risultato vale quindi per funzioni di una variabile reale n purchè definita sulle potenze di a.

L'espressione asintotica di $T(n)$ ottenuta nel teorema precedente e il successivo termine O grande possono essere facilmente calcolati considerando il valore esatto della sommatoria

$$\sum_{j=0}^{\log_a n} \left(\frac{m}{a^c}\right)^j$$

nella dimostrazione appena presentata.

6.4.1 Parti intere

Nel teorema precedente abbiamo risolto le equazioni di ricorrenza solo per valori di n potenza di qualche $a > 1$. Vogliamo ora stimare le soluzioni per un intero n qualsiasi. In questo caso nelle equazioni di ricorrenza compaiono le espressioni $\lceil \cdots \rceil$ e $\lfloor \cdots \rfloor$ che denotano le parti intere dei numeri reali, definite nel capitolo 2.

Per esempio, consideriamo l'algoritmo Mergesort descritto nella sezione 10.3. Tale algoritmo ordina un vettore di n elementi spezzando il vettore in due parti di dimensione $\lfloor n/2 \rfloor$ e $\lceil n/2 \rceil$ rispettivamente; quindi richiama se stesso sui due sottovettori e poi "immerge" le due soluzioni. Così, si può verificare che il numero $M(n)$ di confronti eseguiti per ordinare n elementi, dove n è un qualsiasi intero positivo, soddisfa la seguente ricorrenza:

$$M(n) = \begin{cases} 0 & \text{se } n = 1 \\ M(\lfloor \frac{n}{2} \rfloor) + M(\lceil \frac{n}{2} \rceil) + n - 1 & \text{se } n > 1. \end{cases}$$

In generale le equazioni del tipo "divide et impera" nelle quali compaiono le parti intere possono essere trattate usando il seguente risultato che estende l'analoga valutazione asintotica ottenuta in precedenza.

Teorema 6.4 *Siano a, b e c numeri reali positivi e supponiamo $a > 1$. Consideriamo inoltre due interi $m_1, m_2 \in \mathbb{N}$ tali che $m_1 + m_2 > 0$ e, per ogni intero $n > 0$, definiamo $T(n)$ mediante la seguente equazione:*

$$T(n) = \begin{cases} b & \text{se } n = 1 \\ m_1 T(\lfloor \frac{n}{a} \rfloor) + m_2 T(\lceil \frac{n}{a} \rceil) + bn^c & \text{se } n > 1. \end{cases}$$

Allora, posto $m = m_1 + m_2$, $T(n)$ soddisfa le seguenti relazioni:

$$T(n) = \begin{cases} \Theta(n^c) & \text{se } m < a^c \\ \Theta(n^c \log n) & \text{se } m = a^c \\ \Theta(n^{\log_a m}) & \text{se } m > a^c \end{cases}$$

Dimostrazione. Si prova il risultato nel caso $m_1 = 0$, $m_2 = m > 0$ ottenendo così un limite superiore al valore di $T(n)$. (Nello stesso modo si dimostra il risultato nel caso $m_2 = 0$, $m_1 = m > 0$ ottenendo un limite inferiore.) Supponi che $m_1 = 0$ e sia $k = \lfloor \log_a n \rfloor$. Chiaramente $a^k \leq n < a^{k+1}$. Poiché $\{T(n)\}$ è una sequenza monotona non decrescente, sappiamo che $T(a^k) \leq T(n) \leq T(a^{k+1})$. I valori di $T(a^k)$ e di $T(a^{k+1})$ possono essere valutati usando il teorema 6.3: nel caso $m < a^c$ esistono due constanti c_1 e c_2 tali che

$$T(a^k) \geq c_1 a^{kc} + o(a^{kc}) \geq \frac{c_1}{a^c} n^c + o(n^c),$$

$$T(a^{k+1}) \leq c_2 a^{(k+1)c} + o(a^{(k+1)c}) \leq c_2 a^c n^c + o(n^c).$$

Sostituendo i valori ottenuti nella diseguaglianza precedente si deduce $T(n) = \Theta(n^c)$. I casi $m = a^c$ e $m > a^c$ si trattano in maniera analoga. ∎

Esercizi

1) Considera la sequenza $\{A(n)\}$ definita da

$$A(n) = \begin{cases} 1 & \text{se } n = 1 \\ 3A(\lceil \frac{n}{2} \rceil) + n - 1 & \text{se } n > 1. \end{cases}$$

Calcolare il valore di $A(n)$ per ogni intero n potenza di 2. Determinare l'ordine di grandezza $A(n)$ per n tendente a $+\infty$.

2) Considera la sequenza $\{B(n)\}$ definita dall'equazione

$$B(n) = \begin{cases} 1 & \text{se } n = 1 \\ 2B(\lfloor \frac{n}{2} \rfloor) + n - \sqrt{n} & \text{se } n > 1. \end{cases}$$

Determinare il valore di $B(n)$ per ogni n potenza di 2. Stimare l'ordine di grandezza di $B(n)$ al crescere di n.

3) Siano m, a numeri reali positivi e supponiamo $a > 1$. Per ogni $x \in \mathbb{R}$, definiamo $C(x)$ mediante la seguente equazione:

$$C(x) = \begin{cases} 0 & \text{se } x \leq 1 \\ mC(\frac{x}{a}) + x^2 - x & \text{se } x > 1. \end{cases}$$

Determinare, al variare delle costanti a e m, l'ordine di grandezza di $C(x)$ per x tendente a $+\infty$.

4) Sia $\{D(n)\}$ una sequenza di interi definita dall'equazione

$$D(n) = \begin{cases} 1 & \text{se } n \leq 1 \\ 2D(\lfloor \frac{n}{2} \rfloor) + n \log n & \text{se } n > 1. \end{cases}$$

Determinare, l'ordine di grandezza di $D(n)$ al crescere di n a $+\infty$.

6.5 Equazioni lineari a coefficienti costanti

Un'altra famiglia di equazioni di ricorrenza che compaiono sovente nell'analisi degli algoritmi è quella delle equazioni lineari a coefficienti costanti. Queste sono definite da uguaglianze della forma

$$a_0 t_n + a_1 t_{n-1} + \cdots + a_k t_{n-k} = g_n \tag{6.1}$$

dove $\{t_n\}$ è la sequenza incognita (che per semplicità sostituiamo alla funzione $T(n)$), $k, a_0, a_1, \ldots, a_k$ sono costanti e $\{g_n\}$ è una qualunque sequenza di numeri. Una sequenza di numeri $\{t_n\}$ si dice soluzione dell'equazione se la (6.1) è soddisfatta per ogni $n \geq k$. Chiaramente le varie soluzioni si differenziano per il valore dei primi k termini.

Un'equazione di questo tipo si dice *omogenea* se $g_n = 0$ per ogni $n \in \mathbb{N}$. In questo caso esiste una nota regola generale che permette di ottenere esplicitamente tutte le soluzione dell'equazione. Inoltre, anche nel caso non omogeneo, per le sequenze $\{g_n\}$ più comuni, è possibile definire un metodo per calcolare la famiglia delle soluzioni.

6.5.1 Equazioni omogenee

Per illustrare la regola di soluzione nel caso omogeneo consideriamo un esempio specifico dato dalla seguente equazione:

$$t_n - 7t_{n-1} + 10t_{n-2} = 0 \tag{6.2}$$

Cerchiamo innanzitutto soluzioni della forma $t_n = r^n$ per qualche costante r. Sostituendo tali valori l'equazione diventa

$$r^{n-2}(r^2 - 7r + 10) = 0$$

che risulta verificata per le radici del polinomio $r^2 - 7r + 10$, ovvero per $r = 5$ e $r = 2$. L'equazione $r^2 - 7r + 10 = 0$ è chiamata *equazione caratteristica* della ricorrenza 6.2. Ne segue allora che le sequenze $\{5^n\}$ e $\{2^n\}$ sono soluzioni di (6.2) e quindi, come è facile verificare, lo è anche la combinazione lineare $\{\lambda 5^n + \mu 2^n\}$ per ogni coppia di costanti λ e μ.

Mostriamo ora che ogni soluzione non nulla $\{c_n\}$ della (6.2) è combinazione lineare di $\{5^n\}$ e $\{2^n\}$. Infatti, per ogni $n \geq 2$, possiamo considerare il sistema di equazioni lineari

$$\begin{aligned} 5^{n-1}x + 2^{n-1}y &= c_{n-1} \\ 5^{n-2}x + 2^{n-2}y &= c_{n-2} \end{aligned}$$

nelle incognite x, y. Questo ammette un'unica soluzione $x = \lambda$, $y = \mu$ poiché il determinante dei coefficienti è diverso da 0. Otteniamo così espressioni di c_{n-1} e c_{n-2} in funzione di 5^n e 2^n. Di conseguenza, sostituendo queste ultime in $c_n - 7c_{n-1} + 10c_{n-2} = 0$ e ricordando che anche $\{5^n\}$ e $\{2^n\}$ sono soluzioni di (6.2), si ottiene

$$c_n = \lambda 5^n + \mu 2^n.$$

È facile verificare che i valori di λ e μ ottenuti non dipendono da n e possono essere ricavati considerando il sistema per $n = 2$. Così la relazione precedente è valida per tutti gli $n \in \mathbb{N}$.

Come abbiamo visto, le soluzioni dell'equazione di ricorrenza considerata sono ottenute mediante le radici dell'equazione caratteristica associata. Se le radici sono tutte distinte questa proprietà è del tutto

generale e può essere estesa a ogni equazione di ricorrenza lineare omogenea a coefficienti costanti, cioè ad ogni equazione della forma

$$a_0 t_n + a_1 t_{n-1} + \cdots + a_k t_{n-k} = 0, \tag{6.3}$$

dove $k, a_0, a_1, \ldots, a_k$ sono costanti. Chiaramente, *l'equazione caratteristica* della relazione (6.3) è

$$a_0 x^n + a_1 x^{n-1} + \cdots + a_k = 0.$$

Teorema 6.5 *Sia*

$$a_0 t_n + a_1 t_{n-1} + \cdots + a_k t_{n-k} = 0$$

un'equazione di ricorrenza lineare omogenea a coefficienti costanti e supponiamo che la sua equazione caratteristica abbia k radici distinte $r_1, r_2, \ldots, r_k$. Allora le soluzioni dell'equazione data sono tutte e sole le sequenze $\{t_n\}$ tali che, per ogni $n \in \mathbb{N}$,

$$t_n = \lambda_1 r_1^n + \lambda_2 r_2^n + \cdots + \lambda_k r_k^n$$

dove $\lambda_1, \lambda_2, \ldots, \lambda_k$ sono costanti arbitrarie.

Dimostrazione. Il teorema può essere dimostrato applicando lo stesso ragionamento presentato nell'esempio precedente. Si mostra innanzitutto che l'insieme delle soluzioni dell'equazione forma uno spazio vettoriale di dimensione k e poi si prova che le k soluzioni $\{r_1^n\}, \{r_2^n\}, \ldots, \{r_k^n\}$ sono linearmente indipendenti e formano quindi una base dello spazio stesso. ∎

Esempio 6.3 Numeri di Fibonacci
Considera la sequenza $\{f_n\}$ dei numeri di Fibonacci, definita dall'equazione

$$f_n = \begin{cases} 0 & \text{se } n = 0 \\ 1 & \text{se } n = 1 \\ f_{n-1} + f_{n-2} & \text{se } n \geq 2 \end{cases}$$

La corrispondente equazione caratteristica è $x^2 - x - 1 = 0$ che ha per radici i valori

$$\phi = \frac{1+\sqrt{5}}{2},\ \overline{\phi} = \frac{1-\sqrt{5}}{2}$$

Allora ogni soluzione $\{c_n\}$ dell'equazione considerata è della forma $c_n = \lambda\phi^n + \mu\overline{\phi}^n$, con λ e μ costanti. Imponendo le condizioni iniziali $c_0 = 0, c_1 = 1$, otteniamo il sistema

$$\begin{array}{ll} \lambda + \mu & = 0 \\ \frac{1+\sqrt{5}}{2}\,\lambda + \frac{1-\sqrt{5}}{2}\,\mu & = 1 \end{array}$$

che fornisce le soluzioni $\lambda = \frac{1}{\sqrt{5}}$, $\mu = -\frac{1}{\sqrt{5}}$.
Quindi, per ogni $n \in \mathbb{N}$, otteniamo

$$f_n = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n.$$

∎

Esempio 6.4
Vogliamo calcolare ora la sequenza $\{g_n\}$ definita dalla seguente equazione:

$$g_n = \begin{cases} n & \text{se } 0 \leq n \leq 2 \\ 3g_{n-1} + 4g_{n-2} - 12g_{n-3} & \text{se } n \geq 3 \end{cases}$$

L'equazione caratteristica della ricorrenza è $x^3 - 3x^2 - 4x + 12 = 0$ che ammette come radici i valori $3, -2, 2$.
Ne segue che g_n è della forma $g_n = \lambda 3^n + \mu(-2)^n + \nu 2^n$. Imponendo le condizioni iniziali otteniamo il sistema

$$\begin{array}{ll} \lambda + \mu + \nu & = 0 \\ 3\lambda - 2\mu + 2\nu & = 1 \\ 9\lambda + 4\mu + 4\nu & = 2 \end{array}$$

che fornisce la soluzione $\lambda = \frac{2}{5}$, $\mu = -\frac{3}{20}$, $\nu = -\frac{1}{4}$.
Quindi la sequenza cercata è data dai valori

$$g_n = \frac{2}{5}3^n - \frac{3}{20}(-2)^n - \frac{1}{4}2^n$$

∎

Finora abbiamo considerato solo ricorrenze la cui equazione caratteristica ammette radici semplici. La situazione è solo leggermente più complicata quando compaiono radici multiple. Infatti sappiamo che l'insieme delle soluzioni dell'equazione (6.3) forma uno spazio vettoriale di dimensione k e quindi l'unico problema è quello di determinare k soluzioni linearmente indipendenti. Il seguente teorema, di cui omettiamo la dimostrazione, presenta la soluzione generale.

Teorema 6.6 *Data l'equazione di ricorrenza*

$$a_0 t_n + a_1 t_{n-1} + \cdots + a_k t_{n-k} = 0,$$

supponiamo che la corrispondente equazione caratteristica abbia $h(\leq k)$ radici distinte $r_1, r_2, \ldots, r_h$ e che ciascuna r_i abbia molteplicità m_i. Allora le soluzioni dell'equazione di ricorrenza data sono tutte e sole le combinazioni lineari delle sequenze

$$\{n^j r_i^n\}$$

dove $j \in \{0, 1, \ldots, m_i - 1\}$ e $i \in \{1, 2, \ldots, h\}$.

Esempio 6.5
Vogliamo calcolare la sequenza $\{h_n\}$ definita da

$$h_n = \begin{cases} 0 & \text{se } n = 0, 1 \\ 1 & \text{se } n = 2 \\ 7h_{n-1} - 15h_{n-2} + 9h_{n-3} & \text{se } n \geq 3 \end{cases}$$

In questo caso, l'equazione caratteristica è $x^3 - 7x^2 + 15x - 9 = 0$; essa ammette la radice semplice $x = 1$ e la radice $x = 3$ di molteplicità 2.
Allora h_n è della forma $h_n = \lambda n 3^n + \mu 3^n + \nu$. Imponendo le condizioni iniziali otteniamo il sistema

$$\begin{array}{ll} \mu + \nu & = 0 \\ 3\lambda + 3\mu + \nu & = 0 \\ 18\lambda + 9\mu + \nu & = 1 \end{array}$$

che fornisce la soluzione $\lambda = \frac{1}{6}$, $\mu = -\frac{1}{4}$, $\nu = \frac{1}{4}$.
Quindi la sequenza cercata è data dai valori

$$h_n = \frac{n3^n}{6} - \frac{3^n}{4} + \frac{1}{4}$$

∎

6.5.2 Equazioni non omogenee

Consideriamo ora un'equazione di ricorrenza lineare non omogenea a coefficienti costanti, cioè una relazione della forma

$$a_0 t_n + a_1 t_{n-1} + \cdots + a_k t_{n-k} = g_n \tag{6.4}$$

dove $\{t_n\}$ è la sequenza incognita, $k, a_0, a_1, \ldots, a_k$ sono costanti e $\{g_n\}$ è una qualunque sequenza diversa da quella identicamente nulla. Siano $\{u_n\}$ e $\{v_n\}$ due soluzioni della (6.4). Questo significa che, per ogni $n \geq k$,

$$\begin{aligned} a_0 u_n + a_1 u_{n-1} + \cdots + a_k u_{n-k} &= g_n \\ a_0 v_n + a_1 v_{n-1} + \cdots + a_k v_{n-k} &= g_n \end{aligned}$$

Allora, sottraendo i termini delle due uguaglianze otteniamo

$$a_0(u_n - v_n) + a_1(u_{n-1} - v_{n-1}) + \cdots + a_k(u_{n-k} - v_{n-k}) = 0$$

e quindi la sequenza $\{u_n - v_n\}$ è soluzione dell'equazione omogenea associata alla (6.4).

Viceversa, se $\{u_n\}$ è soluzione di (6.4) e $\{w_n\}$ è soluzione dell'equazione omogenea

$$a_0 t_n + a_1 t_{n-1} + \cdots + a_k t_{n-k} = 0$$

allora anche la loro somma $\{u_n + w_n\}$ è soluzione della (6.4).

Abbiamo così dimostrato che tutte le soluzioni di (6.4) si ottengono sommando una soluzione particolare a tutte le soluzioni dell'equazione omogenea associata. Questo significa che per risolvere la (6.4) possiamo eseguire i seguenti passi:

1. trovare tutte le soluzioni dell'equazione omogenea associata applicando il metodo dell'equazione caratteristica descritto nella sezione precedente;

2. determinare una soluzione particolare dell'equazione data e sommarla alle precedenti.

Il problema è che non esiste un metodo generale per determinare una soluzione particolare di un'equazione non omogenea. Esistono solo tecniche specifiche che dipendono dal valore del termine noto g_n. In alcuni casi tuttavia la determinazione della soluzione particolare è del tutto semplice.

Esempio 6.6
Vogliamo determinare le soluzioni dell'equazione

$$t_n - 2t_{n-1} + 3 = 0$$

L'equazione caratteristica dell'omogenea associata è $x - 2 = 0$ e quindi la sua soluzione generale è $\{\lambda 2^n\}$, con λ costante arbitraria. Inoltre è facile verificare che la sequenza $\{u_n\}$, dove $u_n = 3$ per ogni $n \in \mathbb{N}$, è una soluzione dell'equazione iniziale. Quindi le soluzioni sono tutte e sole le sequenze della forma $\{3 + \lambda 2^n\}$ con λ costante. ∎

Metodo delle costanti indeterminate

Una delle tecniche più comuni per determinare una soluzione particolare di una equazione non omogenea è chiamata *metodo delle costanti indeterminate*. Questo consiste nel sostituire i termini t_n dell'equazione (6.4) con quelli di una sequenza particolare nella quale alcune costanti sono incognite e quindi determinare il valore di queste ultime mediante identificazione. Si può dimostrare che se il termine noto g_n della (6.4) ha la forma

$$g_n = \sum_{i=1}^{h} b_i P_i(n)$$

dove, per ogni $i = 1, 2, \ldots, h$, b_i è una costante e P_i un polinomio in n, allora una soluzione particolare deve essere del tipo

$$u_n = \sum_{i=1}^{h} b_i Q_i(n)$$

dove i $Q_i(n)$ sono polinomi che soddisfano le seguenti proprietà:

1. se b_i non è radice dell'equazione caratteristica dell'omogenea associata a (6.4), allora il grado di $Q_i(n)$ è uguale a quello di di $P_i(n)$;

2. se b_i è radice di molteplicità m_i dell'equazione caratteristica dell'omogenea associata a (6.4), allora il grado di $Q_i(n)$ è la somma di m_i e del grado di $P_i(n)$.

Esempio 6.7
Determiniamo tutte le soluzioni dell'equazione

$$t_n - 3t_{n-1} + t_{n-2} = n + 3^n.$$

L'equazione caratteristica dell'omogenea associata è $x^2 - 3x + 2 = 0$ che ammette le radici $\frac{3+\sqrt{5}}{2}$ e $\frac{3-\sqrt{5}}{2}$. Quindi la soluzione generale dell'equazione omogenea associata è

$$\lambda \left(\frac{3+\sqrt{5}}{2}\right)^n + \mu \left(\frac{3-\sqrt{5}}{2}\right)^n$$

dove λ e μ sono costanti. Cerchiamo ora una soluzione particolare applicando il metodo delle costanti indeterminate. Nel nostro caso $b_1 = 1$, $b_2 = 3$, $Q_1 = n$, $Q_2 = 1$; di conseguenza una soluzione candidata è $u_n = (an + b) + c3^n$, per opportune costanti a, b, c. Per determinare il loro valore sostituiamo u_n nell'equazione e otteniamo

$$u_n - 3u_{n-1} + u_{n-2} = n + 3^n$$

ovvero, svolgendo semplici calcoli,

$$(-a - 1)n + a - b + \left(\frac{c}{9} - 1\right) 3^n = 0$$

che risulta soddisfatta per ogni $n \in \mathbb{N}$ se e solo se

$$a = -1, \;\; b = -1, \;\; c = 9.$$

Quindi una soluzione particolare è $u_n = 3^{n+2} - n - 1$ e di conseguenza le soluzioni dell'equazione iniziale sono

$$\lambda \left(\frac{3+\sqrt{5}}{2}\right)^n + \mu \left(\frac{3-\sqrt{5}}{2}\right)^n + 3^{n+2} - n - 1$$

al variare delle costanti λ e μ. ∎

Esercizi

1) Descrivere un metodo per calcolare una soluzione particolare di un'equazione della forma (6.4) nella quale il termine noto g_n sia costante.

2) Considera la seguente procedura:

```
Procedure Fun(n)
if n ≤ 1 then return n
       else
           begin
                 x = Fun(n − 1)
                 y = Fun(n − 2)
                 return 3x − y
           end
```

Sia $D(n)$ il risultato del calcolo eseguito dalla procedura su input n.

a) Calcolare il valore esatto di $D(n)$ per ogni $n \in \mathbb{N}$.

b) Determinare il numero di operazioni aritmetiche eseguite dalla procedura su input $n \in \mathbb{N}$.

c) Determinare l'ordine di grandezza del tempo di calcolo richiesto dalla procedura su input $n \in \mathbb{N}$ assumendo il criterio di costo logaritmico.

3) Considera la seguente procedura che calcola il valore $B(n) \in \mathbb{N}$ su input $n \in \mathbb{N}, n > 0$.

```
begin
    read n
    a = 2
    for k = 1, ..., n do
        a = 2 + k · a
    output a
end
```

a) Esprimere il valore di $B(n)$ in forma chiusa (mediante una sommatoria).

b) Determinare l'ordine di grandezza del tempo di calcolo e dello spazio di memoria richiesti dalla procedura assumendo il criterio di costo logaritmico.

6.6 Sostituzione di variabile

Molte equazioni che non risultano lineari a coefficienti costanti possono essere ricondotte a tale forma (o a una forma comunque risolubile) mediante semplici sostituzioni.

Un esempio importante è costituito dalle equazioni "divide et impera":

$$T(n) = \begin{cases} b & \text{se } n = 1 \\ mT(\frac{n}{a}) + bn^c & \text{se } n > 1. \end{cases}$$

Sostituendo $n = a^k$ e ponendo $H(k) = T(a^k)$, si ottiene l'equazione

$$H(k) = \begin{cases} b & \text{se } k = 0 \\ mH(k-1) + ba^{kc} & \text{se } k > 0. \end{cases}$$

Questa può essere risolta con la tecnica illustrata nella sezione precedente (oppure con il metodo dei fattori sommanti) ricavando infine $T(n) = H(\log_a n)$.

Esercizio
Dimostrare il teorema 6.3 usando il procedimento appena illustrato.

Un altro esempio di sostituzione è fornito dalla equazione

$$t_n = a(t_{n-1})^b,$$

nella quale a e b sono costanti positive, $b \neq 1$, con la condizione iniziale $t_0 = 1$. In questo caso possiamo porre $\log_a t_n$, ottenendo

$$u_n = b \log_a t_{n-1} + 1 = bu_{n-1} + 1;$$

questa può essere risolta facilmente con il metodo dei fattori sommanti:

$$u_n = \frac{b^n - 1}{b - 1}.$$

Operando di nuovo la sostituzione si ricava

$$t_n = a^{\frac{b^n-1}{b-1}}.$$

Esempio 6.8
Considera l'equazione

$$t_n = t_{n-1}\left(\frac{t_{n-1}}{t_{n-2}} + 1\right)$$

con la condizione iniziale $t_0 = t_1 = 1$. Dividendo per t_{n-1} si ottiene

$$\frac{t_n}{t_{n-1}} = \frac{t_{n-1}}{t_{n-2}} + 1.$$

Quindi operando la sostituzione $v_n = \frac{t_n}{t_{n-1}}$, si ricava l'equazione $v_n = v_{n-1} + 1$ con la condizione iniziale $v_1 = 1$. Chiaramente si ottiene $v_n = n$ e quindi

$$t_n = \prod_{i=1}^{n} v_i = n!.$$

∎

Esercizi

1) Sia $T(n)$ il numero di bit necessari per rappresentare l'intero positivo n. Determinare una equazione di ricorrenza per $T(n)$ e risolverla.

2) Considera la sequenza $\{T(n)\}$ definita da

$$T(n) = \begin{cases} 0 & \text{se } n = 1 \\ 1 + T(\lfloor\sqrt{n}\rfloor) & \text{se } n > 1. \end{cases}$$

Dimostrare che $T(n) = \lfloor \log_2 \log_2 n \rfloor + 1$ per ogni intero $n > 1$.

6.6.1 L'equazione di Quicksort

Come sappiamo Quicksort è uno degli algoritmi più importanti utilizzati per ordinare una sequenza di elementi (vedi la sezione 8.5). L'analisi del suo tempo di calcolo nel caso medio si riduce alla soluzione dell'equazione di ricorrenza

$$t_n = n - 1 + \frac{2}{n}\sum_{k=0}^{n-1} t_k$$

con la condizione iniziale $t_0 = 0$. Si può risolvere tale equazione con un opportuno cambiamento di variabile; il procedimento ha una certa generalità perchè permette di trattare equazioni nelle quali compaiono sommatorie del tipo $\sum_{k=0}^{n} t_k$.

Innanzitutto la ricorrenza può essere scritta nella forma

$$nt_n = n(n-1) + 2\sum_{0}^{n-1} t_k,$$

ovvero, riferendosi al valore n-1 (quindi per ogni $n > 1$),

$$(n-1)t_{n-1} = (n-1)(n-2) + 2\sum_{0}^{n-2} t_k.$$

Sottraendo membro a membro le due uguaglianze ottenute si elimina la sommatoria, ricavando

$$nt_n = (n+1)t_{n-1} + 2(n-1).$$

Nota che quest'ultima relazione vale per ogni $n \geq 1$. Dividendo così per $n(n+1)$ e definendo $u_n = \frac{t_n}{n+1}$, otteniamo

$$u_n = u_{n-1} + 2(\frac{2}{n+1} - \frac{1}{n}).$$

Ora possiamo applicare il metodo dei fattori sommanti ricavando

$$u_n = 2\sum_1^n (\frac{2}{k+1} - \frac{1}{k}) = 2\{\frac{2}{n+1} - 2 + \sum_1^n \frac{1}{k}\};$$

da cui si ottiene l'espressione per t_n:

$$t_n = (n+1)u_n = 2(n+1)\sum_1^n \frac{1}{k} - 4n = 2n\log n + O(n).$$

Esercizi

1) Determina la soluzione dell'equazione

$$t_n = \frac{n}{n+1}t_{n-1} + 1$$

con la condizione iniziale $t_0 = 0$.

2) Considera la sequenza $\{t_n\}$ definita dall'equazione

$$t_n = n(t_{n-1})^2$$

con la condizione iniziale $t_1 = 1$. Dimostrare che $t_n = \Omega(2^{2^n})$ e $t_n = O(n^{2^n})$.

Capitolo 7

Funzioni generatrici

Le funzioni generatrici rappresentano uno strumento classico, introdotto originariamente per risolvere problemi di conteggio in vari settori della matematica, che ha assunto un'importanza particolare nell'analisi degli algoritmi. Si tratta di uno strumento che ha permesso di sviluppare un numero notevole di metodi e tecniche usate sia per determinare la soluzione di equazioni di ricorrenza, sia nello studio delle proprietà delle strutture combinatorie comunemente utilizzate nella progettazione degli algoritmi. Una delle maggiori applicazioni riguarda la possibilità di utilizzare consolidate tecniche analitiche per la determinazione di sviluppi asintotici.

L'idea di fondo che sta alla base di questi metodi è quella di rappresentare una sequenza di numeri mediante una funzione analitica e far corrispondere alle operazioni su sequenze analoghe operazioni tra funzioni. In questo modo, possiamo formulare un problema di enumerazione mediante una o più relazioni definite su funzioni analitiche e, una volta determinata la soluzione in questo ambito, tornare al contesto originario calcolando la o le sequenze associate alle funzioni ottenute. In questo senso le funzioni generatrici possono essere viste come esempio di trasformata. Per motivi storici e soprattutto in testi per applicazioni ingegneristiche le funzioni generatrici sono chiamate anche z-trasformate.

7.1 Definizioni

Consideriamo una sequenza di numeri reali $\{a_0, a_1, \ldots, a_n, \ldots\}$, che nel seguito denoteremo con $\{a_n\}_{n\geq 0}$ oppure, più semplicemente, con $\{a_n\}$. Supponiamo che la serie di potenze

$$\sum_{n=0}^{+\infty} a_n z^n$$

abbia un raggio di convergenza R maggiore di 0 (questo si verifica nella maggior parte dei casi di interesse per l'analisi di algoritmi e in questo capitolo consideriamo solo sequenze che godono di tale proprietà). Chiamiamo allora *funzione generatrice* di $\{a_n\}$ la funzione di variabile reale

$$A(z) = \sum_{n=0}^{+\infty} a_n z^n \tag{7.1}$$

definita sull'intervallo $(-R, R)$ [1].

[1] Nota che in realtà $A(z)$ può essere interpretata come funzione di variabile complessa definita nel cerchio aperto di centro 0 e raggio R.

Viceversa, data una funzione $A(z)$, sviluppabile in serie di potenze con centro nello 0, diremo che $\{a_n\}_{n\geq 0}$ è la sequenza *associata* alla funzione $A(z)$ se l'uguaglianza 7.1 vale per ogni z in un opportuno intorno aperto di 0.

Quest'ultima definizione è ben posta: se $\{a_n\}$ e $\{b_n\}$ sono sequenze distinte, e le serie di potenze

$$\sum_{n=0}^{+\infty} a_n z^n, \quad \sum_{n=0}^{+\infty} b_n z^n$$

hanno raggio di convergenza positivo, allora le corrispondenti funzioni generatrici sono distinte.

Abbiamo così costruito una corrispondenza biunivoca tra la famiglia delle sequenze considerate e l'insieme delle funzioni sviluppabili in serie di potenze con centro in 0. Un esempio particolarmente semplice si verifica quando la sequenza $\{a_n\}$ è definitivamente nulla; in questo caso la funzione generatrice corrispondente è un polinomio. Così per esempio, la funzione generatrice della sequenza $\{1, 0, 0, \ldots, 0, \ldots\}$ è la funzione costante $A(z) = 1$.

In generale, per passare da una funzione generatrice $A(z)$ alla sequenza associata $\{a_n\}$ sarà sufficiente determinare lo sviluppo in serie di Taylor di $A(z)$ con centro in 0:

$$A(z) = A(0) + A'(0)z + \frac{A''(z)}{2}z^2 + \cdots + \frac{A^{(n)}(0)}{n!}z^n + \cdots,$$

dove con $A^{(n)}(0)$ indichiamo la derivata n-esima di $A(z)$ valutata in 0. Ne segue allora che

$$a_n = \frac{A^{(n)}(0)}{n!}$$

per ogni $n \in \mathbb{N}$.

Così, ricordando lo sviluppo in serie di Taylor delle funzioni tradizionali, possiamo determinare immediatamente le sequenze associate in diversi casi particolari:

1. per ogni $m \in \mathbb{N}$, la funzione $(1 + z)^m$ è la funzione generatrice della sequenza $\{\binom{m}{0}, \binom{m}{1}, \ldots, \binom{m}{m}, 0, 0 \ldots, 0 \ldots\}$ poiché

$$(1 + z)^m = \sum_{n=0}^{m} \binom{m}{n} z^n \qquad (\forall z \in \mathbb{R}).$$

2. Per ogni $b \in \mathbb{R}$, la funzione $\frac{1}{1-bz}$ è la funzione generatrice della sequenza $\{b^n\}_{n\geq 0}$ poiché

$$\frac{1}{1 - bz} = \sum_{n=0}^{+\infty} b^n z^n \qquad (|z| < |1/b|).$$

3. La funzione e^z è la funzione generatrice della sequenza $\{\frac{1}{n!}\}_{n\geq 0}$ poiché

$$e^z = \sum_{n=0}^{+\infty} \frac{z^n}{n!} \qquad (\forall z \in \mathbb{R}).$$

4. La funzione $\log \frac{1}{1-z}$ è la funzione generatrice della sequenza $\{0, 1, \frac{1}{2}, \ldots, \frac{1}{n}, \ldots\}$ poiché

$$\log \frac{1}{1 - z} = \sum_{n=1}^{+\infty} \frac{z^n}{n} \qquad (|z| < 1).$$

5. Per ogni $m \in \mathbb{N}$, la funzione $\frac{1}{(1-z)^m}$ è la funzione generatrice della sequenza $\{\binom{m+n-1}{n}\}_{n\geq 0}$ poiché

$$\frac{1}{(1-z)^m} = \sum_{n=0}^{+\infty} \binom{m+n-1}{n} z^n \qquad (|z|<1).$$

6. Generalizzando la nozione di coefficiente binomiale possiamo definire, per ogni numero reale α e ogni $n \in \mathbb{N}$, il coefficiente $\binom{\alpha}{n}$ nel modo seguente:

$$\binom{\alpha}{n} = \begin{cases} 1 & \text{se } n = 0 \\ \frac{\alpha(\alpha-1)\cdots(\alpha-n+1)}{n!} & \text{se } n \geq 1 \end{cases}$$

Così è facile verificare che $(1+z)^\alpha$ è la funzione generatrice di $\{\binom{\alpha}{n}\}_{n\geq 0}$ poiché

$$(1+z)^\alpha = \sum_{n=0}^{+\infty} \binom{\alpha}{n} z^n \qquad (|z|<1)$$

Nota che $\binom{-\alpha}{n} = (-1)^n \binom{n+\alpha-1}{n}$ per ogni $n \in \mathbb{N}$.

Esercizi

1) La sequenza $\{n!\}_{n\geq 0}$ ammette funzione generatrice?
2) Determinare le sequenze associate alle seguenti funzioni generatrici (con $b, \alpha \in \mathbb{R}$, $b, \alpha \neq 0$):

$$\frac{e^{bz}-1}{z}, \quad \frac{1}{z}\log\frac{1}{1-bz}, \quad \frac{(1+bz)^\alpha - 1}{z}$$

3) Dimostrare che la funzione $\frac{1}{\sqrt{1-4z}}$ è funzione generatrice della sequenza $\{\binom{2n}{n}\}_{n\geq 0}$.

7.2 Funzioni generatrici ed equazioni di ricorrenza

Le funzioni generatrici forniscono un metodo generale per determinare o per approssimare le soluzioni di equazioni di ricorrenza. Infatti in molti casi risulta più facile determinare equazioni per calcolare la funzione generatrice di una sequenza $\{a_n\}$ piuttosto che risolvere direttamente equazioni di ricorrenza per $\{a_n\}$. Esistono inoltre consolidate tecniche analitiche che permettono di determinare una stima asintotica di $\{a_n\}$ una volta nota la sua funzione generatrice.

Questi vantaggi suggeriscono il seguente schema generale per risolvere una equazione di ricorrenza di una data sequenza $\{a_n\}$:

1. trasformare la ricorrenza in una equazione tra funzioni generatrici che ha per incognita la funzione generatrice $A(z)$ di $\{a_n\}$;

2. risolvere quest'ultima calcolando una espressione esplicita per $A(z)$;

3. determinare $\{a_n\}$ sviluppando $A(z)$ in serie di Taylor con centro in 0 oppure calcolarne l'espressione asintotica conoscendo i punti di singolarità di $A(z)$.

Per illustrare questo approccio presentiamo un esempio semplice che consente di risolvere un'equazione già considerata nel capitolo precedente. Si tratta dell'equazione descritta nell'esempio (6.3) che definisce i numeri di Fibonacci. Vogliamo calcolare i termini della sequenza $\{f_n\}$ definiti dalla seguente equazione:

$$f_n = \begin{cases} 0 & \text{se } n = 0 \\ 1 & \text{se } n = 1 \\ f_{n-1} + f_{n-2} & \text{se } n \geq 2 \end{cases}$$

In questo caso possiamo calcolare la funzione generatrice $F(z) = \sum_{n=0}^{+\infty} f_n z^n$ con il metodo che segue. Per ogni $n \geq 2$ sappiamo che $f_n = f_{n-1} + f_{n-2}$; quindi moltiplicando a destra e a sinistra per z^n e sommando su tutti gli $n \geq 2$ si ottiene

$$\sum_{n=2}^{+\infty} f_n z^n = \sum_{n=2}^{+\infty} f_{n-1} z^n + \sum_{n=2}^{+\infty} f_{n-2} z^n.$$

Tenendo conto delle condizioni iniziali $f_0 = 0$, $f_1 = 1$, l'equazione può essere scritta nella forma

$$F(z) - z = zF(z) + z^2 F(z),$$

ovvero $F(z) = \frac{z}{1-z-z^2}$. Dobbiamo quindi determinare lo sviluppo in serie di Taylor della funzione

$$\frac{z}{1 - z - z^2}.$$

Per fare questo consideriamo il polinomio $1 - z - z^2$; le sue radici sono $\alpha = \frac{\sqrt{5}-1}{2}$ e $\beta = -\frac{\sqrt{5}+1}{2}$. Calcoliamo ora le costanti reali A e B tali che

$$\frac{A}{1 - \frac{z}{\alpha}} + \frac{B}{1 - \frac{z}{\beta}} = \frac{z}{1 - z - z^2}$$

Questa equazione corrisponde al sistema

$$\begin{aligned} A + B &= 0 \\ A\alpha + B\beta &= -\alpha\beta \end{aligned}$$

che fornisce i valori $A = \frac{1}{\sqrt{5}}$ e $B = -\frac{1}{\sqrt{5}}$ e di conseguenza otteniamo

$$F(z) = \frac{1}{\sqrt{5}} \left\{ \frac{1}{1 - \frac{z}{\alpha}} - \frac{1}{1 - \frac{z}{\beta}} \right\}$$

Ricordando lo sviluppo delle serie geometriche si ricava

$$F(z) = \frac{1}{\sqrt{5}} \left\{ \sum_{n=0}^{+\infty} \frac{z^n}{\alpha^n} - \sum_{n=0}^{+\infty} \frac{z^n}{\beta^n} \right\}$$

e quindi, per ogni $n \geq 0$, abbiamo

$$f_n = \frac{1}{\sqrt{5}} \left\{ \left(\frac{2}{\sqrt{5}-1} \right)^n - \left(-\frac{2}{\sqrt{5}+1} \right)^n \right\} = \frac{1}{\sqrt{5}} \left\{ \left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right\}.$$

Esercizio

Usando le funzioni generatrici, determinare la soluzione delle seguenti equazioni:

$$a_n = \begin{cases} 1 & \text{se } n = 0 \\ 2a_{n-1} + 1 & \text{se } n \geq 1 \end{cases}$$

$$b_n = \begin{cases} 0 & \text{se } n = 0 \\ 1 & \text{se } n = 1 \\ 3b_{n-1} - b_{n-2} & \text{se } n \geq 2 \end{cases}$$

$$c_n = \begin{cases} 0 & \text{se } n = 0 \\ 1 & \text{se } n = 1 \\ 4c_{n-1} - 4c_{n-2} & \text{se } n \geq 2 \end{cases}$$

7.3 Calcolo di funzioni generatrici

L'esempio precedente mostra come l'equazione di ricorrenza che definisce una sequenza $\{f_n\}$ possa essere trasformata in una equazione per la corrispondente funzione generatrice $F(z)$. A tale riguardo presentiamo ora una discussione generale su questa trasformazione, mostrando la corrispondenza che esiste tra operazioni sulle sequenze e corrispondenti operazioni sulle funzioni generatrici. Questa corrispondenza permette in molti casi di trasformare ricorrenze (o più in generale relazioni) tra successioni numeriche in equazioni sulle relative funzioni generatrici.

7.3.1 Operazioni su sequenze numeriche

Definiamo anzitutto alcune operazioni sulle successioni. Siano $\{f_n\}_{n\geq 0}$ e $\{g_n\}_{n\geq 0}$ due sequenze e siano $c \in \mathbb{R}$ e $k \in \mathbb{N}$ due costanti. Allora definiamo:

Moltiplicazione per costante	$c \cdot \{f_n\}$	=	$\{cf_n\}$
Somma	$\{f_n\} + \{g_n\}$	=	$\{f_n + g_n\}$
Convoluzione	$\{f_n\} \otimes \{g_n\}$	=	$\{\sum_{k=0}^{n} f_k g_{n-k}\}$
Spostamento	$E^k\{f_n\}$	=	$\{f_{k+n}\}_{n\geq 0}$
Moltiplicazione per n	$n \cdot \{f_n\}$	=	$\{nf_n\}$
Divisione per $n+1$	$\frac{1}{n+1} \cdot \{f_n\}$	=	$\{\frac{f_n}{n+1}\}_{n\geq 0}$

Osserviamo qui che la somma $\sum_{k=0}^{n} f_k$ è ottenibile mediante la convoluzione:

$$\{1\} \otimes \{f_n\} = \left\{ \sum_{k=0}^{n} f_k \right\}$$

dove $\{1\}$ rappresenta la sequenza $\{b_n\}_{n\geq 0}$ tale che $b_n = 1$ per ogni $n \in \mathbb{N}$.

Una vasta classe di equazioni di ricorrenza è ottenuta applicando le operazioni precedentemente definite.

Esempio 7.1
La sequenza dei numeri di Fibonacci definiti nella sezione precedente

$$f_n = \begin{cases} n & \text{se } n \leq 1 \\ f_{n-1} + f_{n-2} & \text{se } n \geq 2 \end{cases}$$

può essere rappresentata nella forma $f_{n+2} = f_{n+1} + f_n$, ovvero

$$E^2\{f_n\} = E^1\{f_n\} + \{f_n\}$$

insieme alla condizione iniziale $f_n = n$ per $n \leq 1$. ∎

Esempio 7.2
Considera la ricorrenza

$$f_n = \begin{cases} 1 & \text{se } n = 0 \\ \sum_{k=0}^{n-1} f_k f_{n-1-k} & \text{se } n \geq 1 \end{cases}$$

Essa può essere riscritta nella forma

$$E^1\{f_n\} = \{f_n\} \otimes \{f_n\}$$

insieme alla condizione iniziale $f_0 = 1$. ∎

Esempio 7.3
L'equazione di ricorrenza

$$f_n = \begin{cases} 1 & \text{se } n = 0 \\ \frac{1}{n} \sum_{k=0}^{n-1} f_k & \text{se } n \geq 1 \end{cases}$$

può essere riscritta nella forma:

$$n \cdot E^1\{f_n\} + E^1\{f_n\} = \{1\} \otimes \{f_n\}$$

insieme alla condizione iniziale $f_0 = 1$. ∎

7.3.2 Operazioni su funzioni generatrici

Poiché la corrispondenza tra sequenze e loro funzioni generatrici è biunivoca, ad ogni operazione tra successioni corrisponde in linea di principio una precisa operazione tra funzioni generatrici. Descriviamo ora le operazioni su funzioni generatrici corrispondenti alle operazioni su sequenze definite nella sezione precedente.

Denotando con $F(z)$ e $G(z)$ rispettivamente le funzioni generatrici delle sequenze $\{f_n\}_{n\geq 0}$ e $\{g_n\}_{n\geq 0}$, possiamo presentare nella sequente tabella alcune corrispondenze di immediata verifica. Nella prima colonna riportiamo il termine n-esimo della sequenza mentre nella seconda la funzione generatrice corrispondente; inoltre c rappresenta qui una costante reale qualsiasi.

$c \cdot f_n$	$c \cdot F(z)$
$f_n + g_n$	$F(z) + G(z)$
$\sum_{k=0}^{n} f_k g_{n-k}$	$F(z) \cdot G(z)$
f_{k+n}	$\frac{F(z) - f_0 - f_1 z - \cdots - f_{k-1} z^{k-1}}{z^k}$
$n f_n$	$zF'(z)$
$\frac{f_n}{n+1}$	$\frac{1}{z}\int_0^z F(t)dt$

Per quanto riguarda l'uso della derivata e dell'integrale nella tabella precedente ricordiamo che, se $F(z)$ è la funzione generatrice di una sequenza $\{f_n\}$, anche la sua derivata $F'(z)$ è una funzione sviluppabile in serie di potenze con centro in 0; inoltre, tale sviluppo è della forma

$$F'(z) \;=\; \sum_{n=1}^{+\infty} n f_n z^{n-1} \;=\; \sum_{n=0}^{+\infty} (n+1) f_{n+1} z^n$$

Questo significa che $F'(z)$ è la funzione generatrice della sequenza $\{(n+1)f_{n+1}\}_{n\geq 0}$.

Esempio 7.4
Calcoliamo la funzione generatrice di $\{(n+1)\}_{n\geq 0}$. Poiché $\frac{1}{1-z}$ è la funzione generatrice di $\{1\}$, la funzione cercata è semplicemente la derivata di $\frac{1}{1-z}$:

$$\sum_{n=0}^{+\infty} (n+1) z^n \;=\; \frac{d}{dz}\frac{1}{1-z} \;=\; \frac{1}{(1-z)^2}.$$

∎

Un discorso analogo vale per la funzione integrale $I(z) = \int_0^z F(t)dt$: anche $I(z)$ è sviluppabile in serie di potenze con centro in 0 e il suo sviluppo è della forma

$$I(z) \;=\; \int_0^z F(t)dt \;=\; \sum_{n=1}^{+\infty} \frac{f_{n-1}}{n} z^n$$

Di conseguenza $I(z)$ risulta la funzione generatrice della sequenza $\{0, \frac{f_0}{1}, \frac{f_1}{2}, \ldots, \frac{f_{n-1}}{n}, \ldots\}$.

Esempio 7.5
Calcoliamo la funzione generatrice della sequenza $\{0, 1, \frac{1}{2}, \ldots, \frac{1}{n}, \ldots\}$ (che nel seguito denoteremo più semplicemente $\{\frac{1}{n}\}$). Questa può essere ottenuta mediante integrazione della serie geometrica:

$$\sum_{n=1}^{+\infty} \frac{1}{n} z^n \;=\; \int_0^z \frac{1}{1-t} dt \;=\; \log \frac{1}{1-z}.$$

∎

Vediamo ora alcune applicazioni delle corrispondenze riportate nella tabella precedente. Un primo esempio deriva immediatamente dai casi appena presi in esame.

Esempio 7.6
Calcoliamo la funzione generatrice della sequenza $\{H_n\}$ dei numeri armonici. Dalla loro definizione sappiamo che

$$H_n = \sum_{k=1}^{n} \frac{1}{k}$$

e quindi abbiamo $H_n = \{\frac{1}{n}\} \otimes \{1\} = \{1\} \otimes \{\frac{1}{n}\}$. Di conseguenza la funzione generatrice di $\{H_n\}$ è data dal prodotto

$$\frac{1}{1-z} \cdot \log \frac{1}{1-z}.$$

∎

Più in generale, siamo ora in grado di trasformare una equazione di ricorrenza in una equazione tra funzioni generatrici e possiamo quindi applicare compiutamente il metodo descritto nella sezione 7.2. In particolare, possiamo risolvere le equazioni di ricorrenza che compaiono negli Esempi 7.1, 7.2, e 7.3.

Vediamo esplicitamente come si risolve il primo caso (Esempio 7.1). L'equazione

$$f_n = \begin{cases} n & \text{se } n \leq 1 \\ f_{n-1} + f_{n-2} & \text{se } n \geq 2 \end{cases}$$

viene riscritta mediante operatori di shift (spostamento) e condizioni al contorno nella forma:

$$f_{n+2} = f_{n+1} + f_n, \;\; f_0 = 0, f_1 = 1.$$

Applicando ora le regole 2) e 4) della tabella precedente si ottiene

$$\frac{F(z) - z}{z^2} = \frac{F(z)}{z} + F(z).$$

Applicando un ragionamento simile agli altri due esempi si ottengono le seguenti equazioni sulle funzioni generatrici:

$$\frac{F(z) - 1}{z} = F^2(z) \qquad \text{(Esempio 7.2)}$$

$$z \cdot \frac{d}{dz}\left(\frac{F(z) - 1}{z}\right) + \frac{F(z) - 1}{z} = \frac{1}{1-z} \cdot F(z) \qquad \text{(Esempio 7.3)}$$

Di conseguenza, la determinazione di $F(z)$ è qui ridotta rispettivamente alla soluzione di una equazione di primo grado (Esempio 7.1), di secondo grado (Esempio 7.2), di una equazione differenziale lineare del primo ordine (Esempio 7.3). Abbiamo già mostrato nella sezione 7.2 come trattare le equazioni di primo grado; presenteremo nella sezione seguente un'analisi degli altri due casi.

Esercizi

1) Determinare la funzione generatrice delle seguenti sequenze:

$$\{n\}_{n\geq 0},\ \{n-1\}_{n\geq 0},\ \{n^2\}_{n\geq 0},\ \{n2^n\}_{n\geq 0},\ \{\frac{1}{n+2}\}_{n\geq 0},\ \left\{\sum_{k=0}^{n}\frac{k-1}{n-k+1}\right\}_{n\geq 0}.$$

2) Ricordando l'esercizio 3 della sezione 7.1, dimostrare che per ogni $n \in \mathbb{N}$

$$\sum_{k=0}^{n}\binom{2k}{k}\binom{2n-2k}{n-k} = 4^n.$$

3) Determinare la funzione generatrice $F(z)$ della successione f_n dove:

$$f_{n+2} - 2f_{n+1} + f_n = n,\ \ f_0 = f_1 = 0$$

7.4 Applicazioni

Come primo esempio applichiamo i metodi sopra illustrati per determinare la soluzione di equazioni del tipo "divide et impera". Sia $T(n)$ definita da

$$T(n) = \begin{cases} b & \text{se } n = 1 \\ mT\left(\frac{n}{a}\right) + g_n & \text{se } n \geq 2 \end{cases}$$

per $n \in \mathbb{N}$ potenze di a, dove assumiamo $m, b > 0$ e $a > 1$.

Mediante la sostituzione $n = a^k$, posto $f_k = T(a^k)$ e $h_k = g(a^{k+1})$ si ha:

$$f_{k+1} = mf_k + h_k.$$

Denotando quindi con $F(z)$ e $H(z)$ rispettivamente le funzioni generatrici di $\{f_k\}$ e di $\{h_k\}$ si ottiene

$$\frac{F(z) - F(0)}{z} = mF(z) + H(z).$$

Questa è una equazione di primo grado in $F(z)$ che può essere risolta direttamente una volta noti il valore iniziale $F(0) = b$ e la funzione $H(z)$.

Esercizio

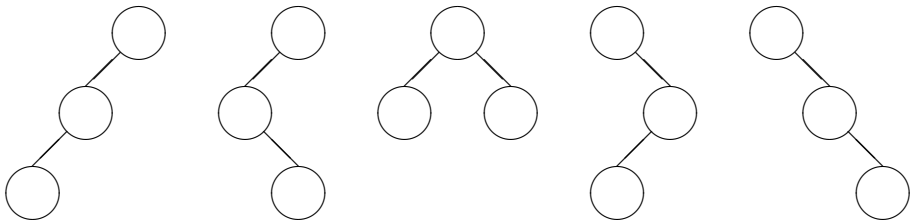
Per tutti gli $n \in \mathbb{N}$ potenze di 2 risolvere l'equazione

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

7.4.1 Conteggio di alberi binari

Un classico problema di enumerazione consiste nel determinare il numero di alberi binari non etichettati di n nodi per $n \in \mathbb{N}$ qualsiasi [2]. Intuitivamente un albero binario non etichettato è un albero binario al quale abbiamo tolto il nome ai nodi; in questo modo i nodi sono indistinguibili e due alberi di questo genere sono diversi solo le corrispondenti rappresentazioni grafiche, private del nome dei nodi, sono distinte. Nella seguente figura rappresentiamo gli alberi binari non etichettati con tre nodi:

[2] Per la definizione di albero binario si veda la sezione 4.6.3



Una definizione induttiva è la seguente: un albero binario non etichettato può essere l'albero vuoto, che denotiamo con ε, oppure è descritto da un nodo chiamato radice e da due alberi binari non etichettati T_1, T_2 (che rappresentano rispettivamente il sottoalbero di sinistra e il sottoalbero di destra).

Denotiamo con b_n il numero di alberi binari non etichettati con n nodi, $n \in \mathbb{N}$. Dalla definizione sappiamo che $b_0 = 1$; inoltre, se $n > 0$, un albero binario con $n+1$ nodi possiede, oltre alla radice, k nodi nel suo sottoalbero di sinistra e $n-k$ nel sottoalbero di destra, per qualche intero k, $0 \leq k \leq n$. Quindi b_n soddisfa la seguente equazione di ricorrenza:

$$b_0 = 1, \;\; b_{n+1} = \sum_{k=0}^{n} b_k b_{n-k}$$

Passando alle funzioni generatrici e denotando con $B(z)$ la funzione generatrice di $\{b_n\}$, l'equazione si traduce in

$$\frac{B(z) - 1}{z} = B^2(z)$$

ovvero

$$B(z) = 1 + z(B(z))^2$$

Risolvendo l'equazione otteniamo le due soluzioni

$$B_1(z) = \frac{1 + \sqrt{1-4z}}{2z}, \quad B_2 = \frac{1 - \sqrt{1-4z}}{2z}$$

La funzione $B_1(z)$ deve essere scartata poiché $\lim_{z\to 0} B_1(z) = \infty$ e quindi B_1 non è sviluppabile in serie di Taylor con centro in 0. Ne segue che B_2 è la funzione generatrice della sequenza $\{b_n\}$ (in particolare si verifica che $\lim_{z\to 0} B_2(z) = 1 = b_0$).

Dobbiamo ora sviluppare $\frac{1-\sqrt{1-4z}}{2z}$ in serie di Taylor con centro in 0. Applicando lo sviluppo di funzioni della forma $(1+z)^\alpha$ riportato nella sezione 7.1, otteniamo

$$\sqrt{1-4z} = \sum_{n=0}^{+\infty} \binom{1/2}{n} (-4)^n z^n$$

dove

$$\binom{1/2}{0} = 1$$

mentre, per ogni $n > 0$, abbiamo

$$\begin{aligned} \binom{1/2}{n} &= \frac{\frac{1}{2}\left(\frac{1}{2}-1\right)\cdots\left(\frac{1}{2}-n+1\right)}{n!} = \\ &= \frac{(-1)^{n-1}}{2^n} \frac{1 \cdot 3 \cdot 5 \cdot \cdots \cdot (2n-3)}{n!} = \end{aligned}$$

$$= \frac{(-1)^{n-1}}{2^n}\frac{(2n-2)!}{n!(2\cdot 4\cdot\cdots\cdot 2n-2)} =$$
$$= \frac{2(-1)^{n-1}}{n4^n}\binom{2n-2}{n-1}$$

Si deduce allora che

$$\frac{1-\sqrt{1-4z}}{2z} = \frac{1-\left(1-\sum_{n=1}^{+\infty}\frac{2}{n}\binom{2n-2}{n-1}z^n\right)}{2z}$$
$$= \sum_{n=1}^{+\infty}\frac{1}{n}\binom{2n-2}{n-1}z^{n-1}$$
$$= \sum_{n=0}^{+\infty}\frac{1}{n+1}\binom{2n}{n}z^n$$

e di conseguenza $b_n = \frac{1}{n+1}\binom{2n}{n}$. Ricordiamo che gli interi $\frac{1}{n+1}\binom{2n}{n}$ sono noti in letteratura come i numeri di Catalan.

7.4.2 Analisi in media di Quicksort

Abbiamo già studiato nella sezione 6.6.1 l'equazione di ricorrenza relativa al tempo di calcolo dell'algoritmo Quicksort nel caso medio. Vediamo ora come risolvere la stessa equazione usando le funzioni generatrici.

L'equazione è data dall'uguaglianza

$$T_n = n - 1 + \frac{2}{n}\sum_{k=0}^{n-1} T_k \tag{7.2}$$

con la condizione iniziale $T_0 = 0$. Moltiplicando entrambi i termini dell'uguaglianza per nz^{n-1} otteniamo

$$nT_n z^{n-1} = n(n-1)z^{n-1} + 2\sum_{k=0}^{n-1} T_k z^{n-1}$$

Sommando i due termini di questa uguaglianza per tutti i valori $n \geq 1$, ricaviamo la seguente equazione:

$$T'(z) = \frac{2z}{(1-z)^3} + \frac{2}{1-z}T(z),$$

dove $T(z)$ e $T'(z)$ denotano rispettivamente la funzione generatrice di $\{T_n\}_{n\geq 0}$ e la sua derivata. Si tratta di una equazione differenziale lineare del primo ordine che può essere risolta con metodi classici. L'equazione differenziale omogenea, associata alla precedente, è

$$T'(z) = \frac{2}{1-z}T(z)$$

che ammette la soluzione $(1-z)^{-2}$; quindi l'integrale generale, valutato per $T(0) = 0$, risulta

$$T(z) = \frac{1}{(1-z)^2}\int_0^z \frac{2t}{1-t}dt = \frac{2}{(1-z)^2}(\log\frac{1}{1-z} - z).$$

Dobbiamo ora sviluppare in serie di Taylor la funzione ottenuta, applicando le proprietà delle operazioni su funzioni generatrici. Ricordiamo che $\left(\frac{1}{1-z}\right)^2$ è la funzione generatrice della sequenza $\{n+1\}$, mentre $\log\frac{1}{1-z}$ è la funzione generatrice di $\{\frac{1}{n}\}$. Di conseguenza la sequenza associata al prodotto $\frac{1}{(1-z)^2}\log\frac{1}{1-z}$ è data dalla convoluzione delle sequenze $\{n+1\}$ e $\{\frac{1}{n}\}$. Questo permette di calcolare direttamente i termini della sequenza cercata:

$$\begin{aligned} T_n &= 2\sum_{k=1}^{n}\frac{1}{k}(n+1-k)-2n \\ &= 2(n+1)\sum_{k=1}^{n}\frac{1}{k}-4n. \end{aligned}$$

Esercizio

Applicando la formula di Stirling determinare l'espressione asintotica dei numeri di Catalan $\frac{1}{n+1}\binom{2n}{n}$ per $n\to+\infty$.

7.5 Stima dei coefficienti di una funzione generatrice

Esistono potenti tecniche che permettono di determinare una stima asintotica di una sequenza f_n conoscendo la sua funzione generatrice $F(z)$ in forma chiusa oppure in modo implicito. In effetti, sono questi i metodi che attribuiscono importanza all'uso delle funzioni generatrici. Non vogliamo qui addentrarci nello studio generale di questa problematica, che richiede nozioni preliminari di teoria delle funzioni analitiche; ci limitiamo a valutare il comportamento asintotico di f_n per alcune particolari funzioni $F(z)$.

7.5.1 Funzioni razionali

Le funzioni razionali in una variabile sono quelle che possono essere rappresentate nella forma $\frac{P(z)}{Q(z)}$ dove $P(z)$ e $Q(z)$ sono polinomi in z. Queste funzioni sono rilevanti nel nostro contesto poiché si può facilmente provare che una sequenza $\{f_n\}$ è soluzione di una equazione di ricorrenza lineare omogenea a coefficienti costanti (vedi la sezione 6.5.1) se e solo se la sua funzione generatrice è razionale. Come sappiamo tali equazioni compaiono spesso nell'analisi di algoritmi.

Consideriamo allora la funzione generatrice $F(z)$ di una sequenza $\{f_n\}$ e supponiamo che

$$F(z)=\frac{P(z)}{Q(z)},$$

dove $P(z)$ e $Q(z)$ sono polinomi primi fra loro nella variabile z. Senza perdita di generalità possiamo assumere che il grado di $P(z)$ sia minore del grado di $Q(z)$. Inoltre è chiaro che le radici di $Q(z)$ sono diverse da 0 altrimenti $F(z)$ non sarebbe continua e derivabile in 0 (e quindi $\{f_n\}$ non sarebbe definita).

Per semplicità supponiamo che $Q(z)$ abbia m radici distinte $z_1, z_2, \ldots, z_m$, di molteplicità 1 e fra queste ve ne sia una sola di modulo minimo. Possiamo allora determinare la decomposizione di $F(z)$ in frazioni parziali

$$\frac{P(z)}{Q(z)}=\sum_{k=1}^{m}\frac{A_k}{z_k-z}$$

Per le ipotesi fatte, tale decomposizione esiste sempre e inoltre, per la regola de l'Hôpital, ogni costante A_k, per $k = 1, 2, \ldots, m$, soddisfa la relazione seguente:

$$A_k = \lim_{z \to z_k} (z_k - z) \cdot \frac{P(z)}{Q(z)} = P(z_k) \cdot \lim_{z \to z_k} \frac{z_k - z}{Q(z)} = -\frac{P(z_k)}{Q'(z_k)}.$$

Poiché $F(z) = \sum_{k=1}^{m} \frac{A_k}{z_k} \cdot \frac{1}{1 - z/z_k}$, ricordando che $\frac{1}{1-z/z_k}$ è la funzione generatrice di $\{\frac{1}{z_k^n}\}_{n \geq 0}$, possiamo concludere che

$$f_n = \sum_{k=1}^{m} \frac{A_k}{(z_k)^{n+1}}.$$

Se siamo interessati al comportamento asintotico, basta osservare che nella somma il termine principale è quello corrispondente alla radice di minor modulo. Tale proprietà è generale e vale anche quando le altre radici hanno molteplicità maggiore di 1. Abbiamo quindi provato la seguente proprietà.

Proposizione 7.1 *Consideriamo una funzione razionale $F(z) = P(z)/Q(z)$, dove $P(z)$ e $Q(z)$ sono polinomi primi fra loro, con $Q(0) \neq 0$; supponiamo che $Q(z)$ abbia un'unica radice $\overline{z}$ di modulo minimo e che tale radice abbia molteplicità 1. Allora, per $n \to +\infty$, la sequenza $\{f_n\}$ associata alla funzione $F(z)$ soddisfa la relazione*

$$f_n \sim -\frac{P(\overline{z})}{Q'(\overline{z})\overline{z}^{n+1}}.$$

Con tecniche analoghe si ottiene il seguente risultato, valido per radici di molteplicità arbitraria:

Proposizione 7.2 *Sia $F(z)$ una funzione razionale $F(z) = P(z)/Q(z)$, dove $P(z)$ e $Q(z)$ sono polinomi primi fra loro, con $Q(0) \neq 0$; supponiamo inoltre che $Q(z)$ ammetta un'unica radice $\overline{z}$ di modulo minimo. Se $\overline{z}$ ha molteplicità m allora la sequenza la sequenza $\{f_n\}$ associata alla $F(z)$ soddisfa la relazione*

$$f_n = \Theta\left(n^{m-1} \cdot \frac{1}{\overline{z}^n}\right).$$

7.5.2 Funzioni logaritmiche

Consideriamo ora una classe di funzioni non razionali e studiamo il comportamente asintotico delle sequenze associate.

Proposizione 7.3 *Per ogni intero $\alpha \geq 1$ la funzione*

$$\frac{1}{(1-z)^\alpha} \cdot \log \frac{1}{1-z}$$

è la funzione generatrice di una sequenza $\{f_n\}$ tale che $f_n = \Theta(n^{\alpha-1} \log n)$.

Dimostrazione. Ragioniamo per induzione su α. Nel caso $\alpha = 1$ il risultato è già stato dimostrato nelle sezioni precedenti; infatti sappiamo che

$$\frac{1}{1-z} \log \frac{1}{1-z} = \sum_{n=1}^{+\infty} H_n z^n$$

dove $H_n = \sum_{k=1}^{n} \frac{1}{k} \sim \log n$.

Supponiamo ora la proprietà vera per α fissato, $\alpha \geq 1$, e dimostriamola vera $\alpha + 1$. Denotiamo con $f_n^{(\alpha)}$ la sequenza associata alla funzione $\frac{1}{(1-z)^\alpha} \log \frac{1}{1-z}$. Si verifica subito che $\{f_n^{(\alpha+1)}\}$ è la convoluzione delle sequenze $\{f_n^{(\alpha)}\}$ e $\{1\}$ poiché la sua funzione generatrice è il prodotto delle funzioni generatrici corrispondenti.

Di conseguenza possiamo scrivere

$$\begin{aligned}
f_n^{(\alpha+1)} &= \sum_{k=0}^{n} f_k^{(\alpha)} = && \text{(per ipotesi di induzione)} \\
&= \sum_{k=1}^{n} \Theta(k^{\alpha-1} \log k) = && \text{(applicando la proposizione 2.5)} \\
&= \Theta\left(\sum_{k=1}^{n} k^{\alpha-1} \log k\right) = && \text{(per la proposizione 2.8)} \\
&= \Theta\left(\int_1^n x^{\alpha-1} \log x dx\right) = && \text{(integrando per parti)} \\
&= \Theta(n^\alpha \log n). &&
\end{aligned}$$

∎

Concludiamo osservando che la proposizione precedente può essere estesa al caso in cui $\alpha \in \mathbb{R}$, purché $\alpha \neq 0, -1, -2, \ldots, -n, \ldots$.

Esercizi

1) Considera le seguenti procedure F e G che calcolano rispettivamente i valori $F(n)$ e $G(n)$ su input $n \in \mathbb{N}$:

```
Procedure G(n)
begin
      S = 0
      for i = 0, 1, 2, ..., n do
          S = S + F(i)
      return S
end
```

```
Procedure F(n)
if n = 0 then return 1
         else return F(n − 1) + G(n − 1)
```

a) Dimostrare che, su input n, le due procedure richiedono $\Omega(a^n)$ operazioni aritmetiche per qualche $a > 1$.
b) Calcolare le funzioni generatrici delle sequenze $\{F(n)\}$ e $\{G(n)\}$ e determinare la loro espressione asintotica per $n \longrightarrow +\infty$.
c) Definire un algoritmo per calcolare $F(n)$ e $G(n)$ che esegua un numero di operazioni aritmetiche polinomiale in n.

2) Considera le seguenti procedure F e G che calcolano rispettivamente i valori $F(n)$ e $G(n)$ su input $n \in \mathbb{N}$:

```
Procedure G(n)
begin
      S = 0
      for i = 0, 1, 2, ..., n do
          S = S + F(n − i)
      return S
end
```

```
Procedure F(n)
if n = 0 then return 0
         else return n + 2F(n − 1)
```

a) Determinare l'espressione asintotica della sequenza $\{G(n)\}_n$ per $n \longrightarrow +\infty$.
b) Assumendo il criterio di costo uniforme, determinare l'ordine di grandezza del tempo di calcolo e dello spazio di memoria richiesti dall'esecuzione della procedura G su input n.
c) Svolgere il calcolo precedente assumendo il criterio di costo logaritmico.

Capitolo 8

Algoritmi di ordinamento

L'efficienza dei sistemi che manipolano insiemi di dati mantenuti in memoria dipende in larga misura dal criterio utilizzato per conservare le chiavi delle informazioni. Uno dei metodi più semplici e più usati è quello di mantenere le chiavi ordinate rispetto a una relazione d'ordine fissata. Ordinare una sequenza di valori è quindi una operazione che ricorre frequentemente nella gestione di sistemi e in applicazioni di varia natura; pertanto le procedure di ordinamento sono spesso usate per la soluzione di problemi più generali e quindi la loro efficienza può di fatto condizionare l'efficacia dei metodi adottati.

8.1 Caratteristiche generali

Per definire formalmente il problema ricordiamo innanzitutto che una relazione d'ordine (parziale) R su un insieme U è una relazione binaria che gode delle proprietà riflessiva, transitiva e antisimmetrica, ovvero:

- per ogni $a \in U$, aRa;
- per ogni $a, b, c \in U$ se aRb e bRc allora aRc;
- per ogni $a, b \in U$ se aRb e bRa allora $a = b$.

Classici esempi sono la relazione di minore o uguale sui numeri reali e l'inclusione tra i sottoinsiemi di un insieme dato. Diciamo che una relazione d'ordine R su U è totale se per ogni $a, b \in U$ vale aRb oppure bRa. In questo caso si dice anche che R definisce un ordine lineare su U.

Il problema di ordinamento per un insieme U, dotato di una relazione d'ordine totale $\leq$, è definito nel modo seguente:

> Istanza: un vettore $A = (A[1], A[2], \ldots, A[n])$ tale che $n > 1$ e $A[i] \in U$ per ogni $i \in \{1, 2, \ldots, n\}$.
>
> Soluzione: un vettore $B = (B[1], B[2], \ldots, B[n])$, ottenuto mediante una permutazione degli elementi di A, tale che $B[i] \leq B[i+1]$ per ogni $i = 1, 2, \ldots, n-1$.

I metodi adottati per risolvere il problema si diversificano in due gruppi principali chiamati rispettivamente di ordinamento *interno* ed *esterno*. Gli algoritmi di ordinamento interno presuppongono che il vettore di ingresso sia interamente contenuto nella memoria RAM della macchina. In questo caso l'accesso al valore di una qualunque delle sue componenti avviene in tempi uguali per tutte. In questa sede ci occuperemo principalmente di algoritmi di questo tipo.

Al contrario gli algoritmi che operano su dati distribuiti principalmente su memorie di massa (dischi o nastri) vengono chiamati algoritmi di ordinamento esterno. In questo caso i tempi di accesso ai dati

non sono più uniformi ma dipendono dal tipo di memoria nella quale sono collocati e uno degli obiettivi delle procedure usate è proprio quello di ridurre il numero di accessi alle memorie di massa.

Gli algoritmi di ordinamento sono suddivisi anche in base alla generalità dell'insieme U sul quale viene definito l'input. Un primo gruppo è costituito dalle procedure basate sul confronto tra gli elementi del vettore di ingresso. In questo caso si suppone di poter sempre eseguire in un numero costante di passi il confronto tra due elementi dell'insieme U rispetto alla relazione d'ordine fissata. Così l'algoritmo può essere applicato a qualunque insieme totalmente ordinato perché non sfrutta alcuna caratteristica specifica dei suoi elementi. Come vedremo, in queste ipotesi, sono necessari $\Omega(n \log n)$ confronti per ordinare un vettore di n elementi ed è possibile descrivere diverse procedure ottimali, cioè in grado di eseguire il calcolo proprio in tempo $O(n \log n)$.

Una seconda classe di algoritmi è invece costituita da quelle procedure specificamente progettate per ordinare una sequenza di stringhe definite su un alfabeto finito, ad esempio binario. In questo caso si possono progettare algoritmi che ispezionano i singoli bits delle varie stringhe, sfruttando direttamente la rappresentazione binaria degli interi. Classici esempi di algoritmi di questo tipo sono quelli che ordinano una sequenza di parole su un dato alfabeto secondo l'ordinamento lessicografico. Come vedremo, sotto opportune ipotesi, si possono definire algoritmi di questo tipo che hanno complessità in tempo lineare.

Nell'analisi degli algoritmi di ordinamento che presentiamo nel seguito assumiamo come modello di calcolo una Random Access Machine (RAM) con criterio di costo uniforme. Il costo di ogni operazione aritmetica in termini di tempo di calcolo e di spazio di memoria è quindi costante e non dipende dalle dimensioni degli operandi. Supponiamo inoltre che la nostra RAM sia in grado di mantenere in ogni cella di memoria un elemento del vettore di input e di eseguire il confronto fra due qualsiasi di questi in tempo costante.

8.2 Numero minimo di confronti

In questa sezione consideriamo gli algoritmi di ordinamento basati su confronti e presentiamo un risultato generale che riguarda il numero minimo di passi che le procedure di questo tipo devono eseguire per completare il calcolo. A tale scopo utilizziamo una proprietà degli alberi binari che risulterà utile anche in altre occasioni.

Lemma 8.1 *Ogni albero binario con k foglie ha altezza maggiore o uguale a $\lceil \log_2 k \rceil$.*

Dimostrazione. Procediamo per induzione sul numero k di foglie dell'albero considerato. Se $k = 1$ la proprietà è banalmente verificata. Supponiamo la proprietà vera per ogni j tale che $1 \leq j < k$ e consideriamo un albero binario T con k foglie di altezza minima. Siano T_1 e T_2 i due sottoalberi che hanno per radice i figli della radice di T. Osserva che ciascuno di questi ha meno di k foglie e uno dei due ne possiede almeno $\lceil \frac{k}{2} \rceil$. Allora, per ipotesi di induzione, l'altezza di quest'ultimo è maggiore o uguale a $\lceil \log_2 \frac{k}{2} \rceil$; quindi anche l'altezza di T è certamente maggiore o uguale a $1 + \lceil \log_2 \frac{k}{2} \rceil = \lceil \log_2 k \rceil$.

Proposizione 8.2 *Ogni algoritmo di ordinamento basato sui confronti richiede, nel caso peggiore, almeno $n \log_2 n - (\log_2 e)n + \frac{1}{2} \log_2 n + O(1)$ confronti per ordinare una sequenza di n elementi.*

Dimostrazione. Consideriamo un qualsiasi algoritmo basato sui confronti che opera su sequenze di oggetti distinti estratti da un insieme U totalmente ordinato. Il funzionamento generale, su input formati da n elementi, può essere rappresentato da un albero di decisione, cioè un albero binario, nel quale ogni nodo interno è etichettato mediante un confronto del tipo $a_i \leq a_j$, dove $i, j \in \{1, 2, \ldots, n\}$. Il calcolo eseguito dall'algoritmo su uno specifico input di lunghezza n, $A = (A[1], A[2], \ldots, A[n])$, identifica un

cammino dalla radice a una foglia dell'albero: attraversando un nodo interno etichettato da un confronto $a_i \leq a_j$, il cammino prosegue lungo il lato di sinistra o di destra a seconda se $A[i] \leq A[j]$ oppure $A[i] > A[j]$.
L'altezza dell'albero rappresenta quindi il massimo numero di confronti eseguiti dall'algoritmo su un input di lunghezza n. Osserviamo che il risultato di un procedimento di ordinamento di n elementi è dato da una delle $n!$ permutazioni della sequenza di input. Ne segue che l'albero di decisione deve contenere almeno $n!$ foglie perché ciascuna di queste identifica un possibile output distinto. Per il lemma precedente, possiamo allora affermare che il numero di confronti richiesti nel caso peggiore è almeno $\lceil \log_2 n! \rceil$. Applicando ora la formula di Stirling sappiamo che

$$\log_2 n! = n \log_2 n - (\log_2 e)n + \frac{\log_2(\pi n) + 1}{2} + o(1)$$

e quindi la proposizione è dimostrata.

8.3 Ordinamento per inserimento

Il primo algoritmo che consideriamo è basato sul metodo solitamente usato nel gioco delle carte per ordinare una sequenza di elementi. Si tratta di inserire uno dopo l'altro ciascun oggetto nella sequenza ordinata degli elementi che lo precedono. In altre parole, supponendo di aver ordinato le prime $i-1$ componenti del vettore, inseriamo l'elemento i-esimo nella posizione corretta rispetto ai precedenti.

L'algoritmo è descritto in dettaglio dal seguente programma:

Procedura Inserimento
Input: un vettore $A = (A[1], A[2], \ldots, A[n])$ tale che $n > 1$ e $A[i] \in U$ per
ogni $i \in \{1, 2, \ldots, n\}$;

```
begin
    for i = 2, ..., n do
        begin
            a := A[i]
            j := i − 1
            while j ≥ 1 ∧ a < A[j] do
                begin
                    A[j + 1] := A[j]
                    j := j − 1
                end
            A[j + 1] := a
        end
end
```

Osserva che l'implementazione dell'algoritmo su macchina RAM esegue al più un numero costante di passi per ogni confronto tra elementi del vettore di ingresso. Possiamo quindi affermare che il tempo di calcolo è dello stesso ordine di grandezza del numero di confronti eseguiti. Il caso peggiore, quello con il massimo numero di confronti, occorre quando $A[1] > A[2] > \cdots > A[n]$. In questo caso la procedura esegue $\sum_1^{n-1} i = \frac{n(n-1)}{2}$ confronti. Di conseguenza il tempo di calcolo richiesto dall'algoritmo su un input di lunghezza n è $\Theta(n^2)$ nel caso peggiore.

Nel caso migliore invece, quando il vettore A è già ordinato, la procedura esegue solo $n-1$ confronti e di conseguenza il tempo di calcolo risulta lineare. Osserviamo tuttavia che il caso migliore non è rappresentativo. Infatti, supponendo di avere in input una permutazione casuale (uniformemente distribuita) di elementi distinti, è stato dimostrato che il numero medio di confronti risulta $\frac{n(n-1)}{4}$. Quindi, anche nel caso medio, il tempo di calcolo resta quadratico.

8.4 Heapsort

L'algoritmo di ordinamento che presentiamo in questa sezione richiede $O(n \log n)$ confronti per ordinare una sequenza di n elementi e risulta quindi ottimale a meno di un fattore costante. Il procedimento adottato si basa su una importante struttura dati, lo heap, che è spesso utilizzata per mantenere un insieme di chiavi dal quale estrarre facilmente l'elemento massimo.

Definizione 8.1 *Fissato un insieme totalmente ordinato U, uno heap è un vettore $A = (A[1], A[2], \ldots, A[n])$, dove $A[i] \in U$ per ogni i, che gode della seguente proprietà:*

$$A[i] \geq A[2i] \text{ e } A[i] \geq A[2i+1] \text{ per ogni intero } i \text{ tale che } 1 \leq i < n/2,$$

e inoltre, se n è pari, $A[n/2] \geq A[n]$.

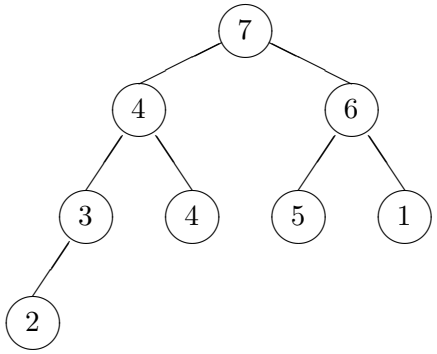
Questo significa che $A[1] \geq A[2]$, $A[1] \geq A[3]$, $A[2] \geq A[4]$, $A[2] \geq A[5]$, ecc..

Uno heap $A = (A[1], A[2], \ldots, A[n])$ può essere rappresentato da un albero binario T di n nodi, denotati dagli interi $1, 2, \ldots, n$, nel quale il nodo 1 è la radice, ogni nodo i è etichettato dal valore $A[i]$ e gode delle seguenti proprietà:

- se $1 \leq i \leq n/2$, allora $2i$ è il figlio sinistro di i;
- se $1 \leq i < n/2$, allora $2i+1$ è il figlio destro di i.

In questo modo, per ogni figlio j di i, $A[i] \geq A[j]$, e quindi l'etichetta di ogni nodo risulta maggiore o uguale a quelle dei suoi discendenti. In particolare la radice contiene il valore massimo della sequenza.

Per esempio, lo heap definito dalla sequenza $(7, 4, 6, 3, 4, 5, 1, 2)$ è rappresentato dal seguente albero binario, nel quale abbiamo riportato le etichette dei nodi al loro interno.



8.4.1 Costruzione di uno heap

Vogliamo ora definire una procedura in grado di trasformare un vettore in uno heap. Dato in input un vettore $A = (A[1], A[2], \ldots, A[n])$, l'algoritmo deve quindi riordinare gli elementi di A in modo da ottenere uno heap.

Il procedimento è basato sulla seguente definizione. Data una coppia di interi i, j, dove $1 \leq i \leq j \leq n$, diciamo che il vettore $(A[i], \ldots, A[j])$ rispetta la proprietà dello heap se, per ogni $\ell \in \{i, \ldots, j\}$ e ogni $\tilde{\ell} \in \{2\ell, 2\ell + 1\}$ tale che $\tilde{\ell} \leq j$, vale la relazione $A[\ell] \geq A[\tilde{\ell}]$. Supponiamo ora che per una data coppia $1 \leq i < j \leq n$ il vettore $(A[i + 1], \ldots, A[j])$ rispetti la proprietà dello heap; possiamo allora definire un procedimento che confronta l'elemento $A[i]$ con i suoi discendenti nell'albero ed esegue gli scambi opportuni in modo tale che anche il vettore $(A[i], A[i + 1], \ldots, A[j])$ rispetti la proprietà dello heap. Il calcolo viene eseguito dalla seguente procedura ricorsiva che si applica a ogni coppia di indici i, j tali che $1 \leq i < j \leq n$, nella quale il vettore A rappresenta una variabile globale. Nel programma si utilizza una chiamata alla procedura Scambia$(A[i], A[j])$ che ha l'effetto di scambiare i valori delle componenti di indice i e j del vettore A.

```
Procedura Aggiornaheap(i, j)
if 2i = j then se A[i] < A[j] allora Scambia(A[i], A[j])
   else if 2i < j then
        begin
              if A[2i] > A[2i + 1] then k := 2i
                                   else k := 2i + 1
              if A[i] < A[k] then
                                   begin
                                         Scambia(A[i], A[k])
                                         Aggiornaheap(k, j)
                                   end
        end
```

L'algoritmo di costruzione dello heap è allora definito dalla seguente procedura che ha come input un vettore $A = (A[1], A[2], \ldots, A[n])$ tale che $n > 1$ e $A[i] \in U$ per ogni $i \in \{1, 2, \ldots, n\}$.

```
Procedura Creaheap(A)
for i = ⌊n/2⌋, ⌊n/2⌋ − 1, . . . , 1 do Aggiornaheap(i, n)
```

La correttezza dell'algoritmo è una conseguenza della seguente osservazione: se $i + 1, \ldots, n$ sono radici di heap allora, dopo l'esecuzione di Aggiornaheap(i, n), i nodi $i, i + 1, \ldots, n$ sono ancora radici di heap.

Per quanto riguarda invece il tempo di calcolo dell'algoritmo possiamo enunciare la seguente proposizione.

Teorema 8.3 *La procedura Creaheap, su un input di dimensione n, esegue al più $4n+O(\log n)$ confronti tra elementi del vettore di ingresso, richiedendo quindi tempo $\Theta(n)$ nel caso peggiore.*

Dimostrazione. Poiché assumiamo il criterio uniforme, è chiaro che il tempo di calcolo richiesto dall'algoritmo su un dato input è proporzionale al numero di confronti eseguiti. Valutiamo quindi tale quantità nel caso peggiore.

Non è difficile verificare che l'esecuzione della procedura Aggiornaheap(i, n) richiede un numero di chiamate ricorsive al più uguale all'altezza del nodo i nello heap, ovvero la massima distanza di i da una foglia. Inoltre ogni chiamata esegue al più 2 confronti; quindi il numero di confronti eseguiti da Aggiornaheap(i, n) è al più 2 volte l'altezza del nodo i. Di conseguenza, nel caso peggiore, il numero totale di confronti eseguiti da Creaheap(A) è due volte la somma delle altezze dei nodi $i \in \{1, 2, \ldots, n-1\}$.

Ora, posto $k = \lfloor \log_2 n \rfloor$, abbiamo $2^k \leq n < 2^{k+1}$. Si può quindi verificare che nello heap ci sono al più un nodo di altezza k, due nodi di altezza $k-1, \ldots$, 2^j nodi di altezza $k-j$ per ogni $j = 0, 1, \ldots, k-1$. Quindi la somma delle altezze dei nodi dello heap è minore o uguale a

$$\sum_{j=0}^{k-1} (k-j)2^j$$

Questa somma può essere calcolata come nell'esempio 2.3, ottenendo

$$k \sum_{j=0}^{k-1} 2^j \; - \; \sum_{j=0}^{k-1} j2^j = k(2^k - 1) - (k-2)2^k - 2 = 2^{k+1} - k - 2.$$

Di conseguenza, il numero totale di confronti richiesti dall'algoritmo su un input di dimensione n è minore o uguale a $4n + O(logn)$ e il tempo di calcolo complessivo risulta quindi $\Theta(n)$.

8.4.2 Descrizione dell'algoritmo

Siamo ora in grado di definire l'algoritmo di ordinamento mediante heap.

Procedura Heapsort(A)
Input: un vettore $A = (A[1], A[2], \ldots, A[n])$ tale che $n > 1$ e $A[i] \in U$ per
ogni $i \in \{1, 2, \ldots, n\}$;

```
begin
    Creaheap(A)
    for j = n, n − 1, . . . , 2 do
        begin
            Scambia(A[1], A[j])
            Aggiornaheap(1, j − 1)
        end
end
```

La correttezza si ottiene provando per induzione che, dopo k iterazioni del ciclo for, il vettore $(A[n-k+1], \ldots, A[n])$ contiene, ordinati, i k elementi maggiori del vettore di ingresso, mentre il vettore $(A[1], \ldots, A[n-k])$ forma uno heap.

Per quanto riguarda la valutazione di complessità osserviamo che ogni chiamata Aggiornaheap$(1, j)$ richiede al più $O(\log n)$ passi e che Heapsort richiama tale procedura $n-1$ volte. Poiché il tempo richiesto da Creaheap è lineare il numero totale di passi risulta $O(n \log n)$.

Concludiamo osservando che Aggiornaheap$(1, j)$ esegue al più $2\lfloor \log_2 j \rfloor$ confronti fra elementi del vettore A. Di conseguenza, nel caso peggiore, il numero totale di confronti eseguiti da Heapsort è minore o uguale a

$$4n + O(\log n) + 2 \sum_{j=1}^{n-1} \lfloor \log_2 j \rfloor = 2n \log_2 n + O(n).$$

Quindi anche il suo tempo di calcolo nel caso peggiore risulta $\Theta(n \log n)$.

Esercizio

Esegui Heapsort sulla sequenza $(1, 3, 2, 5, 6, 7, 9, 4, 2)$.

Esercizio

Descrivi una versione non ricorsiva della procedura Aggiornaheap. Ci troviamo nel caso di una ricorsione terminale?

8.5 Quicksort

L'algoritmo Quicksort è una classica procedura di ordinamento, basata su un metodo di partizione della sequenza di input, che può essere facilmente implementata e offre buone prestazioni nel caso medio. Per questo motivo è utilizzata come routine di ordinamento in molte applicazioni e viene spesso preferita ad altre procedure che pur avendo complessità ottimale nel caso peggiore non offrono gli stessi vantaggi in media.

La versione che presentiamo in questa sezione è randomizzata, ovvero prevede l'esecuzione di alcuni passi casuali. L'idea dell'algoritmo è semplice. La procedura sceglie casualmente un elemento α nella sequenza di input e quindi suddivide quest'ultima in due parti, creando il vettore degli elementi minori o uguali ad α e quello degli elementi maggiori di α. In seguito l'algoritmo richiama ricorsivamente se stesso sui due vettori ottenuti concatenando poi le sequenze ordinate.

Ad alto livello l'algoritmo può essere descritto dalla seguente procedura nella quale gli elementi del vettore di ingresso sono estratti da un insieme U totalmente ordinato e assumono valori non necessariamente distinti. Denotiamo qui con il simbolo $*$ l'operazione di concatenazione tra vettori.

```
Procedure Quicksort(A)
Input: A = (a_1, a_2, ..., a_n) tale che a_i ∈ U per ogni i ∈ {1, 2, ..., n}.
begin
  if n ≤ 1 then return A
          else
              begin
                    scegli a caso un intero k in {1, 2, ..., n}
                    calcola il vettore A_1 degli elementi a_i di A tali che i ≠ k e a_i ≤ a_k
                    calcola il vettore A_2 degli elementi a_j di A tali che a_j > a_k
                    A_1 := Quicksort(A_1)
                    A_2 := Quicksort(A_2)
                    return A_1 * (a_k) * A_2
              end
end
```

La correttezza della procedura può essere dimostrata facilmente per induzione sulla lunghezza del vettore di input.

Per valutare il tempo di calcolo richiesto supponiamo di poter generare in tempo $O(n)$ un numero intero casuale uniformemente distribuito nell'insieme $\{1, 2, \ldots, n\}$. In queste ipotesi l'ordine di grandezza del tempo di calcolo è determinato dal numero di confronti eseguiti fra elementi del vettore di ingresso.

8.5.1 Analisi dell'algoritmo

Denotiamo con $T_w(n)$ il massimo numero di confronti tra elementi del vettore di ingresso eseguiti dall'algoritmo su un input A di lunghezza n. È evidente che i vettori A_1 e A_2 della partizione possono

essere calcolati mediante $n-1$ confronti. Inoltre la dimensione di A_1 e A_2 è data rispettivamente da k e $n-k-1$, per qualche $k \in \{0,1,\ldots,n-1\}$. Questo implica che per ogni $n \geq 1$

$$T_w(n) = n - 1 + \max_{0 \leq k \leq n-1} \{T_w(k) + T_w(n-k-1)\},$$

mentre $T_w(0) = 0$.

Vogliamo ora determinare il valore esatto di $T_w(n)$. Come vedremo, tale valore occorre quando ogni estrazione casuale determina l'elemento massimo o minimo del campione. Infatti, poiché $\max_{0 \leq k \leq n-1}\{T_w(k) + T_w(n-k-1)\} \geq T_w(n-1)$, abbiamo che $T_w(n) \geq n - 1 + T_w(n-1)$ e quindi, per ogni $n \in \mathbb{N}$, otteniamo

$$T_w(n) \geq \sum_{0}^{n-1} k = \frac{n(n-1)}{2} \tag{8.1}$$

Dimostriamo ora per induzione che $T_w(n) \leq \frac{n(n-1)}{2}$. La disuguaglianza è banalmente vera per $n = 0$. Supponiamo che $T_w(k) \leq \frac{k(k-1)}{2}$ per ogni intero k tale che $0 \leq k < n$; sostituendo tali valori nella equazione di ricorrenza e svolgendo semplici calcoli si ottiene

$$T_w(n) \leq n - 1 + \frac{1}{2} \max_{0 \leq k \leq n-1} \{2k(k-n+1) + n^2 - 3n + 2\}$$

Lo studio della funzione $f(x) = 2x(x-n+1)$ mostra che il valore massimo assunto dalla f nell'intervallo $[0, n-1]$ è 0. Di conseguenza

$$T_w(n) \leq n - 1 + \frac{n^2 - 3n + 2}{2} = \frac{n(n-1)}{2}$$

Assieme alla relazione (8.1) quest'ultima disuguaglianza implica

$$T_w(n) = \frac{n(n-1)}{2}$$

Quindi l'algoritmo Quicksort nel caso peggiore richiede un tempo $\Theta(n^2)$.

Valutiamo ora il numero medio di confronti tra elementi del vettore di ingresso eseguiti dall'algoritmo. Chiaramente tale valore determina anche l'ordine di grandezza del tempo medio di calcolo necessario per eseguire la procedura su una macchina RAM.

Per semplicità supponiamo che tutti gli elementi del vettore A di input siano distinti e che, per ogni n, la scelta casuale dell'intero k nell'insieme $\{1,2,\ldots,n\}$ avvenga in maniera uniforme. In altre parole ogni elemento del campione ha probabilità $\frac{1}{n}$ di essere scelto. Inoltre supponiamo che il risultato di ogni scelta casuale sia indipendente dalle altre.

Denotiamo allora con $E(n)$ il numero medio di confronti eseguiti dall'algoritmo su un input di lunghezza n e, per ogni $j \in \{0,1,\ldots,n-1\}$, rappresentiamo con $E(n \mid |A_1| = j)$ il numero medio di confronti eseguiti supponendo che il vettore A_1, ottenuto dal processo di partizione, abbia j componenti. Possiamo allora scrivere

$$E(n) = \sum_{j=0}^{n-1} \Pr\{|A_1| = j\} E(n \mid |A_1| = j) = \sum_{j=0}^{n-1} \frac{1}{n}\{n - 1 + E(j) + E(n-j-1)\}$$

Attraverso semplici passaggi otteniamo la sequente equazione di ricorrenza:

$$E(n) = \begin{cases} 0 & \text{se } n = 0, 1 \\ n - 1 + \frac{2}{n} \sum_{j=0}^{n-1} E(j) & \text{se } n \geq 2 \end{cases} \tag{8.2}$$

Tale ricorrenza può essere risolta applicando il metodo della sostituzione discusso nella sezione 6.6.1 oppure passando alle funzioni generatrici (vedi sezione 7.4.2).

Possiamo così ricavare il valore esatto di $E(n)$ per ogni $n \geq 1$:

$$E(n) = 2(n+1) \sum_{j=1}^{n} \frac{1}{j} - 4n = 2n \log n + O(n)$$

Di conseguenza l'algoritmo Quicksort può essere eseguito in tempo $\Theta(n \log n)$ nel caso medio.

Esercizi

1) Confrontare la valutazione appena ottenuta del numero di confronti di Quicksort nel caso medio con quella dell'algoritmo Heapsort nel caso peggiore.

2) Considera la sequenza $\{C(n)\}$ tale che $C(0) = C(1) = 0$ e, per ogni intero $n \geq 2$,

$$C(n) = n + 1 + \max_{k=0,1,\ldots,n-1} \{C(k) + C(n-1-k)\}$$

Dimostrare che $C(n) = (n^2 + 3n - 4)/2$, per ogni $n \geq 1$.

8.5.2 Specifica dell'algoritmo

Vogliamo ora fornire una versione più dettagliata dell'algoritmo che specifichi la struttura dati utilizzata e il processo di partizione. Il nostro obiettivo è quello di implementare la procedura *in loco*, mediante un procedimento che calcoli la sequenza ordinata attraverso scambi diretti tra i valori delle sue componenti, senza usare vettori aggiuntivi per mantenere i risultati parziali della computazione. In questo modo lo spazio di memoria utilizzato è essenzialmente ridotto alle celle necessarie per mantenere il vettore di ingresso e per implementare la ricorsione.

Rappresentiamo la sequenza di input mediante il vettore $A = (A[1], A[2], \ldots, A[n])$ di $n > 1$ componenti. Per ogni coppia di interi p, q, tali che $1 \leq p < q \leq n$, denotiamo con $A_{p,q}$ il sottovettore compreso tra le componenti di indice p e q, cioè $A_{p,q} = (A[p], A[p+1], \ldots, A[q])$. Il cuore dell'algoritmo è costituito dalla funzione Partition(p, q) che ripartisce gli elementi del vettore $A_{p,q}$ rispetto al valore α della prima componente $A[p]$; questa funzione modifica quindi il valore delle componenti di $A_{p,q}$ e restituisce un indice $\ell \in \{p, p+1, \ldots, q\}$ che gode delle seguenti proprietà:

- $A[\ell]$ assume il valore α;
- $A_{p,\ell-1}$ contiene i valori minori o uguali ad α, originariamente contenuti in $A_{p+1,q}$;
- $A_{\ell+1,q}$ contiene i valori maggiori di α, originariamente contenuti in $A_{p+1,q}$.

La funzione Partition può essere calcolata dalla seguente procedura nella quale due indici (i e j) scorrono il vettore da destra a sinistra e viceversa, confrontando le componenti con l'elemento scelto casualmente.

Supponiamo inoltre che il parametro A rappresenti una variabile globale; per questo motivo gli unici parametri formali della procedura sono p e q che rappresentano gli indici del sottovettore sul quale si opera la partizione (assumiamo sempre $1 \leq p < q \leq n$). Le altre variabili che compaiono nella procedura sono locali.

```
Function Partition(p, q)
begin
  i := p + 1
  j := q
  while i ≤ j do
        begin
              while A[j] > A[p] do j := j − 1
              while A[i] ≤ A[p] ∧ i ≤ j do i := i + 1
              if i < j then
                          begin
                                Scambia(A[i], A[j])
                                i := i + 1
                                j := j − 1
                          end
        end
  Scambia(A[p], A[j])
  return j
end
```

Definiamo ora la procedura Quicksort(p, q) che ordina il vettore $A_{p,q}$ utilizzando la funzione Partition definita sopra; anche in questo caso A è una variabile globale e gli unici parametri formali sono gli indici p e q che definiscono il sottovettore da ordinare.

```
Procedure Quicksort(p, q)
  begin
(1)     scegli a caso un intero k in {p, p + 1, ..., q}
(2)     Scambia(A[p], A[k])
(3)     ℓ := Partition(p, q)
(4)     if p < ℓ − 1 then Quicksort(p, ℓ − 1)
(5)     if ℓ + 1 < q then Quicksort(ℓ + 1, q)
  end
```

L'algoritmo complessivo è quindi dato dalla semplice chiamata Quicksort$(1, n)$ e dalla dichiarazione di A come variabile globale.

Esercizi

1) La chiamata di procedura Partition(p, q) può eseguire, a seconda dei valori del vettore di ingresso A, $n-1$, n oppure $n+1$ confronti su elementi di A. Fornire un esempio per ciascuno di questi casi.

2) Dimostrare che la procedura Quicksort$(1, n)$ esegue nel caso peggiore $\frac{(n+1)(n+2)}{2} - 3$ confronti tra elementi del vettore di ingresso.

3) Stimare il numero medio di confronti eseguiti dalla procedura Quicksort$(1, n)$, assumendo le ipotesi presentate nella sezione 8.5.1.

8.5.3 Ottimizzazione della memoria

Vogliamo ora valutare lo spazio di memoria necessario per implementare su macchina RAM la procedura definita nella sezione 8.5.2. Oltre alle n celle necessarie per contenere il vettore di ingresso, occorre

utilizzare una certa quantità di spazio per mantenere la pila che implementa la ricorsione. Se applichiamo lo schema presentato nella sezione 5.1, la traduzione iterativa su macchina RAM della procedura Quicksort$(1, n)$ utilizza, nel caso peggiore, uno spazio di memoria $O(n)$ per mantenere la pila. Se infatti viene sempre estratto l'elemento maggiore del campione, la pila deve conservare i parametri relativi a un massimo di $n - 1$ chiamate ricorsive. In questa sezione introduciamo alcune ulteriori modifiche all'algoritmo e descriviamo una diversa gestione della pila, in parte basata sulla ricorsione terminale descritta nella sezione 5.2, che permette di ridurre lo spazio di memoria richiesto dalla pila a una quantità $O(\log n)$.

Osserviamo innanzitutto che la procedura Quicksort descritta nella sezione 8.5.2 può essere migliorata modificando l'ordine delle chiamate ricorsive. Più precisamente possiamo forzare la procedura a eseguire sempre per prima la chiamata relativa al sottovettore di lunghezza minore. Si ottiene il nuovo algoritmo semplicemente sostituendo i comandi (4) e (5) della procedura Quicksort(p, q) con le seguenti istruzioni:

```
if ℓ − p ≤ q − ℓ then
   begin
         if p < ℓ − 1 then Quicksort(p, ℓ − 1)
         if ℓ + 1 < q then Quicksort(ℓ + 1, q)
   end
  else
   begin
         if ℓ + 1 < q then Quicksort(ℓ + 1, q)
         if p < ℓ − 1 then Quicksort(p, ℓ − 1)
   end
```

Descriviamo ora una versione iterativa del nuovo algoritmo. Osserviamo innanzitutto che, nel nostro caso, il criterio di gestione della pila può essere semplificato sfruttando il fatto che le due chiamate ricorsive sono le ultime istruzioni della procedura. Possiamo cioè definire una versione iterativa nella quale la pila serve per mantenere l'elenco delle chiamate che devono ancora essere eseguite e non sono state neppure iniziate. In altre parole, nella esecuzione della procedura, la prima chiamata ricorsiva viene attivata dopo aver accantonato in testa alla pila i parametri necessari per eseguire la seconda. Quest'ultima sarà attivata una volta completata la precedente, quando i suoi parametri si troveranno di nuovo in testa alla pila. In particolare non abbiamo bisogno di mantenere nella pila il record di attivazione della procedura chiamante.

L'algoritmo così ottenuto è descritto nella seguente procedura. Come nella sezione precedente, denotiamo con A il vettore di input di lunghezza $n > 1$ che si suppone già presente in memoria. Il vettore corrente è rappresentato dagli indici p e q, $1 \leq p, q \leq n$, che inizialmente coincidono con 1 e n rispettivamente. Se $p \leq q$ il vettore corrente è $(A[p], A[p + 1], \ldots, A[q])$ ed è costituito da $q - p + 1$ elementi; se invece $q < p$ il vettore corrente è vuoto. Usando la funzione Partition la procedura spezza il vettore corrente in due parti rappresentate dagli indici $p, \ell - 1$ e $\ell + 1, q$, dove ℓ indica la posizione del pivot restituita da Partition(p, q). La maggiore tra queste due viene memorizzata nella pila mentre la minore diventa il nuovo vettore corrente.

Nel programma usiamo poi la variabile S per rappresentare la pila e il valore Λ per denotare la pila vuota. Gli elementi di S sono coppie di indici (i, j) che rappresentano i sottovettori accantonati nella pila. Rappresentiamo quindi con Pop, Push e Top le tradizionali operazioni sulla pila.

Procedure Quicksort Iterativo
Input : un vettore $A = (A[1], A[2], \ldots, A[n])$ tale che $n > 1$ e $A[i] \in U$ per
ogni $i \in \{1, 2, \ldots, n\}$;

```
begin
```
$p := 1$
$q := n$
$S := \Lambda$
stop:$= 0$
`repeat`
`while` $p < q$ `do`
`begin`
scegli a caso un intero k in $\{p, p+1, \ldots, q\}$
Scambia$(A[p], A[k])$
$\ell :=$Partition(p, q)

$$\texttt{if } \ell - p < q - \ell \texttt{ then} \begin{cases} i := \ell + 1 \\ j := q \\ q := \ell - 1 \end{cases} \quad \texttt{else} \begin{cases} i := \ell - 1 \\ j := p \\ p := \ell + 1 \end{cases}$$

$S := \mathrm{Push}(S, (i, j))$
`end`

$$\texttt{if } S \neq \Lambda \texttt{ then} \begin{cases} (p, q) := \mathrm{Top}(S) \\ S := \mathrm{Pop}(S) \end{cases}$$

`else` stop:$= 1$
`until` stop$= 1$
`return` A
```
end
```

Si può dimostrare che la procedura è corretta. Infatti, al termine dell'esecuzione di ogni ciclo repeat-until, gli elementi del vettore di ingresso non ancora ordinati sono contenuti nella pila S oppure nel vettore corrente. La verifica di questa proprietà è facile. Di conseguenza, quando si esce dal ciclo, abbiamo che $S = \Lambda$ e $q - p < 1$ e questo garantisce che il vettore di ingresso è ordinato.

Valutiamo ora l'altezza massima raggiunta dalla pila durante l'esecuzione dell'algoritmo. Osserviamo innanzitutto che il vettore corrente sul quale la procedura sta lavorando non è mai maggiore del vettore che si trova in testa alla pila S (tranne quando la pila è vuota). Inoltre, ad ogni incremento di S la dimensione del vettore corrente viene ridotta almeno della metà. Quando un coppia (p, q) viene tolta dalla pila questa rappresenta il nuovo vettore corrente e la sua dimensione non è maggiore di quella dello stesso vettore corrente appena prima dell'inserimento di (p, q) in S. Quindi, durante la computazione la pila può contenere al più $\lfloor \log_2 n \rfloor$ elementi, dove n è la dimensione dell'input.

Esercizi

1) Implementare l'algoritmo iterativo appena descritto su un vettore di elementi distinti e determinare l'altezza massima raggiunta dalla pila. Ripetere l'esecuzione del programma diverse volte e fornire il valore medio dell'altezza raggiunta.

2) Assumendo il criterio di costo logaritmico, determinare l'ordine di grandezza dello spazio di memoria richiesto per mantenere la pila.

8.6 Statistiche d'ordine

In questa sezione descriviamo un algoritmo per un problema legato in modo naturale all'ordi-namento di un vettore: determinare il k-esimo elemento più piccolo di una data sequenza di oggetti. Fissato un insieme totalmente ordinato U il problema è definito formalmente nel modo seguente:

Istanza: un vettore $A = (A[1], A[2], \ldots, A[n])$ e un intero k, tali che $1 \leq k \leq n$ e $A[i] \in U$ per ogni $i \in \{1, 2, \ldots, n\}$.

Richiesta: calcolare il k-esimo più piccolo elemento di A.

Un caso particolarmente interessante di tale problema è il calcolo della mediana, corrispondente al valore $k = \lceil n/2 \rceil$.

Consideriamo ora un algoritmo qualsiasi che risolve correttamente il problema. Risulta evidente che l'elemento dato in uscita deve essere confrontato con tutti i rimanenti, altrimenti modificando l'eventuale componente non confrontata, otterremmo una risposta scorretta. Possiamo così enunciare il seguente risultato.

Teorema 8.4 *Ogni algoritmo per il calcolo del k-esimo elemento più piccolo di un vettore richiede almeno $n - 1$ confronti su un input di lunghezza n.*

Una soluzione al problema può essere quella di ordinare il vettore determinando poi la componente di posto k della sequenza ordinata. Tale metodo richiede $\Omega(n \log n)$ confronti e potrebbe quindi essere migliorabile. Mostreremo infatti un algoritmo in grado di risolvere il problema in tempo $O(n)$, risultando quindi ottimale a meno di un fattore costante.

Fissato un intero dispari $2t + 1$, l'algoritmo che vogliamo descrivere suddivide il vettore di ingresso $A = (A[1], A[2], \ldots, A[n])$ in $\lceil \frac{n}{2t+1} \rceil$ sottovettori di $2t + 1$ elementi ciascuno (tranne al più l'ultimo). Per ognuno di questi viene calcolata la mediana e quindi si richiama ricorsivamente la procedura per determinare la mediana M delle mediane ottenute. Si determinano allora i vettori A_1, A_2 e A_3, formati rispettivamente dagli elementi di A minori, uguali e maggiori di M. Si distinguono quindi i seguenti casi:

- se $k \leq |A_1|$, si risolve ricorsivamente il problema per l'istanza (A_1, k);
- se $|A_1| < k \leq |A_1| + |A_2|$ allora la risposta è M;
- se $|A_1| + |A_2| < k$ si risolve ricorsivamente il problema per l'istanza $(A_3, k - |A_1| - |A_2|)$;

Svolgiamo una breve analisi del tempo di calcolo richiesto dall'algoritmo. Osserviamo anzitutto che ci sono almeno $\lfloor \frac{n}{2(2t+1)} \rfloor$ mediane minori o uguali a M; ciascuna di queste nel proprio campione ammette altri t elementi minori o uguali. Quindi l'insieme $A_1 \cup A_2$ contiene almeno $(t + 1)\lfloor \frac{n}{2(2t+1)} \rfloor$ elementi e di conseguenza A_3 ne possiede al più $n - (t + 1)\lfloor \frac{n}{2(2t+1)} \rfloor$. Lo stesso discorso vale per A_1.

Non è difficile provare che, per n maggiore o uguale a un valore H opportuno, risulta

$$n - (t + 1)\lfloor \frac{n}{2(2t + 1)} \rfloor \leq (\frac{6t + 3}{4(2t + 1)})n.$$

Modifichiamo allora l'algoritmo, richiedendo che su un input di lunghezza minore di H, il k-esimo elemento della sequenza venga calcolato direttamente mentre, su un input di lunghezza $n \geq H$, si esegua il procedimento sopra descritto.

Denotando quindi con $T(n)$ il tempo di calcolo richiesto nel caso peggiore su un input di lunghezza n, otteniamo

$$T(n) = \begin{cases} c & \text{se } n < H \\ T(\frac{n}{2t+1}) + T(\frac{6t+3}{4(2t+1)}n) + O(n) & \text{se } n \geq H. \end{cases}$$

dove c è una costante opportuna. Infatti $T(\frac{n}{2t+1})$ è il tempo di calcolo necessario per determinare la mediana delle mediane; $T(\frac{6t+3}{4(2t+1)}n)$ è il tempo necessario per eseguire l'eventuale chiamata ricorsiva; infine, $O(n)$ passi occorrono per calcolare le $\frac{n}{2t+1}$ mediane e deteminare gli insiemi A_1, A_2 e A_3.

Osserviamo ora che $\frac{1}{2t+1} + \frac{6t+3}{4(2t+1)} < 1$ se e solo se $t > \frac{3}{2}$. Ricordando la soluzione delle equazioni di ricorrenza del tipo

$$T(n) = \begin{cases} c & \text{se } n < b \\ T(\lfloor \alpha n \rfloor) + T(\lceil \beta n \rceil) + n & \text{se } n \geq b. \end{cases}$$

presentata nel corollario 6.2, possiamo concludere che per ogni $t > \frac{3}{2}$ l'algoritmo lavora in tempo $T(n) = O(n)$ nel caso peggiore.

Scegliendo $t = 2$ possiamo fissare $H = 50$ e otteniamo la seguente procedura:

```
Procedura Seleziona(A = (A[1], A[2], ..., A[n]), k)
if n < 50 then
        begin
            ordina (A[1], A[2], ..., A[n])
            return A[k]
        end
    else
        begin
            dividi A in ⌈n/5⌉ sequenze S_1, S_2, ..., S_⌈n/5⌉
            for k ∈ {1, 2, ..., ⌈n/5⌉} do calcola la mediana M_k di S_k
            M := Seleziona((M_1, M_2, ..., M_⌈n/5⌉), ⌈n/10⌉)
            calcola il vettore A_1 degli elementi di A minori di M
            calcola il vettore A_2 degli elementi di A uguali a M
            calcola il vettore A_3 degli elementi di A maggiori di M
            if k ≤ |A_1| then return Seleziona(A_1, k)
            if |A_1| < k ≤ |A_1| + |A_2| then return M
            if |A_1| + |A_2| < k then return Seleziona(A_3, k − |A_1| − |A_2|)
        end
```

8.7 Bucketsort

Gli algoritmi che abbiamo presentato nelle sezioni precedenti sono tutti basati sul confronto e non utilizzano alcuna specifica proprietà degli elementi da ordinare. In questa sezione presentiamo invece un algoritmo che dipende dalle proprietà dell'insieme dal quale gli oggetti della sequenza di input sono estratti.

Come vedremo, il procedimento che illustriamo è stabile, cioè fornisce in uscita un vettore nel quale gli elementi che hanno lo stesso valore conservano l'ordine reciproco che possedevano nella sequenza di input. Questa proprietà è spesso richiesta dalle procedure di ordinamento poiché i valori di ingresso possono essere campi di record che contengono altre informazioni e che sono originariamente ordinati secondo un criterio differente. In molti casi, qualora due elementi abbiano lo stesso valore, si vuole conservare l'ordine precedente.

L'idea dell'algoritmo Bucketsort viene solitamente presentata attraverso un semplice esempio. Supponiamo di voler ordinare n interi $x_1, x_2, \ldots, x_n$, che variano in un intervallo $[0, m-1]$, dove $m \in \mathbb{N}$. Se m è abbastanza piccolo, può essere conveniente usare il seguente procedimento:

```
begin
    for i ∈ {0, 1, ..., m − 1} do crea una lista L(i) inizialmente vuota
    for j ∈ {1, 2, ..., n} do aggiungi x_j alla lista L(x_j)
    concatena le liste L(1), L(2), ..., L(m) nel loro ordine
end
```

La lista ottenuta dalla concatenazione di $L(1), L(2), \ldots, L(m)$ fornisce ovviamente la sequenza ordinata.

Questo metodo richiede chiaramente $O(m+n)$ passi e risulta quindi lineare se m ha lo stesso ordine di grandezza di n, migliorando quindi, in questo caso, le prestazioni degli algoritmi presentati nelle sezioni precedenti.

Con alcune variazioni la stessa idea può essere usata per ordinare una sequenza di parole su un alfabeto finito. Sia Σ un alfabeto finito formato da m simboli distinti e denotiamo con Σ^+ l'insieme delle parole definite su Σ (esclusa la parola vuota). Definiamo ora l'ordinamento lessicografico su Σ^+ che, come è ben noto, corrisponde al tradizionale ordinamento delle parole in un dizionario. Sia $\preceq$ una relazione d'ordine totale su Σ e siano $x = x_1x_2\cdots x_k$ e $y = y_1y_2\cdots y_h$ due parole in Σ^+, dove $x_i, y_j \in \Sigma$ per ogni i e ogni j. Diciamo che x precede lessicograficamente y ($x \leq_\ell y$) se:

- x è un prefisso di y, cioè $k \leq h$ e $x_i = y_i$ per ogni $i \leq k$,
- oppure esiste j, $1 \leq j \leq k$, tale che $x_i = y_i$ per ogni $i < j$, $x_j \neq y_j$ e $x_j \preceq y_j$.

È facile verificare che $\leq_\ell$ è una relazione d'ordine totale su Σ^+. Descriviamo allora una procedura per ordinare, rispetto alla relazione $\leq_\ell$, una n-pla di parole su Σ^+, tutte di lunghezza k. Applicando il procedimento illustrato sopra l'algoritmo costruisce la lista delle parole di ingresso ordinate rispetto all'ultima lettera. Successivamente, si ordina la lista così ottenuta rispetto alla penultima lettera e si prosegue nello stesso modo per tutte le k posizioni in ordine decrescente. Come dimostreremo in seguito la lista che si ottiene risulta ordinata rispetto alla relazione $\leq_\ell$.

Esempio 8.1 Consideriamo l'alfabeto $\{a, b, c\}$, dove $a \preceq b \preceq c$, e applichiamo il metodo illustrato alla sequenza di parole

$$bacc, abac, baca, abbc, cbab.$$

L'algoritmo esegue 4 cicli, uno per ogni posizione delle parole di ingresso. Denotiamo con L_a, L_b, L_c le liste delle parole che hanno rispettivamente la lettera a, b, c nella posizione corrente. Le liste al termine di ciascun ciclo sono le seguenti:

1) $L_a = (baca)$
   $L_b = (cbab)$
   $L_c = (bacc, abac, abbc)$
2) $L_a = (cbab, abac)$
   $L_b = (abbc)$
   $L_c = (baca, bacc)$
3) $L_a = (baca, bacc)$
   $L_b = (cbab, abac, abbc)$
   $L_c = ()$
4) $L_a = (abac, abbc)$
   $L_b = (baca, bacc)$
   $L_c = (cbab)$

La sequenza ordinata è data dalla concatenazione delle liste L_a, L_b e L_c ottenute al temine dell'ultima iterazione.

∎

Descriviamo ora nel dettaglio la procedura. Siano $a_1, a_2, \ldots, a_m$ gli elementi di Σ considerati nell'ordine fissato. L'input è dato da una sequenza di parole $X_1, X_2, \ldots, X_n$ tali che, per ogni $i \in \{1, 2, \ldots, n\}$, $X_i = b_{i1}b_{i2}\cdots b_{ik}$ dove $b_{ij} \in \Sigma$ per ogni j. Per ogni lettera a_i, $L(a_i)$ denota la lista relativa alla lettera a_i; all'inizio di ogni ciclo ciascuna di queste è vuota (Λ). Denotiamo inoltre con Q la lista contenente la concatenazione delle $L(a_i)$ al termine di ogni iterazione. È evidente che in una reale implementazione le liste $L(a_i)$ e Q non conterranno effettivamente le parole di ingresso ma, più semplicemente, una sequenza di puntatori a tali stringhe.

Procedura Bucketsort
Input: una sequenza di parole $X_1, X_2, \ldots, X_n$ tali che, per ogni $i \in \{1, 2, \ldots, n\}$,
$X_i = b_{i1}b_{i2}\cdots b_{ik}$ dove $b_{ij} \in \Sigma$ per ogni $j = 1, 2, \ldots, k$.

```
begin
    for i = 1,2,...,n do aggiungi X_i in coda a Q
    for j = k,k-1,...,1 do
        begin
            for ℓ = 1,2,...,m do L(a_ℓ) := Λ
            while Q ≠ Λ do
                begin
                    X := Front(Q)
                    Q := Dequeue(Q)
                    sia a_t la lettera di X di posto j
                    Inserisci_in_coda(L(a_t), X)
                end
            for ℓ = 1,...,m do concatena L(a_ℓ) in coda a Q
        end
end
```

Teorema 8.5 *L'algoritmo Bucketsort ordina lessicograficamente una sequenza di n parole di lunghezza k, definite su un alfabeto di m lettere, in tempo $O(k(n+m))$.*

Dimostrazione. Si prova la correttezza dell'algoritmo per induzione sul numero di iterazioni del loop più esterno. Al termine della i-esima iterazione la lista Q contiene le parole ordinate lessicograficamente rispetto alle ultime i lettere. Osserva infatti che, se alla $i+1$-esima iterazione, due parole sono poste nella stessa lista $L(a_t)$, queste mantengono l'ordine reciproco che possedevano al termine dell'i-esimo ciclo. Nota infine che ogni ciclo richiede $O(n+m)$ passi per analizzare uno dopo l'altro gli elementi di Q e concatenare le liste $L(a_t)$. Quindi $O(k(n+m))$ passi sono necessari per terminare l'implementazione.

Concludiamo ricordando che l'algoritmo presentato può essere migliorato ed esteso al caso in cui le parole di ingresso sono di lunghezza diversa. Se L è la somma delle lunghezza delle parole di ingresso si possono ordinare queste ultime in tempo $O(m+L)$.

Esercizi

1) Tra gli algoritmi di ordinamento presentati nelle sezioni precedenti quali sono quelli stabili?

2) Ricordando l'algoritmo per il calcolo della mediana di una sequenza, descrivere una nuova versione di Quicksort che richiede un tempo $O(n \log n)$ nel caso peggiore.

Capitolo 9

Strutture dati e algoritmi di ricerca

Analizzando un problema, spesso ci si accorge che esso può essere agevolmente risolto mediante sequenze di prefissate operazioni su opportuni insiemi. In altri termini, il progetto di un algoritmo risolutivo per il problema risulta agevolato se ci si riferisce ad una "naturale" struttura di dati, dove conveniamo di chiamare *struttura dati* la famiglia di tali insiemi ed operazioni.

L'individuazione di una struttura di dati permette di suddividere il problema in (almeno) due fasi:

1. progetto di un algoritmo risolutivo "astratto" espresso in termini di operazioni della struttura di dati.

2. progetto di algoritmi efficienti per la rappresentazione dei dati e l'implementazione delle operazioni sul modello di calcolo a disposizione.

Questa impostazione unisce semplicità, chiarezza e facilità di analisi. Infatti la correttezza dell'algoritmo "eseguibile" può essere ottenuta considerando separatamente la correttezza dell'algoritmo "astratto" e quella dell'implementazione delle operazioni. Infine, si può osservare che molti problemi ammettono come "naturale" la stessa struttura dati: particolare attenzione andrà in tal caso dedicata all'implementazione efficiente delle operazioni (algoritmi di base).

Questo approccio è già stato utilizzato nel capitolo 4 per introdurre le strutture dati di base quali vettori, liste, pile, ecc. Vogliamo ora studiare l'argomento con maggiore sistematicità presentando una nozione generale di struttura dati.

9.1 Algebre eterogenee

Introduciamo innanzitutto un minimo di terminologia. La nozione centrale

STRUTTURA DI DATI = INSIEMI + OPERAZIONI

viene colta dal concetto di algebra eterogenea.

Data una sequenza di insiemi $[A_1, A_2, \ldots, A_n]$, diremo che k è il tipo dell'insieme A_k. Una funzione parziale $f : A_{k(1)} \times \ldots \times A_{k(s)} \to A_l$ è detta operazione su $[A_1, A_2, \ldots, A_n]$ di arietà $(k(1)k(2)\ldots k(s), l)$; la arietà di una operazione è dunque una coppia (w, y) dove w è una parola su $\{1, 2, \ldots, n\}$ e y è un elemento in $\{1, 2, \ldots, n\}$. Se indichiamo con ε la parola vuota, conveniamo che con operazione di arietà (ε, k) si debba intendere un elemento di A_k; tali "operazioni" sono anche dette *costanti*.

Definizione 9.1 *Un'algebra eterogenea A è una coppia $A = \langle [A_1, \ldots, A_n], [f_1, \ldots, f_k] \rangle$, dove A_1, A_2, $\ldots$, A_n sono insiemi e $f_1, f_2, \ldots, f_k$ sono operazioni su $[A_1, A_2, \ldots, A_n]$ di data arietà.*

Vediamo alcuni esempi di algebre eterogenee.

Esempio 9.1 [PARTI DI A]

PARTI DI A $= \langle [A, SUBSET(A), BOOL], [MEMBER, INSERT, DELETE,$
$\emptyset, a_1, a_2, \ldots, a_m] \rangle$
dove:

- $A = \{a_1, a_2, \ldots, a_m\}$ è un insieme fissato ma arbitrario, $SUBSET(A)$ è la famiglia di sottoinsiemi di A con $\emptyset$ insieme vuoto, $BOOL = \{vero, falso\}$;
- $MEMBER : A \times SUBSET(A) \to BOOL$,
  $INSERT, DELETE : A \times SUBSET(A) \to SUBSET(A)$
  sono le seguenti operazioni:

$$MEMBER(x, Y) = \begin{cases} \text{vero} & \text{se } x \in Y \\ \text{falso} & \text{altrimenti} \end{cases}$$
$$INSERT(x, Y) = Y \cup \{x\},$$
$$DELETE(x, Y) = Y - \{x\};$$

Le arietà delle operazioni $MEMBER$, $INSERT$, $DELETE$, $\emptyset$, a_k sono rispettivamente $(12, 3)$, $(12, 2)$, $(12, 2)$, $(\varepsilon, 2)$, $(\varepsilon, 1)$. ∎

Esempio 9.2 [PARTI DI A ORDINATO (PAO)]

PAO $= \langle [A, SUBSET(A), BOOL], [MEMBER, INSERT, DELETE, MIN,$
$\emptyset, a_1, \ldots, a_m] \rangle$
dove:

- $\langle A = \{a_1, a_2, \ldots, a_m\}, \leq \rangle$ è un insieme totalmente ordinato, $SUBSET(A)$ è la famiglia di sottoinsiemi di A con $\emptyset$ insieme vuoto, $BOOL = \{vero, falso\}$;
- $MEMBER, INSERT, DELETE$ sono definite come nell'insieme precedente, $MIN : SUBSET(A) \to A$ è l'operazione di arietà (2,1) con $MIN(Y) = min\{x | x \in Y\}$.

∎

Esempio 9.3 [PARTIZIONI DI A]

PARTIZIONI DI A$= \langle [A, PART(A)], [UNION, FIND, ID, a_1, \ldots, a_m] \rangle$
dove:

- $A = \{a_1, a_2, \ldots, a_m\}$ è un insieme, $PART(A)$ è la famiglia delle partizioni di A (o equivalentemente delle relazioni di equivalenza su A) dove ID è la partizione identità $ID = \{\{a_1\}, \{a_2\}, \ldots, \{a_m\}\}$
- $UNION : A \times A \times PART(A) \to PART(A)$,
  $FIND : A \times PART(A) \to A$
  sono le operazioni:

  $UNION(x, y, P) =$ partizione ottenuta da P facendo l'unione delle classi di equivalenza contenenti x e y;
  $FIND(x, P) =$ elemento rappresentativo della classe di equivalenza in P contenente x. Osserva che, se x e y appartengono alla stessa classe di equivalenza in P, allora $FIND(x, P) = FIND(y, P)$.

  La arietà delle operazioni $UNION$, $FIND$ sono rispettivamente (112,2), (12,1).

∎

Esempio 9.4 [PILE DI A]

PILE DI A $= \langle [A,\ STACK(A),\ BOOL],\ [ISEMPTY,\ PUSH,\ POP,\ TOP,$
$\Lambda,\ a_1, \ldots,\ a_m] \rangle$
dove:

- $A = \{a_1, \ldots, a_m\}$ è un insieme, $STACK(A)$ è l'insieme delle sequenze finite $[a_{k(1)}, \ldots, a_{k(n)}]$ di elementi di A, compresa la sequenza vuota $\Lambda = \{\ \}$.
- $ISEMPTY : STACK(A) \to BOOL$,
  $PUSH : STACK(A) \times A \to STACK(A)$,
  $POP : STACK(A) \to STACK(A)$,
  $TOP : STACK(A) \to A$
  sono le operazioni:

  $$ISEMPTY(S) = \begin{cases} \text{vero} & \text{se } S = \Lambda \\ \text{falso} & \text{altrimenti} \end{cases}$$
  $PUSH([a_{k(1)}, \ldots, a_{k(n)}], a)$ =$[a_{k(1)}, \ldots, a_{k(n)}, a]$
  $POP([a_{k(1)}, \ldots, a_{k(n)}])$ =$[a_{k(1)}, \ldots, a_{k(n-1)}]$
  $TOP([a_{k(1)}, \ldots, a_{k(n)}])$ =$a_{k(n)}$

  Le arietà di $ISEMPTY$, $PUSH$, POP, TOP, Λ, a_k sono rispettivamente (2,3), (21, 2), (2,2), (2,1) (ε,2) , (ε,1). Le operazioni POP e TOP non sono definite su Λ.

∎

Siano ora date due algebre $A = \langle [A_1, A_2, \ldots, A_n], [f_1, \ldots, f_s] \rangle$ e $B = \langle [B_1, B_2, \ldots, B_n], [g_1, \ldots, g_s] \rangle$, dove f_i e g_i hanno la stessa arietà per ogni i.

Un omomorfismo $\phi : A \to B$ è una famiglia di funzioni $\phi_j : A_j \to B_j$ $(1 \leq j \leq n)$ tale che, per ogni indice i, se l'operazione f_i ha arietà $(k(1)k(2)\ldots k(s), l)$ vale:

$$\phi_l(f_i(x_{k(1)}, \ldots, x_{k(s)})) = g_i(\phi_{k(1)}(x_{k(1)}), \ldots, \phi_{k(s)}(x_{k(s)}))$$

Se inoltre le funzioni ϕ_i sono corrispondenze biunivoche, diremo che ϕ è un isomorfismo e che A è isomorfo a B, scrivendo $A \cong B$.

Esempio 9.5
Siano date le algebre $Z = \langle [\{\ldots, -1, 0, 1, \ldots\}], [+, \cdot] \rangle$ e $Z_p = \langle [\{0,1, \ldots,\text{p-1}\}], [+, \cdot] \rangle$, dove $+$ e $\cdot$ denotano (ambiguamente) l'usuale somma e prodotto e la somma e prodotto modulo p. La trasformazione $\langle \cdot \rangle_p : Z \to Z_p$ che associa ad ogni intero z il resto della divisione per p è un omomorfismo di algebre. ∎

Esempio 9.6
Consideriamo l'algebra PARTIZIONI DI A prima definita e l'algebra
FORESTE SU A:

$$\text{FORESTE SU A} = \langle [A, FOR(A)], [UNION, FIND, ID, a_1, \ldots, a_m] \rangle$$

dove:

- $A = \{a_1, \ldots, a_m\}$ è un insieme; $FOR(A)$ è la famiglia delle foreste con vertici in A, in cui ogni albero ha una radice; ID è la foresta in cui ogni vertice è un albero.
- $UNION : A \times A \times FOR(A) \to FOR(A)$,
  $FIND : A \times FOR(A) \to A$
  sono le operazioni:

  $UNION(x, y, F)$ = foresta ottenuta da F collegando la radice dell'albero che contiene il nodo x alla radice dell'albero che contiene il nodo y, identificando quest'ultima come radice del nuovo albero ottenuto.
  $FIND(x, F)$ = radice dell'albero in F contenente il vertice x.

È facile verificare che la funzione identità $I : A \to A$ e la funzione $\phi : FOR(A) \to PART(A)$ che ad ogni foresta F associa la partizione P in cui ogni classe è formata dai vertici di un albero in F realizzano un omomorfismo tra FORESTE SU A e PARTIZIONI DI A. Osserviamo inoltre che l'omomorfismo è una corrispondenza biunivoca se ristretta alle costanti, che sono $[ID, a_1, \ldots, a_m]$ nel caso di FORESTE SU A e [ID, $a_1, \ldots, a_m$] nel caso di PARTIZIONI SU A. ∎

9.2 Programmi astratti e loro implementazioni

Molti problemi incontrati in pratica si possono ridurre a sottoproblemi, ognuno dei quali può essere astrattamente formulato come sequenza di operazioni su una struttura di dati. Questo porta in modo naturale alla seguente:

Definizione 9.2 *Fissata una struttura di dati $A = \langle [A_1, A_2, \ldots, A_n], [f_1, \ldots, f_s] \rangle$, diremo che un* programma astratto *S su A è una sequenza $S_1; S_2, \ldots, S_h$ in cui S_i è una istruzione del tipo*

$$X_i := f_k(Z_1, \ldots, Z_s)$$

dove $Z_1, \ldots, Z_s$ sono costanti o variabili in $\{X_1, \ldots, X_{i-1}\}$. Naturalmente ogni variabile sarà di un dato tipo, e se $(k(1)k(2)\ldots k(s), l)$ è l'arietà di f_k allora X_i è di tipo l, Z_1 di tipo $k(1)$, …, Z_s di tipo $k(s)$.

Dato il programma S sopra definito, possiamo assegnare ad ogni variabile X_i un elemento $res_A(X_i)$ dell'algebra A, definito induttivamente nel modo seguente:

$res_A(a) = a$	per ogni costante a,
$res_A(X_i) = f_k(res_A(Z_1), \ldots, res_A(Z_s))$	se la i-esima istruzione in S e' del tipo $X_i := f_k(Z_1, \ldots, Z_s)$.

Osserva che la definizione è ben posta poiché in S ogni variabile X_i dipende solo dalle precedenti $X_1, X_2, \ldots, X_{i-1}$. Diremo inoltre che il *risultato* del programma $S = (S_1; S_2; \ldots; S_h)$ è l'elemento $res_A(X_h)$.

Un concetto importante è quello di implementazione concreta di una struttura dati astratta; in questo caso la nozione algebrica cruciale è quella di omomorfismo biunivoco sulle costanti.

Diciamo che un omomorfismo $\phi : A \to B$ tra due algebre A e B è *biunivoco sulle costanti* se la restrizione di ϕ sull'insieme delle costanti è una corrispondenza biunivoca.

Vale il seguente:

Teorema 9.1 *Dato un programma S, due algebre A e B e un omomorfismo $\phi : A \to B$ biunivoco sulle costanti, se α è il risultato di S su A e se β è il risultato di S su B allora vale che $\beta = \phi(\alpha)$.*

Dimostrazione. Poiché per ipotesi le costanti sono in corrispondenza biunivoca, basta dimostrare per induzione che $\phi(res_A(X_i)) = res_B(X_i)$ per ogni variabile X_i.
Infatti, se l'istruzione i-esima di S è della forma $X_i := f_k(Z_1, \ldots, Z_s)$ abbiamo:

$$\begin{aligned}
\phi(res_A(X_i)) &= \phi(f_k(res_A(Z_1), \ldots, res_A(Z_s))) \\
&= f_k(\phi(res_A(Z_1)), \ldots, \phi(res_A(Z_s))) && \text{per definizione di omomorfismo} \\
&= f_k(res_B(Z_1), \ldots, res_B(Z_s))) && \text{per ipotesi di induzione} \\
&= res_B(X_i)
\end{aligned}$$

∎

Osserviamo come conseguenza che se due algebre A e B sono isomorfe, allora i risultati dell'esecuzione di un programma S sulle due algebre sono corrispondenti nell'isomorfismo; da questo punto di vista algebre isomorfe risultano implementazioni equivalenti della struttura di dati (si parla di dati astratti, cioè definiti a meno di isomorfismo).

Se vogliamo implementare un programma (astratto) S su una macchina RAM, è necessario che:

1. Ad ogni operazione dell'algebra A sia associata una procedura che opera su strutture di dati efficientemente implementabili su RAM (vettori, liste, alberi); si ottiene in tal modo una nuova algebra B;

2. L'implementazione deve essere corretta; per quanto visto questo è vero se possiamo esibire un omomorfismo $\phi : B \to A$ biunivoco sulle costanti;

3. L'implementazione deve essere efficiente. È possibile che implementazioni "naturali" per una operazione siano fortemente penalizzanti per le altre; in generale sarà necessario scegliere l'algebra B con quella giusta ridondanza che mantenga bilanciato il costo della implementazione per tutte le operazioni.

Per quanto detto, il costo dell'esecuzione su RAM di un programma (astratto) S su una algebra A dipende dalla particolare implementazione. Ci limitiamo nel nostro contesto ad identificare il costo con la "complessità on-line", in cui ci si riferisce esclusivamente alla esecuzione on-line di S, dove cioè l'esecuzione della i-esima istruzione deve essere fatta senza conoscere le seguenti.

Concludiamo osservando che alcune implementazioni non sono realizzabili se il numero di costanti dell'algebra è alto. Chiariamo questo fatto su un esempio specifico: l'implementazione di $PARTI\ DI\ A$. Sia $A = \{e_1, \cdots, e_m\}$; una semplice rappresentazione di un sottoinsieme X di A è ottenuta dal suo vettore caratteristico V_X, dove V_X è un vettore di m bit tale che

$$V_X[k] = \begin{cases} 1 & \text{se } e_k \in X \\ 0 & \text{altrimenti} \end{cases}$$

L'implementazione delle operazioni $MEMBER$, $INSERT$, $DELETE$, è immediata ed un programma astratto di lunghezza n è eseguibile in tempo ottimale $O(n)$; tuttavia dobbiamo definire e mantenere in memoria RAM un vettore di dimensione m, cosa che risulta impraticabile in molte applicazione.

Esempio 9.7
Un compilatore deve tener traccia in una tabella di simboli di tutti gli identificatori presenti sul programma che deve tradurre. Le operazioni che il compilatore esegue sulla tabella sono di due tipi:

1. Inserimento di ogni nuovo identificatore incontrato, con eventuali informazioni (esempio: il tipo);
2. Richiesta di informazioni su un identificatore (esempio: il tipo).

È quindi richiesta una implementazione efficiente delle operazioni $INSERT$, $MEMBER$ su $PARTI\ DI$ "Identificatori"; la rappresentazione di un insieme di identificatori mediante il suo vettore caratteristico risulta impraticabile perché il numero di possibili identificatori in un linguaggio di programmazione è generalmente elevatissimo (nel C sono possibili $27 \cdot 30^{37}$ identificatori). ∎

9.3 Implementazione di dizionari mediante "Hashing"

Molti problemi possono essere modellati in modo naturale facendo riferimento all'algebra *PARTI DI* A; questa struttura viene chiamata "dizionario". Obbiettivo di questa sezione è di presentare le idee di base per l'implementazione di un dizionario con tecniche di "Hashing".

Una funzione di hashing per A è una funzione $h : A \to \{0, 1, 2, \ldots, m-1\}$; qui supponiamo che h possa essere calcolata in tempo costante. Ogni elemento che implementa $PARTI\ DI\ A$ è descritto da un vettore $(V[0], \ldots, V[m-1])$ dove, per ogni k, $V[k]$ è un puntatore a una lista $L_k =< e_{k1}, \cdots, e_{ks_k} >$ di elementi di A, con la richiesta ulteriore che $h(e_{kj}) = k \ \ (1 \leq j \leq s_k)$; in tal modo si individua univocamente il sottoinsieme $X \subseteq A$ formato da tutti gli elementi presenti nelle liste $L_0, L_1, \cdots, L_{m-1}$.

La procedura che implementa $INSERT(a, X)$ (risp. $DELETE(a, X)$, risp. $MEMBER(a, X)$) può essere così descritta:

1. Calcola $h(a)$.
2. Appendi a alla lista $L_{h(a)}$ (risp. cancella a dalla lista $L_{h(a)}$, risp. controlla se a è nella lista $L_{h(a)}$).

L'analisi rispetto al "caso peggiore" non è molto incoraggiante. Supponiamo infatti di avere un programma S che consiste in n istruzioni $INSERT$; può succedere che h associ lo stesso valore (per esempio 0) a tutti gli n elementi inseriti, ed in tal caso il numero di operazioni elementari effettuate sulla lista puntata da $V[0]$ è $\sum_{k=1}^{n} k = \Theta(n^2)$.

L'implementazione precedente risulta invece interessante per il "caso medio", nell'ipotesi che $h(x)$ assuma con uguale probabilità un qualsiasi valore in $\{0, 1, \cdots, m-1\}$ per un qualsiasi elemento da inserire. Allora per ogni programma S con n istruzioni la lunghezza media di ogni lista è al più n/m e il tempo medio risulta quindi $O(\frac{n}{m} \cdot n)$.

Se S è conosciuto prima dell'esecuzione (caso off-line), scegliendo $m = |S|$ (ovvero $m = n$), si ottiene un tempo medio ottimale $O(n)$.

Nel caso di algoritmi on-line, non si può assumere a priori la conoscenza della lunghezza del programma S. Si può allora procedere come segue:

1. Fisso un intero $m \geq 1$ e costruisco una tabella di hash T_0 di m elementi.
2. Quando il numero di elementi inseriti supera m, costruisco una nuova tabella T_1 di dimensione $2 \cdot m$ e ridistribuisco gli elementi secondo T_1; quando il numero di elementi supera $2 \cdot m$ costruisco una nuova tabella T_2 di dimensione $2^2 \cdot m$ ridistribuendo gli elementi secondo T_2, e così via costruendo tabelle T_k di dimensione $2^k \cdot m$, fino ad avere esaurito tutte le istruzioni.

Fissiamo $m = 1$ per semplicità di analisi. Poiché il tempo medio per ridistribuire gli elementi dalla tabella T_{k-1} nella tabella T_k è $O(2^k)$, il tempo complessivo necessario alle varie ridistribuzioni è $O(1 + 2 + \cdots + 2^M)$, essendo $2^M \leq n = |S|$. Poiché $1 + 2 + \cdots + 2^M = 2 \cdot 2^M - 1 = O(2^M)$, tale tempo è $O(n)$. Poiché inoltre ogni lista ha lunghezza aspettata al più 1, l'esecuzione di ogni istruzione in S costa mediamente $O(1)$. In conclusione:

Proposizione 9.2 *L'implementazione con tecnica di hashing di un programma S su un dizionario ha un tempo medio di calcolo $O(|S|)$.*

9.4 Alberi di ricerca binaria

Descriviamo ora una struttura utile per implementare, con buona efficienza nel caso medio, programmi astratti sulla struttura dati PARTI DI A ORDINATO. I sottoinsiemi di A vengono qui rappresentati mediante alberi di ricerca binaria. Le operazioni che dobbiamo implementare sono quelle di MIN, MEMBER, INSERT e DELETE.

Ricordiamo innanzitutto che A è un insieme totalmente ordinato e denotiamo con $\leq$ la relativa relazione d'ordine. Dato un sottoinsieme $S \subseteq A$, un *albero di ricerca binaria* per S è un albero binario T, i cui vertici sono etichettati da elementi di S, che gode delle seguenti proprietà. Denotando con $E(v)$ l'etichetta di ogni vertice v, abbiamo:

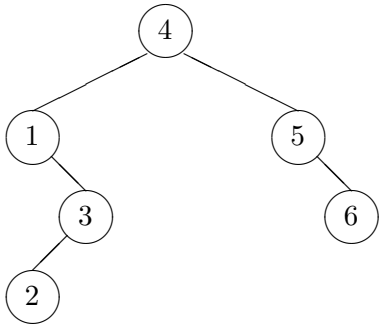
1. per ogni elemento $s \in S$, esiste uno e un solo vertice v in T tale che $E(v) = s$;

2. per ogni vertice v, se u è un vertice del sottoalbero sinistro di v allora $E(u) < E(v)$;

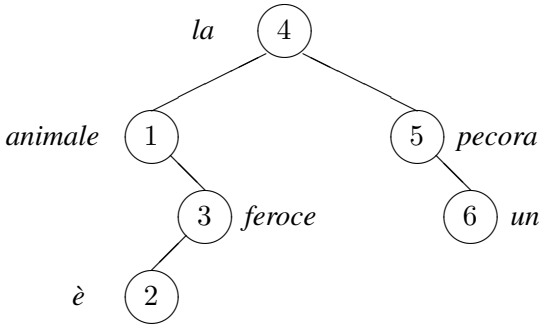
3. per ogni vertice v, se u è un vertice del sottoalbero destro di v allora $E(u) > E(v)$.

Dato $S = \{s_1, s_2, \ldots, s_n\}$, ordinando i suoi elementi in ordine crescente otteniamo un vettore $S = (S[1], S[2], \ldots, S[n])$; è facile dimostrare per induzione che un albero binario T di n nodi diventa un albero di ricerca binaria per S se, per ogni $k = 1, 2, \ldots, n$, si etichetta con $S[k]$ il k-esimo vertice visitato durante l'attraversamento in ordine simmetrico di T.

Per esempio, sia A l'insieme delle parole della lingua italiana ordinate in modo lessicografico e sia S l'insieme { *la, pecora, è , un, animale, feroce* }; tale insieme è rappresentato dal vettore (*animale, è , feroce, la, pecora, un*). Nel seguente albero binario di 6 nodi i vertici sono numerati secondo l'ordine simmetrico:



Il seguente risulta allora un albero di ricerca binaria per S:



Possiamo rappresentare un albero di ricerca binaria mediante le tabelle *sin*, *des*, E e *padre*. Nel nostro caso otteniamo la seguente rappresentazione:

vertici	*sin*	*des*	E	*padre*
1	0	3	*animale*	4
2	0	0	*e'*	3
3	2	0	*feroce*	1
4	1	5	*la*	0
5	0	6	*pecora*	4
6	0	0	*un*	5

È chiaro che, per ogni nodo v di T, i nodi con etichetta minore di $E(v)$ si trovano eventualmente nel sottoalbero sinistro di v. Quindi, possiamo determinare il vertice con etichetta minima semplicemente scorrendo il ramo sinistro dell'albero. Il procedimento è descritto dalla seguente procedura nella quale rappresentiamo con r la radice dell'albero.

```
Procedure MIN(r)
begin
      v := r
      while sin(v) ≠ 0 do v := sin(v)
      return v
end
```

Analogamente, l'operazione MEMBER(x, S) può essere eseguita mediante la seguente procedura ricorsiva CERCA(x, v) che verifica se nel sottoalbero avente per radice v esiste un nodo di etichetta x.

```
Procedure CERCA(x, v)
begin
      if x := E(v) then return vero
      if x < E(v) then if sin(v) ≠ 0 then return CERCA(x, sin(v))
                                     else return falso
      if x > E(v) then if des(v) ≠ 0 then return CERCA(x, des(v))
                                     else return falso
end
```

L'appartenenza di un elemento all'insieme S rappresentato dall'albero di ricerca binaria T può essere allora verificata semplicemente mediante la chiamata CERCA(x, r).

Per quanto riguarda l'operazione INSERT(x, S), osserviamo che se x appartiene a S l'insieme non viene modificato. Se invece S è rappresentato dall'albero di ricerca binaria T e $x \notin S$, l'algoritmo deve inserire in T un nuovo nodo etichettato con x, preservando la struttura di albero di ricerca binaria. In questo modo l'algoritmo restituisce un albero di ricerca binaria per l'insieme $S \cup \{x\}$. Il procedimento è ottenuto mediante la seguente procedura ricorsiva che eventualmente inserisce un nodo etichettato x nel sottoalbero di radice v.

```
Procedure INSERT(x, v)
begin
      if x < E(v) then if sin(v) ≠ 0 then INSERT(x, sin(v))
                                     else CREA_NODO_SIN(v, x)
      if x > E(v) then if des(v) ≠ 0 then INSERT(x, des(v))
                                     else CREA_NODO_DES(v, x)
end
```

Qui la procedura CREA_NODO_SIN(v, x) introduce un nuovo nodo $\hat{v}$, figlio sinistro di v ed etichettato con x, aggiornando la tabella come segue:

$$sin(v) := \hat{v}; \quad padre(\hat{v}) := v; \quad E(\hat{v}) := x; \quad sin(\hat{v}) := 0; \quad des(\hat{v}) := 0$$

La procedura CREA_NODO_DES(v, x) è definita in maniera analoga.

Osserviamo che l'implementazione delle operazioni MIN, DELETE e INSERT non richiedono l'uso della tabella $padre(v)$; tale tabella risulta invece indispensabile per una efficiente implementazione dell'operazione DELETE.

Una possibile implementazione di DELETE(x, S), dove S è rappresentato da un albero di ricerca binaria T, richiede l'esecuzione dei seguenti passi:

1. Determina l'eventuale nodo v tale che $x = E(v)$.

2. Se v possiede al più un figlio, allora:

   a) se v è una foglia basta togliere v dall'albero, ovvero eliminare la riga corrispondente nella tabella aggiornando opportunamente quella relativa al padre;

   b) se v è radice e ha un unico figlio $\hat{v}$ basta ancora togliere v dall'albero e assegnare 0 a $padre(\hat{v})$ rendendo così $\hat{v}$ radice dell'albero;

   c) se v non è radice e ha un unico figlio $\hat{v}$ basta togliere v dall'albero collegando direttamente $\hat{v}$ con $padre(v)$. Naturalmente se v era figlio sinistro si pone $sin(padre(v)) := \hat{v}$, mentre se v era figlio destro si pone $des(padre(v)) := \hat{v}$.

   Nel seguito indichiamo con TOGLI(v) la procedura che esegue i calcoli relativi al punto 2. È chiaro che se v possiede al più un figlio l'esecuzione di TOGLI(v) permette di ottenere un albero di ricerca binaria per $S - \{x\}$.

3. Se v ha due figli si può determinare il vertice v_M che ha etichetta massima nel sottoalbero di sinistra di v. Poiché v_M non ha figlio destro, associando a v l'etichetta di v_M e applicando la procedura TOGLI(v_M) si ottiene un albero di ricerca per $S - \{x\}$.

Il passo 3) richiede il calcolo del vertice di etichetta massima nel sottoalbero che ha per radice un nodo qualsiasi u. Questo può essere determinato scorrendo il ramo di destra dell'albero, come descritto nella seguente procedura.

```
Procedure MAX(u)
begin
      v := u
      while des(u) ≠ 0 do v := des(v)
      return v
end
```

La procedura principale per l'implementazione dell'operazione DELETE è allora la seguente:

```
Procedure ELIMINA(x, v)
begin
      if x < E(v) ∧ sin(v) ≠ 0 then ELIMINA(x, sin(v))
      if x > E(v) ∧ des(v) ≠ 0 then ELIMINA(x, des(v))
      if x = E(v)              then  if v ha al piu' un figlio then TOGLI(v)
                                                               else
                                                                    begin
                                                                          v_M := MAX(sin(v))
                                                                          E(v) := E(v_M)
                                                                          TOGLI(v_M)
                                                                    end
end
```

L'operazione DELETE(x, S), dove S è un insieme rappresentato da un albero di ricerca binaria T, con radice r, può quindi essere implementata dalla semplice chiamata ELIMINA(x, r), dove r è la radice di T.

Le procedure qui delineate MEMBER, INSERT, MIN, e DELETE richiedono, se applicate a un albero di altezza h, un tempo di calcolo $O(h)$; ricordiamo che un albero binario di n nodi ha un'altezza h tale che $\log n \leq h \leq n-1$. Poiché applicando all'insieme vuoto un programma astratto di n operazioni MEMBER, INSERT, MIN e DELETE si ottengono alberi di al più n elementi, ne segue che l'esecuzione di un tale programma richiede tempo $O(n^2)$. Purtroppo questo limite superiore è stretto: se infatti, in un albero di ricerca binaria inizialmente vuoto, inseriamo consecutivamente n elementi $a_1, a_2, \ldots, a_n$ tali che $a_1 < a_2 < \cdots < a_n$, si eseguono $\sum_{k=0}^{n-1} k \sim \frac{n^2}{2}$ operazioni di confronto.

Fortunatamente le prestazioni nel caso medio risultano migliori. Consideriamo infatti n elementi $a_1, a_2, \ldots, a_n$ tali che $a_1 < a_2 < \cdots < a_n$ e supponiamo di voler inserire tali elementi in un ordine qualsiasi nell'ipotesi che tutti i possibili ordinamenti siano equiprobabili. Più precisamente scegliamo una permutazione casuale $(a_{p(1)}, a_{p(2)}, \ldots, a_{p(n)})$ degli n elementi assumendo che tutte le $n!$ permutazioni abbiano la stessa probabilità di essere scelte. Consideriamo quindi il programma INSERT$(a_{p(1)}, S)$, INSERT$(a_{p(2)}, S)$, . . . , INSERT$(a_{p(n)}, S)$ e denotiamo con T_n il numero medio di confronti necessari per eseguire tale programma. Osserviamo innanzitutto che, per ogni $k \in \{1, 2, \ldots, n\}$, $a_{p(1)} = a_k$ con probabilità $1/n$. In questo caso l'albero di ricerca ottenuto alla fine dell'esecuzione dell'algoritmo avrà la radice etichettata da a_k, mentre il sottoalbero di sinistra della radice conterrà $k-1$ elementi e quello di destra $n-k$. Questo significa che durante la costruzione dell'albero, il valore $a_{p(1)}$ sarà confrontato con tutti gli altri elementi della sequenza, per un totale di $n-1$ confronti; inoltre, eseguiremo in media T_{k-1} confronti per costruire il sottoalbero di sinistra e T_{n-k} per costruire quello di destra. Di conseguenza, nel caso $a_{p(1)} = a_k$, si eseguono $n - 1 + T_{k-1} + T_{n-k}$ confronti. Poiché questo evento si verifica con probabiltà $1/n$ per ogni k, otteniamo la seguente equazione, valida per ciascun $n > 1$:

$$T_n = \sum_{k=1}^{n} \frac{1}{n}(n - 1 + T_{k-1} + T_{n-k}).$$

Mediante semplici passaggi questa equazione si riduce alla ricorrenza

$$T_n = \begin{cases} 0 & \text{se } n = 1 \\ \sum_{k=1}^{n} \frac{1}{n}(n - 1 + T_{k-1} + T_{n-k}) & \text{altrimenti} \end{cases}$$

che coincide con quella ottenuta nell'analisi di Quicksort, studiata nelle sezioni 6.6.1 e 7.4.2. Si ottiene in questo modo il valore $T_n = 2n \log n + O(n)$.

Possiamo allora concludere affermando che, nel caso medio, n operazioni MEMBER, INSERT e MIN eseguite su un albero di ricerca binaria inizialmente vuoto, richiedono $O(n \log n)$ passi.

Esercizi

1) Le procedure CERCA, INSERT e ELIMINA appena descritte sono procedure risorsive. Quali di queste forniscono un esempio di ricorsione terminale?

2) Descrivere una versione iterativa delle procedure CERCA, INSERT e ELIMINA.

3) Descrivere un algoritmo per ordinare un insieme di elementi S assumendo in input un albero di ricerca binaria per S. Mostrare che, se S possiede n elementi, l'algoritmo funziona in tempo $\Theta(n)$ (si assuma il criterio uniforme).

4) Usando gli alberi di ricerca binaria (e in particolare la procedura di inserimento) si descriva un algoritmo di ordinamento generico. Svolgere l'analisi dell'algoritmo nel caso peggiore e in quello medio (nelle solite ipotesi di equidistribuzione) e confrontare i risultati con quelli relativi a Quicksort.

5) Consideriamo l'algoritmo che su input $n \in \mathbb{N}$, $n > 2$, introduce in un albero di ricerca binaria inizialmente vuoto la seguente sequenza di numeri interi, utilizzando la tradizionale procedura di inserimento:

$$1, 2n, 2, 2n-1, 3, 2n-2, \ldots, n, n+1$$

Descrivere l'albero ottenuto assumendo il tradizionale ordinamento sugli interi. Inoltre, assumendo il criterio uniforme, valutare in funzione di n l'ordine di grandezza del tempo di calcolo richiesto dall'algoritmo.

9.5 Alberi 2-3

Abbiamo precedentemente osservato che l'implementazione di PARTI DI A ORDINATO mediante alberi di ricerca binaria, permette di eseguire programmi astratti di n istruzioni in tempo medio $O(n \log n)$; tuttavia, nel caso peggiore, si ottengono tempi di calcolo $\Theta(n^2)$. In questa sezione presentiamo una struttura che permette di eseguire lo stesso calcolo in tempo $O(n \log n)$ anche nel caso peggiore.

Osserviamo innanzitutto che, usando strutture ad albero per implementare PARTI DI A ORDINATO, il tempo di esecuzione di una operazione MEMBER dipende essenzialmente dalla profondità del nodo cui è applicata. Ricordiamo che ogni albero con n nodi, nel quale ogni vertice possiede al più k figli, ha altezza maggiore o uguale a $\log_k n$. Quindi, per eseguire in maniera sempre efficiente l'operazione MEMBER, è necessario che l'albero abbia una altezza prossima a $\log_k n$. Per questo motivo chiameremo informalmente *bilanciati* gli alberi con radice di altezza $O(\log n)$, dove n è il numero dei nodi. È allora chiaro che ogni efficiente implemetazione di PARTI DI A ORDINATO, basata su alberi, richiederà l'utilizzo di alberi bilanciati. A tal fine risultano critiche le procedure INSERT e DELETE: esse dovranno preservare la proprietà di bilanciamento degli alberi trasformati.

La famiglia di alberi bilanciati che prendiamo in considerazione in questa sezione è quella degli alberi 2-3. Un albero 2-3 è un albero ordinato in cui ogni nodo che non sia foglia possiede 2 o 3 figli e nel quale tutte le foglie hanno uguale profondità. Nel seguito, per ogni nodo interno v di un albero 2-3, denoteremo con $f_1(v)$, $f_2(v)$, $f_3(v)$ rispettivamente il primo, il secondo e l'eventuale terzo figlio di v.

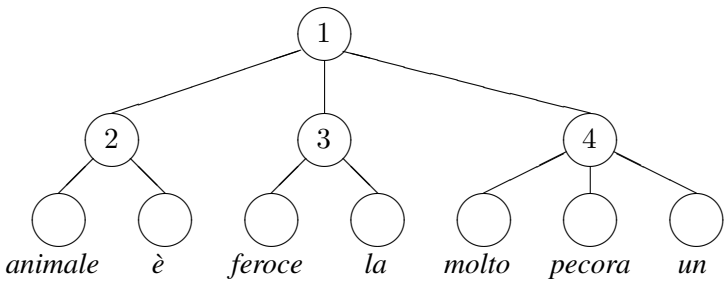
In un albero 2-3 con n nodi e altezza h vale evidentemente la relazione

$$\sum_{k=0}^{h} 2^k \leq n \leq \sum_{k=0}^{h} 3^k$$

e quindi $2^{h+1} - 1 \leq n \leq (3^{h+1} - 1)/2$. Questo implica che $\lfloor \log_3 n \rfloor - 1 \leq h \leq \lceil \log_2 n \rceil$ e di conseguenza gli alberi 2-3 risultano bilanciati nel nostro senso.

Consideriamo ora un insieme $\{a_1, a_2, \ldots, a_n\} \subseteq A$ tale che $a_1 < a_2 < \cdots < a_n$. Rappresentiamo tale insieme mediante un albero 2-3 le cui foglie, da sinistra a destra, sono identificate rispettivamente dai valori $a_1, a_2, \ldots, a_n$. Invece i nodi interni dell'albero contengono le informazioni necessarie alla ricerca degli elementi a_i: per ogni nodo interno v denotiamo con $L(v)$ e $M(v)$ rispettivamente, la massima foglia del sottoalbero di radice $f_1(v)$ e la massima foglia del sottoalbero di radice $f_2(v)$.

Per esempio, si consideri l'insieme di parole { *la, pecora, è , un, animale, molto, feroce* } dotato dell'ordinamento lessicografico. Tale insieme può essere implementato dal seguente albero 2-3, al quale si devono aggiungere le informazioni relative ai valori L e M.



Tale albero è rappresentato dalla corrispondente tabella, nella quale riportiamo solo i valori relativi ai nodi interni, dotata delle ulteriori informazioni L e M.

vertici	f_1	f_2	f_3	L	M
1	2	3	4	*e'*	*la*
2	*animale*	*e'*	0	*animale*	*e'*
3	*feroce*	*la*	0	*feroce*	*la*
4	*molto*	*pecora*	*un*	*molto*	*pecora*

Osserviamo che ogni insieme con almeno due elementi può essere rappresentato da alberi 2-3, poiché l'equazione $3x + 2y = n$ ammette soluzioni intere non negative per ogni intero $n \geq 2$.

La procedura per calcolare l'elemento minimo di un insieme rappresentato da un albero 2-3 di radice r è immediata:

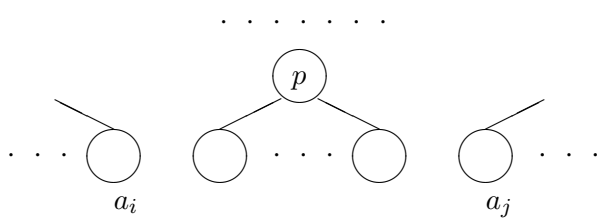
```
Procedure MIN(r)
begin
      v := r
      while f_1(v) ≠ 0 do v := f_1(v)
      return v
end
```

Inoltre, una procedura fondamentale per implementare le altre operazioni è del tutto simile all'algoritmo di ricerca binaria presentato nella sezione 5.2.1:

```
Procedure CERCA(a, v)
begin
      if f_1(v) e' una foglia then return v
      else if a ≤ L(v) then return CERCA(a, f_1(v))
      else if f_3(v) = 0 ∨ a ≤ M(v) then return CERCA(a, f_2(v))
                                    else return CERCA(a, f_3(v))
end
```

Sia $\{a_1, a_2, \ldots, a_n\}$ l'insieme rappresentato dall'albero, con $a_1 < a_2 < \cdots < a_n$. Per ogni valore $a \in A$ e ogni nodo interno v dell'albero, la procedura CERCA(a, v) restituisce un nodo interno p, del sottoalbero che ha per radice v, i cui figli sono foglie e che verifica la seguente proprietà:

> se a_i è l'eventuale elemento in $\{a_1, a_2, \ldots, a_n\}$ che precede il più piccolo figlio di p e a_j l'eventuale elemento di $\{a_1, a_2, \ldots, a_n\}$ che segue il più grande figlio di p, allora $a_i < a < a_j$.



Questo significa che se a appartiene all'insieme, cioè è foglia dell'albero con radice r, allora a è figlio di $p =$ CERCA(a, r). Di conseguenza la seguente procedura implementa l'operazione MEMBER:

```
Procedure MEMBER(a, r)
begin
      p := CERCA(a, r)
      if a e' figlio di p then return vero
                          else return falso
end
```

Una naturale implementazione di INSERT(a, r) (o DELETE(a, r)) richiede preliminarmente di calcolare p = CERCA(a, r), aggiungendo poi (o cancellando) il nodo a ai figli di p. Il problema è che se p ha 3 figli, l'inserimento di a ne crea un quarto e quindi l'albero ottenuto non è più un albero 2-3. Analogamente, se p ha due figli di cui uno è a, la cancellazione di a provoca la formazione di un nodo interno con un solo figlio, violando nuovamente la proprietà degli alberi 2-3.

Per risolvere questi problemi focalizziamo l'attenzione sull'operazione INSERT (l'operazione DELETE può essere trattata in maniera analoga). Definiamo anzitutto una procedura ausiliaria, denominata RIDUCI, che trasforma un albero con un vertice con 4 figli, e gli altri nodi interni con 2 o 3 figli, in un albero 2-3 con le stesse foglie. Questa procedura è descritta dal seguente schema nel quale si prevede l'uso esplicito della tabella $padre$:

```
Procedure RIDUCI(v)
if v ha 4 figli then
      begin
            crea un nodo v̂
            assegna a v̂ i primi due figli v1 e v2 di v
                  (aggiornando opportunamente i valori f1, f2, f3 di v e v̂
                  e i valori padre di v1 e v2)
            if v e' radice then { crea una nuova radice r̂
                                  f1(r̂) := v̂
                                  f2(r̂) := v
                                  aggiorna i valori padre di v e v̂
                        else      { u := padre(v)
                                  poni v̂ figlio di u immediatamente a sinistra di v
                                  RIDUCI(u)
      end
```

Possiamo a questo punto esibire la procedura INSERT(a, r) che implementa correttamente l'operazione di inserimento quando è applicata ad alberi 2-3 con almeno due foglie:

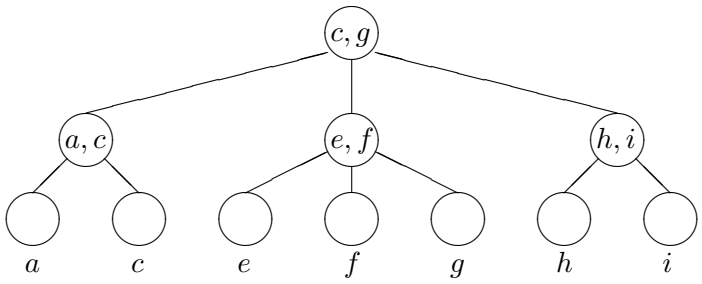
```
Procedure INSERT(a, r)
begin
      p := CERCA(a, r)
      if a non e' figlio di p then { aggiungi in ordine il nodo a ai figli di p
                                   RIDUCI(p)
end
```

Notiamo che le procedure appena descritte vanno opportunamente modificate in modo da aggiornare i valori delle tabelle L e M relative ai nodi che si trovano lungo il cammino da p alla radice e ai nuovi

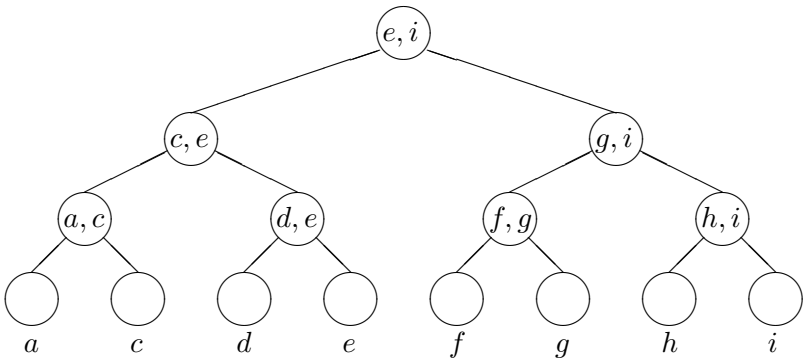
vertici creati dal processo di riduzione. È bene notare che l'aggiornamento delle tabelle L e M può giungere fino alla radice anche se il processo di creazione di nuovi nodi si arresta prima.

Concludiamo osservando che, nelle procedure descritte sopra, il numero di passi di calcolo è al più proporzionale all'altezza dell'albero considerato. Se quest'ultimo possiede n foglie il tempo di calcolo è allora $O(\log n)$ per ogni procedura.

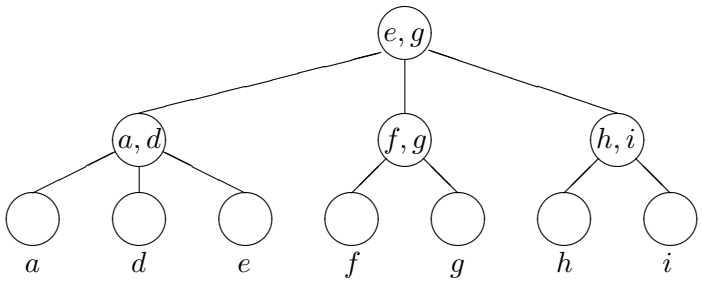
Esempio
L'albero 2-3 descritto nella figura seguente rappresenta l'insieme di lettere $\{a, c, e, f, g, h, i\}$ disposte in ordine alfabetico. Nei nodi interni sono stati posti i corrispondenti valori delle tabelle L e M.



Inserendo la lettera d otteniamo l'albero seguente:



Cancellando ora il nodo c otteniamo:



9.6 B-alberi

I B-alberi possono essere considerati come una generalizzazione degli alberi 2-3 descritti nella sezione precedente. Si tratta anche in questo caso di alberi bilanciati che consentono la rappresentazione di insiemi totalmente ordinati e permettono l'esecuzione di ciascuna operazione MEMBER, INSERT, DELETE e MIN in tempo $O(\log n)$ su ogni insieme di n elementi.

Fissiamo un intero $m \geq 2$. Allora un B-albero (o B-tree) di ordine m su un insieme S di n elementi (che nel seguito chiameremo *chiavi*) è un albero ordinato che gode delle seguenti proprietà:

1. ogni nodo interno possiede al più $2m$ figli;
2. ogni nodo interno possiede almeno m figli, escluso la radice che ne ha almeno 2;
3. gli elementi di S sono assegnati ai nodi interni dell'albero. Ciascuna chiave è assegnata a un solo nodo. Se la chiave a è assegnata al nodo v diremo anche che v contiene a;
4. tutte le foglie hanno la stessa profondità (e non contengono alcuna chiave);
5. ogni nodo interno v con $k+1$ figli $v_0, v_1, \ldots, v_k$ contiene k chiavi ordinate

$$a_1 < a_2 < \cdots < a_k$$

e inoltre :

a) ogni chiave a contenuta nel sottoalbero di radice v_0 soddisfa la relazione $a < a_1$,
b) ogni chiave a contenuta nel sottoalbero di radice v_i ($1 \leq i \leq k-1$) soddisfa la relazione $a_i < a < a_{i+1}$,
c) ogni chiave a contenuta nel sottoalbero di radice v_k soddisfa la relazione $a_k < a$.

L'intero m è anche chiamato *indice di ramificazione* dell'albero. Osserva che ogni nodo interno contiene almeno $m-1$ chiavi (esclusa la radice) e ne possiede al più $2m-1$. Inoltre, poiché il grado dei nodi può variare di molto, conviene rappresentare il B-albero associando a ogni nodo interno v con figli $v_0, v_1, \ldots, v_k$, $k+1$ puntatori $p_0, p_1, \ldots, p_k$, dove ogni p_i punta al nodo v_i.

È chiaro allora come utilizzare le chiavi contenute nei vari nodi per indirizzare la ricerca di un elemento nell'insieme S. Il metodo è del tutto analogo a quello descritto per gli alberi 2-3 ed è basato sulla seguente procedura nella quale x è una possibile chiave e r è la radice dell'albero. La principale differenza è che in questo caso la procedura non è ricorsiva.

```
Procedure CERCA(x, r)
begin
      v := r
      esci := 0
      repeat
            if v e' una foglia then esci := -1
            else
                begin
                      siano a_1, a_2, ..., a_k le chiavi ordinate contenute in v
                      siano f_0, f_1, ..., f_k i corrispondenti figli di v
                      i := 1
                      a_{k+1} := +∞ (valore convenzionale maggiore di ogni possibile chiave)
                      while x > a_i do i := i + 1
                      if x = a_i then esci := i
                                 else v := f_{i-1}
                end
      until esci ≠ 0
```

```
    if esci = −1  then return 0
                  else return (v, esci)
end
```

L'esecuzione di CERCA(x, r) restituisce (v, j) se x è la j-esima chiave del nodo v, altrimenti la procedura fornisce il valore 0. Quindi, la semplice chiamata di CERCA(x, r) permette di verificare se x appartiene all'insieme S mantenuto dall'albero considerato.

Descriviamo ora l'implementazione dell'operazione INSERT. Per inserire una nuova chiave x in un B-albero T occorre eseguire i seguenti passi:

1. svolgi la ricerca di x in T fino a determinare un nodo interno v i cui figli sono foglie;
2. colloca x tra le chiavi di v in maniera ordinata, creando un nuovo figlio di v (una foglia) da porre immediatamente a destra o a sinistra di x;
3. se ora v possiede $2m$ chiavi ordinate (e quindi $2m + 1$ figli) richiama la procedura RIDUCI(v) definita in seguito.

```
Procedure RIDUCI(v)
siano a_1, a_2, ..., a_2m le chiavi di v e siano f_0, f_1, ..., f_2m i suoi figli
if v possiede un fratello adiacente u con meno di 2m − 1 chiavi
    then { p := padre(v)
           y := chiave separatrice di u e v in p
           if u e' fratello maggiore di v
               then { cancella a_2m da v e inseriscilo in p al posto di y
                      inserisci y in u come chiave minima
                      rendi f_2m primo figlio di u togliendolo da v }
               else { cancella a_1 da v e inseriscilo in p al posto di y
                      inserisci y in u come chiave massima
                      rendi f_0 ultimo figlio di u togliendolo da v }
    else { crea un nuovo nodo u
           assegna a u le chiavi a_{m+1}, a_{m+2}, ..., a_2m e i figli f_m, f_{m+1}, ..., f_2m
                togliedoli da v
           if v e' radice
               then { crea una nuova radice r
                      assegna a_m a r togliendolo da v
                      rendi v e u figli di r con a_m come elemento separatore }
               else { p := padre(v)
                      assegna a_m a p togliendolo da v
                      rendi u figlio di p come minimo fratello maggiore di v
                           con a_m come elemento separatore
                      if p possiede ora 2m chiavi  then RIDUCI(p) }
```

L'operazione DELETE può essere implementata in maniera simile. In questo caso si deve verificare se il nodo al quale è stata tolta la chiave contiene almeno $m - 1$ elementi. Se non è questo il caso,

bisogna prelevare una chiave dai nodi vicini ed eventualmente dal padre, ripetendo quindi l'operazione di cancellazione su quest'ultimo.

La procedura che esegue la cancellazione di una chiave x da un B-albero T esegue i seguenti passi:

1. svolgi la ricerca di x in T fino a determinare il nodo interno v che contiene x;
2. se i figli di v non sono foglie esegui le seguenti istruzioni che riducono il problema alla cancellazine di una chiave che si trova in un nodo interno di profondità massima:

   `begin`
   sia f il figlio di v immediatamente a sinistra di x
   calcola la chiave massima z contenuta nel sottoalbero di radice f
   sia g il nodo che contiene z
   sostituisci x con z in v
   $x := z$
   $v := g$
   `end`
3. cancella x in v ed elimina uno dei figli adiacenti (una foglia);
4. se ora v possiede $m-2$ chiavi (e quindi $m-1$ figli) e v non è radice, allora richiama la procedura ESPANDI(v) definita in seguito.

`Procedure` ESPANDI(v)
`if` v e' radice
`then if` v non possiede chiavi `then` { elimina v ; rendi il figlio di v nuova radice }
`else`
`begin`
`if` v possiede un fratello adiacente u con almeno m chiavi
`then` {
$p :=$ padre(v)
$y :=$ chiave separatrice di u e v in p
`if` u e' fratello maggiore di v
`then` { togli da u la chiave minima e inseriscila in p al posto di y ; inserisci y in v come chiave massima ; rendi il primo figlio di u ultimo figlio di v }
`else` { togli da u la chiave massima e inseriscila in p al posto di y ; inserisci y in v come chiave minima ; rendi l' ultimo figlio di u primo figlio di v }
}
`else` {
$p :=$ padre(v)
sia u un fratello di v adiacente
sia y la chiave in p separatrice tra v e u
inserisci in v la chiave y mantenendo l'ordine
inserisci in v le chiavi di u mantenendo l'ordine
assegna a v i figli di u mantenendo l'ordine
cancella da p la chiave y e il figlio u
elimina il nodo u
`if` p possiede ora meno di $m-1$ chiavi (e quindi meno di m figli)
`then` ESPANDI(p)
}

```
end
```

L'efficienza di queste procedure dipende dall'altezza dell'albero. Per valutare tale quantità considera un B-albero di altezza h contenente n chiavi. Osserva che vi sono: un nodo a profondità 0, almeno 2 nodi a profondità 1, almeno $2m$ nodi a profondità 2, almeno $2m^2$ a profondità 3, e così via. La radice contiene almeno una chiave mentre gli altri nodi ne contengono almeno $m-1$ (escluse le foglie che non contengono chiavi). Di conseguenza il numero n di chiavi è maggiore o uguale a

$$1 + \left(\sum_{i=0}^{h-2} 2m^i \right) (m-1)$$

e, svolgendo semplici calcoli, otteniamo

$$h \leq 1 + \log_m \frac{n+1}{2}.$$

Osserva che, scegliendo m in maniera opportuna, si può ridurre di molto il valore dell'altezza dell'albero. Tuttavia, anche la scelta di un valore di m troppo elevato non è vantaggiosa poiché rende costosa la ricerca di un elemento all'interno di ciascun nodo.

Concludiamo questa sezione ricordando che in molti algoritmi è necessario implementare operazioni MIN, INSERT e DELETE su un dato insieme ordinato senza bisogno di eseguire la ricerca di un elemento qualsiasi (MEMBER). Questo si verifica quando le operazioni INSERT e DELETE possono essere eseguite senza effettivamente cercare l'elemento da inserire o da cancellare; nell'operazione INSERT questo significa che l'elemento da introdurre non è presente nella struttura mentre, nell'operazione DELETE, questo vuol dire che è nota la posizione dell'elemento da cancellare nella struttura (per esempio si tratta del minimo). Le strutture dati che eseguono le tre operazioni MIN, INSERT e DELETE in queste ipotesi sono chiamate *code di priorità* e possono essere viste come una semplificazione della struttura PARTI DI A ORDINATO.

Gli alberi binari, gli alberi 2-3 e i B-alberi possono essere utilizzati come code di priorità, perchè eseguono appunto le operazioni sopra considerate. Un'altra struttura che rappresenta un'efficiente implementazione di una coda di priorità è costituita dagli *heap rovesciati*, nei quali cioè il valore associato ad ogni nodo interno è minore di quello associato ai figli. Nota che in questo caso la radice contiene il valore minimo dell'insieme e quindi l'operazione MIN può essere eseguita in tempo costante. Le altre due invece richiedono un tempo $O(\log n)$, dove n è il numero degli elementi presenti nello heap.

9.7 Operazioni UNION e FIND

Una partizione di un insieme A è una famiglia $\{A_1, A_2, \ldots, A_m\}$ di sottoinsiemi di A, a due a due disgiunti (cioè $A_i \cap A_k = \emptyset$ per ogni $i \neq k$), che coprono A (ovvero tali che $A = A_1 \cup A_2 \cup \cdots \cup A_m$).

Il concetto di partizione è strettamente connesso a quello di relazione di equivalenza. Ricordiamo che una relazione di equivalenza su A è una relazione $R \subseteq A \times A$ che verifica le proprietà riflessiva, simmetrica e transitiva. In altre parole, per ogni $x, y, z \in A$, si verifica xRx, $xRy \Rightarrow yRx$, e $xRy \wedge yRz \Rightarrow xRz$ (qui xRy significa $(x, y) \in R$).

Come è noto, per ogni $a \in A$, la classe di equivalenza di a modulo R è l'insieme $[a]_R = \{x \in A \mid xRa\}$. È facile verificare che, per ogni relazione di equivalenza R su un insieme A, l'insieme delle classi

di equivalenza modulo R forma una partizione di A. Viceversa, ogni partizione $\{A_1, A_2, \ldots, A_m\}$ di A definisce automaticamente una relazione di equivalenza R su A: basta infatti definire xRy per tutte le coppie di elementi x, y che appartengono a uno stesso sottoinsieme A_i della partizione. Di conseguenza la partizione può essere agevolmente rappresentata da una n-pla di elementi $a_1, a_2, \ldots, a_m \in A$, dove ogni a_i appartiene a A_i; in questo modo $[a_i] = A_i$ per ogni $i = 1, 2, \ldots, m$ e a_i è chiamato *elemento rappresentativo* della sua classe di equivalenza.

Nell'esempio 9.3 abbiamo definito una struttura dati basata sulla nozione di partizione. Le operazioni fondamentali definite sulle partizioni sono le operazioni UNION e FIND. Data una partizione P di un insieme A e una coppia di elementi $x, y \in A$, abbiamo

$UNION(x, y, P) =$ partizione ottenuta da P facendo l'unione delle classi di equivalenza contenenti x e y;
$FIND(x, P) =$ elemento rappresentativo della classe di equivalenza in P contenente x.

Esempio 9.8
Consideriamo la famiglia delle partizioni dell'insieme $A = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$, convenendo di sottolineare gli elementi rappresentativi. Allora, se P è la partizione $\{\{1, 3, \underline{7}\}, \{\underline{2}\}, \{4, 5, 6, \underline{8}, 9\}\}$, abbiamo

$$\text{FIND}(4, P) = 8$$

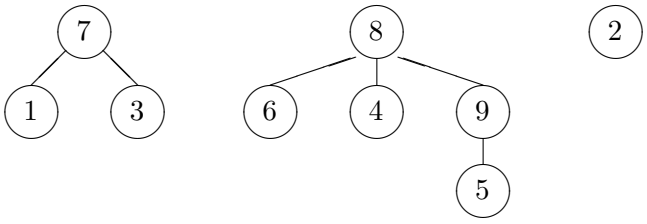
$$\text{UNION}(3, 2, P) = \{\{1, 2, 3, \underline{7}\}, \{4, 5, 6, \underline{8}, 9\}\}$$

∎

Descriviamo ora alcune possibili implementazioni di PARTIZIONI DI A mediante FORESTE SU A. La partizione $\{A_1, A_2, \ldots, A_m\}$ sarà rappresentata da una foresta composta da m alberi con radice $T_1, T_2, \ldots, T_m$ tali che ogni A_i è l'insieme dei nodi di T_i e la radice di T_i è l'elemento rappresentativo di A_i ($1 \leq i \leq m$).

Una foresta può essere facilmente rappresentata mediante una tabella $padre$.

Esempio 9.9
La partizione $\{\{1, 3, \underline{7}\}, \{\underline{2}\}, \{4, 5, 6, \underline{8}, 9\}\}$ può essere rappresentata dalla foresta



a sua volta descritta dalla seguente tabella

vertici	$padre$
1	7
2	0
3	7
4	8
5	9
6	8
7	0
8	0
9	8

Una semplice implementazione dell'operazione FIND consiste nel risalire dal nodo considerato fino alla radice:

```
Procedure FIND(v)
begin
      x := v
      while padre(x) ≠ 0 do x := padre(x)
      return x
end
```

Osserva che, anche in questo caso, il costo della procedura (in termini di tempo) è proporzionale alla profondità del nodo considerato e quindi nel caso peggiore all'altezza dell'albero di appartenenza.

Analogamente, l'operazione UNION può essere realizzata determinando le radici dei due elementi e rendendo quindi la seconda figlia della prima.

```
Procedure UNION(u, v)
begin
      x := FIND(u)
      y := FIND(v)
      if x ≠ y then padre(y) := x
end
```

Anche in questo caso il tempo di calcolo dipende essenzialmente dalla profondità dei due nodi. Notiamo tuttavia che, se i nodi di ingresso u e v sono radici, il tempo è costante. Come si vedrà nel seguito, in molte applicazione le operazioni UNION vengono applicate solo alle radici.

L'implementazione appena descritta non è tuttavia particolarmente efficiente. Infatti, fissato un insieme A di n elementi, è facile definire una sequenza di n operazioni UNION e FIND che richiede $\Theta(n^2)$ passi, se applicata alla partizione identità ID (nella quale ogni elemento di A forma un insieme). A tale scopo è sufficiente definire una sequenza di operazioni UNION che porta alla costruzione di alberi del tutto sbilanciati (per esempio formati da semplici liste di elementi) e quindi applicare le operazioni FIND corrispondenti.

9.7.1 Foreste con bilanciamento

Per rimediare alla situazione appena descritta, si può definire una diversa implementazione delle partizioni, sempre utilizzando foreste, la quale mantiene l'informazione relativa alla cardinalità degli insiemi coinvolti. In questo modo l'operazione UNION può essere eseguita semplicemente rendendo la radice dell'albero più piccolo figlia della radice dell'albero di cardinalità maggiore. Se applicato a una foresta di alberi intuitivamente bilanciati, questo accorgimento consente di mantenere la proprietà di bilanciamento e quindi di ridurre il tempo richiesto dall'esecuzione delle procedure FIND.

Una foresta dotata di questa informazione può essere rappresentata aggiungendo alla tabella $padre$ una tabella num: per ogni radice r della foresta cosiderata, $num(r)$ indica il numero di nodi dell'albero che ha per radice r; conveniamo di porre $num(v) = 0$ per ogni nodo v non radice.

Esempio 9.10

La foresta descritta nell'esempio precedente può essere allora rappresentata dalla seguente tabella:

vertici	$padre$	num
1	7	0
2	0	1
3	7	0
4	8	0
5	9	0
6	8	0
7	0	3
8	0	5
9	8	0

∎

L'operazione UNION può allora essere eseguita nel modo seguente:

`Procedure` UNION(u, v)
`begin`
$x :=$ FIND(u)
$y :=$ FIND(v)
`if` $x \neq y$ `then`

$$\texttt{if } num(x) < num(y) \texttt{ then } \begin{cases} padre(x) := y \\ num(y) := num(x) + num(y) \\ num(x) := 0 \end{cases}$$

$$\texttt{else } \begin{cases} padre(y) := x \\ num(x) := num(x) + num(y) \\ num(y) := 0 \end{cases}$$

`end`

Nel seguito chiameremo *foresta con bilanciamento* l'implementazione appena descritta.
Possiamo ora provare la seguente proprietà :

Proposizione 9.3 *Utilizziamo una foresta con bilanciamento per implementare le partizioni di un insieme di n elementi. Allora, durante l'esecuzione di $n-1$ operazioni UNION a partire dalla partizione identità ID, l'altezza di ogni albero con k nodi non è mai superiore a $\lfloor \log_2 k \rfloor$.*

Dimostrazione. Ragioniamo per induzione su k. Se $k = 1$ la proprietà è banalmente verificata. Sia $k > 1$ e supponiamo la proprietà vera per ogni intero positivo $i < k$. Se T è un albero con k nodi, T è stato costruito unendo due alberi T_1 e T_2, dove il numero di nodi di T_1 è minore o uguale a quello di T_2. Quindi T_1 possiede al più $\lfloor k/2 \rfloor$ nodi e, per ipotesi di induzione, la sua altezza $h(T_1)$ deve essere minore o uguale a $\lfloor \log_2 k \rfloor - 1$. Analogamente, T_2 possiede al più $k-1$ nodi e di conseguenza $h(T_2) \leq \lfloor \log_2 (k-1) \rfloor \leq \lfloor \log_2 k \rfloor$. Osservando ora che

$$h(T) = \max\{h(T_2), h(T_1) + 1\}$$

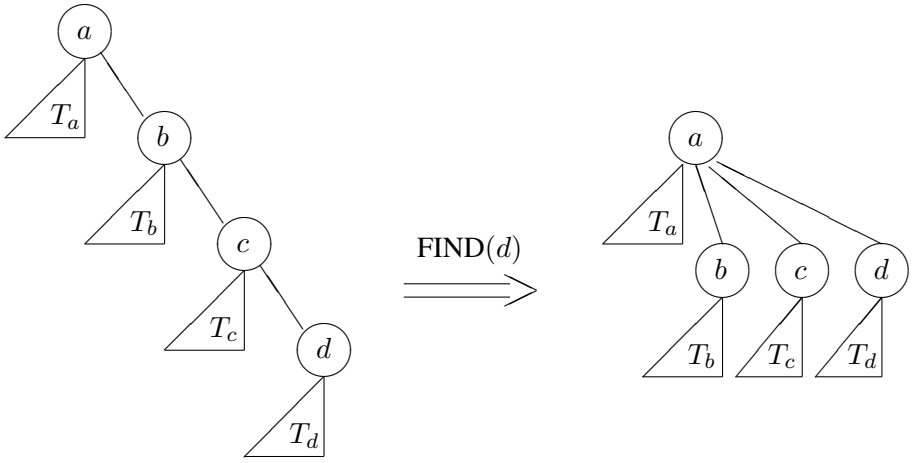
otteniamo $h(T) \leq \lfloor \log_2 k \rfloor$. ∎

L'immediata conseguenza della proposizione precedente è che l'esecuzione di $O(n)$ operazioni UNION e FIND su un insieme di n elementi, a partire dalla partizione identità, può essere eseguita in tempo $O(n \log n)$.

9.7.2 Compressione di cammino

La complessità asintotica $O(n \log n)$ appena ottenuta può essere migliorata eseguendo le operazioni FIND mediante *compressione di cammino*. Più precisamente si tratta di eseguire FIND(u) memorizzando in una lista tutti i nodi che si trovano sul cammino da u alla radice r rendendo poi ciascuno di questi vertici figlio di r. L'effetto quindi della esecuzione di una FIND è quello di comprimere l'albero nel quale si trova l'argomento della procedura. Di conseguenza l'esecuzione di ulteriori operazioni FIND sullo stesso albero potrebbe richiedere una minore quantità di tempo.

Il procedimento è descritto intuitivamente dalla seguente figura.



Rappresentando la foresta mediante la tabella $padre$ il procedimento descritto può essere implementato mediante la seguente procedura.

```
Procedure FIND(u)
begin
      v := u
      L := Λ (lista vuota)
      while padre(v) ≠ 0 do { L := INSERISCI_IN_TESTA(L, v)
                            { v := padre(v)
      for x ∈ L do padre(x) := v
      return v
end
```

Usando la compressione di cammino, la complessità in tempo dell'esecuzione delle operazioni UNION e FIND viene a dipendere dalla funzione $G : \mathbb{N} \to \mathbb{N}$ definita dall'uguaglianza

$$G(n) = \min\{k \in \mathbb{N} \mid n \leq F(k)\},$$

nella quale F è la funzione $F : \mathbb{N} \to \mathbb{N}$, tale che

$$F(k) = \begin{cases} 1 & \text{se } k = 0 \\ 2^{F(k-1)} & \text{se } k \geq 1. \end{cases}$$

Osserviamo che la funzione F cresce molto velocemente come mostra la seguente tabella:

k	$F(k)$
0	1
1	2
2	4
3	16
4	65536
5	2^{65536}

Viceversa la funzione G, essendo di fatto l'inversa di F, ha una crescita estremamente lenta. In realtà si verifica $G(n) \leq 5$ per ogni $n \leq 2^{65536}$ (ovvero per ogni n utilizzabile in pratica).

La seguente proposizione, di cui omettiamo la complicata dimostrazione, fornisce una valutazione asintotica del tempo di calcolo necessario per eseguire le operazioni UNION e FIND combinando le procedure descritte in questa sezione e nella precedente.

Proposizione 9.4 *Dato un insieme A, supponiamo di utilizzare una foresta per rappresentare una partizione di A e implementiamo le operazioni UNION e FIND mediante bilanciamento e compressione di cammino. Allora una sequenza di $O(n)$ operazioni UNION e FIND, eseguita a partire dalla partizione identità, può essere eseguita in tempo $O(n \cdot G(n))$.*

Esercizio

Considera la seguente procedura che esegue una sequenza di operazioni UNION-FIND su un insieme di $n > 2$ elementi $A = \{a_1, a_2, \ldots, a_n\}$, a partire dalla partizione identità.

```
begin
      for i = 2, ..., n do
          if i pari  then UNION(a_1, a_i)
                     else UNION(a_i, a_{i-2})
end
```

a) Supponendo di utilizzare una foresta semplice, descrivere l'albero ottenuto dalla procedura data.

b) Nell'ipotesi precedente, valutare l'ordine di grandezza del tempo di calcolo richiesto in funzione del parametro n (si assuma il criterio uniforme).

c) Supponiamo ora di utilizzare una foresta con bilanciamento: descrivere l'albero ottenuto eseguendo la procedura e valutare il tempo di calcolo richiesto in questo caso.

Capitolo 10

Il metodo Divide et Impera

Un metodo spesso usato per risolvere un problema consiste nel partizionare i dati di ingresso in istanze di dimensioni minori, risolvere il problema su tali istanze e combinare opportunamente i risultati parziali fino ad ottenere la soluzione cercata. Questa strategia è generalmente chiamata "divide et impera" e consente in molti casi di progettare algoritmi asintoticamente efficienti. Nella prossima sezione descriviamo il metodo in generale mentre in quelle successive presentiamo alcuni esempi significativi di algoritmi basati su questa tecnica.

10.1 Schema generale

Consideriamo un problema Π, descritto da una funzione $Sol : I \to R$, dove I è l'insieme delle istanze di Π (che chiameremo anche "dati") e R quello delle soluzioni ("risultati"). Come al solito, per ogni $x \in I$ denotiamo con $|x|$ la sua dimensione. Intuitivamente, per risolvere il problema Π su una istanza x, un algoritmo di tipo "divide et impera" procede nel modo seguente.

1. Se $|x|$ è minore o uguale a un valore C fissato, si determina direttamente la soluzione consultando una tabella in cui sono elencate tutte le soluzioni per ogni istanza $y \in I$ di dimensione minore o uguale a C, oppure applicando un algoritmo opportuno.

2. Altrimenti, si eseguono i seguenti passi:

   (a) partiziona x in b dati ridotti $rid_1(x), rid_2(x), \ldots, rid_b(x) \in I$ tali che, per ogni $j = 1, 2, \ldots, b$,

   $$|rid_j(x)| = \lceil |x|/a \rceil \text{ oppure } |rid_j(x)| = \lfloor |x|/a \rfloor$$

   per un opportuno $a > 1$ (e quindi $|rid_j(x)| < |x|$);

   (b) risolvi ricorsivamente il problema sulle istanze $rid_1(x), rid_2(x), \ldots, rid_b(x)$;

   (c) usa le risposte sui dati ridotti per ottenere la soluzione su x.

Supponiamo ora di avere una procedura Opevarie per l'esecuzione del passo (2c); in altre parole Opevarie restituisce il valore $Sol(x)$ ricevendo in ingresso le soluzioni relative ai dati $rid_1(x)$, $rid_2(x)$, ..., $rid_b(x)$. Allora, la procedura generale che si ottiene è la seguente:

```
Procedura F(x)
if |x| ≤ C then return Sol(x)
        else begin
                for j = 1, 2, ..., b do calcola x_j = rid_j(x)
                for j = 1, 2, ..., b do  z_j := F(x_j)
                w := Opevarie(z_1, z_2, ..., z_b)
                return w
             end
```

Osserviamo che le soluzioni parziali $Sol(x_1)$, $Sol(x_2)$,...,$Sol(x_b)$ vengono calcolate in maniera indipendente le une dalle altre. Come vedremo meglio in seguito questo porta a eseguire più volte eventuali computazioni comuni alle chiamate $F(x_1), F(x_2), \ldots, F(x_b)$. D'altra parte questo procedimento permette una facile parallelizzazione dell'algoritmo; infatti, disponendo di un numero opportuno di processori, le b chiamate ricorsive previste al passo (2b) possono essere eseguite contemporaneamente e in maniera indipendente mentre solo la fase di ricomposizione dei risultati prevede la sincronizzazione dei vari processi.

Passiamo ora all'analisi della procedura. Indichiamo con n la dimensione di una istanza del problema; vogliamo stimare il tempo $T(n)$ richiesto dall'algoritmo per risolvere il problema su dati di dimensione n nel caso peggiore.

Chiaramente, $T(n)$ dipenderà a sua volta dal tempo necessario per eseguire la procedura Opevarie sulle risposte a dati di dimensione n/a. Per calcolare tale quantità è necessario entrare nei dettagli dei singoli casi; a questo livello di generalità possiamo rappresentarla semplicemente con $T_{op}(n)$ che supponiamo nota.

Allora, supponendo che n sia divisibile per a, otteniamo la seguente equazione:

$$T(n) = b \cdot T\left(\frac{n}{a}\right) + T_{op}(n) \qquad (\text{ se } n > C).$$

Nel caso $n \leq C$ possiamo assumere che $T(n)$ sia minore o uguale a una costante fissata.

L'analisi di algoritmi ottenuti con la tecnica "divide et impera" è quindi ricondotta allo studio di semplici equazioni di ricorrenza del tipo

$$T(n) = bT\left(\frac{n}{a}\right) + g(n).$$

trattate in dettaglio nella sezione 6.4.

Illustriamo ora questa tecnica con quattro esempi che riguardano classici problemi.

10.2 Calcolo del massimo e del minimo di una sequenza

Come primo esempio presentiamo un algoritmo per determinare i valori massimo e minimo di una sequenza di interi. Formalmente il problema è definito nel modo seguente.

> Istanza: un vettore $S = (S[1], S[2], \ldots, S[n])$ di n interi;
> Soluzione: gli interi a e b che rappresentano rispettivamente il valore minimo e quello massimo tra gli elementi di S.

L'algoritmo che presentiamo in questa sede è basato sul confronto tra gli elementi di S. Di conseguenza la stessa procedura può essere utilizzata per risolvere l'analogo problema definito su una sequenza di elementi qualsiasi per i quali sia fissata una relazione d'ordine totale. Il nostro obiettivo è quello di determinare il numero di confronti eseguiti dalla procedura su un input di dimensione n: tale quantità fornisce anche l'ordine di grandezza del tempo di calcolo richiesto dalla procedura su una macchina RAM a costo uniforme. Lo stesso ragionamento può essere applicato per stimare altri parametri quali il numero di assegnamenti o di operazioni aritmetiche eseguiti.

Il procedimento più semplice per risolvere il problema scorre la sequenza di input mantenendo il valore corrente del massimo e del minimo trovati e confrontando questi ultimi con ciascuna componente di S.

```
begin
      b := S[1]
      a := S[1]
      for i = 2, ..., n do
          begin
                if S[i] < a then a := S[i]
                if S[i] > b then b := S[i]
          end
end
```

Tale procedura esegue $n-1$ confronti per determinare il minimo e altrettanti per determinare il massimo. In totale quindi vengono eseguiti $2n-2$ confronti.

Descriviamo ora un altro algoritmo che permette di risolvere il medesimo problema con un numero di confronti minore del precedente. L'idea è quella di suddividere la sequenza di ingresso in due parti uguali; richiamare ricorsivamente la procedura su queste due e quindi confrontare i risultati ottenuti. Formalmente l'algoritmo è definito dalla semplice chiamata della procedura ricorsiva Maxmin$(1, n)$ che restituisce i due valori cercati manipolando il vettore S di ingresso come una variabile globale.

```
begin
      read S[1], S[2], ..., S[n]
      (a, b) := Maxmin(1, n)
      return (a, b)
end
```

Per ogni coppia di interi i, j, tali che $1 \leq i \leq j \leq n$, la chiamata Maxmin(i, j) restituisce una coppia di elementi (p, q) che rappresentano rispettivamente i valori minimo e massimo del sottovettore $(S[i], S[i+1], \ldots, S[j])$. Tale procedura è ricorsiva e utilizza due operatori, $\max$ e $\min$, che definiscono rispettivamente il massimo e il minimo tra due interi.

Procedure Maxmin$(1, n)$

```
begin
      if i = j then return (S[i], S[i])
      else if i + 1 = j then return (min(S[i], S[j]), max(S[i], S[j]))
                        else begin
                                   k := ⌊(i+j)/2⌋
                                   (a1, b1) := Maxmin(i, k)
                                   (a2, b2) := Maxmin(k + 1, j)
```

```
                    return (min(a_1, a_2), max(b_1, b_2))
                end
end
```

Denotiamo con $C(n)$ il numero di confronti eseguito dall'algoritmo su un input di dimensione n. È facile verificare che $C(n)$ soddisfa la seguente equazione di ricorrenza:

$$C(n) = \begin{cases} 0 & \text{se } n = 1 \\ 1 & \text{se } n = 2 \\ C(\lfloor \frac{n}{2} \rfloor) + C(\lceil \frac{n}{2} \rceil) + 2 & \text{se } n > 2 \end{cases}$$

Supponendo $n = 2^k$ per $k \in \mathbb{N}$ e sviluppando l'equazione precedente, otteniamo

$$\begin{aligned} C(n) &= 2 + 2C\left(\frac{n}{2}\right) = 2 + 4 + 4C\left(\frac{n}{4}\right) = \\ &= 2 + 4 + \cdots + 2^{k-1} + 2^{k-1}C(2) = \\ &= \sum_{j=1}^{k-1} 2^j + 2^{k-1} = 2^k + 2^{k-1} - 2 \end{aligned}$$

Quindi, ricordando che $k = \log_2 n$, si ottiene $C(n) = \frac{3}{2}n - 2$. Quindi, rispetto all'algoritmo precedente, la nuova procedura esegue un numero di confronti ridotto di circa un quarto su ogni input che ha per dimensione una potenza di 2.

Esercizi

1) Assumendo il criterio di costo uniforme, determinare l'ordine di grandezza dello spazio di memoria richiesto dall'algoritmo sopra descritto.

2) Supponendo che le n componenti del vettore S siano interi compresi tra 1 e n, svolgere l'analisi del tempo di calcolo richiesto dall'algoritmo precedente assumendo il criterio di costo logaritmico.

10.3 Mergesort

In questa sezione presentiamo un altro classico esempio di algoritmo *divide et impera*. Si tratta dell'algoritmo Mergesort per risolvere il problema dell'ordinamento. Tale procedura ha notevoli applicazioni pratiche poiché su di essa sono basate gran parte delle routine di ordinamento esterno, quelle cioé che devono ordinare dati distribuiti su memoria di massa.

L'istanza del problema è data da un vettore $A = (A[1], A[2], \ldots, A[n])$ le cui componenti sono estratte da un insieme U totalmente ordinato rispetto a una relazione d'ordine $\leq$ fissata. L'insieme U potrebbe essere per esempio l'insieme $\mathbb{Z}$ dei numeri interi e $\leq$ l'usuale relazione d'ordine tra interi. Oppure, U potrebbe rappresentare l'insieme Σ^* di tutte le parole su un dato alfabeto finito Σ e $\leq$ potrebbe denotare la relazione di ordinamento lessicografico su Σ^*.

L'algoritmo restituisce il vettore A ordinato in modo non decrescente. Il procedimento è semplice: si suddivide il vettore A in due sottovettori di dimensione quasi uguale, si richiama ricorsivamente la procedura per ordinare ciascuno di questi, quindi si immergono le due sequenze ottenute in un'unica n-pla ordinata. L'algoritmo è descritto formalmente dalla procedura Mergesort(i, j), $1 \leq i \leq j \leq n$, che ordina il sottovettore $(A[i], A[i+1], \ldots, A[j])$. La semplice chiamata di Mergesort$(1, n)$ restituisce l'intero vettore A ordinato. Durante l'esecuzione del calcolo il vettore A è considerato come una variabile globale.

Procedura Mergesort(i, j)
```
begin
      if i < j then
         begin
               k := ⌊(i+j)/2⌋
               Mergesort(i, k)
               Mergesort(k + 1, j)
               Merge(i, k, j)
         end
end
```

Il cuore del procedimento è costituito dalla procedura Merge(i, k, j), $1 \leq i \leq k < j \leq n$, che riceve i vettori ordinati $(A[i], A[i+1], \ldots, A[k])$, $(A[k+1], \ldots, A[j])$ e restituisce il vettore $(A[i], A[i+1], \ldots, A[j])$ ordinato definitivamente.

La procedura Merge(i, k, j) scorre gli elementi delle due sequenze $(A[i], A[i+1], \ldots, A[k])$ e $(A[k+1], \ldots, A[j])$ mediante due puntatori, uno per ciascun vettore, inizialmente posizionato sulla prima componente. Ad ogni passo gli elementi scanditi dai puntatori vengono confrontati, il minore viene aggiunto a una lista prefissata (inizialmente vuota) e il puntatore corrispondente viene spostato sull'elemento successivo. Quando uno dei due vettori è stato scandito interamente si aggiungono alla lista gli elementi rimanenti dell'altro nel loro ordine. Al termine la lista contiene i valori dei due vettori ordinati e viene quindi ricopiata nelle componenti $A[i], A[i+1], \ldots, A[j]$. La procedura è descritta formalmente dal seguente programma nel quale la lista ausiliaria è implementata dal vettore B.

Procedura Merge(i, k, j)
```
begin
      t := 0; p := i; q := k + 1
      while p ≤ k ∧ q ≤ j do
            begin
                  t := t + 1
                  if A[p] ≤ A[q] then { B[t] := A[p]
                                      { p := p + 1
                                 else { B[t] := A[q]
                                      { q := q + 1
            end
      if p ≤ k then
            for ℓ = 0, 1, ..., k − p do A[j − ℓ] := A[k − ℓ]
      for u = 1, 2, ..., t do A[i + u − 1] := B[u]
end
```

Vogliamo calcolare il massimo numero di confronti eseguiti dalla procedura Mergesort su un input di lunghezza n. Denotiamo con $M(n)$ tale quantità. È facile verificare che Merge(i, k, j) esegue, nel caso peggiore, $j - i$ confronti (quanti nel caso migliore?). Ne segue che, per ogni $n > 1$, $M(n)$ soddisfa la seguente equazione:

$$M(n) = \begin{cases} 0 & \text{se } n = 1 \\ M(\lfloor \frac{n}{2} \rfloor) + M(\lceil \frac{n}{2} \rceil) + n - 1 & \text{se } n > 1. \end{cases}$$

Applicando allora i risultati presentati nella sezione 6.4 si prova facilmente che $M(n) = \Theta(n \log_2 n)$. Procedendo con maggior accuratezza si ottiene un risultato più preciso e cioè che $M(n) = n\lceil \log_2 n \rceil - 2^{\lceil \log_2 n \rceil} + 1$.

Esercizi

1) Dimostrare che $M(n) = n \log_2 n - n + 1$ per ogni n potenza di 2.

2) Supponendo n una potenza di 2, determinare nel caso migliore il numero di confronti eseguiti da Mergesort su un input di dimensione n.

3) Assumendo il criterio di costo uniforme, determinare l'ordine di grandezza dello spazio di memoria richiesto Mergesort per ordinare n elementi.

10.4 Prodotto di interi

Un altro esempio significativo di algoritmo "divide et impera" riguarda il problema del calcolo del prodotto di interi. Consideriamo due interi positivi x, y di n bits ciascuno, e siano $\underline{x} = x_1 x_2 \cdots x_n$ e $\underline{y} = y_1 y_2 \cdots y_n$ le rispettive rappresentazioni binarie dove, per ogni $i \in \{1, 2, \ldots, n\}$, $x_i \in \{0, 1\}$ e $y_i \in \{0, 1\}$. Vogliamo calcolare la rappresentazione binaria $\underline{z} = z_1 z_2 \cdots z_{2n}$ del prodotto $z = x \cdot y$. Il problema è quindi definito nel modo seguente:

Istanza: due stringhe binarie $\underline{x}, \underline{y} \in \{0, 1\}^*$ di lunghezza n che rappresentano rispettivamente gli interi positivi x e y;

Soluzione: la stringa binaria $\underline{z} \in \{0, 1\}^*$ che rappresenta il prodotto $z = x \cdot y$.

Assumiamo che la dimensione di una istanza $\underline{x}, \underline{y}$ sia data dalla lunghezza delle due stringhe.

Vogliamo descrivere un algoritmo per la soluzione del problema e determinare il numero delle operazioni binarie richieste per una istanza di dimensione n. È facile verificare che il metodo tradizionale richiede $O(n^2)$ operazioni binarie. In questa sede presentiamo un algoritmo che riduce tale quantità a $O(n^{1.59})$.

Per semplicità, supponiamo che n sia una potenza di 2 e spezziamo $\underline{x}$ e $\underline{y}$ in due vettori di $n/2$ bits ciascuno: $\underline{x} = \underline{ab}$, $\underline{y} = \underline{cd}$. Se denotiamo con a, b, c, d gli interi rappresentati rispettivamente da $\underline{a}, \underline{b}, \underline{c}$ e $\underline{d}$, il prodotto xy può essere calcolato mediante la seguente espressione:

$$xy = (a2^{n/2} + b)(c2^{n/2} + d) = ac2^n + (ad + bc)2^{n/2} + bd$$

Questa espressione permette di calcolare il prodotto xy mediante 4 moltiplicazioni di interi di $n/2$ bits più alcune addizioni e shift (prodotti per potenze di 2). Tuttavia possiamo ridurre da 4 a 3 il numero delle moltiplicazioni osservando che il valore $ad + bc$ si può ottenere da ac e bd mediante un solo ulteriore prodotto: basta calcolare $u = (a + b)(c + d)$ ed eseguire la sottrazione $u - ac - bd = ad + bc$.

In questo modo il calcolo di $z = xy$ richiede l'esecuzione delle seguenti operazioni:

1) le operazioni necessarie per calcolare l'intero u, cioè 2 somme di interi di $n/2$ bits e una moltiplicazione di due interi che hanno al più $n/2 + 1$ bits;

2) 2 moltiplicazioni su interi di $n/2$ bits per calcolare ac e bd;

3) 6 operazioni tra addizioni, sottrazioni e shift su interi di n bits.

È chiaro che ogni addizione, sottrazione e shift su interi di n bits richiede $O(n)$ operazioni binarie. Inoltre anche il calcolo di u al punto 1) può essere eseguito mediante un prodotto tra interi di $n/2$ bits più alcune addizioni e shift.

Per la verifica di questo fatto, sia a_1 il primo bit della somma $(a+b)$ e sia b_1 l'intero rappresentato dai rimanenti. Osserva che $a+b = a_1 2^{n/2} + b_1$. Analogamente esprimiamo $c+d$ nella forma $c+d = c_1 2^{n/2} + d_1$, dove c_1 è il primo bit di $(c+d)$ e d_1 l'intero rappresentato dai bit rimanenti. Allora, è evidente che

$$(a+b)(c+d) = a_1c_1 2^n + (a_1d_1 + b_1c_1)2^{n/2} + b_1d_1.$$

Il calcolo di b_1d_1 richiede il prodotto di due interi di $n/2$ bits ciascuno, mentre gli altri prodotti (a_1c_1, a_1d_1, b_1c_1) possono essere calcolati direttamente in tempo binario $O(n)$ perché si tratta sempre del prodotto di un intero per 1 o per 0.

In totale quindi, per calcolare il prodotto di due interi di n bits, il procedimento illustrato esegue 3 moltiplicazioni su interi di (circa) $n/2$ bits più $O(n)$ operazioni binarie. Di conseguenza, per ogni n potenza di 2, il numero totale delle operazioni binarie è minore o uguale alla soluzione $T(n)$ della seguente equazione di ricorrenza:

$$T(n) = \begin{cases} \alpha & \text{se } n = 1 \\ 3T(\frac{n}{2}) + \beta n & \text{se } n > 1, \end{cases}$$

dove α e β sono costanti opportune. Applicando ora il teorema 6.3 otteniamo

$$T(n) = \Theta(n^{log_2 3}) = O(n^{1.59})$$

10.5 L'algoritmo di Strassen.

Il problema considerato in questa sezione è definito nel modo seguente:

Istanza: due matrici di numeri razionali $A = [a_{ik}]$, $B = [b_{ik}]$, ciascuna di dimensione $n \times n$.

Soluzione: la matrice prodotto $A \cdot B = [c_{ik}]$, con $c_{ik} = \sum_{j=1}^{n} a_{ij} \cdot b_{jk}$.

L'algoritmo tradizionale che calcola direttamente il prodotto di matrici richiede $\Theta(n^3)$ somme e prodotti di numeri reali. È possibile scendere sotto questo limite?

Una prima risposta affermativa è stata data con l'algoritmo di Strassen, di cui delineiamo i fondamenti. Supponiamo per semplicità che n sia una potenza di 2. Allora si può porre:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \quad A \cdot B = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}.$$

dove le matrici $A_{ik},\ B_{ik},\ C_{ik}$ sono di ordine $\frac{n}{2} \times \frac{n}{2}$.

Si calcolino ora le matrici P_i ($1 \leq i \leq 7$):

$P_1 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22})$
$P_2 = (A_{21} + A_{22}) \cdot B_{11}$
$P_3 = A_{11} \cdot (B_{12} - B_{22})$
$P_4 = A_{22} \cdot (B_{21} - B_{11})$
$P_5 = (A_{11} + A_{12}) \cdot B_{22}$
$P_6 = (A_{21} - A_{11}) \cdot (B_{11} + B_{12})$
$P_7 = (A_{12} - A_{22}) \cdot (B_{21} + B_{22})$

Con una laboriosa ma concettualmente semplice verifica si ha:

$C_{11} = P_1 + P_4 - P_5 + P_7$
$C_{12} = P_3 + P_5$

$C_{21} = P_2 + P_4$
$C_{22} = P_1 + P_3 - P_2 + P_6$
Le P_i sono calcolabili con 7 moltiplicazioni di matrici e 10 fra addizioni e sottrazioni. Analogamente, le C_{ij} si possono calcolare con 8 addizioni e sottrazioni a partire dalle P_i.

Il prodotto di due matrici $n \times n$ può essere allora ricondotto ricorsivamente al prodotto di 7 matrici $\frac{n}{2} \times \frac{n}{2}$, mediante l'esecuzione di 18 addizioni di matrici $\frac{n}{2} \times \frac{n}{2}$.

Poiché due matrici $n \times n$ si sommano con n^2 operazioni, il tempo di calcolo dell'algoritmo precedentemente delineato è dato dalla seguente equazione:

$$T(n) = 7T\left(\frac{n}{2}\right) + O(n^2)$$

La soluzione di questa equazione dà $T(n) = \Theta(n^{\lg_2 7})$, con $\lg_2 7 \approx 2.81$.

10.6 La trasformata discreta di Fourier

Il calcolo della convoluzione di due sequenze numeriche è un problema classico che sorge in vari settori scientifici e ha numerose applicazioni sopratutto nell'ambito della progettazione di algoritmi per operazioni su interi e polinomi. La trasformata discreta di Fourier fornisce uno strumento per eseguire in maniera efficiente tale operazione. Si tratta di una trasformazione lineare F tra sequenze finite di elementi, definiti in generale su un anello opportuno, che mappa la convoluzione di due n-ple $\underline{a}$, $\underline{b}$ nel prodotto, termine a termine, delle due immagini $F(\underline{a})$ e $F(\underline{b})$. Nota che il calcolo del prodotto termine a termine di due sequenze è certamente più semplice del calcolo della loro convoluzione. Quindi, se disponiamo di algoritmi efficienti per determinare la trasformata discreta di Fourier e la sua inversa, possiamo di fatto disporre di una procedura per calcolare la convoluzione.

Tra le applicazioni di questo metodo ve ne sono alcune di particolare interesse. Tra queste ricordiamo il noto algoritmo di Schönhage e Strassen per la moltiplicazione di due interi che fornisce tuttora il metodo asintoticamente migliore per risolvere il problema. Esso consente di calcolare il prodotto di due interi di n bit eseguendo $O(n \log n \log\log n)$ operazioni binarie.

10.6.1 La trasformata discreta e la sua inversa

Dato un anello commutativo $A = \langle A, +, \cdot, 0, 1\rangle$, l'elemento $n = \sum_{k=1}^{n} 1$ ammette inverso moltiplicativo se esiste $\frac{1}{n}$ tale che $\frac{1}{n} \cdot n = 1$. Un elemento ω di A tale che $\omega \neq 1$, $\omega^n = 1$, $\sum_{j=0}^{n-1} \omega^{jk} = 0$ per ogni $k = 1, 2, \ldots, n-1$, è detto n-esima radice *principale* dell'unità. Chiameremo invece n-esime radici dell'unità gli elementi $\omega^0 = 1, \omega, \omega^2, \ldots, \omega^{n-1}$.

Analizziamo ora due anelli particolarmente interessanti.

Esempio 10.1
Sia $\mathcal{C} = \langle \mathbb{C}, +, \cdot, 0, 1\rangle$ il campo dei numeri complessi. Allora si verifica facilmente che ogni $n \in \mathbf{N}$ ammette inverso moltiplicativo e che $e^{i\frac{2\pi}{n}}$ è una n-esima radice principale dell'unità. ∎

Esempio 10.2
Dato un intero n potenza di 2 (cioè $n = 2^k$) e posto $m = 2^n + 1$, sia Z_m l'anello dei resti modulo m, cioè $Z_m = \{0, 1, \ldots, m-1\}$, $x + y = \langle x + y\rangle_m$, $x \cdot y = \langle x \cdot y\rangle_m$. Poiché $n = 2^k$ è potenza di 2 mentre $m = 2^n + 1$ è dispari, n e m sono primi fra loro e quindi n ammette inverso moltiplicativo $\frac{1}{n}$ in Z_m; inoltre 4 è una n-esima radice principale dell'unità.

Mostriamo qui che $\omega = 4$ è una n-esima radice principale dell'unità in Z_m. Intanto $4 \neq 1$ ed inoltre

$$\langle 4^n \rangle_{2^n+1} = (\langle 2^n \rangle_{2^n+1})^2 = (-1)^2 = 1.$$

Mostriamo ora che $\sum_{i=0}^{n-1} 4^{\beta i} = 0$ per $\beta = 1, 2, \ldots, n-1$. A tale scopo consideriamo per ogni $a \in Z_m$ la seguente identità

$$\prod_{i=0}^{k-1} \left(1 + a^{2^i}\right) = \sum_{\alpha=0}^{2^k-1} a^{\alpha}$$

che si ottiene da

$$\prod_{i=0}^{k-1} \left(a^{0\cdot 2^i} + a^{1\cdot 2^i}\right) = \sum_{C_0,\ldots,C_{k-1}\in\{0,1\}} a^{C_0+2C_1+\ldots+2^{k-1}C_{k-1}}$$

osservando che per ogni α ($0 \leq \alpha < 2^k - 1$) sono biunivocamente individuabili $C_0, \ldots, C_{k-1} \in \{0, 1\}$ tali che $\alpha = C_0 + 2C_1 + \ldots + 2^{k-1}C_{k-1}$.

Per l'identità precedente basta allora dimostrare che per ogni β ($1 \leq \beta < n$) esiste i ($0 \leq i < k$) tale che $1 + 4^{\beta\cdot 2^i} = 0 \mod(2^n + 1)$. A tale riguardo sia $2\beta = 2^l \cdot d$ con d dispari; si ponga $i = k - l$, vale:

$$4^{\beta\cdot 2^i} + 1 = 2^{2^k\cdot d} + 1 = (-1)^d + 1 = 0 \qquad mod(2^n + 1)$$

∎

Sia da ora in poi A un anello commutativo in cui n ammette inverso moltiplicativo e in cui ω è una n-esima radice principale dell'unità. Consideriamo ora l'algebra A^n formata dai vettori a n componenti in A, cioè $A^n = \{a \mid a = (a[0], \ldots, a[n-1]), a[i] \in A\}$, dotata delle seguenti operazioni:

1) Somma	$+$	$(a + b)[k] = a[k] + b[k]$
2) Prodotto	$\cdot$	$(a \cdot b)[k] = a[k] \cdot b[k]$
3) Prodotto di convoluzione ciclica	$\odot$	$(a \odot b)[k] = \sum_{\langle j+s\rangle_n = k} a[j] \cdot b[s]$

Si definisce trasformata discreta di Fourier $\mathcal{F}$ la trasformazione $\mathcal{F} : A^n \to A^n$ realizzata dalla matrice $\left[\omega^{ij}\right]_{i,j\in\{1,2,\ldots,n\}}$, cioè :

$$(\mathcal{F}(a))[k] = \sum_{s=0}^{n-1} \omega^{k\cdot s} a[s]$$

Valgono le seguenti proprietà:

Teorema 10.1 *$\mathcal{F}$ è una trasformazione invertibile e la sua inversa $\mathcal{F}^{-1}$ è realizzata dalla matrice $\frac{1}{n} \cdot \left[\omega^{-ij}\right]$*

Dimostrazione. Detto $[C_{ij}]$ il prodotto delle matrici $[\omega^{ij}]$ e $\frac{1}{n} \cdot [\omega^{-ij}]$, vale

$$C_{is} = \frac{1}{n} \sum_{k=0}^{n-1} \omega^{i\cdot k} \omega^{-k\cdot s} = \frac{1}{n} \sum_{k=0}^{n-1} \left(\omega^{i-s}\right)^k$$

pertanto

$$C_{is} = \begin{cases} 1 & \text{se } i = s \\ 0 & \text{altrimenti} \end{cases},$$

e quindi $[C_{ij}]$ coincide con la matrice identità. ∎

Teorema 10.2 $\mathcal{F}(a+b) = \mathcal{F}(a) + \mathcal{F}(b)$; $\mathcal{F}^{-1}(a+b) = \mathcal{F}^{-1}(a) + \mathcal{F}^{-1}(b)$
$\mathcal{F}(a \odot b) = \mathcal{F}(a) \cdot \mathcal{F}(b)$; $\mathcal{F}^{-1}(a \cdot b) = \mathcal{F}^{-1}(a) \odot \mathcal{F}^{-1}(b)$

Dimostrazione. Dimostriamo qui che $\mathcal{F}(a \odot b) = \mathcal{F}(a) \cdot \mathcal{F}(b)$. Posto $c = \mathcal{F}(a \odot b)$, vale:

$$c[i] = \sum_{k=0}^{n-1} \omega^{ik} \, (a \odot b)\,[k] = \sum_{k,j} \omega^{ik} a[j] \cdot b\,[\langle k-j \rangle_n] = \sum_{k,j} \omega^{i(k-j)} \cdot b\,[\langle k-j \rangle_n] \cdot \omega^{ij} a[j] =$$

$$= \left(\sum_s \omega^{is} b[s] \right) \cdot \left(\sum_j \omega^{ij} a[j] \right) = [\mathcal{F}(a) \cdot \mathcal{F}(b)]_i \qquad \blacksquare$$

10.6.2 La trasformata veloce di Fourier

In tutto questo paragrafo supporremo che n sia una potenza di 2, cioè $n = 2^k$, che A sia un anello commutativo in cui n sia invertibile e ω sia una n-esima radice principale dell'unità. Le seguenti osservazioni permettono di applicare una strategia "divide et impera" al problema di determinare la trasformata di Fourier di un vettore $(a[0], \ldots, a[n])$. Posto infatti $y = \mathcal{F}(a)$, vale per $0 \leq k < n$:

$$y[k] = a[0] + \omega^k a[1] + \omega^{2k} a[2] + \ldots + \omega^{(n-1)k} a[n-1]$$

Pertanto risulta:

1. $y[k] = (a[0] + \omega^{2k} a[2] + ... + \omega^{2\frac{n-2}{2}} a[n-2]) + \omega^k (a[1] + \omega^{2k} a[3] + \ldots + \omega^{2\frac{n-2}{2}} a[n-1])$.
2. ω^2 è una $\frac{n}{2}$-esima radice principale dell'unità in A.

Questo prova la correttezza della seguente procedura ricorsiva per il calcolo della trasformata di Fourier, che chiameremo FFT (acronimo di "Fast Fourier Transform").

```
Procedura FFT ( ω; (a[0], a[1], ..., a[n−1]) )
 if n = 1 then  return(a[0])
          else  begin
                 (b[0], ..., b[⌈(n−2)/2⌉]) := FFT(ω²; (a[0], a[2], ..., a[n−2]))
                 (c[0], ..., c[⌈(n−2)/2⌉]) := FFT(ω²; (a[1], a[3], ..., a[n−1]))
                 for k = 0, to n − 1 do
                  d[k] := b[⟨k⟩_{n/2}] + ω^k · c[⟨k⟩_{n/2}]
                 return (d[0], ..., d[n−1])
                end
```

Osserviamo che le operazioni di base su cui tale algoritmo è costruito sono la somma di elementi dell'anello A e la moltiplicazione per una potenza di ω.

Detto $T(n)$ il numero di tali operazioni, vale evidentemente:

$$\begin{cases} T(1) = 0 \\ T(n) = 2T\left(\frac{n}{2}\right) + 2n \end{cases}$$

Ne segue che $T(n) = 2n \lg n$. Discorso perfettamente analogo può essere fatto per l'inversa $\mathcal{F}^{-1}$. Una immediata applicazione è un algoritmo veloce per il calcolo della convoluzione circolare. Dal Teorema 2 segue infatti che :

$$a \odot b = \mathcal{F}^{-1}\left(\mathcal{F}(a) \cdot \mathcal{F}(b)\right)$$

Poiché il calcolo della trasformata e della trasformata inversa richiede $O(n \lg n)$ operazioni di somma e di moltiplicazione, e il prodotto richiede n moltiplicazioni, si conclude che:

Fatto 10.1 *La convoluzione circolare $a \odot b$ di due vettori n-dimensionali richiede $O(n \lg n)$ operazioni di somma e di prodotto per una potenza di ω, e n operazioni di prodotto.*

10.6.3 Prodotto di polinomi

Consideriamo qui il problema di moltiplicare due polinomi di grado n a coefficienti reali con un algoritmo che utilizzi un basso numero di somme e prodotti.

Dati due polinomi di grado n, $p(z) = \sum_{k=0}^{n} p_k \cdot z^k$ e $q(z) = \sum_{k=0}^{n} q_k \cdot z^k$, il loro prodotto $p(z) \cdot q(z)$ è il polinomio, di grado $2n$, $s(z) = \sum_{k=0}^{2n} s_k \cdot z^k$ dove:

$$s_k = \begin{cases} \displaystyle\sum_{j=0}^{k} p_j \cdot q_{k-j} & \text{se } 0 \le k \le n \\ \displaystyle\sum_{j=k-n}^{n} p_j \cdot q_{k-j} & \text{se } n < k \le 2n \end{cases}$$

Il calcolo diretto di s_k richiede $k+1$ prodotti e k somme (se $k \le n$) o $2n-k+1$ prodotti e $2n-k$ somme (se $k > n$). Il numero totale di prodotti e di somme è allora $\approx 2n^2$. Tale numero può essere ridotto a $O(n \lg n)$ osservando che la convoluzione circolare dei vettori a $2n+1$ componenti $(p_0, p_1, \ldots, p_n, 0, 0, \ldots, 0)$ e $(q_0, q_1, \ldots, q_n, 0, 0, \ldots, 0)$ è esattamente il vettore $(s_0, s_1, \ldots, s_{2n})$. Ciò unitamente alle proprietà della trasformata discreta di Fourier, prova la correttezza del seguente algoritmo:

ALGORITMO: **Moltiplicazione Veloce**

Ingresso: due polinomi rappresentati dai vettori dei coefficienti $(p_0, p_1, \ldots, p_n)$, $(q_o, q_1, \ldots, q_n)$

1. Calcola l'intero $N = 2^k$ tale che $2n+1 \le N < 2(2n+1)$.
2. a:= vettore a N componenti $(p_0, \ldots, p_n, 0, \ldots, 0)$.
3. b:= vettore a N componenti $(q_0, \ldots, q_n, 0, \ldots, 0)$.
4. $c := \mathcal{F}^{-1}\left(\mathcal{F}(a) \cdot \mathcal{F}(b)\right)$.

Uscita : il polinomio di grado al più $2n$ i cui coefficienti sono $c_0, \ldots, c_{2n}$.

Per quanto riguarda l'analisi di complessità, da **Fatto 10.1** segue immediatamente che l'algoritmo precedente richiede $O(n \lg n)$ somme e prodotti per potenza di $\omega = e^{\frac{2\pi i}{N}}$ e solo $O(n)$ operazioni di prodotto.

Per quanto riguarda l'implementazione su RAM, l'analisi precedente non è realistica, basandosi sulla facoltà di rappresentare arbitrari numeri complessi ed eseguire somme e prodotti in tempo costante. In realtà, se rappresentiamo i numeri con errore di arrotondamento, sia l'errore sul risultato sia il tempo di calcolo viene a dipendere dall'errore di arrotondamento fissato.

Vogliamo qui studiare algoritmi efficienti per il calcolo esatto del prodotto di due polinomi a coefficienti interi, rispetto ad un modello di calcolo in cui:

1. Gli interi sono rappresentati in notazione binaria.

2. La somma di due interi in n bit richiede n operazioni elementari.

3. Il prodotto di due interi di n bit richiede $M(n)$ operazioni elementari.

Ad esempio, utilizzando l'algoritmo di moltiplicazione che si impara alle elementari, abbiamo $M(n) = n^2$ mentre, utilizzando l'algoritmo di Schönhage-Strassen, otteniamo $M(n) = O(n \lg n \lg \lg n)$.

Il problema può essere così formulato:

Problema: **Prodotto Esatto**.

Istanza: due polinomi rappresentati da due vettori $(p_0, \ldots, p_n)$ e $(q_0, \ldots, q_n)$ di interi di al più n bit l'uno.

Richiesta: calcolare $s_k = \sum_{j=0}^{k} p_j q_{k-j}$ con $0 \leq k \leq 2n$, assumendo $p_j = q_j = 0$ per ogni j tale che $j < 0$ oppure $j > n$.

Osserviamo innanzitutto che se $a < m$, allora $\langle a \rangle_m = a$.

Poiché ora i numeri p_i, q_i sono al più di n bit, ne segue che $p_i, q_i < 2^n$ e quindi per ogni k ($0 \leq k \leq 2n$) vale $s_k = \sum_{j=0}^{k} q_j p_{k-j} < n 2^{2n} < 2^{2.5n} + 1$ (a meno che $n < 4$). Posto allora $m \geq 2^{2.5n} + 1$, ne segue: $\langle s_k \rangle_m = s_k \quad (0 \leq k \leq 2n)$.

Per ottenere il corretto prodotto, basta allora considerare i coefficienti p_k, q_i come elementi dell'anello Z_m con le operazioni di somma e prodotto modulo m. Detta $\mathcal{F}_m$ la trasformata di Fourier su Z_m con radice 4 (vedi **Esempio 10.2**) si ha il seguente:

ALGORITMO: **Prodotto Esatto Veloce**.

Ingresso: due polinomi rappresentati da due vettori $(p_0, \ldots, p_n)$ e $(q_0, \ldots, q_n)$ di interi di al più n bit l'uno.

$m := 2N + 1$	dove $N = 2^k$ con $2.5n \leq N \leq 5n$
a:=$(p_0, \ldots, p_n, 0, \ldots, 0)$	vettore a N componenti in Z_m.
b:=$(q_0, \ldots, q_n, 0, \ldots, 0)$	vettore a N componenti in Z_m.
$c := \mathcal{F}_m^{-1}(\mathcal{F}_m(a) \cdot \mathcal{F}_m(b))$.	

Uscita: $s_k = c_k \quad (0 \leq k \leq 2n)$

Per quando riguarda la complessità dell'algoritmo precedente, osserviamo che esso richiede $O(N \lg N)$ operazioni di somma e moltiplicazioni per potenze di 4 nonché N moltiplicazioni di interi di al più N bit.

Ricordando che ogni somma costa al più N operazioni elementari, ogni prodotto per potenze di 4 è uno shift e quindi costa al più N operazioni elementari, ogni prodotto costa al più $M(N)$ operazioni elementari, concludiamo che il tempo di $T(n)$ complessivo è :

$$T(n) = O\left(n^2 \lg n + nM(n)\right)$$

dove abbiamo tenuto conto che $N < 5n$. Implementando il prodotto col metodo di Schönhage-Strassen, si può concludere :

$$T(n) = O\left(n^2 \lg n \lg\lg n\right)$$

10.6.4 Prodotto di interi

Obiettivo di questo paragrafo è il progetto di un algoritmo asintoticamente veloce per il prodotto di interi di n bit.

L'algoritmo di moltiplicazione imparato alle scuole elementari richiede tempo $O(n^2)$; abbiamo visto che una semplice applicazione della tecnica "Divide et impera" permette di realizzare un algoritmo più veloce (Tempo= $O(n^{\log_2 3})$).

Presentiamo qui un' applicazione delle tecniche FFT, disegnando un algoritmo quasi lineare (Tempo= $O(n \lg^5 n)$). Tale algoritmo è una semplificazione didattica di quello proposto da Schönhage-Strassen, che è tuttora l'algoritmo di moltiplicazione asintoticamente più efficiente noto (Tempo= $O(n \lg n \lg\lg n)$).

Parametri della procedura sono gli interi $a = x_{n-1} \dots x_0$ e $b = y_{n-1} \dots y_0$ rappresentati nella notazione binaria. Essi vengono decomposti in $M = \sqrt{n}$ blocchi $a_M, \dots, a_0$ e $b_M, \dots, b_0$ ognuno composto da M bit: in tal modo a e b individuano rispettivamente i polinomi $A(z) = \sum_{k=0}^{M} a_k \cdot z^k$ e $B(z) = \sum_{k=0}^{M} b_k \cdot z^k$. Si noti che $a = \sum_{k=0}^{M} a_k \cdot 2^{M \cdot k} = A(2^M)$ e $b = \sum_{k=0}^{M} b_k \cdot 2^{M \cdot k} = B(2^M)$.

Tali polinomi vengono moltiplicati con la tecnica veloce presentata precedentemente, in cui i prodotti vengono eseguiti richiamando ricorsivamente la procedura. Ottenuto il polinomio $A(z) \cdot B(z) = C(z) = \sum_{k=0}^{M} c_k z^k$, si calcola $C(2^M) = \sum_{k=0}^{M} c_k \cdot 2^{M \cdot k}$. Risulta infatti che $C(2^M) = A(2^M) \cdot B(2^M) = a \cdot b$. Esiste una costante $C > 0$ per cui la seguente procedura calcola il prodotto di 2 interi:

`Procedura` PROD_VEL($a = x_{n-1} \dots x_0; b = y_{n-1} \dots y_0$)
`if` $n < C$ `then` calcola $a \cdot b$ col metodo elementare, `return` $(a \cdot b)$
`else begin`
- $M = \sqrt{n}$
- decomponi a in M blocchi $a_{M-1}, \dots, a_0$ di lunghezza M.
- decomponi b in M blocchi $b_{M-1}, \dots, b_0$ di lunghezza M.
- Calcola l'intero $N = 2^k$ tale che $2.5 \cdot M \leq N < 5 \cdot M$.
- siano a, b i vettori a N componenti
  $a = (a_0, \dots, a_{M-1}, 0, \dots, 0)$
  $b = (b_0, \dots, b_{M-1}, 0, \dots, 0)$
- $(c_0, \dots, c_{N-1}) = \mathcal{F}_{2^N+1}(a)$
  $(d_0, \dots, d_{N-1}) = \mathcal{F}_{2^N+1}(b)$
- `for` $k = 0$ `to` $N - 1$ `do`
  α_k = PROD_VEL($c_k; d_k$)
- $(z_0, \dots, z_{N-1}) := \mathcal{F}^{-1}_{2^N+1}(\alpha_0, \dots, \alpha_{N-1})$
- `return` $\left(\sum_{k=0}^{N-1} z_k \cdot 2^{M \cdot k}\right)$

`end`

Una semplice analisi permette di verificare che, detto $T(n)$ il tempo di calcolo della procedura di cui sopra:

$$T(n) \leq NT(N) + O(n \lg^2 n).$$

Ricordando che $N \leq 5\sqrt{n}$, vale:

$$T(n) \leq 5\sqrt{n}T(5\sqrt{n}) + O(n \lg^2 n).$$

Posto $g(n) = n \lg^5 n$, si verifica che per n sufficientemente grande:

$$5\sqrt{n}\, g(5\sqrt{n}) + O\left(n \lg^2 n\right) = 25n \left(\lg 5 + \frac{\lg n}{2}\right)^5 + O\left(n \lg^2 n\right) \leq g(n).$$

Applicando ora il corollario 6.2 otteniamo:

$$T(n) = O(n \lg^5 n).$$

Esercizi

1) Consideriamo il problema di calcolare il prodotto di n interi $a_1, a_2, \ldots, a_n$ tali che $a_i \in \{1, 2, \ldots, n\}$ per ogni $i = 1, 2, \ldots, n$. Mantenedo gli n valori di input in un vettore $A = (A[1], A[2], \ldots, A[n])$ di variabili globali, possiamo risolvere il problema chiamando la procedura *Prodotto*$(1, n)$ definita nel modo seguente per ogni coppia di interi i, j, $1 \leq i \leq j \leq n$:

Procedure *Prodotto*(i, j)
if $i = j$ **then return** $A[i]$
else begin
$k := \lfloor \frac{i+j}{2} \rfloor$
$a :=$*Prodotto*(i, k)
$b :=$*Prodotto*$(k + 1, j)$
return $c = a \cdot b$
end

a) Assumendo il criterio di costo uniforme, determinare l'ordine di grandezza del tempo di calcolo richiesto dalla procedura su un input di dimensione n nel caso peggiore.

b) Svolgere l'esercizio richiesto al punto a) assumendo il criterio di costo logaritmico.

2) Consideriamo il problema di calcolare l'espressione

$$b = a_1 + 2a_2 + 2^2 a_3 + \cdots + 2^{n-1} a_n$$

su input $a_1, a_2, \ldots, a_n$ tali che $a_i \in \{1, 2, \ldots, n\}$ per ogni $i = 1, 2, \ldots, n$.
Tale calcolo può essere eseguito dalla seguente procedura:

begin
$b := a_1$
for $j = 2, 3, \ldots, n$ **do**
$b := b + 2^{j-1} a_j$
end

a) Assumendo il criterio di costo logaritmico, determinare in funzione di n l'ordine di grandezza del tempo di calcolo e dello spazio di memoria richiesti dalla procedura nel caso peggiore.

b) Descrivere un algoritmo del tipo "divide et impera" che risolva in problema in tempo $O(n)$ assumendo il criterio di costo uniforme.

c) Assumendo il criterio di costo logaritmico, determinare in funzione di n l'ordine di grandezza del tempo di calcolo e dello spazio di memoria richiesti nel caso peggiore dall'algoritmo descritto al punto b).

3) Considera il problema di calcolare l'espressione

$$(a_1 \cdot a_2 + a_2 \cdot a_3 + \cdots + a_{n-1} \cdot a_n) \,(\text{mod } k)$$

assumendo come input una sequenza di interi $a_1, a_2, \ldots, a_n$ preceduta dal valore $k \geq 1$ e dallo stesso parametro $n \geq 2$ ($k, n \in N$).

a) Descrivere un algoritmo del tipo *divide et impera* per risolvere il problema.

b) Assumendo il criterio di costo uniforme, valutare l'ordine di grandezza del tempo di calcolo e dello spazio di memoria richiesti su un input di lunghezza n.

c) Assumiamo il criterio di costo logaritmico e supponiamo che ogni a_i, $i = 1, 2, \ldots, n$, sia un intero di m bits. Determinare una stima O-grande del tempo di calcolo e dello spazio di memoria in funzione dei parametri n, m e k.

4) Considera il problema di calcolare l'espressione

$$(a^n) mod(r)$$

assumendo come input tre interi positivi a, n, r.

a) Descrivere un algoritmo del tipo *divide et impera* per risolvere il problema.

b) Assumendo il criterio di costo uniforme, valutare l'ordine di grandezza del tempo di calcolo e dello spazio di memoria in funzione di n.

c) Assumiamo il criterio di costo logaritmico e supponiamo che $0 < a < r$. Determinare una stima O-grande del tempo di calcolo e dello spazio di memoria in funzione dei parametri n e r.

5) Dati tre interi $a, b, n \in \mathbb{N}$ considera l'espressione

$$F(a,b,n) = \begin{cases} 1 & \text{se } n = 0 \\ a^n + a^{n-1}b + a^{n-2}b^2 + \cdots + ab^{n-1} + b^n & \text{se } n \geq 1 \end{cases}$$

a) Descrivere una procedura del tipo *divide et impera* che calcola x^n su input $x, n \in \mathbb{N}$

b) Utilizzando la procedura descritta nel punto precedente, definire un algoritmo del tipo *divide et impera* che calcola $F(a, b, n)$ su input a, b, n mediante una sola chiamata ricorsiva (a sé stesso).

c) Assumendo il criterio di costo uniforme, valutare la complessità dell'algoritmo descritto al punto precedente in funzione del parametro n, mostrando che il tempo richiesto è di ordine $\Theta(\log^2 n)$ mentre lo spazio è $\Theta(\log n)$.

Capitolo 11

Programmazione dinamica

Come abbiamo visto nel capitolo precedente, gli algoritmi basati sul metodo "divide et impera" suddividono l'istanza di un problema in sottoistanze di ugual dimensione e quindi risolvono queste ultime in maniera indipendente, solitamente mediante chiamate ricorsive. Vi sono però problemi che possono essere decomposti in sottoproblemi definiti su istanze di dimensione diversa e in questo caso il metodo "divide et impera" non può essere applicato (per lo meno secondo la formulazione che abbiamo descritto nel capitolo precedente). Inoltre, in molti casi interessanti, tali sottoproblemi presentano forti dipendenze tra loro: così un'eventuale procedura ricorsiva che richiama semplicemente se stessa su tutte le sottoistanze per le quali è richiesta la soluzione porta ad eseguire più volte gli stessi calcoli. In alcuni casi il tempo dedicato alla ripetizione di computazioni già eseguite è così elevato da rendere l'algoritmo assolutamente inefficiente.

Un metodo solitamente applicato per risolvere problemi di questo tipo è quello della "programmazione dinamica". Intuitivamente esso consiste nel determinare per una data istanza i di un problema l'insieme $S(i)$ di tutte le sottoistanze da cui dipende il calcolo della soluzione per i. Con lo stesso criterio si stabilisce poi una relazione dipendenza tra i vari elementi di $S(i)$. Quindi, rispettando tale dipendenza, si ricavano le soluzioni delle sottoistanze di $S(i)$ a partire da quelle di dimensione minore; i risultati parziali man mano ottenuti vengono conservati in opportune aree di memoria e utilizzati per determinare le soluzioni relative a istanze di dimensione via via crescente. Così, ogni sottoistanza del problema viene risolta una volta sola e il risultato utilizzato tutte le volte che occorre senza dover ripetere il calcolo. In questo modo, ad un modesto aumento dello spazio richiesto per l'esecuzione dell'algoritmo, corrisponde spesso una drastica riduzione dei tempi di calcolo.

11.1 Un esempio semplice

Un esempio di algoritmo che calcola ripetutamente le soluzioni parziali del problema dato è fornito dalla procedura per determinare i numeri di Fibonacci descritta nella sezione 5.1. Su input $n \in \mathbb{N}$ tale procedura calcola l'n-esimo numero di Fibonacci f_n mediante il seguente programma ricorsivo:

```
Procedura FIB(n)
     if n ≤ 1 then return n
            else begin
                      a := n − 1
                      x := FIB(a)
                      b := n − 2
                      y := FIB(b)
                      return (x + y)
                 end
```

È chiaro che in questa procedura i vari termini della sequenza $f_0, f_1, \ldots, f_{n-2}$ sono calcolati varie volte. Per esempio f_{n-2} viene calcolato sia per determinare $f_n = f_{n-1} + f_{n-2}$, sia per determinare $f_{n-1} = f_{n-2} + f_{n-3}$. Il fenomeno poi cresce man mano che decrescono gli indici della sequenza fino al punto che i primi termini vengono calcolati un numero esponenziale di volte, rendendo così l'algoritmo inutilizzabile anche per piccole dimensioni dell'ingresso. Abbiamo infatti già osservato (sezione 7.2) che questa procedura richiede un tempo di calcolo $\Omega((\frac{\sqrt{5}+1}{2})^n)$.

Il modo più semplice per risolvere il problema è proprio basato sulla programmazione dinamica. I numeri di Fibonacci vengono calcolati a partire da quelli di indice minore e quindi memorizzati in un vettore apposito

$$V = (V[1], V[2], \ldots, V[n]);$$

in questo modo essi sono calcolati una volta sola e quindi riutilizzati quando occorre:

```
Procedura DFIB(n)
     begin
          V[0] := 0
          V[1] := 1
          a := 0
          b := 1
          k := 2
          while k ≤ n do
               begin
                    x := V[a]
                    y := V[b]
                    V[k] := x + y
                    a := b
                    b := k
                    k := k + 1
               end
          return V[n]
     end
```

Come si osserva immediatamente, il tempo di calcolo della procedura `DFIB` è $\Theta(n)$. `DFIB` calcola la stessa funzione di `FIB` in modo straordinariamente più efficiente. Con poca fatica, inoltre, possiamo ottimizzare l'algoritmo osservando che non è necessario mantenere in memoria un vettore di n elementi; infatti, per calcolare ogni f_i è sufficiente ricordare i due coefficienti precedenti, cioè f_{i-1} e f_{i-2}. Si ottiene così la seguente procedura:

```
Procedura OttFIB(n)
     if n ≤ 1 then return n
            else
                 begin
                       x := 0
                       y := 1
                       for k = 2, n do
                             t := y
                             y := x + y
                             x := t
                       return (y)
                 end
```

Anche in questo caso abbiamo un tempo di calcolo $\Theta(n)$, mentre lo spazio di memoria richiesto si riduce a $O(1)$.

11.2 Il metodo generale

Vogliamo ora descrivere in generale come opera la tecnica di programmazione dinamica. Consideriamo un algoritmo ricorsivo descritto dall'insieme di procedure $\{P_1, P_2, \ldots, P_M\}$, in cui P_1 sia la procedura principale, e supponiamo per semplicità che ciascuna procedura abbia un solo parametro formale. Consideriamo ora la famiglia delle coppie $[P_k, x]$ dove k è un intero tale che $1 \leq k \leq M$ e x è un possibile valore del parametro formale di P_k. Diciamo che $[P_k, x]$ *dipende* da $[P_s, y]$ se l'esecuzione della procedura P_k con valore x assegnato al parametro formale richiede almeno una volta la chiamata di P_s con valore y assegnato al corrispondente parametro formale. Conveniamo che $[P_k, x]$ dipenda sempre da se stesso.

Data ora la coppia $[P_1, z]$, consideriamo un ordine lineare $\langle \mathcal{L}[P_1, z], < \rangle$ tale che:

1. $\mathcal{L}[P_1, z] = \{[P_s, y] | [P_1, z]$dipende da$[P_s, y]\}$, cioè $\mathcal{L}[P_1, z]$ è l'insieme delle coppie da cui $[P_1, z]$ dipende.

2. Se $[P_k, x],\ [P_s, y] \in \mathcal{L}[P_1, z]$ e $[P_k, x]$ dipende da $[P_s, y]$, allora $[P_s, y] < [P_k, x]$.

Si osservi che in generale si possono introdurre in $\mathcal{L}[P_1, z]$ vari ordini lineari $<$, compatibili con la richiesta (2); supponiamo qui di averne fissato uno.

Dato $\langle \mathcal{L}[P_1, z], < \rangle$ chiameremo PRIMO l'elemento in $\mathcal{L}[P_1, z]$ più piccolo rispetto all'ordinamento, mentre evidentemente l'elemento massimo è $[P_1, z]$. Dato $I \in \mathcal{L}[P_1, z]$, porremo *Succ*$(I)$ l'elemento successivo ad I nell'ordine totale: poiché *Succ*$([P_1, z])$ non risulta definito, essendo $[P_1, z]$ il massimo dell'ordine lineare, considereremo per completezza un nuovo elemento ND (non definito), ponendo *Succ*$([P_1, z])$ =ND.

L'algoritmo che implementa la procedura P_1 usando una tecnica di programmazione dinamica costruisce nella sua esecuzione un vettore V indiciato in $\mathcal{L}[P_1, z]$, ritornando alla fine $V[P_1, z]$.

```
Procedura DP1(λ)
    (1) Definisce l'ordine lineare L[P1, z]
    (2) I:= PRIMO
    (3) while I ≠ ND do
            begin
            (4) Se I = [Pj, x], esegui la procedura Pj assegnando x al
                parametro formale e interpretando:
                    (a) i comandi iterativi con l'usuale semantica
                    (b) le eventuali chiamate del tipo "b := Ps(l)" come "b := V[Ps, l]"
                    (c) le istruzioni del tipo "return E" come "V[I] := E"
            (5) U := I ; I:= Succ(I)
            end
    (6) return V[U]
```

Adottando questo punto di vista, la programmazione dinamica non è altro che una diversa semantica operazionale della programmazione ricorsiva, anche se c'è qualche grado di libertà nella definizione dell'ordinamento lineare su $\mathcal{L}[P_1, z]$.

In conclusione, per risolvere un problema Π usando il metodo illustrato dobbiamo anzitutto considerare un algoritmo per Π definito da una o più procedure ricorsive e fissare un naturale ordine lineare sulle chiamate di tali procedure. Quindi possiamo applicare il metodo descritto nello schema precedente, introducendo eventualmente qualche ulteriore miglioramento che tenga conto delle simmetrie e delle caratteristiche del problema.

Abbiamo già mostrato come il metodo appena illustrato può essere utilizzato per calcolare i numeri di Fibonacci. Nelle sezioni successive invece presentiamo alcuni classici algoritmi basati sulla programmazione dinamica che risolvono problemi nei quali le relazioni tra le soluzioni alle varie istanze sono più complicate e richiedono il mantenimento in memoria di tutte le soluzioni parziali fino al termine della computazione.

11.3 Moltiplicazione di n matrici

Come è noto, la moltiplicazione di due matrici $A = [A_{ik}]$ e $B = [B_{ik}]$, di dimensione $m \times q$ e $q \times p$ rispettivamente, fornisce la matrice $C = [C_{ik}]$ di dimensione $m \times p$ tale che, per ogni i, k,

$$C_{ik} = \sum_{j=1}^{q} A_{ij} \cdot B_{jk}.$$

Per valutare la complessità di calcolo di questa operazione assumiamo che le due matrici siano a componenti intere e teniamo conto, per semplicità, solo del numero di prodotti eseguiti. Di conseguenza, possiamo supporre che la moltiplicazione delle due matrici sopra considerate richieda $m \cdot q \cdot p$ operazioni elementari.

Il problema che si vuole affrontare è quello di determinare il minimo numero di operazioni necessario a calcolare il prodotto $A_1 \cdot A_2 \cdot \ \ldots \ \cdot A_n$ di n matrici $A_1, A_2, \ldots, A_n$, sapendo che A_k è di dimensione $r_{k-1} \times r_k$ per ogni $k = 1, 2, \ldots, n$.

Siano ad esempio A, B, C matrici rispettivamente di dimensione 3×5, 5×10, 10×2. Il prodotto $A \cdot B \cdot C$ può essere eseguito nei due modi diversi $(A \cdot B) \cdot C$ o $A \cdot (B \cdot C)$, che forniscono lo stesso risultato con un diverso numero di operazioni: $(A \cdot B) \cdot C$ richiede $3 \cdot 5 \cdot 10 + 3 \cdot 10 \cdot 2 = 210$ operazioni,

mentre $A \cdot (B \cdot C)$ ne richiede $3 \cdot 5 \cdot 2 + 5 \cdot 10 \cdot 2 = 130$. Risulta conveniente allora applicare il secondo procedimento.

Tornando al problema iniziale, indichiamo con $M[k, s]$ il numero minimo di operazioni necessario a calcolare $A_k \cdot \ \ldots \ \cdot A_s$. Osservando che $A_k \cdot \ \ldots \cdot \ A_s = (A_k \cdot \ \ldots \cdot \ A_j) \cdot (A_{j+1} \cdot \ \ldots \ \cdot A_s)$ per $k \leq j < s$, una semplice procedura ricorsiva per il calcolo di $M[k, s]$ è suggerita dalla seguente regola:

$$M[k,s] = \begin{cases} 0 & \text{se } k = s \\ Min_{k \leq j < s}\{M[k,j] + M[j+1,s] + r_{k-1} r_j r_s\} & \text{altrimenti} \end{cases}$$

La procedura è la seguente:

```
Procedura COSTO[k, s]
if k = s then return 0
       else
            begin
                  m := ∞
                  for j = k, k + 1, ..., s − 1 do
                     begin
                        A := COSTO [k, j]
                        B := COSTO [j + 1, s]
                        if (A + B + r_{k−1} · r_j · r_s) < m then m := A + B + r_{k−1} · r_j · r_s
                     end
                  return m
            end
```

Il programma che risolve il problema è allora:

```
MOLT-MAT
begin
     for 0 ≤ k ≤ n do READ(r_k)
     z := COSTO [1, n]
     write z
end
```

L'implementazione diretta del precedente algoritmo ricorsivo porta a tempi di calcolo esponenziali; applichiamo allora la tecnica di programmazione dinamica.

Poiché la procedura COSTO richiama solo se stessa, potremo senza ambiguità scrivere $\mathcal{L}[k, s]$ anziché $\mathcal{L}$(COSTO[k, s]).

Poiché $[k, s]$ dipende da $[k', s']$, se $k \leq k' \leq s' \leq s$ allora si ha

$$\mathcal{L}[1, n] = \{[k, s] |\ 1 \leq k \leq s \leq n\}.$$

Un possibile ordine lineare compatibile con la precedente nozione di dipendenza è il seguente:

$$[k, s] \leq [k', s'] \ \text{ se e solo se } \ \ s - k < s' - k' \ \text{ oppure } \ s' - k' = s - k \ \text{ e } \ k \leq k'.$$

Otteniamo in questo caso:

$$\text{PRIMO} = [1, 1]$$

$$Succ[k,s] = \begin{cases} [k+1, s+1] & \text{se } s < n \\ [1, n-k+2] & \text{se } s = n \end{cases}$$

$$\text{ND} = [1, n+1]$$

La procedura di programmazione dinamica risulta allora:

```
Procedura DCOSTO [1, n]
begin
     [k, s] := [1, 1]
     while [k, s] ≠ [1, n + 1] do
          begin
               if k = s then V[k, s] := 0
                        else
                             begin
                                  m := ∞
                                  for j = k, k + 1, ..., s − 1 do
                                      begin
                                           A := V[k, j]
                                           B := V[j + 1, s]
                                           if (A + B + r_{k−1} · r_j · r_s) < m then
                                                m := A + B + r_{k−1} · r_j · r_s
                                      end
                                  V[k, s] := m
                             end
               u := [k, s]
               [k, s] := Succ[k, s]
          end
     return V[u]
end
```

Per quanto riguarda il tempo di calcolo (con criterio uniforme) si osserva che il ciclo `while` viene percorso tante volte quante sono le possibili coppie $[k, s]$ con $1 \leq k \leq s \leq n$.

Fissata la coppia $[k, s]$, l'istruzione più costosa nel ciclo `while` è il ciclo `for` che esegue $(s - k) \cdot O(1)$ passi, mentre il tempo richiesto dalle altre istruzioni è al più costante. Questo significa che ogni esecuzione del ciclo `while` richiede un tempo $\Theta(s - k)$, dove $[k, s]$ è l'elemento corrente. Osserva che vi sono $n - 1$ elementi $[k, s]$ tali che $s - k = 1$, ve ne sono $n - 2$ tali che $s - k = 2$ e così via; infine vi è un solo elemento $[k, s]$ tale che $s - k = n - 1$. Quindi il tempo complessivo è dato dalla somma

$$\sum_{[k,s]} \Theta(s-k) = \sum_{i=1}^{n-1} (n-i)\Theta(i) = \Theta(n^3).$$

Il tempo complessivo è allora $\Theta(n^3)$, mentre lo spazio è essenzialmente quello richiesto per memorizzare il vettore $V[k, s]$, quindi $\Theta(n^2)$.

11.4 Chiusura transitiva

Il problema che consideriamo in questa sezione riguarda il calcolo della chiusura transitiva di un grafo. Dato un grafo orientato $G = \langle V, E \rangle$, si vuole determinare il grafo (orientato) $G^* = \langle V, E^* \rangle$ tale che,

per ogni coppia di nodi $u, v \in V$, esiste il lato (u, v) in G^* se e solo se esiste in G un cammino da u a v di lunghezza maggiore di 0. In questa sezione consideriamo solo cammini di lunghezza maggiore di 0; la trattazione può essere facilmente estesa in modo da comprendere anche i cammini di lunghezza 0 (ovvero quelli della forma (v), dove $v \in V$).

Un algoritmo classico per risolvere il problema è basato sul calcolo di una famiglia di coefficienti booleani che assumono valore 1 o 0 a seconda se esiste o meno un cammino tra due nodi passante per un certo insieme di vertici.

Siano $v_1, v_2, \ldots, v_n$ i nodi del grafo G. L'idea è quella di verificare, per ogni coppia di nodi v_i, v_j, se esiste un lato da v_i a v_j oppure un cammino da v_i a v_j passante per il nodo v_1; poi, se esiste un cammino da v_i a v_j passante al più per nodi di indice 1 o 2; quindi se esiste un cammino da v_i a v_j passante per nodi di indice minore o uguale a 3, e così via. Si tratta quindi di eseguire n cicli: al k-esimo ciclo si verifica, per ogni coppia di nodi v_i, v_j, se esiste un cammino da v_i a v_j passante per nodi di indice minore o uguale a k (escludendo i due valori i e j). Tale verifica può essere eseguita tenendo conto dei risultati forniti dal ciclo precedente. Infatti, esiste un cammino da v_i a v_j passante per nodi di indice minore o uguale a k se e solo se si verifica uno dei fatti seguenti:

1. esiste un cammino da v_i a v_j passante per nodi di indice minore o uguale a $k-1$, oppure
2. esiste un cammino da v_i a v_k e uno da v_k a v_j entrambi passanti per vertici di indice minore o uguale a $k-1$.

Per ogni coppia di indici $i, j \in \{1, 2, \ldots, n\}$ definiamo la famiglia di coefficienti C_{ij}^k, dove $k \in \{0, 1, 2, \ldots, n\}$, nel modo seguente:

$$C_{ij}^0 = \begin{cases} 1 & \text{se } (v_i, v_j) \in E, \\ 0 & \text{altrimenti,} \end{cases}$$

mentre, per ogni $k \in \{1, 2, \ldots, n\}$,

$$C_{ij}^k = \begin{cases} 1 & \text{se } (v_i, v_j) \in E \text{ oppure esiste in } G \text{ un cammino da} \\ & v_i \text{ a } v_j \text{ che passa per nodi di indice } t \text{ tale che } t \leq k, \\ 0 & \text{altrimenti.} \end{cases}$$

Nota che $[C_{ij}^0]$ coincide di fatto con la matrice di adiacenza del grafo di ingresso.

Per il ragionamento precedente i coefficienti C_{ij}^k, al variare di i e j in $\{1, 2, \ldots, n\}$, sono legati ai valori C_{ij}^{k-1} dalla seguente equazione:

$$C_{ij}^k = C_{ij}^{k-1} \vee (C_{ik}^{k-1} \wedge C_{kj}^{k-1}) \tag{11.1}$$

Chiaramente $C_{ij}^n = 1$ se e solo se in G esiste un cammino da v_i a v_j. Possiamo allora facilmente descrivere un algorimo che calcola inizialmente i coefficienti C_{ij}^0 e quindi, usando l'equazione (11.1), tutti i successivi: per ogni $k = 1, 2, \ldots, n$ tutti i coefficienti C_{ij}^k, con $i, j = 1, 2, \ldots, n$, possono essere ottenuti dai C_{ij}^{k-1}. Questo calcolo può essere ovviamente eseguito mantenendo due matrici di dimensione $n \times n$, una per conservare i valori C_{ij}^{k-1}, l'altra per i corrispondenti C_{ij}^k. Tuttavia, osserviamo che per ogni i, j, k valgono le seguenti identità:

$$C_{ik}^{k-1} = C_{ik}^k, \quad C_{kj}^{k-1} = C_{kj}^k.$$

Questo significa che possiamo usare una sola matrice per mantenere entrambe le famiglie di coefficienti.

L'algoritmo complessivo è quindi descritto dalla seguente procedura che calcola la matrice di coefficienti booleani $[C_{ij}]_{i,j=1,2,\ldots,n}$. Tale matrice coincide inizialmente con la matrice di adiacenza del grafo G mentre, al termine della computazione, rappresenta la matrice di adiacenza della sua chiusura transitiva G^*.

```
begin
      for i = 1, 2, ..., n do
          for j = 1, 2, ..., n do
              if (v_i, v_j) ∈ E then C_ij := 1
                                else C_ij := 0
      for k = 1, 2, ..., n do
          for i = 1, 2, ..., n do
              for j = 1, 2, ..., n do
                                if C_ij = 0 then C_ij := C_ik ∧ C_kj
end
```

È facile verificare che, assumendo il criterio di costo uniforme, l'algoritmo richiede un tempo di calcolo $\Theta(n^3)$ e uno spazio di memoria $\Theta(n^2)$ su ogni grafo di ingresso di n nodi.

Esercizio

Esegui l'algoritmo sopra descritto sul grafo $G = (\{1,2,3\}, \{(1,2),(2,3),(3,1)\})$. Determina in particolare le matrici di coefficienti $[C^k_{ij}]_{i,j=1,2,3}$ per $k = 0, 1, 2, 3$.

11.5 Cammini minimi

Un altro problema classico che si può risolvere mediante programmazione dinamica riguarda il calcolo dei cammini di peso minimo che connettono i nodi in un grafo pesato.

Consideriamo un grafo diretto $G = \langle V, E\rangle$ e una funzione costo $w : E \longrightarrow \mathbb{Q}$ tale che $w(e) \geq 0$ per ogni $e \in E$. Per ogni cammino ℓ in G, $\ell = (v_1, v_2, \ldots, v_m)$, chiamiamo peso (o costo) di ℓ la somma dei costi dei suoi lati:

$$c(\ell) = \sum_{i=1}^{m-1} w(v_i, v_{i+1})$$

Per ogni coppia di nodi $u, v \in V$ si vuole determinare un cammino ℓ di peso minimo tra tutti i cammini da u a v, insieme al suo costo $c(\ell)$.

Siano $v_1, v_2, \ldots, v_n$ i nodi di G. Il problema può essere risolto calcolando le matrici $D = [d_{ij}]_{i,j=1,2,\ldots,n}$ e $P = [p_{ij}]_{i,j=1,2,\ldots,n}$ le cui componenti sono definite nel modo seguente: per ogni coppia di indici distinti i, j, se esiste un cammino da v_i a v_j in G, d_{ij} rappresenta il costo del cammino di peso minimo che congiunge v_i e v_j e p_{ij} è il predecessore di v_j in tale cammino; se invece non esiste un cammino da v_i a v_j allora d_{ij} assume un valore convenzionale ∞, superiore al peso di ogni cammino in G, e p_{ij} assume il valore indefinito $\perp$.

Conoscendo la matrice P è possibile determinare un cammino di peso minimo da v_i a v_j semplicemente scorrendo in modo opportuno la i- esima riga della matrice: se $k_1 = p_{ij}$ allora v_{k_1} è il penultimo nodo del cammino, se $k_2 = p_{ik_1}$ allora v_{k_2} è il terzultimo, e così via fino a determinare il primo nodo, ovvero v_i.

Abbiamo così definito il problema per grafi con pesi non negativi. L'algoritmo che presentiamo tuttavia risolve il problema nel caso più generale di grafi con pesi di segno qualsiasi, purché questi non formino cicli di costo negativo. Osserviamo che se esiste un ciclo di peso negativo lo stesso problema non è ben definito.

Il metodo utilizzato per determinare la soluzione è simile a quello descritto nella sezione precedente per calcolare la chiusura transitiva di un grafo. Per ogni tripla di indici $i, j, k \in \{1, 2, \ldots, n\}$ definiamo il coefficiente c_{ij}^k come il costo del cammino di peso minimo da v_i a v_j passante per nodi di indice al più uguale a k (esclusi i e j). Chiaramente $d_{ij} = c_{ij}^n$. Inoltre definiamo i coefficienti c_{ij}^0 nel modo seguente:

$$c_{ij}^0 = \begin{cases} w(v_i, v_j) & \text{se } i \neq j \text{ e } (v_i, v_j) \in E \\ 0 & \text{se } i = j \\ \infty & \text{altrimenti} \end{cases}$$

Anche per questa famiglia di coefficienti possiamo definire un'equazione, analoga alla (11.1), che permette di calcolare i valori c_{ij}^k, per $i, j = 1, 2, \ldots, n$, conoscendo i c_{ij}^{k-1}:

$$c_{ij}^k = \min\{c_{ij}^{k-1}, c_{ik}^{k-1} + c_{kj}^{k-1}\}. \tag{11.2}$$

Basta osservare infatti che se ℓ è un cammino di peso minimo da v_i a v_j passante per nodi di indice minore o uguale a k allora si verifica uno dei due fatti seguenti:

1. ℓ passa solo per nodi di indice minore o uguale a $k-1$, oppure

2. ℓ è composto da due cammini che vanno rispettivamente da v_i a v_k e da v_k a v_j, entrambi di peso minimo tra tutti i cammini (congiungenti i rispettivi estremi) passanti per nodi di indice minore o uguale a $k-1$.

Nel primo caso il costo di ℓ è c_{ij}^{k-1} mentre nel secondo è $c_{ik}^{k-1} + c_{kj}^{k-1}$; inoltre in quest'ultimo il predecessore di v_j in ℓ equivale al predecessore di v_j nel cammino di costo minimo da v_k a v_j passante per nodi di indice minore o uguale a $k-1$.

Applicando l'equazione (11.2) possiamo allora descrivere un algoritmo del tutto simile a quello presentato nella sezione precedente. Anche in questo caso possiamo limitarci a mantenere una sola matrice di valori c_{ij}^k poichè valgono le identità

$$c_{ik}^{k-1} = c_{ik}^k \text{ e } c_{kj}^{k-1} = c_{kj}^k$$

per ogni tripla di indici i, j, k. Così l'algoritmo calcola la matrice $C = [c_{ij}]$ dei costi e quella $B = [b_{ij}]$ dei predecessori dei cammini minimi che, al termine della computazione, coincideranno con le matrici D e P rispettivamente.

```
begin
    for i = 1, 2, ..., n do
        for j = 1, 2, ..., n do
            if i = j then { c_ii := 0
                          { b_ii := i
                    else if (v_i, v_j) ∈ E then { c_ij := w(v_i, v_j)
                                               { b_ij := i
                                         else { c_ij := ∞
                                              { b_ij := ⊥
    for k = 1, 2, ..., n do
        for i = 1, 2, ..., n do
            for j = 1, 2, ..., n do
                    if c_ik + c_kj < c_ij then { c_ij := c_ik + c_kj
                                               { b_ij := b_kj
end
```

Concludiamo osservando che, assumendo il criterio di costo uniforme, il tempo di calcolo richiesto dall'algoritmo su ogni grafo di input di n nodi è $\Theta(n^3)$, mentre lo spazio di memoria è $\Theta(n^2)$.

Capitolo 12

Algoritmi greedy

Una delle tecniche più semplici per la progettazione di algoritmi di ottimizzazione è chiamata tecnica *greedy*. Letteralmente questo termine significa "ingordo", ma nel seguito preferiremo tradurlo "miope". Intuitivamente, questo metodo costruisce la soluzione di un problema di ottimizzazione mediante una successione di passi durante ciascuno dei quali viene scelto un elemento "localmente" migliore; in altre parole a ciascun passo la scelta migliore viene compiuta in un ambito limitato, senza controllare che il procedimento complessivo porti effettivamente al calcolo di una soluzione ottima per il problema.

Questa strategia, se da un lato permette solitamente di ottenere algoritmi semplici e facilmente implementabili, dall'altro può portare alla definizione di procedure che non forniscono sempre la soluzione ottima. In questo capitolo vogliamo studiare le proprietà degli algoritmi di questo tipo e verificare in quali casi è possibile garantire che la soluzione costruita sia effettivamente la migliore.

12.1 Problemi di ottimizzazione

Per esprimere questi concetti in maniera precisa, introduciamo la nozione di sistema di indipendenza.

Un *sistema di indipendenza* è una coppia $\langle E, F\rangle$ nella quale E è un insieme finito, F è una famiglia di sottoinsiemi di E chiusa rispetto all'inclusione; in altre parole, $F \subseteq 2^E$ (qui e nel seguito denoteremo con 2^E la famiglia di tutti i sottoinsiemi di E) e inoltre

$$A \in F \wedge B \subseteq A \Rightarrow B \in F$$

È evidente che, per ogni insieme finito E, la coppia $\langle E, 2^E\rangle$ forma un sistema di indipendenza.

Esempio 12.1
Sia $G = (V, E)$ un grafo non orientato; diciamo che un insieme $A \subseteq E$ forma una foresta se il grafo (V, A) è privo di cicli. Denotiamo quindi con $\mathcal{F}_G$ l'insieme delle foreste di G, ovvero

$$\mathcal{F}_G = \{A \subseteq E \mid A \text{ forma una foresta}\}$$

È facile verificare che la coppia $\langle E, \mathcal{F}_G\rangle$ è un sistema di indipendenza.

Viceversa, per ogni $A \subseteq E$, sia V_A l'insieme dei vertici $v \in V$ che sono estremi di un lato in A e diciamo che A forma un albero se il grafo (V_A, A) è connesso e privo di cicli: denotando con $\mathcal{T}_G$ la famiglia degli alberi di G, ovvero $\mathcal{T}_G = \{A \subseteq E \mid A \text{ forma un albero}\}$, si verifica facilmente che $\langle E, \mathcal{T}_G\rangle$ non è un sistema di indipendenza. ∎

Esempio 12.2
Sia sempre $G = (V, E)$ un grafo non orientato. Diciamo che un insieme $A \subseteq E$ forma un matching se, per ogni coppia di lati distinti $\alpha, \beta \in A$, α e β non hanno nodi in comune. Denotiamo inoltre con $\mathcal{M}_G$ la famiglia dei sottoinsiemi di E che formano un matching. È facile verificare che $\langle E, \mathcal{M}_G\rangle$ è un sistema di indipendenza.

Analogamente, diciamo che un insieme $S \subseteq V$ forma una clique di G se, per ogni coppia di nodi distinti $s, u \in S$, il lato $\{s, v\}$ appartiene a E. Denotiamo con $\mathcal{C}_G$ la famiglia delle clique di G. Si può verificare anche in questo caso che $\langle V, \mathcal{C}_G \rangle$ forma un sistema di indipendenza. ∎

Vari problemi di ottimizzazione possono essere definiti in modo naturale usando sistemi di indipendenza pesati nei quali cioè ogni elemento dell'insieme base è dotato di un peso. Una *funzione peso* su un dato sistema di indipendenza $\langle E, F \rangle$ è un'arbitraria funzione $w : E \to \mathbb{R}^+$, dove $\mathbb{R}^+$ è l'insieme dei reali non negativi. Tale funzione può essere ovviamente estesa ai sottoinsiemi di E ponendo, per ogni $A \subseteq E$, $w(A) = \sum_{x \in A} w(x)$. Possiamo allora formulare in modo preciso un problema di massimo:

> Istanza: un sistema di indipendenza $\langle E, F \rangle$ e una funzione peso $w : E \to \mathbb{R}^+$.
> Soluzione : un insieme $M \in F$ tale che $w(M)$ sia massimo (ovvero $A \in F \Rightarrow w(A) \leq w(M)$).

In modo analogo possiamo definire un problema di minimo. In questo caso, dato una sistema di indipendenza $\langle E, F \rangle$, diciamo che un insieme $A \in F$ è *massimale* se non esiste $B \in F$ diverso da A che include A, ovvero, per ogni $B \in F$, $A \subseteq B \Rightarrow A = B$.

> Istanza: un sistema di indipendenza $\langle E, F \rangle$ e una funzione peso $w : E \to \mathbb{R}^+$.
> Soluzione : un insieme $A \in F$ massimale che sia di peso minimo (ovvero, tale che per ogni $B \in F$ massimale $w(A) \leq w(B)$).

Definiamo ora l'algoritmo greedy per il problema di massimo; tale algoritmo è definito dalla seguente procedura.

```
Procedure MIOPE(E, F, w)
begin
      S := ∅
      Q := E
      while Q ≠ ∅ do
                  begin
                        determina l'elemento m di peso massimo in Q
                        Q := Q − {m}
                        if S ∪ {m} ∈ F then S := S ∪ {m}
                  end
      return S
end
```

Fissata una istanza del problema di ottimizzazione, ovvero una tripla E, F, w definita come sopra, la precedente procedura fornisce in uscita un insieme S che appartiene certamente a F (è quindi una soluzione *ammissibile*), ma non è necessariamente ottimo nel senso che può non rappresentare un insieme di peso massimo in F.

Si pongono allora, a questo livello di generalità, due problemi:

1. qual è il tempo di calcolo dell'algoritmo greedy, cioè quanti passi di calcolo devono essere compiuti avendo come ingresso un insieme E di n elementi? A questo proposito osserviamo che l'input dell'algoritmo sarà qui costituito dal solo insieme E e dalla funzione peso w: si suppone che F sia automaticamente definito in maniera implicita mediante una opportuna regola e sia comunque

disponibile una routine per verificare quando un insieme $A \subseteq E$ appartiene a F. La famiglia F potrebbe infatti contenere un numero di elementi esponenziale in n e quindi la sua specifica diretta sarebbe improponibile.

2. In quali casi l'algoritmo greedy fornisce effettivamente una soluzione ottima? Qualora l'algoritmo non fornisca la soluzione ottima, si pone un terzo problema, ovvero quello di valutare la bontà della soluzione prodotta. Questo porta a studiare una classe di algoritmi, detti di approssimazione, che in generale non forniscono la soluzione migliore a un dato problema ma ne producono una che approssima quella richiesta. In molti casi il calcolo della soluzione ottima è troppo costoso in termini di tempo e ci si accontenta di una soluzione approssimata, purchè ovviamente quest'ultima sia calcolabile in un tempo accettabile.

Concludiamo osservando che un algoritmo analogo può essere descritto per il problema di minimo. Esso è definito dalla seguente procedura:

```
Procedure MIOPE-MIN(E, F, w)
begin
      S := ∅
      Q := E
      while Q ≠ ∅ do
                  begin
                        determina l'elemento m di peso minimo in Q
                        Q := Q − {m}
                        if S ∪ {m} ∈ F then S := S ∪ {m}
                  end
      return S
end
```

Anche per tale algoritmo valgono le osservazioni fatte a proposito della procedura MIOPE.

12.2 Analisi delle procedure greedy

Presentiamo ora una analisi dei tempi di calcolo richiesti dall'algoritmo MIOPE descritto nella sezione precendente. Ovviamente, visto il livello di generalità del problema, il tempo di calcolo ottenuto dipenderà dal sistema di indipendenza $\langle E, F\rangle$ di ingresso e non semplicemente dalla dimensione di E.

Una prima soluzione si può ottenere rappresentando l'insieme $E = \{l_1, l_2, \cdots, l_n\}$ mediante un vettore $Q = (Q[1], Q[2], \ldots, Q[n])$ dove, inizialmente, $Q[i] = l_i$ per ogni i. Indichiamo con SORT(Q) una funzione che restituisce il vettore Q ordinato in modo tale che i pesi dei lati formino una progressione non crescente, ovvero $w(Q[1]) \geq w(Q[2]) \geq \cdots \geq w(Q[n])$. La procedura può allora essere riscritta nella forma seguente:

```
Procedure MIOPE
begin
      S := ∅
      Q := (l_1, l_2, ··· , l_n)
      Q := SORT(Q)
      for i = 1, 2, ..., n do
                        if S ∪ {Q[i]} ∈ F then S := S ∪ {Q[i]}
end
```

Come si vede la procedura prevede due passi principali: l'ordinamento di un vettore di n elementi e n test per verificare se un insieme $X \subseteq E$ appartiene a F. Possiamo chiaramente eseguire il primo passo in un tempo $O(n \log n)$. Il secondo tuttavia dipende dal particolare sistema di indipendenza considerato in ingresso. Se comunque assumiamo di poter verificare l'appartenenza $X \in F$ in un tempo $C(n)$, il costo complessivo di questo controllo non è superiore a $O(n \cdot C(n))$. Possiamo quindi concludere affermando che la procedura MIOPE richiede al più un tempo $O(n \log n + nC(n))$.

12.3 Matroidi e teorema di Rado

Diamo in questa sezione una soluzione parziale alla seconda questione che ci siamo posti: in quali casi l'algoritmo greedy fornisce la soluzione ottima?

Il nostro obiettivo è quello di caratterizzare la classe dei sistemi di indipendenza per i quali l'algoritmo greedy fornisce la soluzione ottima qualunque sia la funzione peso considerata. Dimostreremo (teorema di Rado) che un sistema di indipendenza verifica la proprietà precedente se e solo se esso è un matroide.

Un sistema di indipendenza $\langle E, F\rangle$ è detto *matroide* se, per ogni $A, B \in F$ tali che $|B| = |A| + 1$ (qui $|X|$ indica la cardinalità di un insieme X), allora esiste $b \in B - A$ per cui $A \cup \{b\} \in F$. Per esempio, è facile verificare che, per ogni insieme finito E, la coppia $\langle E, 2^E\rangle$ forma un matroide.

La nozione di matroide è stata introdotta nel 1935 da Birkhoff e altri per generalizzare il concetto di dipendenza lineare. Questa nozione ha trovato proficue applicazioni in vari settori, dalla teoria dei grafi agli algoritmi, e può essere considerata un ponte tra l'algebra lineare e la matematica combinatoria.

Esempio 12.3
Sia E un insieme finito di vettori di uno spazio vettoriale V. Sia F la famiglia di sottoinsiemi di E formati da vettori linearmente indipendenti. Si può verificare facilmente che $\langle E, F\rangle$ forma un matroide, detto *matroide vettoriale*. ∎

L'esempio più importante di matroide è tuttavia fornito dal sistema di indipendenza $\langle E, \mathcal{F}_G\rangle$ definito nell'Esempio 12.1. Per dimostrare che $\langle E, \mathcal{F}_G\rangle$ è un matroide ricordiamo prima una proprietà delle foreste.

Lemma 12.1 *Ogni foresta di n nodi formata da k alberi possiede $n - k$ lati.*

Omettiamo la dimostrazione di questo lemma che può essere facilmente provato, per ogni n fissato, ragionando per induzione sul numero m di lati. Nota che per $m = 0$ la proprietà è ovvia perché in questo caso ogni nodo forma un albero (e quindi $n = k$). Inoltre, è noto che ogni albero di n nodi possiede $n-1$ lati, per cui $n - 1$ è il massimo numero di lati in una foresta di n nodi. Osserva anche che aggiungendo un lato ℓ a una foresta U si ottiene una nuova foresta solo se ℓ congiunge due nodi appartenenti ad alberi distinti di U.

Proposizione 12.2 *Per ogni grafo non orientato $G = (V, E)$, la coppia $\langle E, \mathcal{F}_G \rangle$ è un matroide.*

Dimostrazione. Dall'Esempio 12.1 sappiamo che $\langle E, \mathcal{F}_G \rangle$ è un sistema di indipendenza. Per provare che tale sistema è anche un matroide consideriamo due insiemi qualsiasi $A, B \in \mathcal{F}_G$ tali che $|B| = |A| + 1$. Dobbiamo provare che esiste un lato $\ell \in B - A$ tale che $A \cup \{\ell\} \in \mathcal{F}_G$. Per il lemma precedente la foresta (V, A) possiede un albero in più rispetto alla foresta (V, B). Quindi esistono due nodi $u, v \in V$ che appartengono allo stesso albero in (V, B) ma si trovano in alberi distinti in (V, A). Di conseguenza in (V, B) esiste un cammino da u a v, cioè una sequenza di nodi $\{b_1, b_2 \ldots, b_r\}$ tali che $b_1 = u$, $b_r = v$ e $\{b_i, b_{i+1}\} \in B$ per ogni $i = 1, \ldots, r-1$. In tale sequenza devono esistere per forza due nodi consecutivi b_j, b_{j+1} che appartengono ad alberi distinti di (V, A) (altrimenti anche u e v sarebbero nello stesso albero). Ne segue che il lato $\{b_j, b_{j+1}\} \in B$ congiunge due alberi distinti di (V, A) e quindi non può formare cicli. Questo prova che aggiungendo il lato $\{b_j, b_{j+1}\}$ ad A si ottiene un insieme appartenente a $\mathcal{F}_G$. ∎

La coppia $\langle E, \mathcal{F}_G \rangle$ è anche chiamata *matroide grafico*.

Un risultato interessante, che fornisce una interpretazione algoritmica dei matroidi, è il seguente:

Teorema 12.3 *(Rado) Dato un sistema di indipendenza $\langle E, F \rangle$, le seguenti proposizioni sono equivalenti:*

a) per ogni funzione peso $w : E \to \mathbb{R}^+$, l'algoritmo MIOPE fornisce una soluzione ottima al problema di massimo su input E, F, w;

b) $\langle E, F \rangle$ è un matroide.

Dimostrazione. Proviamo innanzitutto che se $\langle E, F \rangle$ non è un matroide allora esiste una funzione peso $w : E \to \mathbb{R}^+$ per la quale l'algoritmo greedy non fornisce la soluzione ottima. Infatti, poiché $\langle E, F \rangle$ non è un matroide, esistono due insiemi $A, B \in F$ tali che, per qualche $k \in \mathbb{N}$,

$$|A| = k, \;\; |B| = k + 1, \;\; \text{e inoltre } b \in B - A \Rightarrow A \cup \{b\} \notin F$$

Definiamo ora una funzione peso w nel modo seguente. Scelto $\alpha > 1$, poniamo per ogni $x \in E$:

$$w(x) = \begin{cases} \alpha & \text{se } x \in A \\ 1 & \text{se } x \in B - A \\ 0 & \text{se } x \in (A \cup B)^c \end{cases}$$

Assegnata tale funzione peso, l'algoritmo greedy fornirà una soluzione S, formata da tutti gli elementi in A (i quali, avendo peso maggiore, verranno selezionati per primi) più , eventualmente, un insieme di elementi $C \subseteq (A \cup B)^c$. Nota che in S non vi possono essere elementi di $B - A$ in quanto, per ogni $b \in B - A$, abbiamo $A \cup \{b\} \notin F$. Posto $t = |A \cap B|$, abbiamo

$$w(S) = w(A \cup C) = w(A) + w(C) = \alpha \cdot |A| = \alpha \cdot k$$

$$w(B) = w(B - A) + w(A \cap B) = (k + 1 - t) + \alpha \cdot t$$

Ne segue allora che

$$w(S) < w(B) \iff \alpha \cdot k < k + 1 - t + \alpha \cdot t \iff \alpha < 1 + \frac{1}{k - t}$$

Quindi, se scegliamo α tale che

$$1 < \alpha < 1 + \frac{1}{k - t},$$

si verifica che la soluzione S costruita dall'algoritmo greedy non è ottima.

Viceversa, dimostriamo ora che, se $\langle E, F\rangle$ è un matroide, comunque si scelga una funzione peso $w : E \to \mathbb{R}^+$, l'algoritmo greedy restituisce la soluzione ottima. Sia infatti $S = \{b_1, b_2, \ldots, b_n\}$ la soluzione fornita dall'algoritmo, con $w(b_1) \geq w(b_2) \geq \cdots \geq w(b_n)$. Sia $A = \{a_1, a_2, \ldots, a_m\}$ un qualunque elemento di F, con $w(a_1) \geq w(a_2) \geq \cdots \geq w(a_m)$.

Innanzitutto verifichiamo che $m \leq n$. Infatti se fosse $n < m$, essendo $\langle E, F\rangle$ un matroide, esisterebbe $a_j \in A - S$ tale che $S \cup \{a_j\} \in F$. Inoltre, tutti i sottoinsiemi di $S \cup \{a_j\}$ apparterrebbero a F, e di conseguenza l'algoritmo non avrebbe scartato a_j ma lo avrebbe inserito nella soluzione, restituendo l'insieme $S \cup \{a_j\}$ invece di S.

Quindi $m \leq n$ e dimostriamo allora che $w(a_i) \leq w(b_i)$ per ogni $i = 1, 2, \ldots, m$. Infatti, per assurdo, sia k il primo intero tale che $w(a_k) > w(b_k)$. Nota che l'insieme $D = \{b_1, b_2, \ldots, b_{k-1}\}$ appartiene a F e inoltre $|D| + 1 = |\{a_1, a_2, \ldots, a_k\}|$. Di conseguenza, essendo $\langle E, F\rangle$ un matroide, esiste un intero j, $1 \leq j \leq k$, tale che $a_j \notin D$ e $D \cup \{a_j\} \in F$. Poiché l'algoritmo greedy sceglie al passo k-esimo l'elemento di peso massimo tra quelli disponibili, abbiamo $w(b_k) \geq w(a_j)$; d'altro lato, essendo $j \leq k$, abbiamo $w(a_j) \geq w(a_k)$ e quindi $w(b_k) \geq w(a_j) \geq w(a_k)$, contro l'ipotesi $w(a_k) > w(b_k)$. Questo prova che $w(A) \leq w(S)$ e quindi la soluzione fornita dall'algoritmo è ottima. ∎

Anche per il problema di minimo è possibile provare che l'algoritmo greedy fornisce la soluzione ottima sui matroidi.

Corollario 12.4 *Se un sistema di indipendenza $\langle E, F\rangle$ è un matroide allora, per ogni funzione peso $w : E \to \mathbb{R}^+$, l'algoritmo MIOPE-MIN fornisce una soluzione ottima al problema di minimo (su input E, F, w).*

Dimostrazione. Innazitutto è facile verificare che in ogni matroide $\langle E, F\rangle$ gli insiemi $A \in F$ massimali hanno la stessa cardinalità. Sia quindi m la cardinalità degli insiemi massimali in F. Inoltre, data una funzione peso $w : E \to \mathbb{R}^+$, fissa $p = \max\{w(x) \mid x \in E\}$ e definisci la funzione $w' : E \to \mathbb{R}^+$ ponendo $w'(x) = p - w(x)$, per ogni $x \in E$. Allora, per ogni $A \in F$ massimale, abbiamo $w'(A) = mp - w(A)$ e quindi $w'(A)$ è massimo se e solo se $w(A)$ è minimo. Per il teorema precedente la procedura MIOPE su input E, F, w' determina l'elemento $S \in F$ di peso massimo rispetto a w'. Tuttavia è facile verificare che S è anche l'output della procedura MIOPE-MIN su input E, F, w. Di conseguenza, per l'osservazione precedente, S è anche insieme massimale in F di peso minimo rispetto a w.

∎

Esercizi

1) Dato un grafo non orientato $G = \langle V, E\rangle$ nel quale V è l'insieme dei nodi ed E quello degli archi, definiamo la seguente famiglia di sottoinsiemi di E:

$$F = \{A \subseteq E \mid \exists v \in V \text{ tale che ogni lato } \alpha \in A \text{ è incidente a } v\}$$

Per ipotesi assumiamo $\emptyset \in F$.

a) La coppia $\langle E, F\rangle$ forma un sistema di indipendenza?

b) La coppia $\langle E, F\rangle$ forma un matroide?

c) Considera il problema di determinare l'elemento di peso massimo in F assumendo per istanza un grafo G con pesi positivi associati agli archi. Descrivere un algoritmo greedy per tale problema.

d) L'algoritmo descritto al punto c) determina sempre la soluzione ottima?

2) Ricordiamo che in un grafo non orientato $G = \langle V, E\rangle$ (dove V è l'insieme dei nodi e E quello dei lati) una *clique* è un sottoinsieme $C \subseteq V$ tale che, per ogni $u, v \in C$, se $u \neq v$ allora $\{u, v\} \in E$. Sia F_G la famiglia di tutte le clique di G, cioè

$$F_G = \{A \subseteq V \mid \forall u, v \in A,\ u \neq v \Rightarrow \{u, v\} \in E\}$$

a) La coppia $\langle V, F_G\rangle$ forma un sistema di indipendenza?

b) La coppia $\langle V, F_G\rangle$ forma un matroide?

c) Dato un grafo non orientato $G = \langle V, E\rangle$ e una funzione peso $w : V \longrightarrow \mathbb{R}^+$, ogni insieme $A \subseteq V$ ammette un peso $w(A)$ definito da

$$w(A) = \sum_{x \in A} w(x).$$

Descrivere una procedura greedy che cerca di determinare un insieme $C \in F_G$ di peso massimo in F_G. La soluzione prodotta dall'algoritmo è sempre ottima?

3) Svolgere l'analisi dell'algoritmo greedy per il problema di minimo introdotto nella sezione 12.1.

12.4 L'algoritmo di Kruskal

Un classico problema di ottimizzazione su grafi consiste nel determinare un albero di copertura di peso minimo in un grafo assegnato. In questa sezione presentiamo uno dei principali algoritmi per risolvere tale problema, noto come algoritmo di Kruskal. Questa procedura fornisce un esempio rilevante di algoritmo greedy e nello stesso tempo rappresenta una applicazione delle operazioni UNION e FIND studiate nella sezione 9.7.

Ricordiamo innanzitutto che un albero di copertura (spanning tree) di un grafo non orientato, connesso $G\langle V, E\rangle$ è un albero T, sottografo di G, che connette tutti i nodi del grafo; formalmente quindi si tratta di un albero $T = \langle V', E'\rangle$ tale che $V' = V$ e $E' \subseteq E$.

Se G è dotato di una funzione peso $w : E \to \mathbb{Q}$, il costo di un albero di copertura $T = \langle V, E'\rangle$ di G è semplicemente la somma dei costi dei suoi lati:

$$w(T) = \sum_{l \in E'} w(l)$$

Il problema che consideriamo è quello di determinare un albero di copertura di peso minimo per G (nota che in generale la soluzione non è unica). Formalmente possiamo definire il problema nel modo seguente:

> Istanza: un grafo non orientato, connesso $G = \langle V, E\rangle$ e una funzione peso $w : E \to \mathbb{Q}$.
> Soluzione: un insieme di lati $S \subseteq E$ tali che $\langle V, S\rangle$ sia un albero di copertura di peso minimo per G.

L'idea dell'algoritmo che presentiamo è quella di costruire la soluzione S considerando uno dopo l'altro i lati di G in ordine di peso non decrescente: inizialmente si pone $S = \emptyset$ e poi si aggiunge il lato corrente l solo se il nuovo insieme $S \cup \{l\}$ non forma cicli, altrimenti l viene scartato e si considera il lato successivo.

Quindi, durante l'esecuzione dell'algoritmo, S rappresenta un sottoinsieme di lati di G che non forma cicli. Si tratta quindi di una foresta F che definisce automaticamente una partizione P dell'insieme V dei nodi nella quale due vertici si trovano nello stesso sottoinsieme se appartengono al medesimo albero di F. Ora, per verificare se $S \cup \{l\}$ forma un ciclo, basta considerare i due estremi u e v di l e controllare se coincidono i due insiemi della partizione P cui appartengono u e v. In caso affermativo

il lato l, aggiunto alla soluzione parziale S, forma un ciclo e deve quindi essere scartato; altrimenti esso va aggiunto alla soluzione poiché unisce due alberi disgiunti della foresta (ovvero, unifica due insiemi distinti della partizione).

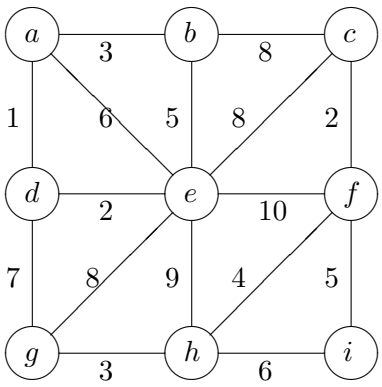
Il procedimento che abbiamo descritto sommariamente manipola tre strutture principali:

1. un insieme S destinato a contenere i lati della soluzione;
2. un sottoinsieme Q dei lati del grafo che mantiene l'insieme degli archi che non sono ancora stati considerati e dal quale dobbiamo via via estrarre il lato di peso minimo;
3. una partizione P dell'insieme dei vertici V che mantiene le famiglie di nodi tra loro connesse mediante lati in S. Inizialmente P coincide con la partizione identità ID e, ogniqualvolta un lato (u, v) viene aggiunto alla soluzione S, si unificano in P i due insiemi cui appartengono i vertici u e v.

Rappresentando con $\{v_1, v_2, \ldots, v_n\}$ l'insieme V dei nodi del grafo, possiamo formalmente descrivere l'algoritmo mediante la seguente procedura:

```
Procedure Kruskal(V, E, w)
begin
      S := ∅
      Q := E
      P := {{v_1}, {v_2}, ..., {v_n}}
      while P contiene piu' di un insieme do
            begin
                  determina il lato (u, v) di peso minimo in Q
                  Cancella (u, v) da Q
                  a := FIND(u)
                  b := FIND(v)
                  if a ≠ b then
                        begin
                              UNION(a, b)
                              Aggiungi (u, v) a S
                        end
            end
end
```

Per illustrare il funzionamento dell'algoritmo, consideriamo il grafo rappresentato nella seguente figura:



Accanto a ogni lato è riportato il peso corrispondente. Nella tabella successiva indichiamo il valore di (u, v) e gli elementi di S e P al termine dell'esecuzione di ciascun ciclo `while`; essi rappresentano il lato di peso minimo estratto da Q, la soluzione parziale dopo il suo eventuale inserimento, e la partizione corrispondente. Nella prima riga inoltre abbiamo posto i valori iniziali.

(u, v)	S	P
-	$\emptyset$	$\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{g\}, \{h\}, \{i\}$
(a, d)	(a, d)	$\{a, d\}, \{b\}, \{c\}, \{e\}, \{f\}, \{g\}, \{h\}, \{i\}$
(c, f)	$(a, d), (c, f)$	$\{a, d\}, \{b\}, \{c, f\}, \{e\}, \{g\}, \{h\}, \{i\}$
(d, e)	$(a, d), (c, f), (d, e)$	$\{a, d, e\}, \{b\}, \{c, f\}, \{g\}, \{h\}, \{i\}$
(a, b)	$(a, d), (c, f), (d, e), (a, b)$	$\{a, b, d, e\}, \{c, f\}, \{g\}, \{h\}, \{i\}$
(g, h)	$(a, d), (c, f), (d, e), (a, b), (g, h)$	$\{a, b, d, e\}, \{c, f\}, \{g, h\}, \{i\}$
(h, f)	$(a, d), (c, f), (d, e), (a, b), (g, h), (h, f)$	$\{a, b, d, e\}, \{c, f, g, h\}, \{i\}$
(b, e)	$(a, d), (c, f), (d, e), (a, b), (g, h), (h, f)$	$\{a, b, d, e\}, \{c, f, g, h\}, \{i\}$
(f, i)	$(a, d), (c, f), (d, e), (a, b), (g, h), (h, f), (f, i)$	$\{a, b, d, e\}, \{c, f, g, h, i\}$
(h, i)	$(a, d), (c, f), (d, e), (a, b), (g, h), (h, f), (f, i)$	$\{a, b, d, e\}, \{c, f, g, h, i\}$
(a, e)	$(a, d), (c, f), (d, e), (a, b), (g, h), (h, f), (f, i)$	$\{a, b, d, e\}, \{c, f, g, h, i\}$
(d, g)	$(a, d), (c, f), (d, e), (a, b), (g, h), (h, f), (f, i), (d, g)$	$\{a, b, d, e, c, f, g, h, i\}$

Osserva che i lati (b, e), (a, e) e (h, i) vengono scartati poiché formano cicli se aggiunti alla soluzione.

La correttezza dell'algoritmo è una conseguenza del teorema di Rado. Infatti è facile verificare che l'algoritmo coincide essenzialmente con la procedura MIOPE-MIN applicata al sistema di indipendenza $\langle E, \mathcal{F}_G \rangle$ definito nell'esempio 12.1. Per la proposizione 12.2 sappiamo che $\langle E, \mathcal{F}_G \rangle$ è un matroide e quindi per il corollario 12.4 la procedura MIOPE-MIN determina la soluzione ottima.

Vogliamo ora valutare i tempi di calcolo richiesti e determinare le strutture dati più idonee per implementare l'algoritmo. Osserviamo innanzitutto che l'insieme S può essere rappresentato banalmente da una lista perchè l'unica operazione da svolgere su questa struttura consiste nell'inserimento di nuovi lati. Sulla partizione P si deve eseguire una sequenza di operazioni UNION e FIND proporzionale al più al numero di lati del grafo di ingresso. Se quindi il grafo G possiede m lati, usando una foresta con bilanciamento, le operazioni UNION e FIND possono essere eseguite in $O(m \log m)$ passi nel caso peggiore. Infine sull'insieme Q dobbiamo eseguire una sequenza di operazioni MIN e DELETE. Possiamo semplicemente ordinare gli elementi di Q in una lista o in un vettore e quindi scorrere i suoi elementi in ordine di peso non decrescente. Tutto questo richiede $\Theta(m \log m)$ passi. Lo stesso ordine di grandezza si ottiene se utilizziamo uno heap rovesciato, nel quale cioè il valore assegnato a ciascun nodo interno è *minore o uguale* a quello dei figli. In questo caso non occorre ordinare inizialmente gli elementi di Q ma basta costruire uno heap, e questo richiede un tempo $O(m)$. Qui la radice corrisponde all'elemento di peso minore e quindi l'operazione MIN può essere eseguita in tempo costante, mentre ogni operazione DELETE richiede un tempo $O(\log m)$ poiché , per ricostruire lo heap, bisogna ricollocare una foglia nella posizione corretta partendo dalla radice e percorrendo un cammino pari al più all'altezza dell'albero.

Poiché la complessità dell'algoritmo può essere ricondotta al costo delle operazioni compiute sulle strutture dati utilizzate, possiamo concludere affermando che la procedura descritta può essere eseguita in $O(m \log m)$ passi nel caso peggiore.

12.5 L'algoritmo di Prim

Non tutti gli algoritmi che adottano una strategia greedy possono essere inquadrati nella teoria generale presentata nelle sezioni precedenti. Uno di questi è l'algoritmo di Prim per il calcolo del minimo albero di copertura di un grafo. Questa procedura si differenzia da quella di Kruskal, descritta nella sezione precedente, poiché l'albero di copertura viene costruito a partire da un nodo sorgente aggiungendo di volta in volta il lato di peso minimo che permette di estendere l'insieme dei nodi raggiunti dalla soluzione. In questo modo la soluzione parziale mantenuta dall'algoritmo non è una foresta come nel caso dell'algoritmo di Kruskal, ma è formata da un albero che ha per radice la sorgente e che inizialmente coincide con quest'ultima. A tale albero vengono via via aggiunti nuovi nodi, scegliendoli tra quelli che si trovano a distanza minima da uno dei suoi vertici; il procedimento prosegue fino a quando tutti i nodi sono stati collegati alla soluzione.

Un'istanza del problema è quindi data da un grafo $G = \langle V, E \rangle$ non orientato, connesso, dotato di una funzione peso $w : E \to \mathbb{Q}$ e da un nodo sorgente $s \in V$. Come al solito rappresentiamo il grafo associando a ogni vertice $v \in V$ la lista $L(v)$ dei nodi adiacenti. La soluzione viene calcolata fornendo l'insieme T dei lati dell'albero di copertura ottenuto.

Sostanzialmente l'algoritmo mantiene una partizione dei nodi del grafo in due insiemi che chiameremo S e R. L'insieme S è formato dai vertici già raggiunti dalla soluzione, mentre R è costituito da tutti gli altri. Per ogni nodo v in R la procedura conserva il peso del lato di costo minimo che congiunge v con un nodo in S e sceglie tra tutti questi lati quello di peso minore, aggiungendolo alla soluzione ed estendendo così l'insieme dei vertici raggiunti. Per descrivere questo procedimento definiamo, per ogni nodo $v \in R$, la distanza di v da S mediante la seguente espressione:

$$dist(v) = \begin{cases} \min\{w(\{v,z\}) \mid z \in S\} & \text{se } \{z \in S \mid \{v,z\} \in E\} \neq \emptyset \\ \infty & \text{altrimenti} \end{cases}$$

Inoltre denotiamo con $vicino(v)$ il nodo $z \in S$ tale che $w(\{v,z\}) = dist(v)$, con la convenzione di lasciare $vicino(v)$ indefinito se $dist(v) = \infty$. I valori $dist(v)$ e $vicino(v)$ possono essere rappresentati efficientemente mediante due vettori indiciati dagli elementi di V. Infine, denoteremo con D l'insieme dei nodi in R che hanno una distanza finita da S:

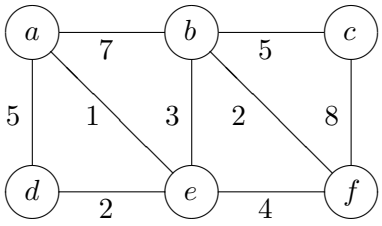
$$D = \{v \in R \mid dist(v) \neq \infty\}$$

Tale insieme viene costantemente aggiornato dall'algoritmo e da esso (insieme ai valori $dist$ e $vicino$) la procedura estrae di volta in volta il nodo che si trova a distanza minima dalla soluzione.

L'algoritmo può ora essere descritto dalla seguente procedura che utilizza effettivamente solo l'insieme D, i valori $dist$ e $vicino$ e l'insieme T che mantiene la soluzione costruita.

```
Procedure Prim(V, E, w, s)
begin
      T := ∅
      for v ∈ V do { dist(v) := ∞
                   { vicino(v) := ⊥
      D := {s}
      dist(s) := 0
      while D ≠ ∅ do
            begin
(1)               determina l'elemento v ∈ D con dist(v) minima
(2)               cancella v da D
                  aggiungi il lato {v, vicino(v)} a T
                  for u ∈ L(v) do
                      if dist(u) = ∞
                                          { aggiungi u a D
(3)                                then   { dist(u) := w({v, u})
                                          { vicino(u) := v
                      else if u ∈ D ∧ w({v, u}) < dist(u)
                                          { dist(u) := w(v, u)
(4)                                 then   { vicino(u) := v
            end
      output T − {{s, ⊥}}
end
```

Per illustrare il funzionamento dell'algoritmo, consideriamo il grafo rappresentato nella seguente figura:



Accanto a ogni lato è riportato il peso corrispondente e si suppone che il nodo sorgente sia d. Nella tabella successiva mostriamo il funzionamento dell'algoritmo su tale istanza, riportando i valori degli insiemi R, D, T, del vettore $dist$ e del nodo v dopo l'esecuzione di ogni ciclo `while`.

v	R	$dist$	D	$T - \{\{s, \perp\}\}$
d	$\{a, b, c, e, f\}$	$(5, \infty, \infty, -, 2, \infty)$	$\{a, e\}$	$\emptyset$
e	$\{a, b, c, f\}$	$(1, 3, \infty, -, -, 4)$	$\{a, b, f\}$	$\{e, d\}$
a	$\{b, c, f\}$	$(-, 3, \infty, -, -, 4)$	$\{b, f\}$	$\{e, d\}, \{e, a\}$
b	$\{c, f\}$	$(-, -, 5, -, -, 2)$	$\{c, f\}$	$\{e, d\}, \{e, a\}, \{e, b\}$
f	$\{c\}$	$(-, -, 5, -, -, -)$	$\{c\}$	$\{e, d\}, \{e, a\}, \{e, b\}, \{b, f\}$
c	$\emptyset$	$(-, -, -, -, -, -)$	$\emptyset$	$\{e, d\}, \{e, a\}, \{e, b\}, \{b, f\}, \{b, c\}$

Scegliamo ora le strutture dati più idonee per implementare l'algoritmo. Innazitutto i valori $dist$ e $vicino$ possono essere rappresentati mediante vettori che hanno per indice i nodi $v \in V$.

La soluzione T può essere semplicemente rappresentata da una lista poiché l'unica operazione eseguita su tale insieme consiste nell'aggiungere un nuovo elemento (operazione che possiamo così eseguire in tempo costante).

Sull'insieme D dobbiamo eseguire 4 operazioni: determinare il nodo v a distanza minima (linea (1) nella procedura), cancellarlo (linea (2)), introdurre un nuovo elemento (linea (3)) e aggiornare la distanza di un nodo già presente in D (linea (4)). Descriviamo due diverse strutture per implementare D. Nel primo caso consideriamo D come una coda di priorità e utilizziamo uno *heap* rovesciato nel quale il valore di ogni nodo v è dato dalla sua distanza $dist(v)$. In questo modo il nodo a distanza minima si trova nella radice dello heap e l'operazione di cancellazione del minimo e di inserimento di un nuovo elemento possono essere eseguite in tempo logaritmico. Osserva che l'aggiornamento della distanza del nodo u, compiuto nella istruzione (4) della procedura, deve essere accompagnato da una ricollocazione di u all'interno dello heap; poiché la sua distanza diminuisce questo corrisponde a spostare u verso la radice. Chiaramente, anche quest'ultima operazione richiede un tempo logaritmico rispetto al numero di elementi contenuti nello heap. Utilizzando tali strutture e le relative procedure, l'algoritmo di Prim, eseguito su un grafo di n nodi e m lati, richiede un tempo di calcolo $O(m \log n)$ (assumendo il criterio di costo uniforme).

Diversamente, possiamo implementare l'insieme D mediante un vettore $\hat{D}$ di n elementi che ha per indici i nodi del grafo di ingresso $G = (V, E)$. Per ogni $v \in V$, poniamo

$$\hat{D}[v] = \begin{cases} \infty & \text{se } v \in R - D \\ dist(v) & \text{se } v \in D \\ 0 & \text{se } v \in S \end{cases}$$

In questo modo il calcolo del nodo a distanza minima in D (linea (1)) richiede un tempo $\Theta(n)$, poiché occorre scorrere l'intero vettore per determinare il minimo; tuttavia le altre operazioni richiedono un tempo costante. È facile verificare che, usando questa struttura, l'algoritmo richiede un tempo dell'ordine di $\Theta(n^2)$. Di conseguenza, se il numero dei lati m è dell'ordine di grandezza di n^2 (cioè il grafo di ingresso contiene "molti" lati) allora quest'ultima implementazione è preferibile alla precedente. Se invece $m << n^2$ (cioè il grafo di ingresso contiene "pochi" lati) l'uso di uno heap per implementare l'insieme D risulta asintoticamente più efficiente.

Concludiamo studiando la correttezza dell'algoritmo appena descritto. Questa è basata sulla seguente proprietà.

Proposizione 12.5 *Sia $G = \langle V, E \rangle$ un grafo non orientato, connesso, dotato di una funzione peso $w : E \to \mathbb{Q}$. Sia inoltre T un albero di copertura minimo per G, sia U un sottoalbero di T e sia S l'insieme dei nodi in U. Consideriamo un lato di costo minimo $\{a, b\}$ che "esce" da S, cioè tale che $a \in S$ e $b \notin S$. Allora esiste un albero di copertura minimo per G che contiene U come sottoalbero e che contiene anche il lato $\{a, b\}$.*

Dimostrazione. Se il lato $\{a, b\}$ appartiene a T allora T è l'albero di copertura cercato. Altrimenti deve esistere in T un cammino semplice che congiunge a e b; tale cammino si svolge inizialmente tra nodi dell'insieme S, quindi "esce" da S raggiungendo b. Deve allora esistere in tale cammino un lato $\{a', b'\}$ tale che $a' \in S$ e $b' \notin S$ (eventualmente si può verificare $a = a'$ oppure $b = b'$, ma non entrambi). Allora, sostituendo in T il lato $\{a', b'\}$ con il lato $\{a, b\}$ non formiamo cicli e otteniamo così un nuovo

albero di copertura T'. Chiaramente T' contiene U come sottoalbero e inoltre il peso dei suoi lati è dato da:

$$w(T') = w(T) - w(\{a', b'\}) + w(\{a, b\})$$

Poiché $\{a, b\}$ è di peso minimo tra tutti i lati che escono da S, abbiamo $w(\{a, b\}) \leq w(\{a', b'\})$ e quindi dall'equazione precedente otteniamo $w(T') \leq w(T)$. Di conseguenza, essendo T un albero di copertura minimo, anche T' risulta un albero di copertura minimo. ∎

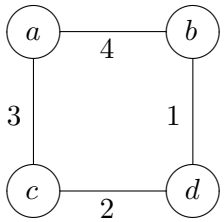
Nota che la proposizione precedente è valida anche quando l'insieme S è ridotto ad un solo nodo: il nodo sorgente. Così l'applicazione della proprietà appena provata consente di dimostrare per induzione che la soluzione fornita dall'algoritmo rappresenta un albero di copertura minimo del grafo di ingresso.

Esercizio

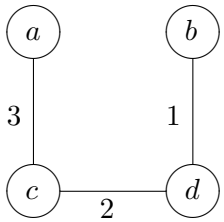
1) Applicando l'osservazione precedente, provare la correttezza dell'algoritmo di Prim.

12.6 L'algoritmo di Dijkstra

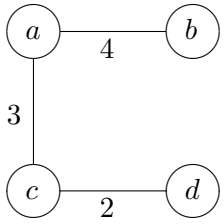
Un procedimento di calcolo del tutto simile a quello presentato nella sezione precedente può essere descritto per risolvere un problema di cammini minimi su un grafo. Abbiamo già affrontato un problema simile nella sezione 11.5 dove è stato presentato un algoritmo per determinare i cammini di peso minimo tra tutte le coppie di nodi in un grafo pesato. In questo caso si vuole affrontare un problema più semplice, quello di determinare, in un grafo pesato, i cammini di costo minimo che congiungono un nodo sorgente fissato con gli altri vertici. Osserviamo che tale problema è sostanzialmente diverso da quello di determinare l'albero di copertura di costo minimo in un grafo che è stato affrontato nelle sezione precedenti. Per esempio, se consideriamo il grafo rappresentato dalla seguente figura



l'albero di copertura di peso minimo è dato dal grafo seguente



mentre l'albero dei cammini di peso minimo che connettono il nodo sorgente a con gli altri nodi è definito dalla seguente figura:



L'algoritmo che presentiamo è noto come Algoritmo di Dijkstra ed è anche questo basato su una tecnica greedy. In questa sezione ne descriviamo formalmente il procedimento ma omettiamo la relativa analisi di complessità e la dimostrazione di correttezza che possono essere sviluppate facilmente seguendo la linea tratteggiata nella sezione precedente.

L'istanza del problema è costituita da un grafo orientato $G = \langle V, E\rangle$, un nodo sorgente $s \in V$ e una funzione peso $w : E \to \mathbb{Q}$ a valori non negativi (ovvero $w(\ell) \geq 0$ per ogni $\ell \in E$). Anche in questo caso denotiamo con $L(v)$ la lista dei nodi z tali che $(v, z) \in E$. Per ogni $v \in V$ si vuole determinare il cammino di peso minimo in G che congiunge s con v e il relativo peso. Il calcolo può essere eseguito determinando i valori $pesocammino(v)$ e $predecessore(v)$ così definiti:

$$pesocammino(v) = \begin{cases} c & \text{se in } G \text{ esiste un cammino da } s \text{ a } v \text{ e } c \text{ è il peso} \\ & \text{del cammino di costo minimo da } s \text{ a } v, \\ \infty & \text{se in } G \text{ non esiste un cammino da } s \text{ a } v. \end{cases}$$

$$predecessore(v) = \begin{cases} u & \text{se in } G \text{ esiste un cammino da } s \text{ a } v \text{ e } u \text{ è il nodo} \\ & \text{che precede } v \text{ nel cammino di costo minimo da } s \text{ a } v, \\ \perp & \text{se in } G \text{ non esiste un cammino da } s \text{ a } v \text{ oppure } v = s. \end{cases}$$

L'algoritmo mantiene una partizione dei nodi del grafo in tre sottoinsiemi: l'insieme S dei vertici per i quali la soluzione è già stata calcolata, l'insieme D dei nodi v, non inclusi in S, per i quali esiste un lato da un vertice in S a v e, infine, l'insieme formato da tutti gli altri nodi. Inoltre, per ogni $v \in D$, la procedura mantiene aggiornati i due valori $C(v)$ e $P(v)$: il primo rappresenta il peso del cammino di costo minimo da s a v che passa solo per nodi in S; il secondo costituisce l'ultimo nodo prima di v in tale cammino. Tali valori possono essere conservati in opportuni vettori. Inizialmente, S è vuoto e D contiene solo la sorgente s mentre, per ogni vertice v, $P(v) = \perp$ e $C(v) = \infty$ tranne $C(s)$ che assume valore 0; al termine del calcolo $C(v)$ e $P(v)$ coincideranno con i valori $pesocammino(v)$ e $predecessore(v)$ da calcolare e tali valori risulteranno definiti solo per i vertici raggiungibili da s.

L'algoritmo sceglie di volta in volta il nodo $v \in D$ per il quale $C(v)$ è minimo, lo toglie dall'insieme D, aggiornando i valori $C(u)$ e $P(u)$ per ogni $u \in L(v)$ (eventualmente inserendo u in D). L'algoritmo non ha in realtà bisogno di mantenere l'insieme S: è sufficiente conservare D.

L'algoritmo termina quando $D = \emptyset$; a questo punto, se $C(v) = \infty$ per qualche nodo v, allora v non è raggiungibile da s mediante un cammino del grafo (risulterà anche $P(v) = \perp$).

L'algoritmo è descritto dalla seguente procedura:

```
Procedure Dijkstra(V, E, w, s)
begin
      D := {s}
      for v ∈ V do { C(v) := ∞
                   { P(v) := ⊥
      C(s) := 0
      while D ≠ ∅ do
            begin
                  determina il nodo v in D con C(v) minima
                  cancella v da D
                  for u ∈ L(v) do
                              if C(v) + w(v,u) < C(u) then
                                begin
                                      if C(u) = ∞ then aggiungi u a D
                                      C(u) := C(v) + w(v,u)
                                      P(u) := v
                                end
            end
      return C, P
end
```

Osserva che i valori $C(v)$ e $P(v)$ svolgono nella procedura appena descritta lo stesso ruolo di $dist(v)$ e $vicino(v)$ nell'algoritmo di Prim. La dimostrazione della correttezza dell'algoritmo viene qui omessa per brevità. Essa è essenzialmente basata sull'ipotesi che i pesi dei lati siano non negativi e sul fatto che, durante l'esecuzione dell'algoritmo, man mano che i nodi scelti in D vengono raggiunti dalla soluzione, i valori corrispondenti $C(v)$ non decrescono.

Inoltre D può essere implementato come nell'algoritmo di Prim, usando uno heap rovesciato oppure un semplice vettore. Come nella sezione precedente si può dimostrare che il tempo di calcolo richiesto dall'algoritmo di Dijkstra, su un input di n nodi e m lati, è $O(m \log n)$ se usiamo uno heap rovesciato per implementare D mentre, se utilizziamo un vettore, lo stesso tempo diviene $O(n^2)$. Anche in questo caso è dunque asintoticamente più vantaggioso utilizzare uno heap rovesciato per grafi con "pochi" lati, mentre conviene usare un vettore nel caso di grafi con "molti" lati.

Esercizi

1) Dimostrare mediante un esempio che l'algoritmo di Dijkstra non fornisce la soluzione esatta al problema quando il peso dei lati è negativo.

2) Dimostrare la correttezza dell'algoritmo di Dijkstra quando il peso dei lati è maggiore o uguale a zero.

12.7 Codici di Huffman

Un problema del tutto naturale in ambito informatico è quello di codificare un file di caratteri, estratti da un alfabeto dato, mediante una stringa binaria. Si tratta di definire un codice binario, ovvero una funzione che associa ad ogni carattere una sequenza di bit in modo tale che ogni stringa binaria corrisponda al più a una sola parola dell'alfabeto dato. Così la codifica del file originario si ottiene semplicemente concatenando le stringhe binarie associate ai vari caratteri che compongono il file.

Il primo obiettivo è quello di definire un codice che permetta di calcolare rapidamente la codifica di ogni carattere e, viceversa, di decodificare in maniera efficiente una stringa binaria determinando la sequenza di simboli corrispondente. In particolare si vuole compiere queste operazioni in tempo lineare, possibilmente mediante la semplice lettura della sequenza in ingresso. Inoltre, per ovvie ragioni di spazio, si vuole definire un codice ottimale che consenta cioè di rappresentare il file mediante la stringa binaria più corta possibile.

È evidente infatti che se le frequenze dei vari caratteri all'interno del file originario sono molto diverse tra loro, sarà conveniente rappresentare i simboli più frequenti mediante stringhe binarie relativamente corte e utilizzare quelle più lunghe per rappresentare i simboli più rari. In questo modo si può risparmiare una notevole quantità di spazio (in alcuni casi anche superiore al 50%).

Esempio 12.4
Supponiamo per esempio di voler codificare in binario una sequenza di 100 caratteri definita sull'alfabeto $\{a, b, c, d, e, f, g\}$, nella quale i vari simboli compaiono con la seguente frequenza:

$$\begin{gathered}35 \text{ occorrenze di } a\\ 20 \text{ occorrenze di } b\\ 5 \text{ occorrenze di } c\\ 30 \text{ occorrenze di } d\\ 5 \text{ occorrenze di } e\\ 2 \text{ occorrenze di } f\\ 3 \text{ occorrenze di } g.\end{gathered}$$

Definiamo ora il codice λ nel modo seguente:

$$\lambda(a) = 000, \lambda(b) = 001, \lambda(c) = 010, \lambda(d) = 011, \lambda(e) = 100, \lambda(f) = 101, \lambda(g) = 11.$$

Utilizzando tale codice, la lunghezza della stringa binaria che rappresenta l'intera sequenza è allora data da $97 \cdot 3 + 2 \cdot 3 = 297$.

Osserva che nella codifica appena definita quasi tutti i caratteri sono rappresentati con stringhe di lunghezza 3, pur avendo questi frequenze notevolmente diverse all'interno della sequenza originaria. Se invece definiamo una codifica che tenga conto delle varie frequenze possiamo migliorare notevolmente la lunghezza della stringa ottenuta. Definiamo per esempio il codice μ tale che

$$\mu(a) = 00, \mu(b) = 010, \mu(c) = 011, \mu(d) = 10, \mu(e) = 110, \mu(f) = 1110, \mu(g) = 1111.$$

È immediato verificare che, in questo caso, la lunghezza della stringa binaria ottenuta è data da $65 \cdot 2 + 30 \cdot 3 + 5 \cdot 4 = 230$, risparmiando quindi, rispetto alla precedente, più del 20% di bit.

∎

I metodi che presentiamo in questa sezione consentono di determinare il codice binario ottimale di una stringa data conoscendo, per ogni carattere, la frequenza corrispondente all'interno della sequenza. Si tratta dei noti codici di Huffman che hanno una notevole importanza in questo ambito e sono ampiamente utilizzati nelle applicazioni. L'algoritmo di costruzione del codice di Huffman di una sequenza assegnata è un tipico esempio di algoritmo greedy.

12.7.1 Codici binari

Per formalizzare il problema ricordiamo che un alfabeto è un insieme finito di simboli (che chiameremo anche lettere) e una parola definita su un alfabeto A è una concatenazione di simboli estratti da A [1]. Denotiamo inoltre con A^+ l'insieme delle parole definite su A. In particolare, $\{0, 1\}^+$ denota l'insieme delle stringhe binarie. Denoteremo con $|x|$ la lunghezza di un parola x.

[1] Non consideriamo qui la parola vuota ϵ, quella nella quale non compare alcun simbolo.

Dato un alfabeto A, una funzione $\gamma : A \to \{0,1\}^+$ è un *codice binario* se per ogni $y \in \{0,1\}^+$ esiste al più una sola sequenza di lettere $x_1, x_2, \ldots, x_n \in A$ tale che $y = \gamma(x_1)\gamma(x_2)\cdots\gamma(x_n)$. In altre parole, γ è un codice binario se ogni stringa in $\{0,1\}^+$ può essere decomposta al più in un solo modo come concatenazione di parole appartenenti a $\gamma(A)$. Nel seguito con il termine *codice* intenderemo sempre un codice binario.

Chiaramente, ogni funzione $\gamma : A \to \{0,1\}^+$ può essere estesa all'insieme A^+ definendo, per ogni $x \in A^+$, l'immagine $\gamma(x) = \gamma(x_1)\gamma(x_2)\cdots\gamma(x_n)$, dove $x = x_1x_2\cdots x_n$ e $x_i \in A$ per ogni i; possiamo quindi affermare che la funzione γ è un codice se la sua estensione all'insieme A^+ è una funzione iniettiva.

Esempio 12.5
Sia $A = \{a, b, c\}$ e sia $\beta : A \to \{0,1\}^+$ la funzione definita da $\beta(a) = 0$, $\beta(b) = 01$, $\beta(c) = 10$. È facile verificare che β non forma un codice perché , ad esempio, la stringa 010 coincide con $\beta(ba) = \beta(ac)$.

Viceversa, è facile verificare che la funzione $\delta : A \to \{0,1\}^+$, definita dalle uguaglianze $\delta(a) = 00$, $\delta(b) = 01$, $\delta(c) = 10$, forma un codice binario. ∎

Tra le proprietà solitamente richieste vi è quella di poter decodificare una stringa binaria "on line", ovvero determinando la decodifica di ciascun prefisso senza bisogno di considerare l'intera sequenza [2]. Questa proprietà può essere formalizzata dalla seguente definizione: un codice $\gamma : A \to \{0,1\}^+$ si dice *prefisso* se non esiste una coppia di lettere distinte $a, b \in A$ per le quali $\gamma(a)$ sia un prefisso di $\gamma(b)$. Per esempio la funzione δ definita nell'esempio precedente è un codice prefisso. Invece, il codice $\eta : \{a, b\} \to \{0,1\}^+$ definito da $\eta(a) = 0$, $\eta(b) = 01$, non è un codice prefisso perchè $\eta(a)$ è un prefisso di $\eta(b)$.

Inoltre un codice $\gamma : A \to \{0,1\}^+$ si dice a *lunghezza fissa* se esiste $k \in \mathbb{N}$ tale che $|\gamma(a)| = k$ per ogni $a \in A$. È facile verificare che ogni codice a lunghezza fissa è anche un codice prefisso.

Infine, data una parola $x \in A^+$, chiameremo *costo* di un codice γ su A la lunghezza di $\gamma(x)$ e lo denoteremo mediante $C(\gamma)$. Se rappresentiamo con $|x|_a$ il numero di occorrenze della lettera a in x, allora abbiamo

$$C(\gamma) = \sum_{a \in A} |x|_a \cdot |\gamma(a)|$$

Un codice γ si dirà *ottimale* rispetto una parola x se $C(x)$ è minimo tra tutti i costi di codici prefissi definiti su A.

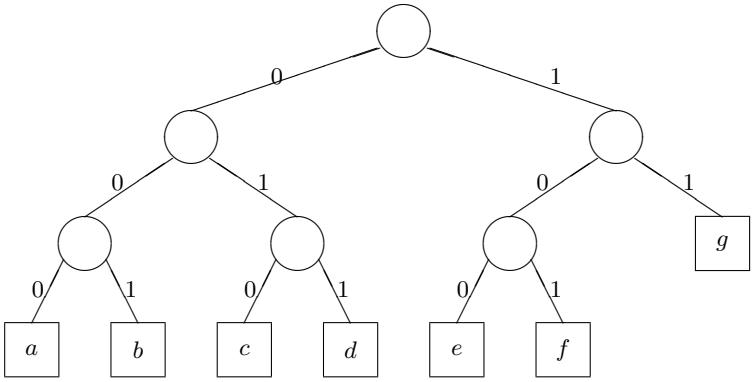
I codici prefissi ammettono una efficace rappresentazione mediante un albero binario che consente in molti casi di definire facilmente algoritmi di costruzione e di manipolazione di queste funzioni. L'albero binario che rappresenta un codice prefisso $\gamma : A \to \{0,1\}^+$ è definito nel modo seguente:

1. le foglie dell'albero coincidono con le lettere dell'alfabeto;
2. ogni lato che congiunge un nodo interno con il suo figlio sinistro è etichettato dal bit 0;
3. ogni lato che congiunge un nodo interno con il suo figlio destro è etichettato dal bit 1;
4. per ogni $a \in A$, la stringa $\gamma(a)$ coincide con l'etichetta del cammino che congiunge la radice dell'albero con la foglia a.

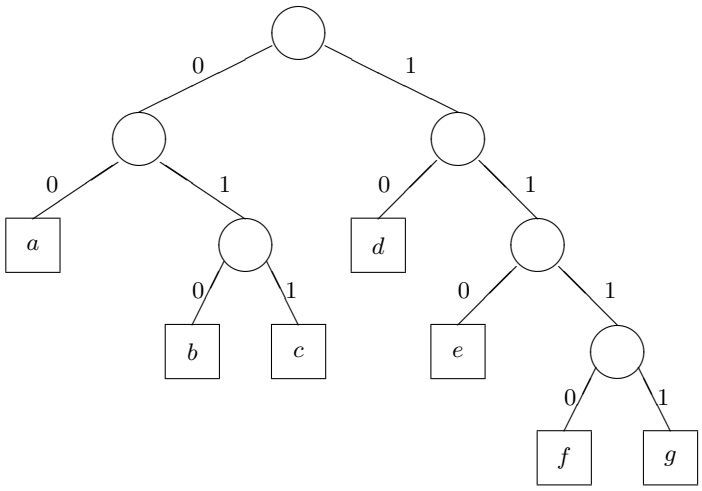
Esempio 12.6

La funzione λ sull'alfabeto $\{a, b, c, d, e, f, g\}$, a valori in $\{0,1\}^+$, definita nell'Esempio 12.4, è un codice prefisso rappresentato dall'albero seguente:

[2] Ricordiamo che il prefisso di una parola x è una parola y tale che $x = yz$ per qualche stringa z (eventualmente vuota).



Analogamente, la funzione μ, definita nel medesimo Esempio 12.4, è un codice prefisso rappresentato dal seguente albero binario:



12.7.2 Descrizione dell'algoritmo

La definizione del codice di Huffman di una data stringa si può comprendere immediatamente descrivendo l'algoritmo che costruisce il corrispondente albero binario. Tale procedura è basata su una strategia greedy e il suo input è dato da un alfabeto A e da una funzione fr (che chiameremo *frequenza*) che associa da ogni lettera $a \in A$ il numero di occorrenze di a nella stringa da codificare.

Si comincia a costruire l'albero partendo dalle foglie, che sono ovviamente rappresentate dalle lettere dell'alfabeto A. Quindi si scelgono le due foglie con frequenza minore e si definisce un nodo interno rendendo quest'ultimo padre delle due precedenti. Si sostituiscono le due foglie estratte con il loro padre, attribuendo a quest'ultimo un valore di frequenza pari alla somma delle frequenze dei due figli; quindi si ripete il procedimento, scegliendo nuovamente i due vertici di minor frequenza, rendendoli entrambi figli di un nuovo nodo interno e sostituendo questi con il padre nell'insieme delle possibili scelte successive. Questo procedimento va ripetuto fino a quando l'insieme delle possibile scelte si riduce a un solo nodo che risulta per forza la radice dell'albero ottenuto.

Per definire l'algoritmo abbiamo bisogno di due strutture dati. La prima sarà data da un albero binario T che definirà automaticamente il codice prodotto dall'algoritmo. La seconda sarà invece una

coda di priorità Q, che mantiene l'insieme dei nodi tra i quali occorre scegliere la coppia di elementi di frequenza minore, e sulla quale operare le sostituzioni opportune. Su Q dovremo quindi eseguire le operazioni Min, Insert e Delete e, inizialmente, tale struttura sarà rappresentata dall'insieme delle foglie dotate della corrispondente frequenza.

Possiamo formalmente definire l'algoritmo mediante la seguente procedura:

```
Procedure Huffman(A, fr)
begin
      n := ♯A
      crea un albero binario T, con insieme delle foglie A, prevedendo n − 1 nodi interni
      crea una coda di priorita' Q inizialmente vuota
      for a ∈ A do
          inserisci a in Q con peso fr(a)
      for i = 1, 2, ..., n − 1 do
          begin
                determina i due elementi u, v in Q di peso minimo
                cancella u e v da Q
                crea un nodo interno z dell'albero T
                rendi u figlio sinistro di z
                rendi v figlio destro di z
                inserisci z in Q con peso fr(z) = fr(u) + fr(v)
          end
      return T
end
```

Per implementare l'algoritmo si possono usare una coppia di vettori sin e des per rappresentare l'albero T e un albero $2-3$, o una qualunque struttura bilanciata, per l'insieme Q. Osserviamo, in particolare, che una struttura dati molto comoda per Q è data da uno heap rovesciato nel quale l'operazione Min è immediata, essendo il minimo posizionato sulla radice dello heap. Con queste strutture le operazioni Min, Delete e Insert richiedono comunque un tempo $O(\log n)$, assumendo n la cardinalità dell'alfabeto, e quindi il tempo complessivo richiesto dall'algoritmo risulta $O(n \log n)$.

12.7.3 Correttezza

Vogliamo ora provare che il codice di Huffman di una qualsiasi stringa è ottimale tra tutti i codici prefissi.

Proposizione 12.6 *Dato un alfabeto A e una frequenza fr definita su A, il codice di Huffman, ottenuto mediante la procedura data sopra è un codice prefisso ottimale.*

Dimostrazione. Ragioniamo per induzione sulla cardinalità n di A. Se $n = 2$ la proprietà è ovvia. Supponiamo $n > 2$ e assumiamo la proprietà vera per ogni alfabeto di dimensione $k \leq n-1$. Consideriamo due lettere u, v di frequenza minima in A e definiamo l'alfabeto B ottenuto da A sostituendo u e v con una nuova lettera x di frequenza $fr(x) = fr(u) + fr(v)$. Sia β il codice di Huffman di B. Per ipotesi d'induzione β è un codice prefisso ottimale per B. Possiamo allora definire il codice α per l'alfabeto A nel modo seguente:

$$\alpha(a) = \begin{cases} \beta(a) & \text{se } a \neq u \text{ e } a \neq v \\ \beta(x)0 & \text{se } a = v \\ \beta(x)1 & \text{se } a = u \end{cases}$$

Chiaramente, α è un codice prefisso per A e coincide con il codice di Huffman perchè di fatto si può ottenere applicando la stessa procedura.

Vogliamo ora provare che α è ottimale. Osserviamo anzitutto che

$$C(\alpha) = C(\beta) - |\beta(x)|fr(x) + |\alpha(u)|fr(u) + |\alpha(v)|fr(v) = C(\beta) + fr(u) + fr(v). \qquad (12.1)$$

Consideriamo ora un qualsiasi codice prefisso γ sull'alfabeto A. Vogliamo provare che $C(\alpha) \leq C(\gamma)$. Siano s e t le due lettere in A che hanno lunghezza di codice maggiore (cioè tali che $|\gamma(s)|$ e $|\gamma(t)|$ siano massime in $\{|\gamma(a)| \mid a \in A\}$). Senza perdita di generalità, possiamo assumere che s e t siano fratelli nell'albero che rappresenta γ, altrimenti sarebbe possibile costruire un nuovo codice, nel quale s e t sono fratelli, che possiede un costo minore o uguale a quello di γ e quindi proseguire la dimostrazione per questo.

Definiamo un nuovo codice γ' che si ottiene da γ scambiando i le stringhe binarie associate a s e u e quelle associate a t e v. Si può facilmente provare che il costo del nuovo codice non è maggiore di quello del precedente:

$$C(\gamma') = C(\gamma) + (|\gamma(u)| - |\gamma(s)|)(fr(s) - fr(u)) + (|\gamma(v)| - |\gamma(t)|)(fr(t) - fr(v))$$

Per le ipotesi fatte sulle lettere u, v, s e t, i due prodotti nel secondo termine dell'equazione precedente forniscono valori minori o uguali a zero; questo prova che $C(\gamma') \leq C(\gamma)$. Ora poiché u e v sono fratelli nell'albero che rappresenta γ', possiamo denotare con ℓ il massimo prefisso comune di $\gamma'(u)$ e $\gamma'(v)$, definendo così un nuovo codice δ per l'alfabeto B:

$$\delta(b) = \begin{cases} \gamma'(b) & \text{se } b \neq x \\ \ell & \text{se } b = x \end{cases}$$

Osserva che $C(\gamma') = C(\delta) + fr(u) + fr(v)$. Inoltre, poiché β è un codice ottimale su B, abbiamo che $C(\beta) \leq C(\delta)$. Quindi, applicando la disuguaglianza (12.1), otteniamo

$$C(\alpha) = C(\beta) + fr(u) + fr(v) \leq C(\delta) + fr(u) + fr(v) = C(\gamma') \leq C(\gamma)$$

∎

Esercizi

1) Costruire il codice di Huffman delle seguenti frasi interpretando il blank tra una parola e l'altro come un simbolo dell'alfabeto:

il mago merlino e la spada nella roccia
re artù e i cavalieri della tavola rotonda

2) Considera il problema Knapsack 0-1 (Zaino) definito nel modo seguente:

Istanza: un intero positivo H, n valori $v_1, v_2, \ldots, v_n \in \mathbb{N}$ e n dimensioni $d_1, d_2, \ldots, d_n \in \mathbb{N}$.
Soluzione : un insieme $S \subseteq \{1, 2, \ldots, n\}$ di dimesione al più H e di valore massimo, ovvero tale che

$$\sum_{i \in S} d_i \leq H,$$

$$\sum_{i \in S} v_i = \max\{\sum_{i \in A} v_i \mid A \subset \{1, 2, \ldots, n\}, \sum_{i \in S} d_i \leq H\}.$$

a) Definire un algoritmo greedy per il problema dato (per esempio basato sul rapporto tra il valore e la dimensione di ogni elemento).

b) Dimostrare mediante un esempio che l'algoritmo non fornisce sempre la soluzione ottima.

3) Considera la seguente variante al problema precedente (Knapsack frazionato):

Istanza: un intero positivo H, n valori $v_1, v_2, \ldots, v_n \in \mathbb{N}$ e n dimensioni $d_1, d_2, \ldots, d_n \in \mathbb{N}$.
Soluzione : n numeri razionali $f_1, f_2, \ldots, f_n$ tali che $0 \leq f_i \leq 1$ per ogni i e inoltre

$$\sum_{i=1}^{n} f_i \cdot d_i \leq H,$$

$$\sum_{i=1}^{n} f_i \cdot v_i = \max\{\sum_{i=1}^{n} g_i \cdot v_i \mid g_1, g_2, \ldots, g_n \in \mathbb{Q}, \forall i\ 0 \leq g_i \leq 1, \sum_{i=1}^{n} g_i \cdot d_i \leq H\}.$$

a) Definire un algoritmo greedy per il problema assegnato.

b) Dimostrare che tale algoritmo determina sempre la soluzione ottima.

Capitolo 13

I problemi NP-completi

Gli argomenti principali trattati nei capitoli precedenti riguardano la sintesi e l'analisi degli algoritmi. Lo scopo era quello di illustrare le tecniche principali di progettazione di un algoritmo e i metodi che permettono di valutarne le prestazioni con particolare riferimento alle risorse di calcolo utilizzate (tempo e spazio).

Vogliamo ora spostare l'attenzione sui problemi, studiando la possibilità di classificare questi ultimi in base alla quantità di risorse necessarie per ottenere la soluzione. Si è riscontrato infatti, che per certi gruppi di problemi, le difficoltà incontrate per trovare un algoritmo efficiente sono sostanzialmente le stesse. Tenendo conto dei risultati ottenuti nella letteratura, possiamo grossolanamente raggruppare i problemi in tre categorie:

1. I problemi che ammettono algoritmi di soluzione efficienti;

2. I problemi che per loro natura non possono essere risolti mediante algoritmi efficienti e che quindi sono intrattabili;

3. I problemi per i quali algoritmi efficienti non sono stati trovati ma per i quali nessuno ha finora provato che tali algoritmi non esistano.

Molti problemi di notevole interesse appartengono al terzo gruppo e presentano tra loro caratteristiche così simili dal punto di vista algoritmico che risulta naturale introdurre metodi e tecniche che consentano di studiarne le proprietà complessivamente. Questo ha portato a definire e studiare le cosiddette "classi di complessità ", cioè classi di problemi risolubili utilizzando una certa quantità di risorse (per esempio di tempo oppure di spazio). Questi strumenti consentono anche di confrontare la difficoltà intrinseca dei vari problemi, verificando per esempio se un problema dato è più o meno facile di un altro, o se è possibile trasformare un algoritmo per il primo in uno per il secondo che richieda all'incirca la stessa quantità di risorse.

Tra le classi di complessità definite in base al tempo di calcolo quelle più note sono le classi P e NP. Queste contengono gran parte dei problemi considerati in letteratura e le problematiche legate allo studio delle loro proprietà sono in realtà comuni all'analisi di molte altre classi di complessità. L'interesse per lo studio di questi argomenti è inoltre legato alla presenza di molti problemi aperti alcuni dei quali sono fra i più importanti dell'informatica teorica.

13.1 Problemi intrattabili

Descriviamo ora con maggior precisione le tre classi di problemi sopra menzionate [1].

La principale classe di problemi considerata in letteratura è quella dei problemi risolubili in tempo polinomiale. Vengono così chiamati quei problemi che ammettono un algoritmo di soluzione il cui tempo di calcolo, nel caso peggiore, è limitato da un polinomio nelle dimensioni dell'input. Questa classe può essere definita mediante diversi modelli di calcolo ed è sostanzialmente robusta, cioè non dipende dal particolare modello considerato. Intuitivamente essa rappresenta la classe dei problemi "trattabili" cioè quelli che ammettono un algoritmo di soluzione efficiente. Quasi tutti i problemi considerati nei capitoli precedenti appartengono a questa classe. Chiaramente è poco realistico considerare trattabile un problema risolubile solo da algoritmi che richiedono un tempo dell'ordine di n^k con k elevato. Tuttavia, quasi tutti i problemi di interesse, risolubili in tempo polinomiale, ammettono un algoritmo di soluzione che richiede $O(n^3)$ passi su un input di dimensione n. Inoltre la maggior parte di questi problemi hanno algoritmi che ammettono tempi poco più che lineari e questo spesso significa, per esempio, poter risolvere il problema in pochi secondi anche su istanze di dimensione relativamente elevate ($n = 10^6$). Ricordiamo che la classe dei problemi di decisione risolubili in tempo polinomiale è nota in letteratura come classe P.

La naturale controparte della classe appena descritta è quella dei problemi che *non* possono essere risolti in un tempo polinomiale. Per questi problemi si può provare che ogni algoritmo risolutivo richiede, nel caso peggiore, un tempo di calcolo esponenziale o comunque asintoticamente superiore ad ogni polinomio. Questi problemi sono chiamati "intrattabili" perché , pur essendo risolubili per via automatica, richiedono un tempo di calcolo molto elevato, tale da rendere ogni algoritmo di fatto inutilizzabile anche per dimensioni piccole dell'input. Nella tabella presentata nella sezione 1.3 abbiamo visto per esempio che con tempi di calcolo dell'ordine di 2^n, anche eseguendo 10^6 operazioni al secondo, sono necessari parecchi anni per risolvere istanze di dimensione $n = 50$.

Sono certamente intrattabili tutti quei problemi che ammettono soluzione di dimensione esponenziale rispetto all'input. Un esempio è costituito dal problema di determinare i circuiti hamiltoniani di un grafo, ovvero i cicli che passano una e una volta sola per ogni nodo. È chiaro infatti che se il grafo possiede n nodi, può esistere un numero esponenziale di circuiti hamiltoniani (se il grafo è completo ve ne sono $n - 1!$).

Vi sono poi altri problemi (in verità piuttosto rari) che pur ammettendo una uscita di dimensione limitata non possono essere risolti in tempo polinomiale. In generale dimostrare una tale proprietà è abbastanza difficile e sono pochi i risultati di questo genere in letteratura [2].

Le due classi precedenti sono ben definite ed essendo l'una il complementare dell'altra, dovrebbero contenere tutti i problemi risolubili. In realtà molti problemi di notevole interesse non sono stati collocati in nessuna delle due classi. Non sono stati trovati, per questi problemi, algoritmi di soluzione che

[1] È bene ricordare che l'analisi che svolgiamo si riferisce comunque a problemi che ammettono un algoritmo di soluzione. Infatti, è noto che non tutti i problemi possono essere risolti mediante un algoritmo. I problemi che *non* ammettono un algoritmo di soluzione sono detti *non risolubili* o *indecidibili* e sono stati ampiamente studiati a partire dagli anni '30. Tra questi quello più noto è il *problema dell'arresto*; nel nostro contesto, tale problema può essere formulato nel modo seguente:

> Istanza : un programma RAM P e un input I per P.
> Domanda : il programma P si arresta sull'input I?

Questo è solo l'esempio più famoso di una vasta gamma di problemi indecidibili che riguardano il comportamento dei programmi oppure le proprietà delle funzioni da loro calcolate.

[2] Si veda per esempio J.E.Hopcroft, J.D.Ullman, *Introduction to automata theory, languages and computation*, Addison-Wesley, 1979.

funzionano in tempo polinomiale e neppure è stato dimostrato che tali algoritmi non esistono. Tra questi ricordiamo, come esempio, alcuni problemi di decisione già incontrati nei capitoli precedenti e altri, dall'evidente significato intuitivo, che definiremo formalmente nelle sezioni successive: Clique, Knapsack 0-1, Circuito hamiltoniano, Commesso viaggiatore. Si tratta di problemi classici ampiamente studiati in letteratura che trovano applicazioni in vari settori e per i quali sono noti algoritmi che funzionano in tempo esponenziale o subesponenziale, oppure algoritmi di approssimazione polinomiali.

La classe più rappresentativa di questo folto gruppo è quella dei problemi NP-completi. Questa è stata definita all'inizio degli anni '70 e raccoglie una grande varietà di problemi che sorgono naturalmente in vari settori dell'informatica, della ricerca operativa e della matematica discreta. Su di essi è stata sviluppata una notevole letteratura [3] e il loro numero è ormai di parecchie migliaia, raggruppati e studiati per settori specifici.

I problemi NP-completi sono problemi di decisione definiti mediante un paradigma basato sulla nozione di macchina non deterministica (ma anche qui si possono dare diverse definizioni equivalenti). Tali problemi sono computazionalmente equivalenti fra loro, nel senso che un eventuale algoritmo polinomiale per uno solo di essi implicherebbe l'esistenza di un analogo algoritmo per tutti gli altri. Proprio l'assenza di una tale procedura, nonostante gli sforzi compiuti, ha portato alla congettura che i problemi NP-completi non siano risolubili in tempo polinomiale e quindi non siano contenuti nella classe P (anche se finora nessuno ha dimostrato un tale risultato). Questa congettura è ormai così affermata in ambito informatico che stabilire la NP-completezza di un problema equivale di fatto a qualificare il problema come intrattabile o comunque talmente difficile che nessun ricercatore sarebbe oggi in grado di definire un algoritmo efficiente per risolverlo.

13.2 La classe P

Ricordiamo innanzitutto che un *problema di decisione* è un problema che ammette solo due possibili soluzioni (intuitivamente "si" o "no"). Esso può essere rappresentato da una coppia $\langle I, q\rangle$, dove I è l'insieme delle istanze del problema, mentre q è un predicato su I, ovvero una funzione $q : I \to \{0, 1\}$. Nel seguito rappresenteremo spesso il predicato q mediante una opportuna domanda relativa a una istanza qualsiasi $x \in I$.

Definiamo allora P come la classe dei problemi di decisione risolubili da una RAM in tempo polinomiale secondo il criterio di costo logaritmico. Quindi, usando i simboli introdotti nel capitolo 3, possiamo affermare che un problema di decisione π appartiene a P se e solo se esiste una RAM Ψ che risolve π e un polinomio $p(n)$ tali che, per ogni $n \in \mathbb{N}$ e ogni input x di dimensione n,

$$T^l_\Psi(x) \leq p(n).$$

Chiaramente, per dimensione di una istanza x intendiamo il numero di bit necessari per rappresentarla.

È bene osservare che nella definizione precedente il criterio logaritmico non può essere sostituito con quello uniforme. Infatti i due criteri possono fornire valutazioni molto diverse fra loro, anche se per molti programmi RAM i corrispondenti tempi di calcolo sono polinomialmente legati tra loro. In particolare, è abbastanza facile descrivere programmi RAM che, su un input x, richiedono un tempo lineare in $|x|$ secondo il criterio uniforme e uno esponenziale secondo quello logaritmico. Abbiamo già incontrato un programma di questo genere nell'esempio 3.3 .

[3] Si veda in proposito M.R.Garey, D.S.Johnson, *Computers and intractability: a guide to the theory of NP-completeness*, W.H. Freeman, 1979.

13.3 Macchine non deterministiche

Introduciamo in questa sezione un modello di calcolo non deterministico. Si tratta di un modello per alcuni aspetti controintuitivo ma tuttavia importante se vogliamo cogliere a pieno il significato della classe NP che sarà introdotta nella sezione successiva.

In parole povere possiamo dire che una macchina non deterministica è un modello di calcolo che può compiere una o più scelte durante ogni computazione. Di conseguenza il suo funzionamento su un dato input non è più determinato da un'unica computazione ma da un insieme di computazioni distinte, una per ogni possibile sequenza di scelte compiute. La macchina in questo caso è un semplice accettore, ovvero ogni computazione di arresto termina restituendo in uscita 1 oppure 0. Diremo allora che una macchina non deterministica risolve un problema di decisione se, per ogni istanza che ammette risposta positiva, esiste una computazione della macchina su tale istanza che restituisce il valore 1 e, per ogni istanza che ammette risposta negativa, tutte le computazioni relative forniscono in uscita il valore 0.

Prima di dare la definizione formale del modello che consideriamo ricordiamo il funzionamento di una macchina RAM tradizionale seguendo la descrizione data nel capitolo 3. Data una macchina RAM M e un input x, la computazione di M su x consiste in una sequenza (finita o infinita) di configurazioni

$$C_0, C_1, \ldots, C_i, \ldots,$$

ciascuna delle quali descrive l'immagine della macchina in un qualunque istante del calcolo prima dell'esecuzione di ciascuna istruzione. Ogni configurazione definisce quindi il contenuto dei registri $R_0, R_1, \ldots, R_n, \ldots$, il valore del contatore che indica l'istruzione corrente da eseguire, il contenuto dei due nastri di input e output e la posizione della testina di ingresso. La configurazione C_0 è quella iniziale, nella quale tutti i registri R_i sono azzerati, il nastro di uscita contiene solo blank e quello di ingresso l'input x, le due testine sono posizionate sulle prime celle dei rispettivi nastri e il contatore indica la prima istruzione del programma. Ogni C_i, con $i \geq 1$, viene ottenuta dalla configurazione precedente eseguendo l'istruzione indicata dal contatore in C_{i-1}. Diciamo che la macchina M passa dalla configurazione C_{i-1} alla configurazione C_i in un passo e rappresentiamo questa transizione mediante l'espressione $C_{i-1} \vdash_M C_i$. Inoltre, se l'istruzione corrente in C_i non è ben definita (cioè il valore del contatore non indica correttamente una istruzione del programma), allora C_i è di arresto e non esiste una configurazione C' tale che $C_{i-1} \vdash_M C'$. In questo caso la computazione è una sequenza finita e C_i è l'ultima configurazione raggiunta dalla macchina. Se questo non si verifica allora la sequenza $C_0, C_1, \ldots$ è infinita e diciamo che la macchina M non si *arresta* sull'input considerato.

In questo modo $\vdash_M$ definisce una relazione binaria sull'insieme delle configurazioni della macchina M, chiamata anche *relazione di transizione*. Essa è una relazione univoca per ogni macchina RAM M, nel senso che per ogni configurazione C esiste al più una configurazione C' tale che $C \vdash_M C'$.

Una macchina RAM *non deterministica* è una macchina RAM nel cui programma oltre alle istruzioni note possono comparire istruzioni della forma

$$\text{CHOICE}(0, 1)$$

L'esecuzione di questa istruzione avviene semplicemente scegliendo di inserire nell'accumulatore (cioè il registro R_0) il valore 0 oppure il valore 1. La macchina sceglie quindi di eseguire l'istruzione `LOAD` $= 0$ oppure l'istruzione `LOAD` $= 1$. L'esecuzione delle altre istruzioni rimane invariata rispetto al modello tradizionale (che nel seguito chiameremo *deterministico*) così come restano invariate le altre caratteristiche della macchina.

Anche per una RAM non deterministica M possiamo definire la nozione di configurazione e quella di relazione di transizione $\vdash_M$. Ovviamente in questo caso la relazione $\vdash_M$ non è più univoca: per ogni configurazione C su M, se l'istruzione corrente indicata in C è un'istruzione di scelta $\mathrm{CHOICE}(0, 1)$, esisteranno due configurazioni C_1 e C_2 che si ottengono da C eseguendo rispettivamente $\texttt{LOAD} = 0$ e $\texttt{LOAD} = 1$; di conseguenza avremo $C \vdash_M C_1$ e $C \vdash_M C_2$.

Tutto questo implica che, per ogni input x di M, vi saranno in generale più computazioni distinte di M su x, tutte della forma

$$C_0, C_1, \ldots, C_i, \ldots,$$

tali che $C_{i-1} \vdash_M C_i$ per ogni $i \geq 1$ e dove C_0 è la configurazione iniziale di M su input x. Tali computazioni si ottengono semplicemente eseguendo le istruzioni di scelta via via incontrate in tutti i modi possibili.

L'insieme delle computazioni di M su un dato input sono rappresentabili da un albero con radice, dotato eventualmente di un numero infinito di nodi, che gode delle seguenti proprietà:

(i) ogni nodo è etichettato da una configurazione di M;

(ii) la radice è etichettata dalla configurazione iniziale sull'input assegnato;

(iii) se un nodo v è etichettato da una configurazione C ed esiste una sola configurazione C' tale che $C \vdash_M C'$, allora v possiede un solo figlio etichettato da C';

(iv) se un nodo v è etichettato da una configurazione C ed esistono due configurazioni C_1 e C_2 tali che $C \vdash_M C_1$ e $C \vdash_M C_2$, allora v possiede due figli etichettati rispettivamente da C_1 e C_2;

(iv) un nodo v è una foglia se e solo se v è etichettato da una configurazione di arresto.

Dalla definizione precedente è evidente che una computazione di M su un dato input è determinata da un ramo dell'albero di computazione corrispondente, ovvero da un cammino che va dalla radice a una foglia.

Poiché in una macchina non deterministica non esiste una sola computazione su un dato input, dobbiamo definire un paradigma particolare per descrivere il problema risolto da un modello di questo genere. Diremo che un problema di decisione $\pi = \langle I, q \rangle$ è risolto da una RAM non deterministica M se si verificano le condizioni seguenti:

1. per ogni $x \in I$ tale che $q(x) = 1$ esiste una computazione di M su input x che termina restituendo il valore 1 in uscita (si dice anche che la computazione *accetta* l'input);

2. per ogni $x \in I$ tale che $q(x) = 0$ tutte le computazioni di M su input x terminano restituendo il valore 0 in uscita (*rifiutano* l'input).

Nota che in questo modo la stessa macchina M non può essere usata per risolvere il problema complementare di π, definito da $\pi^c = \langle I, p \rangle$ con $p(x) = 1 - q(x)$.

Esempio 13.1
Considera il seguente problema di decisione:

> CLIQUE
> Istanza: un grafo non orientato $G = \langle V, E \rangle$ e un intero $k \leq \sharp V$.
> Domanda: esiste una clique di dimensione k in G, ovvero un insieme $C \subseteq V$ di cardinalità k tale che $\{v, w\} \in E$ per ogni coppia di nodi distinti $v, w \in C$?

Il problema può essere risolto da una RAM non deterministica che esegue la seguente procedura:

```
begin
    A := ∅
    for v ∈ V do
```

```
        begin
              i := Choice(0,1)
              if i = 1 then A := A ∪ {v}
        end
    if ♯A = k ∧ A forma una clique
              then return 1
              else return 0
end
```

Come si verifica facilmente, ogni computazione della macchina è divisa in due fasi distinte: nella prima si costruisce un insieme A di nodi usando l'istruzione di scelta, si dice anche che la macchina *genera in modo non deterministico* l'insieme A; nella seconda fase si verifica se A è effettivamente una clique di dimensione k in G. Nota che questa seconda parte del calcolo è puramente deterministica perché l'istruzione di scelta non viene usata. Inoltre sul grafo $G = \langle V, E\rangle$ la macchina ammette esattamente una computazione per ogni sottoinsieme di V. Quindi se n è il numero dei nodi di G, abbiamo 2^n possibili computazioni distinte sull'input $G = \langle V, E\rangle$.

Chiaramente la procedura (non deterministica) risolve il problema CLIQUE secondo il paradigma formulato: se $G = \langle V, E\rangle$ ammette una clique C di dimensione k la computazione che genera il sottoinsieme C accetta l'input; se invece $G = \langle V, E\rangle$ non ammette una clique di dimensione k tutte le computazioni rifiutano.

Notiamo infine che il numero di computazioni che accettano l'input coincide con il numero di clique di dimensione k in G. ∎

Esempio 13.2
Un altro esempio significativo è relativo al problema del circuito hamiltoniano:

> CIRCUITO HAMILTONIANO
> Istanza: un grafo $G = \langle V, E\rangle$ non orientato.
> Domanda: esiste in G un circuito hamiltoniano, ovvero una permutazione $(v_1, v_2, \ldots, v_n)$ dei nodi del grafo tale che $\{v_i, v_i + 1\} \in E$ per ogni $i = 1, \ldots, n-1$, e inoltre $\{v_n, v_1\} \in E$?

Questo problema può essere risolto da una RAM non deterministica M che, in una prima fase, genera in modo non deterministico una sequenza $(v_{i_1}, v_{i_2}, \ldots, v_{i_n})$ di vertici di G ($n = \sharp V$); in seguito M controlla che tale sequenza sia una permutazione dei nodi del grafo, ovvero che tutti i suoi elementi siano distinti, e quindi verifica se, per ogni $j = 1, 2, \ldots, n-1$, $\{v_{i_j}, v_{i_{j+1}}\}$ appartiene a E e se $\{v_{i_n}, v_{i_1}\} \in E$. Se tutti questi controlli danno esito positivo la macchina accetta, altrimenti rifiuta.

Il suo funzionamento è descritto dalla procedura seguente:

```
begin
    siano v_1, v_2, ..., v_n i vertici di G
    A := Λ (lista vuota)
    for j = 1, 2, ..., n do
        begin
            k := 1
            for i = 1, 2, ..., n − 1 do { ℓ := Choice(0,1)
                                        { k := k + ℓ
            A := INSERISCI_IN_TESTA(A, v_k)
        end
    if esistono due nodi uguali in A
        then return 0
        else if la sequenza di nodi in A forma un ciclo
            then return 1
            else return 0
end
```

Questa procedura fornisce un esempio di costruzione di una permutazione in modo non deterministico.

∎

13.4 La classe NP

Il tempo di calcolo di una macchina non deterministica è definito dalla più lunga computazione della macchina sull'input considerato. Nel nostro caso per valutare il tempo di calcolo di una RAM non deterministica assumiamo il criterio logaritmico. Nel seguito tutti i tempi di calcolo saranno valutati assumendo questo criterio.

Quindi, data una RAM non deterministica M e un input x per M, il tempo richiesto da M su input x è il massimo tempo di calcolo richiesto da una computazione di M su input x secondo il criterio logaritmico. Denoteremo tale quantità con $T_M(x)$. Se esiste una computazione che non termina poniamo chiaramente $T_M(x) = +\infty$. Se invece tutte le computazioni hanno termine possiamo considerare l'albero di computazione di M su x, associando ad ogni lato il costo logaritmico dell'istruzione corrispondente; in questo modo ogni cammino dalla radice a una foglia (computazione) possiede un costo dato dalla somma dei costi dei suoi lati (questi non è altro che il tempo logaritmico della corrispondente computazione). $T_M(x)$ coincide allora con il valore massimo tra i costi di questi cammini.

Inoltre, per ogni $n \in \mathbb{N}$, denotiamo con $T_M(n)$ il massimo tempo di calcolo richiesto da M su un input di dimensione n (come al solito la dimensione di un input in questo caso è data dal numero di bit necessari per rappresentarlo). Quindi

$$T_M(n) = \max\{T_M(x) \mid |x| = n\}$$

Nel seguito diremo anche che M funziona in tempo $f(n)$ se $T_M(n) \leq f(n)$ per ogni $n \in \mathbb{N}$ (con $f : \mathbb{N} \to \mathbb{N}$ funzione qualsiasi).

La classe NP è allora la classe dei problemi di decisione risolubili da una macchina RAM non deterministica che funziona in tempo polinomiale. Quindi un problema di decisione π appartiene a NP se esiste un polinomio $p(x)$ e una macchina RAM non deterministica M che risolve π, tali che $T_M(n) \leq p(n)$ per ogni $n \in \mathbb{N}$.

È facile verificare che le procedure descritte negli esempi 13.1 e 13.2 corrispondono a RAM non deterministiche che funzionano in tempo polinomiale. Di conseguenza i problemi CLIQUE e CIRCUITO HAMILTONIANO appartengono a NP.

Un problema che sorge ora in modo naturale è quello di confrontare i tempi di calcolo delle macchine deterministiche e di quelle non deterministiche. Se per esempio un problema di decisione è risolubile in tempo $f(n)$ su una RAM non deterministica, ci chiediamo in quanto tempo possiamo risolvere lo stesso problema usando una RAM tradizionale (il problema inverso non si pone perché ogni RAM deterministica è una particolare RAM non deterministica). Una macchina non deterministica può essere simulata da una deterministica che semplicemente esplora l'albero di computazione della precedente. Il procedimento è illustrato dalla seguente proposizione che mostra anche come i due modelli di calcolo risolvano la stessa classe di problemi di decisione anche se impiegando tempi di calcolo differenti.

Proposizione 13.1 *Per ogni funzione $f : \mathbb{N} \longrightarrow \mathbb{N}$ e ogni problema di decisione π risolubile da una RAM non deterministica in tempo $f(n)$, esiste un valore $c > 1$ e una RAM deterministica che risolve π in tempo $O(c^{f(n)})$.*

Dimostrazione. Sia M una RAM non deterministica che risolve π in tempo $f(n)$. L'idea è quella di simulare M mediante una RAM deterministica M' che, su input x, esplora l'albero di computazione di M su x mediante una ricerca in ampiezza. Durante il calcolo Q rappresenta la coda che mantiene le configurazioni di M raggiungibili dalla configurazione iniziale. Il funzionamento della macchina M' sull'input x è definito dalla seguente procedura.

```
begin
  Sia C_0(x) la configurazione iniziale di M su input x
  Q := ENQUEUE(Λ, C_0(x))
  A := 0
  while Q ≠ Λ ∧ A = 0 do
        begin
              C := FRONT(Q)
              Q := DEQUEUE(Q)
              if C accettante then A = 1
              if esiste una sola configurazione C' tale che C ⊢_M C'
                    then Q := ENQUEUE(Q, C')
              if esistono due configurazioni C_1, C_2 tali che C ⊢_M C_1 e C ⊢_M C_2
                    then { Q := ENQUEUE(Q, C_1)
                         { Q := ENQUEUE(Q, C_2)
        end
  return A
end
```

Non è difficile verificare che ogni configurazione di M su un input x, dove $|x| = n$, può essere rappresentata da una sequenza di interi di dimensione complessiva $O(f(n))$. Data la rappresentazione di una configurazione C di M, la macchina M' deve essere in grado di determinare la rappresentazione delle configurazioni successive, cioè delle configurazioni C' tali che $C \vdash_M C'$. Questo calcolo può essere certamente eseguito in un tempo $O(f(n))^2$, tenendo anche conto del costo di indirizzo degli interi coinvolti.

Ritornando al comportamento di M su input x, osserviamo che nel corrispondente albero di computazione vi sono al più $2^{f(n)+1}$ nodi. Quindi il ciclo while della precedente procedura può essere eseguito al più $O(2^{f(n)})$ volte. Ogni ciclo esegue un numero costante di operazioni su configurazioni di M e, per l'osservazione precedente, ogni singola esecuzione richiede al più $O(f(n))^2$ passi. Il tempo complessivo risulta così $O(f(n)^2 2^{f(n)}) = O(3^{f(n)})$. ∎

Una immediata conseguenza della proposizione precedente è data dal seguente

Corollario 13.2 *Ogni problema in NP è risolubile da una RAM deterministica in tempo $O(c^{p(n)})$ per qualche $c > 1$ e qualche polinomio $p(x)$.*

Questo significa che tutti i problemi in NP possono essere risolti in tempo esponenziale da una macchina deterministica.

Esercizio

Dimostrare che il seguente problema appartiene a NP:

COMMESSO VIAGGIATORE

Istanza: un intero $k > 0$, un grafo diretto $G = \langle V, E \rangle$ e una funzione peso $d : E \to \mathbb{N}$.

Domanda: esiste un circuito che congiunge tutti i nodi del grafo di peso minore o uguale a k, ovvero una permutazione $(v_1, v_2, \ldots, v_n)$ di V tale che $\sum_{i=1}^{n-1} d(v_i, v_{i+1}) + d(v_n, v_1) \leq k$?

13.5 Il problema della soddisfacibilità

In questa sezione illustriamo il problema della soddisfacibilità per formule booleane in forma normale congiunta. Questo problema ha una particolare importanza in questo contesto perché è stato il primo problema NP-completo introdotto in letteratura.

Innanzitutto ricordiamo che, per definizione, una formula booleana è un letterale, cioè una variabile x o una variabile negata $\overline{x}$, oppure una delle seguenti espressioni:

(i) $(\neg\phi)$,

(ii) $\phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_k$,

(iii) $(\phi_1 \vee \phi_2 \vee \cdots \vee \phi_k)$,

dove $k \geq 2$, mentre $\phi, \phi_1, \phi_2, \ldots, \phi_k$ sono formule booleane. Nel seguito rappresenteremo con i simboli di somma e prodotto le tradizionali operazioni $\vee$ e $\wedge$. Ogni formula booleana nella quale compaiono k variabili distinte definisce in modo ovvio una funzione $f : \{0,1\}^k \longrightarrow \{0,1\}$.

Diciamo che una formula booleana ϕ è in forma normale congiunta (FNC per brevità) se ϕ è un prodotto di clausole di letterali, ovvero

$$\phi = E_1 \cdot E_2 \cdot \ldots \cdot E_k \tag{13.1}$$

dove $E_i = (\ell_{i1} + \ell_{i2} + \cdots + \ell_{it_i})$ per ogni $i = 1, 2, \ldots, k$, e ciascun ℓ_{ij} è un letterale.

Un esempio di formula booleana in FNC è dato dall'espressione

$$U(y_1, y_2, \ldots, y_k) = (\sum_{i=1}^{k} y_i) \cdot \prod_{i \neq j} (\overline{y_i} + \overline{y_j}). \tag{13.2}$$

È evidente che $U(y_1, y_2, \ldots, y_k)$ vale 1 se e solo se esattamente una delle sue variabili y_i assume valore 1.

Si può dimostrare che ogni funzione booleana può essere rappresentata da una formula in forma normale congiunta. Per verificare questa semplice proprietà, introduciamo la seguente definizione: per ogni variabile x e ogni $c \in \{0,1\}$, poniamo

$$\overline{x}^c = \begin{cases} \overline{x} & \text{se } c = 1 \\ x & \text{se } c = 0 \end{cases}$$

Nota che $\overline{x}^c$ assume il valore 1 se e solo se i valori di x e c sono diversi.

Lemma 13.3 *Sia $f : \{0,1\}^n \longrightarrow \{0,1\}$ una funzione booleana e sia $S_0 = \{\underline{c} \in \{0,1\}^n / f(\underline{c}) = 0\} \neq \emptyset$. Allora per ogni $(x_1, x_2, \ldots x_n) \in \{0,1\}^n$, denotando con $\underline{c}$ il vettore $(c_1, c_2, \ldots c_n)$, abbiamo*

$$f(x_1, x_2, \ldots, x_n) = \prod_{\underline{c} \in S_0} (\overline{x}_1^{c_1} + \overline{x}_2^{c_2} + \cdots + \overline{x}_n^{c_n})$$

Dimostrazione. Osserviamo che, per la definizione precedente, la clausola $(\overline{x}_1^{c_1} + \overline{x}_2^{c_2} + \cdots + \overline{x}_n^{c_n})$ ha valore 1 se e solo se, per qualche i, x_i assume un valore diverso da c_i. Ne segue che la produttoria precedente è uguale a 1 se e solo se la n-pla $(x_1, x_2, \ldots x_n)$ assume un valore in $\{0,1\}^n$ che non appartiene a S_0. □

Il problema della soddisfacibilità per formule booleane in FNC è definito nel modo seguente:

SODD-FNC
Istanza: una formula booleana ϕ in forma normale congiunta.
Domanda: esiste una assegnamento di valori 0 e 1 alle variabili che rende vera ϕ?

È facile provare che questo problema appartiene a NP. Infatti, su input ϕ, una RAM può generare in modo non deterministico un assegnamento A di valori alle variabili di ϕ. Quindi la macchina verifica in modo deterministico se l'assegnamento A rende vera ϕ. In caso affermativo la macchina accetta, altrimenti rifiuta. È evidente che ogni computazione della macchina termina dopo un numero polinomiale di passi.

Proposizione 13.4 *SODD-FNC appartiene alla classe NP.*

Esercizi

1) Dare la definizione di formula booleana in forma normale disgiunta e dimostrare che ogni funzione booleana, a un solo valore, può essere rappresentata da una formula di questo tipo.
2) Il problema della soddisfacibilità per formule booleane in forma normale *disgiunta* appartiene a P?

13.6 Riducibilità polinomiale

Consideriamo due problemi di decisione $\pi_1 = \langle I_1, q_1\rangle$ e $\pi_2 = \langle I_2, q_2\rangle$, dove I_1 e I_2 sono gli insiemi delle istanze, mentre q_1 e q_2 i rispettivi predicati. Diciamo che π_1 è *polinomialmente riducibile* a π_2 se esiste una funzione $f : I_1 \longrightarrow I_2$, calcolabile in tempo polinomiale da una RAM deterministica (secondo il criterio logaritmico), tale che per ogni $x \in I_1$, $q_1(x) = 1$ se e solo se $q_2(f(x)) = 1$. Diremo anche che f definisce una riduzione polinomiale da π_1 a π_2 o anche che π_1 è riducibile a π_2 mendiante la funzione f.

È facile verificare che la riduzione polinomiale gode della proprietà transitiva. Infatti, dati tre problemi di decisione $\pi_1 = \langle I_1, q_1\rangle$, $\pi_2 = \langle I_2, q_2\rangle$ e $\pi_3 = \langle I_3, q_3\rangle$, se π_1 è polinomialmente riducibile a π_2 mediante una funzione f, e π_2 è polinomialmente riducibile a π_3 mediante una funzione g, allora la funzione composta $h(x) = g(f(x))$ definisce una riduzione polinomiale da π_1 a π_3: per ogni $x \in I_1$, $q_1(x) = 1$ se e solo se $q_3(h(x)) = 1$; inoltre la funzione h è calcolabile da una RAM che, su input $x \in I_1$, simula prima la macchina che calcola f, ottenendo $f(x)$, e quindi la macchina che calcola g, ricavando $g(f(x))$. Il tempo richiesto da quest'ultimo calcolo è comunque polinomiale perché maggiorato dalla composizione di due polinomi.

Le classi P e NP definite nelle sezioni precedenti sono chiaramente chiuse rispetto alla riduzione polinomiale.

Proposizione 13.5 *Dati due problemi di decisione $\pi_1 = \langle I_1, q_1\rangle$ e $\pi_2 = \langle I_2, q_2\rangle$, supponiamo che π_1 sia polinomialmente riducibile a π_2. Allora se π_2 appartiene a P anche π_1 appartiene a P. Analogamente, se π_2 appartiene a NP, anche π_1 appartiene a NP.*

Dimostrazione. Sia f la funzione che definisce una riduzione polinomiale da π_1 a π_2 e sia M una RAM che calcola f in tempo $p(n)$, per un opportuno polinomio p. Se $\pi_2 \in P$ allora esiste una RAM deterministica M' che risolve π_2 in tempo $q(n)$, dove q è un'altro polinomio. Allora possiamo definire una RAM deterministica M'' che, su input $x \in I_1$, calcola la stringa $y = f(x) \in I_2$ simulando la macchina M; quindi M'' simula la macchina M' su input $f(x)$ e mantiene la risposta di quest'ultima.

Poiché la lunghezza di $f(x)$ è al più data da $p(|x|)$, la complessità in tempo di M'' è maggiorata da un polinomio:

$$T_{M''}(|x|) \leq p(|x|) + q(p(|x|))$$

In modo analogo si prova che, se π_2 appartiene a NP, anche π_1 appartiene a NP. ∎

Una immediata conseguenza della proprietà precedente è data dal seguente corollario.

Corollario 13.6 *Dati due problemi di decisione π_1 e π_2, supponiamo che π_1 sia polinomialmente riducibile a π_2. Allora se π_1 non appartiene a P anche π_2 non appartiene a P. Analogamente, se π_1 non appartiene a NP, anche π_2 non appartiene a NP.*

13.6.1 Riduzione polinomiale da SODD-FNC a CLIQUE

Presentiamo ora un esempio di riduzione polinomiale tra problemi di decisione.

Proposizione 13.7 *SODD-FNC è polinomialmente riducibile a CLIQUE.*

Dimostrazione. Descriviamo una funzione f tra le istanze dei due problemi che definisce la riduzione. Sia ϕ una formula in FNC definita dalla uguaglianza

$$\phi = E_1 \cdot E_2 \cdot \ldots \cdot E_k$$

dove $E_i = (\ell_{i1} + \ell_{i2} + \cdots + \ell_{it_i})$ per ogni $i = 1, 2, \ldots, k$, e ciascun ℓ_{ij} è un letterale. Allora $f(\phi)$ è l'istanza del problema CLIQUE data dall'intero k, pari al numero di clausole di ϕ, e dal grafo $G = \langle V, E \rangle$ definito nel modo seguente:

- $V = \{[i, j] \mid i \in \{1, 2, \ldots, k\}, j \in \{1, 2, \ldots, t_i\}\}$;

- $E = \{\{[i, j], [u, v]\} \mid i \neq u, \ell_{ij} \neq \overline{\ell_{uv}}\}$.

Quindi i nodi di G sono esattamente le occorrenze di letterali in ϕ, mentre i suoi lati sono le coppie di letterali che compaiono in clausole distinte e che non sono l'uno il negato dell'altro.

È facile verificare che la funzione f è calcolabile in tempo polinomiale.

Proviamo ora che ϕ ammette un assegnamento che la rende vera se e solo se G ha una clique di dimensione k. Supponiamo che esista un tale assegnamento A. Chiaramente A assegna valore 1 ad almeno un letterale ℓ_{ij_i} per ogni clausola E_i di ϕ. Sia $\{j_1, j_2, \ldots, j_k\}$ l'insieme degli indici di questi letterali. Allora l'insieme $C = \{[i, j_i] \mid i = 1, 2, \ldots, k\}$ è una clique del grafo G perché, se per qualche $i, u \in \{1, 2, \ldots, k\}$, $\ell_{ij_i} = \overline{\ell_{uj_u}}$, l'assegnamento A non potrebbe rendere veri entrambi i letterali.

Viceversa, sia $C \subseteq V$ una clique di G di dimensione k. Allora, per ogni coppia $[i, j]$, $[u, v]$ in C, abbiamo $i \neq u$ e $\ell_{ij} \neq \overline{\ell_{uv}}$. Sia ora S_1 l'insieme delle variabili x tali che $x = \ell_{ij}$ per qualche $[i, j] \in C$. Analogamente, denotiamo con S_0 l'insieme delle variabili y tali che $\overline{y} = \ell_{ij}$ per qualche $[i, j] \in C$. Definiamo ora l'assegnamento A che attribuisce valore 1 alle variabili in S_1 e valore 0 alle variabili in S_0. A è ben definito perché, essendo C una clique, l'intersezione $S_1 \cap S_0$ è vuota. Inoltre A rende vera la formula ϕ, perché per ogni clausola E_i vi è un letterale ℓ_{ij_i} che assume valore 1. ∎

13.6.2 Riduzione polinomiale da SODD-FNC a 3-SODD-FNC

Il problema SODD-FNC può essere ridotto polinomialmente anche a una sua variante che si ottiene restringendo le istanze alle formule booleane in FNC che possiedono solo clausole con al più 3 letterali.

3-SODD-FNC
Istanza: un formula booleana $\psi = F_1 \cdot F_2 \cdot \ldots \cdot F_k$, dove $F_i = (\ell_{i1} + \ell_{i2} + \ell_{i3})$ per ogni $i = 1, 2, \ldots, k$ e ogni ℓ_{ij} è una variabile o una variabile negata.
Domanda: esiste una assegnamento di valori 0 e 1 alle variabili che rende vera ψ?

Proposizione 13.8 *SODD-FNC è polinomialmente riducibile a* 3*-SODD-FNC.*

Dimostrazione. Sia ϕ una formula booleana in FNC e sia $E = (\ell_1 + \ell_2 + \cdots + \ell_t)$ una clausola di ϕ dotata di $t \geq 4$ letterali. Sostituiamo E in ϕ con un prodotto di clausole $f(E)$ definito da

$$f(E) = (\ell_1 + \ell_2 + y_1) \cdot (\ell_3 + \overline{y_1} + y_2) \cdot (\ell_4 + \overline{y_2} + y_3) \cdot \ldots \cdot (\ell_{t-2} + \overline{y_{t-4}} + y_{t-3}) \cdot (\ell_{t-1} + \ell_t + \overline{y_{t-3}})$$

dove $y_1, y_2, \ldots, y_{t-3}$ sono nuove variabili.

Proviamo ora che esiste un assegnamento che rende vera la clausola E se e solo se ne esiste uno che rende vera $f(E)$. Infatti, se per qualche i $\ell_i = 1$, basta porre $y_j = 1$ per ogni $j < i - 1$ e $y_j = 0$ per ogni $j \geq i - 1$; in questo modo otteniamo un assegnamento che rende vera $f(E)$. Viceversa, se esiste un assegnamento che rende vera $f(E)$ allora deve esistere qualche letterale ℓ_i che assume valore 1. Altrimenti è facile verificare che il valore di $f(E)$ sarebbe 0. Ne segue che lo stesso assegnamento rende vera la E.

Operando questa sostituzione per tutte le clausole di ϕ dotate di più di 3 addendi, otteniamo una formula 3-FNC che soddisfa le condizioni richieste. È inoltre evidente che il tempo di calcolo necessario per realizzare la sostituzione è polinomiale. ∎

13.7 Il teorema di Cook

Come abbiamo illustrato nella sezione precedente, la riduzione polinomiale può essere interpretata come una relazione che confronta la difficoltà computazione dei problemi: intuitivamente, se un problema è polinomialmente riducibile a un altro problema significa che il primo è computazionalmente più facile del secondo (a almeno tanto difficile quanto il secondo). In questo contesto diventa allora interessante individuare i problemi computazionalmente più difficili, che chiameremo NP-completi e che saranno i massimi della relazione di riduzione polinomiale.

Definizione 13.1 *Un problema di decisione π è NP-completo se*

1. *π appartiene a NP,*
2. *per ogni $\pi' \in$ NP si verifica che π' è polinomialmente riducibile a π.*

Intuitivamente i problemi NP-completi sono quindi i problemi più difficili nella classe NP. L'esistenza di un algoritmo che risolve in tempo polinomiale un problema NP-completo implicherebbe l'equivalenza delle classi P e NP. Tuttavia l'ipotesi P=NP è considerata molto improbabile, dato l'elevato numero di problemi in NP per i quali non sono stati trovati algoritmi polinomiali, anche se finora nessuno ha

formalmente provato che le due classi sono diverse [4]. Quindi dimostrare che un problema π è NP-completo significa sostanzialmente provare che il problema è intrattabile ovvero che, quasi certamente, π non ammette algoritmi di soluzione che lavorano in tempo polinomiale.

Il primo problema NP-completo scoperto in letteratura è proprio il problema SODD-FNC definito nella sezione 13.5. Questo risultato, provato da Cook nel 1971, è molto importante perché ha consentito di determinare l'NP-completezza di molti altri problemi per i quali non erano noti algoritmi di risoluzione polinomiali. Questo da una parte ha permesso di spiegare l'inerente difficoltà di questi problemi, dall'altra ha dato avvio allo studio delle loro proprietà comuni.

Teorema 13.9 *Il problema SODD-FNC è NP-completo.*

Osserviamo subito che, poiché la riducibilità polinomiale gode della proprietà transitiva, se SODD-FNC è polinomialmente riducibile a un problema $\pi \in NP$, anche quest'ultimo risulta NP-completo. Applicando quindi i risultati presentati nella sezione precedente, possiamo enunciare la seguente proposizione.

Corollario 13.10 *I problemi CLIQUE e 3-SODD-FNC sono NP-completi.*

Questo corollario mostra come, usando la riduzione polinomiale sia possibile dimostrare l'NP-completezza di un problema. Quasi tutti i problemi NP-completi noti in letteratura sono stati ottenuti mediante riduzione polinomiale da un problema dello stesso tipo.

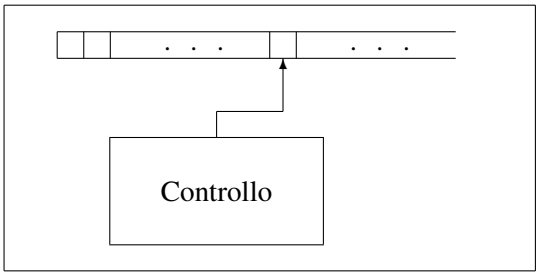
La dimostrazione tradizionale del teorema di Cook richiede l'introduzione di un modello di calcolo elementare, più semplice rispetto alle macchine RAM. Si tratta della nota macchina di Turing introdotta già negli anni '30 per studiare le proprietà dei problemi indecidibili. Questo modello di calcolo è computazionalmente equivalente alle RAM (assumendo il costo logaritmico) ma è troppo elementare per descrivere in maniera efficace il funzionamento di algoritmi e procedure come sono oggi comunemente intese. La sua semplicità tuttavia consente di provare in maniera abbastanza diretta alcune proprietà generali di calcolo tra le quali anche l'esistenza delle riduzioni polinomiali menzionate proprio nel teorema di Cook.

13.7.1 Macchine di Turing

Intuitivamente una macchina di Turing (MdT nel seguito) è un modello di calcolo costituito da un insieme di stati, un nastro suddiviso in celle e una testina di lettura posizionata su una di queste. L'insieme degli stati contiene uno stato particolare, chiamato stato iniziale e un sottoinsieme di stati detti finali. Ogni cella contiene un solo simbolo estratto da un alfabeto fissato. Il funzionamento della macchina è determinato da un controllo centrale che consente di eseguire una sequenza di mosse a partire da una configurazione iniziale. In questa configurazione lo stato corrente è quello iniziale, una stringa di input è collocata nelle prime celle del nastro e la testina legge il primo simbolo; tutte le altre celle contegono un simbolo speciale che chiameremo blanck. Quindi ogni mossa è univocamente determinata dallo stato nel quale la macchina si trova e dal simbolo letto dalla testina. Eseguendo una mossa la macchina può entrare in un nuovo stato, stampare un nuovo simbolo nella cella su cui è posizionata la testina di lettura e, infine, spostare quest'ultima di una posizione a destra oppure a sinistra. Se la sequenza di mosse eseguite è finita diciamo che la macchina si arresta sull'input considerato e diciamo che tale input è accettato se lo stato raggiunto nell'ultima configurazione è finale. In questo modo si può definire precisamente il

[4] Il problema di dimostrare che P è diverso da NP è considerato oggi uno dei più importanti problemi aperti della teoria degli algoritmi.

problema di decisione risolto da una MdT: diciamo che la macchina M risolve un problema di decisione $\pi = \langle I, q\rangle$ se I è l'insieme delle possibili stringhe di input della macchina, M si arresta su ogni input $x \in I$ e accetta x se e solo se $q(x) = 1$.



Formalmente possiamo quindi definire una macchina di Turing (deterministica) come una vettore

$$M = \langle Q, \Sigma, \Gamma, q_0, B, \delta, F\rangle$$

dove Q è un insieme finito di stati, Γ un alfabeto di lavoro, $\Sigma \subset \Gamma$ un alfabeto di ingresso, $B \in \Gamma\backslash\Sigma$ un simbolo particolare che denota il blank, $q_0 \in Q$ lo stato iniziale, $F \subseteq Q$ l'insieme degli stati finali e δ una funzione transizione, cioè una funzione parziale

$$\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{-1, +1\}$$

Per ogni $q \in Q$ e ogni $a \in \Gamma$, il valore $\delta(q, a)$ definisce la mossa di M quando la macchina si trova nello stato q e legge il simbolo a: se $\delta(q, a) = (p, b, \ell)$ allora p rappresenta il nuovo stato, b il simbolo scritto nella cella di lettura, ℓ lo spostamento della testina. La testina si sposta rispettivamente di una cella a sinistra o a destra a seconda se $\ell = -1$ oppure $\ell = +1$.

Una configurazione di M è l'immagine complessiva della macchina prima o dopo l'esecuzione di una mossa. Questa è determinata dallo stato corrente, dal contenuto del nastro e dalla posizione della testina di lettura. Definiamo quindi una configurazione di M come una stringa [5]

$$\alpha q \beta$$

dove $q \in Q$, $\alpha \in \Gamma^*$, $\beta \in \Gamma^+$. α rappresenta la stringa collocata a sinistra della testina di lettura mentre β, seguita da infiniti blank, è la sequenza di simboli che si trova alla sua destra. In particolare la testina di lettura è posizionata sul primo simbolo di β. Inoltre supporremo che B non sia un suffisso di β a meno che $\beta = B$. In questo modo $\alpha\beta$ definisce la parte significativa del nastro: nel seguito diremo che essa rappresenta la porzione non blank del nastro. Denoteremo inoltre con $\mathcal{C}_M$ l'insieme delle configurazioni di M.

[5] Ricordiamo che una stringa (o parola) su un dato alfabeto A è una concatenazione finita di simboli estratti da A. L'insieme di tutte le stringhe su A, compresa la parola vuota ϵ, è denotato da A^*, mentre A^+ rappresenta l'insieme delle parole su A diverse da ϵ. La lunghezza di una parola x è il numero di simboli che compaiono in x ed è denotata da $|x|$. Chiaramente $|\epsilon| = 0$.

La configurazione iniziale di M su input $w \in \Sigma^*$ è q_0B, se w è la parola vuota, mentre è la stringa q_0w, se $w \in \Sigma^+$. Nel seguito denoteremo con $C_0(w)$ tale configurazione. Diremo inoltre che una configurazione $\alpha q\beta$ è accettante se $q \in F$.

Possiamo ora definire la relazione di transizione in un passo. Questa è una relazione binaria $\vdash_M$ sull'insieme $\mathcal{C}_M$ delle configurazioni di M. Intuitivamente, per ogni $C, C' \in \mathcal{C}_M$, vale $C \vdash_M C'$ se la macchina M nella configurazione C raggiunge mediante una mossa la configurazione C'. Più precisamente, data la configurazione $\alpha q\beta \in \mathcal{C}_M$, supponiamo che $\beta = b\beta'$ dove $b \in \Gamma$, $\beta' \in \Gamma^*$ e che $\delta(q, b) = (p, c, \ell)$. Distinguiamo quindi i seguenti casi:

1) se $\ell = +1$ allora

$$\alpha qb\beta' \vdash_M \begin{cases} \alpha cp\beta' & \text{se } \beta' \neq \epsilon \\ \alpha cpB & \text{se } \beta' = \epsilon \end{cases}$$

2) se $\ell = -1$ e $|\alpha| \geq 1$ allora, ponendo $\alpha = \alpha' a$ con $\alpha' \in \Gamma^*$ e $a \in \Gamma$, abbiamo

$$\alpha qb\beta' \vdash_M \begin{cases} \alpha' pa & \text{se } c = B \text{ e } \beta' = \epsilon \\ \alpha' pac\beta' & \text{altrimenti} \end{cases}$$

Nel seguito denotiamo con $\vdash_M^*$ la chiusura riflessiva e transitiva di $\vdash_M$.

Osserviamo che se $\delta(q, b)$ non è definito, oppure $\ell = -1$ e $\alpha = \epsilon$, allora non esiste alcuna configurazione $C' \in \mathcal{C}_M$ tale che $\alpha q\beta \vdash_M C'$. In questo caso diciamo che $\alpha q\beta$ è una configurazione di arresto per M. Senza perdita di generalità possiamo supporre che ogni configurazione accettante sia una configurazione di arresto.

Una sequenza finita $\{C_i\}_{0 \leq i \leq m}$ di configurazioni di M è una computazione di M su input $w \in \Sigma^*$ se $C_0 = C_0(w)$, $C_{i-1} \vdash_M C_i$ per ogni $i = 1, 2, \ldots, m$ e C_m è una configurazione di arresto per M. Se inoltre C_m è accettante diciamo che M accetta l'input w, altrimenti diciamo che lo rifiuta.

Se invece la macchina M su input w non si ferma la sua computazione è definita da una sequenza infinita di configurazioni $\{C_i\}_{0 \leq i < +\infty}$, tali che $C_0 = C_0(w)$ e $C_{i-1} \vdash_M C_i$ per ogni $i \geq 1$.

Supponi ora che M si arresti su ogni input $x \in \Sigma^*$. Allora diciamo che M risolve il problema di decisione $\langle \Sigma^*, q \rangle$ dove, per ogni $x \in \Sigma^*$,

$$q(x) = \begin{cases} 1 & \text{se } M \text{ accetta } x \\ 0 & \text{altrimenti} \end{cases}$$

Esempio 13.3
Consideriamo ora il linguaggio $L = \{x \in \{a, b\}^* \mid x = x^R\}$, dove x^R è l'inversa di x. Si può facilmente descrivere una MdT che verifica se una stringa appartiene a L. Tale macchina, su un input $y \in \{a, b\}^*$ di lunghezza n, confronta i simboli di posizione i e $n - i + 1$ e accetta se questi sono uguali tra loro per ogni $i = 1, 2, \ldots, n$. Il calcolo avviene percoccendo n volte la stringa di input alternando la direzione do spostamento della testina e marcando opportunamente i simboli letti. ∎

Definiamo ora la nozione di tempo di calcolo di una MdT su un dato input. Data una MdT $M = \langle Q, \Sigma, \Gamma, q_0, B, \delta, F \rangle$, per ogni $w \in \Sigma^*$, denotiamo con $T_M(w)$ il numero di mosse compiute da M su input w. Se M non si arresta poniamo $T_M(w) = +\infty$. Inoltre, per ogni $n \in \mathbb{N}$, denotiamo con $T_M(n)$ il massimo numero di mosse compiute da M su un input di lunghezza n:

$$T_M(n) = \max\{T_M(w) \mid w \in \Sigma^*, |x| = n\}$$

In questo modo T_M è una funzione $T_M : \mathbb{N} \longrightarrow \mathbb{N} \cup \{+\infty\}$ che chiameremo complessità in tempo di M. Data una funzione $f : \mathbb{N} \longrightarrow \mathbb{R}^+$, diremo che M lavora in tempo $f(n)$ se $T_M(n) \leq f(n)$ per ogni $n \in \mathbb{N}$. Se inoltre $f(n)$ è un polinomio in n diremo che M funziona in tempo polinomiale.

Esempio 13.4
È facile verificare che la MdT descritta nell'esempio 13.3 lavora in tempo $O(n^2)$. ∎

È evidente che per ogni MdT possiamo definire una macchina RAM che esegue la stessa computazione (a patto di codificare opportunamente i simboli dell'alfabeto di lavoro). Vale inoltre una proprietà inversa che consente di determinare, per ogni macchina RAM, una MdT equivalente. Inoltre i tempi di calcolo delle due macchine sono polinomialmente legati tra loro.

Proposizione 13.11 *Se un problema di decisione π è risolubile in tempo $T(n)$ da una RAM M secondo il criterio logaritmico, allora esiste una MdT M' che risolve π in tempo $p(T(n))$ per un opportuno polinomio p.*

Questo significa che la classe P coincide con la classe dei problemi di decisione risolubili da una MdT in tempo polinomiale.

Come per le RAM anche per le macchine di Turing possiamo definire un modello non deterministico.

Formalmente una MdT non deterministica è un vettore $M = \langle Q, \Sigma, \Gamma, q_0, B, \delta, F\rangle$ dove Q, Σ, Γ, q_0, B e F sono definite come nel caso precedente, mentre δ è una funzione

$$\delta : Q \times \Gamma \longrightarrow 2^{Q\times\Gamma\times\{-1,+1\}}$$

La macchina M, trovandosi nello stato $q \in Q$ e leggendo il simbolo $a \in \Gamma$, può scegliere la mossa da compiere nell'insieme $\delta(q, a)$. Come nella sezione precedente, possiamo definire le configurazioni di M, le relazioni di transizione $\vdash_M$ e $\vdash^*_M$, e le computazioni di M. È evidente che, in questo caso, per ogni configurazione $C \in \mathcal{C}_M$ possono esistere più configurazioni raggiungibili da C in un passo. Per questo motivo, su un dato input, la macchina M può eseguire computazioni diverse, una per ogni possibile sequenza di scelte compiute da M a ogni passo.

Diremo che un input $w \in \Sigma^*$ è accettato da M se esiste una computazione di M su input w che conduce la macchina in una configurazione accettante, ovvero se esiste $C \in \mathcal{C}_M$ tale che $C = \alpha q\beta$, $q \in F$ e $C_0(w) \vdash^*_M C$.

Viceversa, se tutte le computazioni di M su input x terminano in una configurazione non accettante, diciamo che M rifiuta x.

Se tutte le computazioni di M hanno termine su ogni possibile input diciamo che M risolve il problema di decisione $\pi = \langle \Sigma^*, q\rangle$ dove, per ogni $x \in \Sigma^*$,

$$q(x) = \begin{cases} 1 & \text{se } M \text{ accetta } x \\ 0 & \text{altrimenti} \end{cases}$$

Data una MdT non deterministica $M = \langle Q, \Sigma, \Gamma, q_0, B, \delta, F\rangle$ e una stringa $w \in \Sigma^*$, denotiamo con $T_M(w)$ il massimo numero di mosse che la macchina può compiere in una computazione su input w. In altre parole $T_M(w)$ rappresenta l'altezza dell'albero di computazione di M su input w. Chiaramente, $T_M(w) = +\infty$ se e solo se esiste una computazione di M su tale input che non si arresta.

Inoltre, per ogni $n \in \mathbb{N}$, denotiamo con $T_M(n)$ il massimo valore di $T_M(w)$ al variare delle parole $w \in \Sigma^*$ di lunghezza n:

$$T_M(n) = \max\{T_M(w) \mid w \in \Sigma^*, |w| = n\}$$

Di nuovo, data una funzione $f : \mathbb{N} \longrightarrow \mathbb{R}^+$, diremo che una MdT non deterministica M lavora in tempo $f(n)$ se $T_M(n) \leq f(n)$ per ogni $n \in \mathbb{N}$.

Anche per le MdT non deterministiche valgono le proprietà di simulazione del corrispondente modello RAM. Questo consente di enunciare la seguente

Proposizione 13.12 *Un problema di decisione π appartiene a NP se e solo se esiste un polinomio p e una MdT M non deterministica che risolve π in un tempo $f(n)$ tale che $f(n) \leq p(n)$ per ogni $n \in \mathbf{N}$.*

13.7.2 Dimostrazione

In questa sezione presentiamo la dimostrazione del teorema 13.9. Nella sezione 13.5 abbiamo già dimostrato che SODD-FNC appartiene a NP. Dobbiamo quindi provare che ogni problema $\pi \in$ NP è polinomialmente riducibile a SODD-FNC. In altre parole vogliamo dimostrare che per ogni MdT non deterministica M, che lavora in tempo polinomiale, esiste una funzione f, calcolabile in tempo polinomiale, che associa a ogni stringa di input w di M una formula booleana ϕ, in forma normale congiunta, tale che w è accettata dalla macchina se e solo se esiste un assegnamento di valori alle variabili che rende vera ϕ.

Senza perdita di generalità, possiamo supporre che, per un opportuno polinomio p e per ogni input w di M, tutte le computazione della macchina su w abbiano la stessa lunghezza $p(|w|)$. Infatti, se M non soddisfa quest'ultima condizione, possiamo sempre costruire una nuova macchina che, su input w, prima calcola $p(|w|)$, poi simula M su w tenendo un contatore del numero di mosse eseguite e prolungando ogni computazione fino a $p(|w|)$ passi.

Supponiamo inoltre che la macchina M sia definita da $M = \langle Q, \Sigma, \Gamma, q_1, B, \delta, F\rangle$, dove $Q = \{q_1, q_2, \ldots, q_s\}$ e $\Gamma = \{a_1, a_2, \ldots, a_r\}$. Data ora una stringa $w \in \Sigma^*$ di lunghezza n, sappiamo che ogni computazione di M su w è una sequenza di $p(n) + 1$ configurazioni, ciascuna delle quali è rappresentabile da una stringa $\alpha q \beta$ tale che $|\alpha\beta| \leq p(n) + 1$. La corrispondente formula $\phi = f(w)$ sarà definita su tre tipi di variabili booelane: $S(u, t), C(i, j, t)$ e $L(i, t)$, dove gli indici i, j, t, u variano in modo opportuno. Il significato intuitivo di queste variabili è il seguente:

- la variabile $S(u, t)$ assumerà il valore 1 se al tempo t la macchina si trova nello stato q_u;

- la variabile $C(i, j, t)$ assumerà valore 1 se al tempo t nella cella i-esima si trova il simbolo a_j;

- la variabile $L(i, t)$ assumerà il valore 1 se al tempo t la testina di lettura è posizionata sulla cella i-esima.

È chiaro che $u \in \{1, 2, \ldots, s\}$, $t \in \{0, 1, \ldots, p(n)\}$, $i \in \{1, 2, \ldots, p(n) + 1\}$ e $j \in \{1, 2, \ldots, r\}$. Un assegnamento di valori 0 e 1 a queste variabili rappresenta una computazione accettante di M su input w se le seguenti condizioni sono soddisfatte:

1. per ogni t esiste un solo u tale che $S(u, t) = 1$ ovvero, in ogni istante la macchina si può trovare in un solo stato;

2. per ogni t esiste un solo i tale che $L(i, t) = 1$ ovvero, in ogni istante la macchina legge esattamente una cella;

3. per ogni t e ogni i esiste un solo j tale che $C(i, j, t) = 1$ ovvero, in ogni istante ciascuna cella contiene esattamente un simbolo;

4. i valori delle variabili che hanno indice $t = 0$ rappresentano la configurazione iniziale su input w;

5. esiste una variabile $S(u, p(n))$, che assume valore 1, tale che $q_u \in F$, ovvero M raggiunge uno stato finale al termine della computazione;

6. per ogni t e ogni i, se $L(i,t) = 0$, allora le variabili $C(i,j,t)$ e $C(i,j,t+1)$ assumono lo stesso valore. In altre parole il contenuto delle celle che non sono lette dalla macchina in un dato istante, resta invariato all'istante successivo;

7. se invece $L(i,t) = 1$, allora il valore delle variabili $C(i,j,t+1)$, $S(u,t+1)$ e $L(i,t+1)$ rispettano le mosse della macchina all'istante t.

Associamo ora a ciascuna condizione una formula booleana, definita sulle variabili date, in modo tale che ogni assegnamento soddisfi la condizione se e solo se rende vera la formula associata. A tale scopo utilizziamo l'espressione $U(y_1, y_2, \ldots, y_k)$ definita in (13.2) che assume valore 1 se e solo se una sola delle sue variabili ha valore 1. La sua lunghezza è limitata da un polinomio nel numero delle variabili (in particolare $|U(y_1, y_2, \ldots, y_k)| = O(k^2)$).

1. La prima condizione richiede che per ogni t vi sia un solo u tale che $S(u,t) = 1$. Possiamo allora scrivere la seguente formula

$$A = \prod_{t=0}^{p(n)} U(S(1,t), S(2,t), \ldots, S(s,t))$$

   la cui lunghezza è chiaramente $O(np(n))$, perché s è una costante che dipende solo da M e non dalla dimensione dell'input. È chiaro che un assegnamento rende vera la formula A se e solo se soddisfa la condizione 1.

   Le altre formule si ottengono in modo simile e hanno tutte lunghezza polinomiale in n.

2. $B = \displaystyle\prod_{t=0}^{p(n)} U(L(1,t), L(2,t), \ldots, L(p(n)+1,t))$

3. $C = \displaystyle\prod_{t,i} U(C(i,1,t), C(i,2,t), \ldots, C(i,r,t))$

4. Supponendo che $w = x_1 x_2 \cdots x_n$ e rappresentando impropriamente l'indice di ogni x_i,

$$D = S(1,0) \cdot L(1,0) \cdot \prod_{i=1}^{n} C(i, x_i, 0) \cdot \prod_{i=n+1}^{p(n)+1} C(i, B, 0)$$

5. $E = \displaystyle\sum_{q_u \in F} S(u, p(n))$

6. Per rappresentare la sesta condizione denotiamo con $x \equiv y$ l'espressione $(x + \overline{y}) \cdot (\overline{x} + y)$, che vale 1 se e solo se le variabili x e y assumono lo stesso valore. Per ogni i e ogni t $(t \neq p(n))$, poniamo

$$F_{it} = L(i,t) + \prod_{j=1}^{r} (C(i,j,t) \equiv C(i,j,t+1)).$$

   Quindi definiamo

$$F = \prod_{t=0}^{p(n)-1} \prod_{i=1}^{p(n)+1} F_{it}$$

7. Per ogni u, t, i, j definiamo

$$G_{utij} = \overline{S(u,t)} + \overline{L(i,t)} + \overline{C(i,j,t)} + \\ + \sum_{(q_{u'}, a_{j'}, v) \in \delta(q_u, a_j)} (S(u', t+1) \cdot C(i, j', t+1) \cdot L(i+v, t+1)).$$

Osserviamo che questa formula non è in forma normale congiunta. Tuttavia, per quanto dimostrato nella sezione 13.5, sappiamo che esiste una formula equivalente in FNC che denoteremo con $\tilde{G}_{utij}$. Si può verificare che la lunghezza di $\tilde{G}_{utij}$ è polinomiale. Ne segue che la formula associata alla settima condizione è

$$G = \prod_{u,t,i,j} \tilde{G}_{utij}$$

e anche la sua lunghezza risulta polinomiale in n.

Poiché l'espressione U è in forma normale congiunta, tutte le formule precedenti sono in FNC. Di conseguenza, anche la formula ϕ, ottenuta dal prodotto booleano delle formule precedenti, è in FNC:

$$\phi = A \cdot B \cdot C \cdot D \cdot E \cdot F \cdot G.$$

Tale formula seleziona esattamente gli assegnamenti che rappresentano computazioni accettanti di M su input w. Possiamo allora affermare che esiste un assegnamento che rende vera la formula ϕ se e solo la macchina M accetta l'input w.

È inoltre facile verificare che, per una M fissata, ϕ può essere costruita in tempo polinomiale a partire dall'input w.

Bibliografia

[1] A.V. Aho, J.E. Hopcroft, J.D. Ullman, *The design and analysis of computer algorithms*, Addison-Wesley, 1974.

[2] A.V. Aho, J.E. Hopcroft, J.D. Ullman, *Data structures and algorithms*, Addison-Wesley, 1983.

[3] A. Bertossi, *Algoritmi e strutture di dati*, UTET Libreria, 2000.

[4] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, *Introduzione agli algoritmi e strutture dati*, Seconda edizione, McGraw-Hill, 2005.

[5] C. Demetrescu, I. Finocchi, G.F. Italiano, *Algoritmi e strutture dati*, McGraw-Hill, 2004.

[6] P. Flajolet, R. Sedgewick, *An introduction to the analysis of algorithms*, Addison-Wesley, 1996.

[7] C. Froidevaux, M.-C. Gaudel, M. Soria, *Types de données et algorithmes*, Ediscience International, 1993.

[8] M.R. Garey, D.S. Johnson, *Computers and intractability: a guide to the theory of NP-completeness*, W.H. Freeman, 1979.

[9] R.L. Graham, D.E. Knuth, O. Patashnik, *Concrete mathematics*, Addison-Wesley, 1989.

[10] J.E. Hopcroft, J.D. Ullman, *Introduction to automata theory, languages and computation*, Addison-Wesley, 1979.

[11] E. Horowitz, S. Sahni, *Fundamentals of computer algorithms*, Computer Science Press, 1978.

[12] D.E. Knuth, *The art of computer programming (volume 1): fundamental algorithms*, Addison-Wesley, 1973.

[13] D.E. Knuth, *The art of computer programming (volume 2): seminumerical algorithms*, Addison-Wesley, 1973.

[14] D.E. Knuth, *The art of computer programming (volume 3): sorting and searching*, Addison-Wesley, 1973.

[15] K. Mehlhorn, *Data structures and algorithms (volume 1): sorting and searching*, Springer-Verlag, 1984.

[16] K. Mehlhorn, *Data structures and algorithms (volume 2): graph algorithms and NP-completeness*, Springer-Verlag, 1984.

[17] R. Neapolitan, K. Naimipour, *Foundations of algorithms*, D.C. Heath and Company, 1996.

[18] R.H. Papadimitriou, *Computational complexity*, Addison-Wesley, 1994.

[19] R. Sedgewick, *Algorithms*, Addison-Wesley Publishing Company, 1988.