Appunti di
Analisi e Progettazione di Algoritmi

Vincenzo Acciaro Teresa Roselli Vittorio Marengo

Indice

Presentazione

Ho seguito con attenzione, con cura e con affetto, dalle idee generali sino a quasi tutti i dettagli, la stesura di questo libro perche' scritto da tre dei miei allievi piu' cari.

Con enorme soddisfazione vedo ora realizzata questa opera nella quale ritrovo numerosi contenuti delle lezioni dei corsi di Teoria ed Applicazione delle Macchine Calcolatrici, Programmazione ed Algoritmi e Strutture Dati delle Lauree in Scienze dell'Informazione e in Informatica dell'Università di Bari.

Sono sicuro che questo volume avrà un gran numero di lettori e risulterà utile per gli studenti dei nuovi corsi di laurea della classe di Scienze e Tecnologie Informatiche.

V.L. Plantamura

Capitolo 1

Prefazione

Questo libro intende esaminare le generalità degli algoritmi e delle strutture dati fondamentali. Puo' essere usato come libro di testo nei corsi di Algoritmi e Strutture Dati e Programmazione per il Corso di Laurea in Informatica e nel corso di Fondamenti di Informatica per il Corso di Laurea in Ingegneria.

Struttura del volume

La successione degli argomenti segue un percorso già collaudato in diversi corsi in ambito universitario (Algoritmi e Strutture Dati I e II) condotti dagli autori. Particolare attenzione e' stata riservata alla valutazione della complessità computazionale degli algoritmi riportati nel testo al fine di ottenere una maggiore comprensione da parte dello studente. Il libro e' costituito da 12 capitoli dedicati alle seguenti aree: Macchine astratte (o modelli di calcolo), Nozioni di complessità, Algoritmi ricorsivi, Tipi di dati astratti fondamentali (liste, pile, code, grafi ed alberi).

Capitolo 1 Il capitolo introduttivo presenta gli aspetti generali della materia con alcune interessanti note storiche.

Capitolo 2 Nel secondo capitolo vengono presentati due fondamentali modelli di calcolo: la Macchina di Turing e la Random Access Machine.

Capitolo 3 Il terzo capitolo esamina da un punto di vista formale la complessità computazionale di un algoritmo, introducendo il concetto di risorse di calcolo, definendo le delimitazione asintotiche e le notazioni corrispondenti, la complessità degli algoritmi e quella dei problemi.

Capitolo 4 Il quarto capitolo presenta uno pseudo-linguaggio per la descrizione di algoritmi ed alcune regole base per il calcolo della complessità di algoritmi codificati utilizzando tale pseudo-linguaggio.

Capitolo 5 Il quinto capitolo introduce il concetto di ricorsione nell'ambito degli algoritmi e dei linguaggi di programmazione.

Capitolo 6 Il sesto capitolo illustra l'approccio Divide et Impera alla progettazione di algoritmi efficienti. Vengono forniti numerosi esempi per introdurre tale approccio.

Capitolo 7 Nel settimo capitolo vengono esaminati i principali metodi per l'analisi di algoritmi ricorsivi, le equazioni di ricorrenza e le rispettive tecniche di soluzione.

Capitolo 8 Nell'ottavo capitolo viene presentato l'approccio della programmazione dinamica alla progettazione di algoritmi efficienti.

Capitolo 9 Il nono capitolo presenta la struttura dati heap, ed i possibili campi di applicazione delle heaps. Viene, inoltre, esaminato brevemente l'heap-sort, un algoritmo di ordinamento ottimale nel caso peggiore.

Capitolo 10 Nel decimo capitolo vengono esaminate le tecniche hash per implementare efficientemente dizionari, la loro implementazione pratica e l'analisi di complessità.

Capitolo 11 L'undicesimo capitolo illustra il Bucket-Sort, un algoritmo di ordinamento ottimale nel caso medio che combina l'efficienza delle tecniche hash con l'efficienza del Merge-Sort. Vengono esaminati brevemente i casi medio e pessimo per quanto riguarda la complessità computazionale. Il problema dell'ordinamento viene ulteriormente approfondito utilizzando alberi decisionali per dimostrare l'esistenza di una delimitazione inferiore alla complessità del problema ordinamento utilizzando il modello basato su confronti e scambi.

Capitolo 12 Nel dodicesimo capitolo viene affrontato il problema della selezione in tempo lineare e viene descritto un algoritmo ottimale per la soluzione di tale problema.

Appendice A In questa appendice vengono brevemente trattati i principali tipi di dati astratti fondamentali: liste, pile e code. Per ognuno di essi vengono

fornite diverse tecniche di implementazione con i relativi vantaggi e svantaggi. È inclusa una breve discussione sui grafi e sugli alberi, e su alcune delle possibili implementazioni.

Appendice B In questa appendice viene illustrato il modello multi-nastro off-line della macchina di Turing.

Non sono necessarie conoscenze pregresse della materia, tuttavia, per poter comprendere appieno gli argomenti presentati è conveniente aver acquisito le nozioni base della matematica discreta (teoria degli insiemi, relazioni e funzioni, vettori e matrici) e dell'analisi matematica (successioni e serie).

Il libro contiene un buon numero di esempi e di esercizi.

Ringraziamenti

Numerose persone hanno fornito il loro contributo alla realizzazione del testo. In particolare, ringraziamo il dott. Marcello Di Leonardo, il dott. Emanuele Covino ed il dott. Vittorio Saponaro per l'accurato lavoro di revisione svolto e per gli utilissimi consigli dati.

Capitolo 2

Introduzione

In prima istanza l'attività di programmazione può essere distinta in due aree:

1 programmazione in grande;

2 programmazione in piccolo.

La programmazione in grande si occupa della soluzione informatica di *problemi di grandi dimensioni*, mentre la programmazione in piccolo si preoccupa di trovare una *buona soluzione algoritmica* a specifici problemi ben formalizzati.

Il questo volume forniremo una introduzione alle nozioni di base ed ai metodi della *programmazione in piccolo*. In particolare studieremo:

- Come organizzare e rappresentare l'informazione nelle *strutture dati* al fine di ottenere una sua efficiente manipolazione tramite *gli algoritmi*;
- Come valutare la bontà di un algoritmo.

La potenza di calcolo che si trova su un personal computer di oggi, appena qualche anno fa era disponibile solo sui mainframes (grossi calcolatori). Disponendo di calcolatori più potenti, si cerca di risolvere problemi più complessi, che a loro volta domandano maggiore potenza di calcolo... entrando cosi' in un giro vizioso.

L'applicazione degli strumenti informatici a svariati settori ha contribuito al progresso economico, scientifico e tecnologico, ecc., ma d'altro canto, ha creato tanti miti, uno dei quali (da sfatare) è il seguente:

> *Se un problema è ben posto (formalizzato) allora è possibile risolverlo se si disponga della adeguata potenza di calcolo.*

Ovvero, se il calcolatore a nostra disposizione non è sufficientemente potente per risolvere un problema assegnato in tempo ragionevole, possiamo sempre risolvere il problema utilizzando un calcolatore provvisto di risorse maggiori.

Nel linguaggio comune, la parola risolvere già implica in tempo ragionevole. Intuitivamente, un intervallo di tempo è ragionevole se è tale quando lo si rapporti alla durata media della vita dell'uomo (1 minuto è ragionevole, 20 anni no!).

Il seguente argomento, dovuto a A.R. Meyer e L.J. Stockmeyer, di natura puramente fisica, mostra efficacemente cosa si intenda per limiti fisici del calcolabile:

> *Il più potente calcolatore che si possa costruire non potrà mai essere piú grande dell'universo (meno di 100 miliardi di anni luce di diametro), nè potrà essere costituito da elementi più piccoli di un protone (*10^{-13} *cm di diametro), nè potrà trasmettere informazioni ad una velocità superiore a quella della luce (300000 km al secondo).*
> *Un tale calcolatore non potrebbe avere più di* 10^{126} *componenti.*
> *Esso impiegherebbe almeno 20 miliardi di anni per risolvere dei problemi la cui risolubilità é nota in linea di principio. Poiché presumibilmente l'universo non ha una età superiore ai 20 miliardi di anni, ciò sembra sufficiente per affermare che questi problemi sfidano l'analisi del calcolatore.*

Alcuni obiettivi che ci poniamo:

1 Studio delle proprietà dei problemi computazionali, delle caratteristiche che ne rendano facile o difficile la risoluzione automatica.

2 Studio delle tecniche basilari di analisi e progettazione di algoritmi.

3 Definizione e studio dei problemi computazionalmente intrattabili, ovvero di quei problemi per cui si ritiene che non possa esistere alcun algoritmo che li risolva in tempo ragionevole.

2.1 Algoritmi e problemi

Un algoritmo è un metodo di risoluzione di un problema, che presuppone l'esistenza di:

1 un *committente*, ovvero colui che pone il problema e desidera conoscerne la soluzione;

2 un *esecutore*, in grado di svolgere determinati compiti elementari.

- All'inizio dell'esecuzione, l'esecutore riceve dal committente un insieme di *dati* che rappresentano la descrizione della particolare istanza del problema che si intende risolvere.
- I dati appartengono ad un *insieme finito di simboli*.
- Deve essere definito un meccanismo con il quale l'esecutore comunica al committente la *soluzione* del problema (il risultato) al termine dell'esecuzione.
- L'algoritmo è costituito da una sequenza di istruzioni che indicano in modo non ambiguo le azioni che l'esecutore deve compiere.
- L'esecuzione avviene per passi discreti e termina dopo un numero di passi *finito*, che è funzione della particolare istanza del problema.

Esempio 1 Un possibile problema è costituito dall'addizione di due numeri interi. Chiameremo questo problema ADDIZIONE.
Una istanza del problema ADDIZIONE è la seguente:

Somma i numeri interi 45 e 6

La *descrizione* di tale istanza è (i dati di input):

I numeri interi 45 e 6

Il *risultato* (output) sarà costituito da:

Il numero intero 51

Esempio 2 Un possibile problema è costituito dall'ordinamento di un certa sequenza di numeri interi. Chiameremo questo problema ORDINAMENTO.
Una istanza del problema ORDINAMENTO è la seguente:

Ordina la sequenza 3,8,6,0,2

La *descrizione* di tale istanza è (i dati di input):

La sequenza 3,8,6,0,2

Il *risultato* (output) sarà costituito da:

La sequenza 0,2,3,6,8

Il precedente esempio ORDINAMENTO è stato presentato in modo informale.

Una descrizione informale del problema che si intende risolvere può essere molto utile, in un primo momento, ad esempio per chiarirsi le idee.

Molto spesso sappiamo che un certo problema esiste, tuttavia non siamo in grado di formalizzarlo esattamente.

Una descrizione informale del problema può essere sufficiente nel caso in cui l'esecutore sia un essere dotato di capacità cognitiva simile alla nostra, e che condivida con noi:

1 l'esperienza (ad esempio possiamo dire ad un nostro collaboratore - sbrigami la pratica del 3 dicembre, ed il nostro collaboratore saprà esattamente cosa fare) oppure

2 il senso comune (ad esempio uno sconosciuto: mi scusi, che ora è?).

In attesa di costruire degli oggetti artificiali dotati della nostra esperienza o del nostro senso comune (questo è uno degli obiettivi della Intelligenza Artificiale), occorre specificare esattamente, in una prima fase, la natura del problema da risolvere. Occorre far attenzione a non confondere la natura del problema da risolvere con la descrizione del metodo di risoluzione del problema.

Esempio 3 Il problema Ordinamento può essere definito (specificato) nel modo seguente:

- **Dati:** Una sequenza di n chiavi $a_1, \ldots, a_n$ che è possibile confrontare usando l'operatore $\leq$. Le chiavi sono numeri interi positivi più piccoli di un limite prefissato (ad esempio 1000).
- **Risultato:** Una sequenza $b_1, \ldots, b_n$ di chiavi che soddisfa la relazione $b_i \leq b_{i+1}$ per ogni $i = 1, 2, ..., n-1$.

- **Relazione tra Input e Output:** La sequenza $b_1, \ldots, b_n$ è una permutazione della sequenza $a_1, \ldots, a_n$.

La Matematica, ed in particolare la Logica Matematica, forniscono degli strumenti eccellenti di formalizzazione, poichè ci permette di stabilire in modo non ambiguo:

- le relazioni che legano tra loro i dati di input (esempio: nel caso dell'Ordinamento, tutti i dati di input appartengono ad uno stesso insieme, l'insieme delle chiavi);
- le relazioni che legano l'output all'input (i dati di output come funzione dei dati di input);
- le relazioni che dovranno legare tra loro i dati di output (esempio: nel caso dell'Ordinamento, la relazione $b_i \leq b_{i+1}$ per ogni $i = 1, 2, \ldots, n-1$).

2.1.1 Esempio di algoritmo

Un algoritmo elementare è quello dell'addizione di due interi. Per addizionare due numeri interi, composti da più cifre, occorre eseguire una sequenza di passi (azioni) elementari, ognuno dei quali coinvolge solo due cifre, una delle quali può essere una barretta che denota il riporto. Tali azioni sono di due tipi:

1 scrivere la somma di due cifre corrispondenti (che si trovano cioè nella stessa colonna);

2 segnare il riporto di una unità a sinistra.

Nota bene 1 *Si noti che*

- *viene specificato l'ordine appropriato con cui le operazioni devono essere eseguite (non ambiguità);*
- *le operazioni elementari sono puramente formali, ovvero possono essere eseguite automaticamente facendo uso di una tabellina per l'addizione.*

Quindi, è possibile delegare ad un esecutore automatico (che sia in grado di svolgere, nel corretto ordine, le operazioni indicate sopra) l'esecuzione dell'algoritmo di addizione, senza che l'esecutore debba conoscere gli aspetti più profondi della aritmetica.

2.1.2 L'algoritmo come funzione

Un algoritmo definisce implicitamente una funzione dall'insieme dei dati di ingresso all'insieme dei dati in uscita, ed al tempo stesso indica un procedimento effettivo che permette di determinare, per ogni possibile configurazione in ingresso, i corrispondenti valori in uscita.

Dato un algoritmo A, indicheremo con f_A la funzione che associa ad ogni valore in ingresso il corrispondente valore in uscita $f_A(x)$.

2.1.3 Nota storica

> *La parola algoritmo ha origine nel Medio Oriente. Essa proviene dall'ultima parte del nome dello studioso persiano Abu Jàfar Mohammed Ibn Musa Al-Khowarizmi, il cui testo di aritmetica (825 d.C. circa) esercitò una profonda influenza nei secoli successivi.*

Le seguenti osservazioni sono dovute a D.E. Knuth, uno dei padri fondatori della computer science

> *Tradizionalmente gli algoritmi erano impiegati esclusivamente per risolvere problemi numerici. Tuttavia, l'esperienza con i calcolatori ha dimostrato che i dati possono virtualmente rappresentare qualunque cosa.*
>
> *Di conseguenza, l'attenzione della scienza dei calcolatori si è trasferita allo studio delle diverse strutture con cui si possono rappresentare le informazioni, e all'aspetto ramificato o decisionale degli algoritmi, che permette di eseguire differenti sequenze di operazioni in dipendenza dello stato delle cose in un particolare istante.*
>
> *È questa la caratteristica che rende talvolta preferibili, per la rappresentazione e l'organizzazione delle informazioni, i modelli algoritmici a quelli matematici tradizionali.*

Definizione 1 *Un problema è una funzione $P : D_I \to D_S$ definita su un insieme D_I di elementi che chiamiamo istanze, ed a valori su un insieme D_S di soluzioni.*

Diciamo che un algoritmo A risolve un problema P se $P(x) = f_A(x)$, per ogni istanza x.

2.2 Programma

Le seguenti osservazioni sono dovute sempre a D.E. Knuth:

> *Un programma è l'esposizione di un algoritmo in un linguaggio accuratamente definito. Quindi, il programma di un calcolatore rappresenta un algoritmo, per quanto l'algoritmo stesso sia un costrutto intellettuale che esiste indipendentemente da qualsiasi rappresentazione. Allo stesso modo, il concetto di numero 2 esiste nella nostra mente anche quando non sia espresso graficamente.*

Programmi per risolvere problemi numerici sono stati scritti sin dal 1800 a.C., quando i matematici babilonesi del tempo di Hammurabi stabilirono delle regole per la risoluzione di alcuni tipi di operazioni.

Le regole erano determinate come procedure passo-passo, applicate sistematicamente ad esempi numerici particolari.

2.3 Risorse di calcolo

Ad ogni programma di calcolo sono associati dei costi.

- L'Ingegneria del Software si occupa, tra l'altro, di minimizzare i costi relativi allo sviluppo (analisi del problema, progettazione, implementazione) dei programmi ed alla loro successiva manutenzione;
- La Teoria della Complessità Computazionale si occupa, tra l'altro, di minimizzare i costi relativi alla esecuzione dei programmi.

Le due aree difficilmente si conciliano tra loro: non solo gli obiettivi sono diversi, ma anche i metodi utilizzati.

Definizione 2 *Il costo relativo all'esecuzione di un programma viene definito come la quantità di risorse di calcolo che il programma utilizza durante l'esecuzione.*

Le risorse di calcolo a disposizione del programma sono:

1 Il Tempo utilizzato per eseguire l'algoritmo;

2 Lo Spazio di lavoro utilizzato per memorizzare i risultati intermedi.

3 Il Numero degli esecutori, se più esecutori collaborano per risolvere lo stesso problema.

Tale classificazione delle risorse è totalmente indipendentemente dalla sua interpretazione informatica. Qualora ci si riferisca al campo specifico dell'informatica:

- lo spazio di lavoro diventa la memoria del calcolatore;
- il numero degli esecutori diventa il numero dei processori a nostra disposizione, in un sistema multi-processore.

Definizione 3 *Un algoritmo è* efficiente *se fa un uso contenuto (ovvero parsimonioso) delle risorse a sua disposizione.*

È molto importante saper valutare la quantità di risorse consumate, poichè un consumo eccessivo di risorse può pregiudicare la possibilità di utilizzo di un algoritmo.

Per valutare correttamente il consumo di risorse di un algoritmo è necessario fissare a priori un **modello di calcolo**, e definire in base a questo:

- la nozione di algoritmo;
- la nozione di risorse consumate.

2.3.1 Modelli di calcolo

Definizione 4 *Un modello di calcolo è semplicemente una astrazione di un esecutore reale, in cui si omettono dettagli irrilevanti allo studio di un algoritmo per risolvere un problema.*

Esistono tanti differenti modelli di calcolo (Macchina di Turing, RAM, ecc.). L'adozione di un particolare modello di calcolo dipende da vari fattori:

1 *capacità espressiva del modello in relazione al problema assegnato;*
in altre parole, un modello è preferibile ad un altro per esprimere la soluzione algoritmica di un determinato problema;

2 *livello di astrazione;*
è tanto maggiore quanto maggiore è la quantità di dettagli che vengono omessi;

3 *generalità;*
esistono modelli più generali di altri (un primo modello è più generale di un secondo se tutti i problemi risolubili con il secondo sono risolubili con il primo).

Uno dei risultati più sorprendenti della teoria della computabilità riguarda *l'esistenza di modelli di calcolo assolutamente generali.* Uno di questi è la macchina di Turing (capitolo 2).

2.3.2 Irrisolubilità

Definizione 5 *Un problema è non risolubile algoritmicamente se nessun procedimento di calcolo è in grado di fornire la soluzione in tempo finito.*

Un risultato piuttosto sconcertante riguarda l'esistenza di problemi non risolubili algoritmicamente.

Esempio 4 Un noto problema non risolubile algoritmicamente è il problema dell'ALT della macchina di Turing [19].

La logica matematica si occupa (tra l'altro) dei limiti della computabilità, ovvero dello studio e della classificazione dei problemi non risolubili algoritmicamente.

2.3.3 Intrattabilità

Definizione 6 *Un problema è intrattabile se qualunque algoritmo che lo risolva richieda una quantità molto elevata di risorse.*

La logica matematica fornisce alla teoria degli algoritmi gli strumenti per riconoscere e classificare i problemi intrattabili.

Problemi intrattabili sorgono ad esempio in giochi quali la dama o gli scacchi.

Capitolo 3

Modelli di calcolo

In questo capitolo introduciamo due modelli di calcolo, la macchina di Turing [33] e la Random Access machine [13]. Malgrado la loro semplicità, entrambe sono un modello delle capacità computazionali di un computer; formulazioni alternative possono essere trovate in [2], [6], [7], [21], [22].

3.1 La macchina di Turing

In questa sezione sarà esaminato il modello di calcolo denominato *Macchina di Turing*.

Questo è un eccellente strumento didattico, poichè ci consente di definire esattamente la nozione di algoritmo, ma sopratutto ci consente di definire in modo semplice ed inequivocabile la nozione di risorse utilizzate da un algoritmo (spazio e tempo).

In virtù dell'esistenza di un modello di calcolo assolutamente generale, la nozione di irrisolubilità già introdotta nel capitolo precedente assumerà un significato formale.

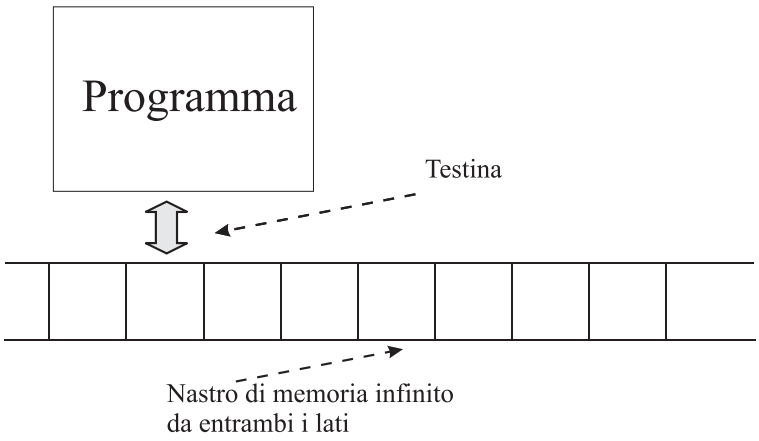
Al fine di agevolare la comprensione dell'argomento, saranno presentate alcune Macchine di Turing molto semplici.

3.1.1 Definizione di Macchina di Turing

Una Macchina di Turing consiste di:

- Un nastro infinito, suddiviso in celle. Ogni cella può contenere un solo simbolo, tratto da un insieme finito S detto alfabeto esterno.

- Una testina capace di leggere un simbolo da una cella, scrivere un simbolo in una cella, e muoversi di una posizione sul nastro, in entrambe le direzioni.
- Un insieme finito Q di stati, tali che la macchina si trova esattamente in uno di essi in ciascun istante.
- Un programma, che specifica esattamente cosa fare per risolvere un problema specifico.



L'alfabeto esterno della M.d.T.

L'alfabeto esterno $S = \{s_1, \ldots, s_k\}$ è utilizzato per codificare l'informazione in input e quella che la MdT produce nel corso della computazione.

Assumiamo che S contenga un simbolo speciale detto lettera vuota o blank, indicato con b. Diciamo che una cella è vuota se contiene b. Scrivendo la lettera vuota b in una cella viene cancellato il contenuto di quella cella.

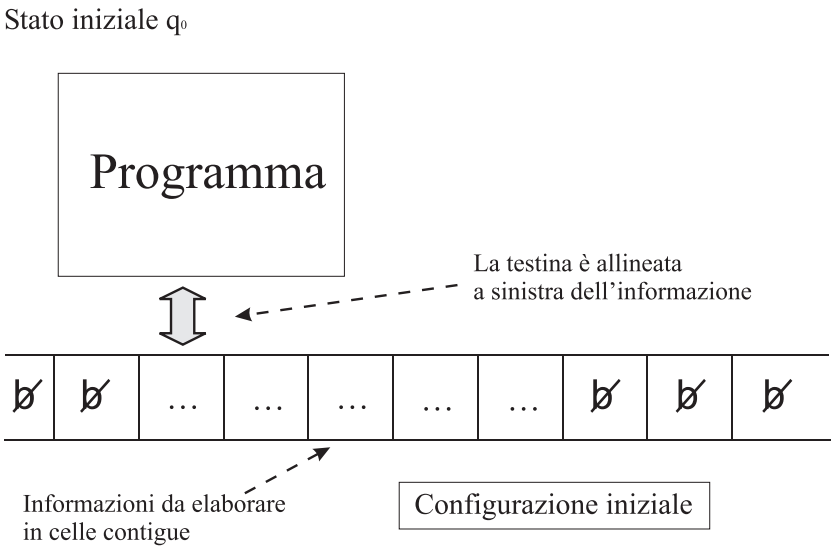
Gli stati della M.d.T.

I possibili stati di una macchina di Turing sono denotati $q_1, q_2, \ldots, q_n, q_0, q_f$.

- Gli stati $q_1, \ldots, q_n$ sono detti stati ordinari.
- Lo stato q_0 è detto stato iniziale.
- Lo stato q_f è detto stato finale.

Configurazione iniziale della MdT.

La macchina di Turing si trova inizialmente nello stato q_0. L'informazione da elaborare è contenuta in celle contigue del nastro, ed è codificata utilizzando i simboli dell'alfabeto esterno S. Tutte le altre celle del nastro contengono inizialmente la lettera vuota. La testina di lettura/scrittura è posizionata in corrispondenza del primo simbolo valido (quello che si trova più a sinistra).



Il programma nella M.d.T.

Indichiamo con q lo stato in cui la MdT si trova ad un certo istante, e con s il simbolo che si trova sul nastro in corrispondenza della testina.

Per ogni possibile coppia $(q, s) \in Q \times S$ il programma dovrà specificare:

- in quale nuovo stato q la MdT dovrà portarsi;
- il simbolo s da scrivere sul nastro nella posizione corrente;
- se la testina di lettura debba rimanere ferma, spostarsi di una posizione a sinistra, o spostarsi di una posizione a destra.

Il Programma come funzione

Sia $T := \{ferma, sinistra, destra\}$. Possiamo vedere il programma eseguito da una MdT come una funzione

$$f : Q \times S \to Q \times S \times T$$

È possibile specificare un programma utilizzando una matrice, detta matrice funzionale o matrice di transizione, le cui righe sono indicizzate utilizzando l'alfabeto esterno, e le cui colonne sono indicizzate utilizzando l'insieme degli stati.

Il generico elemento della matrice di indice (q_i, s_j) conterrà $f(q_i, s_j)$.

Terminazione della computazione.

La computazione termina non appena la macchina raggiunge lo stato q_f. Al termine della computazione, sul nastro sarà presente il risultato della computazione.

3.1.2 Ipotesi fondamentale della teoria degli algoritmi

> **(Tesi di Church)** *Qualunque algoritmo può essere espresso sotto forma di matrice funzionale ed eseguito dalla corrispondente Macchina di Turing.*

Irrisolubilità

Alla luce della Tesi di Church, possiamo riformulare la nozione di irrisolubilità data in precedenza come segue:

> *Un problema è non risolubile algoritmicamente se nessuna Macchina di Turing è in grado di fornire la soluzione al problema in tempo finito.*

Abbiamo visto in precedenza che esistono problemi irrisolubili algoritmicamente, e che la logica matematica si occupa (tra l'altro) dei limiti della computabilità, ovvero dello studio e della classificazione di tali problemi.

3.1.3 Esempi

1) **Controllo di paritá.**

Siano: $S = \{1, b, \underbrace{P}_{Pari}, \underbrace{D}_{Dispari}\}$ $\quad Q = \{q_0, q_1, q_f\}$.

Inizialmente il nastro conterrá una stringa di 1. Definiamo una MdT in grado di determinare se la stringa assegnata contiene un numero pari o dispari di 1. Ad esempio, data la configurazione iniziale:

ø	ø	ø	1	1	1	ø	ø	ø	ø

la MdT dovrá scrivere D sul nastro, ovvero al termine della computazione il nastro dovrá contenere:

ø	ø	ø	1	1	1	D	ø	ø	ø

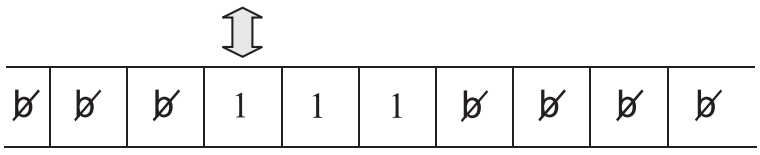
Qui di seguito é descritta una MdT adatta al nostro scopo:

$$\begin{array}{rcl} (q_0, 1) & \rightarrow & (q_1, 1, DESTRA) \\ (q_1, b) & \rightarrow & (q_f, D, FERMO) \\ (q_1, 1) & \rightarrow & (q_0, 1, DESTRA) \\ (q_0, b) & \rightarrow & (q_f, P, FERMO) \end{array}$$

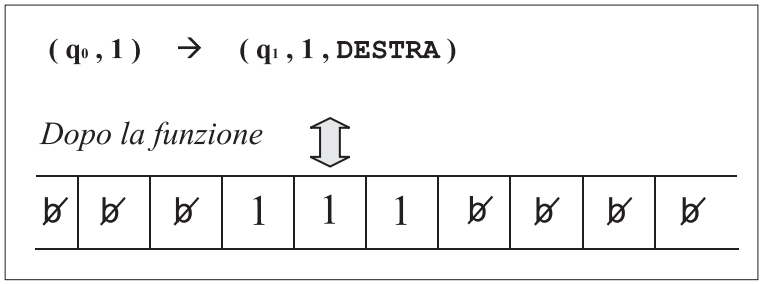
La matrice di transizione corrispondente è:

	q_0	q_1	q_f
1	$(q_1, 1, DESTRA)$	$(q_0, 1, DESTRA)$	
b	$(q_f, P, FERMO)$	$(q_f, D, FERMO)$	
P			
D			

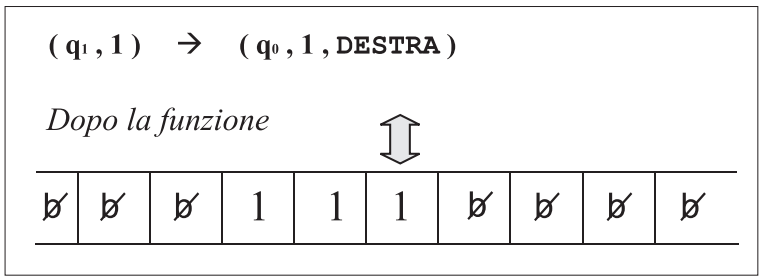
Riportiamo la traccia dell'esecuzione a partire dalla configurazione iniziale:



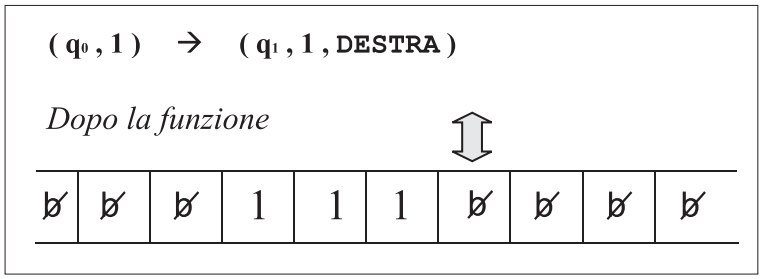
Primo Passo:



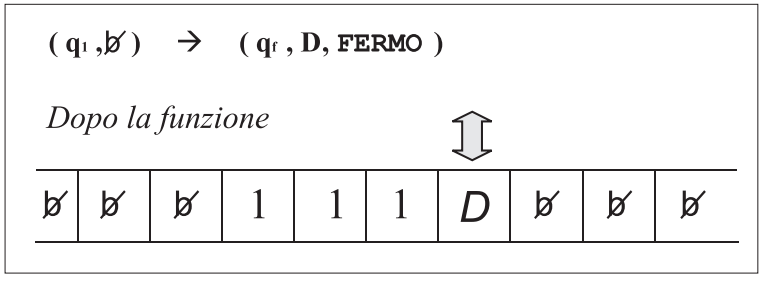
Secondo Passo:



Terzo Passo:



Quarto Passo:



2) **Addizione di due numeri espressi in notazione unaria.**

Siano $S = \{1, b, +\}$ $Q = \{q_0, q_1, q_2, q_f\}$.

Rappresentiamo l'addizione dei numeri 3 e 4 espressi in notazione unaria come segue:

	⊬	1	1	1	+	1	1	1	1	⊬	⊬	

Voglio ottenere la seguente configurazione sul nastro:

	⊬	1	1	1	1	1	1	1	⊬	⊬	⊬	

Qui di seguito é descritta una Macchina di Turing adatta al nostro scopo:

$$\begin{array}{lcl} (q_0, 1) & \rightarrow & (q_0, 1, DESTRA) \\ (q_0, +) & \rightarrow & (q_1, 1, DESTRA) \\ (q_1, 1) & \rightarrow & (q_1, 1, DESTRA) \\ (q_1, b) & \rightarrow & (q_2, b, SINISTRA) \\ (q_2, 1) & \rightarrow & (q_f, b, FERMO) \end{array}$$

La matrice di transizione corrispondente è:

	q_0	q_1	q_2	q_f
1	$(q_0, 1, DESTRA)$	$(q_1, 1, DESTRA)$	$(q_f, b, FERMO)$	
+	$((q_1, 1, DESTRA)$			
b		$(q_2, b, SINISTRA)$		

3.1.4 Esercizi

1 Costruire una MdT capace di riconoscere se una stringa assegnata è palindroma. Una stringa si dice *palindroma* se è uguale quando viene letta da sinistra verso destra o da destra verso sinistra. Ad esempio, le stringhe anna o madam I'm adam sono palindrome.

2 Costruire una MdT capace di riflettere (capovolgere) una stringa data in input.

3.2 La Random Access Machine (RAM)

3.2.1 Definizione

Un altro modello di calcolo molto generale è la RAM (acronimo di *Random Access Machine*, macchina ad accesso diretto), introdotta da Cook e Reckhov agli inizi degli anni '70. [13]

Tale modello è utilizzato come strumento di analisi della complessità degli algoritmi piuttosto che come strumento di progettazione. La RAM consta di:

- Un nastro di lunghezza infinita, suddiviso in celle, su cui sono contenuti i dati di input; su tale nastro è consentita la sola operazione di lettura, in modo sequenziale;
- Un nastro di lunghezza infinita, suddiviso in celle, su cui vengono scritti i dati di output; su tale nastro è consentita la sola operazione di scrittura, in modo sequenziale;
- Un programma che viene eseguito sequenzialmente (ovvero una istruzione alla volta);
- Una memoria di dimensione infinita (il nostro spazio di lavoro), su cui conservare i risultati intermedi.

Si assume che:

1 Il programma non possa modificare se stesso;

2 Tutti i calcoli avvengano utilizzando una locazione fissa di memoria detta accumulatore;

3 Ogni locazione di memoria ed ogni cella del nastro di input ed output siano in grado di contenere un simbolo arbitrario (oppure, se si preferisce, un numero intero di dimensione arbitraria);

4 Il programma possa accedere alle singole locazioni di memoria in ordine arbitrario.

Il punto (4) giustifica l'aggettivo diretto. Equivalentemente, possiamo formulare il punto (4) come segue:

- Il tempo di accesso ad una cella di memoria deve essere indipendente dalla cella stessa.

3.2.2 Complessità computazionale di programmi RAM

Esistono due criteri per determinare la quantità di tempo e di spazio richieste durante l'esecuzione di un programma RAM:

1 Il criterio di costo uniforme;

2 Il criterio di costo logaritmico.

Il criterio di costo uniforme

L'esecuzione di ogni istruzione del programma richiede una quantità di tempo costante (indipendente dalla grandezza degli operandi). Lo spazio richiesto per l'utilizzo di un registro di memoria è di una unità, indipendentemente dal suo contenuto.

Il criterio di costo logaritmico

Attribuiamo ad ogni istruzione un costo di esecuzione che dipende dalla dimensione degli operandi. Tale criterio è cosí chiamato perché per rappresentare un numero intero n occorrono $\lfloor \log n \rfloor + 1$ bits.

Il criterio di costo logaritmico ci dà una misura più realistica del tempo e dello spazio consumati da un programma RAM.

Capitolo 4

Nozioni base di complessità

Nei capitoli precedenti sono state introdotte le nozioni di algoritmo, problema, e risorse di calcolo utilizzate da un algoritmo. È stato, inoltre, esaminato un modello di calcolo, la Macchina di Turing, al fine di formalizzare la nozione di algoritmo e fornire al tempo stesso una descrizione delle nozioni di spazio e tempo utilizzati da un programma, come numero di celle utilizzate e transizioni della Macchina.

Abbiamo visto che un uso improprio delle risorse di calcolo può pregiudicare la possibilità di utilizzare praticamente un algoritmo. Si è definito inoltre *efficiente* un algoritmo che fa un uso parsimonioso delle risorse di calcolo a propria disposizione.

In questo capitolo verranno ripresi brevemente tali concetti, e verrà definita, inizialmente in maniera informale, la nozione di complessità di un algoritmo.

4.1 Introduzione

Le problematiche riguardanti lo studio degli algoritmi possono essere suddivise in tre aree:

- *Sintesi (o progetto):*
  dato un problema P, costruire un algoritmo A per risolvere P;
- *Analisi:*
  dato un algoritmo A ed un problema P:

  1 dimostrare che A risolve P (correttezza);

2 valutare la quantità di risorse utilizzate da A (complessità);

- *Classificazione :*
  data una quantità T di risorse individuare la classe dei problemi risolubili da algoritmi che utilizzano al più tale quantità.

4.1.1 Obiettivi in fase di progetto.

In fase di progetto vogliamo un algoritmo che sia:

- facile da capire, codificare e testare;
- efficiente in termini di spazio e tempo.

I due obiettivi sono spesso in antitesi.

Se un programma deve essere utilizzato poche volte, l'obiettivo *facilità di codifica* dovrebbe prevalere sull'obiettivo EFFICIENZA.

Se un programma si utilizza spesso, i costi associati alla sua esecuzione possono eccedere di gran lunga il costo associato alla sua progettazione ed implementazione. In tal caso occorre un programma efficiente.

Se intervengono fattori di sicurezza quali:

- Controllo di processi pericolosi (ad esempio reattori nucleari);
- Sistemi in tempo reale (ad esempio per il controllo di aeromobili o di strumentazione ospedaliera)

allora l'obiettivo efficienza è prioritario.

Può accadere che dati due algoritmi per risolvere lo stesso problema P, il primo faccia un uso più efficiente della risorsa spazio, mentre il secondo privilegi la risorsa tempo. Quale algoritmo scegliere?

> *Tra le due risorse spazio e tempo, il tempo va quasi sempre privilegiato. Infatti, lo spazio è una risorsa riutilizzabile, mentre il tempo non lo è.*

4.2 Il tempo di esecuzione di un programma

Il tempo di esecuzione di un programma dipende, intuitivamente, da fattori quali:

1 La potenza di calcolo;

2 La bontà dell'interprete o la qualità del codice generato dal compilatore;

3 L'input del programma;

4 La complessità temporale dell'algoritmo sottostante.

Denotiamo con $T_A(x)$ e $S_A(x)$ rispettivamente il tempo di calcolo e lo spazio di memoria richiesti da un algoritmo A su un input x.

Se utilizziamo come modello di calcolo la Macchina di Turing, allora:

1 $T_A(x)$:= numero di passi richiesti;

2 $S_A(x)$:= numero di celle utilizzate;

per eseguire l'algoritmo A sull'istanza x.

> **Problema:** Descrivere le funzioni $T_A(x)$ e $S_A(x)$ può essere molto complicato, poichè la variabile x varia arbitrariamente sull'insieme di tutti gli input!
>
> **Soluzione:** Introduciamo la nozione di dimensione di una istanza, raggruppando così tutti gli input che hanno la stessa dimensione.
>
> **Giustificazione Assiomatica:** È ragionevole assumere che problemi più grandi richiedano un maggior tempo di esecuzione.

Questa è una delle tante assunzioni non dimostrabili, che vengono spesso introdotte nel campo dell'informatica, per poter sviluppare una teoria soddisfacente.

La validità di tale assunzione si poggia esclusivamente sul buon senso e sulla nostra esperienza!

La dimensione del problema è definita come la quantità di dati necessaria per descrivere l'istanza particolare del problema assegnato.

Misuriamo il consumo di risorse (tempo e spazio) in funzione della dimensione del problema.

Se indichiamo con x la particolare istanza di un problema P, allora denoteremo con $|x|$ la dimensione di x.

Occorre definire in anticipo come si misura la dimensione della particolare istanza del problema (input). In altre parole, non esiste un criterio universale di misura.

Esempio 5 Nel caso di algoritmi di ordinamento, una misura possibile (ma non l'unica) è data dal numero di elementi da ordinare. Quindi, se $(50, 4, 1, 9, 8)$ rappresenta una lista da ordinare, allora

$$|(50, 4, 1, 9, 8)| = 5.$$

Esempio 6 Nel caso di algoritmi di addizione e moltiplicazione di interi, una misura possibile (ma non l'unica) è data dal numero di cifre decimali necessario ad esprimere la lunghezza degli operandi.

4.3 Complessità temporale

Definizione intuitiva: *La complessità temporale di un algoritmo A su una istanza x del problema P è uguale al numero di passi $T_A(x)$ necessari per eseguire l'algoritmo.*

> **Problema:** Può accadere che date due istanze x ed y di un problema P, aventi la stessa dimensione, risulti $T_A(x) \neq T_A(y)$.

Come definire allora il tempo di calcolo in funzione della sola dimensione?

Prima soluzione: La Complessità nel caso peggiore

La complessità nel caso peggiore è definita come il tempo di esecuzione necessario ad eseguire l'algoritmo A su un'istanza del problema di dimensione n, nel caso pessimo. Si indica con $T_A^p(n)$.

La nozione di caso pessimo dipende dal particolare algoritmo considerato.

Ad esempio, per un determinato algoritmo di ordinamento, il caso pessimo potrebbe verificarsi quando gli elementi da ordinare sono disposti in maniera decrescente.

Tuttavia, per un altro algoritmo di ordinamento, questo caso potrebbe essere abbastanza favorevole!

Seconda soluzione: La Complessità nel caso medio

La complessità nel caso medio è definita come il tempo di esecuzione medio necessario ad eseguire l'algoritmo su un'istanza del problema di dimensione n, assumendo per semplicità che tutte le istanze del problema siano equidistribuite (si presentino con uguale frequenza) per un valore di n prefissato. Si indica con $T_A^m(n)$.

Nel caso di un algoritmo di ordinamento, ciò equivale a supporre che tutte le possibili $n!$ permutazioni degli n dati in ingresso siano equiprobabili.

Nessuna delle due soluzioni è ottimale. Infatti:

- La valutazione nel caso peggiore fornisce spesso una stima troppo pessimistica del tempo di esecuzione di un programma;
- Al contrario, nella valutazione nel caso medio l'ipotesi che le istanze del problema siano equidistribuite non trova spesso riscontro nella realtà.

4.4 Confronto di algoritmi

Di seguito elenchiamo il tempo di esecuzione, misurato in secondi, di 4 programmi, che implementano rispettivamente gli algoritmi A_1 ed A_2, su due diverse architetture (computer1 e computer2)

	computer 1		computer 2	
Dim. input	A_1	A_2	A_1	A_2
50	0.005	0.07	0.05	0.25
100	0.03	0.13	0.18	0.55
200	0.13	0.27	0.73	1.18
300	0.32	0.42	1.65	1.85
400	0.55	0.57	2.93	2.57
500	0.87	0.72	4.60	3.28
1000	3.57	1.50	18.32	7.03

La superiorità del secondo algoritmo non è evidente per input di dimensione piccola! Poichè il tempo di esecuzione effettivo (in secondi) dipende dalla bontà del compilatore o dell'interprete, dalla velocità del calcolatore e dalla abilità del programmatore nel codificare l'algoritmo in un programma, non è possibile esprimere la complessità temporale intrinseca di un algoritmo utilizzando unità di misura quali i secondi, o il numero di passi.

È ragionevole dire piuttosto:

> *il tempo di esecuzione del programma che implementa l'algoritmo è proporzionale ad n^3, per un input di dimensione n, nel caso pessimo.*

Si noti che la costante di proporzionalità non è specificata, poichè dipende da più fattori (calcolatore, compilatore, ecc.).

Aho, Hopcroft ed Ullman (1974) sostennero per primi che ciò che conta, per effettuare una comparazione tra algoritmi diversi per risolvere lo stesso problema, non è il comportamento temporale su input di dimensione piccola, bensì su input la cui dimensione possa crescere arbitrariamente (sia grande a piacere).

Il tasso di crescita o complessità temporale misura appunto il comportamento temporale di un algoritmo quando la dimensione dell'input cresce arbitrariamente.

La tirannia del tasso di crescita

Di seguito mostriamo la dimensione massima di un problema che può essere risolto rispettivamente in 1 secondo, 1 minuto, 1 ora, da 5 algoritmi la cui complessità temporale è specificata nella seconda colonna.

Alg.	complessità temporale	dimensione massima del problema		
		1 sec	1 min	1 ora
A_1	n	1000	60000	3600000
A_2	$n \log n$	140	4893	200000
A_3	n^2	31	244	1897
A_4	n^3	10	39	153
A_5	2^n	9	15	21

Si noti che gli algoritmi dotati di complessità temporale lineare o quasi lineare sono utilizzabili in maniera efficiente anche quando la dimensione dell'input è piuttosto elevata. Algoritmi che hanno una complessità dell'ordine di n^k per $k > 1$ sono applicabili solo quando la dimensione dell'input non è troppo elevata. Infine, gli algoritmi che hanno una complessità esponenziale sono inutilizzabili persino quando la dimensione dell'input è relativamente piccola.

Pertanto, considereremo inefficienti gli algoritmi che hanno una complessità temporale dell'ordine di a^n, per qualche $a > 1$.

Complessità ed evoluzione tecnologica

Cosa succederebbe se utilizzassimo un calcolatore 10 volte più potente?

Alg.	complessità temporale	dimensione 1 sec	massima del 1 min	problema 1 ora
A_1	n	10000	600000	360000000
A_2	$n \log n$	9990	599900	3599900
A_3	n^2	97	750	5700
A_4	n^3	21	80	300
A_5	2^n	12	18	24

Conclusioni

- Gli algoritmi lineari (n) o quasi ($n \log n$) traggono pieno vantaggio dalla evoluzione tecnologica;
- Negli algoritmi polinomiali (n^2) il vantaggio è ancora evidente, sebbene in maniera inferiore;
- Infine, negli algoritmi esponenziali (2^n) i vantaggi che derivano dall'evoluzione della tecnologia sono pressochè irrilevanti.

Esempio di calcolo della complessità (nel caso pessimo).

Ordinamento di un vettore di N interi.

```
For i:=1 to n-1 do
    Cerca il minimo di A[i]...A[n]
    Scambia il minimo e A[i]
```

Per la ricerca del minimo utilizzeremo la seguente procedura:

```
Ricerca del minimo:
1. min := A[i]
2. For k:=i+1 to n do
3.  If a[k]<min
4.  then min := A[k]
```

Convenzioni.

La complessità rappresenterà in questo esempio il numero di operazioni di assegnazione (passi 1 e 4), di operazioni di confronto (passo 3) e di operazioni

di scambio, al più effettuati per un input di dimensione n.

$$\begin{aligned} T_A^p p(n) &= (n+2)+(n+1)+\ldots+4= \\ &= [(n+2)+n+\ldots+4+3+2+1]-6= \\ &= \frac{(n+3)(n+2)}{2}-6= \\ &= \frac{1}{2}n^2+\frac{5}{2}n-3 \end{aligned}$$

Osservazioni

Nell'esempio precedente possiamo asserire che *il tempo di esecuzione è al più proporzionale al quadrato della dimensione dell'input.*

In genere, la costante di proporzionalità non viene specificata poichè dipende da vari fattori (tecnologici): bontà del compilatore, velocità del computer, ecc..

Infatti, quando la dimensione dell'input cresce arbitrariamente (tende all'infinito), le costanti di proporzionalità non sono prese in considerazione.

Esempio 7 Supponiamo che:

$$\begin{aligned} T_1(n) &= 2^n \\ T_2(n) &= \frac{1}{2}n^3 \\ T_3(n) &= 5n^2 \\ T_4(n) &= 100n \end{aligned}$$

Sappiamo dall'analisi matematica che esistono 5 costanti n_0, c_1, c_2, c_3, c_4 tali che, per $n > n_0$, risulta:

$$c_1 T_1(n) > c_2 T_2(n) > c_3 T_3(n) > c_4 T_4(n)$$

Per considerare esclusivamente il comportamento asintotico di un algoritmo introduciamo la notazione $\mathcal{O}$.

4.5 Definizione formale di $\mathcal{O}$

Definizione 7 *Diciamo che $T(n) = \mathcal{O}(f(n))$ se esistono due costanti positive c ed n_0 tali che per ogni $n > n_0$ risulti $T(n) < cf(n)$.*

In altre parole: per n sufficientemente grande il tasso di crescita di $T(n)$ è al più proporzionale a $f(n)$.

Nota bene 2 *la costante n_0 dipende dalla costante di proporzionalità c prefissata.*

Esempio 8 Sia

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

Si ha

$$2n + \frac{1}{2} \leq n^2$$

per $n \geq 3$, da cui

$$T(n) \leq \frac{3}{2}n^2$$

per $n \geq 3$, ovvero

$$T(n) = \mathcal{O}(n^2)$$

Le costanti utilizzate sono quindi:

$$c = \frac{3}{2} \qquad n_0 = 3$$

Esempio 9 Consideriamo il seguente programma per calcolare il quadrato di un numero n tramite somme ripetute:

```
quadrato := 0;
For i:=1 to n do
  quadrato := quadrato + n
```

La complessità rappresenterà in questo esempio il numero di operazioni di assegnazione e di somma effettuati nel caso pessimo. Certamente il tempo di esecuzione sarà lineare in n.

Nota bene 3 *Per rappresentare n in binario occorrono $\lfloor logn \rfloor + 1$ bits, dove $\lfloor x \rfloor$ denota la parte intera di x.*

Quindi, se n è rappresentato in binario allora l'algoritmo avrà una complessità esponenziale nella dimensione dell'input!

Se il nostro input n è rappresentato in notazione unaria allora l'algoritmo avrà una complessità lineare nella dimensione dell'input.

4.5.1 Alcuni ordini di grandezza tipici

Sia $T(n)$ una funzione definita sui numeri naturali. Qui di seguito mostriamo alcuni ordini di grandezza tipici, enumerati in maniera crescente. In altre parole, se $T(n) = \mathcal{O}(f(n))$, allora i seguenti sono tipici esempi di $f(n)$:

$$1$$

$$(\log n)^k$$

$$\sqrt{(n)}$$

$$n$$

$$n(\log n)^k$$

$$n^2$$

$$n^2(\log n)^k$$

$$n^3$$

$$n^3(\log n)^k$$

$$\dots$$

$$a^n$$

Nota bene 4 $\mathcal{O}$ *rappresenta* una *delimitazione asintotica superiore alla complessità dell'algoritmo, e non* la *delimitazione asintotica superiore.*

Infatti, se $T(n) = (n^4)$, è anche vero che:

$$T(n) = \mathcal{O}(n^7) \qquad T(n) = O(n^4 \log n) \qquad \text{ecc.}$$

Se per una funzione $T(n)$ sono note più delimitazioni asintotiche superiori, allora è da preferire quella più piccola.

4.6 La notazione Ω

È possibile definire anche una delimitazione inferiore asintotica al tasso di crescita di una funzione.

Sia $T(n)$ una funzione definita sull'insieme N dei numeri naturali (ad esempio $T(n)$ potrebbe rappresentare il tempo di esecuzione di un programma in funzione della dimensione dell'input).

Definizione 8 *Diciamo che $T(n) = \Omega(f(n))$ se esistono due costanti positive c ed n_0 tali che per ogni $n > n_0$ risulti $T(n) \geq c\ f(n)$.*

Nota bene 5 *$f(n)$ rappresenta una delimitazione inferiore asintotica, e non la delimitazione inferiore asintotica, al tasso di crescita di $T(n)$.*

Esempi

Sia $T(n) = 7n^2 + 6$. Si dimostra facilmente che $T(n) = \Omega(n^2)$. Attenzione: è anche vero che:

$$\begin{aligned} T(n) &= \Omega(n\ \log n) \\ T(n) &= \Omega(n) \\ T(n) &= \Omega(1) \end{aligned}$$

Sia $T(n) = 7n^2(\log n)^4 + 6n^3$. Si dimostra facilmente che $T(n) = \Omega(n^3)$. Attenzione: è anche vero che:

$$\begin{aligned} T(n) &= \Omega(n\ \log n) \\ T(n) &= \Omega(n) \\ T(n) &= \Omega(1) \end{aligned}$$

Se una funzione non è nota, ma sono note più delimitazioni asintotiche inferiori, va preferita la più grande tra esse. Ad esempio, se sapessimo che

$$\begin{aligned} T(n) &= \Omega(n^2 \log n) \\ T(n) &= \Omega(n(logn)^2) \\ T(n) &= \Omega(1) \end{aligned}$$

allora sarebbe opportuno asserire che $T(n) = \Omega(n^2 \log n)$.

Regola pratica

Per tutte le funzioni polinomiali e poli-logaritmiche, cioè della forma generale:

$$T(n) = \sum c_i n^t (\log n)^k$$

la delimitazione inferiore e quella superiore coincidono, e sono rispettivamente:

$$\mathcal{O}(n^h (\log n)^z) \text{ e } \Omega(n^h (\log n)^z)$$

dove h è il più grande esponente tra le t che compaiono nella somma, e z il più piccolo tra gli esponenti di $\log n$.

Esempio 10 La funzione $T(n) = 4n^5(\log n)^3 + 9n^3(logn)^5 + 125n^4(\log n)^7$ è $\mathcal{O}(n^5(\log n)^3)$ ed è $\Omega(n^5(\log n)^3)$.

4.6.1 Definizione alternativa di Ω

Definizione 9 *Diciamo che una funzione $T(n)$ è $\Omega(g(n))$ se esiste una costante positiva c ed una sequenza infinita $n_1, n_2, \ldots \to \propto$ tale che per ogni i risulti $T(n_i) > cg(n_i)$.*

Questa seconda definizione ha molti vantaggi sulla prima ed è diventata oramai la definizione standard.

Ricordiamo che dati due numeri reali a e b è sempre possibile confrontarli utilizzando l'operatore relazionale $\leq$. In altre parole, sull'insieme dei numeri reali è definita una relazione di ordinamento totale.

A differenza dei numeri reali, se utilizziamo la prima definizione di Ω, non è sempre possibile confrontare asintoticamente due funzioni.

Se utilizziamo però la seconda definizione di Ω allora, date due funzioni $T(n)$ e $g(n)$ asintoticamente positive risulta:

- $T(n) = \mathcal{O}(g(n))$; oppure
- $T(n) = \Omega(g(n))$.

4.7 La notazione Θ

Definizione 10 *Diciamo che $T(n) = \Theta(g(n))$ se esistono tre costanti positive c, d, n_0 tali che per ogni $n > n_0$ risulti $cg(n) < T(n) < dg(n)$.*

In altre parole, quando una funzione $T(n)$ è contemporaneamente $\mathcal{O}(g(n))$ e $\Omega(g(n))$, allora diciamo che $g(n)$ rappresenta una delimitazione asintotica stretta al tasso di crescita di $T(n)$, e scriviamo $T(n) = \Theta(g(n))$.

Esempio 11 La funzione $T(n) = 4n^5(\log n)^3 + 9n^3(\log n)^5 + 125n^4(\log n)^7$ è $\Theta(n^5(\log n)^3)$

4.8 Alcune proprietà di $\mathcal{O}, \Omega, \Theta$

1 La *transitività* vale per $\mathcal{O}, \Omega, \Theta$.
Ad esempio, $T(n) = \mathcal{O}(f(n))$ e $f(n) = \mathcal{O}(h(n))$ implicano

$$T(n) = \mathcal{O}(h(n))$$

2 La *riflessività* vale per $\mathcal{O}, \Omega, \Theta$.
Ad esempio, per ogni funzione T risulta $T(n) = \mathcal{O}(f(n))$.

3 La *simmetria* vale per Θ.
Ovvero, $T(n) = \Theta(f(n))$ se e solo se $f(n) = \Theta(T(n))$.

4 La *simmetria trasposta* vale per $\mathcal{O}$ e Ω.
Ovvero, $T(n) = \mathcal{O}(f(n))$ se e solo se $f(n) = \Omega(T(n))$.

I simboli $\mathcal{O}, \Theta, \Omega$ possono essere visti in due modi distinti:

1 Come **operatori relazionali**, che ci permettono di confrontare il comportamento asintotico di due funzioni;

2 Come **classi di funzioni** - in tal senso, $\mathcal{O}(g(n))$ denota ad esempio la classe di tutte le funzioni maggiorate asintoticamente da $g(n)$.

Alcuni autori assumono che le funzioni coinvolte siano asintoticamente positive. Per noi questa non è una limitazione, in quanto le nostre funzioni rappresentano tempo o spazio di esecuzione, e sono pertanto sempre positive.

4.9 Ricapitolando

Per studiare la complessità temporale di un algoritmo A occorre definire a priori:

1 Il modello di macchina utilizzato;

2 Cosa si intende per passo elementare di calcolo;

3 Come misurare la dimensione dell'input.

Esempio 12 Nel caso di una Macchina di Turing, avremo che il passo elementare è dato dalla transizione, e la dimensione dell'input non è altro che il numero di celle utilizzate per rappresentare l'istanza del problema.

La complessità nel caso pessimo non è altro che una funzione $T : N \to N$ definita come:

$$T(n) := \max \{\text{num. dei passi eseguiti da } A \text{ su un input } x \mid x \text{ ha dimensione } n\}$$

Poichè è difficile calcolare esattamente la funzione $T(n)$, ricorriamo alla notazione asintotica: $\mathcal{O}, \Theta, \Omega$.

4.10 Complessità di problemi

È possibile estendere la nozione di complessità temporale dagli algoritmi ai problemi, considerando che dato un problema P, esistono più algoritmi per risolvere P.

4.10.1 La notazione $\mathcal{O}$ applicata ai problemi

Definizione 11 *Un problema P ha complessità $\mathcal{O}(f(n))$ se e solo se esiste un algoritmo A per risolvere P di complessità $\mathcal{O}(f(n))$.*

Questa è una definizione *costruttiva*. Infatti essa presuppone che:

1 esista (sia noto) un algoritmo A per risolvere il problema P assegnato;

2 la complessità di A sia $\mathcal{O}(f(n))$.

Esempio 13 Il problema ordinamento. L'algoritmo *bubblesort* ha complessità $\mathcal{O}(n^2)$. Possiamo dire che il problema ordinamento ha complessità $\mathcal{O}(n^2)$.

Nota bene 6 *Esiste un algoritmo di ordinamento, l'* heapsort*, di complessità $\mathcal{O}(n \log n)$. Quindi, è anche vero che il problema ordinamento ha complessità $\mathcal{O}(n \log n)$.*

Come nel caso degli algoritmi, tra tante delimitazioni superiori scegliamo quella più piccola.

4.10.2 La notazione Ω applicata ai problemi.

Purtroppo, nel caso di un problema P, non è possibile definire una delimitazione asintotica inferiore alla sua complessità in termini degli algoritmi noti, in quanto il migliore algoritmo per risolvere P potrebbe (in generale) non essere ancora noto!

La definizione che viene data in questo caso non è più di natura costruttiva!

Definizione 12 *Un problema P ha complessità $\Omega(g(n))$ se qualunque algoritmo (noto o ignoto) per risolvere P richiede tempo $\Omega(g(n))$.*

Ω definisce quindi un limite inferiore alla complessità *intrinseca* di P.

In genere:

1 È relativamente semplice calcolare un limite asintotico superiore alla complessità di un problema assegnato.

2 È molto difficile stabilire un limite asintotico inferiore, che non sia banale.

I metodi per stabilire delimitazioni inferiori sono tratti generalmente da: teoria dell'informazione, logica matematica, algebra, ecc..

4.11 Algoritmi ottimali

Definizione 13 *Un algoritmo A che risolve un problema P è ottimale se:*

1 P ha complessità $\Omega(f(n))$.

2 A ha complessità $\mathcal{O}(f(n))$.

Esempio 14 Utilizzando tecniche tratte dalla teoria dell'informazione si dimostra che il problema ordinamento ha complessità $\Omega(n \log n)$. L'algoritmo *heapsort* ha complessità $\mathcal{O}(n \log n)$. L'algoritmo *heapsort* è quindi ottimale.

Nota sull'ottimalità.

Le nozioni fin qui discusse si riferiscono alle valutazioni di complessità nel caso pessimo. È possibile definire analoghe nozioni nel caso medio.

Esempio 15 Si dimostra che l'algoritmo *quicksort* ha complessità $\mathcal{O}(n^2)$ nel caso peggiore, e $\mathcal{O}(n \log n)$ nel caso medio. Quindi, l'algoritmo *quicksort* è ottimale nel caso medio.

4.12 Funzioni limitate polinomialmente

Definizione 14 *Diciamo che una funzione $f(n)$ è limitata polinomialmente se $f(n) = \mathcal{O}(n^k)$ per qualche $k > 0$.*

Alcuni autori scrivono $f(n) = n^{\mathcal{O}(1)}$ per indicare che la costante k è ignota.

4.13 Crescita moderatamente esponenziale

Definizione 15 *Una funzione $f(n)$ che cresca più velocemente di n^a per qualsiasi costante a, e più lentamente di c^n, per qualunque costante $c > 1$, è detta possedere crescita moderatamente esponenziale.*

Esempio 16 *Fattorizzazione di un intero:* dato un intero m, trovare un fattore non triviale di m ($\neq 1$ e $\neq m$).

Nota bene 7 *La complessità è misurata nel numero n di bits necessari per rappresentare m.*

I migliori algoritmi di fattorizzazione noti ad oggi hanno crescita moderatamente esponenziale.

4.14 Crescita esponenziale

Definizione 16 *Una funzione $f(n)$ ha crescita esponenziale se esiste una costante $c > 1$ tale che $f(n) = \Omega(c^n)$, ed esiste una costante $d > 1$ tale che $f(n) = \mathcal{O}(d^n)$.*

4.15 Appendice: Notazione asintotica all'interno di eguaglianze

I simboli $\mathcal{O}, \Theta, \Omega$ possono comparire anche all'interno di eguaglianze.

Esempio 17 L'espressione:

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

Sta ad indicare che

$$f(n) = g(n) + h(n)$$

per una qualche funzione $h(n)$ che è $\Theta(g(n))$.

Se i simboli $\mathcal{O}, \Theta, \Omega$ compaiono in entrambi i membri di una eguaglianza, questo sta ad indicare che per ogni assegnazione valida nella parte sinistra esiste una corrispondente assegnazione nella parte destra che rende l'eguaglianza vera.

Esempio 18 L'espressione

$$f(n) + \Theta(d(n)) = g(n) + \Theta(g(n))$$

Sta ad indicare che, per ogni funzione $i(n)$ che è $\Theta(d(n))$, esiste una funzione $h(n)$ che è $\Theta(g(n))$ tale che

$$f(n) + i(n) = g(n) + h(n)$$

4.16 Esercizi

Sia $f_1(n) := n^2$ se n è pari, $n^3 + 1$ se n è dispari.

Sia $f_2(n) := n^3$ se $n < 10$, $n \log n$ altrimenti.

Sia $f_3(n) := n^{2.5}$.

Per ogni coppia $(i, j) \quad 1 \leq i, j \leq 3$, determinare se

$$\begin{aligned} f_i(n) &= O(f_j(n)) \\ f_i(n) &= \Omega(f_j(n)) \\ f_i(n) &= \Theta(f_j(n)) \end{aligned}$$

Capitolo 5

Linguaggi per la descrizione di algoritmi

5.1 Introduzione

Per poter eseguire un algoritmo su una macchina occorre tradurlo in un programma, codificato utilizzando un linguaggio di programmazione.

In generale, un linguaggio è composto da un alfabeto di simboli e da un insieme di regole sintattiche. Nel caso dei linguaggi di programmazione l'alfabeto è quello dei simboli sulla tastiera e l'insieme di regole sintattiche è un insieme di costrutti base rigorosi che servono per gestire il flusso dei dati.

Per la descrizione degli algoritmi si adotta, in genere, uno pseudo-linguaggio. Uno pseudo-linguaggio è un misto di espressioni in linguaggio naturale ed un sottoinsieme dei costrutti fondamentali di un linguaggio di programmazione. I costrutti del linguaggio di programmazione utilizzati sono: il salto, la selezione, i cicli, l'assegnazione.

L'utilità di uno pseudo-linguaggio sta nella sua semplicità di scrittura e nella facile e rapida comprensione (costrutti dei linguaggi di programmazione) e nella sinteticità (espressioni in linguaggio naturale).

Mostriamo qui di seguito alcune regolette che definiscono un semplice pseudo-linguaggio derivato dal Pascal.

- Istruzioni semanticamente semplici sono scritte in linguaggio naturale. Ad esempio:
  scambia x **ed** y
  sta per:

1. temp := x
2. x:= y
3. y:= temp
rendendo cosi' l'algoritmo più facile da leggere ed analizzare.

- Etichette non numeriche sono utilizzate per le operazioni di salto. Ad esempio:
goto *uscita*

- All'interno di funzioni l'istruzione:
return (*espressione*)
è usata per valutare l'espressione, assegnare il valore alla funzione, ed infine terminarne l'esecuzione.

- All'interno di procedure l'istruzione:
return
è usata per terminare l'esecuzione della procedura.

5.2 Complessità di algoritmi espressi in pseudo-codice

- La complessità di una operazione basica di assegnazione o di Input-Output è $\mathcal{O}(1)$;

- La complessità di un numero costante (indipendente dall'input) di operazioni di costo $\mathcal{O}(1)$ è ancora $\mathcal{O}(1)$;

- La complessità di una sequenza di istruzioni è data dalla somma delle loro complessità (attenzione: il numero di istruzioni che compongono la sequenza deve essere costante).
Nota: in notazione $\mathcal{O}$ la somma delle complessità individuali di un numero costante di termini è data dal massimo delle complessità individuali:
$\mathcal{O}(1) + \mathcal{O}(1) + \mathcal{O}(n) + \mathcal{O}(n^2) = \mathcal{O}(n^2)$
$\mathcal{O}(n^2 \log n) + \mathcal{O}(n^2) + \mathcal{O}((\log n)^7) = \mathcal{O}(n^2 \log n)$

- La complessità di una istruzione condizionale:
if condizione then $azione_1$ **else** $azione_2$

è $\mathcal{O}(1)$ più il costo di valutazione della condizione più il costo associato all'esecuzione dell'azione effettuata.

- La complessità di un ciclo: (**for...**, **while...**, **repeat...**)
  è $\mathcal{O}(1)$ più il costo di ciascuna iterazione che è dato dalla complessità di valutazione della condizione di uscita dal ciclo ($\mathcal{O}(1)$ nel caso del **for**) più la complessità delle istruzioni eseguite.

Attenzione. Occorre fare molta attenzione quando si calcola la complessità di istruzioni espresse in linguaggio naturale:

scambia x **e** y
costa $\mathcal{O}(1)$ se x ed y sono variabili di dimensione costante, ad esempio interi. Ciò non è più vero se ad esempio x ed y sono vettori di dimensione dipendente dall'input!

ordina il vettore A
costa $\mathcal{O}(1)$ se la dimensione del vettore A è costante. Ciò non è vero se la dimensione di A dipende dall'input.

inserisci l'elemento E **nella lista** L
può richiedere tempo costante o meno, dipendentemente dall'implementazione della lista.

Esempio.

```
1. x := 0
2. For i:=1 to n do
3.    x := x+i
```

Il passo 1 costa $\mathcal{O}(1)$. Il passo 3 costa $\mathcal{O}(1)$. I passi 2 e 3 costano: $\mathcal{O}(1) + n\,[\mathcal{O}(1)+\mathcal{O}(1)] = \mathcal{O}(1)+n\,\mathcal{O}(1) = \mathcal{O}(1)+\mathcal{O}(n) = \mathcal{O}(n)$. L'intero algoritmo ha costo: $\mathcal{O}(1) + \mathcal{O}(n) = \mathcal{O}(n)$, ovvero lineare nella dimensione dell'input.

Esercizio

```
1. x := 0
2. i :=1
3. While i<=n do
4.    For j:=1 to i do
```

5. Begin
6. x := x+i *j
7. y := i+1
8. End
9. i:=i+1

Dimostrare che il tempo richiesto è $\mathcal{O}(n^2)$. (Suggerimento: quando si analizza la complessità di cicli annidati, procedere dall'interno verso l'esterno)

5.3 Alcune regole per il calcolo di $\mathcal{O}$

Regola della somma

Se una sezione di codice A di complessità $\mathcal{O}(f(n))$ è seguita da una sezione di codice B di complessità $\mathcal{O}(g(n))$ allora la complessità di $A + B$ (ovvero, la complessità del tutto) è data da $\mathcal{O}(\max[f(n), g(n)])$.

Regola del prodotto

Se una sezione di codice A di complessità $\mathcal{O}(f(n))$ è eseguita $\mathcal{O}(g(n))$ volte, allora la complessità globale è $\mathcal{O}(f(n)\ g(n))$.

Caso particolare: Se $g(n) = k$, costante indipendente da n, allora la complessità globale è $\mathcal{O}(k\ f(n)) = \mathcal{O}(f(n))$ (poichè k è $\mathcal{O}(1)$).

Applicazione: funzioni polinomiali
Sia

$$T(n) = c_k n^k + c_{k-1} n^{k-1} + \ldots + c_0$$

il tempo di esecuzione di un algoritmo. Possiamo supporre che il nostro algoritmo sia composto da una sequenza di $k+1$ sezioni di codice $P_0, \ldots, P_k$, di complessità rispettivamente $\mathcal{O}(n^i)$ iterata ciascuna c_i volte. Si ha che $T(n) = \mathcal{O}(n^k)$. Tale risultato si deduce facilmente dalla regola della somma:

$$T(n) = T_k(n) + \ldots + T_0(n)$$

dove

$$T_i(n) = c_i\ n^i$$

e dalla regola del prodotto, poichè

$$T_i(n) = \mathcal{O}(c_i\ n^i) = \mathcal{O}(n^i)$$

Capitolo 6

Algoritmi ricorsivi

6.1 Introduzione

> *Si definisce ricorsiva una procedura P che invoca, direttamente o indirettamente, se stessa.*

L'uso della ricorsione permette di ottenere spesso una descrizione chiara e concisa di algoritmi. Diremo che un algoritmo è ricorsivo se figurano in esso delle procedure ricorsive.

Gli algoritmi ricorsivi:

1 Spesso costituiscono il metodo più naturale di risoluzione di un problema;

2 Sono facili da comprendere e analizzare:

- La correttezza di un programma ricorsivo si dimostra facilmente utilizzando il Principio di Induzione;
- Il calcolo della complessità temporale di un algoritmo ricorsivo si riduce alla soluzione di relazioni di ricorrenza.

6.2 Esempio

Calcolo del fattoriale di un numero

Mostriamo qui di seguito una procedura iterativa ed una ricorsiva per calcolare il fattoriale $n!$ di un numero n. Ricordiamo che

$$n! = n \cdot (n-1) \cdot (n-2) \cdots 1$$

Versione iterativa

```
1. Fattoriale(n)
2. int fatt;
3. fatt=1;
4. for i=2 to n
5.   fatt := fatt * i
6.   return( fatt )
```

Versione ricorsiva

```
1. Fattoriale(n)
2. if n=1
3.   then return(1)
4.  else return(n * Fattoriale(n-1))
```

Come si evince dal precedente esempio, la versione ricorsiva è più concisa, più facile da comprendere e da analizzare. Occorre comunque notare che l'implementazione ricorsiva non offre alcun vantaggio in termini di velocità di esecuzione (cioè di complessità).

La dimostrazione di correttezza della prima versione è alquanto laboriosa. Per quanto riguarda la seconda versione, la correttezza si dimostra facilmente utilizzando il Principio di Induzione. Infatti:

1 *(correttezza del passo base)*
L'algoritmo è certamente corretto per $n = 1$, in quanto $1! = 1$;

2 *(correttezza del passo induttivo)*
Se l'algoritmo calcola correttamente $(n-1)!$, allora esso calcola correttamente $n!$, che è dato da $n \cdot (n-1)!$.

6.3 Linguaggi che consentono la ricorsione

Non tutti i linguaggi di programmazione permettono la ricorsione: ad esempio, il FORTRAN oppure il COBOL non la permettono.
Il linguaggio PASCAL, oppure il linguaggio C, la permettono, utilizzando i seguenti metodi:

Procedura attiva

Con il termine *procedura attiva* indichiamo la procedura in esecuzione in un determinato istante. Tale procedura è caratterizzata da un contesto, che è costituito da tutte le variabili globali e locali note alla procedura in tale istante.

Pila e record di attivazione

I linguaggi che permettono la ricorsione utilizzano in fase di esecuzione una pila per tenere traccia della sequenza di attivazione delle varie procedure ricorsive. In cima a tale pila sarà presente il *record di attivazione* della procedura correntemente attiva. Tale record contiene:

- le variabili locali alla procedura;
- i parametri ad essa passati;
- l'indirizzo di ritorno, ovvero il contenuto del *program counter* nel momento in cui la procedura è stata invocata.

Quando una procedura P è invocata viene posto sulla pila un nuovo record di attivazione. Ciò indipendentemente dal fatto che siano già presenti nella pila altri record di attivazione relativi alla stessa procedura.

Quando la procedura attiva P termina viene rimosso dalla pila il record di attivazione che si trova in cima ad essa, e l'indirizzo di ritorno in essa contenuto viene utilizzato per cedere il controllo nuovamente alla procedura chiamante (vale a dire, alla penultima procedura attiva).

6.3.1 Visita di un albero binario

Due esempi di algoritmo ricorsivo e non-ricorsivo, rispettivamente, sono:

```
procedura VisitaInOrdine(v):

    begin
        if figliosnistro[v]!=0 then
           VisitaInOrdine(filgiosinistro[v]);
        num[v]=count;
        count=count+1;
```

```
        if figliodestro[v]!=0 then
            VisitaInOrdine{figliodestro[v]}
    end

    begin
            count=1;
            v=root;
            stack=null;
left:       while figliosinistro[v]!=0 do
                begin
                    PUSH v;
                    v=figliosinistro[v]
                end;
center:     num[v]=count;
            count=count+1;
            if figliodestro[v]!=0 then
                begin
                    v=figliodestro[v];
                    goto left
                end;
            if stack!=null then
                begin
                    v=primo elemento dello stack;
                    POP stack;
                    goto center
                    end
    end
```

Si noti che la versione non-ricorsiva fa uso esplicito di una pila per tenere traccia dei nodi visitati.

La versione ricorsiva è molto più chiara da comprendere; *la pila c'è ma non si vede,* ed è semplicemente lo stack che contiene i record di attivazione delle procedure.

Capitolo 7

L'approccio Divide et Impera

Esistono differenti tecniche generali di progetto di algoritmi che consentono di risolvere efficientemente molti problemi. In un primo approccio evidenziamo:

- Il metodo Divide et Impera;
- La programmazione dinamica.

7.1 Introduzione

Il metodo Divide et Impera è una tra le tecniche di progetto più importanti, e dal più vasto spettro di applicazioni. Esso consiste nel dividere un problema in sottoproblemi di dimensione più piccola, risolvere ricorsivamente tali sottoproblemi, e quindi ottenere dalle soluzioni dei sottoproblemi la soluzione del problema globale.

In un algoritmo ottenuto attraverso tale tecnica si individuano 3 fasi:

1 *suddivisione del problema originale in sottoproblemi*
Il problema originale di dimensione n è decomposto in a sottoproblemi di dimensione minore $f_1(n), \ldots, f_a(n)$;

2 *soluzione ricorsiva dei sottoproblemi*
Gli a sottoproblemi sono risolti ricorsivamente;

3 *combinazione delle soluzioni*
Le soluzioni degli a sottoproblemi sono combinate per ottenere la soluzione del problema originale.

A ciascuna fase sono associati dei costi. Indichiamo con $Sudd(n)$ il tempo richiesto dal punto 1, e con $Fus(n)$ il tempo richiesto dal punto 3, per un input di dimensione n. Il costo associato al punto 2 sarà ovviamente

$$T(f_1(n)) + \ldots + T(f_a(n))$$

Si noti che problemi di dimensione sufficientemente piccola possono essere risolti in tempo costante c, consultando una tabella costruita in precedenza. Otteniamo pertanto la seguente equazione che esprime il tempo di esecuzione dell'algoritmo:

$$\begin{cases} T(n) = c & n < n_0 \\ T(n) = T(f_1(n)) + \ldots + T(f_a(n)) + Sudd(n) + Fus(n) & n \geq n_0 \end{cases}$$

Si noti che, molto spesso, il tempo di suddivisione in sottoproblemi è irrilevante, oppure è assorbito dagli altri termini.

In ogni caso indichiamo con $d(n)$ la somma dei termini $Sudd(n)$ e $Fus(n)$, e chiamiamo tale termine la *funzione guida* della equazione di ricorrenza.

7.2 Esempio: il Mergesort

Sia S una lista di n elementi da ordinare. Assumiamo che n sia una potenza di 2, ovvero $n = 2^k$, in modo da poter applicare un procedimento dicotomico. Nell'algoritmo che segue L, L_1 e L_2 indicano variabili locali di tipo lista.

1. Mergesort(S **):**
2. Se S **ha 2 elementi**
3. **ordina** S **con un solo confronto e ritorna la lista ordinata**
4. altrimenti
5. **Spezza** S **in due sottoliste** S_1 **e** S_2 **aventi la stessa cardinalità**
6. **Poni** L_1 **= Mergesort(** S_1 **)**
7. **Poni** L_2 **= Mergesort(** S_2 **)**
8. **Fondi le due liste ordinate** L_1 **ed** L_2 **in una unica lista** L
9. **Ritorna(** L **)**

La fusione di due liste ordinate in una unica lista si effettua molto semplicemente in n passi, vale a dire in tempo $\Theta(n)$. Ad ogni passo occorre

semplicemente confrontare i due elementi più piccoli delle liste L_1 ed L_2, estrarre il minore tra i due dalla lista a cui appartiene, ed inserirlo in coda alla nuova lista L.

Si nota facilmente che:

- *il tempo richiesto per ordinare una lista di cardinalità 2 è costante;* Quindi possiamo scrivere $T(2) = O(1)$.
- *per ordinare una lista di dimensione maggiore di 2 occorre:*
  - *spezzare la lista in due sottoliste*
    questo richiede tempo costante
  - *ordinare ricorsivamente le 2 sottoliste di dimensione* $n/2$
    questo richiede tempo $T(n/2) + T(n/2)$
  - *fondere il risultato*
    questo richiede tempo $\Theta(n)$

  Quindi $T(n) = O(1) + T(n/2) + T(n/2) + \Theta(n) = 2\ T(n/2) + \Theta(n)$.

Le due formule per $T(n)$ appena viste, che esprimono tale funzione quando $n = 2$ e quando n è una potenza di 2, insieme costituiscono ciò che si chiama una *equazione di ricorrenza*. In altre parole, possiamo calcolare $T(n)$ se conosciamo $T(n/2)$, e cosi' via ... Poichè $T(2)$ è noto, ci è possibile calcolare $T(n)$ per qualunque valore di n che sia una potenza di due.

Mostreremo in seguito come risolvere tale relazione, ovvero come ottenere una stima per $T(n)$ che non dipenda da $T(h)$ per qualche $h < n$. Per ora anticipiamo che $T(n) = O(n \log n)$, e quindi il Mergesort è sicuramente ottimale (essendo $n \log n$ un limite inferiore al problema dell'ordinamento).

7.3 Bilanciamento

Abbiamo visto che nel Mergesort il problema originale viene suddiviso in sottoproblemi di pari dimensione. Questo non è un caso. Un principio generale (dettato dall'esperienza) per ottenere algoritmi efficienti utilizzando l'approccio Divide et Impera, è quello di partizionare il problema assegnato in sottoproblemi aventi all'incirca la stessa dimensione.

Se indichiamo con a il numero dei sottoproblemi e con n/b la dimensione di ciascun sottoproblema, allora il tempo di esecuzione di un tale algoritmo

sarà descritto dalla seguente equazione di ricorrenza:

$$\begin{cases} T(1) &= c & n < n_0 \\ T(n) &= a\ T(n/b) + d(n) & n \geq n_0 \end{cases}$$

Tali equazioni di ricorrenza possono essere risolte utilizzando tecniche standard che vedremo nei seguenti capitoli. Si noti che nel Mergesort a e b sono uguali tra loro (pari a 2); questo è semplicemente un caso, e non costituisce una regola generale. Nell'algoritmo di moltiplicazione di matrici di Strassen vedremo, ad esempio, che $a = 7$ e $b = 2$.

Controesempio: l'Insertion Sort

L'Insertion Sort può essere visto come un caso particolare del Mergesort, dove le sottoliste da ordinare hanno dimensione rispettivamente 1 ed $n - 1$, essendo n la dimensione della lista originale. È noto che la complessità di tale algoritmo è $O(n^2)$. La sua versione ricorsiva è la seguente:

1. InsertionSort(S **):**
2. Se S **ha 1 elemento**
3. **Ritorna** S
4. altrimenti
5. **Dividi** S **in due sottoliste** S_1 **e** S_2 **di cardinalità** 1 **ed** $n - 1$
6. **Poni** L_1 **= InsertionSort(** S_1 **)**
7. **Poni** L_2 **= InsertionSort(** S_2 **)**
8. **Fondi le due liste ordinate** L_1 **ed** L_2 **in una unica lista** L
9. **Ritorna(** L **)**

La complessità di tale algoritmo è espressa dalla relazione di ricorrenza:

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

la cui soluzione è $T(n) = O(n^2)$.

7.4 L'algoritmo di Strassen

Siano assegnate due matrici quadrate $A = [a_{ij}]$ e $B = [b_{ij}]$ di dimensione n. Si voglia calcolare la matrice prodotto $A \cdot B = [c_{ij}]$, dove $c_{ij} = \sum_{k=1}^{n} a_{ik}\ b_{kj}$.

L'algoritmo tradizionale che calcola il prodotto delle due matrici richiede $O(n^3)$ operazioni aritmetiche (somme e prodotti).

È possibile fare di meglio? Supponiamo per semplicità che n sia una potenza di 2. Allora possiamo decomporre le matrici A, B e C ciascuna in 4 sottomatrici quadrate di dimensione $n/2$ (ovvero possiamo considerarle come matrice a blocchi):

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \qquad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \qquad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & B_{22} \end{pmatrix} \quad (7.1)$$

Si noti ora che la generica sottomatrice C_{ij} è data da

$$A_{i1}\ B_{1j} + A_{i2}\ B_{2j} \quad (7.2)$$

Possiamo quindi pensare al seguente algoritmo ricorsivo:

1. Se A e B hanno dimensione $n = 1$
2. Allora moltiplica (banalmente) le due matrici
3. Altrimenti
4. Dividi A e B come in (7.1)
5. Calcola le sottomatrici C_{ij} di C utilizzando la relazione (7.2)

Qual è la complessità di tale algoritmo? È possibile calcolare ogni matrice C_{ij} con due moltiplicazioni di matrici quadrate di dimensione $n/2$ ed una somma di matrici quadrate di dimensione $n/2$.

Il prodotto di due matrici quadrate di dimensione n può essere allora ricondotto ricorsivamente ad 8 prodotti di matrici quadrate di dimensione $n/2$, e quattro addizioni di matrici quadrate di dimensione $n/2$.

Poichè due matrici quadrate di dimensione $n/2$ si sommano con $(n/2)^2$ operazioni elementari (somme di scalari), è possibile esprimere il tempo di esecuzione dell'algoritmo attraverso la seguente equazione:

$$\begin{cases} T(1) &= 1 \\ T(n) &= 8\ T(n/2) + 4\ (n/2)^2 \quad (n > 1) \end{cases}$$

La soluzione di questa equazione di ricorrenza è data da $T(n) = 0(n^3)$. Pertanto tale algoritmo non presenta alcun vantaggio sull'algoritmo tradizionale.

L'idea di Strassen

Si calcolino le seguenti matrici quadrate P_i, di dimensione $n/2$:

$$\begin{aligned} P_1 &= (A_{11} + A_{22})\ (B_{11} + B_{22}) \\ P_2 &= (A_{21} + A_{22})\ B_{11} \\ P_3 &= A_{11}\ (B_{12} - B_{22}) \\ P_4 &= A_{22}\ (B_{21} - B_{11}) \\ P_5 &= (A_{11} + A_{12})\ B_{22} \\ P_6 &= (A_{21} - A_{11})\ (B_{11} + B_{12}) \\ P_7 &= (A_{12} - A_{22})\ (B_{21} + B_{22}) \end{aligned}$$

Si verifica che:

$$\begin{aligned} C_{11} &= P_1 + P_4 - P_5 + P_7 \\ C_{12} &= P_3 + P_5 \\ C_{21} &= P_2 + P_4 \\ C_{22} &= P_1 + P_3 - P_2 + P_6 \end{aligned}$$

È possibile calcolare le matrici P_i con 7 moltiplicazioni di matrici quadrate di dimensione $n/2$ e 10 fra addizioni e sottrazioni di matrici quadrate di dimensione $n/2$. Analogamente, le C_{ij} si possono calcolare con 8 addizioni e sottrazioni a partire dalle matrici P_i.

Il prodotto di due matrici quadrate di dimensione n può essere allora ricondotto ricorsivamente al prodotto di 7 matrici quadrate di dimensione $n/2$, e 18 tra addizioni e sottrazioni di matrici quadrate di dimensione $n/2$.

Poichè due matrici quadrate di dimensione $n/2$ si sommano (o sottraggono) con $(n/2)^2$ operazioni, è possibile esprimere il tempo di esecuzione dell'algoritmo di Strassen attraverso la seguente equazione di ricorrenza:

$$\begin{cases} T(1) &= 1 \\ T(n) &= 7\ T(n/2) + 18\ (n/2)^2 \qquad (n > 1) \end{cases}$$

La soluzione di questa equazione di ricorrenza è data da $T(n) = \mathcal{O}(n^{\log_2 7})$.

Poichè $\log_2 7 < \log_2 8 = 3$, l'algoritmo di Strassen risulta più efficiente dell'algoritmo classico di moltiplicazione di matrici.

Capitolo 8

Tecniche di analisi di algoritmi ricorsivi

Esistono delle semplici regole per calcolare la complessità di un algoritmo iterativo espresso con uno pseudo-linguaggio per la descrizione di algoritmi.

In particolare sono state presentate:

- La *regola della somma*,
  per calcolare la complessità di una sequenza costituita da un numero *costante* di blocchi di codice;

- La *regola del prodotto*,
  per calcolare la complessità di una sezione di programma costituita da un blocco di codice che viene eseguito iterativamente.

Le tecniche per analizzare la complessità di algoritmi ricorsivi sono differenti dalle precedenti, ed utilizzano a tal fine le relazioni 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 termini precedenti. Presentiamo le principali tecniche utilizzate per risolvere equazioni di questo tipo o almeno per ottenere una soluzione approssimata.

Supponiamo di dover analizzare un algoritmo definito mediante un insieme di procedure $P_1, P_2, \ldots, P_m$ che si richiamano ricorsivamente fra loro.

Vogliamo quindi stimare, per ogni $i = 1, 2, \ldots, m$ la funzione $T_i(n)$ che rappresenta il tempo di calcolo impiegato dalla i-esima 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 $\forall j \in \{1, 2, \ldots, m\} : k < n$.

Supponiamo per semplicità di avere una sola procedura P che chiama ricorsivamente se stessa su dati di dimensione minore. Sia $T(n)$ il tempo di calcolo richiesto da P su un input di dimensione n, nel caso peggiore. Sarà in generale possibile esprimere $T(n)$ come funzione dei precedenti termini $T(m)$ con $1 \leq m < n$. In molti degli algoritmi presentati in questo libro $T(n)$ dipende funzionalmente da un solo termine $T(m)$, con $m < n$.

Una equazione che lega funzionalmente $T(n)$ a $T(m)$ con $m < n$ è detta equazione di ricorrenza.

Definizione 17 *Diciamo che una sequenza $a_1, a_2, \ldots$ di elementi appartenenti ad un insieme S è definita attraverso una equazione di ricorrenza di ordine k se il generico termine a_n dipende funzionalmente dai precedenti k, ovvero se*

$$a_n = f(a_{n-1}, \ldots, a_{n-k})$$

per qualche funzione $f : S^k \to S$.

Si noti che per poter definire univocamente la sequenza occorre assegnare le condizioni iniziali, ovvero specificare i valori dei primi k termini della sequenza: $a_1, \ldots, a_k$.

Le equazioni di ricorrenza che incontreremo in seguito sono tutte di ordine 1 (del primo ordine).

Fattoriali. La sequenza dei fattoriali dei numeri naturali:

$$1, 2, 6, 24, \ldots$$

si può esprimere attraverso la seguente equazione di ricorrenza del primo ordine:

$$\begin{aligned} 1! &= 1 \\ n! &= n \cdot (n-1)! \qquad\qquad (n > 1) \end{aligned}$$

Numeri di Fibonacci. Un'altra sequenza notevole è costitituita dai numeri di Fibonacci F_n, ed è definita dalla seguente equazione di ricorrenza del

secondo ordine:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \qquad (n > 1) \end{aligned}$$

Problema: Data una sequenza definita attraverso una equazione di ricorrenza di ordine k vogliamo trovare una *formula chiusa* per il termine n-mo di tale sequenza, ovvero una formula per esprimere l'n-mo termine senza fare riferimento ai termini precedenti.

Esempio: Consideriamo la seguente equazione di ricorrenza:

$$\begin{aligned} a_0 &= 1 \\ a_n &= c\ a_{n-1} \qquad (n > 0) \end{aligned}$$

dove c è una costante assegnata. Evidentemente, l'unica soluzione con la condizione al contorno $a_0 = 1$ è data da $a_n = c^n$.
Si osservi che data la condizione al contorno (ovvero, se sono noti tutti i valori di $T(n)$ per $0 < n \leq n_0$) allora esiste un'unica funzione $T(n)$ che soddisfa l'equazione di ricorrenza assegnata.

Per collegare quanto appena detto all'analisi di algoritmi, consideriamo il Mergesort. Ricordiamo che il tempo di esecuzione di tale algoritmo su un input di dimensione n dipende funzionalmente (in qualche modo) dal tempo di esecuzione dello stesso algoritmo su un input di dimensione $n/2$. Possiamo inoltre assumere che il tempo si esecuzione dell'algoritmo su un problema di dimensione 2 sia dato da una costante c. Pertanto otteniamo la seguente sequenza:

$$T(2),\ T(4),\ T(8),\ T(16), \ldots$$

ovvero, se utilizziamo la notazione precedentemente introdotta, otteniamo una sequenza $a_1,\ a_2, \ldots$ dove $a_1 = c$ e $a_n = T(2^n)$.

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

- Deduzione delle relazioni di ricorrenza contenenti come incognita la funzione $T(n)$ da stimare;
- Soluzione delle equazioni di tali relazioni di ricorrenza.

8.1 Esempio: Visita di un albero binario

Si consideri la seguente procedura ricorsiva di attraversamento di un albero binario:

1. preorder(v):
2. inserisci v in una lista
3. se v ha un figlio sinistro w
4. allora preorder(w)
5. se v ha un figlio destro w
6. allora preorder(w)

Tale procedura viene invocata passando come parametro la radice dell'albero da attraversare. Qual è la sua complessità? Si supponga che il passo (2) richieda tempo costante. Denotiamo con $T(n)$ il tempo richiesto per attraversare un albero costituito da n nodi. Supponiamo che il sottoalbero sinistro di v abbia i nodi ($0 \leq i < n$). Allora, il sottoalbero destro di v avrà $n-i-1$ nodi. Pertanto, l'attraversamento di un albero binario avente radice v richiederà:

1 Tempo costante c per il passo (2);

2 Tempo $T(i)$ per i passi (3) e (4);

3 Tempo $T(n-i-1)$ per i passi (5) e (6).

da cui si ricava la seguente equazione di ricorrenza:

$$\begin{aligned} T(0) &= 0 \\ T(n) &= c + T(i) + T(n-i-1) \qquad (0 \leq i < n) \end{aligned}$$

Lo studente può verificare facilmente per induzione che la soluzione di tale equazione di ricorrenza è data da $T(n) = cn$, e pertanto $T(n) = \Theta(n)$.

8.2 Soluzione delle equazioni di ricorrenza

La risoluzione delle equazioni di ricorrenza è essenziale per l'analisi degli algoritmi ricorsivi. Sfortunatamente non c'è un metodo generale per risolvere equazioni di ricorrenza arbitrarie. Ci sono, comunque, diverse strade per attaccarle.

1 **La forma generale della soluzione potrebbe essere nota.**
In questo caso occorre semplicemente determinare i parametri incogniti.

Esempio 19 Sia data l'equazione:

$$\begin{cases} T(1) &= 7 \\ T(n) &= 2T(\frac{n}{2}) + 8 \end{cases}$$

Si ipotizzi che la soluzione abbia la seguente forma:

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

Occorre ora determinare i parametri a e b. Questo si riduce alla risoluzione di un sistema lineare di due equazioni in due incognite, come segue:

$$\begin{array}{rcrcl} T(1) & = & a+b & = & 7 \\ T(n) & = & an+b & = & 2T(\frac{n}{2}) + 8 \\ & & & = & 2(a\frac{n}{2} + b) + 8 \\ & & & = & an + 2b + 8 \end{array}$$

da cui

$$b = -8 \qquad a = 15$$

quindi la nostra ipotesi era corretta, e si ha:

$$T(n) = 15n - 8$$

2 **Espansione della formula di ricorrenza**
In altre parole, occorre sostituire $T(\cdot)$ sul lato destro della ricorrenza finchè non vediamo un pattern.

Esempio 20 Sia data l'equazione:

$$T(n) = \begin{cases} 1 & per \quad n = 2 \\ 2T(\frac{n}{2}) + 2 & per \quad n > 2 \end{cases}$$

Si ha allora

$$\begin{aligned} T(n) &= 2T(\tfrac{n}{2}) + 2 \\ &= 2[2T(\tfrac{n}{4}) + 2] + 2 \\ &= 4T(\tfrac{n}{4}) + 4 + 2 \\ &= 4[2T(\tfrac{n}{8}) + 2] + 4 + 2 \\ &= 8T(\tfrac{n}{8}) + 8 + 4 + 2 \\ &\vdots \\ &= \underbrace{2^{k-1}}_{\frac{n}{2}} \underbrace{T(\frac{n}{2^{k-1}})}_{T(2)=1} + \underbrace{2^{k-1} + 2^{k-2} + 2^{k-3} + \ldots + 4 + 2}_{\sum_{p=1}^{k-1} 2^p} \\ &= \tfrac{n}{2} + n - 2 = \tfrac{3}{2}n - 2 \end{aligned}$$

Ricordiamo, per comodità del lettore, che la somma dei primi $k+1$ termini di una *serie geometrica*:

$$q^0 + q^1 + q^2 + \ldots + q^{k-1} + q^k = \sum_{p=0}^{k} q^p$$

è data da

$$\frac{q^{k+1} - 1}{q - 1}$$

da cui

$$\sum_{p=1}^{k-1} 2^p = \frac{2^k - 1}{2 - 1} - 1 = 2^k - 2 = n - 2$$

8.3 Il caso Divide et Impera

Una importante classe di equazioni di ricorrenza è legata all'analisi di algoritmi del tipo divide et impera trattati in precedenza.

Ricordiamo che un algoritmo di questo tipo suddivide il generico problema originale di dimensione n in un certo numero a di sottoproblemi (che sono istanze del medesimo problema, di dimensione inferiore) ciascuno di dimensione n/b per qualche $b > 1$; quindi, richiama ricorsivamente se stesso su tali istanze ridotte e poi fonde i risultati parziali ottenuti per determinare la soluzione cercata.

Senza perdita di generalità, il tempo di calcolo di un algoritmo di questo tipo può essere descritto da una equazione di ricorrenza del tipo:

$$T(n) = a\,T(n/b) + d(n) \qquad (n = b^k \;\; k > 0) \tag{8.1}$$
$$T(1) = c \tag{8.2}$$

dove il termine $d(n)$, noto come *funzione guida*, rappresenta il tempo necessario per spezzare il problema in sottoproblemi e fondere i risultati parziali in un'unica soluzione.

Si faccia ancora una volta attenzione al fatto che $T(n)$ è definito solo per potenze positive di b. Si assuma inoltre che la funzione $d(\cdot)$ sia *moltiplicativa*, ovvero

$$d(xy) = d(x)\;d(y) \qquad x, y \in N$$

Il comportamento asintotico della funzione $T(\cdot)$ è determinato confrontando innanzitutto a con $d(b)$.

Teorema Principale Si dimostra che:

$$\begin{array}{ll} \text{se } a > d(b): & T(n) = O(n^{\log_b a}) \\ \text{se } a < d(b): & T(n) = O(n^{\log_b d(b)}) \\ \text{se } a = d(b): & T(n) = O(n^{\log_b d(b)} \log_b n) \end{array}$$

Inoltre, se $d(n) = n^p$, si ha

$$\begin{array}{ll} \text{se } a > d(b): & T(n) = O(n^{\log_b a}) \\ \text{se } a < d(b): & T(n) = O(n^p) \\ \text{se } a = d(b): & T(n) = O(n^p \log_b n) \end{array}$$

Si noti che il caso $d(n) = n^p$ è piuttosto comune nella pratica (ad esempio, nel MERGESORT si ha $d(n) = n$).

Caso particolare: $d(\cdot)$ non è moltiplicativa

Supponiamo che la funzione guida non sia moltiplicativa, ad esempio:

$$T(1) = c$$
$$T(n) = 2\,T(n/2) + f \cdot n \qquad n > 1$$

Si noti che la funzione $d(n) = f\ n$ non è moltiplicativa. Consideriamo allora la funzione $U(n)$ definita come $U(n) = T(n)/f$, da cui deriva $T(n) = U(n) \cdot f$. Possiamo scrivere quindi

$$\begin{aligned} f\ U(1) &= c \\ f\ U(n) &= 2 \cdot f \cdot U(n/2) + f \cdot n \qquad\qquad n > 1 \end{aligned}$$

da cui

$$\begin{aligned} U(1) &= c/f \\ U(n) &= 2 \cdot U(n/2) + n \qquad\qquad n > 1 \end{aligned}$$

Si noti che in tale equazione la funzione guida è $d(n) = n$ che è moltiplicativa. Risolviamo per $U(n)$, ottenendo $U(n) = \mathcal{O}(n \log n)$. Sostituiamo, infine, ottenendo $T(n) = f \cdot U(n) = f \cdot \mathcal{O}(n \log n) = \mathcal{O}(n \log n)$.

8.3.1 Dimostrazione del Teorema Principale

Sia assegnata la seguente equazione di ricorrenza:

$$\begin{cases} T(1) &= c \\ T(n) &= aT(\frac{n}{b}) + d(n) \end{cases}$$

Si noti che

$$T\left(\frac{n}{b^i}\right) = aT\left(\frac{n}{b^{i+1}}\right) + d\left(\frac{n}{b^i}\right)$$

da cui discende che

$$\begin{aligned} T(n) &= aT\left(\frac{n}{b}\right) + d(n) \\ &= a\left[aT\left(\frac{n}{b^2}\right) + d\left(\frac{n}{b}\right)\right] d(n) \\ &= a^2 T\left(\frac{n}{b^2}\right) + ad\left(\frac{n}{b}\right) + d(n) \\ &= a^2\left[aT\left(\frac{n}{b^3}\right) + d\left(\frac{n}{b^2}\right)\right] + ad\left(\frac{n}{b}\right) + d(n) \\ &= a^3 T\left(\frac{n}{b^3}\right) + a^2 d\left(\frac{n}{b^2}\right) + ad\left(\frac{n}{b}\right) + d(n) \\ &= \ldots \\ &= a^i T\left(\frac{n}{b^i}\right) + \sum_{j=0}^{i-1} a^j d\left(\frac{n}{b^j}\right) \end{aligned}$$

Si assuma ora che

$$n = b^k$$

da cui

$$k = \log_b n$$

Si avrà pertanto

$$a^k = a^{\log_b n} = n^{\log_b a}$$

In generale, quindi si ha:

$$T(n) = \underbrace{c\, n^{\log_b a}}_{\substack{\text{Soluz.Omogenea:} \\ \text{tempo per risolvere} \\ \text{i sottoproblemi}}} + \underbrace{\sum_{j=0}^{k-1} a^j d(b^{k-j})}_{\substack{\text{Soluz.Particolare:} \\ \text{tempo per combinare} \\ \text{i sottoproblemi}}}$$

Effettuiamo ora delle assunzioni sulla funzione guida $d(\cdot)$ per semplificare l'ultima sommatoria. Assumiamo che $d(\cdot)$ sia *moltiplicativa*, cioè $d(xy) = d(x) \cdot d(y)$ per ogni $x, y \in N$. *Ad esempio* $d(n) = n^p$ *è moltiplicativa, poichè* $d(xy) = (xy)^p = x^p y^p = d(x)d(y)$.

Se $d(\cdot)$ è moltiplicativa, allora ovviamente:

$$d(x^i) = d(x)^i$$

8.3.2 Soluzione Particolare

Definiamo la soluzione particolare $P(n)$ della nostra equazione di ricorrenza come segue:

$$P(n) \;=\; \sum_{j=0}^{k-1} a^j d(b^{k-j})$$

Allora, si avrà

$$P(n) \;=\; d(b^k) \sum_{j=0}^{k-1} a^j d(b^{-j})$$

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

Pertanto:

$$P(n) = \frac{a^k - d(b)^k}{\frac{a}{d(b)} - 1}$$

Abbiamo ora tre possibili casi:

CASO 1 : $a > d(b)$

$$P(n) = \mathcal{O}(a^k) = \mathcal{O}(a^{\log_b n}) = \mathcal{O}(n^{\log_b a})$$

da cui

$$T(n) = \underbrace{c\, n^{\log_b a}}_{sol.omogenea} + \underbrace{\mathcal{O}(n^{\log_b a})}_{sol.particolare} = \mathcal{O}(n^{\log_b a})$$

ovvero

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

Si noti che le soluzioni *particolare* e *omogenea* sono equivalenti (nella notazione $\mathcal{O}()$).

Per migliorare l'algoritmo occorre diminuire $\log_b a$ (ovvero, aumentare b oppure diminuire a). Si noti che diminuire $d(n)$ non aiuta.

CASO 2 : $a < d(b)$

$$P(n) = \frac{d(b)^k - a^k}{1 - \frac{a}{d(b)}} = \mathcal{O}(d(b)^k) = \mathcal{O}(d(b)^{\log_b n}) = \mathcal{O}(n^{\log_b d(b)})$$

da cui

$$T(n) = \underbrace{c\, n^{\log_b a}}_{sol.omogenea} + \underbrace{\mathcal{O}(n^{\log_b d(b)})}_{sol.particolare} = \mathcal{O}(n^{\log_b d(b)})$$

ovvero

$$T(n) \quad = \quad \mathcal{O}(n^{\log_b d(b)})$$

Caso speciale: $d(n) = n^p$

Si ha

$$d(b) = b^p \text{ e } \log_b d(b) = \log_b b^p = p$$

da cui

$$T(n) = \mathcal{O}(n^p)$$

La soluzione particolare eccede la soluzione omogenea. Per migliorare l'algoritmo occorre diminuire $\log_b d(b)$, ovvero cercare un metodo più veloce per fondere le soluzioni dei sottoproblemi.

CASO 3 : $a = d(b)$

La formula per le *serie geometriche* è inappropriata poichè $\frac{a}{d(b)} - 1 = 0$

$$P(n) = d(b)^k \sum_{j=0}^{k-1} \left(\frac{a}{d(b)}\right)^j = d(b)^k \sum_{j=0}^{k-1} 1 = d(b)^k k = d(b)^{\log_b n} \log_b n = n^{\log_b d(b)} \log_b n$$

da cui

$$T(n) = c\, n^{\log_b d(b)} + n^{\log_b d(b)} \log_b n$$

ovvero

$$T(n) \quad = \quad \mathcal{O}(n^{\log_b d(b)} \log_b n)$$

Caso speciale : $d(n) = n^p$

Si ha

$$d(b) = b^p$$

da cui

$$T(n) = \mathcal{O}(n^p \log_b n)$$

La soluzione particolare supera la soluzione omogenea. Per migliorare l'algoritmo occorre diminuire $\log_b d(b)$, ossia cercare un metodo più veloce per fondere le soluzioni dei sottoproblemi.

8.3.3 Esempi

			a	b	$d(b)$
$T(1)$	$=$	c			
$T_1(n)$	$=$	$2T(\frac{n}{2}) + n$	2	2	2
$T_2(n)$	$=$	$4T(\frac{n}{2}) + n$	4	2	2
$T_3(n)$	$=$	$4T(\frac{n}{2}) + n^2$	4	2	4
$T_4(n)$	$=$	$4T(\frac{n}{2}) + n^3$	4	2	8

Si ha, in base alle precedenti considerazioni:

$$\begin{aligned} T_1(n) &= \mathcal{O}(n \log_{\cdot} n) \\ T_2(n) &= \mathcal{O}(n^2) \\ T_3(n) &= \mathcal{O}(n^2 \log_{\cdot} n) \\ T_4(n) &= \mathcal{O}(n^3) \end{aligned}$$

Si noti che $T_1(n)$ rappresenta il tempo di esecuzione del Merge Sort.

Capitolo 9

Programmazione Dinamica

9.1 Introduzione

Spesso, la soluzione di un problema assegnato può essere ricondotta alla soluzione di problemi simili, aventi dimensione minore della dimensione del problema originale. È su tale concetto che si basa la tecnica di progetto nota come Divide et Impera, che abbiamo incontrato nel capitolo 7.

Da un punto di vista della efficienza computazionale, tale tecnica ha generalmente successo se il numero dei sottoproblemi in cui il problema viene suddiviso è costante, ovvero indipendente dalla dimensione dell'input.

Se invece accade che il numero dei sottoproblemi sia funzione della dimensione dell'input, allora non esiste più alcuna garanzia di ottenere un algoritmo efficiente.

La *programmazione dinamica* è una tecnica tabulare molto ingegnosa per affrontare tali situazioni. La risoluzione procede in maniera bottom-up, ovvero dai sottoproblemi più piccoli a quelli di dimensione maggiore, memorizzando i risultati intermedi ottenuti in una tabella.

Il vantaggio di questo metodo é che le soluzioni dei vari sottoproblemi, una volta calcolate, sono memorizzate, e quindi non devono essere più ricalcolate.

9.1.1 Un caso notevole

Si consideri la moltiplicazione di n matrici

$$M = M_1 \cdot M_2 \cdot \ldots \cdot M_n$$

dove M_i é una matrice con r_{i-1} righe e r_i colonne. L'ordine con cui le matrici sono moltiplicate ha un effetto significativo sul numero totale di moltiplicazioni richieste per calcolare M, indipendentemente dall'algoritmo di moltiplicazione utilizzato. Si noti che il calcolo di

$$\begin{matrix} M_i & \cdot & M_{i+1} \\ [r_{i-1} \cdot r_i] & & [r_i \cdot r_{i+1}] \end{matrix}$$

richiede $r_{i-1} \cdot r_i \cdot r_{i+1}$ moltiplicazioni, e la matrice ottenuta ha r_{i-1} righe e $r_i + 1$ colonne. Disposizioni differenti delle parentesi danno luogo ad un numero di moltiplicazioni spesso molto differenti tra loro.

Esempio 21 Consideriamo il prodotto

$$\begin{matrix} M & = & M_1 & \cdot & M_2 & \cdot & M_3 & \cdot & M_4 \\ & & [10 \cdot 20] & & [20 \cdot 50] & & [50 \cdot 1] & & [1 \cdot 100] \end{matrix}$$

Per calcolare

- $M_1 \cdot (M_2 \cdot (M_3 \cdot M_4))$ sono necessari 125000 prodotti;
- $(M_1 \cdot (M_2 \cdot M_3) \cdot M_4)$ sono necessari 2200 prodotti;

Si desidera minimizzare il numero totale di moltiplicazioni. Ciò equivale a tentare tutte le disposizioni valide di parentesi, che sono $\mathcal{O}(2^n)$, ovvero in numero *esponenziale* nella dimensione del problema.

La programmazione dinamica ci permette di ottenere la soluzione cercata in tempo $\mathcal{O}(n^3)$. riassumiamo qui di seguito l'idea alla base di tale metodo.

9.1.2 Descrizione del metodo

Sia

$$m_{ij} \qquad (1 \leq i \leq j \leq n)$$

il costo minimo del calcolo di

$$M_i \cdot M_{i+1} \cdot \ldots \cdot M_j$$

Si ha la seguente equazione di ricorrenza:

$$m_{ij} = \begin{cases} 0 & se\ i = j \\ \min_{i \leq k < j}(m_{ik} + m_{k+1,j} + r_{i-1} \cdot r_k \cdot r_j) & se\ i < j \end{cases}$$

Questo perchè, se calcoliamo $M_i \cdot \ldots \cdot M_j$ come

$$(M_i \cdot \ldots \cdot M_k) \cdot (M_{k+1} \cdot \ldots \cdot M_j)$$

sono necessarie:

- m_{ik} moltiplicazioni per calcolare $M' = M_i \cdot \ldots \cdot M_k$
- $m_{k+1,j}$ moltiplicazioni per calcolare $M'' = M_{k+1} \cdot \ldots \cdot M_j$
- $r_{i-1} \cdot r_k \cdot r_j$ moltiplicazioni per calcolare $M' \cdot M''$:

$$\begin{array}{ccc} M' & \cdot & M'' \\ [r_{i-1} \cdot r_k] & & [r_k \cdot r_j] \end{array}$$

9.1.3 Schema base dell'algoritmo

1. Per l **che va da 0 a** $n-1$
2. **Calcola tutti gli** m_{ij} **tali che** $|j-i| = l$ **e memorizzali**
3. Restituisci m_{1n}

9.1.4 Versione definitiva dell'algoritmo

1. Per i **che va da 0 a** n
2. **Poni** $m_{ii} := 0$
3. Per l **che va da 1 a** $n-1$
4. **Per** i **che va da 1 a** $n-l$
5. **Poni** $j := i + l$
6. **Poni** $m_{ij} := \min_{i \leq k < j}(m_{ik} + m_{k+1,j} + r_{i-1} \cdot r_k \cdot r_j)$
7. Restituisci m_{1n}

Nota bene 8 *Si noti che, l'utilizzo di una tabella per memorizzare i valori m_{ij} via via calcolati, e l'ordine in cui vengono effettuati i calcoli ci garantiscono che al punto (6) tutte le quantità di cui abbiamo bisogno siano già disponibili*

9.1.5 Un esempio svolto

Nell'esempio 21 da noi considerato, otteniamo

$l = 0$	$m_{11} = 0$	$m_{22} = 0$	$m_{33} = 0$	$m_{44} = 0$
$l = 1$	$m_{12} = 10000$	$m_{23} = 1000$	$m_{34} = 5000$	
$l = 2$	$m_{13} = 1200$	$m_{24} = 3000$		
$l = 3$	$m_{14} = 2200$			

Si noti che i calcoli sono eseguiti nel seguente ordine:

	1	2	3	4
	5	6	7	
	8	9		
	10			

Capitolo 10

Le heaps

10.1 Le code con priorità

Una coda con priorità è un tipo di dato astratto simile alla coda, vista in precedenza.

Ricordiamo che la coda è una struttura FIFO (First In First Out), vale a dire, il primo elemento ad entrare è anche il primo ad uscire. Il tipo di dato astratto coda può essere utilizzato per modellizzare molti processi del mondo reale (basti pensare alla coda al semaforo, alla cassa di un supermercato, ecc.). Vi sono molte applicazioni in cui la politica FIFO non è adeguata.

Esempio 22 Si pensi alla coda dei processi in un sistema operativo in attesa della risorsa CPU. La politica FIFO in tale caso non terrebbe conto delle priorità dei singoli processi. In altre parole, nel momento in cui il processo correntemente attivo termina l'esecuzione, il sistema operativo deve cedere il controllo della CPU *non al prossimo processo che ne ha fatto richiesta* bensì al processo *avente priorità più alta tra quelli che ne hanno fatto richiesta.* Assegnamo pertanto a ciascun processo un numero intero p, tale che ad un valore minore di P corrisponda una priorità maggiore. Il sistema operativo dovrà di volta in volta estrarre dal pool dei processi in attesa quello avente priorità maggiore, e cedere ad esso il controllo della CPU.

Per modellizzare tali situazioni introduciamo le code con priorità. Sia U un insieme totalmente ordinato. Una coda con priorità A è un sottinsieme di U su cui sono ammesse le seguenti operazioni:

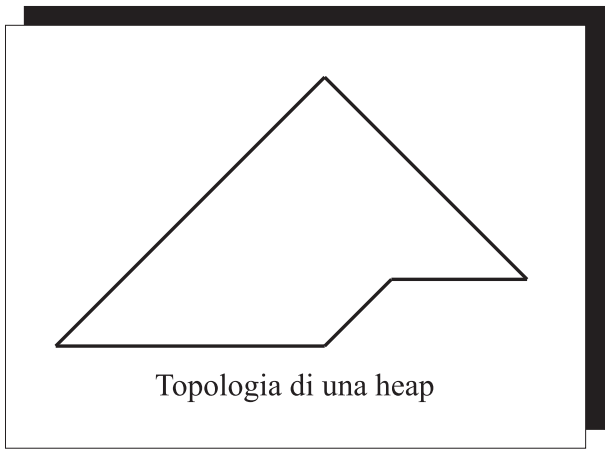
- **Inizializza**(A)
  crea una coda con priorità vuota A, ovvero poni $A = \emptyset$;
- **Inserisci**(x, A)
  inserisci l'elemento x nell'insieme A;
- **CancellaMin**(A)
  cancella il più piccolo elemento di A;
- **Minimo**(A)
  restituisce il più piccolo elemento di A.

Come implementare efficientemente una coda con priorità? La soluzione è fornita dalle heaps.

10.2 Le heaps

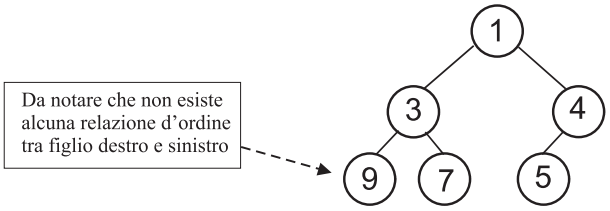
Una heap è un albero binario che gode delle seguenti due proprietà:

1 *Forma:*
Una heap è ottenuta da un albero binario completo eliminando 0 o più foglie. Le eventuali foglie eliminate *sono contigue* e si trovano sulla estrema destra.



Topologia di una heap

2 *Ordinamento relativo:*
La chiave associata a ciascun nodo è minore o uguale delle chiavi associate ai propri figli;



Nota bene 9 *Come conseguenza della* 2^a *proprietà si ha che la radice contiene la chiave più piccola.*

Le altre conseguenze sono dovute alla forma particolare che ha una heap.

1 L'altezza h di una heap avente n nodi è $\mathcal{O}(\log n)$.

2 Una heap può essere memorizzata molto efficientemente in un vettore, senza utilizzare puntatori.

La prima asserzione è semplice da dimostrare: poichè un albero binario completo di altezza h ha $2^{h+1} - 1$ nodi, si ha che $n \geq 2^h - 1$, da cui $h \leq \log(n+1) = \mathcal{O}(\log n)$.

Per quanto riguarda la rappresentazione di una heap, utilizziamo un vettore **Nodi** contenente in ciascuna posizione la chiave associata al nodo. La radice della heap verrà memorizzata nella prima posizione del vettore (ovvero, in posizione 1). Se un nodo si trova in posizione i, allora l'eventuale figlio sinistro sarà memorizzato in posizione $2i$ e l'eventuale figlio destro in posizione $2i+1$. La seguente tabella riassume quanto detto:

NODI	*POSIZIONE*
Radice	1
nodo p	i
padre di p	$\lfloor \frac{i}{2} \rfloor$
figlio sinistro di p	$2i$
figlio destro di p	$2i+1$

Se x è un numero reale, con $\lfloor x \rfloor$ si intende la parte intera inferiore di x.

Si noti che in virtù di tale rappresentazione la foglia che si trova più a destra al livello h, dove h è l'altezza dell'albero, è memorizzata nel vettore **Nodi** in posizione n.

10.3 Ricerca del minimo

In virtù delle proprietà viste precedentemente, sappiamo che il più piccolo elemento di una heap si trova in corrispondenza della radice, che è memorizzata nella prima posizione del vettore. Quindi, la procedura che implementa la ricerca del minimo è banalmente:

Procedura Minimo()
1. Ritorna (Nodi[1])

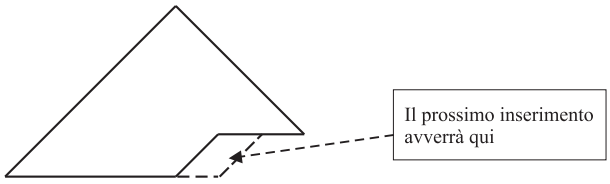
Ovviamente, il tempo richiesto è $\mathcal{O}(1)$.

10.4 Inserimento

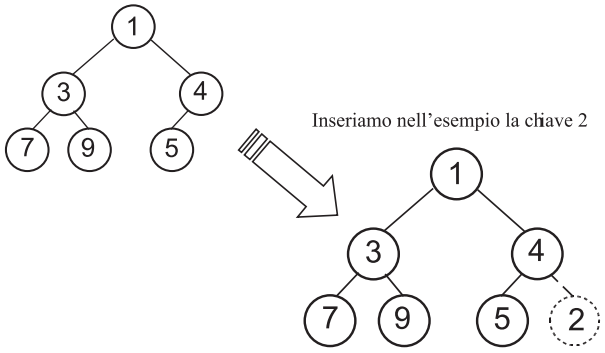
L'inserimento di un nuovo elemento in una heap procede in due fasi:

1 Viene creata innanzitutto una nuova foglia contenente la chiave da inserire. Tale foglia è inserita in posizione tale da rispettare la forma che deve avere una heap.

2 Quindi, la chiave contenuta nella foglia viene fatta salire lungo l'albero (scambiando di volta in volta il nodo che contiene correntemente la chiave con il padre) fino a che la proprietà di ordinamento relativo sia garantita.



Ad esempio:



Per quanto riguarda l'implementazione effettiva, lo pseudo-codice della procedura che implementa l'inserimento è il seguente:

Procedura Inserisci(Chiave):
1. **n := n + 1**
2. **Nodi[n] := Chiave**
3. **SpostaSu(n)**

Procedura SpostaSu(i):
1. **Se** $\lfloor \frac{i}{2} \rfloor \neq 0$
2. **/* Nodi[i] ha un padre */**
3. **Se Nodi[i] < Nodi[** $\lfloor \frac{i}{2} \rfloor$ **]**
4. **Scambia Nodi[i] e Nodi[** $\lfloor \frac{i}{2} \rfloor$ **]**
5. **SpostaSu(**$\lfloor \frac{i}{2} \rfloor$**)**

Il tempo richiesto sarà nel caso peggiore dell'ordine dell'altezza della heap, ovvero $\mathcal{O}(h) = \mathcal{O}(\log n)$.

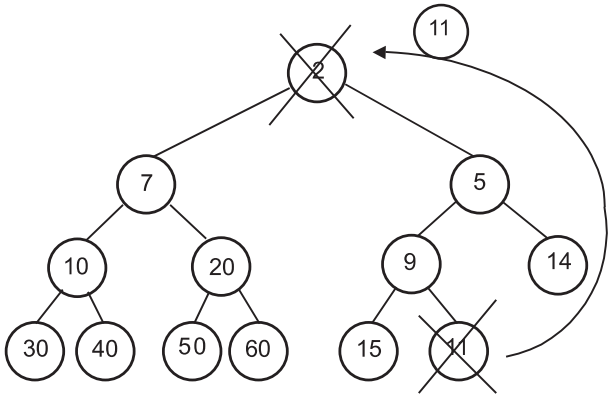
10.5 Cancellazione del minimo

La cancellazione del più piccolo elemento di una heap, che sappiamo trovasi in corrispondenza della radice, procede in due fasi:

1 Viene innanzitutto copiata nella radice la foglia che si trova più a destra al livello h.

2 Tale foglia è quindi rimossa dalla heap. Si noti che in seguito alla rimozione di tale foglia l'albero ottenuto ha ancora la forma di heap.

3 Quindi, la chiave contenuta nella radice viene fatta scendere lungo l'albero (scambiando di volta in volta il nodo che contiene correntemente la chiave con uno dei propri figli) fino a che la proprietà di ordinamento relativo sia garantita.



Per quanto riguarda l'implementazione effettiva, lo pseudo-codice della procedura che implementa l'inserimento è il seguente:

Procedura CancellaMin():
1. Nodi[1] := Nodi[n]
2. n := n - 1
3. SpostaGiu(1)

Procedura SpostaGiu(i):
0. Se 2i $\leq$ n
1. /* Nodi[i] ha almeno un figlio */
2. Sia m l'indice del figlio contenente la chiave più piccola
3. Se Nodi[m] < Nodi[i]
4. Scambia Nodi[m] e Nodi[i]
5. SpostaGiu(m)

Poichè il numero totale di confronti e scambi è al più proporzionale all'altezza della heap, il tempo richiesto sarà ancora una volta $O(\log n)$.

10.6 Costruzione di una heap

Sia S un insieme di n chiavi, tratte da un insieme universo U totalmente ordinato. Si voglia costruire una heap contenente le n chiavi.

Nota bene 10 *Si noti che date n chiavi, possono esistere più heaps contenenti tali chiavi. Ad esempio, se* $S = \{7, 8, 9\}$ *allora esistono esattamente due*

heaps costruite a partire da S: entrambe saranno due alberi binari completi di altezza 1, con la chiave 7 in corrispondenza della radice.

Esistono più strategie per costruire una heap a partire da un insieme S assegnato. La strategia da noi utilizzata si basa sulla seguente proprietà:

Teorema 1 *Sia H una heap avente n nodi, memorizzata in un vettore* **Nodi**. *Allora, i nodi corrispondenti alle ultime i posizioni del vettore (con $1 \leq i \leq n$) formeranno un insieme di alberi, ciascuno dei quali è a sua volta una heap.*

La dimostrazione di tale fatto è banale ed è lasciata allo studente. L'algoritmo da noi utilizzato procede come segue:

- Inizialmente gli elementi di S sono disposti in maniera arbitraria nel vettore **Nodi**.
- Quindi, per i che va da 1 ad n, gli ultimi i elementi del vettore vengono opportunamente permutati per garantire che la proprietà appena enunciata sia valida.

Per quanto riguarda l'implementazione effettiva, lo pseudo-codice della procedura che implementa la costruzione di una heap è il seguente:

Procedura CostruisciHeap (S):
1. **Inserisci arbitrariamente gli elementi di S nel vettore Nodi**
2. **Per i che va da n fino a** 1
3. **SpostaGiu(i)**

Correttezza

La correttezza dell'algoritmo si dimostra per induzione sul numero i di nodi della foresta costituita dagli ultimi i elementi del vettore **Nodi**.

- La foresta costituita dall'ultimo nodo del vettore è banalmente una heap.
- Supponiamo ora che la foresta costituita dagli ultimi $i-1$ elementi del vettore sia costituita da un certo numero di heaps. Allora, quando la procedura **SpostaGiu(i)** è invocata, sono possibili 3 casi:

  1 Il nodo in posizione i non ha figli. In tal caso la procedura ritorna senza fare niente.

2 Il nodo in posizione i ha *un solo figlio*, il sinistro. Si noti che in tal caso il sottoalbero avente radice in i ha due soli nodi! Al termine della procedura il nodo in posizione i conterrà certamente una chiave più piccola della chiave contenuta nel figlio.

3 Il nodo in posizione i ha entrambi i figli. In tal caso, al termine della procedura il sottoalbero avente radice nel nodo i godrà della proprietà di ordinamento relativo di una heap.

Complessità

La costruzione di una heap di n elementi richiede tempo $O(n)$.

Dimostrazione: Indichiamo con $l(v)$ la massima distanza di un nodo v da una foglia discendente di v. Indichiamo poi con n_l il numero di nodi v della heap tali che $l(v) = l$. Si noti che

$$n_l \leq 2^{h-l} = \frac{2^h}{2^l} \leq \frac{n}{2^l}$$

Se il nodo v contenuto nella posizione i-esima del vettore **Nodi** è tale che $l(v) = l$, allora la procedura **SpostaGiu(i)** richiede al più un numero di operazioni elementari (confronti più scambi) proporzionale ad l. Quindi, il tempo totale richiesto sarà dato da:

$$\begin{aligned} T(n) &= \sum_{l=1}^{h} l \cdot n_l \\ &\leq \sum_{l=1}^{h} l \cdot \frac{n}{2^l} \\ &= n \sum_{l-1}^{h} \frac{l}{2^l} \end{aligned}$$

Poichè

$$\sum_{l=1}^{h} \frac{l}{2^l} < \sum_{l=0}^{\infty} \frac{l}{2^l} = \frac{1/2}{(1 - 1/2)^2} = 2$$

si ha che $T(n) < 2n$ ovvero $T(n) = \mathcal{O}(n)$.

10.7 Heapsort

L'Heapsort è un algoritmo di ordinamento ottimale nel caso peggiore, che sfrutta le proprietà delle *heaps* viste in precedenza. Questo algoritmo ordina in loco, cioè utilizza una quantità di spazio pari esattamente alla dimensione dell'input.

Sia S un insieme di elementi da ordinare di dimensione n. L'algoritmo heapsort costruisce dapprima una heap contenente tutti gli elementi di S, quindi estrae ripetutamente dalla heap il piu' piccolo elemento finché la heap risulti vuota.

Heapsort(S):
1. CostruisciHeap(S)
2. Per i **che va da** 1 **a** n
3. **A[i] = Minimo()**
4. **CancellaMin()**

Qual è la complessità di tale algoritmo? Ricordiamo che il passo 1 richiede tempo $O(n)$, il passo 2 tempo costante ed il passo 4 tempo $O(\log n)$ nel caso pessimo. Dalla regola della somma e dalla regola del prodotto si deduce che la complessità nel caso pessimo è $O(n \ \log n)$.

10.8 Esercizio

Modificare le procedure viste in questo capitolo in modo tale che ciascun elemento del vettore contenga altre informazioni oltre alla chiave, ed implementare tali procedure in C.

Capitolo 11

Tecniche Hash

11.1 Introduzione

Il *dizionario* è uno tra i tipi di dati astratti più utilizzati nella pratica.

Gli elementi che costituiscono un dizionario sono detti *chiavi*, e sono tratti da un insieme universo U. Ad ogni chiave è associato un blocco di informazioni. Scopo del dizionario è la memorizzazione di informazioni; le chiavi permettono il reperimento delle informazioni ad esse associate.

Su un dizionario vogliamo poter effettuare fondamentalmente le operazioni di inserimento, ricerca e cancellazione di elementi.

Una delle implementazione migliori dei dizionari si ottiene attraverso le *tecniche Hash,* che consentono di effettuare efficientemente le suddette operazioni, nel caso medio. In particolare dimostreremo che tali operazioni richiedono tempo costante nel caso medio, e pertanto gli algoritmi che implementano tali operazioni su una tabella Hash sono ottimali nel caso medio.

Sfortunatamente, la complessità delle tre operazioni viste sopra è lineare nel numero degli elementi del dizionario, nel caso peggiore.

Le tecniche Hash di organizzazione delle informazioni sono caratterizzate dall'impiego di alcune funzioni, dette *Hashing* (o equivalentemente *scattering*), che hanno lo scopo di disperdere le chiavi appartenenti al nostro universo U all'interno di una tabella T di dimensione finita m, generalmente molto più piccola della cardinalità di U.

Una funzione Hash è una funzione definita nello spazio delle chiavi U ed a valori nell'insieme N dei numeri naturali. Tale funzione accetta in input una chiave e ritorna un intero appartenente all'intervallo $[0, m-1]$.

La funzione Hash deve essere progettata in modo tale da distribuire uniformemente le chiavi nell'intervallo $[0, m-1]$. Il valore intero associato ad una chiave è utilizzato come indice per accedere ad un vettore di dimensione m, detto *Tabella Hash*, contenente le chiavi del nostro dizionario. Le chiavi sono inserite nella tabella utilizzando la funzione Hash per calcolare l'indice.

Quando la funzione Hash ritorna lo stesso indice per due chiavi differenti abbiamo una collisione. Le chiavi che collidono sono dette *sinonimi*. Un algoritmo completo di Hashing consiste di

1 Un metodo per generare indirizzi a partire dalle chiavi, ovvero una *funzione Hash*;

2 Un metodo per trattare il problema delle collisioni, detto *schema di risoluzione delle collisioni*.

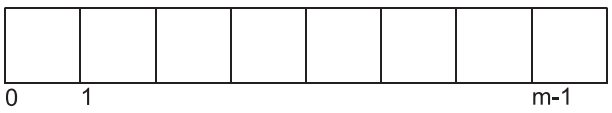
Esistono due classi distinte di schemi di risoluzione delle collisioni:

1 La prima classe è costituita dagli *schemi ad indirizzamento aperto*.
Gli schemi in tale classe risolvono le collisioni calcolando un nuovo indice basato sul valore della chiave - in altre parole effettuano un rehash nella tabella;

2 La seconda classe è costituita dagli *schemi a concatenamento*.
In tali schemi tutte le chiavi che dovrebbbero occupare la stessa posizione nella tabella Hash formano una lista concatenata.

11.2 Caratteristiche delle funzioni Hash

Si noti che una funzione Hash h dovrà consentire di mappare l'insieme universo U sull'insieme $[0, \ldots, m-1]$ degli indici della tabella, che ha generalmente cardinalità molto più piccola.



Una buona funzione Hash h deve essere innanzitutto semplice da calcolare. In altre parole, l'algoritmo che la calcola deve essere un algoritmo *efficiente*.

Inoltre, per ridurre il numero di collisioni, una buona funzione Hash h deve distribuire uniformemente le chiavi all'interno della tabella T. In termini probabilistici, ciò equivale a richiedere che

$$P\left(h(k_1) = h(k_2)\right) \leq 1/m$$

se k_1 e k_2 sono due chiavi estratte a caso dall'universo U.

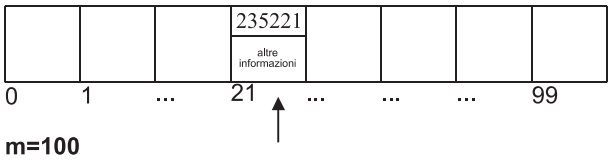
In altri termini, utilizzando una buona funzione *Hash* i sinonimi partizionano l'insieme delle chiavi in sottinsiemi di cardinalità approssimativamente uguale.

11.3 Esempi di funzioni Hash

Se l'universo U è costituito da numeri interi, allora una buona funzione Hash è la funzione riduzione modulo m:

$$h(k) = k \bmod m$$

Esempio 23 Sia $x = 235221,\ m = 100$. Il resto della divisione di 235221 per 100 è 21. Quindi $h(x) = 235221 \bmod 100 = 21$.



Supponendo di considerare la nostra chiave come un numero costituito da una sequenza di cifre in base b, abbiamo in aggiunta le seguenti notevoli funzioni Hash:

1 **Estrazione di k cifre**

$$h(x) = \begin{cases} \text{prime } k \text{ cifre} \\ \text{ultime } k \text{ cifre} \\ k \text{ cifre centrali} \end{cases} \bmod m$$

Esempio 24 Siano $x = 235221,\ k = 3,\ m = 100$. Le prime 3 cifre di x formano il numero 235. Quindi $h(x) = 235 \bmod 100 = 35$.

2 **Cambiamento di Base** (dalla base b alla base B):

$$x = \sum_{i=0}^{t} z_i \; b^i \;\; \Rightarrow \;\; h(X) = \sum_{i=0}^{t} z_i \; B^i \bmod m$$

Per evitare problemi di overflow, si consiglia di far seguire a ciascuna operazione aritmetica (somma o prodotto) una riduzione modulo m.

Esempio 25 Siano $x = 235221,\ b = 10,\ B = 3,\ m = 100$. Allora

$$h(x) = 2 \cdot 3^5 + 3 \cdot 3^4 + 5 \cdot 3^3 + 2 \cdot 3^2 + 2 \cdot 3^1 + 1 \cdot 3^0 \bmod 100$$

3 **Folding**
Si divide la chiave originale in blocchi di cifre di pari lunghezza k e se ne calcola la somma modulo m.

Esempio 26 Siano $x = 235221,\ k = 2,\ m = 100$. Suddividiamo x in blocchi di 2 cifre ed otteniamo $h(x) = 23 + 52 + 21 \bmod 100 = 96$.

Se l'universo U è costituito da **stringhe alfanumeriche**, possiamo considerare tali stringhe come numeri in base b, dove b è la cardinalità del codice utilizzato (ad esempio ASCII). In tal caso, supponendo che la stringa sia composta dai caratteri $s_1, s_2, \ldots, s_k$ possiamo porre:

$$h(k) = \sum_{i=0}^{k-1} b^i s_{k-i} \pmod m$$

Per evitare l'insorgere di problemi di overflow, si consiglia di far seguire la riduzione modulo m a ciascuna operazione aritmetica (somma o prodotto).

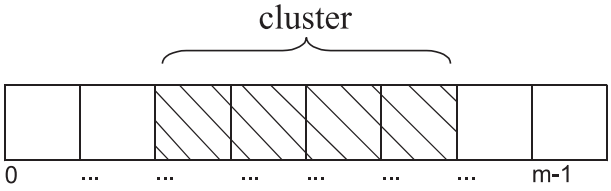
11.4 Schemi ad indirizzamento aperto

Per inserire una chiave utilizzando gli schemi ad indirizzamento aperto occorre scandire la tabella secondo una politica predefinita, alla ricerca di una posizione libera. La sequenza di posizioni scandite è chiamata *percorso*.

La chiave sarà inserita nella prima posizione libera lungo tale percorso, avente origine nella posizione calcolata attraverso la funzione Hash. Ci sono

esattamente $m!$ possibili percorsi, corrispondenti alle $m!$ permutazioni dell'insieme $\{0, \ldots, m-1\}$, e la maggior parte degli schemi ad indirizzamento aperto usa un numero di percorsi di gran lunga inferiore a $m!$ Si noti che a diverse chiavi potrebbe essere associato lo stesso percorso.

La porzione di un percorso interamente occupato da chiavi prende il nome di *catena*. L'effetto indesiderato di avere catene più lunghe di quanto sia atteso è detto *clustering*. Esistono delle tecniche di scansione (scansione uniforme e scansione casuale) che non soffrono del problema del clustering.



Sia m la cardinalità della nostra tabella, ed n il numero delle chiavi in essa memorizzate. La quantità $\alpha = n/m$ è detta *fattore di carico della tabella*. Ovviamente nelle tecniche ad indirizzamento aperto α deve essere minore uguale di uno, mentre non esiste nessuna restrizione sul fattore di carico α se vengono utilizzate delle liste concatenate per gestire le collisioni.

Intuitivamente, l'efficienza di queste tecniche dipende fondamentalmente dal fattore di carico della tabella: in altre parole, quanto più è piena la tabella, tanto più lunga sara' l'operazione di scansione per cercare la chiave assegnata o per cercare una posizione libera in cui inserire una nuova chiave.

11.4.1 Tecniche di scansione della tabella

Qui di seguito elenchiamo alcune delle tecniche di scansione della tabella maggiormente utilizzate negli schemi ad indirizzamento aperto. Ricordiamo che la scansione avviene sempre a partire dalla posizione $h(x)$ calcolata applicando la funzione Hash $h(\cdot)$ alla chiave x. L'obiettivo della scansione è quello di trovare la chiave cercata (nel caso della *ricerca*) oppure una posizione libera in cui inserire una nuova chiave (nel caso dell'*inserimento*).

Scansione lineare

In tale schema la scansione avviene considerando, di volta in volta, la successiva locazione della tabella, modulo m. In altre parole, la scansione della

tabella avviene sequenzialmente, a partire dall'indice $h(x)$, finchè non si raggiunga la fine della tabella e riprendendo, in tal caso, la scansione a partire dall'inizio della tabella. Questo metodo utilizza un solo percorso circolare, di lunghezza m.

Esempio 27 Siano $m = 10,\ h(x) = 3$. Il percorso sarà

$$3, 4, 5, 6, 7, 8, 9, 0, 1, 2$$

La scansione lineare è una delle tecniche più semplici di risoluzione delle collisioni. Purtroppo soffre di un problema, detto *clustering primario*. Quanto più cresce una sequenza di chiavi contigue, tanto più aumenta la probabilità che l'inserimento di una nuova chiave causi una collisione con tale sequenza. Pertanto le sequenze lunghe tendono a crescere più velocemente delle corte.

Tale schema è indesiderabile quando il fattore di carico è alto, ovvero in quelle applicazioni in cui la tabella potrebbe riempirsi quasi completamente.

Hashing doppio

L'Hashing doppio è una tecnica ad indirizzamento aperto che risolve il problema delle collisioni attraverso l'utilizzo di una seconda funzione Hash $h_1(x)$. La seconda funzione Hash è usata per calcolare un valore compreso tra 1 ed $m - 1$ che verrà utilizzato come incremento per effettuare la scansione della tabella a partire dalla posizione calcolata attraverso la prima funzione Hash $h(x)$. Ogni differente valore dell'incremento da' origine ad un percorso distinto, pertanto si hanno esattamente $m - 1$ percorsi circolari.

L'Hashing doppio è pratico ed efficiente. Inoltre, dal momento che l'incremento utilizzato non è costante ma dipende dalla chiave specifica, tale schema non soffre del problema del clustering primario.

Esempio 28 Siano $m = 10,\ h(x) = 3,\ h_1(x) = 2$. Il percorso sarà

$$3, 5, 7, 9, 1$$

Se si desidera garantire che ciascun percorso abbia lunghezza massima, ovvero lunghezza pari ad m, basta prendere m primo.

Esempio 29 Siano $m = 11,\ h(x) = 3,\ h_1(x) = 2$. Il percorso sarà

$$3, 5, 7, 9, 0, 2, 4, 6, 8, 10, 1$$

Hashing quadratico

L'Hashing quadratico è una tecnica di scansione della tabella che utilizza un passo variabile. All'i-esimo tentativo, la posizione scandita sarà data dalla posizione iniziale $h(x)$ più il quadrato di i (chiaramente, il tutto modulo la dimensione della tabella). In altre parole, le posizioni scandite saranno:

$$\begin{array}{lll} h(k) & & (\text{mod } m), \\ h(k) & +1 & (\text{mod } m), \\ h(k) & +4 & (\text{mod } m), \\ h(k) & +9 & (\text{mod } m), \\ h(x) & +\ldots & (\text{mod } m) \end{array}$$

Affinchè il metodo sia efficace, occorre che la dimensione m della tabella sia un numero primo. In tal caso ciascun percorso avrà lunghezza pari esattamente a $\lfloor \frac{m}{2} \rfloor$.

Esempio 30 Siano $m = 11,\ h(x) = 3$. Il percorso sarà

$$3, 4, 7, 1, 8, 6$$

Scansione uniforme

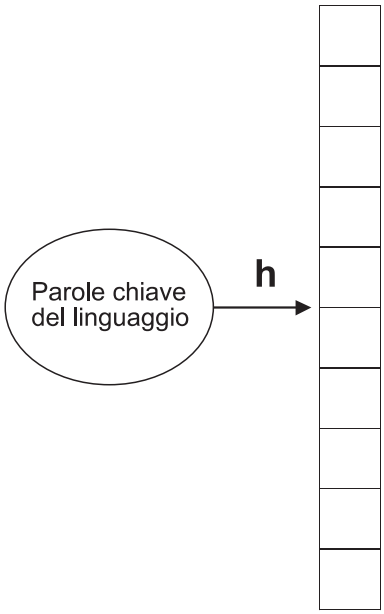
La scansione uniforme è uno schema di indirizzamento aperto che risolve le collisioni scandendo la tabella secondo una permutazione degli interi $[0, \ldots, m-1]$ che dipende unicamente dalla chiave in questione. Quindi, per ogni chiave, l'ordine di scansione della tabella è una permutazione pseudo-casuale delle locazioni della tabella. Questo metodo utilizza con uguale probabilità tutti gli $(m-1)!$ possibili percorsi.

Tale schema va inteso maggiormente come un modello teorico, essendo relativamente semplice da analizzare.

Hashing perfetto

Una funzione Hash è detta perfetta se non genera collisioni. Pertanto è richiesto sempre un solo accesso alla tabella.

Tale funzione Hash è costruita a partire dall'insieme delle chiavi che costituiscono il dizionario, che deve essere noto a priori. Pertanto tale schema è utilizzabile solo se l'insieme delle chiavi è statico (ad esempio le parole chiave di un linguaggio).



11.4.2 Implementazione pratica

Per implementare una tabella Hash, occorre affrontare due poblemi:

1 Trovare una buona funzione HASH.

2 Gestire efficientemente le collisioni.

Indichiamo con $|T| = m$ il numero totale delle chiavi memorizzabili, e con n il numero di chiavi attualmente memorizzate. Ricordiamo che si definisce densità di carico della Tabella il rapporto $\alpha = \frac{n}{m}$. È chiaro che in uno schema ad indirizzamento aperto si avrà necessariamente $0 \leq \alpha \leq 1$.

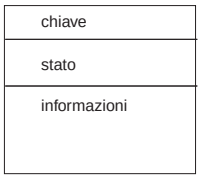
Qui di seguito presentiamo le procedure di inserimento, ricerca e cancellazione di una chiave all'interno di una tabella Hash. L'unica procedura che presenta qualche difficoltà concettuale è la procedura di cancellazione di una chiave. In tal caso occorre modificare opportunamente la tabella per far si che l'algoritmo di ricerca non si arresti, erroneamente, in presenza di una chiave cancellata.

Assumiamo che ogni celletta della nostra tabella Hash contenga almeno i seguenti due campi:

- **chiave**
  utilizzata per identificare la chiave appartenente al dizionario;

- **stato**
  utilizzato per indicare la situazione corrente della celletta.

Chiaramente, in aggiunta a tali campi la celletta potrà contenere tutte le informazioni che desideriamo vengano associate alla chiave:



Il campo **stato** è essenziale per il funzionamento delle procedure di ricerca, inserimento e cancellazione, e può assumere i seguenti valori:

- *Libero*
  se la celletta corrispondente è vuota;

- *Occupato*
  se la celletta corrispondente è occupata da una chiave.

Qui di seguito mostriamo una versione semplificata degli algoritmi fondamentali per gestire una tabella Hash.

Per evitare di appesantire inutilmente la descrizione, abbiamo omesso il caso in cui la scansione ci riporti alla celletta da cui siamo partiti. Ovviamente questa eccezione va gestita opportunamente durante la procedura di inserimento, segnalando in tal caso l'impossibilità di inserire la nuova chiave, e durante la procedura di ricerca, segnalando in tal caso che la chiave cercato non è presente nella tabella.

Inizializzazione

Per inizializzare la tabella, occorre porre il campo *stato* di ciascuna celletta uguale a libero. Ciò chiaramente richiede tempo lineare in m, la dimensione della tabella.

Inserimento

. Sia $h(x)$ la funzione Hash applicata alla chiave x
. Se la celletta di indice $h(x)$ è libera
. allora inserisci x in tale celletta

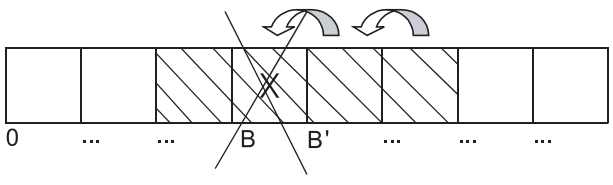
. altrimenti *(la celletta è occupata)*
. scandisci la tabella alla ricerca della prima celletta vuota B
. inserisci la chiave in tale celletta B

Ricerca

. Sia $h(x)$ la funzione Hash applicata alla chiave x
. Se la celletta di indice $h(x)$ è vuota
. allora ritorna non trovato
. Altrimenti
. se la celletta contiene x
. allora ritorna la sua posizione
. altrimenti
. scandisci la tabella finchè
. trovi la chiave cercata x (e ritorna la sua posizione)
. oppure
. trovi una celletta vuota (e ritorna non trovato)

Cancellazione

. Cerca la chiave x
. Sia B la celletta in cui è contenuta la chiave x
. Cancella la celletta B
. Scandisci la tabella a partire dalla celletta B
. per ogni celletta B' incontrata
. se B' contiene una chiave y con $h(y) \leq B$
. allora
. copia y in B,
. cancella la celletta B',
. poni $B := B'$
. finchè incontri una celletta vuota.



11.4.3 Complessità delle operazioni

Si assumano le seguenti ipotesi:

1 Le n chiavi $x_1, \ldots, x_n$ sono selezionate a caso e sono inserite in una tabella Hash di dimensione m (pertanto il fattore di carico α è n/m);

2 h é una buona funzione Hash, ovvero, se x è una chiave estratta a caso dall'universo U, allora la probabilità che $h(x) = y$ é uguale per tutte le celle y:

$$P(h(x) = y) = \frac{1}{m}$$

3 $h(x)$ può essere calcolata in tempo costante;

Definiamo quindi le seguenti quantità:

- U_n := numero medio di passi necessario per cercare una chiave *non presente* nella tabella;
- S_n := numero medio di passi necessari per cercare una chiave *presente* nella tabella.

Si dimostra in tal caso che adottando la scansione lineare risulta

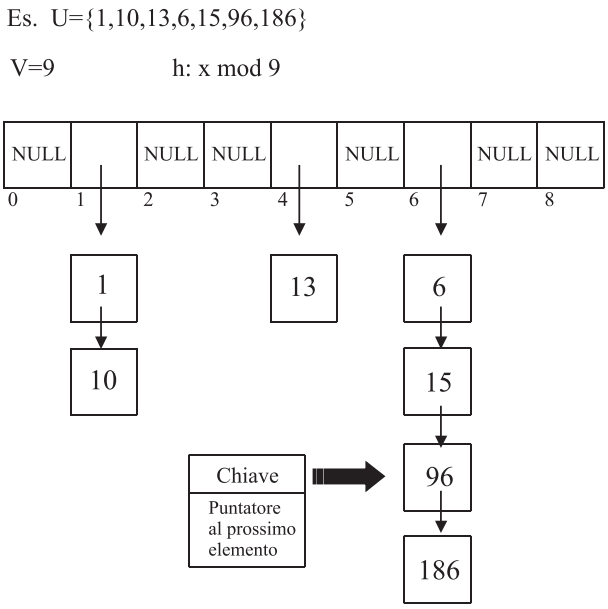
$$U_n = 1 + \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

e

$$S_n = 1 + \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$$

11.5 Tecniche a concatenamento

Tale metodo fa uso di funzioni Hash e liste concatenate come segue. La funzione Hash viene utilizzata per calcolare un indice nella tabella. Tale tabella non contiene le chiavi, bensì puntatori a liste concatenate di chiavi che collidono nella stessa posizione. Pertanto, tale metodo può essere considerato come una combinazione della tecnica Hash con le liste concatenate.



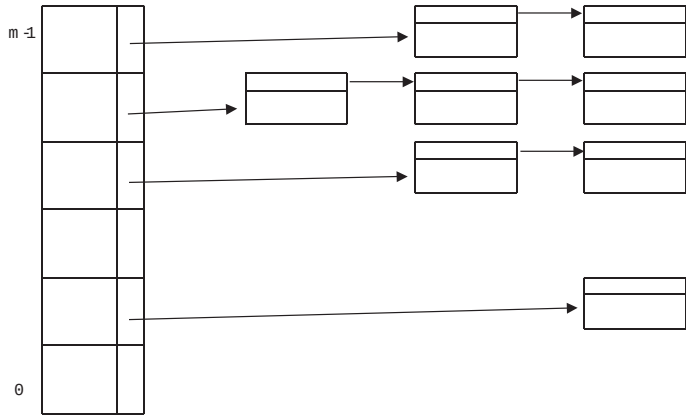
Questa tecnica ha tanti vantaggi rispetto all'indirizzamento aperto:

- Il numero medio di accessi per una singola operazione di ricerca, in caso di successo o insuccesso, è molto basso;
- La cancellazione di una chiave avviene molto efficientemente, semplicemente eliminando tale chiave da una delle liste concatenate;
- Il fattore di carico della tabella può essere maggiore di uno. In altre parole non è necessario conoscere a priori la cardinalità massima del dizionario per poter dimensionare la tabella.

Questo metodo si presta molto bene ad una implementazione utilizzando memoria secondaria. In questo caso il vettore dei puntatori è conservato in memoria centrale, e le liste concatenate nella memoria secondaria.

Hashing attraverso catene separate

In una semplice variante del metodo precedente, nota come *Hashing attraverso catene separate*, la tabella è costituita da record contenenti ciascuno un campo chiave ed un campo puntatore. In caso di collisione viene creata una lista concatenata avente origine nel campo puntatore del record. Un valore convezionale nel campo chiave del record stara' ad indicare l'assenza della chiave corrispondente.



11.5.1 Analisi di complessità

Si assuma come in precedenza che le n chiavi $x_1, \ldots, x_n$ siano selezionate a caso dall'insieme universo U, la tabella Hash abbia dimensione m, ed infine che h sia una buona funzione Hash, calcolabile in tempo costante. Dimostreremo ora che

$$U_n = 1 + \alpha$$

e

$$S_n = 1 + \frac{\alpha}{2}$$

Infatti, se n chiavi sono distribuite uniformemente su m catene, allora la lunghezza media di ciascuna catena è data da n/m, ovvero dal fattore di carico α.

Se la chiave non è presente nella tabella, occorrerà tempo costante per calcolare $h(x)$ più α passi per scandire tutta la catena, da cui $U_n = 1 + \alpha$.

Se la chiave è invece presente nella tabella, occorrerà come al solito tempo costante per calcolare $h(x)$. In più occorreranno in media $\alpha/2$ passi per trovare la chiave all'interno di una catena di lunghezza α. Ne consegue che $S_n = 1 + \alpha/2$.

In definitiva, U_n e S_n sono entrambe $\mathcal{O}(\alpha)$.

In caso di inserimento, occorre prima cercare la chiave e, se non sia già presente, inserirla. Pertanto occorreranno $\mathcal{O}(\alpha) + \mathcal{O}(1) = \mathcal{O}(\alpha)$ passi.

In caso di cancellazione, occorre cercare la chiave e cancellarla nel caso sia presente. Pertanto occorreranno anche in questo caso $\mathcal{O}(\alpha) + \mathcal{O}(1) = \mathcal{O}(\alpha)$ passi.

Ricapitolando, otteniamo i seguenti tempi:

Ricerca con successo	$S_n = 1 + \frac{\alpha}{2} = \mathcal{O}(\alpha)$
Ricerca senza successo	$U_n = 1 + \alpha = \mathcal{O}(\alpha)$
Inserimento	$\mathcal{O}(\alpha)$
Cancellazione	$\mathcal{O}(\alpha)$

Qualora α sia nota a priori, tutte queste operazioni richiedono *in media tempo costante*. Occorre, a tal fine, che l'amministratore del sistema conosca in anticipo (almeno approssimativamente) il massimo numero di chiavi che saranno presenti, per poter dimensionare opportunamente la tabella.

Analisi nel caso pessimo

Nel caso pessimo, tutte le n chiavi appartengono ad una unica catena. Il tempo richiesto dalle operazioni di Ricerca, Inserimento e Cancellazione é, in tal caso, $\mathcal{O}(n)$.

Occupazione di spazio

Per calcolare lo spazio richiesto in uno schema a concatenamento occorre tenere conto dell'occupazione di spazio delle celle che costituiscono la tabella più l'occupazione di spazio di ciascuna catena. Lo spazio richiesto sarà pertanto al più proporzionale a $m + n$.

11.6 Esercizi di ricapitolazione

1 Si assuma una tabella di dimensione $m = 13$. Costruire tutti i possibili percorsi circolari aventi origine nella posizione $h(x) = 0$ ottenuti utilizzando la scansione lineare, l'Hashing doppio e la scansione quadratica.

2 Si svolga l'esercizio precedente con $m = 35$. Cosa si nota? Discutere i risultati.

3 Perchè nell'algoritmo di cancellazione viene effettuato il confronto tra $h(y)$ e B? Discutere i problemi che potrebbero sorgere adottando una implementazione naive dell'algoritmo di cancellazione.

4 Si vuole costruire una tabella Hash per memorizzare i dati anagrafici degli abitanti di un piccolo comune. Dall'analisi dell'andamento demografico negli ultimi anni, si stima che nei prossimi 10 anni il numero di abitanti del comune non dovrebbe superare la soglia di 10000.

Dimensionare opportunamente la tabella in modo da garantire un utilizzo parsimonioso della risorsa spazio, ed al tempo stesso un tempo di ricerca inferiore ad 1 secondo, nel caso medio.

Si assuma che l'accesso alla singola celletta della tabella richieda esattamente un accesso su disco. Si assuma inoltre che ciascun accesso su disco richieda 1 ms. nel caso pessimo.

Discutere accuratamente i risultati adottando schemi ad indirizzamento aperto, schemi a concatenamento diretto e attraverso catene separate. In particolare, assumere che negli schemi a concatenamento diretto il vettore dei puntatori sia mantenuto in memoria centrale, mentre negli schemi che utilizzano catene separate sia la tabella che le catene siano mantenute su disco (memoria secondaria).

11.7 Esercizi avanzati

I seguenti esercizi richiedono qualche conoscenza di algebra, in particolare di teoria dei gruppi.

1 Sia Z_p^* il gruppo degli elementi invertibili dell'anello Z_p delle classi di resto modulo p, dove p è un numero primo. Sia G il sottogruppo moltiplicativo di Z_p^* costituito dagli elementi $\{-1, +1\}$. Si dimostri che l'applicazione che associa ad un elemento a di Z_p^* il proprio quadrato modulo p è un endomorfismo del gruppo Z_p^*. Si dimostri che il kernel di tale omomorfismo ha cardinalità 2, ed è precisamente il sottogruppo G di Z_p^* (suggerimento: in un campo, una equazione di grado k ha al più k soluzioni). Si deduca, utilizzando il primo teorema di omomorfismo, che il numero dei quadrati e dei non quadrati modulo p è identico, ovvero $(p-1)/2$.

2 Dedurre dal precedente esercizio che nell'Hashing quadratico, se utilizziamo un m primo, allora ciascun percorso ha lunghezza pari esattamente a $1 + (m-1)/2$.

3 Verificare sperimentalmente con qualche esempio che se m non è primo, allora le considerazioni fatte negli esercizi precedenti non sono più vere.

4 Dimostrare che il gruppo additivo Z_m è ciclico. Ogni elemento g coprimo con m è un generatore di tale gruppo. Il numero dei generatori di Z_m è $\phi(m)$, dove $\phi(\cdot)$ denota la funzione di Eulero.

Per ogni divisore d di m esiste precisamente un sottogruppo (ciclico) di Z_m di indice d (ovvero di ordine m/d). Tale sottogruppo è generato da un qualunque s tale che $\gcd(m, s) = d$. Il numero di generatori dell'unico sottogruppo di indice d è dato da $\phi(m/d)$. Dedurre la formula di inversione di Moebius:

$$\sum_{d|m} \phi(m/d) = m$$

5 Quale relazione esiste tra i risultati dell'esercizio precedente e la lunghezza dei percorsi nell'Hashing doppio? Per semplicità si considerino i soli percorsi aventi origine da una posizione prefissata, ad esempio la prima.

Capitolo 12

Il BucketSort

Gli algoritmi di ordinamento considerati finora si basano sul modello dei *confronti e scambi*. Le operazioni ammissibili utilizzando questo modello sono:

- Confrontare due chiavi;
- Scambiare i record associati a tali chiavi.

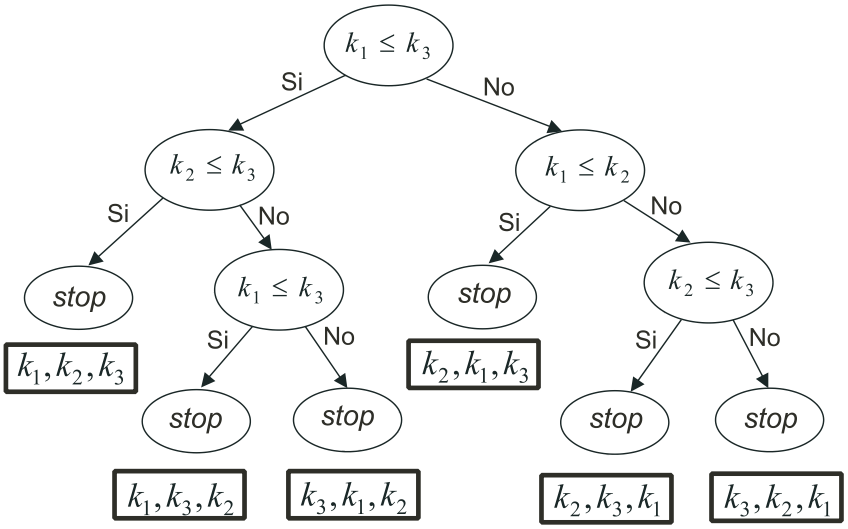
Dimostriamo ora che non puó esistere alcun algoritmo di ordinamento *basato su confronti e scambi* avente complessitá nel caso pessimo inferiore a $\mathcal{O}(n \log n)$. In altre parole, $\Omega(n \log n)$ rappresenta un limite inferiore alla complessitá del problema ordinamento. Poiché il Mergesort ha complessitá $O(n \log n)$, deduciamo che:

- Il Mergesort é ottimale;
- La complessitá del problema ordinamento é $\Theta(n \log n)$.

12.1 Alberi decisionali

Possiamo rappresentare un qualunque algoritmo di ordinamento basato sul modello dei confronti e degli scambi attraverso un *Albero Decisionale*, ovvero un albero binario in cui ciascun nodo interno rappresenta un possibile confronto tra due chiavi, e gli archi uscenti rappresentano l'esito del confronto.

Consideriamo tre chiavi k_1, k_2, k_3; un possibile albero decisionale, che rappresenta un algoritmo di ordinamento basato su confronti e scambi, sarà ad esempio:



Ciascuna foglia dell'albero rappresenta una permutazione dei dati in ingresso. Pertanto, un albero decisionale relativo ad un algoritmo di ordinamento di 3 chiavi avrá $3! = 3 \cdot 2 \cdot 1$ foglie. L'altezza h di tale albero rappresenterá pertanto il numero di confronti che l'algoritmo effettuera' nel caso peggiore. Poiché a ciascun confronto corrisponde al piú uno scambio, é evidente che l'altezza h dell'albero decisionale rappresenta la complessitá dell'algoritmo di ordinamento considerato.

Supponendo di fissare il numero di chiavi n, è possibile trovare una delimitazione inferiore all'altezza di tale albero?

Sia A un algoritmo di ordinamento basato su confronti e scambi. L'albero decisionale associato a tale algoritmo avrá $n!$ foglie. Poiché un albero binario completo ha altezza pari al logaritmo in base 2 del numero delle foglie, si ha che l'altezza h di un albero decisionale avente $n!$ foglie non potra' essere minore di $\log_2 n!$. Poiché

$$\begin{aligned} n! &= n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 3 \cdot 2 \cdot 1 \\ &\geq n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot (\frac{n}{2}) \\ &\geq \left(\frac{n}{2}\right)^{\frac{n}{2}} \end{aligned}$$

si ha che

$$h \geq \log(n!) \geq \log\left(\left(\frac{n}{2}\right)^{\frac{n}{2}}\right) = \frac{n}{2} \cdot \log\left(\frac{n}{2}\right)$$

Quindi un albero decisionale che rappresenta un qualunque algoritmo di ordinamento su n elementi dovra' avere altezza $h \geq \frac{n}{2}\log(\frac{n}{2})$.

12.2 Il Bucketsort

Abbiamo appena visto che non puo' esistere alcun algoritmo di ordinamento avente complessità inferiore a $\mathcal{O}(n\ \log n)$ nel caso pessimo.

Si può fare meglio nel caso medio? Ovviamente, nessun algoritmo di ordinamento potrà effettuare un numero di passi sub-lineare, nel caso medio. In altre parole, poichè occorre considerare ciascun elemento da ordinare almeno una volta all'interno dell'algoritmo, nessun algoritmo potrà in nessun caso avere una complessità inferiore a $\mathcal{O}(n)$. In questo capitolo mostreremo un algoritmo di ordinamento che ha complessità $\mathcal{O}(n)$ nel caso medio, il bucketsort. Tale algoritmo è pertanto ottimale nel caso medio. Nel caso peggiore vedremo che la sua complessità degenera in $\mathcal{O}(n\ \log n)$, e pertanto è la stessa di quella dei migliori algoritmi noti.

Il bucket sort combina l'efficienza nel caso medio delle tecniche hash con l'efficienza nel caso pessimo del mergesort.

12.3 Descrizione dell'algoritmo

Assumiamo come al solito che le nostre chiavi siano tratte da un insieme universo U totalmente ordinato. Per semplificare la trattazione, assumeremo che U sia un sottinsieme dell'insieme N dei numeri naturali.

Occorre che le n chiavi da ordinare siano tutte note a priori; in altre parole il bucket sort è un algoritmo statico, non si presta bene ad ordinare un insieme di chiavi la cui cardinalità possa variare nel tempo.

I passi da effettuare sono i seguenti:

1 In tempo lineare, con una sola scansione dell'insieme S determiniamo (banalmente) il minimo min ed il massimo max elemento di S.

2 Inseriamo gli elementi di S in una tabella hash di dimensione n, che utilizza catene separate (vedi cap. 11), utilizzando la seguente funzione hash:

$$h(x) = \lfloor (n-1)\ \frac{x - min}{max - min} \rfloor \tag{12.1}$$

dove $\lfloor y \rfloor$ denota come al solito la parte intera inferiore di y.

3 Ordiniamo le (al più n) catene utilizzando un algoritmo ottimale nel caso pessimo, ad esempio il mergesort.

4 Inseriamo le (al più n) catene ordinate, sequenzialmente (ovvero, una dopo l'altra), nella lista desiderata.

12.4 Correttezza dell'algoritmo

La correttezza dell'algoritmo discende molto banalmente dalla particolare funzione hash utilizzata.

Si noti che, per come è stata definita, la nostra funzione hash $h(\cdot)$ manda:

- Le chiavi comprese tra

$$min \quad \text{e} \quad min + \frac{max - min}{n}$$

esclusa, nella cella 0. Tali chiavi andranno a costituire quindi la prima catena.

- Le chiavi comprese tra

$$min + \frac{max - min}{n} \quad \text{e} \quad min + 2\,\frac{max - min}{n}$$

esclusa, nella cella 1. Tali chiavi andranno a costituire quindi la seconda catena.

- ...

- Le chiavi comprese tra

$$min + i\,\frac{max - min}{n}$$

e

$$min + (i+1)\,\frac{max - min}{n}$$

esclusa, nella cella i. Tali chiavi andranno a costituire quindi la i-esima catena.

- ...

Occorre, comunque, notare che alcune catene potrebbero essere totalmente assenti. Quindi, ogni chiave presente nella prima catena sarà più piccola di ogni chiave contenuta nella seconda catena, ogni chiave presente nella seconda catena sarà più piccola di ogni chiave contenuta nella terza catena, e cosi' via...

Pertanto è sufficiente ordinare le singole catene e poi concatenarle per ottenere la lista ordinata desiderata.

12.5 Complessità nel caso medio

Poichè abbiamo n records ed una tabella di dimensione n, il fattore di carico della tabella sarà per definizione $\alpha = n/n = 1$. Ricordiamo che, se assumiamo che le n chiavi da ordinare siano equidistribuite all'interno dell'insieme universo U, allora il fattore di carico α rappresenta proprio la lunghezza media di ciascuna catena. Pertanto la lunghezza media di ciascuna catena è *costante*, ovvero indipendente dalla dimensione del problema.

Il calcolo di $h(x)$ per una singola chiave x richiede ovviamente tempo costante (ovvero, occorre effettuare un numero costante di operazioni aritmetiche). Inoltre, l'inserimento di una chiave in una catena richiede anch'esso tempo costante. La prima fase dell'algoritmo, ovvero l'inserimento delle n chiavi, richiede dunque tempo $\mathcal{O}(n)$.

Il costo dell'ordinamento di una singola catena di lunghezza prefissata è costante. Poichè occorre ordinare al più n catene, la seconda fase richiede dunque tempo $\mathcal{O}(n)$.

Occorre, quindi, inserire gli elementi appartenenti alle catene (precedentemente ordinate), sequenzialmente, in una unica lista. Poichè si hanno n elementi, il tempo richiesto, utilizzando ad esempio un vettore per implementare la lista, sarà $\mathcal{O}(n)$.

Dalla regola della somma si deduce quindi che la complessità globale nel caso medio è $\mathcal{O}(n)$, ovvero lineare.

12.6 Complessità nel caso pessimo

Nel caso pessimo tutte le chiavi collidono in una cella i, ovvero

$$h(x) = i \qquad \forall x \in S$$

In tal caso si ha una sola catena di lunghezza n avente origine dalla celletta i-esima. In tal caso, il tempo dominante sarà costituito dal tempo necessario per ordinare le n chiavi che costituiscono l'unica catena. Tale tempo è $\mathcal{O}(n \log n)$ utilizzando un algoritmo ottimale nel caso pessimo, quale il mergesort.

12.7 Esercizi

Negli esercizi che seguono si assuma di avere n elementi da ordinare.

1 Si supponga di utilizzare una tabella di dimensione costante c, anzichè n. Qual è la funzione hash da utilizzare adesso? (Soluzione: sostituire c ad n nella formula (12.1)). Analizzare la complessità del risultante algoritmo nel caso medio e nel caso peggiore.

2 Svolgere il precedente esercizio utilizzando una tabella di dimensione n/c anzichè n, dove c è sempre una costante.

3 Svolgere il precedente esercizio utilizzando una tabella di dimensione $\sqrt{n}$ anzichè n.

4 Svolgere il precedente esercizio utilizzando una tabella di dimensione $\log n$ anzichè n.

Capitolo 13

Selezione in tempo lineare

13.1 Introduzione

Sia A un insieme di n elementi tratti da un insieme universo U totalmente ordinato. Un problema apparentemente correlato al problema ordinamento, ma in effetti distinto da esso, è la selezione del k^{esimo} elemento di A. Prima di affrontare tale problema, occorre dare una definizione rigorosa di k-mo elemento di un insieme, poiche' si verifica facilmente che la nozione intuitiva fallisce se l'insieme ha elementi ripetuti.

Definizione 18 *Sia A un insieme di n elementi tratti da un insieme universo U totalmente ordinato. Il k^{esimo} elemento più piccolo di A, con $1 \leq k \leq n$, è quell'elemento x di A tale che al più $k-1$ elementi di A siano strettamente minori di x ed almeno k elementi di A siano minori o uguali di x.*

In base a tale definizione, l'elemento 8 è il terzo ed il quarto più piccolo elemento dell'insieme $\{8, 8, 0, 5\}$.

Una possibile soluzione al problema consiste nel creare una lista ordinata contenente gli elementi di A, e restituire quindi il k-mo elemento di tale lista. Assumendo, ad esempio, di implementare tale lista attraverso un vettore, avremmo la seguente procedura:

Procedura Seleziona(k, A)
1. Disponi gli elementi di A in un vettore V
2. Ordina V
3. Restituisci V[k]

Tale soluzione richiede tempo $O(n \log n)$ nel caso pessimo. Esiste una soluzione migliore? Ciò equivale a chiedere se la complessità di tale problema sia inferiore a $\Omega(n \log n)$, la complessità del problema ordinamento. La risposta è affermativa!

13.2 Un algoritmo ottimale

L'algoritmo mostrato in tale capitolo ha complessità $O(n)$. Poiché ovviamente nessun algoritmo di selezione potrà avere mai tempo di esecuzione sub-lineare, dovendo prendere in considerazione ciascun elemento dell'insieme almeno una volta, l'algoritmo qui di seguito presentato è ottimale.

Procedura Seleziona(k **,** A **)**
0. Se $|A| < 50$
1. Allora
2. **Disponi gli elementi di** A **in un vettore** V
3. **Ordina** V
4. **Restituisci** $V[k]$
5. altrimenti
6. **Dividi** A **in** $\lceil \frac{|A|}{5} \rceil$ **sequenze, di 5 elementi ciascuna**
7. **Ordina ciascuna sequenza di 5 elementi**
8. **Sia** M **l'insieme delle mediane delle sequenze di 5 elementi**
9. **Poni** $m =$ **Seleziona**$(\lceil \frac{|M|}{2} \rceil, M)$
10. **Costruisci** $A_1 = \{x \in A | x < m\}$
10. **Costruisci** $A_2 = \{x \in A | x = m\}$
10. **Costruisci** $A_3 = \{x \in A | x > m\}$
11. **Se** $|A_1| \geq k$
12. **Allora**
13. **Poni** $x =$ **Seleziona(** k **,** A_1 **)**
13. **Restituisci** x
14. **Altrimenti**
15. **Se** $|A_1| + |A_2| \geq k$
16. **Allora**
17. **Restituisci** m
18. **Altrimenti**
19. **Poni** $x =$ **Seleziona(** $k - |A_1| - |A_2|$ **,** A_3**)**
20. **Restituisci** x

Analisi di correttezza

Si noti che la scelta particolare di m da noi fatta non influenza affatto la correttezza dell'algoritmo. Un qualsiasi altro elemento di A ci garantirebbe certamente la correttezza, ma non il tempo di esecuzione $O(n)$.

Analisi di complessità

Immaginiamo di disporre le $\left\lceil \frac{|A|}{5} \right\rceil$ sequenze ordinate di 5 elementi ciascuna in una tabellina T, una sequenza per ogni colonna, in modo tale che la terza riga della tabellina, costituita dalle mediane delle sequenze, risulti ordinata in maniera crescente. In base a tale disposizione, la mediana m delle mediane occuperà la posizione centrale

$$l = \left\lceil \frac{\left\lceil \frac{|A|}{5} \right\rceil}{2} \right\rceil$$

della terza riga. Si verifica ora facilmente che:

- Un quarto degli elehenti di A sono sicuramente minori o uguali di m. Tali elementi occupano le prime 3 righe delle prime l colonne.
- Un quarto degli elementi di A sono sicuramente maggiori o uguali di m. Tali elementi occupano le ultime 3 righe delle ultime l colonne.

Da ciò deduciamo che:

$$|A_1| \leq \frac{3}{4}\,|A| \qquad |A_3| \leq \frac{3}{4}\,|A|$$

Questa stima ci permette di analizzare il tempo di esecuzione dell'algoritmo nel caso peggiore.

Notiamo subito che se $n = |A| < 50$ il tempo richiesto $T(n)$ è costante. Se $|A| \geq 50$ invece occorre:

- Costruire l'insieme M, e ciò richiede tempo lineare;
- Selezionare m, la mediana delle mediane, e ciò richiede tempo $T(n/5)$;
- Costruire i tre sottinsiemi $A_1,\ A_0,\ A_3$ di A, e ciò richiede tempo lineare;

- Invocare ricorsivamente l'algoritmo su un insieme di cardinalità pari ad al più tre quarzi della cardinalità di A (ricordiamo che la nostra analisi è relativa al caso peggiore).

Da ciò deduciamo la seguente relazione di ricorrenza:

$$\begin{cases} T(n) &= \mathcal{O}(1) & \text{se} \quad n < 50 \\ T(n) &\leq T(\frac{n}{5}) + T(\frac{3}{4}\ n) + c\ n & \text{altrimenti} \end{cases}$$

Dimostriamo ora, per induzione su n, che $T(n) \leq 20cn = \mathcal{O}(n)$.

L'asserzione è sicuramente vera se $n < 54$; infatti in tal caso il tempo richiesto è costante, ovvero indipendente da n.

Supponiamo ora che l'asserzione sia vera per tutte le dimensioni dell'input inferiori ad n. Vogliamo dimostrare che l'asserzione sia vera anche per n. Si ha infatti:

$$\begin{aligned} T(n) &\leq T\left(\frac{n}{5}\right) + T\left(\frac{3}{4}\ n\right) + c\ n \\ &\leq 20\ c\ \frac{n}{5} + 20\ c\ \frac{3}{4}\ n + c\ n \\ &= \left(\frac{20}{5} + \frac{20 \cdot 3}{4} + 7\right)\ c\ n \\ &= \frac{85 + 300 + 20}{20}\ c\ n \\ &= 30\ c\ n \end{aligned}$$

come volevasi dimostrare.

Appendice A

Strutture dati

In questa appendice vengono richiamati alcuni concetti fondamentali riguardanti le strutture dati che vengono considerate nel volume.

A.1 Richiami sui puntatori

Ricordiamo che un *puntatore* non e' altro che un riferimento ad una zona di memoria, ovvero una celletta che contiene l'indirizzo di un'altra celletta (o se si preferisce, una variabile che contiene l'indirizzo di un'altra variabile).

Ricordiamo che è possibile creare strutture dinamiche con i puntatori sfruttando i meccanismi di *allocazione* e *deallocazione* dinamica della memoria offerti dal particolare ambiente di sviluppo utilizzato.

- Allocare significa riservare un area di memoria da utilizzare in futuro. Tale area di memoria è nota attraverso l'indirizzo iniziale e la dimensione;
- Deallocare significa liberare la suddetta area di memoria.

Per allocare dinamicamente un'area di memoria, il linguaggio C offre la funzione $malloc(k)$; k indica il numero di bytes che si desidera allocare, ovvero riservare, in una particolare zona di memoria nota come *heap*. Tale funzione restituisce il puntatore alla zona di memoria riservata.

In C++ occorre dichiarare una variabile puntatore P ad un oggetto di tipo A, utilizzando la forma

```
A* P;
```

Inizialmente il puntatore P contiene un valore indefinito. Volendo creare un oggetto dinamico di tipo A occorre allocare la memoria necessaria usando l'operatore *new*

```
P = new A;
```

In tal modo riserviamo la quantità di memoria necessaria a contenere un oggetto di tipo A ed assegnamo l'indirizzo base a P. Usiamo quindi l'espressione $*P$ per riferirci all'oggetto di tipo A appena creato (dinamicamente).

Per deallocare la memoria in $C++$ si usa il comando *delete*:

```
delete P
```

In tal modo viene liberata l'area di memoria occupata dall'oggetto puntato da P.

Lo svantaggio dell'uso dei puntatori è nella produzione di spazzatura (*garbage*). In altre parole, se la memoria allocata e non più utilizzata non viene liberata, si arriverà al punto in cui lo spazio c'è, ma non è disponibile.

A.2 Il tipo di dato astratto LISTA

Ricordiamo che *tipo di dato astratto* è la rappresentazione di una entita' del mondo reale attraverso un modello matematico ed un insieme di operatori che agiscono sul modello stesso. Il modello matematico è implementato utilizzando una struttura dati; si avrà inoltre un algoritmo per ciascun operatore relativo al nostro tipo di dato astratto.

Due tipi di dati astratti sono identici se sono identici i modelli sottostanti e gli operatori corrispondenti.

Il tipo di dati astratto più semplice che prendiamo in considerazione è la *LISTA*: da un punto di vista matematico il modello sottostante consiste di una sequenza di n oggetti $(a_1, a_2, \ldots, a_n)$ appartenenti ad un insieme prefissato S.

Sia $L = (a_1, a_2, \ldots, a_n)$ una lista. La lunghezza di L è per definizione n, ovvero il numero dei suoi elementi. Se la lista è vuota, cioè non contiene alcun

elemento, allora $n = 0$. Ogni elemento è caratterizzato da una posizione in L, indicata con i e da un valore a_i.

Data una lista L è possibile aggiungere o togliere elementi da essa. Per fare questo, occorre specificare la posizione relativa all'interno della sequenza nella quale il nuovo elemento va aggiunto o dalla quale il vecchio elemento va tolto. Nella tabella che segue indichiamo le operazioni definite sul tipo di dato astratto LISTA.

Operatore	*Operazione*
crea_lista()	crea una lista vuota L e la restituisce
cercaChiave(e,L)	cerca l'elemento e (appartenente ad S) nella lista L e restituisce l'indice compreso tra 1 e la dimensione corrente della lista nel caso in cui il valore cercato sia stato trovato. In caso contrario restituisce 0
cercaPosizione(i, L)	restituisce l'elemento in posizione i
inserisci(e, L, i)	inserisce e nella posizione i della lista L
cancellaPosizione(i, L)	cancella l'elemento in posizione i
cancellaChiave(e, L)	cancella l'elemento avente chiave e
primo_elemento(L)	restituisce il primo elemento della lista L
ultimo_elemento(L)	restituisce l'ultimo elemento della lista L
stampa(L)	stampa l'intera lista L

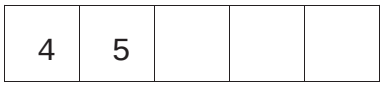
Per valutare la complessità degli operatori sopra definiti, occorre considerare la struttura di dati sottostante, che implementa il nostro modello matematico.

La più semplice implementazione di una lista richiede un vettore V di dimensione m prefissata, in cui l'elemento a_i occupa la posizione i-esima. Per tener traccia della dimensione corrente della lista utilizziamo una variabile *contatore* che assume inizialmente il valore 0, e viene incrementata ad ogni operazione di inserimento e decrementata ad ogni ogni operazione di cancellazione.

Poichè la dimensione m del vettore deve essere prestabilita, tale implementazione non consente di rappresentare liste contenenti piu' di m elementi, ovvero non è dinamica.

Operazione di Inserimento

Supponiamo che il vettore contenga gli elementi 4 e 5.



La variabile *contatore* conterra' il valore 2. Supponiamo di voler inserire nella seconda posizione il valore 10; in seguito a tale operazione la variabile *contatore* conterrà il valore 3, ed otterremo la situazione raffigurata di seguito:

Per cercare un elemento, nota la chiave e, occorre scandire tutto il vettore; nel caso pessmo tale operazione richiede quindi tempo lineare in n. Si noti che il caso pessimo per l'operazione di ricerca si verifica nel momento in cui l'elemento da cancellare non è presente affatto nella lista!

L'operazione di inserimento di un elemento a_i nel vettore V richiede nel caso pessimo tempo lineare in n, poichè occorre spostare a destra di una posizione tutti gli elementi aventi indice maggiore o uguale ad i prima di poter inserire a_i.

Analogamente, l'operazione di cancellazione di un elemento a_i richiede nel caso pessimo tempo lineare in n, poichè occorre spostare a sinistra di una posizione tutti gli elementi aventi indice maggiore di i.

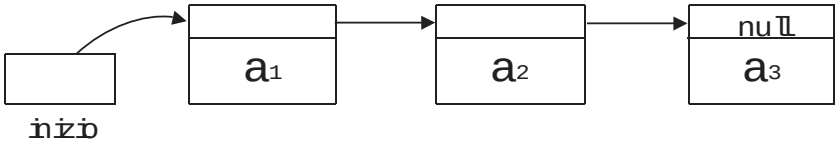
Il vantaggio principale della implementazione di una lista attraverso un vettore risiede nella sua semplicità.

Per ovviare al vincolo della staticita' ricorriamo alle implementazioni mediante puntatori e doppi puntatori.

A.2.1 Implementazione mediante puntatori

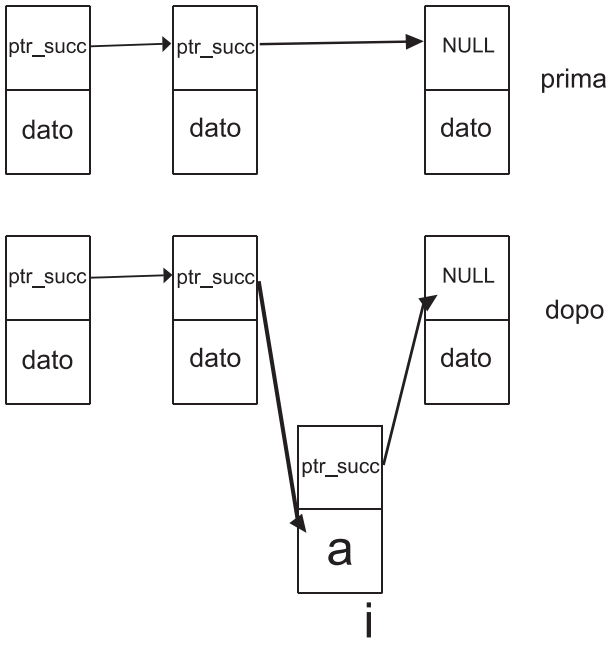
È possibile implementare una lista di n elementi con n record, tali che l'i-esimo record contenga l'elemento a_i della lista (in un campo denominato *chiave*) e l'indirizzo del record contenente la chiave a_{i+1} (in un campo denominato *successivo*). Utilizzando tale implementazione la scansione sequenziale della lista risulta molto semplice. Una lista implementata in questo modo è detta *Lista Concatenata*.

Il primo record è indirizzato da una variabile *Testa* di tipo puntatore. Il campo *successivo* dell'ultimo record contiene un valore convenzionale *null*.



Per cercare una chiave nella lista, occorre scandire la lista utilizzando i puntatori per ottenere di volta in volta l'indirizzo del record successivo. Pertanto nel caso pessimo la ricerca di una chiave richiede tempo lineare nella dimensione della lista.

Per inserire una chiave a nella lista, in posizione i, occorre innanzitutto creare (ovvero, allocare dinamicamente) un nuovo record R ed assegnare al campo chiave di R il valore a. Quindi occorre scandire la lista sino a giungere al record i-esimo, assegnare al campo *successivo* del record precedente l'indirizzo del record R, ed infine assegnare al campo *successivo* di R l'indirizzo del record i-esimo. Poichè nel caso peggiore occorre scandire l'intera lista l'operazione di inserimento richiede tempo lineare nella dimensione corrente n della lista.



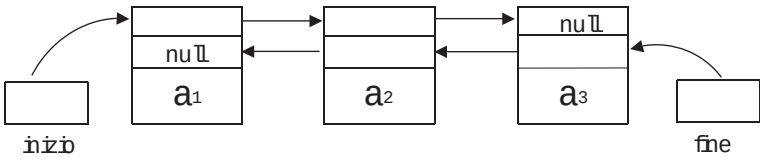
Per cancellare un record, occorre copiare l'indirizzo del record successivo nel campo *successivo* del record precedente e quindi deallocare il record da cancellare.

Si noti che l'operazione di cancellazione richiede tempo costante qualora sia noto l'indirizzo del record che precede il record da cancellare.

Il vantaggio di questo tipo di implementazione è dato dal fatto che non vi sono vincoli sulla dimensione della lista (l'unico vincolo è dato dalla quantita' di memoria disponibile).

A.2.2 Implementazione mediante doppi puntatori

L'implementazione con puntatori appena considerata può essere modificata leggermente introducendo in ogni cella un puntatore al record che contiene l'elemento precedente. Otteniamo così una lista doppiamente concatenata. Questa implementazione presenta il vantaggio di poter accedere all'elemento contenente la chiave a_{i-1}, qualora sia noto l'indirizzo del record contenente la chiave a_i, in tempo costante.



A.3 Il tipo di dato astratto LISTA ORDINATA

Una lista ordinata L è una sequenza di n elementi $(a_1, a_2, \ldots, a_n)$ tratti da un insieme totalmente ordinato S, tale che $a_i \leq a_{i+1} \quad \forall i \in \{1, \ldots, n-1\}$.

Si noti che a causa della proprieta' di ordinamento non è possibile l'inserimento di un elemento in posizione arbitraria.

Per implementare una lista ordinata si utilizza comunemente un vettore V ed una variabile *contatore* che indica la dimensione corrente della lista. Le chiavi sono memorizzate nel vettore in posizioni contigue e in ordine crescente a partire dalla prima posizione.

L'operazione base per questo tipo di dato astratto è la ricerca per chiave, che avviene attraverso un procedimento dicotomico (ovvero suddividendo ricorsivamente in due parti uguali la lista), e richiede tempo logaritmico nella dimensione della lista.

Per verificare l'appartenenza di una chiave k, si confronta il valore da ricercare k con il valore v che occupa la posizione centrale della porzione di

vettore effettivamente utilizzata. Se $k = v$, allora l'elemento è stato trovato, mentre se $k < v$ (risp. $k > v$) allora l'elemento va ricercato nella prima (risp. seconda) metà del vettore, che contiene solo elementi minori (maggiori) di k, senza più considerare l'altra metà dell'insieme. Questo procedimento viene riapplicato ricorsivamente sulla metà del vettore così determinata, fino ad individuare k o a stabilire che k non appartiene alla lista.

Si verifica facilmente che il numero di confronti tra la chiave k e le chiavi presenti nel vettore che sono eseguiti dalla ricerca binaria è, nel caso pessimo, logaritmico nel numero delle chiavi presenti nella lista.

L'operazione di inserimento consiste di quattro fasi:

- Ricerca dell'elemento contenente la chiave assegnata. Tale operazione restituisce un indice i che rappresenta la posizione che la chiave dovrebbe occupare;
- Shift verso destra degli elementi di indice maggiore di i;
- Inserimento del nuovo elemento in posizione i;
- Incremento della variabile *contatore*.

L'operazione di shift verso destra degli elementi di indice maggiore di i richiede tempo lineare nella dimensione della lista, nel caso pessimo. Il costo di tale operazione domina gli altri costi. Pertanto possiamo asserire che l'operazione di inserimento richiede nel caso pessimo tempo lineare nella dimensione n della lista.

L'operazione di cancellazione consiste di tre fasi:

- Ricerca dell'elemento contenente la chiave assegnata. Tale operazione restituisce un indice i che rappresenta la posizione che la chiave occupa;
- Shift verso sinistra degli elementi di indice maggiore di i;
- Decremento della variabile *contatore*.

Analogamente all'inserimento, possiamo asserire che l'operazione di cancellazione richiede nel caso pessimo tempo lineare nella dimensione n della lista.

A.4 Il tipo di dato astratto PILA

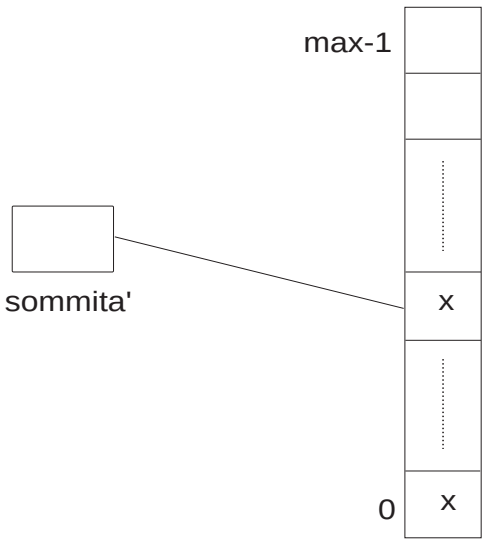
Una pila è una sequenza di elementi di un tipo prefissato, sulla quale sono possibili esclusivamente due operazioni:

- l'inserimento di un elemento ad una estremita', detta *sommitá*;
- l'estrazione dell'elemento presente nella *sommitá*.

Le due operazioni vengono generalmente denominate PUSH e POP. Quindi, a differenza delle liste, non è possibile accedere direttamente ad un elemento presente in posizione arbitraria. Questo tipo di dato può essere visto come una specializzazione della LISTA, in cui l'ultimo elemento inserito è anche il primo ad essere estratto. Tale meccanismo è noto come LIFO (Last In First Out).

A.4.1 Implementazione mediante vettore

L'implementazione piu' semplice di una pila richiede un vettore di dimensione prefissata max, in cui memorizziamo gli elementi, ed una variabile che contiene l'indice della *sommita'* della pila.



Inizialmente la variabile *sommita'* contiene il valore -1, indicante la condizione di pila vuota. Per effettuare l'estrazione salviamo in una variabile

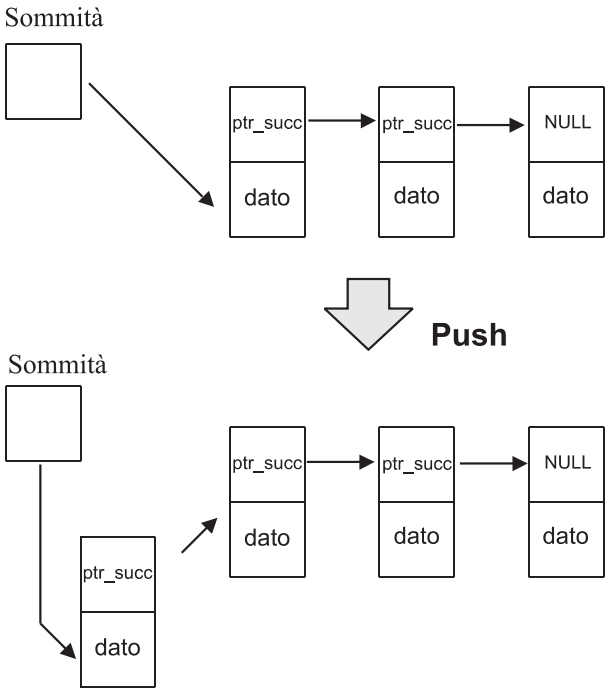
temporanea la celletta del vettore di indice *sommita'* e decrementiamo di una unita' la variabile *sommita'*. Allo stesso modo, per inserire un elemento incrementiamo di una unita' la variabile *sommita'*, e salviamo l'elemento nella celletta del vettore di indice *sommita'*. In tal modo la pila occupa sempre posizioni consecutive del vettore. Con questa implementazione le operazioni di inserimento ed estrazione richiedono ovviamente tempo costante.

Si noti che se $sommita' = max - 1$ la pila è piena, ovvero non è possibile inserire nuovi elementi. Per ovviare a tale inconveniente, possiamo implementare la pila attraverso una lista concatenata, come di seguito descritto.

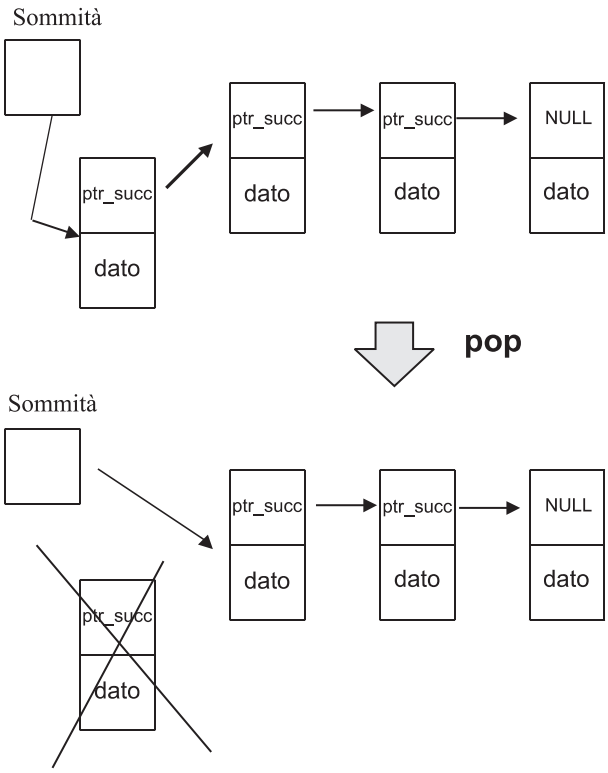
A.4.2 Implementazione mediante puntatori

Ricorriamo ad una lista concatenata e ad una variabile di tipo puntatore, denominata *sommitá* che viene utilizzata per riferirsi all'ultimo elemento inserito nella pila. Tale puntatore contiene il valore convenzionale *null* se la pila è vuota.

L'operazione di inserimento (PUSH) modifica la lista come segue:



L'operazione di estrazione (POP) modifica la lista come segue:



L'implementazione della pila attraverso una lista concatenata è caratterizzata da tempi costanti per l'inserimento e per l'estrazione. A differenza della precedente implementazione, l'unico limite è costituito dalla quantita' di memoria disponibile nel sistema.

A.5 Il tipo di dato astratto CODA

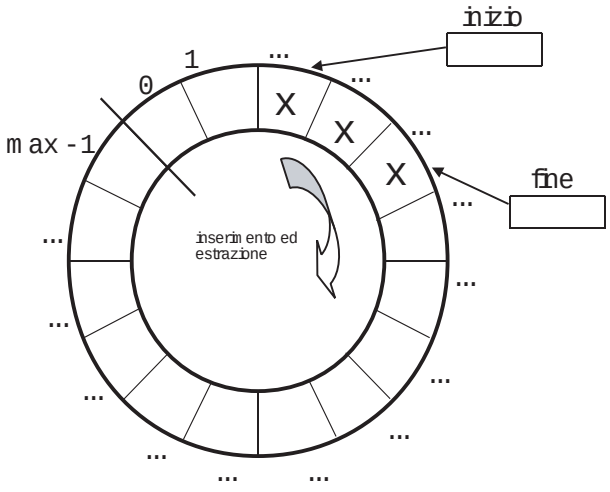
Una coda è una sequenza di elementi di un tipo prefissato, sulla quale sono possibili esclusivamente due operazioni: l'inserimento di un elemento ad una estremita', detta *fine*, e l'estrazione dell'elemento presente nell'altra estremita', detta *inizio*. Quindi, a differenza delle liste, non è possibile accedere direttamente ad un elemento presente in posizione arbitraria. Questo tipo di dato può essere visto come una specializzazione della LISTA, in cui il primo elemento inserito è anche il primo ad essere estratto. Tale meccanismo è noto come FIFO (First In First Out).

A.5.1 Implementazione mediante vettore circolare

Un *vettore circolare* è formalmente un vettore di dimensione infinita che viene mappato su un vettore di dimensione finita.

Supponiamo di avere un vettore di max elementi, di indice compreso tra 0 e $max - 1$, in cui consideriamo formalmente l'elemento di indice 0 come successore di quello di indice $max - 1$. Ricorriamo a due variabili di tipo intero, denominate *inizio* e *fine* che vengono utlizzate per riferirsi rispettivamente all'indice del primo elemento ed all'indice dell'ultimo elemento presente in coda.

La coda è contenuta in posizioni consecutive del vettore. Per effettuare l'estrazione salviamo in una variabile temporanea la celletta del vettore di indice *inizio* ed incrementiamo di una unita' (modulo max) la variabile *inizio*. Allo stesso modo, per inserire un elemento incrementiamo di una unita' (modulo max) la variabile *fine*, e salviamo l'elemento nella celletta del vettore di indice *fine*. In tal modo la coda migra lungo il vettore circolare e, ad ogni istante, si trova sempre in posizioni consecutive (modulo max) a partire dall'indice *inizio* del vettore.



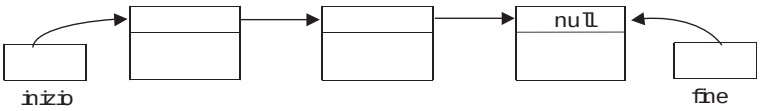
Si noti che l'incremento delle variabili *inizio* e $fine$ va effettuato modulo max; in altre parole, se $inizio < max - 1$ allora poniamo $inizio = inizio + 1$, altrimenti poniamo $inizio = 0$.

Si noti che se $fine = inizio - 1$ allora la coda ha una lunghezza effettiva pari alla dimensione del vettore; in tal caso non è possibile inserire nuovi

elementi. Per ovviare a tale inconveniente, possiamo implementare la coda attraverso una lista concatenata, come di seguito descritto.

A.5.2 Implementazione mediante puntatori

Ricorriamo ad una lista concatenata e a due variabili di tipo puntatore, denominate *inizio* e *fine* che vengono utilizzate per riferirsi rispettivamente al primo ed all'ultimo elemento della coda. I due puntatori contengono il valore convenzionale *null* se la coda è vuota.



L'implementazione della coda attraverso una lista concatenata è caratterizzata da tempi costanti per l'inserimento e per l'estrazione.

A.6 Grafi

Un *grafo orientato* o *grafo diretto*, è una coppia $G = (V, E)$, dove

- V è un insieme finito, non vuoto, di elementi detti nodi o vertici;
- E è un sottinsieme di $\mathsf{V} \times \mathsf{V}$, ovvero un insieme finito di coppie ordinate di nodi, detti archi.

I grafi orientati possono essere rappresentati graficamente indicando i nodi con dei cerchietti e ogni arco (i, j) con una freccia che esce dal nodo i ed entra nel nodo j.

Se indichiamo con n il numero di nodi e con e il numero di archi avremo $0 \leq e \leq n^2$ poichè possono esserci al più n archi uscenti da ciascun nodo.

In generale, i nodi sono usati per rappresentare oggetti e gli archi per rappresentare relazioni tra coppie di oggetti.

In un grafo orientato G, un *cammino diretto* è una sequenza di nodi $v_0, v_1, \ldots, v_k$ tali che $(v_i, v_{i+1}) \in E$, per $i = 0, \ldots, k-1$. Il cammino parte dal nodo v_0, attraversa i nodi $v_1, \ldots, v_{k-1}$, arriva al nodo v_k, ed ha lunghezza uguale a k. Se non ci sono nodi ripetuti, cioè $v_i \neq v_j$ per $0 \leq i < j \leq k$, allora il cammino si dice *semplice*, mentre se $v_0 = v_k$ si dice *chiuso*. Un cammino semplice e chiuso è detto *ciclo*.

In un grafo orientato G, un *cammino non diretto* è una sequenza di nodi $v_0, v_1, \ldots, v_k$ tali che $(v_i, v_{i+1}) \in E$ oppure $(v_{i+1}, v_i) \in E$ per $i = 0, \ldots, k-1$.

Un grafo orientato si dice *connesso* se, per ogni coppia di nodi v e w, esiste almeno un cammino non diretto da v a w.

Un *grafo non orientato* è una coppia $G = (V, E)$, dove

- V è un insieme finito, non vuoto, di elementi detti nodi o vertici;
- E è un un insieme finito di coppie non ordinate $[i, j]$ di nodi distinti.

Quindi per definizione $[i, j]$ e $[j, i]$ indicano lo stesso arco, che è detto *incidere* sui due nodi, i quali sono *adiacenti* tra loro. I grafi non orientati sono usati per rappresentare relazioni simmetriche tra oggetti.

In un grafo non orientato G, una *catena* è una sequenza di nodi $v_0, v_1, \ldots, v_k$ tali che

$$[v_i, v_{i+1}] \in E, \quad \text{per} \quad i = 0, 1, \ldots, k-1.$$

La catena collega v_0 con v_k, o viceversa, ed ha lunghezza uguale a k. Se non ci sono nodi ripetuti, cioè $v_i \neq v_j$ per $0 < i < j < k$, allora la catena è *semplice*, altrimenti , se $v_0 = v_k$, è *chiusa*. Una catena, semplice o chiusa che sia, i cui archi sono tutti distinti, è un *circuito*.

Un grafo non orientato è connesso se per ogni coppia di nodi v e w esiste una catena tra v e w.

Definizione 19 *Dato un grafo orientato G, il grado entrante di un nodo v è dato dal numero di archi che entrano in quel nodo, e si indica con* **Indegree(v).**

$$Indegree(v) = |\{(w, v) \quad con \quad w \in V\}|$$

Definizione 20 *Dato un grafo orientato G, il grado uscente di un nodo v è dato dal numero di archi uscenti dal nodo, e si indica con* **Outdegree(v).**

$$Outdegree(v) = |\{(v, w) \quad con \quad w \in V\}|$$

Definizione 21 *Il grado Degree(v) di un nodo v è dato dalla somma del grado entrante e del grado uscente di v, e si indica con* **Degree(v).**

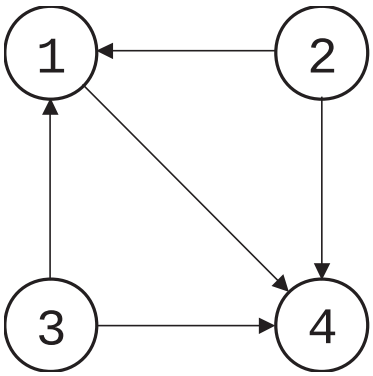
$$Degree(v) = Indegree(v) + Outdegree(v)$$

Definizione 22 *Il grado di un grafo G è il massimo dei gradi dei suoi nodi, e si indica con* **Degree(G).**

A.6.1 Rappresentazione in memoria

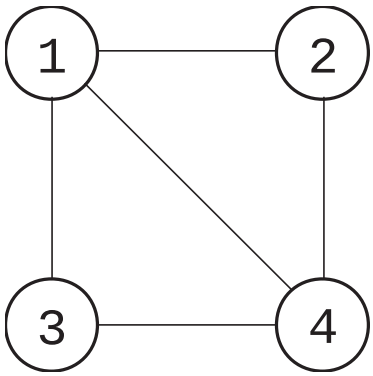
Diverse strutture dati possono essere usate per rappresentare un grafo in memoria. La scelta appropriata dipende dalle operazioni che si vogliono svolgere sui nodi e sugli archi del grafo.

Il metodo più semplice per rappresentare un grafo in memoria è attraverso una *matrice di adiacenza*. Se il grafo ha n vertici, etichettiamo ogni vertice assegnandogli un numero intero distinto compreso tra 1 ed n. Associamo quindi al grafo $G = (V, E)$ la matrice $A[i, j]$ quadrata di ordine n, tale che $A[i, j] = 1$ se $(i, j) \in E$ e $A[i, j] = 0$ se $(i, j) \notin E$.



0	0	0	1
1	0	0	1
1	0	0	1
0	0	0	0

Se il grafo è non orientato occorre rendere tale matrice simmetrica, ovvero occorre porre $A[i, j] = A[j, i]$ per ogni coppia di vertici i e j.

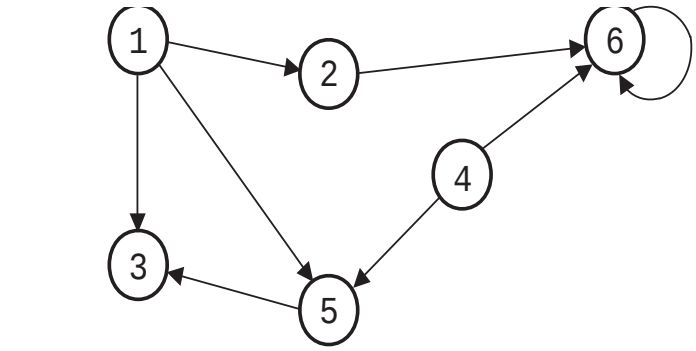


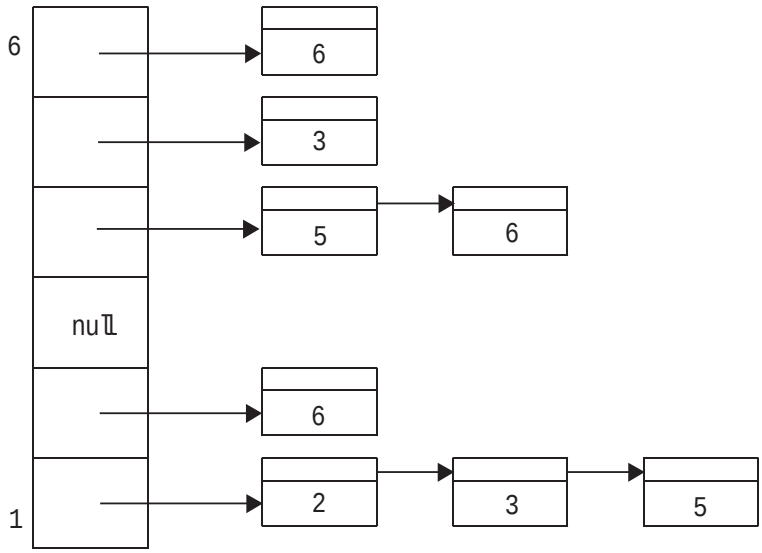
0	1	1	1
1	0	0	1
1	0	0	1
1	1	1	0

Si noti che utilizzando tale rappresentazione le operazioni di ricerca di un arco, inserimento di un arco e cancellazione di un arco richiedono tutte tempo costante.

Purtroppo, lo spazio di memoria occupato è quadratico nel numero dei vertici. Tale rappresentazione è quindi antieconomica in termini di occupazione di spazio se il grafo è *sparso*, cioè contiene un numero ridotto di archi.

Per sopperire all'occupazione eccessiva di spazio, è possibile rappresentare un grafo $G = (V, E)$ mediante ***liste di adiacenza***. La lista di adiacenza per il nodo i è una lista concatenata contenente i nodi adiacenti ad i. Possiamo rappresentare il grafo G mediante un vettore **Nodi** di dimensione n, tale che l'iesimo elemento di tale vettore contenga un puntatore alla lista di adiacenza del nodo i.





La rappresentazione di un grafo mediante liste di adiacenza richiede una quantità di spazio proporzionale alla somma del numero di vertici e del nu-

mero di archi. Lo svantaggio di questo tipo di rappresentazione è che richiede tempo lineare nel numero dei vertici per determinare se esiste un arco fra il nodo i ed il nodo j, visto che possono esserci al piu' n nodi nella lista di adiacenza del nodo i e, come è noto, la ricerca di un elemento in una lista concatenata richiede tempo lineare.

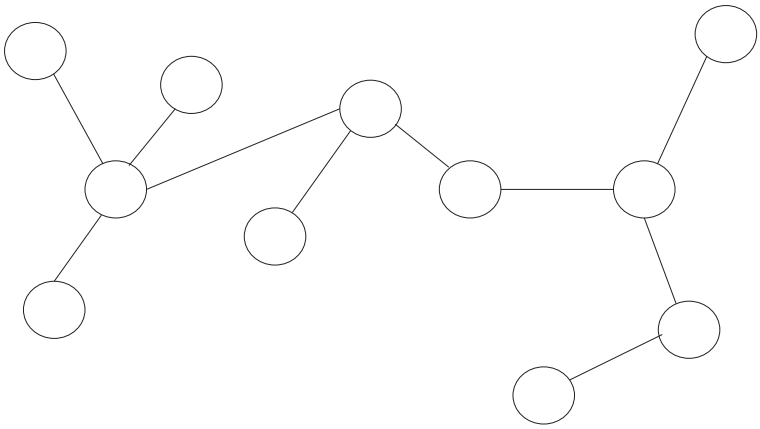
A.7 Alberi liberi

Un albero libero T è definito formalmente come una coppia $T = (V, E)$, dove V è un insieme finito di nodi ed E è un insieme di coppie non ordinate di nodi, detti archi, tali che:

- il numero di archi è uguale al numero di nodi meno uno;
- T è *connesso*, ovvero per ogni coppia di nodi v e w in V esiste una sequenza di nodi distinti $v_0, v_1, \ldots, v_k$ tali che $v = v_0$, $w = v_k$, e la coppia $[v_i, v_{i+1}]$ è un arco di E, per $i = 0, 1, \ldots, k-1$.

Pertanto un *albero libero* è un grafo non orientato connesso ed aciclico.

È facile verificare che un albero libero è un grafo non orientato connesso avente il minimo numero di archi.



A.8 Alberi orientati

L'albero orientato è una struttura fondamentale, che si presta naturalmente per rappresentare:

1 Organizzazioni gerarchiche;

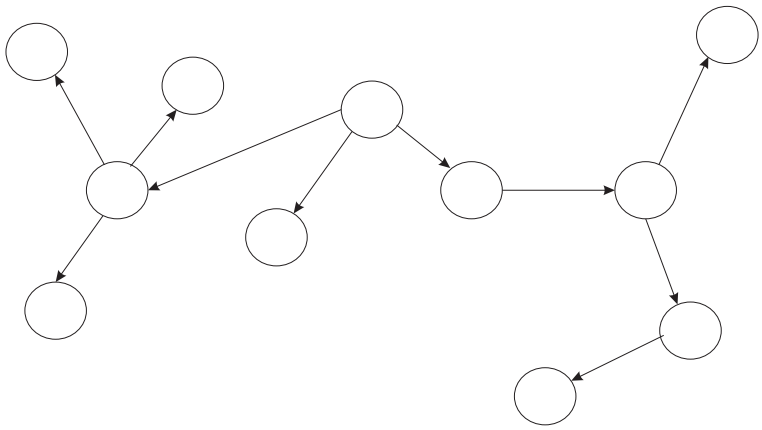
2 Partizioni successive di un insieme in sottoinsiemi disgiunti;

3 Procedimenti enumerativi o decisionali.

Definizione 23 *Sia $T = (V, E)$ un grafo orientato. Diciamo che T è un albero orientato se sono rispettate tutte le seguenti condizioni:*

- *T è connesso;*
- *T non contiene cicli non orientati;*
- *Esiste un unico nodo avente grado entrante 0; tale nodo è detto radice;*
- *Ogni altro nodo ha grado entrante 1.*

I nodi aventi grado uscente 0 si definiscono *foglie*. Tutti i nodi che non sono nè foglie nè radice sono detti *nodi interni*.



Per ogni nodo dell'albero esiste ed è unico il cammino diretto dalla radice al nodo stesso. La lunghezza di questo cammino è definita *altezza del nodo*. La maggiore delle altezze di tutti i nodi dell'albero viene definita *altezza dell'albero*.

Se (v, w) è un arco del nostro albero, diremo che w è *figlio* di v, ovvero v è padre di w. Si noti che per definizione ogni nodo che non sia la radice ha uno ed un solo padre.

Ogni nodo w raggiungibile da un nodo v con un cammino diretto di lunghezza ≥ 1 è definito *discendente* di v; il nodo v è un *antenato* del nodo w.

La rappresentazione grafica di un albero è detta in forma *canonica* se la radice è collocata in testa e tutti i figli di ciascun nodo sono posti allo stesso livello.

Un albero T è detto binario se ogni nodo che non sia una foglia ha al piu' due figli. La rapprentazione grafica canonica di un albero binario induce un ordinamento relativo tra i figli di un nodo assegnato: in altre parole è possibile definire il figlio sinistro ed il figlio destro di un nodo assegnato. Si noti che un nodo puo' possedere il solo figlio sinistro oppure il solo figlio destro.

In tal modo, dato un albero binario T ed un suo nodo v, si definisce *sottoalbero sinistro* (risp. *sottoalbero destro*) di v l'albero avente per radice il figlio sinistro (risp. destro) di v.

Appendice B

Macchina di Turing

In questa sezione definiamo la *Macchina di Turing deterministica*. Per i nostri scopi, il concetto di macchina è quello di macchina *multinastro off-line*. Essa consiste di:

1 un insieme di stati (chiamati anche *controllo finito* della macchina);

2 un numero fissato di *nastri di lavoro* semi-infiniti, divisi in *celle*; ciascuno di essi è dotato di una testina che può spostarsi di una cella a destra oppure a sinistra, scandendo le celle del nastro una alla volta;

3 un nastro supplementare (con la corrispondente testina), chiamato *nastro di input*.

Con nastro seminfinito intendiamo dire che esso non ha un limite destro, ma ha una cella estrema a sinistra: se la testina si trova nella cella estrema, non può spostarsi verso sinistra.

In ogni istante discreto di tempo il dispositivo si trova in uno degli stati, e legge il contenuto delle celle scandite dalle testine; può modificare il contenuto delle celle scandite (scrivendo un nuovo simbolo in ciscuna di esse), spostare ogni testina a sinistra oppure a destra, cambiando il suo stato interno. Tutte queste operazioni corrispondono ad uno *step* e sono definite da una *funzione di transizione*.

Si osservi che non è possibile modificare il contenuto del nastro di input (le macchine sono off-line).

Una macchina M comincia le sue operazioni su una parola in input w, con il nastro di input contenente w, e il simbolo speciale $\backslash b$ in tutte le altre

celle. Lo stato interno iniziale è q_0. M procede applicando le funzioni di transizione finchè non giunge in uno *stato finale*, di accetazione o rifiuto; oppure si arresta se la funzione di transizione è indefinita. Più formalmente:

Definizione 24 *Una macchina di Turing con k nastri è una quintupla*

$$M = (Q, \Sigma, \delta, q_0, F)$$

dove:

1 Q è l'insieme finito di stati interni;

2 Σ è l'alfabeto di nastro;

3 $q_0 \in Q$ è lo stato iniziale;

4 F è l'insieme di stati finali accettanti;

5 $\delta : \mathsf{Q} \times \Sigma^{\mathsf{k}} \to \mathsf{Q} \times \Sigma^{\mathsf{k-1}} \times \{\mathsf{R}, \mathsf{N}, \mathsf{L}\}^{\mathsf{k}}$ è una funzione parziale, detta funzione di transizione di M.

Se la funzione δ è definita (in caso contrario la computazione si arresta) essa deve essere interpretata come segue:

$$\delta(q, (x_1, \ldots, x_k)) = (p, (y_1, \ldots, y_{k-1}), (A_1, \ldots, A_k))$$

se M essendo nello stato q e scandendo i simboli $x_1, \ldots, x_k$ nei k nastri, passa nello stato p, scrive i simboli $y_1, \ldots, y_{k-1}$ nei $k-1$ nastri di lavoro, e sposta le testine sui k nastri ($A_i = L, R, N$ se la testina deve essere spostata a sinistra, destra o tenuta ferma sull'i-esimo nastro).

Definizione 25 *Sia M una macchina di Turing. Una* configurazione *di M (detta anche* descrizione istantanea*) è una descrizione dell'intero stato della computazione. In essa troviamo il contenuto di ciascun nastro, le posizioni di ciascuna testina e lo stato della macchina.*

Se M ha k nastri, una configurazione è la $k + 1 - pla$

$$(q, x_1, \ldots, x_{k-1}, x_k)$$

dove

1 q è lo stato corrente di M;

2 $x_j \in \Sigma^*\#\Sigma^*$. $(j = 1, \ldots, k)$ rappresenta il contenuto del j-imo nastro; il simbolo # non è in Σ, e marca la posizione della testina del relativo nastro (convenzione: la testina scandisce il simbolo immediatamente a destra di #). Tutti i simboli che non sono in x_j sono b.

All'inizio della computazione tutti i nastri sono vuoti (contengono b) con l'eccezione del nastro di input.

Definizione 26 *La* configurazione iniziale *di una macchina M su un input w è:*

$$(q_0, \#w, \#, \ldots, \#)$$

Definizione 27 *Una* configurazione accettante *è una configurazione di M il cui stato è uno stato accettante ($\in F$).*

La definizione di macchina considerata ci impone di non modificare l'input durante la computazione. Quindi non c'è l'esigenza di includere il contenuto del nastro di input nella configurazione.

Definizione 28 *Sia M una macchina e w un input fissato. Una* configurazione *di M su w ha la forma:*

$$(q, i, x_2, \ldots, x_k)$$

dove i è la posizione della testina relativa al nastro di input.

Definizione 29 *Sia M una macchina e w un input fissato. Una* computazione parziale *di M su w è una sequenza (finita o infinita) di configurazioni di M, nella quale ogni passaggio da una configurazione alla successiva rispetta la funzione di transizione.*

Una computazione *è una computazione parziale che comincia con la configurazione iniziale di M su w e termina in una configurazione dalla quale non possono essere effettutati più passi.*

Una *computazione accettante* è una computazione che termina con una configurazione accettante, e in tal caso la parola in input si dice *accettante*. Il *linguaggio accettato* da M (denotato con $L(M)$) è l'insieme delle parole accettate da M.

B.1 Macchina di Turing per il calcolo di funzioni di interi

La macchina di Turing può essere utilizzata per il calcolo di funzioni. L'approccio tradizionale vede la rappresentazione dei numeri interi con un numerale nuovo: $i \geq 0$ è rappresentato dalla stringa 0^i. Se una funzione ha k argomenti $(i_1, \ldots, i_k)$, essi sono collocati sul nastro di input di M nella forma $0^{i_1}1\ 0^{i_2}1 \ldots 1\ 0^{i_k}$;

Se la macchina si arresta con il numerale 0^m su un determinato nastro, si ha che $f(i_1, \ldots, i_k) = m$ e la funzione f si dice *calcolata* da M.

B.2 Modelli alternativi di macchina

Un modello differente di macchina di Turing è la macchina *on_line*; la differenza rispetto alla macchina off_line è che nella prima la funzione di transizione non può specificare la 'mossa' L per la testina di input. Quindi l'input può solo essere letto in avanti.

Un altro modello è la macchina a un solo nastro, che legge e scrive i simboli sull'unico nastro di input.

In letteratura si trovano numerosi altri modelli: la macchina two-way, quella multitraccia , le non deterministiche, le multidimensionali, le macchine a più testine. Tali modelli sono computazionalmente equivalenti al modello descritto, nel senso che accettano la stessa classe di linguaggi (quelli ricorsivamente enumerabili); le dimostrazioni di equivalenze possono essere trovate in [19].

Data l'esistenza di questi modelli (e di molti altri modelli di calcolo ad essi equivalenti), la macchina di Turing corrisponde al concetto intuitivo e naturale di algoritmo.

Il seguente assioma (la tesi di Curch) è quindi accettabile:

Ogni algoritmo può essere descritto da una macchina di Turing.

Naturalmente il modello di Turing non è amichevole. Scrivere con esso un programma e verificarne la correttezza o la quantità di risorse richieste è difficile. Infatti, la descrizione degli algoritmi presentati in questo libro è stata fatta attraverso uno pseudo-codice simile al Pascal, e facendo appello alla tesi di Churh.

Ma sorge immediatamente la seguente domanda: perchè scegliere la macchina di Turing come modello di computazione, visto che la presentazione degli algoritmi avviene in pseudo-codice? Le risposte sono almeno due: innanzitutto, questo modello definisce chiaramente e non ambiguamente il concetto di passo della computazione, e quindi di *tempo* richiesto. Secondo, offre la definizione di unità di memoria (la cella del nastro), e quindi di *spazio* utilizzato. Ciò non accade per tutti i modelli di calcolo.

Inoltre, è provato che scelti due modelli di calcolo qualsiasi, ciascuno di essi simula l'altro in tempo (al più) quadratico e spazio lineare. Le m.d.t. sono esponenti rappresentativi della classe dei modelli che godono di tale proprieta' (tesi dell'invarianza), fondamentale, fra l'altro, nella definizione delle classi di complessità centrali, la cui definizione deve essere indipendente dal modello usato.

L'obiettivo di questo libro non è di studiare i rapporti fra i modelli di computazione, o le relazioni fra le classi centrali; si fa comunque notare che l'equivalenza computazionale non vale per alcuni interessanti modelli di calcolo.

Bibliografia

[1] *A.V.Aho, J.E.Hopcroft, J.D.Ullman,* **Data structures and algorithms,** Addison Wesley, 1983

[2] *A.V.Aho, J.E.Hopcroft, J.D.Ullman,* **The design and Analysis of Computer Algorithms,** Addison Wesley, 1974

[3] *S.Baase,* **Computer Algorithms: Introduction to Design and Analysis,** Addison Wesley, second edition, 1988

[4] *R.Bellman,* **Dynamic Programming,** Princeton University Press, 1957

[5] *A.A.Bertossi,* **Strutture, algoritmi, complessità,** Ed. Culturali Internazionali Genova, 1990

[6] *A.Church,* **An unsolvable problem of elementary number theory,** American J. Math, 58, 345-363, 1936

[7] *A.Church,* **The calculi of Lambda-Conversion,** *Annals of Mathematics Studies*, 6, Princeton Univ. Press, Princeton, N.J., 1941

[8] *T.H.Cormen, C.E.Leiserson, R.L.Rivest,* **Introduzione agli algoritmi,** Vol. 1, Jackson Libri, 1995

[9] *T.H.Cormen, C.E.Leiserson, R.L.Rivest,* **Introduzione agli algoritmi,** Vol. 2, Jackson Libri, 1995

[10] *J.A.Bondy, U.S.R.Murty,* **Graph Theory with Applications,** American Elsevier, 1976

[11] *G.Brassard, P.Bratley,* **Algorithmics: Theory and Practice,** Printice-Hall, 1988

[12] *S.Cook,* **The complexity of theorem proving procedures,** In *Proceedings of the Third Annual ACM Symposium on Theory Computing* pp 151-158, 1971

[13] *S.Cook, R.A.Reckhow* **Time bounded random access machines,** J. computer and System Sciences 7:4, 343-353, 1973

[14] *S.Even,* **Graph Algorithms,** Computer Science Press, 1979

[15] *M.R.Garey, D.S.Johnson,* **Computer and Intractability: A Guide to the theory of NP-Completeness,** W.H. Freeman, 1979

[16] *G.H.Gonnet,* **Handbook of Algorithms and Data Structures,** Addison-Wesley, 1984

[17] *F.Harary,* **Graph Theory,** Addison-Wesley, 1969

[18] *J.Hartmanis, R.E.Stearns,* **On the computational complexity of algorithms,** Transactions of the American Mathematical Society, 117:285-306, 1965

[19] *J.E.Hopcroft, J.D. Ullman,* **Introduction to Automata Theory, Languages, and Computation,** Addison-Wesley, 1979

[20] *E.Horowits, S. Sahni,* **Fundamentals of Computer Algorithms,** Computer Science Press, 1978

[21] *S.C.Kleene,* **General recursive functions of natural numbers,** Mathematische Annalen, 112, 727-742, 1936

[22] *S.C.Kleene,* **Introduction to Metamathematics,** D. Van Nostrand, Princeton, N.J., 1952

[23] *D.E.Knuth,* **Fundamental Algorithms,** volume 1, **The Art of Computer Programming,** Addison-Wesley, 1968. Second edition, 1973

[24] *D.E.Knuth,* **Seminumerical Algorithms,** volume 2, **The Art of Computer Programming,** Addison-Wesley, 1968. Second edition, 1981

[25] *D.E.Knuth,* **Sorting and Searching,** volume 3, **The Art of Computer Programming,** Addison-Wesley, 1973

[26] *E.Lodi, G.Pacini,* **Introduzione alle strutture dati**, Bollati Boringhieri, 1990

[27] *U.Manber,* **Introduction to Algorithms: A creative Approach,** Addison-Wesley, 1989

[28] *K.Mehlhorn,* **Sorting and Searching, Data Structures and Algorithms,** volume 1, Springer-Verlag, 1984

[29] *K.Mehlhorn,* **Graph Algorithms and NP-Completeness, Data Structures and Algorithms,** volume 2, Springer-Verlag, 1984

[30] *M.O.Rabin,* **Probabilistic algorithms,** in F.Traub, editor, **Algorithms and Complexity: New Directions and Recent Results,** pp 21-39, Academic Press, 1976

[31] *J.E.Savage,* **The Complexity Of Computing,** John Wiley & Sons, 1976

[32] *R.Sedgewick,* **Algorithms,** Addison-Wesley, 1988

[33] *A.Turing,* **On computable numbers with an application to the Entscheidungs-problems,** Proc. London Math. Soc.,2:42 230-265, 1936. A correction, ibid., 43 544-546

[34] *H.S.Wilf,* **Algorithms and Complexity,** Prentice-Hall, 1986