

Appunti del corso di Algoritmi e Strutture Dati

Vincenzo Acciario

Per suggerimenti, commenti o segnalazione di errori scrivere a

acciaio@di.uniba.it

oppure a

m_di_leonardo@yahoo.it

Indice

| | | |
|----------|--|-----------|
| 1 | Introduzione | 9 |
| 1.1 | Programmazione in piccolo ed in grande | 9 |
| 1.2 | Obiettivi del corso | 9 |
| 1.3 | Motivazioni | 10 |
| 1.3.1 | I limiti del calcolabile | 11 |
| 1.4 | Algoritmo | 11 |
| 1.4.1 | Specifiche del problema | 13 |
| 1.4.2 | Strumenti di formalizzazione | 14 |
| 1.4.3 | Esempio di algoritmo | 14 |
| 1.4.4 | L'algoritmo come funzione | 15 |
| 1.4.5 | Nota storica | 15 |
| 1.5 | Definizione formale di problema. | 16 |
| 1.6 | Programma | 16 |
| 1.7 | Costi | 16 |
| 1.7.1 | Risorse di calcolo | 17 |
| 1.7.2 | Efficienza. | 17 |
| 1.7.3 | Modello di calcolo | 18 |
| 1.7.4 | Irrisolubilità | 18 |
| 1.7.5 | Intrattabilità | 19 |
| 2 | La macchina di Turing | 21 |
| 2.1 | Definizione di Macchina di Turing | 21 |
| 2.1.1 | L'alfabeto esterno della M.d.T. | 22 |
| 2.1.2 | Gli stati della M.d.T. | 22 |
| 2.1.3 | Configurazione iniziale della MdT. | 22 |
| 2.1.4 | Il programma nella M.d.T. | 22 |
| 2.1.5 | Il Programma come funzione | 23 |
| 2.1.6 | Terminazione della computazione. | 23 |

| | | |
|----------|--|-----------|
| 2.2 | Ipotesi fondamentale della teoria degli algoritmi | 23 |
| 2.2.1 | Irrisolubilità | 23 |
| 2.3 | Esempi | 24 |
| 2.4 | Esercizi | 25 |
| 3 | La Random Access machine (RAM) | 27 |
| 3.1 | Definizione | 27 |
| 3.2 | Complessità computazionale di programmi RAM | 28 |
| 4 | Nozioni base di complessità | 31 |
| 4.1 | Introduzione | 32 |
| 4.1.1 | Obiettivi in fase di progetto. | 32 |
| 4.1.2 | Ancora sulle risorse | 33 |
| 4.2 | Il tempo di esecuzione di un programma | 33 |
| 4.2.1 | Due misure apparentemente ragionevoli | 33 |
| 4.2.2 | Dimensione del problema | 34 |
| 4.2.3 | Misura della Dimensione | 34 |
| 4.3 | Complessità temporale | 34 |
| 4.4 | Confronto di algoritmi | 36 |
| 4.5 | Definizione formale di \mathcal{O} | 39 |
| 4.5.1 | Alcuni ordini di grandezza tipici | 41 |
| 4.6 | La notazione Ω | 41 |
| 4.6.1 | Definizione alternativa di Ω | 43 |
| 4.7 | La notazione Θ | 43 |
| 4.8 | Alcune proprietà di $\mathcal{O}, \Omega, \Theta$ | 44 |
| 4.8.1 | Sulle notazioni asintotiche | 44 |
| 4.9 | Ricapitolando | 44 |
| 4.10 | Complessità di problemi | 45 |
| 4.10.1 | La notazione \mathcal{O} applicata ai problemi | 45 |
| 4.10.2 | La notazione Ω applicata ai problemi. | 46 |
| 4.11 | Algoritmi ottimali | 46 |
| 4.12 | Funzioni limitate polinomialmente | 47 |
| 4.13 | Crescita moderatamente esponenziale | 47 |
| 4.14 | Crescita esponenziale | 47 |
| 4.15 | Appendice: Notazione asintotica all'interno di eguaglianze | 48 |
| 4.16 | Esercizi | 48 |

| | | |
|-----------|---|-----------|
| 5 | Linguaggi per la descrizione di algoritmi | 49 |
| 5.1 | Introduzione | 49 |
| 5.2 | Valutazione della complessità di algoritmi scritti in pseudo-codice | 50 |
| 5.3 | Alcune regole per il calcolo di \mathcal{O} | 51 |
| 5.3.1 | Applicazione: funzioni polinomiali | 52 |
| 6 | Algoritmi ricorsivi | 53 |
| 6.1 | Introduzione | 54 |
| 6.2 | Qualche mini esempio | 54 |
| 6.3 | Linguaggi di programmazione che consentono la ricorsione . . | 55 |
| 6.3.1 | Visita di un albero binario | 56 |
| 7 | L'approccio Divide et Impera | 57 |
| 7.1 | Introduzione | 59 |
| 7.1.1 | Esempio: il Mergesort | 60 |
| 7.2 | Bilanciamento | 61 |
| 7.3 | L'algoritmo di Strassen | 62 |
| 8 | Tecniche di analisi di algoritmi ricorsivi | 65 |
| 8.1 | Introduzione | 66 |
| 8.1.1 | Esempio: Visita di un albero binario | 68 |
| 8.2 | Soluzione delle equazioni di ricorrenza | 69 |
| 8.3 | Il caso Divide et Impera | 70 |
| 8.3.1 | Dimostrazione del Teorema Principale | 72 |
| 8.3.2 | Soluzione Particolare | 74 |
| 8.3.3 | Esempi | 76 |
| 9 | Liste, Pile e Code | 77 |
| 10 | Grafi e loro rappresentazione in memoria | 79 |
| 11 | Programmazione Dinamica | 81 |
| 11.1 | Introduzione | 81 |
| 11.1.1 | Un caso notevole | 81 |
| 11.1.2 | Descrizione del metodo | 82 |
| 11.1.3 | Schema base dell'algoritmo | 83 |
| 11.1.4 | Versione definitiva dell'algoritmo | 83 |
| 11.1.5 | Un esempio svolto | 84 |

| | |
|--|------------|
| 12 Dizionari | 85 |
| 13 Alberi | 87 |
| 14 Alberi di Ricerca Binari | 89 |
| 15 Alberi AVL | 91 |
| 16 2-3 Alberi e B-Alberi | 93 |
| 17 Le heaps | 95 |
| 17.1 Le code con priorità | 95 |
| 17.2 Le heaps | 96 |
| 17.3 Ricerca del minimo | 97 |
| 17.4 Inserimento | 97 |
| 17.5 Cancellazione del minimo | 98 |
| 17.6 Costruzione di una heap | 99 |
| 17.7 Esercizi | 101 |
| 18 Heapsort | 103 |
| 19 Tecniche Hash | 105 |
| 19.1 Introduzione | 106 |
| 19.2 Caratteristiche delle funzioni hash | 108 |
| 19.3 Tecniche di scansione della tabella | 109 |
| 19.3.1 Scansione uniforme | 109 |
| 19.3.2 Scansione lineare | 109 |
| 19.3.3 Hashing doppio | 110 |
| 19.3.4 Hashing quadratico | 110 |
| 19.4 Hashing tramite concatenamento diretto | 111 |
| 19.5 Hashing perfetto | 112 |
| 19.6 Implementazione pratica di uno schema ad indirizzamento aper- to | 112 |
| 19.7 Analisi di complessità | 115 |
| 19.7.1 Dimostrazione per le tecniche a concatenazione | 116 |
| 19.8 Esercizi di ricapitolazione | 117 |
| 19.9 Esercizi avanzati | 118 |

| | |
|--|------------|
| 20 Il BucketSort | 121 |
| 20.1 Descrizione dell'algoritmo | 121 |
| 20.1.1 Correttezza dell'algoritmo | 122 |
| 20.1.2 Complessità nel caso medio | 123 |
| 20.1.3 Complessità nel caso pessimo | 124 |
| 20.2 Esercizi | 124 |
| 21 Complessità del problema ordinamento | 125 |
| 21.1 Alberi decisionali | 125 |
| 22 Selezione in tempo lineare | 129 |
| 22.1 Introduzione | 129 |
| 22.2 Un algoritmo ottimale | 130 |
| 23 Algoritmi su grafi | 135 |

Capitolo 1

Introduzione

1.1 Programmazione in piccolo ed in grande

Distinguiamo grossolanamente l'attività di programmazione in due aree distinte:

- 1 La programmazione in grande;
- 2 La programmazione in piccolo.

La programmazione in grande si occupa della soluzione informatica di *problemi di grandi dimensioni* (ad esempio l'informatizzazione della Regione Puglia).

La programmazione in piccolo si preoccupa di trovare una *buona soluzione algoritmica* a specifici problemi ben formalizzati (ad esempio, l'ordinamento di una lista di elementi).

1.2 Obiettivi del corso

Il presente corso si preoccupa di fornire una introduzione alle nozioni di base ed ai metodi della *programmazione in piccolo*. In particolare studieremo:

- Come organizzare e rappresentare l'informazione (*le strutture dati*) al fine di ottenere una sua efficiente manipolazione (*gli algoritmi*);
- Come valutare la bontà di un programma (o meglio, di un algoritmo).

Testi di riferimento

I testi di riferimento sono indicati in bibliografia. Come libro di testo è fortemente consigliato il volume di Lodi e Pacini [1].

Come libro di consultazione si consiglia il classico testo di Aho, Hopcroft e Ullman [4]. Questo è un "must" per gli addetti al settore, ovvero un volume di cui non si può fare a meno.

Infine, la trattazione matematica del problema della complessità degli algoritmi è molto rigorosa nel primo volume di Cormen, Leiserson e Rivest [2].

1.3 Motivazioni

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 così in un giro vizioso.

La volgarizzazione dell'informatica, se da una parte ha contribuito al progresso economico, scientifico e tecnologico, ecc., dall'altra ha creato tanti miti, uno dei quali è il seguente:

Mito (da sfatare): *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 sufficiente per risolvere un problema assegnato in tempo ragionevole, possiamo sempre risolvere il problema utilizzando un calcolatore più potente.

Nota bene 1 *Nel linguaggio comune, la parola "risolvere" già implica "in tempo ragionevole".*

Ma, cosa si intende per "tempo ragionevole"? Intuitivamente, un intervallo di tempo è "ragionevole" se è tale quando si rapporta alla durata media della vita dell'uomo (1 minuto è ragionevole, 20 anni no!).

1.3.1 I limiti del calcolabile

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

Il più potente calcolatore che si possa costruire non potrà mai essere più 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).

Quindi tale calcolatore non potrebbe avere più di 10^{126} componenti.

A.R. Meyer e L.J. Stockmeyer hanno dimostrato che tale calcolatore impiegherebbe almeno 20 miliardi di anni per risolvere dei problemi la cui risolubilità è 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 risolva il problema in tempo ragionevole.

1.4 Algoritmo

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.

Ricapitoliamo qui di seguito alcuni concetti fondamentali:

- 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* (ad esempio $\{a, \dots, z\}$ oppure $\{14, 78, 98, 1\}$).
- 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 é 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.
- L'esecuzione termina dopo un numero di passi *finito*, che é funzione della particolare istanza del problema.

1 Attenzione

Occorre che sia ben chiara la distinzione tra problema, istanza del problema, e descrizione dell'istanza.

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 una 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

1.4.1 Specifiche del problema

Il precedente esempio Ordinamento è stato presentato in modo molto informale ("discorsivo").

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 c'è un problema, tuttavia non siamo in grado di formalizzare esattamente tale problema.

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.

2 Attenzione

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:** Un insieme di n chiavi a_1, \dots, 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, \dots, b_n che costituisce un riordinamento dell'insieme dato di chiavi e che soddisfa la relazione $b_i \leq b_{i+1}$ per ogni $i = 1, 2, \dots, n - 1$.

1.4.2 Strumenti di formalizzazione

La Matematica, ed in particolare la Logica Matematica, ci fornisce degli strumenti eccellenti di formalizzazione, poichè ci permette di asserire 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 $i = 1, 2, \dots, n - 1$).

1.4.3 Esempio di algoritmo

Uno degli algoritmi più elementari è quello dell'addizione di due interi, che si apprende nelle scuole elementari. Per addizionare due numeri interi composti da più cifre, occorre eseguire una sequenza di passi ("azioni") elementari, ognuna delle 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 2 *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 "bambino" (che sia in grado di svolgere le operazioni indicate sopra, e nel corretto ordine) l'esecuzione dell'algoritmo di addizione, senza che l'esecutore debba conoscere gli aspetti più profondi della aritmetica.

1.4.4 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 ingresso x di A il corrispondente valore in uscita $f_A(x)$.

1.4.5 Nota storica

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

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 (D.E. Knuth)

1.5 Definizione formale di problema.

Definizione 1 *Un problema è una funzione $P : D_I \rightarrow D_S$ 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 .

1.6 Programma

La seguente definizione di programma è dovuta a D.E. Knuth, uno dei padri fondatori dell'informatica:

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. (D.E. Knuth)

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.

1.7 Costi

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.

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

1.7.1 Risorse di calcolo

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.

Almeno inizialmente, è bene pensare al concetto di risorsa cercando dei riferimenti con il mondo reale, ad esempio l'organizzazione di un ufficio.

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.

1.7.2 Efficienza.

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.

1.7.3 Modello 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;*
ovvero il livello di dettaglio;
- 3 *generalità;*
ovvero 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 riguarda *l'esistenza di modelli di calcolo assolutamente generali*. Uno di questi è la macchina di Turing.

1.7.4 Irrisolubilità

Definizione 5 *Un problema è non risolubile algebricamente 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 algebricamente.

Esempio 4 Un noto problema non risolubile alitmicamente è il problema dell'ALT della macchina di Turing.

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

1.7.5 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 2

La macchina di Turing

In questa lezione sarà esaminato un modello di calcolo molto generale, la Macchina di Turing. Tale argomento è stato probabilmente già presentato nel corso di Programmazione, per introdurre in modo assolutamente formale la nozione di algoritmo e la Tesi di Church.

Nel contesto del nostro corso la Macchina di Turing è un eccellente strumento didattico, poichè ci consente di definire esattamente la nozione di algoritmo, ma soprattutto 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 nelle lezioni precedenti assumerà nuova luce.

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

2.1 Definizione di Macchina di Turing

Una Macchina di Turing consiste di:

- Un nastro infinito, suddiviso in cellette. Ogni celletta può contenere un solo simbolo, tratto da un insieme finito S detto alfabeto esterno.
- Una testina di lettura capace di leggere un simbolo da una celletta, scrivere un simbolo in una celletta, e muoversi di una posizione sul nastro, in entrambe le direzioni.

- Un insieme finito Q di stati, tali che la macchina possa trovarsi in esattamente uno di essi in ciascun istante.
- Un programma, che specifica esattamente cosa fare per risolvere un problema specifico.

2.1.1 L'alfabeto esterno della M.d.T.

L'alfabeto esterno $S = \{s_1, \dots, s_k\}$ 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.

2.1.2 Gli stati della M.d.T.

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

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

2.1.3 Configurazione iniziale della MdT.

La macchina di Turing si trova inizialmente nello stato q_0 . L'informazione da elaborare è contenuta in cellette contigue del nastro, ed è codificata utilizzando i simboli dell'alfabeto esterno S . Tutte le altre cellette 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).

2.1.4 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;

2.2. IPOTESI FONDAMENTALE DELLA TEORIA DEGLI ALGORITMI²³

- 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.

2.1.5 Il Programma come funzione

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

$$f : QxS \rightarrow QxSxT$$

E' 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 $f(q_i, s_j)$.

2.1.6 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.

2.2 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.

2.2.1 Irrisolubilità

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

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

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

2.3 Esempi

1) Controllo di parità.

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

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

| | | | | | | | |
|-----|-----|---|---|---|-----|-----|-----|
| b | b | 1 | 1 | 1 | b | b | b |
|-----|-----|---|---|---|-----|-----|-----|

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

| | | | | | | | |
|-----|-----|---|---|---|-----|-----|-----|
| b | b | 1 | 1 | 1 | D | b | b |
|-----|-----|---|---|---|-----|-----|-----|

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

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

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 | | | |

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:

| | | | | | | | | | | | | |
|-----|-----|---|---|---|---|---|---|---|---|-----|-----|-----|
| b | b | 1 | 1 | 1 | + | 1 | 1 | 1 | 1 | b | b | b |
|-----|-----|---|---|---|---|---|---|---|---|-----|-----|-----|

Voglio ottenere la seguente configurazione sul nastro:

| | | | | | | | | | | | | |
|-----|-----|-----|---|---|---|---|---|---|---|-----|-----|-----|
| b | b | b | 1 | 1 | 1 | 1 | 1 | 1 | 1 | b | b | b |
|-----|-----|-----|---|---|---|---|---|---|---|-----|-----|-----|

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

| | | |
|------------|---------------|----------------------|
| $(q_0, 1)$ | \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 | $(q_f, b, FERMO)$ |

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 | | $(q_2, b, SINISTRA)$ | | |

2.4 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, la stringa "anna" é palindroma.
- 2 Costruire una MdT capace di riflettere ("capovolgere") una stringa data in input.

Capitolo 3

La Random Access machine (RAM)

Questo capitolo non è essenziale in un corso introduttivo di algoritmi e strutture dati.

L'argomento può essere trattato, se il tempo a disposizione lo consente, in appendice al corso, per introdurre il modello di costo logaritmico, e mostrare la sua validità nella valutazione della complessità delle operazioni aritmetiche.

3.1 Definizione

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

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

- Un nastro di lunghezza infinita 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 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 può modificare se stesso;
- 2 Tutti i calcoli avvengono utilizzando una locazione fissa di memoria della "accumulatore";
- 3 Ogni locazione di memoria ed ogni celletta del nastro di input ed output sono in grado di contenere un simbolo arbitrario (oppure, se si preferisce, un "numero intero di dimensione arbitraria");
- 4 Il programma può accedere alle singole locazioni ("cellette") di memoria in ordine arbitrario.

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

- Il tempo di accesso ad una locazione ("celletta") di memoria deve essere indipendente dalla celletta.

3 Attenzione

Perché non è possibile implementare praticamente ("costruire") una RAM?

Se poniamo alcune limitazioni (in particolare la rimozione dei termini "infinito" ed "arbitrario", ovunque essi compaiano) in modello RAM è praticamente implementabile, e rispecchia l'architettura della maggior parte dei calcolatori attuali.

3.2 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 così 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

Nelle lezioni precedenti sono state definite formalmente le nozioni di algoritmo, problema, e risorse di calcolo utilizzate da un algoritmo.

È stato inoltre introdotto il concetto di modello di calcolo, quale astrazione di un esecutore reale.

In particolare, è stato esaminato in dettaglio un modello di calcolo, la Macchina di Turing, al fine di formalizzare la nozione di algoritmo e fornire al tempo stesso una immagine molto accurata delle nozioni di spazio e tempo utilizzati da un programma, come numero di cellette 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.

A parte tali definizioni assolutamente generali, non è stato mostrato alcun esempio concreto di algoritmo efficiente o poco efficiente.

Nella presente lezione verranno ripresi brevemente tali concetti, e verrà definita, inizialmente in maniera informale, la nozione di complessità di un algoritmo.

Quindi sarà definita la nozione di complessità di un problema, e la relazione tra le due nozioni di complessità.

4.1 Introduzione

Abbiamo visto nelle precedenti lezioni che possiamo suddividere le problematiche riguardanti lo studio degli algoritmi 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 , dimostrare che A risolve P (correttezza) e valutare la quantità di risorse utilizzate da A ;
- *Classificazione (o complessità strutturale)*:
data una quantità T di risorse individuare la classe dei problemi risolvibili 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 contraddittori.

Se un programma deve essere utilizzato poche volte, l'obiettivo facilità di codifica è alquanto importante.

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 per il controllo di aeromobili;
- Sistemi in tempo reale per il controllo di strumentazione ospedaliera

allora l'obiettivo efficienza è prioritario.

4.1.2 Ancora sulle risorse

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 da fattori quali:

- 1 La velocità del calcolatore;
- 2 La bontà dell'interprete o la qualità del codice generato dal compilatore;
- 3 La complessità temporale dell'algoritmo sottostante;
- 4 L'input del programma.

4.2.1 Due misure apparentemente ragionevoli

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 cellette 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 (sinonimi: "assiomi", "presupposti") 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!

4.2.2 Dimensione del problema

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 .

4.2.3 Misura della Dimensione

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, attenzione!) 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, attenzione) 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 utilizzando "carta e penna".

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 una istanza del problema di dimensione n , nel caso pessimo. Si indica con $T_A^p(n)$.

4 Attenzione

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 una 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.

Quale soluzione è migliore?

Nessuna delle due. 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 A1 ed A2, su due diverse architetture (computer1 e computer2)

| | computer 1 | | computer 2 | |
|------------|------------|------|------------|------|
| Dim. input | A1 | A2 | A1 | A2 |
| 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 é evidente per input di dimensione piccola!

Poiché 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,

5 Attenzione

Non é possibile esprimere la complessità temporale intrinseca di un algoritmo utilizzando unità di misura quali i secondi.

É ragionevole dire piuttosto:

il tempo di esecuzione del programma che implementa l'algoritmo é proporzionale ad n^3 , per un input di dimensione n .

Si noti che la costante di proporzionalità non é specificata, poiché 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 é 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 |
| A1 | n | 1000 | 60000 | 3600000 |
| A2 | $n \log n$ | 140 | 4893 | 200000 |
| A3 | n^2 | 31 | 244 | 1897 |
| A4 | n^3 | 10 | 39 | 153 |
| A5 | 2^n | 9 | 15 | 21 |

Considerazioni sulla tabella.

Si noti che gli algoritmi dotati di complessità temporale lineare o quasi 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 | massima del | problema |
|------|-----------------------|------------|-------------|-----------|
| | | 1 sec | 1 min | 1 ora |
| A1 | n | 10000 | 600000 | 360000000 |
| A2 | $n \log n$ | 9990 | 599900 | 3599900 |
| A3 | n^2 | 97 | 750 | 5700 |
| A4 | n^3 | 21 | 80 | 300 |
| A5 | 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 é ancora evidente, sebbene in maniera inferiore;
- Infine, negli algoritmi esponenziali (2^n) i vantaggi che derivano dall'evoluzione della tecnologia sono pressochè irrilevanti.

Mini 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)
- operazioni di confronto (passo 3)
- operazioni di scambio

al più effettuati per un input di dimensione n .

$$\begin{aligned} T_{AP}^p(n) &= (n+2) + (n+1) + \dots + 4 = \\ &= [(n+2) + n + \dots + 4 + 3 + 2 + 1] - 6 = \end{aligned}$$

$$\begin{aligned}
&= \frac{(n+3)(n+2)}{2} - 6 = \\
&= \frac{1}{2}n^2 + \frac{5}{2}n - 3
\end{aligned}$$

Osservazioni

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

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 ci interessano affatto.

Esempio 7 Supponiamo che:

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

Esistono allora 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 $T(n)$ é al più proporzionale a $f(n)$.

6 Attenzione

la costante n_0 dipende dalla costante di proporzionalità c prefissata.

Esempio 8 Sia

$$T(n) = \frac{1}{2}n^2 + 2n + 12$$

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:

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

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. Che possiamo dire sulla complessità di questo algoritmo? In particolare, possiamo dire che sia lineare?

Certamente il tempo di esecuzione sarà lineare in n .

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

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

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

4.5.1 Alcuni ordini di grandezza tipici

$$[T(n) = O(f(n))]$$

$$\begin{array}{c} 1 \\ (\log n)^k \\ \sqrt{n} \\ n \\ n(\log n)^k \\ n^2 \\ n^2(\log n)^k \\ n^3 \\ n^3(\log n)^k \\ \dots \\ a^n \end{array}$$

7 Attenzione

\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:

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

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 $cT(n) \geq f(n)$.

8 Attenzione

$f(n)$ 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

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

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 tra le t che compaiono nella somma, e z il corrispondente esponente di $\log n$.

Esempio 10 La funzione $T(n) = 4n^5(\log n)^3 + 9n^3(\log n)^5 + 125n^4(\log n)^7$ 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 $f(n)$ è $\Omega(g(n))$ se esiste una costante positiva c ed una sequenza infinita $n_1, n_2, \dots \rightarrow \infty$ tale che per ogni i risulti $f(n_i) > cg(n_i)$.

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

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

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

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

4.7 La notazione Θ

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

In altre parole, quando una funzione $f(n)$ 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 $f(n)$, e scriviamo $f(n) = \Theta(g(n))$.

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} , Ω , Θ

- 1 La *transitività* vale per \mathcal{O} , Ω , Θ .

Ad esempio, $f(n) = \mathcal{O}(g(n))$ e $g(n) = \mathcal{O}(h(n))$ implicano

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

- 2 La *riflessività* vale per \mathcal{O} , Ω , Θ .

Ad esempio, per ogni funzione f risulta $f(n) = \mathcal{O}(f(n))$.

- 3 La *simmetria* vale per Θ .

Ovvero, $f(n) = \Theta(g(n))$ se e solo se $g(n) = \Theta(f(n))$.

- 4 La *simmetria trasposta* vale per \mathcal{O} e Ω .

Ovvero, $f(n) = \mathcal{O}(g(n))$ se e solo se $g(n) = \Omega(f(n))$.

4.8.1 Sulle notazioni asintotiche

I simboli \mathcal{O} , Θ , Ω 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 cellette utilizzate per rappresentare l'istanza del problema.

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

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

Poiché 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.

Nota bene 4 *dato un problema P , possono esistere in generale 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 5 *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 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 .

Nota bene 6 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) $1 \leq i, j \leq 3$, determinare se

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

Capitolo 5

Linguaggi per la descrizione di algoritmi

5.1 Introduzione

Per descrivere un algoritmo è diventato comune adottare una variante semplificata del PASCAL o del C (pseudo-linguaggio).

- 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 così 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 Valutazione della complessità di algoritmi scritti 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 azione1 else azione2
 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 data 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 ed 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 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.

Mini 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 $i \leq n$ 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 $\mathcal{O}(n^2)$.

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 $\mathcal{O}(k f(n)) = \mathcal{O}(f(n))$.

5.3.1 Applicazione: funzioni polinomiali

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

Si ha $T(n) = \mathcal{O}(n^k)$. Si deduce dalla regola della somma:

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

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

Si presuppone che lo studente abbia già incontrato la nozione di *algoritmo ricorsivo* all'interno del corso di "Programmazione".

Sicuramente lo studente avrà incontrato alcuni esempi canonici, quali l'algoritmo per il *calcolo del fattoriale di un numero* e l'algoritmo per risolvere il problema della *Torre di Hanoi*, ed avrà studiato la loro implementazione in Pascal.

Probabilmente lo studente avrà studiato anche il *quicksort* o il *mergesort*, dipendentemente dal contenuto del corso di "Programmazione".

Si presuppone che lo studente abbia già incontrato la tecnica di dimostrazione per induzione nel corso di "Programmazione" e/o in uno dei corsi di Matematica del primo anno.

Agli algoritmi ricorsivi saranno dedicate due lezioni di due ore ciascuna.

Verrà innanzitutto presentata la definizione di algoritmo ricorsivo e verranno ripresi alcuni esempi canonici già incontrati nel corso di "Programmazione".

Si mostrerà poi come la ricorsione permette una migliore comprensione di un algoritmo e semplifica l'analisi di correttezza dello stesso, mostrando come esempio i due approcci ricorsivo ed iterativo alla visita di un albero.

Infine si accennerà brevemente al problema di calcolare la complessità di un algoritmo ricorsivo, problema che sarà affrontato in dettaglio in un ciclo di lezioni successivo.

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 più chiara e concisa di algoritmi, di quanto sia possibile senza la ricorsione.

Gli algoritmi ricorsivi:

- 1 Spesso costituiscono il metodo più naturale di risoluzione di un problema;
- 2 Sono facili da comprendere;
- 3 Sono facili da analizzare:
 - La correttezza di un programma ricorsivo si dimostra facilmente per induzione;
 - Il calcolo della complessità temporale di un algoritmo ricorsivo si riduce alla soluzione di relazioni di ricorrenza.

6.2 Qualche mini esempio

Calcolo del fattoriale di un numero

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

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

```
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 per induzione matematica. Infatti:

1 (*correttezza dal passo base*)

L'algoritmo è certamente corretto per $n = 1$, in quanto $1! = 1$;

2 (*correttezza del passo induttivo*)

Se l'algoritmo calcola correttamente $n!$, allora esso calcola correttamente $(n + 1)!$, che è dato da $(n + 1) \cdot n!$.

6.3 Linguaggi di programmazione che consentono la ricorsione

Non tutti i linguaggi di programmazione permettono la ricorsione: ad esempio, il FORTRAN oppure il COBOL non la permettono.

Fortunatamente, il linguaggio PASCAL, che è stato studiato nel corso di "Programmazione", oppure il linguaggio C, che viene trattato nel "Laboratorio di Algoritmi e Strutture Dati", lo permettono.

Procedura attiva

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

Stack 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, e l'indirizzo di ritorno, vale a dire 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

Nel classico libro [4] a pp. 56–57 sono riportati un algoritmo ricorsivo ed uno non-ricorsivo per visitare un albero binario.

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

Nel corso degli anni, i ricercatori nel campo dell'informatica hanno identificato diverse tecniche generali di progetto di algoritmi che consentono di risolvere efficientemente molti problemi.

Lo studente avrà già avuto modo di incontrare una di tali tecniche, il backtracking, nel corso di "Programmazione", in relazione al classico problema delle "Otto Regine". Tale tecnica sarà ripresa e trattata in dettaglio nei corsi di "Intelligenza artificiale"

In un corso introduttivo di "Algoritmi e strutture dati" ritengo essenziale includere almeno:

- Il metodo Divide et Impera;
- La programmazione dinamica;
- Le tecniche greedy.

Le tecniche di ricerca locale sono anch'esse molto importanti e meritano una trattazione approfondita. Ritengo possibile omettere tali tecniche dal momento che saranno affrontate in dettaglio nei corsi di "Intelligenza artificiale".

Si noti comunque che, prima di passare alla progettazione di un algoritmo, è necessario individuare se il problema da risolvere è irrisolvibile o intrattabile. In tali casi nessuna tecnica di progetto ci permetterà di ottenere, ovviamente, algoritmi efficienti.

Il metodo Divide et Impera è una tra le tecniche più importanti, e dal più vasto spettro di applicazioni.

A tale tecnica saranno dedicate due lezioni da due ore ciascuna.

Nella prima lezione, dopo una definizione formale della tecnica, sarà presentato un esempio notevole, il mergesort. Si mostrerà quindi come effettuare l'analisi di complessità di tale algoritmo, espandendo la relazione di ricorrenza che ne definisce il tempo di esecuzione. Il risultato di tale analisi, $O(n \log n)$ consentirà allo studente di apprezzare meglio tale algoritmo, confrontandone la complessità con quella di un algoritmo di ordinamento già noto, il Bubble-sort. Si accennerà quindi al fatto che il problema ordinamento ha complessità $\Omega(n \log n)$, e quindi lo studente avrà incontrato un algoritmo ottimale di ordinamento. La dimostrazione della complessità del problema ordinamento, attraverso alberi decisionali, sarà comunque relegata ad un ciclo di lezioni successivo dedicato al problema ordinamento.

Si passerà quindi a discutere il problema del bilanciamento, e per mostrare le implicazioni si farà vedere che modificando opportunamente il Mergesort in modo da avere sottoproblemi di dimensione differente, l'algoritmo risultante sarà decisamente meno efficiente, ed avrà una complessità pari a quella del Bubblesort.

Nella seconda lezione si mostrerà come, tale metodo, sebbene si utilizzino sottoproblemi di dimensione simile, non porta in genere ad algoritmi più efficienti. In altre parole, occorre comunque effettuare sempre l'analisi della complessità dell'algoritmo per poter dire se sia efficiente o meno.

A tal fine, si mostrerà un algoritmo ricorsivo di moltiplicazione di matrici, che richiede ad ogni passo ricorsivo 8 moltiplicazioni di matrici di dimensione dimezzata. Dall'analisi del tempo di esecuzione di tale algoritmo lo studente potrà evincere il fatto che la sua complessità, in termini di operazioni aritmetiche elementari, è pari a quella dell'algoritmo classico.

Si mostrerà quindi l'algoritmo di Strassen, che è una evoluzione della tecnica (intuitiva) ricorsiva appena vista, ma richiede ad ogni passo il calcolo di 7 prodotti di matrici di dimensione pari alla metà della dimensione originale. Si mostrerà quindi come analizzare il tempo di esecuzione di tale algoritmo.

Per concludere la lezione, si accennerà brevemente alla complessità del problema Moltiplicazione di Matrici, ed al fatto che nessun algoritmo ottimale è noto sinora per risolvere tale problema.

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 nello spezzare un problema in sottoproblemi di dimensione più piccola, risolvere ricorsivamente tali sottoproblemi, e quindi ottenere dalle soluzioni dei sottoproblemi la soluzione del problema globale.

Distinguiamo quindi in un algoritmo ottenuto attraverso tale tecnica 3 fasi:

1 *suddivisione del problema originale in sottoproblemi*

Il problema originale di dimensione n è decomposto in a sottoproblemi di dimensione minore $f_1(n), \dots, f_a(n)$;

2 *soluzione ricorsiva dei sottoproblemi*

Gli a sottoproblemi sono risolti ricorsivamente;

3 *fusione delle soluzioni, per ottenere la soluzione del problema originale*

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

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

Si noti che problemi di dimensione sufficiente piccola possono essere risolti trivialmente in tempo costante c , consultando una tabella costruita in precedenza.

Otteniamo pertanto la seguente equazione che esprime il tempo di esecuzione dell'algoritmo:

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

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.

7.1.1 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 $L_1 = \text{Mergesort}(S_1)$**
7. **Poni $L_2 = \text{Mergesort}(S_2)$**
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 $\Theta(n)$

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

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 così' via ... Poichè $T(2)$ è noto, ci è possibile calcolare $T(n)$ per qualunque valore di n che sia una potenza di due.

Nelle lezioni successive mostreremo come risolvere tale relazione, ovvero come ottenere una stima per $T(n)$ che non dipenda da $T(h)$ per qualche $h < n$. Per ora basti sapere che asintoticamente $T(n) = O(n \log n)$, quindi il Mergesort è sicuramente (asintoticamente) migliore dello stranoto Bubblesort.

7.2 Bilanciamento

Abbiamo visto che nel Mergesort il problema originale viene suddiviso in sottoproblemi di pari dimensione. Questo non è un caso. Infatti, 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 siffatto algoritmo sarà descritto dalla seguente equazione di ricorrenza:

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

Tali equazioni di ricorrenza possono essere risolte utilizzando delle tecniche standard che vedremo nelle seguenti lezioni.

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 $O(n^2)$. Ciò è stato dimostrato in una lezione introduttiva dedicata al calcolo della complessità di algoritmi espressi utilizzando uno

pseudo-linguaggio. In tale lezione, l'insertion sort è stato codificato in forma iterativa. La corrispondente versione ricorsiva è la seguente:

1. **InsertionSort(S):**
2. **Se S ha 1 elemento**
3. **Ritorna S**
4. **altrimenti**
5. **Spezza S in due sottoliste S_1 e S_2 di cardinalità 1 ed $n - 1$**
6. **Poni $L_1 = \text{InsertionSort}(S_1)$**
7. **Poni $L_2 = \text{InsertionSort}(S_2)$**
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 $T(n) = O(n^2)$.

7.3 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 $c_{ij} = \sum_{k=1}^n a_{ik} b_{kj}$.

Lo studente può verificare facilmente che l'algoritmo tradizionale che calcola il prodotto delle due matrici richiede $O(n^3)$ operazioni aritmetiche (somme e prodotti).

E 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 una matrice a blocchi):

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

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 ad un primo algoritmo ricorsivo, come segue:

1. **Se A e B hanno dimensione $n = 1$**
2. **Allora moltiplica (banalmente) le due matrici**
3. **Altrimenti**
4. **Spezza 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 2 moltiplicazioni di matrici quadrate di dimensione $n/2$ ed 1 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 4 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 $T(n) = O(n^3)$. Pertanto tale algoritmo non presenta alcun vantaggio sull'algoritmo tradizionale.

L'idea di Strassen

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

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

Lo studente, se lo desidera, può verificare che:

$$C_{11} = P_1 + P_4 - P_5 + P_7$$

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

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:

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

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

Nel corso delle lezioni sul calcolo della complessità di un algoritmo sono state date delle semplici regole per calcolare la complessità di un algoritmo iterativo espresso utilizzando 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.

Per mostrare come applicare tali regole, sono stati utilizzati come esempi gli algoritmi di *bubble sort*, *insertion sort* e *moltiplicazione di matrici*.

Le tecniche per analizzare la complessità di algoritmi ricorsivi sono molto differenti dalle tecniche viste finora, ed utilizzano a tal fine delle relazioni di ricorrenza. Le tecniche per risolvere tali relazioni di ricorrenza sono molto spesso ad hoc, e ricordano piuttosto vagamente le tecniche per risolvere le equazioni differenziali. Non è richiesto allo studente alcuna conoscenza di equazioni differenziali o di matematiche superiori per poter comprendere le tecniche base che saranno trattate nelle seguenti due lezioni. L'unico pre-requisito matematico è la conoscenza della formula che esprime la somma di una serie geometrica finita, formula che verrà richiamata durante la lezione.

8.1 Introduzione

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, \dots, P_m che si richiamano ricorsivamente fra loro.

Vogliamo quindi stimare, per ogni $i = 1, 2, \dots, m$ la funzione $T_i(n)$ che rappresenta il tempo di calcolo impiegato dalla i -ma procedura su dati di dimensione n . Se ogni procedura richiama le altre su dati di dimensione minore, sarà possibile esprimere $T_i(n)$ come funzione dei valori $T_j(k)$ tali che $\forall j \in \{1, 2, \dots, m\} : k < n$.

Per semplificare il discorso, supponiamo 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 corso $T(n)$ 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.

Equazioni di ricorrenza

Diciamo che una sequenza a_1, a_2, \dots 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

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

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

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, \dots, 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:

$$1, 2, 6, 24, \dots$$

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

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

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

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

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.

Mini esempio: Consideriamo la seguente equazione di ricorrenza:

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

dove c è una costante assegnata. Evidentemente, l'unica soluzione con la condizione al contorno $a_0 = 1$ è data da $a_n = c^n$.

Le equazioni di ricorrenza serviranno ad analizzare la complessità degli algoritmi ricorsivi.

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 di esecuzione dell'algoritmo su un problema di dimensione 2 sia dato da una costante c . Pertanto otteniamo la seguente sequenza:

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

ovvero, se utilizziamo la notazione precedentemente introdotta, otteniamo una sequenza a_1, a_2, \dots 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 relazioni di tali relazioni di ricorrenza.

8.1.1 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 complessità di tale procedura? 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 (3,4);
- 3 Tempo $T(n - i - 1)$ per i passi (5,6)

da cui si ricava la seguente equazione di ricorrenza:

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

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 "attaccare" le equazioni di ricorrenza.

1 La forma generale della soluzione potrebbe essere nota.

In questo caso, occorre semplicemente determinare i parametri incogniti.

Esempio 19 Sia data l'equazione:

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

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:

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

da cui

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

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:

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

Si ha allora

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

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

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

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) ciascuna 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:

$$T(n) = a T(n/b) + d(n) \quad (n = b^k \quad k > 0) \quad (8.1)$$

$$T(1) = c \quad (8.2)$$

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

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

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

Teorema Principale Si dimostra che:

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

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

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

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:

$$\begin{aligned} T(1) &= c \\ T(n) &= 2 T(n/2) + f n \quad n > 1 \end{aligned}$$

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 $T(n) = U(n) f$. Possiamo scrivere quindi

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

da cui

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

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

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

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

da cui discende che

$$\begin{aligned} T(n) &= aT\left(\frac{n}{b}\right) + d(n) \\ &= a \left[aT\left(\frac{n}{b^2}\right) + d\left(\frac{n}{b}\right) \right] + d(n) \end{aligned}$$

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

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:

$$T(n) = c n^{\log_b a} + \sum_{j=0}^{k-1} a^j d(b^{k-j})$$

Soluz.Omogenea: Soluz.Particolare:

tempo per risolvere tempo per combinare

i sottoproblemi 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

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

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

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

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

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

ovvero

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

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

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

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

Si ha

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

da cui

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

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

8.3.3 Esempi

| $T(1)$ | $=$ | c | a | b | $d(b)$ |
|----------|-----|-------------------------|-----|-----|--------|
| $T_1(n)$ | $=$ | $2T(\frac{n}{2}) + n$ | 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:

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

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

Capitolo 9

Liste, Pile e Code

Si consiglia **fortemente** allo studente di studiare dal testo [4] le sezioni 1, 2, 3 e 4 del capitolo 2.

In alternativa é possibile studiare le sezioni 1, 2 e 3 del capitolo 11 del testo [2].

Capitolo 10

Grafi e loro rappresentazione in memoria

Si consiglia **fortemente** allo studente di studiare dal testo [4]

- le sezioni 1 e 2 del capitolo 6;
- la sezione 1 del capitolo 7.

Si richiede inoltre di implementare in C o C++ gli algoritmi base descritti in tali sezioni.

Capitolo 11

Programmazione Dinamica

11.1 Introduzione

Spesso, la soluzione di un problema assegnato può essere ricondotta alla soluzione di problemi simili, aventi dimensione minore della dimensione del problema originale. E' 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.

11.1.1 Un caso notevole

Si consideri la moltiplicazione di n matrici

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

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

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

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

Esempio 21 Consideriamo il prodotto

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

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.

11.1.2 Descrizione del metodo

Sia

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

il costo minimo del calcolo di

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

Si ha la seguente equazione di ricorrenza:

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

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

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

sono necessarie:

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

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

11.1.3 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}

11.1.4 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 $m_{ij} := \min_{i \leq k \leq j} (m_{ik} + m_{k+1,j} + r_{i-1} \cdot r_k \cdot r_j)$
7. Restituisci m_{1n}

Nota bene 7 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

11.1.5 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 12

Dizionari

Si consiglia **fortemente** allo studente di studiare dal testo [4] le sezioni 1, 3, 4, 5 e 6 del capitolo 4.

Capitolo 13

Alberi

Si consiglia **fortemente** allo studente di studiare dal testo [4] le sezioni 1, 2, 3 e 4 del capitolo 3 (omettere il paragrafo "An Example: Huffman Codes").

In alternativa é possibile studiare dal testo [2] le sezioni 4 e 5 del capitolo 5.

Capitolo 14

Alberi di Ricerca Binari

Si consiglia **fortemente** allo studente di studiare dal testo [4] le sezioni 1 e 2 del capitolo 5.

In alternativa, é possibile studiare le sezioni 1, 2, 3 e 4 del capitolo 13 del testo [2].

Nota bene 8 *La trattazione dell'analisi di complessit  nel testo [4] (sezione 2, p. 160) e' molto pi  accessibile.*

Capitolo 15

Alberi AVL

Si consiglia **fortemente** allo studente di studiare dal testo [1] la sezione 6 del capitolo 13.

Capitolo 16

2-3 Alberi e B-Alberi

Si consiglia **fortemente** allo studente di studiare dal testo [4]

- la sezione 4 del capitolo 5;
- dalla sezione 4 del capitolo 11 i paragrafi "Multiway Search Trees", "B-trees" e "Time Analysis of B-tree Operations".

In alternativa é possibile studiare il capitolo 19 del testo [3].

Capitolo 17

Le heaps

17.1 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*. Assegniamo 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, come al solito. Una coda con priorità A è un sottinsieme di U su cui sono ammesse le seguenti operazioni:

- **Inizializza**(A)
crea una cosa 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.

17.2 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.
- 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:

| <i>NODI</i> | <i>POSIZIONE</i> |
|------------------------|-------------------------------|
| Radice | 1 |
| nodo p | i |
| padre di p | $\lfloor \frac{i}{2} \rfloor$ |
| figlio sinistro di p | $2i$ |
| figlio destro di p | $2i + 1$ |

Come al solito, 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 .

17.3 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 $\mathcal{O}(1)$.

17.4 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.

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. $\text{Nodi}[n] := \text{Chiave}$
3. $\text{SpostaSu}(n)$

Procedura SpostaSu(i):

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

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

17.5 Cancellazione del minimo

La cancellazione del più piccolo elemento di una heap, che sappiamo trovarsi 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. $\text{Nodi}[1] := \text{Nodi}[n]$
2. $n := n - 1$
3. $\text{SpostaGiu}(1)$

Procedura SpostaGiu(i):

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

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

17.6 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.

9 Attenzione

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 di **Nodi** ha $l(v) = l$, 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 $T(n) = \mathcal{O}(n)$.

17.7 Esercizi

- 1 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 18

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 più piccolo elemento finché la heap risulti vuota.

Heapsort(S):

1. **CostruisciHeap(S)**
2. **Per i che va da 1 a n**
3. **$A[i] = \text{Minimo}()$**
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 $O(n \log n)$.

Capitolo 19

Tecniche Hash

Alle tecniche hash saranno dedicate 2 lezioni da (circa) 2 ore ciascuna all'interno di un corso introduttivo di "Algoritmi e Strutture Dati" della durata di 120 ore.

- 1 Nella prima lezione saranno presentati i concetti fondamentali, e la classificazione delle diverse tecniche. In tale lezione verrà omesso completamente il problema della cancellazione negli schemi ad indirizzamento aperto.
- 2 Nella seconda lezione sarà mostrata l'implementazione effettiva delle procedure di inserimento, cancellazione e ricerca nel caso degli schemi ad indirizzamento aperto, ricorrendo all'uso di pseudo-codice laddove necessario. In particolare verrà posto l'accento sui problemi legati alla cancellazione di un elemento.

Quindi verrà mostrata la complessità nel caso peggiore e nel caso medio delle operazioni elementari di inserimento, ricerca e cancellazione. Per semplicità l'analisi di complessità verrà effettuata per i soli schemi a concatenamento.

Lo studente dovrà avere appreso dalle lezioni precedenti:

- Il concetto di *lista concatenata*,
come (possibile) struttura dati per implementare il tipo di dato astratto *lista*;
- Il concetto di *insieme*,
come tipo di dato astratto;

- Il concetto di *dizionario*, come tipo di dato astratto, ottenuto specializzando il tipo di dato astratto insieme;
- Come implementare un dizionario utilizzando un *vettore di bit*.

19.1 Introduzione

Abbiamo visto nelle lezioni precedenti che il tipo di dato astratto *dizionario* è uno tra i 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.

Le tabelle hash consentono di effettuare efficientemente queste tre operazioni, nel caso medio. In particolare dimostreremo che tali operazioni richiedono tempo medio costante, 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 in un intervallo predefinito $[0, m - 1]$.

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 i records del nostro dizionario. I records sono inseriti nella tabella (e quindi possono essere successivamente trovati)

utilizzando la funzione hash per calcolare l'indice nella tabella a partire dalla chiave del record.

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 tutti i record che dovrebbero occupare la stessa posizione nella tabella hash formano una lista concatenata.

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

Negli schemi ad indirizzamento aperto 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 $\{0, \dots, m-1\}$, 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 dei records in essa memorizzati. 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.

Verrà dimostrato al termine della lezione che l'efficienza di queste tecniche dipende fondamentalmente dal fattore di carico della tabella: in altre parole, quanto più è piena la tabella, tante più operazioni di scansione occorre effettuare per cercare la nostra chiave o per cercare una posizione libera in cui inserire una nuova chiave.

19.2 Caratteristiche delle funzioni hash

Ricordiamo che una funzione hash h dovrà consentire di mappare l'insieme universo U sull'insieme $[0, \dots, m-1]$ 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

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

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

In altri termini, per una tale funzione *hash*, la preimmagine di ciascun indice della tabella T dovrà avere all'incirca cardinalità $|U|/|T|$ (i sinonimi partizionano l'insieme delle chiavi in sottinsiemi di cardinalità approssimativamente uguale).

Chiavi intere

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

$$h(k) = k \bmod m \qquad k \in U$$

Chiavi alfanumeriche

Se l'universo U è costituito da stringhe alfanumeriche, possiamo ad esempio 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, \dots, 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).

19.3 Tecniche di scansione della tabella

Qui di seguito elenchiamo alcune delle tecniche maggiormente usate di scansione della tabella, 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*).

19.3.1 Scansione uniforme

La scansione uniforme è uno schema di indirizzamento aperto che risolve le collisioni scandendo la tabella secondo una permutazione degli interi $[0, \dots, m-1]$ 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.

19.3.2 Scansione lineare

In tale schema la scansione avviene considerando di volta in volta la prossima locazione della tabella \pmod{m} . In altre parole, la scansione della tabella avviene sequenzialmente, a partire dall'indice ottenuto attraverso l'applicazione della funzione hash alla chiave, finché non si raggiunga la fine della tabella, riprendendo in tal caso la scansione a partire dall'inizio della tabella.

Questo metodo utilizza un solo percorso circolare, di lunghezza m .

Esempio 23 Sia $m = |T| = 10$ ed $h(x) = 3$. 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.

19.3.3 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 $h_1(\cdot)$. 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 $h(\cdot)$. 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 24 Sia $m = |T| = 10$, $h(x) = 3$ ed $h_1(x) = 2$. 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 25 Sia $m = |T| = 11$, $h(x) = 3$ ed $h_1(x) = 2$. Il percorso sarà

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

19.3.4 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:

$$\begin{aligned} h(k) & \pmod{m}, \\ h(k) + 1 & \pmod{m}, \\ h(k) + 4 & \pmod{m}, \\ h(k) + 9 & \pmod{m}, \\ h(x) + \dots & \pmod{m} \end{aligned}$$

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 metà della dimensione della tabella.

Esempio 26 Sia $m = |T| = 11$ ed $h(x) = 3$. Il percorso sarà

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

19.4 Hashing tramite concatenamento diretto

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 records bensì puntatori a liste concatenate di record 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 sull'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 un record avviene molto efficientemente, semplicemente eliminando tale record 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

E' una semplice variante del metodo precedente, in cui la tabella contiene sia i record che i puntatori a catene di record. Le collisioni sono gestite attraverso l'uso di liste concatenate.

19.5 Hashing perfetto

Una funzione hash perfetta è una funzione che 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).

19.6 Implementazione pratica di uno schema ad indirizzamento aperto

Per implementare una tabella hash, occorre affrontare due problemi:

- 1 Trovare una buona funzione HASH.
- 2 Gestire efficientemente le collisioni.

Indichiamo con $|T| = m$ il numero totale dei record memorizzabili, e con n il numero di record attualmente memorizzati. Ricordiamo che si definisce densità di carico della Tabella il rapporto $\alpha = \frac{n}{m}$. E' chiaro che in uno schema ad indirizzamento aperto si avrà necessariamente $0 \leq \alpha \leq 1$.

Abbiamo visto nell'introduzione che la scelta di una funzione hash dipende dall'applicazione. Supponendo di considerare la nostra chiave come un numero costituito da una sequenza di cifre in base b , abbiamo in aggiunta a quanto già visto le seguenti notevoli funzioni hash:

1 Estrazione di k cifre

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

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

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

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

Esempio 27

$$\left[\begin{array}{l} x = 235221 \quad b = 10 \quad B = 3 \quad m = 50 \\ \Rightarrow h(x) = 2 \cdot 3^5 + 3 \cdot 3^4 + 5 \cdot 3^3 + 2 \cdot 3^2 + 2 \cdot 3^1 + 1 \cdot 3^0 \pmod{50} \end{array} \right.$$

3 Folding

Si divide la chiave originale in vari blocchi e se ne calcola la somma modulo m :

$$\begin{aligned} X &= x_1|x_2|x_3|x_4 \Rightarrow h(x) = x_1 + x_2 + x_3 + x_4 \pmod{m} \\ x &= 23|52|21 \Rightarrow h(x) = 23 + 52 + 21 \pmod{m} \end{aligned}$$

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 un record. In tal caso occorre modificare opportunamente la tabella per far sì che l'algoritmo di ricerca non si arresti, erroneamente, in presenza di un record cancellato.

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

- **chiave**
utilizzata per identificare un record nel 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 un record.

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

Per evitare di appesantire inutilmente la descrizione, abbiamo ommesso il caso in cui la scansione ci riporti alla celletta da cui siamo partiti. Tale eccezione va ovviamente gestita opportunamente durante la procedure di inserimento, segnalando in tal caso l'impossibilità di inserire il nuovo record, e durante la procedura di ricerca, segnalando in tal caso che il record 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 il record 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.

19.7 Analisi di complessità

Si assumino le seguenti ipotesi:

- 1 Le n chiavi x_1, \dots, 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 f 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 ($\mathcal{O}(1)$ passi);

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.

Abbiamo il seguente risultato:

| | U_n | S_n |
|-----------------------------------|---|---|
| ind. aperto con scansione lineare | $1 + \frac{1}{2}(1 + \frac{1}{(1-\alpha)^2})$ | $1 + \frac{1}{2}(1 + \frac{1}{1-\alpha})$ |
| concatenazione | $1 + \alpha$ | $1 + \frac{\alpha}{2}$ |

19.7.1 Dimostrazione per le tecniche a concatenazione

Se n chiavi sono distribuite uniformemente su m catene allora la lunghezza media di ciascuna catena è proprio $n/m = \alpha$.

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

Se la chiave è invece presente nella tabella, occorrerà un passo per calcolare $h(x)$, più in media $\alpha/2$ passi per trovare la chiave all'interno di una catena di lunghezza α , da cui $S_n = 1 + \alpha/2$.

Riepilogo della complessità

Si noti anzitutto che U_n e S_n sono entrambe $O(\alpha)$.

In caso di inserimento, occorre cercare prima il record ed inserirlo se non sia già presente. Pertanto occorreranno $O(\alpha) + O(1) = O(\alpha)$ passi.

In caso di cancellazione, occorre cercare prima il record e cancellarlo nel caso sia presente. Pertanto occorreranno anche in questo caso $O(\alpha) + O(1) = O(\alpha)$ passi.

Ricapitolando, otteniamo i seguenti tempi:

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

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, adottando tale criterio, $\mathcal{O}(n)$.

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 $m + n$.

19.8 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 Perché 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).

19.9 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 20

Il BucketSort

Ricordiamo che la delimitazione inferiore alla complessità del problema ordinamento nel caso pessimo è $\Omega(n \log n)$. Abbiamo visto nelle lezioni precedenti un algoritmo di ordinamento che ha complessità $\mathcal{O}(n \log n)$ nel caso pessimo, il mergesort. Tale algoritmo è dunque ottimale 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 questa lezione mostreremo un algoritmo di ordinamento che ha complessità $\mathcal{O}(n)$ nel caso medio, il bucket sort. 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.

20.1 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 capitolo relativo all'hash), utilizzando la seguente funzione hash:

$$h(x) = \lfloor (n-1) \frac{x - min}{max - min} \rfloor \quad (20.1)$$

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.

20.1.1 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

$$min$$

e

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

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

- Le chiavi comprese tra

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

e

$$min + 2 \frac{max - min}{n}$$

esclusa, nella celletta 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 celletta 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 così' via...

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

20.1.2 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 $\mathcal{O}(n)$.

Ordinare una singola catena di lunghezza costante costa tempo 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.

20.1.3 Complessità nel caso pessimo

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

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

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.

20.2 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 (20.1)). 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 21

Complessità del problema ordinamento

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 in questo capitolo che non può esistere alcun algoritmo di ordinamento *basato su confronti e scambi* avente complessità nel caso pessimo inferiore a $O(n \log n)$. In altre parole, $\Omega(n \log n)$ rappresenta una delimitazione inferiore alla complessità del problema ordinamento. Poiché il Mergesort ha complessità $O(n \log n)$, deduciamo che:

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

21.1 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.

Ciascuna foglia dell'albero rappresenta una permutazione dei dati in ingresso. Pertanto, un albero decisionale relativo ad un algoritmo di ordinamento di 5 chiavi avrà $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$ foglie. L'altezza h di tale albero rappresenterà pertanto il numero di confronti da effettuare nel caso peggiore. Poiché a ciascun confronto corrisponde al più uno scambio, è evidente che l'altezza h dell'albero decisionale ci darà appunto la complessità dell'algoritmo di ordinamento considerato.

Sia A un algoritmo di ordinamento basato su confronti e scambi. L'albero decisionale associato a tale algoritmo avrà $n!$ 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 dovrà essere almeno $\log_2 n!$. Poiché

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

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 che ordina n elementi avrà altezza $h \geq \frac{n}{2} \log\left(\frac{n}{2}\right)$.

Esempio 28 Albero decisionale per l'Insertion Sort di 3 elementi: k_1, k_2, k_3 (vedi figura a p. 164 del testo [2])

| FOGLIA | PERMUTAZIONE | INPUT DI ESEMPIO |
|------------|---------------------------|------------------|
| <i>I</i> | $k_1 \quad k_2 \quad k_3$ | 7, 9, 10 |
| <i>II</i> | $k_1 \quad k_3 \quad k_2$ | 7, 10, 9 |
| <i>III</i> | $k_3 \quad k_1 \quad k_2$ | 9, 10, 7 |
| <i>IV</i> | $k_2 \quad k_1 \quad k_3$ | 9, 7, 10 |
| <i>V</i> | $k_2 \quad k_3 \quad k_1$ | 10, 7, 9 |
| <i>VI</i> | $k_3 \quad k_2 \quad k_1$ | 10, 9, 7 |

Nel caso peggiore $T(n) = h$, con h =altezza dell'albero.

$$\begin{aligned}\#foglie &= \# \text{ tutte le possibili permutazioni di } k_1, k_2, \dots, k_n \\ &= n \cdot (n-1) \cdot \dots \cdot 3 \cdot 2 \cdot 1 \\ &= n!\end{aligned}$$

Capitolo 22

Selezione in tempo lineare

22.1 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, e' la selezione del k -mo 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 17 *Sia A un insieme di n elementi tratti da un insieme universo U totalmente ordinato. Il k elemento di A , con $1 \leq k \leq n$, e' quell'elemento x di A tale che al piu' $k - 1$ elementi di A siano piu' strettamente minori di x ed almeno k elementi di A siano minori o uguali di x .*

In base a tale definizione, l'elemento 8 e' il terzo ed il quarto piu' 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 Selezione(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? Cio' equivale a chiedere se la complessita' di tale problema sia inferiore a $\Omega(n \log n)$, la complessita' del problema ordinamento. La risposta e' affermativa!

22.2 Un algoritmo ottimale

L'algoritmo mostrato in tale capitolo ha complessita' $O(n)$. Poiche' ovviamente nessun algoritmo di selezione potra' avere mai tempo di esecuzione sub-lineare, dovendo prendere in considerazione ciascun elemento dell'insieme almeno una volta, l'algoritmo qui di seguito presentato e' 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 $m = \text{Seleziona}(\lceil \frac{|M|}{2} \rceil, M)$
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 $x = \text{Seleziona}(k, A_1)$
13. Restituisci x
14. Altrimenti
15. Se $|A_1| + |A_2| \geq k$
16. Allora
17. Restituisci m
18. Altrimenti
19. Poni $x = \text{Seleziona}(k - |A_1| - |A_2|, A_3)$
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 complessita'

Immaginiamo di disporre le $\lceil \frac{|A|}{5} \rceil$ 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

occupera' la posizione centrale

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

della terza riga. Si verifica ora facilmente che:

- Un quarto degli elementi 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 cio' deduciamo che:

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

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 $T(n)$ e' costante. Se $|A| \geq 50$ invece occorre:

- Costruire l'insieme M , e cio' richiede tempo lineare;
- Selezionare m , la mediana delle mediane, e cio' richiede tempo $T(n/5)$;
- Costruire i tre sottinsiemi A_1 , A_2 , A_3 di A , e cio' richiede tempo lineare;
- Invocare ricorsivamente l'algoritmo su un insieme di cardinalita' pari ad al piu' tre quarti della cardinalita' di A (ricordiamo che la nostra analisi e' relativa al caso peggiore).

Da cio' deduciamo la seguente relazione di ricorrenza:

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

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

L'asserzione e' sicuramente vera se $n < 50$; infatti in tal caso il tempo richiesto e' 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:

$$\begin{aligned} T(n) &\leq T\left(\frac{n}{5}\right) + T\left(\frac{3}{4}n\right) + c n \\ &\leq 20 c \frac{n}{5} + 20 c \frac{3}{4} n + c n \\ &= \left(\frac{20}{5} + \frac{20 \cdot 3}{4} + 1\right) c n \\ &= \frac{80 + 300 + 20}{20} c n \\ &= 20 c n \end{aligned}$$

come volevasi dimostrare.

Capitolo 23

Algoritmi su grafi

Si consiglia **fortemente** allo studente di studiare dal testo [4]

- le pagine 212-213 (il paragrafo "Transitive Closure")
- la sezione 5 del capitolo 6;
- le pagine 221-222 (i paragrafi "Test for Acyclicity" e "Topological Sort").

In alternativa é possibile studiare

- le sezioni 1, 3 e 4 del capitolo 23;
- le pagine 536 e 537 (paragrafo "Chiusura transitiva di un grafo orientato")

dal testo [3].

Bibliografia

- [1] *E. Lodi, G. Pacini* **Introduzione alle strutture dati**, Bollati Boringhieri, 1990
- [2] *T.H. Cormen, C.E. Leiserson, R.L. Rivest*, **Introduzione agli algoritmi**, Vol. 1, Jackson Libri, 1995
- [3] *T.H. Cormen, C.E. Leiserson, R.L. Rivest*, **Introduzione agli algoritmi**, Vol. 2, Jackson Libri, 1995
- [4] *A.V. Aho, J.E Hopcroft, J.D. Ullman*, **Data structures and algorithms**, Addison Wesley, 1983
- [5] *A.A. Bertossi*, **Strutture, algoritmi, complessita'**, Ed. Culturali Internazionali Genova, 1990