Università degli Studi di Udine
Facoltà di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea e Diploma in Informatica

Laboratorio di Informatica
Algoritmi e Strutture Dati

Simone Martini

con la collaborazione per il Capitolo 3 di

Nadia Ugel
Loris Vicario

Dispense ad uso esclusivo degli studenti
prodotte con il contributo del Progetto CAMPUS-CRUI

November 9, 2001

Contents

Chapter 1

Complessità e prestazioni

1.1 Introduzione

Vogliamo introdurre in questo capitolo alcuni dei concetti fondamentali dello studio analitico e sperimentale del tempo d'esecuzione di un programma. Inizieremo discutendo i concetti di dimensione dei dati di un algoritmo e di tempo d'esecuzione migliore e peggiore. Cercheremo poi di motivare una comune ipotesi di lavoro dell'analisi dei programmi, ovvero che il tempo d'esecuzione di una qualsiasi istruzione elementare che non contenga chiamate di funzione è una costante. Introdurremo infine la questione della valutazione sperimentale delle prestazioni di un algoritmo, discutendo il problema dell'errore di misurazione ed alcune tecniche per misurare eventi di breve durata.

Per una buona comprensione del materiale è necessario conoscere il concetto di complessità asintotica e le relative notazioni (O-grande).

1.2 Tempo d'esecuzione di un programma

Tra gli oggetti di studio di questo testo, il *tempo d'esecuzione* di un programma riveste un'importanza particolare. Supporremo fissato sin dall'inizio un linguaggio di programmazione (per noi sarà Pascal, o C, ma qualsiasi altro linguaggio va bene) e considereremo programmi scritti in quel linguaggio. Dato dunque un programma P da eseguirsi sui dati di ingresso d, siamo interessati a determinare il tempo che trascorre dall'inizio dell'esecuzione di P su d sino al momento in cui l'esecuzione termina; per il momento possiamo pensare di indicare questo valore con $\mathcal{T}_P(d)$. In particolare, ci interesserà studiare $\mathcal{T}_P(d)$ al variare di d, determinando, se possibile, una dipendenza funzionale esplicita.

Osserviamo, per cominciare, che sebbene il tempo d'esecuzione di P dipenda dai dati d, raramente dipende da tutte le informazioni contenute in d. Ad esempio, il tempo di esecuzione di un programma per la somma degli elementi di un vettore di interi non dipende dagli elementi contenuti nel vettore, ma solo dal loro numero.[1] Chiamiamo *dimensione* dei dati quegli aspetti di essi che influenzano il tempo d'esecuzione di un programma. Nel caso della somma di un vettore, dunque, la dimensione dei dati è costituita dal numero di elementi del vettore stesso. In altri casi, determinare quale sia la dimensione dei dati può essere meno ovvio.

Esempio: 1.2.1 Ricerca lineare incerta
Consideriamo il programma della Figura 1.1. Si tratta di una semplice ricerca lineare incerta: si vuole determinare se il valore x compare nel vettore V, di dimensione n; in caso positivo, viene stampato l'indice i per cui $V[i] = x$; altrimenti viene stampato $n + 1$. I numeri di linea riportati nella figura non fanno parte del programma e sono stati inseriti solo per facilitare il riferimento ai singoli comandi.

[1] Non prendiamo in considerazione qui il caso (solo teorico) in cui gli elementi del vettore possono essere arbitrariamente grandi (cioè non limitati dalle dimensioni della parola di macchina).

```
program RicLinIncerta (input, output);
    const
        n = . . .;
    type
        Vettore = array[1..n] of integer;
    var
        V: Vettore;
        i: integer;
    begin
{0}  {inserire qui istruzioni per leggere V e x}
{1}  i := 1;
{2}  while (i < n) and (V[i] <> x) do
{3}      i := i + 1;
{4}  if V[i] <> x then
{5}      i := n + 1;
{6}  write(i)
    end.
```

Figure 1.1: Ricerca Lineare Incerta

Il tempo di esecuzione di questo programma dipende da due fattori: la lunghezza n del vettore V e la posizione di x in V. D'altra parte, determinare la posizione di x in V è proprio lo scopo per cui il programma è stato scritto. Appare pertanto irragionevole inserire questo aspetto dei dati nella loro dimensione (equivarrebbe a supporre risolto il problema). Anche in questo caso, dunque, una ragionevole scelta per la dimensione dei dati è la lunghezza del vettore, n. □

Per programmi che manipolano strutture dati, la dimensione delle strutture in gioco è spesso una buona scelta quale dimensione dei dati. Ad esempio, nel caso di un programma di ordinamento, il numero di dati da ordinare costituirà la scelta ovvia. In altri casi, la dimensione può dipendere dal valore dei dati. Un esempio di questo tipo è dato da un programma per determinare l'n-simo numero primo. Siccome ci aspettiamo che il tempo sia una funzione crescente di n, appare naturale far dipendere dal valore n la dimensione dei dati.[2]

Come altro esempio, consideriamo un compilatore, cioè quel programma che, letta una stringa corrispondente (si spera) ad un programma in un linguaggio ad alto livello, genera un programma in un altro linguaggio (in genere di livello inferiore). In questo caso una buona nozione di dimensione dei dati è data dalla lunghezza della stringa di ingresso, cioè del programma da compilare.

Consideriamo, infine, il problema del commesso viaggiatore. Vi sono n città, tutte collegate tra loro; la distanza tra la città i e la città j è data da un intero positivo (rappresentabile) $d_{i,j}$. Si vuole determinare il giro più corto delle n città. In questo caso i dati del problema sono costituiti dalla matrice (quadrata $n \times n$) delle distanze: la dimensione dei dati è dunque n^2.

TO DO: Altri esempi qui

L'esempio della ricerca lineare incerta mostra che, sebbene il tempo d'esecuzione dipenda dalla dimensione dei dati, questa non è il solo parametro. Nel caso della ricerca, rimane il fattore, cruciale, della posizione nel vettore dell'elemento da ricercare. È chiaro, in altri termini, che dati con la stessa dimensione (vettori di uguale lunghezza) corrispondono a tempi d'esecuzione diversi, a seconda, appunto, della posizione di

[2] In realtà n è una misura largamente sovrastimata della dimensione dei dati. Infatti, i *dati* in questo caso consistono nella *rappresentazione* (decimale o binaria) del numero n, per la quale sono sufficienti $\log n$ (o $\log_{10} n$) simboli (in questo testo, come in tutta l'informatica, log senza indicazione di base sta per $\log_2$). La dimensione dei dati del problema dei numeri primi dovrebbe dunque essere $\log n$. Questa differenza tra valore di un numero e lunghezza della sua rappresentazione è molto importante nella teoria della complessità, ma non ne avremo bisogno in questo testo.

x nel vettore. Tra tutti i possibili tempi d'esecuzione che si ottengono con dati di una fissata dimensione, due sono i casi più interessanti e studiati.

- Il *caso migliore* (o ottimo) è quello che corrisponde ai dati che generano l'esecuzione più breve. Nell'esempio, si tratta del caso in cui $x = V[1]$. Indicheremo il tempo del caso migliore del programma P su dati di dimensione n con $\mathcal{T}_P^{ott}(n)$.
- Il *caso peggiore* (o pessimo) è quello che corrisponde ai dati che generano l'esecuzione più lunga. Nell'esempio, si tratta del caso in cui x non compare in V. Indicheremo il tempo del caso peggiore del programma P su dati di dimensione n con $\mathcal{T}_P^{pes}(n)$, o più semplicemente $\mathcal{T}_P(n)$

Il tempo del caso migliore dà una limitazione inferiore al tempo di calcolo di un programma. Pur con eccezioni interessanti, spesso non dipende dalla dimensione dei dati e non è di grande aiuto nell'analisi di un programma. Al contrario, il tempo del caso peggiore, o tempo peggiore *tout court*, è di grande importanza, perché stabilisce un limite superiore al tempo necessario al programma per dati della dimensione scelta. È abbastanza frequente che anche per dati che non costituiscono il caso peggiore, il programma si discosti di poco dal suo comportamento peggiore, almeno per dati di grandi dimensioni.

Talvolta sarà interessante studiare anche il tempo d'esecuzione per casi "intermedi", cioè che non sono né quello migliore, né quello peggiore. Introdurremo di volta in volta la notazione che useremo in questi casi. Infine, un altro valore di grande importanza, e che rivestirà un ruolo di primo piano in questo testo, è il *tempo medio* d'esecuzione, $\mathcal{T}_P^{med}(n)$, dove la media è intesa su tutti i dati della stessa dimensione (dunque quelli del caso migliore, del caso peggiore, e di tutti quelli intermedi). Rimandiamo una discussione più approfondita del tempo medio al Capitolo 2.

1.3 Analisi della complessità

Scopo dell'analisi di un programma è quello di determinare (o stimare sperimentalmente) la forma della funzione $\mathcal{T}_P(n)$ al variare di n.

Un momento di riflessione mostra subito, però, che l'effettivo tempo d'esecuzione pessimo $\mathcal{T}_P(n)$ dipende in realtà non solo dai parametri P e n, ma anche da una miriade di altri fattori, solo alcuni dei quali noti al momento in cui un programma viene scritto. Tra questi ricordiamo i seguenti.

1. Il calcolatore su cui il programma viene eseguito: è chiaro che un Pentium a 300Mhz eseguirà P più velocemente di un vecchio 86000 a 16Mhz. Anche due macchine che eseguano (in media) lo stesso numero di istruzioni macchina per secondo eseguiranno P in tempi diversi, perché il repertorio delle istruzioni (e dei modi di indirizzamento) influenza non poco l'efficienza di uno specifico programma.
2. Il compilatore utilizzato: P, per poter essere eseguito su uno specifico calcolatore, deve essere tradotto nel suo linguaggio macchina. Due compilatori diversi per lo stesso linguaggio generano sequenze diverse di istruzioni macchina, e, dunque, influenzano in modo diverso il tempo di esecuzione.[3] Anche se per un certo linguaggio venisse sempre usato lo stesso compilatore, la scelta delle istruzioni macchina (e dunque il tempo di calcolo) potrebbe essere molto diversa per costrutti ad alto livello distinti ma equivalenti (ad esempio un'iterazione determinata espressa in Pascal con il costrutto **for** potrebbe (e spesso è, in specie con i compilatori più semplici) tradotta in modo assai diverso dalla corrispondente iterazione espressa con il costrutto **while**).

[3]Ricordiamo qui, in particolare, i cosiddetti compilatori *ottimizzanti*, che possono riuscire a guadagnare frazioni considerevoli del tempo di calcolo. Tale risultato si ottiene mediante un dettagliato esame del testo del programma, durante il quale si ricercano, tra le altre cose: il migliore assegnamento di registri alle variabili (per minimizzare i tempi di interazione con la memoria); quelle istruzioni che possono essere tolte dall'interno dei cicli (per diminuire il tempo d'esecuzione del corpo); le chiamate a procedure "corte", per sostituirle col loro corpo (per risparmiare il tempo di chiamata e di ritorno). Il prezzo di tali ottimizzazioni si paga in termini di tempo di compilazione. In linea teorica, se si vogliono ottenere tutte le ottimizzazioni possibili e non ci si accontenta di approssimazioni della soluzione "ottima", il tempo di compilazione passa da lineare ad esponenziale nella lunghezza del programma; in pratica, ci si limita a soluzioni approssimate (e spesso solo per alcuni dei problemi cui si è accennato), con un costo complessivo polinomiale. Tali tempi di compilazione sono più che accettabili per programmi che devono essere eseguiti molte volte (ma compilati una volta sola) e per i quali l'efficienza sia un requisito fondamentale.

3. Il carico del sistema: durante l'esecuzione di P possono succedere molte cose, per esempio può arrivare un'interruzione ad alta priorità che richiede di essere gestita. Il fenomeno è ancora più chiaro in un calcolatore (e relativo sistema operativo) che permetta l'esecuzione "simultanea" di più programmi (che si risolve, in realtà, nell'esecuzione a rotazione di piccole quote dei vari programmi in gioco). È chiaro che tutti questi eventi hanno un'influenza tutt'altro che trascurabile su $\mathcal{T}_P(n)$.

4. Il sistema operativo: il modo in cui il sistema operativo gestisce la memoria può avere grande influenza sul tempo d'esecuzione, in specie quando questa richieda grosse quantità di memoria (che potrebbe essere allocata tutta insieme all'inizio dell'esecuzione, oppure per blocchi durante l'esecuzione stessa; la memoria potrebbe essere *virtuale*, cioè vista come illimitata dall'utente, ma gestita dal sistema operativo con accessi alla memoria secondaria).

Ciò che abbiamo espresso come $\mathcal{T}_P(n)$, dunque, è una funzione di molti altri parametri, così tanti che appare difficile poter estrapolare da essa qualche informazione generale. Osserviamo, tuttavia, che se si eccettuano gli ultimi due punti menzionati, gli altri fattori elencati non dipendono dai dati. Sfruttando questa osservazione potremo mostrare che, con ragionevole approssimazione e a meno dell'influenza dei fattori (3) e (4) (e di altri aspetti che dipendano non dal programma ma dal *contesto* nel quale è eseguito), possiamo stabilire per $\mathcal{T}_P(n)$ la relazione

$$a_1 \cdot T_P(n) + b_1 \leq \mathcal{T}_P(n) \leq a_2 \cdot T_P(n) + b_2 \tag{1.1}$$

dove $T_P(n)$ dipende *solo* da P e dalla dimensione dei dati n, mentre a_1, a_2, b_1, b_2 sono costanti che dipendono da P, dal compilatore e dal calcolatore utilizzati, ma non dai dati.

1.3.1 Passi di calcolo

Per giustificare la relazione (1.1), prendiamo in considerazione due assegnamenti $x := E_1$ e $x := E_2$, dove E_1 ed E_2 sono espressioni senza chiamate di funzione. Fissati un compilatore ed un calcolatore, questi due assegnamenti saranno eseguiti con tempi diversi, ma che non dipendono dal valore di E_1 ed E_2 (ma solo dal loro tipo e dalla loro struttura): eseguire la somma $1 + 1$ oppure la somma $1023 + 1023$ richiede lo stesso tempo, purché operandi e risultato appartengano all'insieme dei valori rappresentabili sul calcolatore scelto (un'ipotesi che sottintenderemo sempre nel seguito). Se indichiamo con t_{min} (rispettivamente t_{max}) il tempo minimo (massimo) necessario ad eseguire uno degli assegnamenti di P, è chiaro che un qualsiasi assegnamento di P viene eseguito in un tempo compreso nell'intervallo $[t_{min}, t_{max}]$, e che tali due valori dipendono solo da P (quali assegnamenti vengono eseguiti), dal compilatore (quali e quante istruzioni macchina sono generate per ogni assegnamento) e dal calcolatore (con quale velocità vengono eseguite le istruzioni).

Un ragionamento analogo può essere compiuto per le istruzioni di ingresso (*read*) e di uscita (*write*): il loro tempo d'esecuzione non dipende dai valori letti o stampati, ma solo dal tipo dei valori in gioco e, nel caso della stampa, della complessità delle espressioni di cui si vuole stampare il valore.[4]

Possiamo dunque affermare che per un programma P di k istruzioni che siano tutte assegnamenti o ingresso/uscita vale la relazione

$$k \cdot t_{min} + b \leq \mathcal{T}_P(d) \leq k \cdot t_{max} + b,$$

dove la costante b tiene conto di fattori quali l'allocazione della memoria per i dati, il caricamento del programma, e così via.

Nel caso di un semplice programma senza cicli o condizionali abbiamo dunque stabilito la (1.1), con $a_1 = t_{min}$, $a_2 = t_{max}$ e $T_P(n) = k$ (si osservi che, in questo caso, $T_P(n)$ non dipende da n). A meno dunque di una costante, *possiamo considerare equivalenti tutte le istruzioni di assegnamento e di ingresso/uscita.* Ognuna di esse contribuisce con un'unità alla quantità $T_P(n)$: chiamiamo *passo di calcolo* questa unità (convenzionale) di tempo. Il tempo effettivo del caso peggiore, $\mathcal{T}_P(n)$, si ottiene

[4] L'osservazione non è vera se il linguaggio permette di leggere e/o scrivere stringhe di lunghezza arbitraria.

moltiplicando la somma dei passi di calcolo $T_P(n)$ per un'opportuna costante che dipende dal tipo delle operazioni in gioco.

Consideriamo ora programmi più complessi, in cui compaiano cicli e comandi condizionali (*if–then–else*).

Supponiamo di avere a che fare con un ciclo

$$P \equiv \texttt{while } E \texttt{ do } C$$

il cui corpo sia costituito solo da assegnamenti e/o istruzioni di ingresso/uscita. Per il ragionamento precedente, sappiamo che, per opportune costanti a_i, b_j, vale la relazione

$$a_1 \cdot T_C(n) + b_1 \leq \mathcal{T}_C(n) \leq a_2 \cdot T_C(n) + b_2. \tag{1.2}$$

Per la valutazione della guardia E (purché non contenga chiamate di funzione) possiamo ripetere quanto osservato per un assegnamento: il suo tempo di esecuzione non dipende dal valore dei dati e, dunque, possiamo considerarlo con un singolo passo di calcolo. Stiamo studiando il caso peggiore del nostro programma P: sappiamo pertanto che il ciclo sarà eseguito, al massimo, un certo numero di volte, che dipenderà, in generale, da n. Supponiamo dunque che, nel caso peggiore, il corpo C del ciclo P sia eseguito $f(n)$ volte; la guardia sarà valutata $f(n) + 1$ volte (l'ultima volta sarà falsa). Ne segue che

$$\mathcal{T}_P(n) = f(n)(\mathcal{T}_C(n) + 1) + 1.$$

Dalla (1.2), con semplici calcoli e maggiorazioni, si ottiene:

$$\min\{a_1, b_1 + 1\} \cdot (f(n)T_C(n) + f(n)) + 1 \leq \mathcal{T}_P(n) \leq \max\{a_2, b_2 + 1\} \cdot (f(n)T_C(n) + f(n)) + 1,$$

da cui si ottiene la funzione $T_P(n) = f(n)T_C(n) + f(n)$ e le nuove costanti per cui vale la relazione (1.1).

Il caso di un comando condizionale è analogo: la scelta tra i due rami del condizionale, ricordando che stiamo analizzando il caso pessimo, sara' in genere funzione della dimensione dei dati. Con calcoli simili a quelli fatti per i cicli, possiamo concludere che un comando condizionale contribuisce al tempo totale con un passo di calcolo, più i passi necessari all'esecuzione dell'opportuno ramo.

Applicando le regole appena date a partire dai cicli e condizionali più interni, non è difficile calcolare i passi di calcolo di un programma complesso e, nello stesso tempo, convincersi che vale la relazione (1.1):

$$a_1 \cdot T_P(n) + b_1 \leq \mathcal{T}_P(n) \leq a_2 \cdot T_P(n) + b_2$$

dove $T_P(n)$ indica il numero di passi di calcolo necessari all'esecuzione di P sui dati d di dimensione n, mentre le costanti dipendono dal tipo di istruzioni presenti in P (con una certa approssimazione, a_1 (rispettivamente a_2) può essere interpretata come il tempo minimo (massimo) per eseguire un'istruzione elementare, per esempio un assegnamento).

Esempio: 1.3.1 Ricerca lineare incerta
Applichiamo il metodo descritto nel paragrafo precedente per il calcolo del numero dei passi al programma della Figura 1.1. Valuteremo sia $T^{ott}(n)$ che $T^{pes}(n)$. Il caso ottimo si ha, chiaramente, quando x compare in $V[1]$. La tabella seguente riassume i passi con cui le varie linee del programma contribuiscono al tempo d'esecuzione in questo caso, escludendo l'input/output.

LINEA	PASSI	RIPETIZIONI	TOTALE
1	*1*	*1*	*1*
2	*1*	*1*	*1*
3	*1*	*0*	*0*
4	*1*	*1*	*1*
5	*1*	*0*	*0*
PASSI TOTALI NEL CASO MIGLIORE			*4*

L'assegnamento della linea 1 contribuisce con un passo di calcolo. La linea 2, la guardia del ciclo, contribuisce con un passo (quello dell'espressione logica composta), che viene moltiplicato per 1, il numero di volte che la guardia viene valutata nel caso migliore. Il corpo del ciclo (linea 3) è un assegnamento (1 passo) che non viene mai eseguito. Infine, la guardia del comando condizionale della linea 4 contribuisce per un passo di calcolo, mentre il ramo `then`*, che da' un passo di calcolo, non è mai eseguito nel caso migliore. Possiamo pertanto concludere che* $T^{ott}(n) = 3$ *e che l'effettivo tempo d'esecuzione soddisfa, per ogni* n*, la relazione*

$$a_1 \cdot 3 + b_1 \leq \mathcal{T}_P^{ott}(n) \leq a_2 \cdot 3 + b_2$$

Il calcolo del numero dei passi nel caso peggiore fornisce, in modo analogo, la seguente tabella.

LINEA	PASSI	RIPETIZIONI	TOTALE
1	1	1	1
2	1	n	n
3	1	$n-1$	$n-1$
4	1	1	1
5	1	1	1
PASSI TOTALI NEL CASO PEGGIORE			$2n+2$

Dalla tabella deriviamo dunque che $T^{pes}(n) = 2n + 2$. $\square$

1.3.2 Vari compilatori, vari calcolatori

Abbiamo introdotto il concetto di passo di calcolo come strumento per la semplificazione dell'analisi della complessità di un programma. Vogliamo ora mostrare che con il suo uso possiamo astrarre anche da quelle differenze nel tempo di esecuzione che sono indotte dall'uso di calcolatori e compilatori diversi.

Siano infatti C_1 e C_2 due compilatori per il nostro linguaggio. Un assegnamento $x := E$, dove E è un'espressione senza chiamate di funzione, sarà tradotto con k_1 istruzioni macchina da C_1 e con k_2 istruzioni macchina da C_2, ma è chiaro che il *valore* di E non influenza k_1 e k_2. Siamo dunque nella stessa situazione del paragrafo 1.3.1: mentre in quel contesto due assegnamenti diversi differivano nel tempo di calcolo per al più una costante, nella situazione attuale è il tempo d'esecuzione dello stesso assegnamento tradotto con due compilatori diversi a differire per al più una costante. Siccome la combinazione di costanti è ancora una costante, possiamo affermare che, fissato un calcolatore, e scelti arbitrariamente due assegnamenti del nostro programma P e due compilatori per il relativo linguaggio, il tempo di esecuzione dei due assegnamenti differisce per una costante.

Facciamo infine variare anche il calcolatore su cui eseguiamo P. Il passaggio da una macchina all'altra si risolverà in una variazione del tempo d'esecuzione che certo non dipende dal valore dei dati in gioco. Il generico assegnamento sarà eseguito, per esempio, in modo 100 volte più veloce. La conseguenza per i nostri scopi è solo la comparsa di un altro fattore costante (100, nell'esempio) da considerare per ottenere il tempo d'esecuzione effettivo di un assegnamento.

Come già nel Paragrafo 1.3.1, possiamo ripetere le considerazioni appena fatte per le istruzioni di ingresso/uscita, e poi derivare delle regole per derivare il tempo d'esecuzione per programmi più complessi.

In conclusione, a meno di fattori costanti, possiamo determinare il tempo d'esecuzione di un programma P nel caso pessimo (o ottimo, o in un qualsiasi caso intermedio) calcolando il numero di passi di calcolo, che indichiamo con $T_P^{pes}(n)$, o, più semplicemente, con $T_P(n)$. Tale valore limita il tempo effettivo d'esecuzione a meno di fattori costanti, secondo quanto espresso dalla (1.1). La funzione $T_P(n)$ si dice *complessità in tempo* del programma P.

Esempio: 1.3.2 Altri esempi
da fare $\square$

Osservazione Chiamate di funzione
Se un'espressione contiene chiamate di funzione, è chiaro che essa non può essere conteggiata come un solo passo di computazione, ma contribuirà al tempo totale con tanti passi quanti sono necessari per la valutazione della funzione. Si tratta di un valore che, in generale, dipende dalla dimensione dei dati.

Esercizio 1.3.3 Dimostrazione formale
Si dimostri formalmente la relazione (1.1) per un linguaggio (semplificato) a piacere. Si sfrutti il fatto che che ogni programma del linguaggio scelto si può scrivere applicando induttivamente poche regole della grammatica che definisce il linguaggio stesso. Ciò ci fornisce un metodo per dimostrare proprietà di programmi (come la (1.1)): ragionando per induzione *sulla struttura dei programmi. Il caso base e' quello di un comando elementare (un assegnamento o un comando di ingresso uscito) ed è stato già trattato poco sopra. I vari casi del passo induttivo sono analoghi alle considerazioni informali prima fatte per cicli e comandi condizionali.*

1.4 Complessità asintotica

TO DO: richiamare notazione O-grande; esempi;operazioni

1.5 Valutazione delle prestazioni

Abbiamo fatto di tutto, sino a questo punto, per svincolare la complessità di un programma dai fattori collegati a specifici esecutori. In questo processo di progressiva astrazione, la notazione O-grande è uno strumento di formidabile semplificazione e chiarezza. Ci sono situazioni, tuttavia, in cui l'analisi della complessità in termini asintotici non è sufficiente, oppure non adatta.

Un caso in cui è necessario conoscere tempi effettivi, è costituito da quei programmi che devono interagire con l'ambiente entro un dato intervallo di tempo (cioè, secondo la terminologia in uso, in *tempo reale*). In tali situazioni è indispensabile fornire con esattezza una limitazione superiore all'effettivo tempo di calcolo, ovvero determinare con una certa precisione le costanti coinvolte nella relazione (1.1).

In altri casi, si possono presentare algoritmi complessi per i quali una valutazione esatta dei passi di calcolo del caso pessimo non è fattibile o, più semplicemente, non è disponibile. D'altra parte potrebbe essere non difficile una valutazione asintotica del comportamento del programma. Abbiamo già visto come in una limitazione O-grande siano nascosti (almeno) due importanti parametri: la costante con cui si deve moltiplicare la funzione nell'O-grande per ottenere il numero dei passi reali, e il valore n_0 della dimensione dei dati a partire dal quale il comportamento asintotico si manifesta. Spesso non si ha alcuna stima di tali costanti, che, invece, influenzano grandemente l'effettiva efficienza di un programma. A parità di comportamento asintotico, privilegeremo quegli algoritmi che hanno costanti moltiplicative piccole. D'altra parte, un programma asintoticamente lineare, ma che esibisce tale linearità solo per dimensioni dei dati estremamente grandi, mentre per piccole dimensioni ha un costo elevato, potrebbe essere peggiore, in tutti i casi pratici, di un algoritmo quadratico.

Se per alcuni programmi usati di rado una valutazione accurata delle costanti in gioco può apparire inopportuna, vi sono situazioni in cui, al contrario, non solo si vuole determinarle con precisione, ma si vuole anche adottare tutti quegli accorgimenti che, per quanto possibile, le riducano. È il caso di alcuni programmi cosiddetti di sistema, tra cui un ruolo importante è giocato dai programmi di ordinamento e di ricerca in tabelle. I primi sono il cuore di molti programmi di gestione dei dati, per i quali la velocità di risposta è un fattore assai importante, quando non decisivo. La ricerca in tabelle è un ingrediente di molti altri algoritmi o problemi; tanto per fare un esempio, essa compare decine di volte nel corso di una singola compilazione di un programma ad alto livello. È chiaro come, per queste applicazioni, non si vogliono solo usare programmi asintoticamente efficienti, ma davvero "veloci" nei casi in cui saranno applicati più di frequente.

Sono queste alcune delle ragioni per le quali all'analisi asintotica di un programma è talvolta opportuno associare la *valutazione delle prestazioni* del programma stesso, intendendo con questo la rilevazione dell'effettivo tempo d'esecuzione al variare della dimensione dei dati, dopo aver fissato compilatore, esecutore e tutti gli altri parametri al contorno. Quando sia nota l'espressione esatta dei passi di calcolo, la valutazione delle prestazioni di un certo programma permette una stima accurata delle costanti presenti nella disequazione (1.1). Quando sia nota un'analisi asintotica, la valutazione delle prestazioni consente, in genere, di valutare, almeno come ordine di grandezza, la costante moltiplicativa e la dimensione critica dei dati a partire dalla quale si manifesta il comportamento asintotico. La determinazione di queste costanti permette così un'espressione funzionale accurata dell'effettivo tempo di calcolo $\mathcal{T}_P(n)$, che può essere usata per scopi predittivi (cioè per sapere quanto impiegherà il programma su dati di una certa dimensione, su cui non si era fatta una valutazione sperimentale esplicita).

Il primo problema che si presenta con la valutazione delle prestazioni è quello di misurare con una certa precisione il tempo d'esecuzione di un certo programma (o di sue parti). Il problema è assai semplificato in un ambiente ad unico utente, quale un calcolatore personale, nel quale una volta mandato in esecuzione un programma, questo mantiene il controllo del calcolatore sino al termine dell'esecuzione, con l'eccezione del trattamento delle interruzioni. In un contesto del genere, il modo più semplice e diretto, per effettuare le misure che ci interessano è quello di usare esplicitamente l'orologio di macchina, un meccanismo presente su tutti i calcolatori e tutti i sistemi operativi. Si tratta, in genere, di un registro contenente un numero naturale, che viene incrementato ogni δ secondi. Il valore di questo registro viene letto con opportune chiamate di sistema, quasi sempre disponibili nei maggiori linguaggi ad alto livello. La risoluzione dell'orologio, δ, dipende dalla macchina e dal sistema operativo usati e varia da qualche decimo a pochi millesimi di secondo. Leggendo l'orologio prima e dopo l'evento di nostro interesse otteniamo il tempo trascorso, purché l'evento abbia avuto una durata superiore alla risoluzione dell'orologio.

Esempio: 1.5.1 Mac e ThinkPascal
Nel caso dell'ambiente ThinkPascal per Macintosh, si può fare uso di una chiamata alla funzione predefinita `tickcount`*, che fa parte del TimeManager di Macintosh. Ritorna un intero che viene incrementato ogni sessantesimo di secondo. Nei programmi faremo uso della seguente funzione* `Time`*:*

```
function Time: real;
{ Legge l'orologio di macchina, in secondi }
begin
    Time := tickcount / 60
end;
```

Alternativamente, chiameremo direttamente `tickcount`*.* □

Negli esempi che seguiranno, si riporterenno i tempi rilevati su un (vecchio) Macintosh Powerbook 165 (molto lento...).

Esempio: 1.5.2 Ricerca Lineare Incerta
Valutiamo le prestazioni della ricerca lineare incerta, il programma della Figura 1.1, nel caso pessimo, al variare della dimensione dei dati, cioè della dimensione del vettore V. Ci aspettiamo, in accordo alla valutazione di $T_P(n) = 2n + 2$ ottenuta nel Paragrafo 1.3.1, che il tempo d'esecuzione sia lineare. Uno degli scopi principali dell'esperimento sarà quindi quello di calcolare il coefficiente angolare della retta corrispondente. In linea di principio, tale coefficiente è determinabile con due sole osservazioni; è opportuno, tuttavia, non limitarci a due sole misure. In primo luogo, infatti, come tutte le misure sperimentali anche le nostre andranno soggette ad errore ed è quindi opportuno effettuare più rilevazioni e determinare la retta che meglio si adatta (in un senso che sarà precisato in seguito) a tutte le misure compiute. In secondo luogo (anche se questo non è il caso della ricerca lineare incerta, per la quale abbiamo un calcolo esatto dei passi d'esecuzione), scopo di questi esperimenti è spesso anche quello di verificare che il tempo effettivo d'esecuzione si comporti in accordo alla dipendenza funzionale asintotica stabilita in sede di analisi. Effettuare più misure è dunque utile anche per tale verifica.

Misureremo dunque il tempo di calcolo del programma in esame per dimensioni del vettore 10, 20, 30, 40, 50, 100, 500, 1000. È opportuno organizzare l'esperimento nel modo più possibile automatico, per non dover inserire manualmente dati e trascrivere risultati. Una possibile struttura del programma è quella della Figura 1.2. Il programma non presenta difficoltà. Il caso pessimo è garantito andando a ricercare il valore 1 in un vettore precedentemente riempito con 0. I tempi sono rilevati all'interno della procedura di ricerca per non misurare anche il tempo necessario alla chiamata della funzione ed al passaggio dei parametri. □

L'esecuzione del programma della Figura 1.2 produce i risultati della Figura 1.3. I risultati appaiono a prima vista misteriosi e senza parentela con la relazione $T_P(n) = 2n + 2$. La loro spiegazione non è tuttavia difficile: abbiamo cercato di misurare la durata di eventi che sono di gran lunga troppo brevi per la risoluzione del nostro strumento di misura. Per questo abbiamo ottenuto molti tempi pari a zero; il tempo per $n = 500$ è solo "rumore", e dipende probabilmente dal verificarsi di qualche evento al contorno, quale per esempio un'interruzione.

1.5.1 Errori di misurazione

Prima di cercare una soluzione al problema che abbiamo appena incontrato, poniamoci una questione generale: quale deve essere la durata minima di un evento, affinché la si possa misurare con un orologio di risoluzione δ con un errore "ragionevole"? Osserviamo, innanzitutto, che ogni misurazione che fornisce un tempo t corrisponde ad un tempo effettivo che appartiene all'intervallo $[t - \delta, t + \delta]$. La Figura 1.5.1 esemplifica questa situazione: i due eventi A e B sono misurati con un orologio di risoluzione δ che viene incrementato in corrispondenza delle tacche indicate sulla linea orizzontale. La durata di A e B è misurata leggendo l'orologio all'inizio ed alla fine. Le misure di A e B danno luogo agli stessi valori ($t = 2\delta$), ma A è, in realtà, di poco superiore a δ, mentre B è di poco inferiore a 3δ. L'errore di misurazione δ non può essere eliminato se non ricorrendo ad uno strumento più preciso. D'altra parte, una cosa è fare un errore di un millimetro misurando una distanza di due millimetri; altra cosa, avere lo stesso errore misurando una distanza di un metro: nel primo caso la misura presenta un errore percentuale del cinquanta per cento; nel secondo, dell'uno per mille. Se il valore di t e quello di δ sono più o meno della stessa grandezza, l'errore della misurazione è del 100%, una situazione chiaramente inaccettabile. Se $t = 10\delta$, l'errore è del 10%; in generale, se $t = k\delta$, l'errore è del $(100/k)\%$.

Pertanto: per ottenere una misura con errore dello $x\%$ usando un orologio con risoluzione δ occorre misurare un evento della durata di $100/x\delta$.

Per i nostri scopi una stima con errore massimo percentuale del 2% o del 5% è più che sufficiente. Con δ uguale a un sessantesimo di secondo, misureremo eventi dell'ordine di un terzo di secondo per avere errori del 5% e dell'ordine del secondo per un errore del 2%. Se l'orologio ha un risoluzione δ del centesimo di secondo, misureremo eventi di almeno 2 decimi di secondo per un errore del 5% e di almeno 5 decimi per un errore del 2%.

Come fare, dunque, per misurare le prestazioni del nostro programma per la ricerca lineare incerta per valori piccoli di n? La risposta non è difficile: basta ripetere più volte lo stesso evento, diciamo r volte, misurando il tempo trascorso per le r ripetizioni. Il valore r sarà scelto, con qualche tentativo, in modo che i tempi misurati siano dell'ordine di grandezza che ci permette la precisione desiderata. A questo punto una semplice divisione per r ci dà il tempo del singolo evento. Il programma che realizza questa tecnica per la ricerca lineare incerta è mostrato nella Figura 1.5. Si è aggiunto un parametro alla procedura di ricerca per il fattore di ripetizione: il numero di ripetizioni diminuisce al crescere della dimensione del vettore. Il fattore iniziale è stato determinato con qualche prova, in modo da produrre eventi di durata superiore al secondo. I risultati ottenuti con questo programma sono riportati nella Figura 1.6.

Il programma, oltre a stampare il tempo della singola ricerca, stampa anche la durata cumulativa dell'evento misurato, per poter verificare che si tratti di un tempo superiore a quello minimo relativamente allo standard d'errore adottato (nel nostro caso tutti tali tempi sono superiori al secondo). Una semplice verifica "a braccio" ci conforta della ragionevolezza dei dati misurati (cioè che sono effettivamente lineari). Volendo effettuare delle misure più raffinate, occorrerebbe defalcare dai tempi cronometrati la

```
program ValPrestRicLinIncerta (input, output);
  type
    Vettore = array[1..1000] of integer;
  var
    k, j: integer;
    inizio, fine: real;
    V: Vettore;

  function Time: double;
  begin
    Time := tickcount / 60
  end;

  procedure RicLinInc (var V: Vettore; n, x: integer; var i: integer);
  begin
    inizio := Time;
{prendi il tempo d'inizio}
    i := 1;
    while (i < n) and (V[i] <> x) do
      i := i + 1;
    if V[i] <> x then
      i := n + 1;
    fine := Time
{prendi il tempo di fine}
  end;

begin
  showtext;
{ inizializzazione di V }
  for k := 1 to 1000 do
    V[k] := 0;

{chiamate alla procedura di ricerca }
  writeln('  n   ', '  tempo per ricerca  ');
  for k := 1 to 5 do
    begin
      RicLinInc(V, 10 * k, 1, j);
      writeln(10 * k, fine - inizio : 12 : 9)
    end;
  RicLinInc(V, 100, 1, j);
  writeln(100, fine - inizio : 12 : 9);
  RicLinInc(V, 500, 1, j);
  writeln(500, fine - inizio : 12 : 9);
  RicLinInc(V, 1000, 1, j);
  writeln(1000, fine - inizio : 12 : 9);
  writeln('Tempi in secondi')
end.
```

Figure 1.2: Valutazione delle prestazioni della ricerca lineare incerta, I

n	Tempo per ricerca
10	0.000000000
20	0.000000000
30	0.000000000
40	0.000000000
50	0.000000000
100	0.000000000
500	0.000601562
1000	0.000000000
Tempi in secondi	

Figure 1.3: Primo tentativo di valutazione delle prestazioni

Figure 1.4: L'errore di misurazione

durata aggiuntiva introdotta dalle strutture per il controllo dell'esperimento. Si tratta delle chiamate alla funzione `Time` (che richiede tempo sia per la chiamata di funzione in sè che per la chiamata alla funzione di sistema `tickcount`) e del tempo necessario all'incremento della variabile di controllo ed al controllo della guardia del ciclo `for` di ripetizione. Analogamente a quanto fatto per la ricerca, possiamo cercare di misurare questi tempi ripetendo per molte volte gli eventi corrispondenti. I frammenti di programma corrispondenti sono riportati nella Figura 1.7. I risultati ottenuti, in millisecondi, sono i seguenti:

	Tempo globale (sec)	Tempo per operazione (sec)
`Time`	1150.0000	0.2300
`For`	100.0000	0.0030

Una chiamata di `Time` contribuisce per $23 \cdot 10^{-2}$ millisecondi (più correttamente, tenendo conto della precisione dell'orologio, contribuisce per $23 \cdot 10^{-2} \pm 10^3 \dot{1}/(60 * 5000)$ ovvero $23 \cdot 10^{-2} \pm 3 \cdot 10^{-3}$). Per quanto riguarda le strutture di controllo del `for`, la durata complessiva delle `maxint` iterazioni non è sufficiente a dare la sicurezza di una buona misurazione (in termini percentuali), ma in ogni caso, 10000 iterazioni contribuiscono per un tempo stimabile in qualche centesimo di secondo.

Ricapitolando: le due chiamate a `Time` influenzano la durata dell'esperimento in modo del tutto trascurabile rispetto alla precisione dell'orologio. Per quanto riguarda il ciclo di ripetizione, per $n = 10$ si usa un fattore di ripetizione di 20000: il ciclo impiega un tempo pari a circa .06 sec. (≈ 3.5 sessantesimi di secondo); al crescere di n, il ciclo di ripetizione contribuisce per tempi trascurabili. Appare pertanto poco significativo sottrarre esplicitamente tali tempi da quelli riportati nella tabella. Possiamo pertanto concludere che le misure della Figura 1.6 sono, nei limiti della precisione voluta, abbastanza accurate, eccetto al più il valore riportato per $n = 10$, che potrebbe essere influenzato (nella seconda cifra decimale del tempo cumulativo) dalle strutture di controllo per la ripetizione.

Usiamo ora le misure compiute sulla ricerca lineare incerta per determinare la retta corrispondente alle misure stesse. Abbiamo già osservato come non sia opportuno scegliere due misure qualsiasi ed usare la formula della retta passante per due punti, in quanto i risultati da noi ottenuti sono tutte misure sperimentali, e dunque affette da errore. Dobbiamo pertanto andare a cercare la retta che "meglio si adatta" ai dati sperimentali della Figura 1.6. Una tecnica assai usata per questo scopo è quella detta dei *minimi quadrati*, che consiste nel determinare la retta che minimizza la somma delle distanze tra i punti rilevati sperimentalmente e la retta stessa. Il Capitolo 4 discute il problema in dettaglio. Nel nostro caso, i parametri a e b della retta di equazione

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

si ottengono dai dati della Figura 1.6 risolvendo il seguente sistema lineare

$$\begin{pmatrix} 1265500 & 1750 \\ 1750 & 8 \end{pmatrix} \begin{pmatrix} b \\ a \end{pmatrix} = \begin{pmatrix} 11308.736 \\ 15.7128 \end{pmatrix}$$

```
program ValPrestRicLinIncerta (input, output);

type
Vettore = array[1..1000] of integer;
var
k, j: integer;
inizio, fine: real;
V: Vettore;
const
r = 30000; { fattore di ripetizione }

function Time: double;
begin
Time := tickcount / 60
end;

procedure RicLinInc (var V: Vettore; n, x, rip: integer; var i: integer);
var
h, k: integer;
begin
inizio := Time; {prendi il tempo d'inizio}
for h := 1 to rip do { ciclo di ripetizione }
{for k := 1 to rip do}
begin
i := 1;
while (i < n) and (V[i] <> x) do
i := i + 1;
if V[i] <> x then
i := n + 1
end;
fine := Time; {prendi il tempo di fine}
end;

begin
showtext;
{ inizializzazione di V }
for k := 1 to 1000 do
V[k] := 0;

{chiamate alla procedura di ricerca }
writeln(' n ', ' tempo cumulativo ', ' tempo per ricerca ', errore
percentuale ');
for k := 1 to 5 do
begin
RicLinInc(V, 10 * k, 1, r, j);
writeln(10 * k, fine - inizio:12:9, (fine - inizio) / (r div k):12:9,
        100 / (fine - inizio)*60:4:2)
end;
RicLinInc(V, 100, 1, 8000, j);
writeln(100, fine - inizio : 12 : 9, (fine - inizio) / 2000 : 12 : 9,
        100 / (fine - inizio)*60:4:2);
RicLinInc(V, 500, 1, 2000, j);
writeln(500, fine - inizio : 12 : 9, (fine - inizio) / 400 : 12 : 9,
        100 / (fine - inizio)*60:4:2);
RicLinInc(V, 1000, 1, 1000, j);
writeln(1000, fine - inizio : 12 : 9, (fine - inizio) / 200 : 12 : 9,
        100 / (fine - inizio)*60:4:2);
writeln('Tempi in secondi')
end.
```

Figure 1.5: Valutazione delle prestazioni della ricerca lineare incerta, II

n	Tempo cumulativo	Tempo per ricerca	Errore percentuale
10	2100.097656	0.105005	0.79
20	1916.503906	0.191650	0.86
30	1850.097656	0.277542	0.90
40	1833.007812	0.366602	0.90
50	1816.406250	0.454102	0.91
100	1800.292969	0.900146	0.92
500	1799.804688	4.499512	0.92
1000	1783.691406	8.918457	0.93
Tempi in millisecondi			

Figure 1.6: Tempi rilevati per la ricerca lineare incerta

```
program timeFOR;

const
   k = 5000;

  var
    i: integer;
    h, h1, t: double;
  function Time: double;
  begin
    Time := tickcount / 60
  end;

begin
  showtext;
  h := Time;
  for i := 1 to k do
    t := Time;
  h1 := Time;
  t := h1 - h;
  writeln('Time :', t : 12 : 9, t / k : 12 : 9);
  h := Time;
  for i := 1 to maxint do
    ;
  h1 := Time;
  t := h1 - h;
  writeln('For :', t : 12 : 9, t / maxint : 12 : 9);

end.
```

Figure 1.7: Misura tempi di un ciclo `for`.

RETTA APPROSSIMANTE

Figure 1.8: Retta approssimante il tempo d'esecuzione

che ha come soluzione $a = 0.01334$ e $b = 0.008917$, cui corrisponde pertanto la retta

$$T(n) = 0.008917x + 0.01334$$

il cui grafico è mostrato nella Figura 1.8 insieme ai dati della Figura 1.6. Facendo i calcoli, si osserva come la funzione $T(n)$ approssima molto bene i dati sperimentali: i valori di $T(n)$ e quelli riportati nella tabella sono in accordo fino alla seconda cifra decimale. La cosa non deve meravigliare: sappiamo dall'analisi del tempo, infatti, che $T(n)$ è una funzione lineare e dunque l'approssimazione ai minimi quadrati è una buona tecnica. Possiamo sfruttare l'espressione di $T(n)$ per stimare il tempo di calcolo per dimensioni dei dati non comprese nella Tabella 1.6. Ad esempio, otteniamo $T(600) = 5.36354$ e $T(50000) = 445.86334$.

1.5.2 Uso di particolari strumenti

In alternativa alla misura diretta dei tempi con l'orologio di macchina, possiamo far uso di strumenti più sofisticati, che fanno parte dei cosiddetti *ambienti di programmazione*, cioè di insiemi di strumenti e programmi che agevolano la programmazione in un determinato linguaggio. Un tipico ambiente di programmazione può comprendere:

- un *editor* per la scrittura dei programmi, con la possibilità di organizzare automaticamente il testo del programma secondo la sua struttura logica (per esempio rientrando il corpo del ciclo, o allineando i rami di un comando condizionale; questa funzione viene detta *pretty printing*);
- un *debugger*, cioè uno strumento utile per stanare eventuali errori: permette di inserire opportuni punti di controllo nel testo del programma (*check points*), raggiunti i quali l'esecuzione si ferma ed è possibile controllare (o modificare a mano) il valore delle variabili; permette l'esecuzione del programma un'istruzione alla volta; permette di andare a vedere il contenuto della pila di sistema, o delle altre strutture dati di supporto all'esecuzione;
- un *gestore dei progetti* che permette la compilazione separata di moduli del programma e che gestisce eventuali modifiche di essi ricompilando automaticamente solo i moduli modificati e quelli che dipendono da essi;
- un *profiler*, uno strumento che permette di controllare alcuni parametri delle esecuzione, quali il numero di volte che una certa istruzione viene eseguita, il numero di volte che una procedura è chiamata, la frazione di tempo d'esecuzione trascorsa all'interno di una procedura, il tempo totale di calcolo, e così via.

È chiaro che l'uso di un *profiler* è di grande utilità nella valutazione delle prestazioni di un programma. Tanto più utile se siamo interessati ad effettuare la valutazione in un ambiente multiprogrammato (cioè in cui più utenti si contendono il controllo dell'unica cpu, che viene assegnata a rotazione, per piccoli quanti di tempo, a ciascuno di essi). I *profiler* per questi ambienti, infatti, permettono in genere di scorporare dal tempo globale quella frazione in cui la cpu è stata assegnata ad altri compiti, consentendo così di determinare il tempo effettivo trascorso all'interno del codice del programma.

Dal momento che l'uso di un *profiler* è fortemente dipendente dall'ambiente di programmazione e dal sistema operativo utilizzati, nel corso di questo libro useremo quasi sempre misure esplicite del tempo di calcolo, usando l'orologio di macchina. Il lettore che abbia a disposizione un *profiler* è invitato a confrontare i risultati ottenuti con esso con quelli prevenienti dalle misure dirette.

Chapter 2

Tempo medio d'esecuzione

2.1 Introduzione

Tempo medio d'esecuzione tempo medio; semplici derivazioni analitiche metodi MonteCarlo: motivazioni e stima degli intervalli di confidenza

2.2 Tempo medio o tempi medi?

La valutazione (analitica o sperimentale) del tempo peggiore lascia spesso il principiante insoddisfatto. Questi teme, infatti, che l'analisi sia sovradimensionata, e che, nella pratica, il caso peggiore si presenti talmente di rado da rendere la sua valutazione solo un caso da manuale. Ci sono algoritmi in cui la situazione è davvero quella temuta (o forse, meglio, sperata) dal pricipiante[1], e ci sono, al contrario, algoritmi (e sono probabilmente la maggioranza) per i quali la valutazione asintotica del caso peggiore non è molto diversa da quella "media". Ciò non vuol dire, ovviamente, che le costanti in gioco siano le stesse, o che il comportamento asintotico inizi a manifestarsi per dati di dimensione analoga. In questo testo ci occuperemo spesso di valutazioni medie, ed avremo modo, con ciò, di confermare proprio la validità delle valutazioni del caso peggiore.

Il primo problema che si pone con la valutazione del tempo d'esecuzione "in media" è che esso dipende, evidentemente, dai dati che vengono presentati al programma. Mentre, cioè, il caso peggiore (e quello ottimo) è perfettamente determinato, il caso medio non esiste: esistono casi (istanze del problema) che non sono né quello peggiore né quello migliore, e siamo interessati al *tempo medio* necessario ad eseguire il nostro programma al variare dei dati (di dimensione n) tra i due estremi costituiti dal caso migliore e da quello peggiore. D'altra parte, tale tempo medio dipende da quali di questi casi intermedi si presentano più di frequente. È chiaro che se la ricerca lineare incerta è usata nel 50% dei casi per ricercare un elemento che sappiamo appartenere alla prima metè del vettore, il suo tempo medio è migliore (cioè minore) di quello che si otterrebbe con dati di cui sappiamo che l'elemento da ricercare appartiene al vettore solo nel 20% dei casi.

Il tempo medio d'esecuzione di un programma, dunque, ha senso solo dopo aver stabilito una certa *distribuzione dei dati*, cioè con quale frequenza si presentano tutte le varie istanze del problema della fissata dimensione n. Non esiste *il* tempo medio d'esecuzione di un programma, ma solo il tempo medio *supposta una certa distribuzione.*[2]

Il caso più semplice e studiato è quello in cui tutti i dati si suppongono *distribuiti in modo uniforme*, ovvero tutti identicamente probabili. Supponiamo, pertanto, che esistano k_n istanze diverse del problema di

[1] Un caso assai importante è costituito dall'algoritmo di ordinamento Quicksort, che sarà discusso più avanti.

[2] Questo costituisce anche una limitazione delle valutazioni del tempo medio, visto che raramente i dati "nella pratica" si conformeranno a quelle distribuzioni che assumiamo durante la valutazione, in specie in sede di analisi.

```
procedure RicLinCerta (var V: Vettore; n, x: integer; var i: integer);
begin
   i := 1;
   while (i < n) and (V[i] <> x) do
      i := i + 1;
end;
```

Figure 2.1: Ricerca lineare certa

dimensione n, individuate ciascuna da un intero i, $1 \leq i \leq k_n$. Indichiamo con $T_P(n,i)$ il tempo necessario ad eseguire il programma P sull'istanza i (di dimensione n). In ipotesi di distribuzione uniforme, il tempo medio si ottiene con la semplice media aritmetica dei tempi:

$$T_P^{med}(n) = \frac{1}{k_n}\sum_{i=1}^{k_n} T_P(n,i). \tag{2.1}$$

Esempio: 2.2.1 Ricerca lineare certa

La Figura 2.1 riporta una semplice procedura per ricercare un elemento all'interno di un vettore, nell'ipotesi che tale elemento effettivamente compaia all'interno del vettore stesso. Una semplice analisi dei passi di calcolo mostra che $T^{pes}(n) = 2n$, che si ottiene quando x compare come ultimo elemento di V (questa è l'istanza, o caso, peggiore). Le varie istanze dei dati sono determinate dalla posizione con cui x compare in V; possiamo pertanto identificarle con questa informazione. L'istanza 1, dunque, è quella in cui x compare in $V[1]$ (è l'istanza del caso migliore); nell'istanza i, x compare in $V[i]$. Indicando con $T(n,i)$ il numero di passi di calcolo necessari per l'istanza i di dimensione n, un semplice conto mostra che

$$T(n,i) = 2i.$$

Supponiamo ora i dati distribuiti in modo uniforme. Assumiamo, cioè, che i dati vengano presentati al programma in modo che la frequenza del caso "x è in $V[i]$" sia la stessa per ogni i; ovvero, tutti i casi si presentano con la stessa probabilità. *In tali ipotesi il tempo medio d'esecuzione del programma della Figura 2.1 si ottiene con la (2.1), dove, nel caso in esame, $k_n = n$:*

$$T_P^{med}(n) = \frac{1}{n}\sum_{i=1}^{n} 2i.$$

È utile cercare una forma chiusa per il numero medio dei passi di calcolo. In questo caso particolarmente semplice, sfruttando la nota formula

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}, \tag{2.2}$$

si ha

$$T_P^{med}(n) = n + 1.$$

Per questo problema molto semplice, se n è dispari il tempo medio coincide col tempo dell'istanza $\lceil n/2 \rceil$, mentre se n è pari il tempo medio coincide con la media dei tempi delle due istanze $n/2$ e $n/2+1$. Si osservi dunque che, in generale, il tempo medio non *coincide col tempo di calcolo di qualche istanza.* □

Esempio: 2.2.2 Ricerca lineare incerta

Riprendiamo ora la ricerca lineare incerta della Figura 1.1, di cui abbiamo studiato i casi migliore e peggiore nell'Esempio 1.3.1. Vi sono in questo caso $n+1$ istanze diverse di dimensione n: le prime n istanze corrispondono ai casi in cui x compare in una posizione del vettore; l'istanza $n+1$ corrisponde al caso in cui x non è presente in V. Col solito metodo tabellare usato nell'Esempio 1.3.1 otteniamo, per il numero di passi, la seguente tabella

LINEA	PASSI	RIPET. IST. 1	RIPET. IST. $i \leq n$	RIPET. IST. $n+1$
1	*1*	*1*	*1*	*1*
2	*1*	*1*	i	n
3	*1*	*0*	$i-1$	$n-1$
4	*1*	*1*	*1*	*1*
5	*1*	*0*	*0*	*1*
$T(n,i)$		*3*	$2i+1$	$2n+2$

Se ora supponiamo tutte le $n+1$ istanze equiprobabili, possiamo calcolare il numero medio di passi di calcolo usando la (2.1):

$$T^{med}(n) = \frac{1}{n+1}\sum_{i=1}^{n+1} T(n,i) = \frac{1}{n+1}\sum_{i=1}^{n}(2i+1) + \frac{2n+2}{n+1}.$$

Con semplici calcoli, ed usando ancora la (2.2), si ottiene

$$T^{med}(n) = \frac{n}{n+1} + n + \frac{2n+2}{n+1} = n + 3 - \frac{1}{n+1}.$$

La relazione appena ottenuta, lo ricordiamo, è il numero medio di passi di calcolo necessari alla ricerca lineare incerta nell'ipotesi in cui le $k_n = n+1$ istanze di dimensione n sono tutte equiprobabili. In questa ipotesi il caso in cui x non compare nel vettore è assai raro: esso è supposto presentarsi al programma con frequenza $1/(n+1)$. La probabilità del caso in cui x compare *nel vettore è invece assai elevata: $1 - 1/(n+1) = n/(n+1)$: per n abbastanza grande si tratta di un valore prossimo a 1. Ricordiamo che questa è una ipotesi che stiamo facendo, che riguarda l'uso che del programma verrà fatto (e di conseguenza, fornisce un valore del tempo di calcolo medio valido solo in quell'ipotesi). In molti casi si tratterà di una ipotesi irrealistica.*

Si potrebbe per esempio argomentare che un caso assai più interessante è quello in cui i due casi "x appartiene a V" e "x non appartiene a V" sono tra loro equiprobabili; quandi poi x compare in V, sono a loro volta equiprobabili tutti i vari casi "x compare in $V[i]$" al variare di i tra 1 e n. Ricapitolando, l'istanza $n+1$ si presenta nel 50% dei casi; nel rimanente 50%, si presentano in modo equiprobabile le istanze tra 1 e n. Come calcolare il tempo medio in questo caso? La prossima sezione affronta il problema. □

2.2.1 Media pesata

Quando i dati non sono distribuiti in modo uniforme, ma, come alla fine dell'esempio precedente, la frequenza delle diverse istanze non è la stessa, occorre sostituire la media aritmetica della (2.1) con una relazione più generale. La *media pesata* è data da una sommatoria in cui "pesano di più" quegli addendi che sono supposti presentarsi più frequentemente di altri. Il tempo d'esecuzione relativo ad una particolare istanza viene pesato con la *frequenza* (o *probabilità*) con la quale supponiamo presentarsi quell'istanza. Supponendo di avere a che fare con k_n istanze di dimensione n, ognuna di esse verrà pesata con una probabilità p_i, per $1 \leq i \leq k_n$. La somma di tali probabilità, ovviamente, deve essere uguale ad uno (le k_n istanze esauriscono infatti tutte le possibilità che si possono presentare):

$$\sum_{i=1}^{k_n} p_i = 1. \tag{2.3}$$

Identificando, come prima, ogni istanza con il proprio indice i, $1 \leq i \leq k_n$, la media pesata del tempo d'esecuzione è allora definita come:

$$T_P^{med}(n) = \sum_{i=1}^{k_n} p_i T_P(n,i). \tag{2.4}$$

Esercizio 2.2.3 *La media aritmetica è un caso particolare della media pesata quando tutti i pesi sono uguali a $1/k_n$.*

Se riprendiamo il caso della ricerca lineare incerta che abbiamo appena discusso, sappiamo che vi sono $k_n = n + 1$ istanze dei dati di dimensione n. Avendo supposto che il caso peggiore si presenti nel 50% dei casi, mentre tutti gli altri casi sono tra loro equiprobabili, concludiamo che la probabilità (peso) del caso peggiore (l'istanza $n + 1$) è 0.5, mentre ogni singolo altro caso si presenta con probabilità $0.5/n$. I pesi così assegnati soddisfano alla relazione (2.3):

$$\sum_{i=1}^{n+1} p_i = \sum_{i=1}^{n} \frac{1}{2n} + \frac{1}{2} = \frac{1}{2} + \frac{1}{2} = 1.$$

Il tempo medio d'esecuzione (in passi di calcolo) si ottiene applicando la (2.4):

$$\begin{aligned}
T^{med}(n) &= \sum_{i=1}^{n+1} p_i \cdot T(n,i) \\
&= \sum_{i=1}^{n} \frac{1}{2n} T(n,i) + \frac{1}{2} T(n, n+1) \\
&= \frac{1}{2n} \sum_{i=1}^{n} (2i+1) + \frac{2n+2}{2} \\
&= \frac{1}{2n} (n + \frac{2n(n+1)}{2}) + \frac{2n+2}{2} \\
&= \frac{3}{2} n + 2.
\end{aligned}$$

Si osservi come con questa ipotesi di distribuzione dei pesi il tempo medio sia sensibilmente maggiore di quello che avevamo ottenuto in ipotesi di distribuzione uniforme.

Fissata la distribuzione dei dati (cioè i pesi p_i), la (2.4) può essere usata anche per determinare sperimentalmente l'effettivo tempo medio d'esecuzione: si procede alla misura dei tempi di tutte le istanze, e si calcola poi la media pesata secondo la distribuzione opportuna.

Esempio: 2.2.4 Valutazione riclin certa ed incerta
da fare □

2.3 Stima del tempo medio

Abbiamo visto nel paragrafo precedente che, per determinare il tempo medio e fissata una distribuzione dei dati, occorre procedere al calcolo (o alla misura) dei tempi necessari all'esecuzione di ogni singola istanza.

Se siamo interessanti ad un'analisi del comportamento asintotico del caso medio, dovremo manipolare algebricamente la somma ottenuta mediante la (2.4), alla ricerca, se possibile, di una sua formulazione "chiusa" (cioè che non contenga sommatorie) e che dia pertanto immediata informazione sull'ordine della funzione. Si tratta di uno studio assai importante e significativo, ma che talvolta, in specie in presenza di distribuzioni non uniformi, si presenta come molto difficile, quando non impossibile.[3]

Esempio: 2.3.1 Esempio semplice
da fare □

Se siamo interessati alla valutazione sperimentale, dovremo misurare il tempo necessario ad ogni singola istanza. Vi sono casi, tuttavia, in cui ciò non si può (o non è conveniente) fare. Prendiamo in considerazione il caso di un algoritmo di ordinamento. Abbiamo già osservato come una buona nozione di

[3] Ancora una volta, l'algoritmo Quicksort fornisce un bell'esempio di un caso in cui tale analisi è possibile e significativa.

dimensione dei dati sia in questo caso il numero, n, dei dati da ordinare. Il numero di possibili istanze di dimensione n è $k_n = n!$, una funzione che cresce molto velocemente al crescere di n. Cresce al tal punto che, per valori di n di poche decine, il tempo necessario alla misura sperimentale del tempo su tutte le possibili istanze diviene irrealisticamente lungo e, nei fatti, assolutamente fuori portata. Scartata dunque l'ipotesi di misurare tutte le istanze, se si vuole mantenere la scelta sperimentale e non analitica, non resta che ricorrere ad una *stima approssimata* del tempo medio d'esecuzione.

Possiamo ottenere una stima del tempo medio, selezionando un opportuno *campione* tra tutte le istanze dei dati, e calcolando poi la media su tale campione e non su tutto l'universo delle istanze. È chiaro che il valore così determinato costituisce un'approssimazione del valore reale, la cui bontà dipende, in sostanza, da due fattori:

- la buona distribuzione: il campione deve essere scelto in modo da rispecchiare la distribuzione dell'universo;
- la dimensione: pur essendo assai più piccolo dell'intero universo, il campione deve essere sufficientemente ampio da rappresentare adeguatamente l'universo stesso.

In una parola, il campione deve essere *rappresentativo* dell'universo da cui è stato estratto.

Ricapitolando, vogliamo stimare la media di un certo parametro X (nel nostro caso, il tempo di calcolo di un programma assegnato), al variare della dimensione dei dati, in cui l'universo dei dati di dimensione n è costituito da k_n istanze diverse. Sappiamo che la media è data da

$$E[X(n)] = \sum_{i=1}^{k_n} p_i X(n,i).$$

Scegliamo dunque un valore per n e generiamo un campione C, di cardinalità $c_n << k_n$, di istanze di dimensione n. Per ciascuna istanza i in C misuriamo il parametro $X(n,i)$ e calcoliamo la media di tali valori:

$$E'[X] = \frac{1}{c_n} \sum_{i=1}^{c_n} X(n,i)$$

Un risultato fondamentale della statistica matematica ci assicura che *se il campione C è distribuito nello stesso modo dell'universo da cui è estratto*, allora $E'[X]$ è una ragionevole stima della media $E[X(n)]$, la cui bontà dipende dalla cardinalità del campione stesso. In altri termini, al crescere di c_n, $E'[X]$ è garantito approssimare sempre meglio il valore della media (ma la "velocità" di tale approssimazione è proporzionale alla radice quadrata di c_n: quadruplicando c_n si raddoppia soltanto la bontà della stima).

Due problemi si pongono immediatamente: come determinare (generare) C con la stessa distribuzione dell'universo? E poi, una volta calcolato $E'[X]$ per un certo campione, è possibile avere qualche informazione su quanto tale valore si discosta dal valore reale della media $E[X(n)]$? Affronteremo queste questioni, nell'ordine, nelle due prossime sezioni.

2.4 Numeri pseudo-casuali

Le metodologie di selezione di campioni formano un capitolo assai importante della statistica. Il nostro caso, però, è particolarmente favorevole: abbiamo piena conoscenza di come è fatto l'universo delle istanze, e possiamo arbitrariamente scegliere in esso le istanze che più ci aggradano.

Per semplicità (e perché si tratta dell'unico caso che tratteremo in questo testo) supporremo che l'universo delle istanze sia distribuito in modo uniforme: le k_n istanze di dimensione n si presentano ciascuna con probabilità $1/k_n$. Per generare un campione distribuito in modo uniforme, possiamo allora utilizzare un generatore di numeri pseudo-casuali. Si tratta di un programma che, a partire da un dato *seme* fornito come dato, genera una sequenza (determinata dal seme) di numeri razionali compresi tra 0 ed 1. La proprietà cruciale di tale sequenza è che essa "somiglia" in tutto ad una successione di numeri

```
var seed: double

function Random: real;
  const
    a = 16807;
    m = 2147483647;
    q = 127773;
    r = 2836;
  var
    lo, hi, test: double;
begin
  hi := trunc(seed / q);
  lo := seed - q * hi;
  test := a * lo - r * hi;
  if test < 0.0 then
    seed := test + m
  else
    seed := test;
  Random := seed / m
end;
```

Figure 2.2: Un generatore di numeri pseudo-casuali **inserire biblio e commenti**

estratti mediante un processo davvero casuale (per esempio effettuando estrazioni con rimpiazzamento da un'urna). Qui "somiglianza" significa che eventuali test statistici effettuati sulla sequenza non sono in grado di rilevare significative regolarità nei suoi elementi, così da indicare la provenienza della successione da un algoritmo deterministico (come in effetti è un generatore di numeri pseudo-casuali). La sequenza prodotta da un generatore di numeri pseudo-casuali è in genere ciclica (cioè ad un certo punto torna a ripetersi dall'inizio), ma il *periodo* di tale ciclo è sufficientemente lungo da poter essere trascurato per gli scopi che ci interessano.

Costruire un buon generatore di numeri pseudo-casuali non è cosa semplice, né funzionano artifici tanto ingenui quanto diffusi come quello di usare l'orologio di macchina. Ciò che è ancora più sorprendente è che la stragrande maggioranza dei cosiddetti "generatori di numeri pseudo-casuali" messi a disposione come funzioni predefinite (col nome `random`, o simile) nei linguaggi di programmazione non sono... buoni generatori! Semplici test statistici rilevano la non uniforme distribuzione delle sequenze da essi generati. D'altra parte sono noti in letteratura generatori semplici, con buone proprietà statistiche, che possono essere programmati facilmente in un qualsiasi linguaggio di programmazione. Uno di questi, quello che useremo nel seguito del testo, è quello della Figura 2.2.

La funzione `random` è definita in un ambiente globale in cui è dichiarata la variabile reale (in doppia precisione) `seed`, il seme del generatore. Dopo aver inizializzato il seme, chiamate successive a `random` generano una successione di numeri pseudo-casuali compresi tra 0 e 1. Le proprietà di buona distribuzione della successione *non* dipendono dal seme scelto, ma solo dalle costanti numeriche definite nella funzione. Non è una buona idea cambiare tali valori.

Possiamo usare il generatore appena introdotto per generare istanze casuali distribuite in modo uniforme per molti problemi.

Esempio: 2.4.1 Ric Lin Certa per n = 30000
Piu' confronto con il risultato "esatto". □

2.5 Intervalli di confidenza

Abbiamo visto come, con l'ausilio di un processo di generazione pseudo-casuale, possiamo determinare un certo valore $E'[X(n)]$, che approssima il valore reale $E[X(n)]$. Sappiamo anche che, al crescere della dimensione del campione, $E'[X(n)]$ approssima sempre meglio $E[X(n)]$.

Vogliamo ora cercare di quantificare questa approssimazione, in termini probabilistici. È evidente che, ripetendo con altri valori l'esperimento che ci ha fornito la stima $\overline{X}_1 = E'[X(n)]$, otterremo un diverso valore della media campionaria, diciamo $\overline{X}_2$. Ripetendo l'esperimento un gran numero di volte, otterremo sempre valori diversi (una successione $\overline{X}_i$) che tuttavia avranno grande probabilità di non discostarsi troppo (se i campioni sono sufficientemente grandi) dal valore reale della media $E[X(n)]$. Si può sempre presentare il caso in cui un campione generato casualmente è particolarmente "cattivo", dando luogo ad un valore della media campionaria molto distante dalla media reale, ma, se i campioni sono generati secondo la stessa distribuzione dell'universo, tale evento sarà molto raro. Vi sono dunque in gioco due parametri:

1. un valore Δ (per ora ignoto) per il quale gran parte degli $\overline{X}_i$ distano da $E[X(n)]$ meno di Δ;

2. la percentuale α di esperimenti "cattivi", cioè che hanno probabilità di non rientrare nella distanza Δ da $E[X(n)]$.

La statistica matematica fornisce un metodo per collegare tra loro (e calcolare) tali parametri. Fissato un valore per α (usualmente piccolo, con 0.05, 0.1, 0.01 e 0.001 in ordine decrescente di popolarità) daremo un modo per calcolare, fissato un campione sufficientemente grande, la media campionaria $E'[X(n)]$ ed un certo valore per Δ. L'intervallo così determinato

$$[E'[X(n)] - \Delta, E'[X(n)] + \Delta]$$

si dice *intervallo di confidenza all'*$1-\alpha$ ed il suo significato è il seguente: se ripetessimo l'esperimento un gran numero di volte, e per ognuna di esse calcolassimo un intervallo di confidenza, approssimativamente solo $\alpha \cdot 100$ intervalli su cento *non* conterrebbero il vero valore della media. Pur con qualche imprecisione, possiamo dunque dire che un intervallo di confidenza ci fornisce non solo una stima della media (data da $E'[X(n)]$), ma anche di quanto tale stima può allontanarsi dalla media reale $E[X(n)]$.

Come calcolare dunque il parametro Δ? Ci accontenteremo qui di riportare una formula, rimandando la sua giustificazione al corso di statistica. Il calcolo fa uso di un ulteriore parametro del campione, la *varianza campionaria*, definita come

$$s(n)^2 = \frac{1}{c_n}\sum_{i=1}^{c_n}(X(n,i) - E'[X(n)])^2, \tag{2.5}$$

dove, si ricordi, c_n è la cardinalità del campione (costituito a sua volta da istanze del problema in cui i dati hanno dimensione n). Fissato il *coefficiente di confidenza* $1-\alpha$ (per esempio, il 95%, o il 90%, o il 99%), se il campione è abbastanza grande (diciamo di dimensione circa 100), si ha

$$\Delta = \frac{1}{\sqrt{c_n}} \cdot z(1-\frac{\alpha}{2}) \cdot s(n). \tag{2.6}$$

Il nuovo parametro $z(1-\frac{\alpha}{2})$ che compare nella 2.6 si riferisce ad una funzione molto importante nel calcolo delle probabilità che è nota come funzione di distribuzione *normale*. Alcuni suoi valori, al variare di α, sono riportati nella tabella della Figura 2.3 e sono comunque disponibili in molti testi di statistica.

Concludiamo questo paragrafo con un'osservazione di programmazione. Il calcolo della varianza campionaria usando la definizione (2.5) richiede la memorizzazione di tutti i c_n valori $X(n,i)$ e, dunque, una discreta quantità di memoria. Con facili calcoli, sviluppando il quadrato ed usando la definizione della media $E'[X(n)]$, si osserva che

$$s(n)^2 = \frac{1}{c_n}\sum_{i=1}^{c_n} X(n,i)^2 - E'[X(n)]^2.$$

α	$z(1-\frac{\alpha}{2})$
0.05	1.96
0.10	1.64

Figure 2.3: Valori di $z(1-\alpha/2)$

Con questa nuova espressione, non è più necessario mantenere tutti i distinti $X(n,i)$; la memorizzazione delle due somme parziali $\sum_{i=1}^{k} X(n,i)$ e $\sum_{i=1}^{k} X(n,i)^2$ (con k che partendo da 1 viene via via incrementato fino a c_n) è sufficiente per il calcolo sia della media $E'[X(n)]$, sia della varianza.

Chapter 3

Valutazione di algoritmi di ordinamento

3.1 Introduzione

L'obiettivo di questo capitolo é quello di misurare i tempi di esecuzione di alcuni algoritmi, per verificarne sperimentalmente le curve teoriche di complessitá e fornire un'esempio completo di come misurare i tempi di esecuzione di un'algoritmo .

Si analizzeranno in dettaglio i seguenti algoritmi di ordinamento: Insertionsort, Mergesort e Quicksort e dall'analisi dei tempi ottenuti, si deriverá un nuovo algoritmo di ordinamento piú veloce dei tre proposti.

I tre algoritmi e l'intera prova saranno implementati in due linguaggi di programmazione: Pascal e Java.

3.1.1 Algoritmi di ordinamento e complessitá computazionali

La seguente tabella riassume quali sono le complessitá computazionali degli algoritmi in esame, nel caso in cui i vettori da ordinare siano inizializzati in maniera casuale e nel caso peggiore.

Algoritmo	Worst-Case	Average-Case
InsertionSort	n^2	n
QuickSort	n^2	n log n
MergeSort	n log n	n log n

3.1.2 Svolgimento della prova

Nella prova si misureranno i tempi di ordinamento dei tre algoritmi sia nel caso medio che quello peggiore. Fa eccezione MergeSort; esso non risente della posizione dei numeri nel vettore. La sua complessitá computazionale é sempre n log n.

Si prenderanno i tempi, con precisione pari a 0.03, su vettori di lunghezza n pari a: 10, 20, 30, 40 ,50, 60, 70, 80, 90, 100, 150, 200, 250, 300, 350, 400, 450, 500, 750, 1000. Le misure si infittiscono per vettori piccoli per studiare meglio il loro andamento. Infatti per n grande l'andameno asintotico é garantito dalla teoria: $O(n) = \{f(n) : \exists c \geq 0, n_0 \; tale \; che \; \forall n \geq n_0 \; 0 \leq f(n) \leq cg(n)\}$, ma per n piccolo non abbiamo un supporto teorico, l'andamento infatti potrebbe essere:

Gli ambienti di sviluppo adottati nella prova sono:
Delphi (Borland) per il pascal;
JBuilder per java.

Il computer sul quale é stata eseguita la prove é un PC Pentium PRO 133Mhz con 64Mb di RAM del laboratorio del triennio.

3.1.3 Struttura del programma

Il programma é strutturato in due parti: calcolo dei tempi nei casi medi e nei casi peggiori. Nella prima parte, viene calcolato il fattore di ripetizione, usato per ridurre l'errore relativo. Quindi si stimano il tempo medio, la varianza e l'intervallo di confidenza di Insertionsort, Mergesort, Quicksort al variare della lunghezza del vettore. Inoltre viene sperimentato un nuovo algoritmo di ordinamento, Quinssort, basato su Insertionsort, Quicksort e un parametro blocco, che stabilisce come i due algoritmi debbano lavorare assieme. I dati si determineranno su un vettore lungo 200 al variare di tale parametro. Una volta determinati i vettori che determinano la complessitá peggiore per Insertionsort e Quicksort (Mergosort ha sempre complessitá N log N), si calcola il tempo medio di esecuzione.

3.1.4 Calcolo dei tempi di esecuzione

Per calcolare il tempo di esecuzione di ogni algoritmo si utilizzeranno delle funzioni di libreria che restituiscono il tempo come differenza tra una certa data di riferimento e l'istante della chiamata. Per entrambi i linguaggi usati tale differenza ha una precisione pari a 0.001 secondi.

In Pascal la funzione é Now che restituisce un oggetto di tipo TdateTime. Tramite la funzione DecodeTime si ottiene il tempo in ore, minuti, secondi e millisecondi.

```
function GetTime: double;
var
        Present: TDateTime;
        Hour, Min, Sec, MSec: Word;
    begin
        Present:= Now;
        DecodeTime(Present, Hour, Min, Sec, MSec);
        GetTime:=Hour*60*60*1000+Min*60*1000+Sec*1000+MSec
    end;
```

Figure 3.1: Codice per la funzione GetTime implementata in pascal

In Java é disponibile la classe Date con il metodo getTime() nel package java.util.Date. Tale funzione ritorna il tempo calcolato all'istante in cui l'oggetto Date viene istanziato. Perció si rende necessario creare un nuovo oggetto Date ogni volta che si invoca il metodo getTime(). Ogni volta che si deve

```
private static double GetTime()
{
Date tempo;
tempo = new Date();
return (double)tempo.getTime();
}
```

Figure 3.2: Codice per la funzione GetTime implementata in java

misurare un intervallo di tempo, si deve racchiudere il codice tra due chiamate della funzione GetTime per poterne calcolare la differenza.

L'errore che si commetterebbe eseguendo una sola volta gli algoritmi in esame sarebbe troppo elevato, in alcuni casi superiore al 100%, dato che l'evento sarebbe troppo corto per essere rilevato. Per limitare l'errore relativo al 3%, si deve misurare un evento della durata almeno di δ, con δ tale che:

$$100 : 3 = \delta : 1$$

L'esecuzione della procedura piú breve deve essere quindi ripetuta un fattore r di volte. Dunque si dovrá fare una chiamata a GetTime prima del ciclo for di ripetizioni e una dopo. É importante sottolineare che le due chiamate non vanno eseguite all'interno del ciclo for, altrimenti l'effetto sarebbe quello di misurare ripetutamente eventi troppo corti.

La procedura piú corta che misuriamo quella di inizializzazione del vettore. Nel calcolo dei tempi medi tale procedura é la stessa per Insertionsort, Mergesort, Quicskort e Quinssort e quindi il fattore di ripetione viene calcolato una volta sola sulla procedura che inizializza i vettori in modo casuale. Nel calcolo dei tempi per l'ordinamento nel caso peggiore questo fattore deve essere ricalcolato perché la procedura che inizializza il vettore peggiore che nel caso di Insertionsort é il vettore in ordine inverso e nel caso di Quicksort é sia il vettore ordinato in modo crescente che quello in modo decrescente.

Il codice che calcola r per ogni lunghezza n del vettore e lo memorizza nel vettore VR, il seguente:

```
for j := 1 to 20 do
begin
T1 := 0;
T2 := 0;
r := 1;
while ((T2 - T1) < Delta) do
begin
r := r * 2;
T1 := Time;
for i := 1 to r do
InitAverage(v,1, L[j]);
T2 := Time
end;
a := Trunc(r / 2);
b := r;

while ((b - a) >= 5) do
begin
r := Trunc((a + b) / 2);
T1 := Time;
for i := 1 to r do
InitAverage(v,1, L[j]);
T2 := Time;
if ((T2 - T1) <= Delta) then
a := r
else
b := r
end;
VR[j] := b
end;
```

Figure 3.3: Codice per il fattore di ripetizione implementato in pascal

Il primo ciclo while serve a determinare una stima per difetto ed una per eccesso del fattore di ripetizione: r viene raddoppiato ad ogni ciclo finché l'intervallo di tempo misurato supera δ. Alla fine del ciclo tale fattore sará compreso tra a e b, estremi dell'intervallo.

Il secondo ciclo while riduce questo intervallo attraverso un processo di bisezione. Alla fine del ciclo, b sará una stima per eccesso del fattore di ripetizione con un errore di 5.

```
for (j=0;j<20;j++)
{
A = new InsertionSort(L[j]);
T1 = T2 = 0;
r = 1;
while((T2-T1)<Delta)
{
r *= 2;
T1 = GetTime();
for (i=1;i<r;i++)
seed = A.InitAverage(1,100000,seed);
T2 = GetTime();
}
int a = (int)(r/2);
int b = r;
while((b-a)>=5)
{
r = (int)((a+b)/2);
T1 = GetTime();
for (i=1;i<r;i++)
seed = A.InitAverage(1,100000,seed);
T2 = GetTime();
if ((T2-T1)<=Delta)
a = r;
else
b = r;
}
R[j] = b;
}
```

Figure 3.4: Codice per il fattore di ripetizione implementato in java

Il vettore dei fattori di ripetizione VR sará usato come fattore di ripetizione per tutti e tre gli algoritmi, in quanto la procedura piú corta che misuriamo, InitAverage, é la stessa.

3.1.5 Stima dei tempi medi, varianza e intervallo di confidenza

Per il calcolo dei tempi medi la prova dovrebbe generare n! vettori con n lunghezza del vettore e fare la media aritmetica dei tempi ottenuti. Data l'eccessiva dimensione dell'universo ció non é possibile. La prova pertanto si deve limitare ad un campione di vettori.

I risultati sperimentali ottenuti, pertanto, definiranno una variabile casuale normale con una certa media e una certa varianza. Con questi dati si puó ricavare facilmente l'intervallo di confidenza della media.

Per garantire che tale intervallo sia inferiore a tot il numero di campioni deve essere di almeno 500 elementi.

Il codice che calcola la media, la varianza, e l'intervallo di confidenza é il seguente: La struttura del codice é composta da tre cicli for innestati: quello piú esterno serve per misurare i tempi al variare della lunghezza del vettore;
il secondo ciclo calcola la media e la varianza sul campione di 500 vettori diversi inizializzati casualmente;
il ciclo piú interno ripete l'ordimento dello stesso vettore per il fattore di ripetizione r calcolato in precedenza per limitare l'errore relativo al 3% sui dati ottenuti.

```
OldSeed := 1;
for j := 1 to 20 do
begin
s1 := 0;
s2 := 0;
for k := 1 to Ncampioni do
begin
T1 := Time;
for i := 1 to VR[j] do
begin
                                        seed := OldSeed;
                                        InitAverage(v,1, L[j]);
                                        InsertionSort(v, 1, L[j])
end;
T2 := Time;
Tlordo := T2 - T1;

T1 := Time;
for i := 1 to VR[j] do
begin
                                        seed := OldSeed;
                                        InitAverage(v,1, L[j])
end;
T2 := Time;
Ttara := T2 - T1;

                              OldSeed:=seed;

Tnetto := (Tlordo - Ttara) / (VR[j]*1000);
s1 := s1 + sqr(Tnetto);
s2 := s2 + Tnetto
end;

Tmedio := s2 / Ncampioni;
Varianza := (s1 - Ncampioni * Tmedio * Tmedio) / (Ncampioni - 1);
writeln('Lunghezza =  ', L[j], '    Ripetizioni = ', VR[j]);
writeln('Media = ', Tmedio , '   Varianza = ', Varianza );
writeln('Media = ', Tmedio , ' + / -  ', 1.96 * sqrt(Varianza / Ncampioni) );
writeln;
writeln
end;
```

Figure 3.5: Codice per calcolare la media, la varianza, e l'intervallo di confidenza implementato in pascal

Per ottenere lo stesso vettore inizializzato casualmente per r volte, si deve tener traccia del seme della funzione Random. Quindi inizializzare piú volte il vettore a partire dallo stesso seme.

3.1.6 Linguaggio di programmazione

Gli algoritmi studiati sono stati implementati con due linguaggi di programmazione Pascal e Java. I due si differenziano per alcuni aspetti. Il primo un linguaggio compilativo (esistono anche delle versioni

```
NewSeed = 1;
for(j=0;j<20;j++)
{
A = new InsertionSort(L[j]);
s1 = s2 = 0;
for(k=0;k<Ncampioni;k++)
{
seed = NewSeed;
T1 = GetTime();
for(i=0;i<R[j];i++)
{
A.InitAverage(1,100000,seed);
A.Sort(0,A.GetLength()-1);
}
T2 = GetTime();
Tlordo = T2 - T1;

T1 = GetTime();
for(i=0;i<R[j];i++)
NewSeed = A.InitAverage(1,100000,seed);
T2 = GetTime();
Ttara = T2 - T1;

Tnetto = ((Tlordo-Ttara)/R[j])/1000;

s1 = s1 + Tnetto*Tnetto;
s2 = s2 + Tnetto;
}

Tmedio = s2/Ncampioni;
Varianza = (s1-Ncampioni*Tmedio*Tmedio)/(Ncampioni-1);

System.out.println("Lunghezza = "+L[j]+" Ripetizioni = "+R[j]);
System.out.println("Media = "+Tmedio+" Varianza = "+Varianza);
System.out.println("Media = "+Tmedio+" +/- "+1.96*Math.sqrt(Varianza/Ncampioni));
System.out.println();
}
```

Figure 3.6: Codice per calcolare la media, la varianza, e l'intervallo di confidenza implementato in java

interpretative), mentre il secondo é interpretativo. Inoltre il pascal adotta una filosofia function-oriented, mentre java é orientato agli oggetti, non gestisce i puntatori e sfrutta l'allocazione dinamica in modo pesante. Per questi motivi i tempi di esecuzione del programma in Java saranno piú elevati di quelli in Pascal, ma quello che si osserver che le curve che otterremo avranno lo stesso andamento asintotico. Quello che cambia sono le costanti moltiplicative.

3.2 Insertion

3.2.1 Filosofia e complessitá

Per ordinare il vettore [1..n], l'algoritmo considera il sottovettore [1..i-1] ordinato, con che parte da 2. Quindi considera l'elemento i-esimo e lo confronta con gli elementi giá ordinati per poterlo inserire al posto giusto fra di essi. A questo punto il sottovettore [1..i] sará ordinato e si potrá procedere in questo modo fino ad n.

É un algoritmo in place di complessitá O(n^2) (caso peggiore: vettore ordinato in senso inverso).

3.2.2 Codice

Pascal

```
procedure InsertionSort (var v: Vettore; p, q: integer);
var
j, i: integer;
key: integer;
begin
for j := p + 1 to q do
begin
key := v[j];
i := j - 1;
while (i >= p ) and (v[i] > key) do
begin
v[i + 1] := v[i];
i := i - 1
end;
v[i + 1] := key
end
end;
```

Figure 3.7: Codice per Insertionsort implementato in pascal

3.2.3 Tempo di esecuzione nei casi medio e peggiore

3.2.4 Confronto dei casi (curva teorica)

3.3 Merge

3.3.1 Filosofia e complessitá

L'algoritmo é diviso in tre fasi: nella prima il vettore [p..q] viene diviso in due sottovettori [p..floor(q-p/2)] e [floor(q-p/2)..q], che vengono quindi ordinati ricorsivamente. Alla fine si richima la procedura Merge che fonde i due sottovettori ordinati. Tale procedura non lavora in place.

Ha complessitá O(n log n) sia nel caso migliore che nel caso peggiore.

3.3.2 Codice

Java

```
class InsertionSort extends Vettore implements VettoreOrdinabile
{
    InsertionSort(int dimensione)
    {
super(dimensione);
    }
    public double InitAverage(int inf,int sup,double seed)
    {
int i;
Random num;
num = new Random(seed);
for(i=0;i<GetLength();i++)
   vett[i] = (int) num.get(inf,sup);
return num.GetSeed();
    }
    public void InitWorst()
    {
int i;
for(i=0;i<GetLength();i++)
   vett[i] = vett.length-i;
    }
    public void Sort(int p,int q)
    {
int j,i;
int key;
for(j=p+1;j<=q;j++)
{
    key = vett[j];
    for(i=j-1;(i>=p)&&(vett[i]>key);i--)
vett[i+1] = vett[i];
    vett[i+1] = key;
}
    }
}
```

Figure 3.8: Codice per Insertionsort implementato in java

Dimensione	*Tempi*
10	$1.57\,10^{-6}$
20	$4.37\,10^{-6}$
30	$9.01\,10^{-6}$
40	0.000014
50	0.00002
60	0.0000268
70	0.0000356
80	0.000046
90	0.0000557
100	0.0000683
150	0.000141
200	0.000249
250	0.00038
300	0.000542
350	0.000731
400	0.000946
450	0.00119
500	0.00147
750	0.00325
1000	0.00575

Figure 3.9: Caso medio Insertionsort in pascal

Dimensione	*Tempi*
10	$4.85\,10^{-6}$
20	0.0000159
30	0.0000318
40	0.0000544
50	0.0000813
60	0.000117
70	0.000151
80	0.000199
90	0.000246
100	0.000309
150	0.000678
200	0.00116911
250	0.00182
300	0.00260738
350	0.00354
400	0.00461215
450	0.00585
500	0.00722143
750	0.0162
1000	0.0287108

Figure 3.10: Caso medio Insertionsort in java

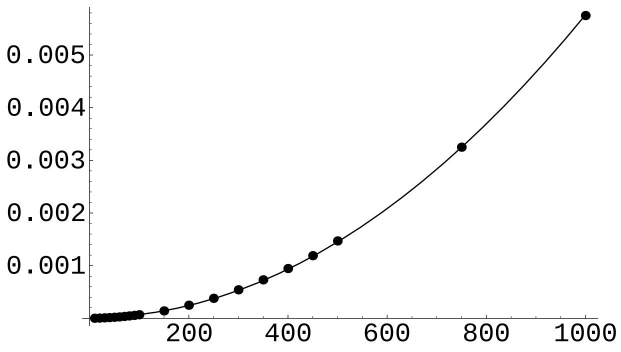

Figure 3.11: Caso medio Insertionsort in pascal

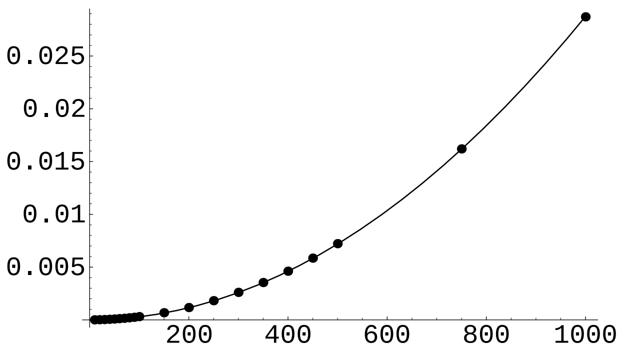

Figure 3.12: Caso medio Insertionsort in java

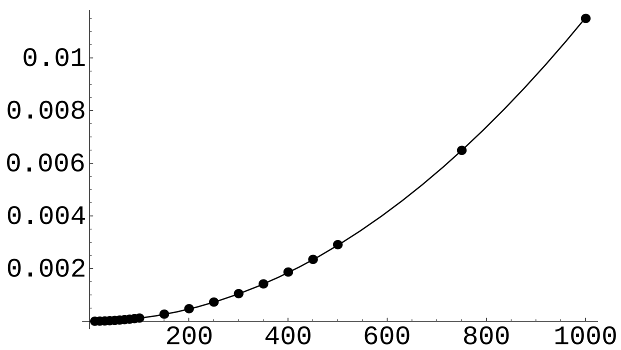

Figure 3.13: Caso peggiore Insertionsort in pascal

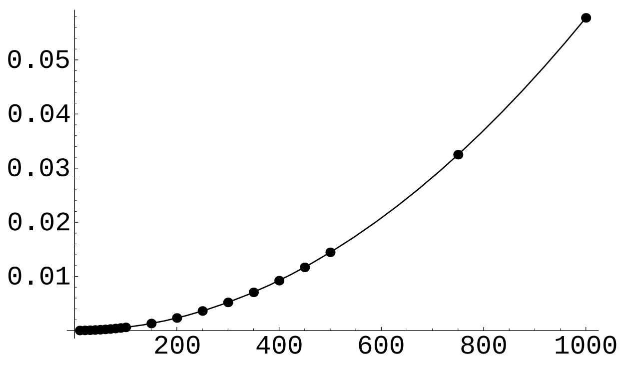

Figure 3.14: Caso peggiore Insertionsort in java

3.3.3 Tempo di esecuzione nei casi medio e peggiore

3.3.4 Confronto dei casi (curva teorica)

3.4 Quick

3.4.1 Filosofia e complessitá

L'algoritmo divide il vettore [p..q] da ordinare in due sottovettori [p..r] e [r+1..q] in modo che gli elementi del primo vettore siano minori o uguali ad ogni elemento del secondo vettore. Questa divisione é ottenuta attraverso la funzione Partition che ritorna r, l'indice in base al quale il vettore da ordinare viene diviso. Quindi i due sottovettori sono ordinati ricorsivamente e dato che sono ordinati in place, il vettore ottenuto é giá ordinato.

La complessitá é O(n log n) nel caso medio e O(n^2) nel caso peggiore (vettore in ordine crescente o decrescente).

3.4.2 Codice

3.4.3 Tempo di esecuzione nei casi medio e peggiore

3.4.4 Confronto dei casi (curva teorica)

3.5 Confronto dei tre algoritmi

Dall'analisi dei tempi ottenuti, si verifica che le curve sperimentali hanno lo stesso andamento teorico. Inoltre le curve ottenute fanno vedere che anche per n molto piccoli, dove non c'é una controparte teorica, le curve rispettano lo stesso andamento. Insertion dunque l'algoritmo piú lento sia nel caso peggiore che nel caso medio. Si osservi solo che per n minore di xxx é piú veloce di Quicksort e per n minore di yyy é piú veloce di Mergesort.

Sebbene MergeSort e QuickSort abbiano la stessa complessitá teorica (n log n), con questa prova si puó notare la differenza di complessitá, misurando le costanti moltiplicative. Risulta infatti che MergeSort ha

Pascal

```
procedure MergeSort (var v: Vettore; p, q: integer);
var
r: integer;
procedure Merge (var v: Vettore; p, r, q: integer);
var
k, i, j: integer;
aux: Vettore;
begin
k := 1;
i := p;
j := r;
while (i < r) and (j <= q) do
begin
if (v[i] <= v[j]) then
begin
aux[k] := v[i];
i := i + 1
end
else
begin
aux[k] := v[j];
j := j + 1
end;
k := k + 1
end;
while (i < r) do
begin
aux[k] := v[i];
k := k + 1;
i := i + 1
end;
while (j <= q) do
begin
aux[k] := v[j];
k := k + 1;
j := j + 1
end;
k := 1;
for i := p to q do
begin
v[i] := aux[k];
k := k + 1
end
end;

begin
if (p < q) then
begin
r := Trunc((p + q) / 2);
MergeSort(v, p, r);
MergeSort(v, r + 1, q);
Merge(v, p, r + 1, q)
end;
end;
```

Figure 3.15: Codice per Mergesort implementato in pascal

Java

```
class MergeSort extends Vettore implements VettoreOrdinabile
{
    MergeSort(int dimensione)
    {
super(dimensione);
    }
    public double InitAverage(int inf,int sup,double seed)
    {
int i;
Random num;
num = new Random(seed);
for(i=0;i<GetLength();i++)
   vett[i] = (int) num.get(inf,sup);
return num.GetSeed();
    }
    public void InitWorst()
    {
int i;
for(i=0;i<GetLength();i++)
   vett[i] = i;
    }
    public void Sort(int p,int q)
    {
int r;
if(p<q)
{
    r = (int)Math.floor((p+q)/2);
    Sort(p,r);
    Sort(r+1,q);
    Merge(p,r,q);
}
    }
    private void Merge(int p,int r,int q)
    {
int k,i,j;
int aux[];
aux = new int[q-p+1];
for(k=0,i=p,j=r+1;(i<=r)&&(j<=q);k++)
    if(vett[i]<=vett[j])
aux[k] = vett[i++];
    else
aux[k] = vett[j++];
while(i<=r)
    aux[k++] = vett[i++];
while(j<=q)
    aux[k++] = vett[j++];
for(k=0,i=p;i<=q;i++,k++)
 vett[i] = aux[k];
    }

}
```

Figure 3.16: Codice per Mergesort implementato in java

Dimensione	*Tempi*
10	9.10^{-6}
20	0.0000215
30	0.0000366
40	0.0000488
50	0.0000646
60	0.0000792
70	0.0000936
80	0.000106
90	0.000126
100	0.000141
150	0.000215
200	0.000301
250	0.000383
300	0.000461
350	0.000551
400	0.000641
450	0.00073
500	0.000814
750	0.00126
1000	0.00172

Figure 3.17: Caso medio Mergesort in pascal

Dimensione	*Tempi*
10	0.0000473
20	0.000108
30	0.000175
40	0.000242
50	0.000313
60	0.000427
70	0.000475
80	0.000562
90	0.000644
100	0.000727
150	0.00108418
200	0.0015315
250	0.00189335
300	0.00218058
350	0.002602
400	0.00293645
450	0.00343619
500	0.00383
750	0.0058633
1000	0.00804488

Figure 3.18: Caso medio Mergesort in java

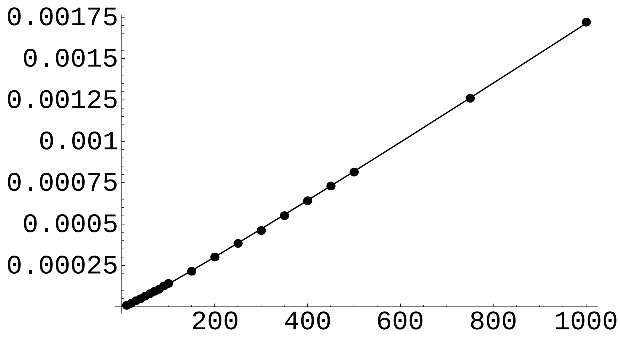

Figure 3.19: Caso medio Mergesort in pascal

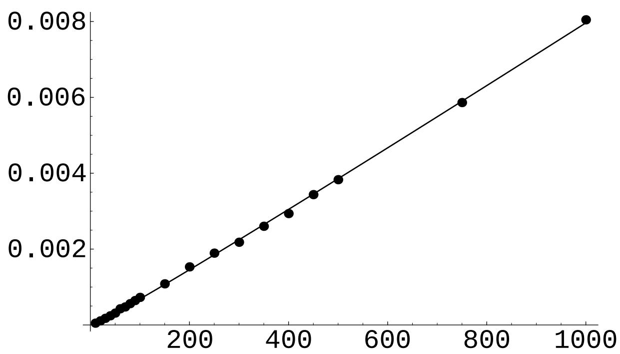

Figure 3.20: Caso medio Mergesort in java

Pascal

```
function Partition (var v: Vettore; p, q: integer): integer;
var
x, i, j: integer;
fine: boolean;
procedure Swap (var v: Vettore; i, j: integer);
var
aux: integer;
begin
aux := v[i];
v[i] := v[j];
v[j] := aux
end;

begin
x := v[p];
i := p - 1;
j := q + 1;
fine := true;
while (fine) do
begin
repeat
j := j - 1
until (v[j] <= x);
repeat
i := i + 1
until (v[i] >= x);
if (i < j) then
Swap(v, i, j)
else
fine := false;
end;
Partition := j
end;

procedure QuickSort (var v: Vettore; p, q: integer);
var
r: integer;
begin
if (p < q) then
begin
r := Partition(v, p, q);
QuickSort(v, p, r);
QuickSort(v, r + 1, q)
end
end;
```

Figure 3.21: Codice per Quicksort implementato in pascal

Java

```
class QuickSort extends Vettore implements VettoreOrdinabile
{
    QuickSort(int dimensione)
    {
super(dimensione);
    }
    public double InitAverage(int inf,int sup,double seed)
    {
int i;
Random num;
num = new Random(seed);
for(i=0;i<GetLength();i++)
   vett[i] = (int) num.get(inf,sup);
return num.GetSeed();
    }
    public void InitWorst()
    {
int i;
for(i=0;i<GetLength();i++)
   vett[i] = i;
    }
    public void Sort(int p,int q)
    {
if(p<q)
{
    int r;
    r = Partition(p,q);
    Sort(p,r);
    Sort(r+1,q);
}
    }
    private int Partition(int p,int q)
    {
int x,i,j,r;
boolean fine;
x = vett[p];
i = p-1;
j = q+1;
fine = true;
while(fine)
{
    while(vett[--j]>x);
    while(vett[++i]<x);
    if(i<j)
swap(i,j);
    else
fine = false;
}
return j;
    }
    private void swap(int i,int j)
    {
int aux;
aux = vett[i];
vett[i] = vett[j];
vett[j] = aux;
    }
}
```

Figure 3.22: Codice per Quicksort implementato in java

Dimensione	*Tempi*
10	$5.96\,10^{-6}$
20	0.0000143
30	0.0000243
40	0.0000335
50	0.0000437
60	0.0000539
70	0.0000638
80	0.0000724
90	0.0000839
100	0.0000959
150	0.000147
200	0.000202
250	0.000258
300	0.000317
350	0.000371
400	0.000434
450	0.000487
500	0.000544
750	0.000859
1000	0.00116

Figure 3.23: Caso medio Quicksort in pascal

Dimensione	*Tempi*
10	0.0000103
20	0.0000257
30	0.0000434
40	0.0000609
50	0.0000785
60	0.000094
70	0.000113
80	0.000135
90	0.000148
100	0.00017
150	0.000271
200	0.000377
250	0.000482
300	0.000589
350	0.000691
400	0.000798
450	0.000905
500	0.00101814
750	0.00159
1000	0.00220662

Figure 3.24: Caso medio Quicksort in java

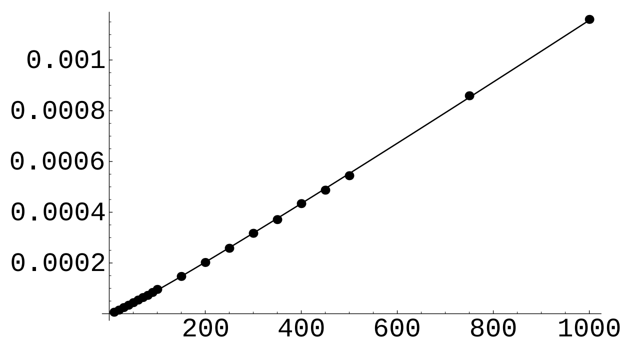

Figure 3.25: Caso medio Quicksort in pascal

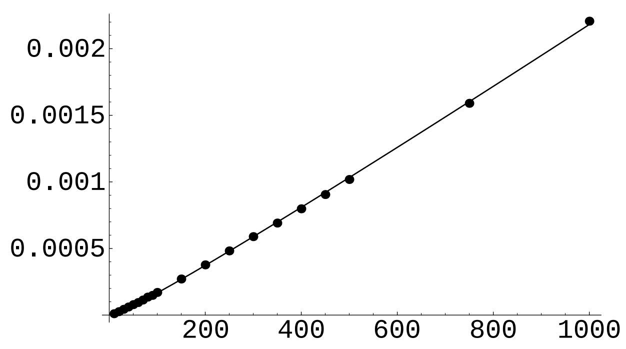

Figure 3.26: Caso medio Quicksort in java

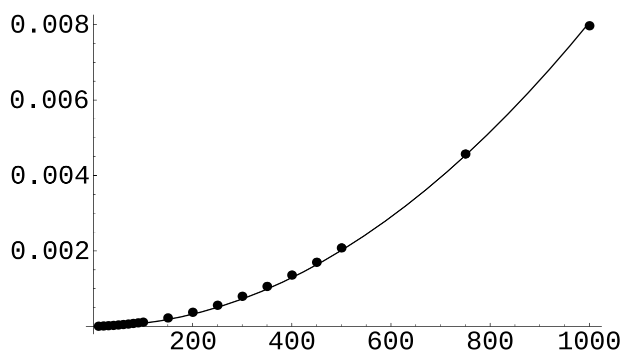

Figure 3.27: Caso peggiore Quicksort in pascal

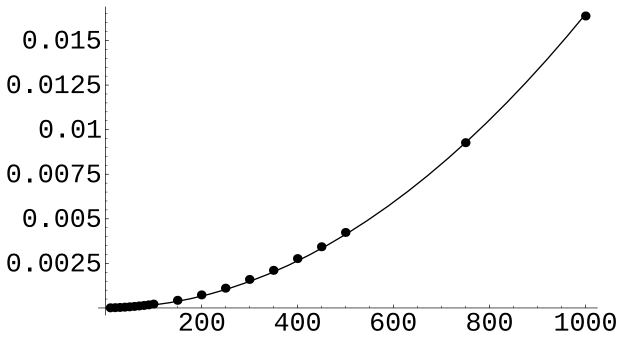

Figure 3.28: Caso peggiore Quicksort in java

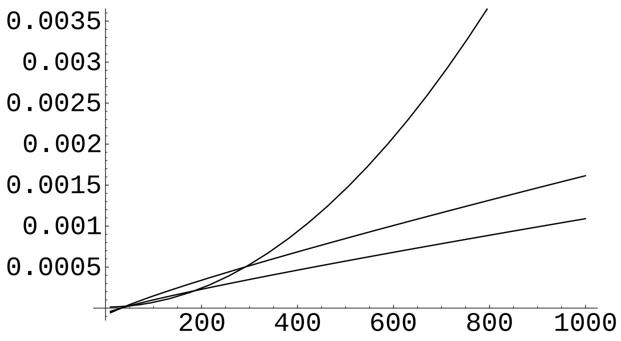

Figure 3.29: Confronti nel caso medio in pascal

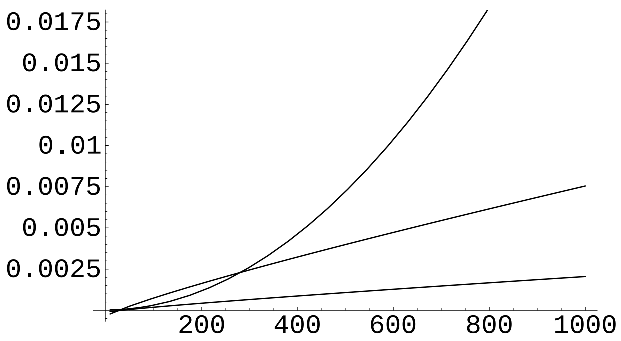

Figure 3.30: Confronti nel caso medio in java

costanti moltiplicative maggiori rispetto a QuickSort. Nella pratica si preferisce infatti usare il secondo, nonostante MergeSort sia sempre N log n anche nel caso peggiore.

InsertionSort non comunque un algoritmo da scartare a causa della sua elevata complessitá. Molto spesso il vettore da ordinare non un vettore completamente disordinato (), ma é giá stato ordinato in precedenza. A questo punto ci sono state delle operazioni nel vettore che hanno alterato parzialmente l'ordinamento. In tali situazioni questo algoritmo risulta vincente rispetto agli altri. Infatti si puó notare che la complessitá di Insertionsort dipende essenzialmente dalla distanza degli elementi nel vettore di partenza dalla loro posizione nel vettore ordinato e non dalla percentuale di elementi non disordinati.

La prova seguente misura i tempi necessari per ordinare un vettore lungo ??? con il 100% dei numeri fuori posto ed un vettore con solo il 10% dei numeri fuori posto.

Spiegazione e commento .

3.6 Un algoritmo di ordinamento piú veloce

3.6.1 Introduzione: il meglio di insertion e di quick

Si quindi osservato che InsertionSort é un algoritmo veloce per vettori disordinati a blocchi. In questo caso lineare. Inoltre QuickSort é l'algoritmo piú veloce con vettori disordinati. Cercando di prendere il meglio da entrambi gli algoritmi, se ne puó costruire un terzo con prestazioni ottime. L'idea quella di usare QuickSort per ordinare il vettore disordinato fino ad ottenere un vettore ordinato a blocchi () Poi con un'unica passata di InsertionSort si ordina il vettore cosí ottenuto.

Il problema é quello di determinare qual é la dimensione k del blocco che la nuova versione di Quinssort deve considerare. Se k é troppo grande il rischio é quello di far lavorare troppo InsertionSort, facendo tendere la complessitá alla conplessitá quadratica di Insertion. Se k é troppo piccolo l'algoritmo proposto diventa QuickSort, vanificando lo sforzo di ottenere un algoritmo con costanti moltiplicative inferiori.

La dimensione del blocco dipende al solito da una molteplicitá di fattori, quali l'architettura, il compilatore, il sistema operativo, ecc. ecc. Quindi oltre a dare una stima teorica al blocco, il valore di k verrá calcolato sperimentalmente.

3.6.2 Una stima teorica del blocco

Una volta ottenute le curve teoriche di Insertion e di Quicksort, si puó dedurre la lunghezza k del vettore per cui i due algoritmi impiegano lo stesso tempo (intersezione delle curve). É chiaro che il tempo necessario per ordinare un vettore con questo algoritmo con un blocco di dimensione minore di k sará inferiore al tempo di entrambi. La stima ottima di k che permetterá di ottenere l'algoritmo piú veloce, é la massima differenza tra i tempi di insertion e quicksort.

3.6.3 Codice

3.6.4 Tempo di esecuzione nel caso medio

3.6.5 Confronto dei casi

3.7 Conclusioni

Pascal

```
procedure InsertionSort (var v: Vettore; p, q: integer);
var
j, i: integer;
key: integer;
begin
for j := p + 1 to q do
begin
key := v[j];
i := j - 1;
while (i >= p ) and (v[i] > key) do
begin
v[i + 1] := v[i];
i := i - 1
end;
v[i + 1] := key
end
end;

function Partition (var v: Vettore; p, q: integer): integer;
var
x, i, j: integer;
fine: boolean;
procedure Swap (var v: Vettore; i, j: integer);
var
aux: integer;
begin
aux := v[i];
v[i] := v[j];
v[j] := aux
end;

begin
x := v[p];
i := p - 1;
j := q + 1;
fine := true;
while (fine) do
begin
repeat
j := j - 1
until (v[j] <= x);
repeat
i := i + 1
until (v[i] >= x);
if (i < j) then
Swap(v, i, j)
else
fine := false;
end;
Partition := j
end;
procedure QuinsSort (var v: Vettore; p, q, blocco: integer);
{ ordino a blocchi con quick poi ordino i blocchi con insertion}
procedure BlockQuick (var v: Vettore; p, q, blocco: integer);
{ordino a blocchi con quick poi ordino i blocchi con insertion}
var
r: integer;
begin
if ((q - p) > blocco) then
begin
r := Partition(v, p, q);
BlockQuick(v, p, r, blocco);
BlockQuick(v, r + 1, q, blocco)
end
end;
begin
BlockQuick(v, p, q, blocco);
InsertionSort(v, p, q)
end;
```

Figure 3.31: Codice per Quinssort implementato in pascal

Java

```
class QuinsSort extends Vettore
{
    QuinsSort(int dimensione)
    {
super(dimensione);
}

    public double InitAverage(int inf,int sup,double seed)
    {
int i;
Random num;
num = new Random(seed);
for(i=0;i<GetLength();i++)
   vett[i] = (int) num.get(inf,sup);
return num.GetSeed();
    }
    public void InitWorst()
    {
int i;
for(i=0;i<GetLength();i++)
   vett[i] = i;
    }
    public void Sort(int p,int q,int blocco) //ordino a blocchi con quick poi ordino i blocchi con
insertion
    {
BlockQuick(p,q,blocco);
InsertionSort(p,q);
    }
    private int Partition(int p,int q)
    {
int x,i,j,r;
boolean fine;
x = vett[p];
i = p-1;
j = q+1;
fine = true;
while(fine)
{
    while(vett[--j]>x);
    while(vett[++i]<x);
    if(i<j)
swap(i,j);
    else
fine = false;
}
return j;
    }
    private void swap(int i,int j)
    {
int aux;
aux = vett[i];
vett[i] = vett[j];
vett[j] = aux;
    }
    public void InsertionSort(int p,int q)
    {
int j,i;
int key;
for(j=p+1;j<=q;j++)
{
    key = vett[j];
    for(i=j-1;(i>=0)&&(vett[i]>key);i--)
vett[i+1] = vett[i];
    vett[i+1] = key;
}
    }
    public void BlockQuick(int p,int q,int blocco) //ordino a blocchi con quick poi ordino i blocchi
con insertion
    {
if((q-p)>blocco)
{
    int r;
    r = Partition(p,q);
    BlockQuick(p,r,blocco);
    BlockQuick(r+1,q,blocco);
}
    }
}
```

Figure 3.32: Codice per Quinssort implementato in java

Figure 3.33: Caso medio Quinssort in pascal

Figure 3.34: Caso medio Quinssort in java

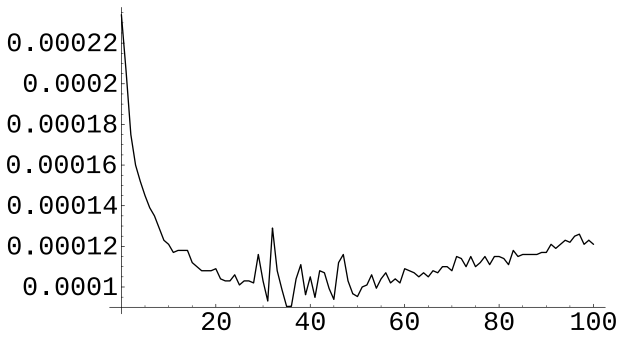

Figure 3.35: Caso medio di Quinssort in pascal

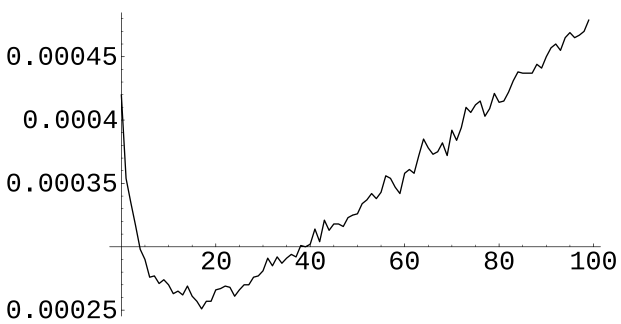

Figure 3.36: Caso medio di Quinssort in java

Chapter 4

Minimi Quadrati

Il problema che ci poniamo è il seguente:

> Data una collezione di m punti del piano cartesiano, (x_i, y_i), per $i = 1, \ldots, m$, determinare una retta che "approssimi" i punti assegnati.

La soluzione del problema dipende, ovviamente, dal senso che attribuiamo alla parola "approssima." Uno dei significati più comuni ed usati nella pratica matematica è quello di cercare, tra tutte le rette del piano, quella retta che minimizzi la distanza tra i singoli punti assegnati e la retta stessa. La trattazione completa del problema è compito dell'Analisi Numerica (che affronta il problema per curve qualsiasi, e non solo per la retta, all'interno del più vasto quadro del *fitting*). Nel nostro semplice caso, tuttavia, possiamo almeno enunciare la formula risolutiva.

Formalmente, si vuole determinare una retta

$$y = bx + a$$

in modo che sia minima la quantità

$$\sum_{i=0}^{m} [y_i - (bx_i + a)]^2.$$

In altre parole, si cercano di determinare a e b in modo da minimizzare la somma dei quadrati delle distanze dei singoli punti dalla retta determinata da a e b. Con calcoli non troppo complessi si può mostrare che le due constanti cercate sono la soluzione del sistema lineare

$$\begin{pmatrix} \Sigma x_i^2 & \Sigma x_i \\ \Sigma x_i & m \end{pmatrix} \begin{pmatrix} b \\ a \end{pmatrix} = \begin{pmatrix} \Sigma y_i x_i \\ \Sigma y_i \end{pmatrix}. \tag{4.1}$$

Lo stesso problema è noto in Statistica Matematica come "analisi di regressione lineare". Indicando, com'è comune in quel contesto, con $\overline{x}$ il valor medio (cioè la media aritmetica) degli x_i (e analogamente per $\overline{y}$) possiamo esprimere le soluzioni della (4.1) come

$$\begin{aligned} a &= \overline{y} - b\overline{x} \\ b &= \frac{S_{xy}}{S_{xx}}, \end{aligned}$$

dove

$$\begin{aligned} S_{xx} &= \Sigma x^2 - \frac{(\Sigma x)^2}{m} \\ S_{xy} &= \Sigma xy - \frac{\Sigma x \cdot \Sigma y}{m}. \end{aligned}$$

Molte semplici calcolatrici da tasca permettono di determinare tutti questi parametri direttamente dalla collezione dei dati (x_i, y_i).