DENTRO LA SCATOLA

Rubrica a cura di

Fabio A. Schreiber

Il Consiglio Scientifico della rivista ha pensato di attuare un'iniziativa culturalmente utile presentando in ogni numero di Mondo Digitale un argomento fondante per l'Informatica e le sue applicazioni; in tal modo, anche il lettore curioso, ma frettoloso, potrà rendersi conto di che cosa sta "**dentro la scatola**". È infatti diffusa la sensazione che lo sviluppo formidabile assunto dal settore e di conseguenza il grande numero di persone di diverse estrazioni culturali che - a vario titolo - si occupano dei calcolatori elettronici e del loro mondo, abbiano nascosto dietro una cortina di nebbia i concetti basilari che lo hanno reso possibile.
Il tema scelto per il 2004 è stato: "***Perché gli anglofoni lo chiamano computer***, ovvero: ***introduzione alle aritmetiche digitali***". Per il 2005 il filo conduttore della serie sarà: "***Ma ce la farà veramente?***, ovvero: ***introduzione alla complessità computazionale e alla indecidibilità***" e il suo intento è di guidare il lettore attraverso gli argomenti fondanti dell'Informatica e alle loro implicazioni pratiche e filosofiche. La realizzazione degli articoli è affidata ad autori che uniscono una grande autorevolezza scientifica e professionale a una notevole capacità divulgativa.

Potenza e limiti del calcolo automatico: la complessità di calcolo degli algoritmi

Angelo Morzenti

1. Introduzione

Questo è il terzo articolo di una serie di quattro destinata a illustrare gli aspetti più concettuali e teorici dell'informatica. I primi due si sono focalizzati sulla nozione di algoritmo come formalizzazione del concetto di calcolo automatico e sulla possibilità di risolvere problemi in modo algoritmico. Qui l'attenzione è rivolta esclusivamente ai problemi risolubili algoritmicamente, per valutare l'efficienza degli algoritmi, intesa come stima della loro capacità di fare uso in maniera ottimale delle risorse di calcolo. Nel fare ciò, cercheremo di essere precisi, adottando metodi quantitativi di valutazione basati su strumenti matematici e di prescindere da dettagli inessenziali, mettendo a fuoco gli aspetti più concettuali degli algoritmi e dei problemi da essi risolti.
Illustreremo concetti e metodi relativi a due dei problemi più rilevanti e maggiormente studiati: quello di ricercare un elemento in un insieme dato e quello di ordinare un insieme di elementi.

2. Oggetto e metodi dell'analisi di complessità

Avendo definito l'efficienza di un algoritmo come la sua capacità di fare un uso il più possibile limitato delle risorse di calcolo, la prima domanda da porsi è: di quali risorse siamo interessati a valutare l'utilizzo. La risposta non è univoca e dipende dallo scopo e dal contesto in cui l'analisi viene svolta. Tra i fattori da considerare ci sono, per esempio, il tempo impiegato per completare l'esecuzione dell'algoritmo, la quantità di memoria, centrale o esterna, richiesta per memorizzare dati e risultati, i dispositivi di interfaccia utilizzati, i canali di comunicazione e le relative larghezze di banda nel caso che il calcolatore comunichi con altri a esso collegati. Nel breve spazio di questo articolo ci concentreremo sulla valutazione del tempo di calcolo, che assume grande rilevanza sia pratica che concettuale. Nel seguito, quando parleremo di "costo" o di "complessità di calcolo" di un'operazione o dell'esecuzione di un algoritmo, intenderemo perciò riferirci al tempo impiegato. L'analisi dello spazio di memoria utilizzato da un algoritmo diventa rilevante per gli algoritmi

che utilizzano complesse strutture dati, che in questo ambito per semplicità e brevità non consideriamo.
Come valutare il tempo di calcolo di un algoritmo? Un semplice e immediato approccio di tipo empirico porterebbe a codificarlo in un linguaggio di programmazione, effettuare una serie di esecuzioni su un determinato calcolatore fornendogli valori di ingresso opportunamente selezionati, cronometrare (tramite opportuni strumenti hardware e software) i tempi di esecuzione per poi compilare tabelle, calcolare statistiche. Questo approccio, se pur del tutto sensato e corretto in certi contesti (tipicamente quando si vogliano effettuare delle operazioni di *benchmarking* per calcolatori), è insoddisfacente ai nostri scopi, perché il tempo di esecuzione di un algoritmo su un determinato calcolatore dipende da svariati fattori di natura contingente che fanno perdere generalità ai risultati ottenuti, per esempio dalla velocità della CPU, delle interfacce, dei bus e delle periferiche, dalla qualità del codice generato dal compilatore (se l'algoritmo è codificato in un linguaggio di alto livello), dalle eventuali interferenze con altri programmi in corso di esecuzione (nel caso che il calcolatore sia multiprogrammato). In realtà noi siamo interessati a studiare gli algoritmi in sé, prescindendo il più possibile dagli aspetti relativi alla piattaforma di calcolo sulla quale vengono eseguiti.
Il primo passo per rendere le nostre valutazioni indipendenti dalla piattaforma è quello di assumere un modello di macchina calcolatrice programmabile il più semplice possibile, al quale tutte le altre (o un sottoinsieme molto significativo) siano riconducibili e adottare un linguaggio per codificare gli algoritmi similmente universale e rappresentativo. Per evidenti ragioni di spazio non possiamo dare una definizione precisa e rigorosa della piattaforma di calcolo e del linguaggio di programmazione cui faremo riferimento nel seguito: facciamo appello all'esperienza e all'intuito del lettore, e lo rimandiamo ai testi citati in bibliografia.
Assumiamo un classico modello di macchina RAM (*Random Access Machine*), cioè di una macchina:

- che può accedere con un costo uniforme a tutte le celle della sua memoria;
- le cui istruzioni vengono eseguite una alla volta senza alcun grado di parallelismo;
- che possiede nel suo repertorio le usuali operazioni aritmetiche (+, –, *, /, arrotondamenti a interi), logiche (connettivi prepositionali di congiunzione, disgiunzione, negazione) e istruzioni per spostare dati dalla memoria alle unità di elaborazione e viceversa;
- che memorizza nelle celle della memoria numeri interi o con parte decimale, caratteri, simboli rappresentanti elementi di insiemi finiti;
- che NON possiede operazioni dissimili da quelle elementari di un comune calcolatore (per esempio, supponiamo di *non* poter calcolare con una singola operazione un'arbitraria potenza di un numero o la versione ordinata di un insieme di elementi);
- che NON fa uso di una gestione gerarchica della memoria, utilizzata nei calcolatori moderni per ottimizzare le prestazioni (memorie cache, memoria virtuale): tali sistemi ne rendono il comportamento assai complesso, difficile da analizzare e poco predicibile.

Il lettore ha sicuramente notato la grande differenza tra la macchina RAM e il modello della macchina di Turing adottato nei precedenti articoli per formalizzare la nozione di algoritmo. La macchina RAM è equivalente alla macchina di Turing per quanto attiene alle questioni di calcolabilità; per gli aspetti riguardanti la complessità di calcolo degli algoritmi la macchina RAM è molto più vicina ai calcolatori elettronici impiegati nella realtà per l'elaborazione automatica dell'informazione.
Il linguaggio in cui esprimiamo gli algoritmi ha le seguenti caratteristiche:

- è ispirato ai più comuni linguaggi di programmazione (C, Pascal, Java...) e contiene le istruzioni della programmazione strutturata (istruzioni iterative e condizionali); nelle istruzioni composte usiamo l'incolonnamento per indicare le sequenze di sotto-istruzioni; l'operatore di assegnamento viene indicato con una freccia verso sinistra "←" (quando l'istruzione $v \leftarrow e$ viene eseguita la variabile v assume il valore dell'espressione e); i commenti iniziano con una doppia sbarra "//"e terminano alla fine della riga;
- gli algoritmi sono scritti sotto forma di procedure con parametri, l'istruzione *restituisci* può essere usata per terminare l'esecuzione restituendo esplicitamente un risultato;
- le variabili rappresentano un dato semplice in una cella di memoria, con l'unica eccezione dei vettori, che rappresentano un insieme di dati elementari memorizzati in celle consecutive: $\boldsymbol{A}[1..n]$

indica un vettore di n elementi, $A[i]$ indica l'i-simo elemento, e *lung*(A) la dimensione del vettore.
Come primo esempio su cui illustrare il metodo di analisi della complessità di calcolo degli algoritmi, consideriamo il problema della ricerca di un elemento in un insieme prefissato, memorizzato in un vettore, e l'algoritmo della cosiddetta ricerca sequenziale che confronta in successione, dal primo all'ultimo, gli elementi del vettore con quello ricercato, fino a quando lo trova (e allora restituisce la posizione nel vettore) oppure oltrepassa la fine del vettore (allora restituisce 0 in segno di insuccesso della ricerca). Il riquadro 1 riporta l'algoritmo; una prima colonna separata indica, mediante costanti simboliche, il costo attribuito a ogni singola operazione e una seconda colonna il numero di volte in cui ogni operazione viene ripetuta.

Riquadro 1

Algoritmo di ricerca sequenziale

Ricerca Sequenziale (A, elem)	Costo	# esecuzioni
$i \leftarrow 1$	c_1	1
fintantoche $i \leq$ lung(A) ripeti	c_2	K, con $1 \leq K \leq$ lung(A) + 1
se $A[i]$ = elem	c_3	H, con $1 \leq H \leq$ lung(A)
allora restituisci i	c_4	0 oppure 1
altrimenti $i \leftarrow i + 1$	c_5	J, con $0 \leq J \leq$ lung(A)
restituisci 0	c_6	0 oppure 1

Il tempo T di esecuzione dell'algoritmo è dato dalla somma dei tempi delle singole istruzioni, e si può esprimere diversamente a seconda che l'elemento cercato sia presente o assente. Nel primo caso, supponendo che l'elemento cercato si trovi in posizione P,

$$T = c_1 + P \cdot (c_2 + c_3) + (P - 1) \cdot c_5 + c_4$$

mentre nel secondo (elemento non presente)

$$T = c_1 + (n + 1) \cdot c_2 + n \cdot (c_3 + c_5) + c_6$$

Il caso a costo minimo (chiamato caso ottimo) è quello in cui l'elemento cercato si trova in prima posizione: in questo caso $T = c_1 + c_2 + c_3 + c_4$; il caso in cui l'elemento non è presente è anche quello in cui viene eseguito il maggior numero di operazioni elementari, e per questo è detto caso pessimo.
Di solito il caso ottimo ha un interesse limitato, perché non è frequente né rappresentativo. Il caso pessimo è invece considerato con grande attenzione quando si vuole adottare un approccio cautelativo e prudenziale tipico delle attività progettuali in cui, per esempio nell'ingegneria civile o meccanica, si utilizzano ipotesi di massimo carico per dimensionare i manufatti progettati. Ciò è particolarmente opportuno nella progettazione dei sistemi informatici che presentano delle criticità (si pensi ad apparecchiature medicali, o a sistemi di controllo dei mezzi di trasporto) in cui è essenziale poter contare sul fatto che una certa componente che esegue un algoritmo termini producendo il risultato richiesto entro un tempo prefissato. Per questi motivi nel seguito concentreremo le nostre valutazioni di complessità sul caso pessimo. Per i sistemi, meno critici, che hanno al più l'esigenza di massimizzare le prestazioni a fronte di un numero molto alto di esecuzioni con dati diversi (per esempio sistemi transazionali in ambito amministrativo), i casi ottimo e pessimo non sono rappresentativi delle prestazioni complessive, che vengono valutate con metodi di tipo statistico, a partire da appropriate ipotesi sulla distribuzione dei dati. La valutazione del caso medio è spesso più complessa e laboriosa di quelli ottimo e pessimo: per la ricerca sequenziale, supponendo che il dato ricercato sia presente e che con la stessa probabilità si trovi in una qualsiasi delle n posizione nel vettore, il costo medio richiede di valutare la media di tutti i valori della posizione P, ottenendo perciò:

$$T = c_1 + \frac{n}{2} \cdot (c_2 + c_3) + \frac{n-1}{2} \cdot c_5 + c_4$$

Possiamo constatare, nei casi pessimo e medio appena esaminati, che la complessità temporale dipende da n, la dimensione dei dati in ingresso. Ciò è tipico di tutti i problemi non banali, perciò per un algoritmo che abbia dei dati in ingresso di dimensione n il tempo di esecuzione viene indicato, evidenziando la dipendenza dal parametro, come una funzione $T(n)$.
Inoltre nel considerare la funzione $T(n)$ si cerca di darne una caratterizzazione il più possibile sintetica e generale, che permetta di prescindere dai fattori relativi alla piattaforma di calcolo su cui viene eseguito l'algoritmo (e quindi dalle costanti c_1, ... c_6 nel nostro esempio) e dall'influenza che possono avere sull'andamento della funzione le parti marginali dell'algoritmo, per esempio quelle che vengono eseguite un numero fisso di volte. Si astrae da questi fattori

mediante lo strumento matematico della valutazione dell'andamento asintotico delle funzioni, che stima la loro velocità di crescita prescindendo dalle costanti di proporzionalità e considerando valori della variabile indipendente n sufficientemente elevati da escludere i termini di ordine inferiore. Per esempio, per la complessità della ricerca sequenziale si dice che $T(n)$ è $\Theta(n)$, intendendo che cresce allo stesso modo della funzione $f(n) = n$. In generale, quando la funzione $T(n)$ è composta da più termini e contiene costanti moltiplicative, si considera il termine dominante (quello che cresce più velocemente con la variabile n) e si ignorano le costanti moltiplicative; così facendo si ottiene la più semplice funzione $f(n)$ che ha lo stesso andamento asintotico di $T(n)$, e si dice che $T(n)$ è $\Theta(f(n))$.
La notazione Θ è un potente strumento di astrazione: due algoritmi A_1 e A_2 che hanno complessità $T_1(n)$ e $T_2(n)$ che siano entrambe $\Theta(f(n))$ per qualche funzione $f(n)$ sono considerate equivalenti per quanto riguarda la complessità temporale.

3. Ricerca e ordinamento

Ricordiamo ora che la nozione di algoritmo è ben distinta da quella di problema: per ogni problema si possono trovare più algoritmi che lo risolvono e che differiscono per vari aspetti, tra cui la complessità di calcolo. Illustriamo questo fatto su due problemi classici dell'informatica, tra i meglio studiati a causa della loro grande rilevanza teorica e applicativa: quello della ricerca (per il quale si è già visto l'algoritmo di ricerca sequenziale) e quello dell'ordinamento (che, dato un insieme di elementi disposti in sequenza in un ordine qualsiasi, consiste nel disporli in senso crescente). È istruttivo considerare un algoritmo alternativo alla ricerca sequenziale che, sotto l'ipotesi che l'insieme degli elementi in cui effettuare la ricerca siano disposti in ordine crescente, permette di svolgere la ricerca in modo molto più efficiente. Si tratta della ricerca binaria, codificata nella procedura mostrata nel riquadro 2. La ricerca viene svolta con riferimento a un segmento di vettore individuato dai due indici *inf* e *sup*; ad ogni iterazione l'elemento ricercato *elem* viene confrontato con quello presente nel punto mediano del segmento considerato: se è uguale la ricerca termina con successo, se è maggiore, la ricerca può proseguire nella seconda parte del segmento (perché essendo il vettore ordinato, *elem* è maggiore anche di tutti gli elementi della prima parte del segmento), altrimenti, prosegue nella prima metà.

Riquadro 2
Algoritmo di ricerca binaria

```
Ricerca Binaria (A, elem)
  inf ← 1
  sup ← lung(A)
  med ← (inf + sup) / 2
  fintantoche (inf ≤ sup) e (elem ≠ A[med]) ripeti
    se elem > A[i]
      allora inf ← med + 1 //scarta la prima metà del segmento di vettore
      altrimenti sup ← med – 1 //scarta la seconda metà del segmento
                                //di vettore
  se inf ≤ sup //se inf ≤ sup allora elem = A[med]
    allora restituisci med
    altrimenti restituisci 0
```

Valutiamo questo algoritmo in modo informale e sintetico, tenendo presente che siamo interessati alla complessità asintotica nel caso pessimo. Basta quindi contare il numero di volte in cui viene ripetuto il corpo del ciclo: poiché il numero degli elementi del segmento sotto esame viene dimezzato a ogni ripetizione, il numero massimo di ripetizioni è pari al numero di volte per cui occorre dividere un numero n per 2 per arrivare a 0, e questo valore è strettamente legato al logaritmo in base 2 di n, quindi $T(n)$ è $\Theta(\log n)$. È noto che la funzione $\log n$ cresce molto lentamente al crescere di n (aumenta di 1 al raddoppiare di n; per n pari a circa un miliardo $\log n$ vale circa 30, per n pari a circa mille miliardi vale circa 40...). Effettuare una ricerca in un insieme di dimensioni rilevanti è molto più efficiente se l'insieme è ordinato, come ben sanno tutti i genitori che cercano di (far) tenere in ordine la camera dei propri figli, e tutti coloro che hanno mai provato a immaginare quanto sarebbe utile una guida telefonica con gli abbonati riportati in un ordine casuale.
L'algoritmo di ricerca binaria ci dà l'opportunità di constatare un fatto che vale per tutti gli algoritmi di ricerca e anche per quelli di ordinamento: la valutazione della complessità asintotica può essere svolta conteggiando il numero di confronti tra elementi, trascurando le altre operazioni (valutazione e assegnamento di variabili, gestione delle istruzioni iterative...). La ragio-

ne di ciò è duplice. In primo luogo, nel caso in cui gli elementi da confrontare abbiano una complessa struttura (e.g., siano numeri elevati, polinomi, vettori di numeri o stringhe) allora l'operazione di confronto diventa più onerosa e il suo costo prevalente rispetto a quello delle altre; la seconda, e più importante ragione, particolarmente evidente nel caso della ricerca binaria, è che il numero delle altre operazioni è proporzionale al numero dei confronti e quindi la loro inclusione nel conteggio non porterebbe a risultati diversi per quanto riguarda la valutazione asintotica della complessità.

Non c'è bisogno di enfatizzare la rilevanza dal punto di vista pratico dell'operazione di ricerca di un elemento in un insieme: si pensi anche solo alla miriade di applicazioni informatiche di tipo amministrativo e gestionale che ne fanno uso. Alla luce del rilevantissimo miglioramento nell'efficienza ottenibile dalla ricerca binaria rispetto a quella sequenziale grazie all'ipotesi di disposizione ordinata degli elementi nel vettore ***A***, l'operazione di ordinamento assume anch'essa grande importanza e infatti questo problema è stato studiato a fondo, portando alla formulazione di un gran numero di algoritmi; ne analizziamo ora due tra i più significativi: l'ordinamento per inserzione e quello per fusione.

L'operato dell'ordinamento per inserzione, riportato nel riquadro 3, è simile a quello di un giocatore che, all'inizio di una partita, raccoglie le carte dal banco e le dispone ordinatamente in un mazzo che tiene in mano. Se ha già raccolto il 3, il 4 e il 7 e li tiene in mano in questo ordine, e poi raccoglie il 6, lo infila tra il 4 e il 7.

L'istruzione "*per tutti i*" scandisce gli elementi del vettore dal secondo all'ultimo e l'istruzione *fintantoché*, per ogni elemento in posizione j, con $2 \le j \le \text{lung}(A)$, lo porta nella posizione che gli compete nel segmento iniziale di vettore, lungo $j-1$, già ordinata nelle passate precedenti. A ogni ripetizione del ciclo *fintantoché*, la lunghezza del segmento iniziale ordinato si allunga di 1, quindi alla fine del processo tutti gli elementi sono disposti in ordine crescente. Lo sforzo complessivo per portare l'elemento $A[j]$ nella posizione giusta dipende dal suo valore relativamente a quello degli elementi che occupano le posizioni precedenti da 1 a $j-1$: se $A[j]$ è minore di tutti lo sforzo è massimo, proporzionale a j (il valore P_j è pari a j, perché occorre confrontare $a[j]$ con tutti gli elementi che lo precedono); all'estremo opposto, se $A[j]$ è maggiore di tutti gli $A[j]$, $1 \le i < j$, poiché questi sono ordinati in senso crescente basterà confrontare $A[j]$ con $A[j-1]$ per concludere che $A[j]$ si trova già in posizione corretta e non deve essere spostato. Abbiamo quindi individuato il caso ottimo e quello pessimo. Il caso ottimo è quello in cui il vettore è già ordinato, il parametro P_j nella sommatoria vale 1 e $T(n)$ è $\Theta(n)$, cioè la funzione di complessità è lineare. Il caso pessimo è quello con $A[j]$ minore di tutti gli elementi che lo precedono, cioè il vettore è ordinato in senso decrescente, il parametro P_j vale sempre j e, poiché la sommatoria dei primi n numeri interi vale $n \cdot (n-1)/2$, $T(n)$ è $\Theta(n^2)$. L'ordinamento per inserzione è quindi quadratico nel caso pessimo.

Esaminiamo ora il secondo algoritmo di ordinamento, quello per fusione. L'algoritmo è molto intuitivo nella sua formulazione più astratta, ma ben più complesso dei precedenti se ne consideriamo tutti i dettagli: ne diamo quindi una descrizione di massima e uno schema di codice incompleto, omettendo alcuni dettagli ininfluenti ai fini della valutazione di complessità.

L'idea alla base dell'algoritmo, che ne determina anche il nome, è descritta nel riquadro 4: il vettore da ordinare viene diviso in due parti, le parti ottenute vengono ordinate (*ricorsivamente*, cioè applicando a esse lo stesso algoritmo) e poi fuse ottenendo la versione ordinata del vettore di partenza. La procedura di ordinamento per fusione, riportata nel riquadro 5, ha quindi due parametri aggiuntivi, gli indici *inf* e *sup* che

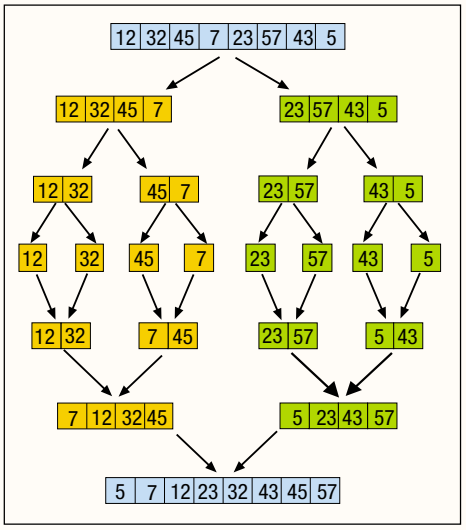
Riquadro 3

Ordinamento per inserzione

Ordina per Inserzione (A)	Costo	# esecuzioni
per tutti i j da 2 a lung(A) ripeti	c_1	n
elem $\leftarrow A[j]$	c_2	$n-1$
//inserisci $A[j]$ nella porzione		
//ordinata $A[1..j-1]$ del vettore		
$i \leftarrow j-1$	c_3	$n-1$
fintantoche ($i > 0$) e (elem $< A[i]$) ripeti	c_4	$\sum_{j=2}^{n} P_j$
$A[i+1] \leftarrow A[i]$	c_5	$\sum_{j=2}^{n} (P_j - 1)$
$i \leftarrow i-1$	c_6	$\sum_{j=2}^{n} (P_j - 1)$
$A[i+1] \leftarrow$ elem	c_7	$n-1$

Riquadro 4

Ordinamento per fusione: simulazione grafica



Riquadro 5

Ordinamento per fusione: codice

```
Ordina per Fusione (A, inf, sup)
  se inf < sup
    allora  med ← (inf + sup) / 2
            Ordina per Fusione (A, inf, med) //ordina la metà inferiore
                                             //del segmento
            Ordina per Fusione (A, med + 1, sup)//ordina la metà superiore
                                             //del segmento
            fondi (A, inf, med, sup)
```

(come nella ricerca binaria) delimitano la porzione del vettore alla quale viene applicata.

Per valutare la complessità dell'ordinamento per fusione osserviamo che la divisione in due del segmento da ordinare non effettua alcuna operazione sul vettore (consiste solo nel calcolo della posizione mediana) e quindi ha un costo costante, indipendente dal numero di elementi del segmento di vettore da ordinare. Il grosso del lavoro viene quindi svolto nell'operazione di fusione; per brevità e semplicità non codifichiamo la procedura *fondi*, ma, osservando il riquadro 4, è facile comprendere come si svolge: gli elementi da inserire nel segmento di vettore risultato della fusione vengono presi in sequenza, dai più piccoli ai più grandi (cioè andando da sinistra a destra) da una o dall'altra delle due porzioni da fondere, sfruttando il fatto che queste sono già (ricorsivamente) ordinate. A ogni inserzione si sceglie il minore tra il primo elemento di una porzione e il primo dell'altra, lo si preleva dalla porzione in cui si trova e lo si immette nella porzione di vettore ordinato in corso di costruzione, in posizione successiva a quella degli elementi immessi nei passi precedenti (cioè alla loro destra), in modo che anch'essa risulti ordinata in senso crescente. Sulla base di questa, pur informale, descrizione possiamo facilmente concludere che la complessità dell'operazione di fusione è proporzionale al numero complessivo degli elementi delle due porzioni che vengono fuse, cioè è $\Theta(k)$, dove $k = sup\text{-}inf + 1$; l'algoritmo *fondi* ha perciò complessità $T(n)$, è $\Theta(n)$ ovvero è lineare. Per completare la valutazione della complessità facciamo riferimento al riquadro 4 e "tiriamo le somme" considerando tutte le operazioni di fusione effettuate globalmente dall'algoritmo, prescindendo dal loro ordine, evidentemente irrilevante. Considerando le sezioni orizzontali del riquadro 4 che descrivono le operazioni di fusione, notiamo che in ognuna di esse il numero complessivo di elementi coinvolti nell'operazione di fusione è sempre lo stesso, cioè n, il numero complessivo di elementi del vettore da ordinare (nelle sezioni alte del riquadro 4 si fondono molti segmenti piccoli, precisamente 2^i segmenti con $n/2^i$ elementi ognuno, mentre nelle sezioni basse si fondono pochi segmenti più grandi, ma il numero totale di elementi è sempre n). Quante sono le sezioni orizzontali del riquadro 4? Possiamo stabilirlo con un ragionamento simile a quello fatto per valutare la complessità della ricerca binaria: ogni volta un segmento viene diviso in due, quindi il numero delle sezioni orizzontali è $\log n$. In conclusione il costo complessivo dell'algoritmo di ordinamento per fusione è proporzionale a n (il costo delle fusioni complessivamente effettuate in corrispondenza a una delle sezioni orizzontali del riquadro 4) moltiplicato per $\log n$ (il numero delle sezioni orizzontali), quindi la complessità $T(n)$ è $\Theta(n \cdot \log n)$.

4. Si può fare di meglio? Problemi aperti e chiusi

Abbiamo visto sui due esempi della ricerca e dell'ordinamento che, come nella realtà quotidiana ci sono più modi per risolvere un problema, anche in informatica si possono trovare di-

versi algoritmi, che differiscono per logica di funzionamento e per complessità di calcolo. È chiaro che gli algoritmi più facili e immediati da concepire e formulare non hanno molta probabilità di essere efficienti: per ottenere i risultati migliori sono necessari una profonda comprensione della natura del problema e fantasia per inventare soluzioni originali e non banali. La fiducia nell'ingegno e nell'inventiva umani ci può far credere che, con sufficienti sforzi, per qualsiasi problema possano essere trovate soluzioni sempre più efficienti. Lo scenario alternativo a quello appena prospettato è quello in cui per un determinato problema, esista una certa soglia di efficienza algoritmica invalicabile e quindi, una volta trovato un algoritmo che appartenga a questa classe di complessità asintotica, per quanti sforzi si facciano per ideare nuovi algoritmi, non si riesca a oltrepassare questa barriera, perché i nuovi algoritmi portano soltanto miglioramenti marginali (per esempio, abbassano le costanti moltiplicative o migliorano le prestazioni per valori limitati delle dimensioni dei dati). Ebbene, purtroppo questo secondo scenario è quello corrispondente alla realtà. Per esempio, per il problema dell'ordinamento, è possibile dimostrare che un qualsiasi algoritmo che si basi su operazioni di confronto tra gli elementi da ordinare ha necessariamente complessità asintotica di classe $\Theta(n \cdot \log n)$ o peggiore. In altri termini, $n \cdot \log n$ costituisce un *limite inferiore* alla complessità del problema dell'ordinamento; sappiamo inoltre che questo limite viene raggiunto dall'algoritmo, relativamente semplice, di ordinamento per fusione. Similmente, si dimostra che qualsiasi algoritmo di ricerca basato su operazioni di confronto deve avere complessità almeno logaritmica, come l'algoritmo di ricerca binaria.

Se, per un determinato problema, si riesce a trovare una classe di complessità che ne costituisce un limite inferiore e allo stesso tempo è noto un algoritmo con complessità appartenente a tale classe, si dice che tale problema è *chiuso*. Il problema dell'ordinamento è dunque chiuso. Oltre all'algoritmo di ordinamento per fusione, sono noti in letteratura e utilizzati nelle applicazioni industriali dell'informatica, altri algoritmi, quali il *quicksort* e l'*heapsort*, che sono migliori dell'ordinamento per fusione sotto certi aspetti (per esempio, la complessità nel caso medio, o l'uso della memoria) ma gli sono equivalenti dal punto di vista della complessità asintotica per il caso pessimo.

Fornire un limite inferiore alla complessità di un problema è solitamente difficile, a causa della generalità e dell'astrattezza del procedimento dimostrativo, che deve riuscire a prescindere da ogni particolare algoritmo risolutivo di tale problema e allo stesso tempo esser valido per tutti i possibili algoritmi (inclusi anche quelli non ancora inventati), modellando gli aspetti essenziali che li accomunano; tali dimostrazioni si basano spesso su ragionamenti di natura combinatoria, e utilizzano risultati della matematica non elementare.

Bibliografia

Gli argomenti trattati in questo articolo sono tipicamente oggetto dei corsi universitari di algoritmi e strutture dati. Tra gli innumerevoli testi a carattere divulgativo, didattico o scientifico, segnaliamo in particolare i due seguenti. Il testo di Udi Manber [1] costituisce un'eccellente introduzione agli algoritmi, alle tecniche per la loro progettazione e analisi, molto godibile e avvincente nella lettura. L'opera di Cormen, Leisersin, Rivest e Stein [2] fornisce una trattazione sistematica, di carattere enciclopedico e approfondito.

[1] Udi Manber: *Introduction to Algorithms: a Creative approach*. Addison-Wesley, Reading, MA, 1989.

[2] Cormen Thomas H., Leiserson Charles E., Rivest Ronald R., Stein Clifford: *Introduction to Algorithms*. 2-nd Edition, The MIT Press, Cambridge, MA, 2001 (disponibile anche l'edizione in italiano da McGraw-Hill Italia, 2005).

Angelo Morzenti consegue il dottorato di ricerca in Ingegneria Elettronica dell'Informazione e dei Sistemi al Politecnico di Milano nel 1988. Dal 1990 è ricercatore e dal 2001 professore ordinario al Politecnico di Milano. I suoi interessi di ricerca sono rivolti ai linguaggi, i metodi e gli strumenti per la specifica, l'analisi, la verifica e la progettazione di sistemi informatici ad alta criticità, embedded e in tempo reale. Su questi temi Angelo Morzenti pubblica numerosi lavori sulle più autorevoli riviste internazionali e conduce progetti di ricerca nazionali e internazionali.
morzenti@elet.polimi.it