Capitolo 2

ANALISI DI ALGORITMI

Per risolvere un problema, spesso sono disponibili molti algoritmi diversi. Come scegliere il "migliore"? Un criterio generalmente adottato consiste nel valutare la "bontà" di un algoritmo in base alla quantità di risorsa utilizzata per il calcolo. La risorsa di interesse dominante è il tempo richiesto per eseguire le azioni elementari, ma talvolta è considerato anche lo spazio necessario per immagazzinare (o memorizzare) e manipolare i dati.

2.1 TEMPO DI CALCOLO

In pratica, non è possibile misurare il tempo di calcolo di un algoritmo col numero di secondi richiesto per eseguire su un calcolatore elettronico una procedura che lo descriva. Infatti tale tempo dipende pesantemente dai dati di ingresso, dal linguaggio in cui la procedura è descritta, dalla qualità con cui viene automaticamente tradotta dal compilatore in una sequenza di bit e dalla natura e velocità del calcolatore elettronico. Occorre una misura "astratta" che tenga maggiormente conto del "metodo di risoluzione" con cui l'algoritmo effettua la computazione.

Poiché i problemi da risolvere hanno una dimensione che dipende dalla grandezza dei dati di ingresso, viene spontaneo esprimere il tempo di calcolo come il costo complessivo delle operazioni elementari in funzione della dimensione n dei dati di ingresso. Sono considerate elementari le operazioni aritmetiche, logiche, di confronto e di assegnamento. Talvolta, è utile considerare non tutte le operazioni, ma solo quelle dominanti, cioè quelle che incidono maggiormente sul tempo di esecuzione. Il costo delle operazioni è principalmente valutato nel *caso pessimo*, cioè sul dato d'ingresso più sfavorevole, tra tutti quelli di dimensione n, ma talvolta è valutato anche nel *caso medio*, cioè mediando su tutti i possibili dati di dimensione n, tenendo conto della probabilità con cui ciascun dato può occorrere.

2.1 ESEMPIO – [Tempo di calcolo di min() iterativa]
Per valutare il tempo di calcolo della funzione iterativa min(), si osservi che la funzione min() è formata da un numero finito di istruzioni elementari. Ogni istruzione richiede un tempo costante di esecuzione, ma la costante può essere diversa da istruzione a istruzione. Indichiamo con c_h la costante richiesta per l'esecuzione dell'istruzione h-esima. Rivediamo la funzione min() indicando, per ogni istruzione, il costo di un'esecuzione ed il numero totale di volte che l'istruzione viene eseguita. Tenendo conto che la dimensione del problema è n si ottiene:

ITEM min(ITEM[] A, **integer** n)

	Costo	#Volte
ITEM $min \leftarrow A[1]$	c_1	1
for integer $i \leftarrow 2$ **to** n **do**	c_2	n
if $A[i] < min$ **then**	c_3	$n-1$
$min \leftarrow A[i]$	c_4	$n-1$
return min	c_5	1

Si noti che l'incremento dell'indice i nel ciclo **for** è ripetuto n volte e non $n-1$. Infatti il **for**, come spiegato precedentemente, è un modo più compatto di scrivere un **while**, nel quale la condizione $i \leq n$ deve essere verificata una volta in più per poter uscire dal ciclo quando i supera n. Il tempo di calcolo $T(n)$ di min() si ottiene sommando il prodotto del costo di ciascuna istruzione per il numero di volte che tale istruzione è eseguita:

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

Pertanto, il tempo di calcolo di min() può essere espresso come

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

con a e b costanti intere positive. Si noti che questo andamento di $T(n)$ lineare in n vale sia nel caso pessimo che in quello medio, poiché il corpo del **for** è sempre ripetuto esattamente $n-1$ volte, qualsiasi sia il dato A d'ingresso.

2.2 ESEMPIO – [Tempo di calcolo di binarySearch() ricorsiva]
Valutare il tempo di un algoritmo ricorsivo quale binarySearch() è un po' più complicato. Innanzitutto, l'elemento cercato potrebbe essere trovato "al primo colpo", ovvero subito nella posizione mediana; questo è il caso ottimo. Noi invece valuteremo il caso in cui non sia presente, che corrisponde al caso pessimo. Secondo, vengono eseguite parti diverse dell'algoritmo a seconda dei valori di i e j: se $i > j$, la ricorsione si ferma, altrimenti viene effettuata una chiamata ricorsiva. Nel codice della binarySearch() inseriamo quindi due colonne # che valutano il numero di volte che una particolare riga viene eseguita in ognuno dei possibili casi.

integer binarysearch(ITEM[] A, ITEM v, **integer** i, **integer** j)

	Costo	$\#(i > j)$	$\#(i \leq j)$
if $i > j$ **then**	c_1	1	1
return 0	c_2	1	0
else			
integer $m \leftarrow \lfloor (i+j)/2 \rfloor$	c_3	0	1
if $A[m] = v$ **then**	c_4	0	1
return m	c_5	0	0
else if $A[m] < v$ **then**	c_6	0	1
return binarySearch(A, v, $m+1$, j)	$c_7 + T(\lfloor (n-1)/2 \rfloor)$	0	0/1
else			
return binarySearch(A, v, i, $m-1$)	$c_7 + T(\lfloor n/2 \rfloor)$	0	0/1

Si indichi ancora con $T(n)$ il tempo richiesto per un dato d'ingresso di dimensione n. Si assuma dapprima che $n = 0$ (cioè che $i > j$). In tal caso la funzione esegue direttamente la condizione di chiusura. Se non vi è alcun elemento, il tempo di calcolo è $c_1 + c_2$ e quindi:

$$T(0) = c$$

dove c è una costante.

Altrimenti, la funzione suddivide il vettore in due porzioni grandi $\lfloor (n-1)/2 \rfloor$ e $\lfloor n/2 \rfloor$ (se n è pari, queste hanno dimensione $n/2 - 1$ e $n/2$; se n è dispari, hanno entrambe dimensione $\lfloor n/2 \rfloor$). Per semplificare i calcoli, assumiamo che n sia una potenza di 2, ovvero $n = 2^k$ per un qualche k intero positivo. Ancora per il caso pessimo, supponiamo che si scelga sempre il lato più grande (quello di dimensione $n/2$, essendo n pari). Il costo della funzione è quindi:

$$\begin{aligned} T(n) &= T(n/2) + c_1 + c_3 + c_4 + c_6 + c_7 = \\ &= T(n/2) + d \end{aligned}$$

dove d è una costante. Il tempo di calcolo $T(n)$ può essere dunque ricavato dalla soluzione delle seguenti relazioni, dette *relazioni di ricorrenza*:

$$\begin{aligned} T(n) &= c && \text{se } n = 0 \\ T(n) &= T(n/2) + d && \text{se } n \geq 1 \end{aligned}$$

Una tecnica utile per risolvere relazioni di ricorrenza consiste nel produrre una catena di uguaglianze ottenute per sostituzioni successive, applicando le relazioni per $n/2$, $n/4$, $n/8$, ..., 2, 1, 0. Si ottiene:

$$\begin{aligned} T(n) &= T(n/2) + d \\ &= T(n/4) + 2d \\ &= T(n/8) + 3d \\ &\ldots \\ &= T(1) + kd \\ &= T(0) + (k+1)d \\ &= kd + (c+d) \\ &= d \log n + e \end{aligned}$$

Quindi il caso pessimo di binarySearch() richiede un tempo di calcolo che, a meno dei valori delle costanti d e e, cresce come il logaritmo di n.

In pratica, non è necessario tener conto esplicitamente dei costi delle singole operazioni. Infatti, ha poca importanza che una moltiplicazione sia, mettiamo, dieci volte più lenta di una addizione, quando alla fin fine entrambe le operazioni richiedono tempi costanti, seppur diversi. Di fatto, tutte le costanti delle singole operazioni possono venire "inglobate" in poche costanti, per esempio a e b, quando si conclude che il tempo di calcolo $T(n)$ cresce linearmente in n, cioè come $an + b$. Pertanto, il tempo di calcolo può essere valutato semplicemente contando il numero di operazioni elementari eseguite nel caso pessimo (o medio) su tutti i dati di ingresso di dimensione n.

Un'obiezione che può essere fatta alla valutazione del tempo di calcolo nel caso pessimo è che sia appunto troppo "pessimistica", perché il caso pessimo potrebbe verificarsi molto raramente. Tale valutazione presenta però l'indubbio vantaggio di garantire che l'algoritmo non richiederà mai, per nessun dato di ingresso di dimensione n, un tempo maggiore. Inoltre il caso pessimo, per certi problemi, occorre molto di frequente, come nel verificare se un dato elemento appartiene oppure no ad un insieme, il caso pessimo occorre tutte le volte che si inserisce un elemento e si verifica prima che non sia già incluso! Inoltre, benché la valutazione nel caso medio possa apparire più realistica, spesso non è chiaro sotto quali ipotesi di distribuzione di probabilità debba essere svolta. In genere viene assunta una distribuzione uniforme (in cui tutti i casi sono equiprobabili), che per molti problemi è irrealistica o porta allo stesso risultato, a meno di costanti, del caso pessimo. Per esempio, assumendo che un dato elemento appaia come i-esimo in un insieme di n elementi con la stessa probabilità $1/n$, per reperirlo occorre un tempo lineare in n sia nel caso medio che in quello pessimo, perché bisogna esaminare $n/2$ elementi in media ed n nel caso pessimo. Inoltre, la valutazione nel caso medio richiede l'impiego di formule matematiche molto complicate e di difficile risoluzione. A parte poche eccezioni, la valutazione del tempo di calcolo principalmente considerata è quella nel caso pessimo. Talvolta è anche considerato il cosiddetto *tempo ammortizzato*, dove il tempo $T(m)$ richiesto nel caso pessimo per eseguire una sequenza di m operazioni (ad esempio, ricerche, inserimenti e cancellazioni di elementi in un insieme) viene diviso per il numero di operazioni, ottenendo un tempo medio $T(m)/m$ per operazione, detto appunto ammortizzato. È bene notare che in questo caso, a differenza della valutazione nel caso medio, non sono affatto coinvolte probabilità, poiché il tempo ammortizzato è una sorta di tempo medio per ciascuna operazione valutato nel caso pessimo su tutte le sequenze di m operazioni.

2.2 ORDINE DI GRANDEZZA E COMPLESSITÀ

Nel valutare il tempo di calcolo $T(n)$ di una procedura, è in genere assai difficile quantificare con esattezza le costanti coinvolte. Questa difficoltà viene aggirata valutando il numero di operazioni in *ordine di grandezza*, cioè esprimendolo come limitazione della funzione $T(n)$ al tendere all'infinito della dimensione n, trascurando le costanti moltiplicative ed additive. Si parla così di *complessità computazionale asintotica* in ordine di grandezza (o, brevemente, *complessità*) di una procedura.

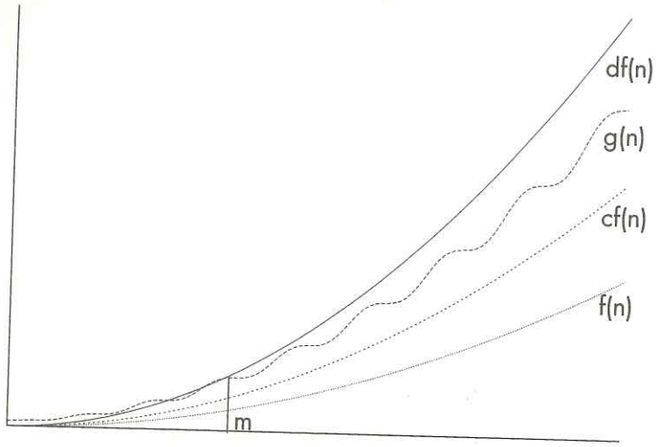

Fig. 2.1 Una funzione $g(n)$ che è $\Theta(f(n))$.

A tal fine, si usano le notazioni "O", "Ω" e "Θ" definite come segue:

- $O(f(n))$ è l'insieme di tutte le funzioni $g(n)$ tali che esistano due costanti positive c ed m per cui $g(n) \leq cf(n)$ per ogni $n \geq m$;
- $\Omega(f(n))$ è l'insieme di tutte le funzioni $g(n)$ tali che esistano due costanti positive c ed m per cui $cf(n) \leq g(n)$ per ogni $n \geq m$;
- $\Theta(f(n))$ è l'insieme di tutte le funzioni $g(n)$ tali che esistano tre costanti positive c, d ed m per cui $cf(n) \leq g(n) \leq df(n)$ per ogni $n \geq m$.

Si dice che "una funzione $g(n)$ è di ordine omicron di $f(n)$" (in breve: "$g(n)$ è $O(f(n))$") per indicare una limitazione superiore al comportamento asintotico di $g(n)$. Analogamente, si dice che "$g(n)$ è di ordine omega di $f(n)$" (in breve: "$g(n)$ è $\Omega(f(n))$") per indicarne una limitazione inferiore, e che "è di ordine theta di $f(n)$" (in breve: "$g(n)$ è $\Theta(f(n))$") per indicarne sia una limitazione inferiore che superiore[1].

2.3 ESEMPIO – [Ordini di grandezza]
La funzione $g(n) = 4n^2 + 4n - 1$ è $O(n^2)$, poiché esistono le costanti $c = 5$ ed $m = 4$ per cui $g(n) \leq 5n^2$ per $n \geq 4$. $g(n)$ è anche $\Omega(n^2)$, perché esistono le costanti $c = 1$ ed $m = 1$ tali che $g(n) \geq n^2$ per $n \geq 1$. Si noti che, in accordo alla definizione, $g(n)$ è $O(n^k)$ per ogni $k \geq 2$, ed anche $\Omega(n^k)$ per ogni $k \leq 2$. In genere però si richiede l'ordine più stretto che è $\Theta(n^2)$, poiché $g(n)$ è sia $O(n^2)$ che $\Omega(n^2)$. Similmente, è facile verificare che la funzione $T(n)$ dell'Esempio 2.1 è $\Theta(n)$.

[1] Dato che $\Theta(f(n))$ è un insieme, alcuni autori, per indicare che $g(n)$ è $\Theta(f(n))$, utilizzano correttamente la notazione $g(n) \in \Theta(f(n))$, mentre altri usano impropriamente la notazione $g(n) = \Theta(f(n))$, intendendo esprimere con ciò che $\Theta(g(n)) = \Theta(f(n))$.

Per stimare gli ordini di grandezza, non è necessario ogni volta applicarne le definizioni e ricavare esplicitamente le costanti richieste. Si possono invece usare le seguenti regole, la cui dimostrazione è lasciata per esercizio, che permettono di semplificare la stima dell'ordine di grandezza di una funzione non negativa combinando insieme le stime delle sue parti.

(1) *Riflessività*: Per ogni costante c (inclusa quindi $c = 1$) ed ogni funzione f, $cf(n)$ è $O(f(n))$ (lo stesso vale per Ω e per Θ);

(2) *Transitività*: Se $g(n)$ è $O(f(n))$ ed $f(n)$ è $O(h(n))$, allora $g(n)$ è $O(h(n))$ (lo stesso vale per Ω e per Θ);

(3) *Simmetria*: $g(n)$ è $\Theta(f(n))$ se e solo se $f(n)$ è $\Theta(g(n))$;

(4) *Simmetria trasposta*: $g(n)$ è $O(f(n))$ se e solo se $f(n)$ è $\Omega(g(n))$;

(5) *Somma*: La funzione $f(n) + g(n)$ è $O(max\{f(n), g(n)\})$ (lo stesso vale per Ω e per Θ);

(6) *Prodotto*: Se $g(n)$ è $O(f(n))$ ed $h(n)$ è $O(q(n))$, allora la funzione $g(n)h(n)$ è $O(f(n)q(n))$ (lo stesso vale per Ω e per Θ).

2.3 ESEMPIO – [continua]
La funzione $g(n) = 4n^2 + 4n - 1$ è $O(n^2)$, poiché $4n$ è $O(n)$ per la regola (1), $4n^2 = 4nn$ è $O(nn)$ per la regola (6), ed infine $4n^2 + 4n - 1$ è $O(\max\{n^2, n, 1\})$ applicando due volte la (5).

Si noti che le proprietà 1, 2, e 3 implicano che Θ definisce una relazione di equivalenza sulle funzioni, tale che ogni insieme $\Theta(f(n))$ è una classe di equivalenza (detta *classe di complessità*).

2.4 ESEMPIO – [Classificazione ordini di grandezza]
I seguenti ordini di grandezza sono via via crescenti: $\Theta(1)$ (ordine costante), $\Theta(\log n)$ (logaritmico), $\Theta(n)$ (lineare), $\Theta(n \log n)$ (pseudolineare), $\Theta(n^2)$ (quadratico), $\Theta(n^3)$ (cubico), $\Theta(2^n)$ (esponenziale in base 2), $\Theta(n^n)$ (esponenziale in base n). È possibile tracciare delle relazioni di inclusione anche più generali, per qualsiasi $r < s$, $h < k$ e $1 < a < b$:

$$O(1) \subset O(\log^r n) \subset O(\log^s n) \subset O(n^h) \subset O(n^h \log^r n) \subset$$
$$\subset O(n^h \log^s n) \subset O(n^k) \subset O(a^n) \subset O(b^n) \subset O(n^n)$$

In generale, un ordine $\Theta(n^k)$, con k costante positiva, viene detto *polinomiale*, mentre $\Theta(a^n)$, con a costante maggiore di 1, è detto *esponenziale*. Un ordine esponenziale o maggiore è detto *superpolinomiale*.

Le definizioni degli ordini di grandezza valgono per funzioni qualsiasi, ma nella valutazione della complessità di algoritmi sono applicate a funzioni come la $T(n)$ che contano il numero di operazioni nel caso pessimo (o medio) eseguite su tutti i possibili dati di ingresso di dimensione n. Poiché contano operazioni, tali funzioni $T(n)$ di complessità sono a valori non negativi (e quindi i loro ordini di grandezza verificano le proprietà (1)-(6) su menzionate).

Utilizzando gli ordini di grandezza, ogni operazione elementare costa $O(1)$ tempo. Le uniche istruzioni che non danno un contributo unitario sono ovviamente quelle condizionali, di iterazione (limitata o illimitata) e le chiamate di procedure e funzioni. Per esempio, la complessità della seguente istruzione condizionale è $O(f(n) + \max\{g(n), h(n)\})$ per la regola (5):

if condizione che richiede $O(f(n))$ tempo
  then qualcosa che richiede $O(g(n))$ tempo
  else qualcosa che richiede $O(h(n))$ tempo;

Invece, la complessità del seguente brano contenente tre costrutti iterativi, i primi due in sequenza, ma il terzo annidato nel secondo, è $O(\max\{ n \cdot f(n), n \cdot m \cdot g(m)\})$ per le regole (5) e (6):

for $i \leftarrow 1$ **to** n **do**
  qualcosa che richiede $O(f(n))$ tempo;
for $j \leftarrow 1$ **to** n **do**
  for $k \leftarrow 1$ **to** m **do**
    qualcosa che richiede $O(g(m))$ tempo;

A questo punto, può sorgere una domanda legittima: poiché si trascurano le costanti, è sempre lecito preferire, tra due algoritmi, quello avente complessità di ordine più basso? Ad esempio, un algoritmo di complessità $\Theta(n^2)$ è da preferire ad un altro di complessità $\Theta(n^3)$, anche se la costante moltiplicativa nel primo è 200 e quella del secondo è 4? La risposta dipende dalla dimensione attesa dell'input. Per $n < 50$ il secondo algoritmo è più veloce del primo; pertanto se l'algoritmo deve essere eseguito sempre su dati di piccola dimensione si sceglierà quello con tempo $\Theta(n^3)$. Se invece sono previsti dati di dimensione grande, il rapporto tra i tempi di esecuzione, che è $4n^3/200n^2 = n/50$, cresce arbitrariamente con n e l'algoritmo di complessità $\Theta(n^2)$ risulta nettamente il migliore. Poiché in generale l'algoritmo deve poter essere eseguibile su dati di dimensione arbitraria, risulta sensato e conveniente scegliere sempre l'algoritmo la cui complessità è di ordine più basso possibile.

2.3 ALGORITMI EFFICIENTI E INEFFICIENTI

Per un singolo problema, è possibile progettare un gran numero di algoritmi diversi, ognuno caratterizzato dal proprio tempo di calcolo. Gli algoritmi con complessità superpolinomiale sono considerati inefficienti e non sono accettabili, a meno che non sia possibile dimostrare che il problema è inerentemente "difficile" [cfr. capitolo 18]. Una volta realizzato un algoritmo con complessità polinomiale, si cerca di migliorarne le prestazioni progettando algoritmi di complessità sempre inferiore. Per illustrare questi concetti, affrontiamo un problema molto importante, alla base della risoluzione di molti problemi che vedremo in seguito: l'ORDINAMENTO.

ORDINAMENTO (SORTING) Data una sequenza di valori $a_1, \ldots, a_n$, trovare una permutazione π di $\{1, \ldots, n\}$ tale che $a_{\pi_1} \leq a_{\pi_2} \leq \cdots \leq a_{\pi_n}$.

È possibile trarre ispirazione dalla definizione del problema, come già visto nel capitolo 1, anche se spesso l'algoritmo risultante non è molto efficiente. Supponiamo che la sequenza sia memorizzata in un vettore $A[1 \ldots n]$. Un semplice algoritmo è il seguente:

> Genera tutte le permutazioni degli n elementi dell'array e verifica, per ciascuna permutazione, se per ogni coppia di elementi adiacenti $A[i]$ e $A[i + 1]$ risulti $A[i] \leq A[i + 1]$.

Poiché verificare che un array sia ordinato richiede $\Theta(n)$ tempo (basta un ciclo **for**) e il numero di permutazioni possibili è $n!$, la complessità di questo algoritmo è $\Theta(nn!)$, che è superpolinomiale.

Un semplice algoritmo polinomiale invece è il *Selection Sort*, che è basato sulla proprietà che in una sequenza ordinata il primo elemento ha valore minimo:

> Trova il minimo elemento tra gli n iniziali e scambialo con quello che occupa la prima posizione; ripeti il procedimento sui restanti $n - 1$ elementi, poi sui rimanenti $n - 2$ e così via fino ad aver esaurito gli elementi.

Tale algoritmo è realizzato dalla procedura selectionSort(), la quale richiama una nuova versione iterativa di min() che restituisce la posizione dell'elemento minimo nella porzione di array $A[i \ldots n]$, per $i = 1, 2, \ldots, n$, invece che il suo valore:

```
selectionSort(ITEM[] A, integer n)
for integer i ← 1 to n do
    integer j ← min(A, i, n)
    A[i] ↔ A[j]

integer min(ITEM[] A, integer k, integer n)
    integer min ← k                          % Posizione del minimo parziale
    for integer h ← k + 1 to n do
        if A[h] < A[min] then min ← h        % Nuovo minimo parziale
    return min
```

La procedura selectionSort() esegue il ciclo **for** n volte. All'iterazione i viene chiamata la funzione min() che a sua volta esegue un altro **for** $n - i$ volte. Il tempo complessivo di calcolo della selectionSort() cresce, a meno di costanti, come

$$\sum_{i=1}^{n} (n-i) = n(n-1)/2 = n^2/2 - n/2$$

La complessità di selectionSort() è quindi $\Theta(n^2)$.

Vediamo ora l'algoritmo *Insertion Sort*, ispirato a quello utilizzato per ordinare una mano di ramino o scopone:

> Considera le carte una per volta consecutivamente, per esempio da sinistra verso destra: ogni volta che viene considerata una nuova carta, inseriscila nella posizione giusta rispetto alle altre carte già considerate e ordinate, traslando di una posizione verso destra tutte le carte maggiori.

```
insertionSort(ITEM[] A, integer n)
for integer i ← 2 to n do
    ITEM temp ← A[i]
    integer j ← i
    while j > 1 and A[j − 1] > temp do
        A[j] ← A[j − 1]
        j ← j − 1
    A[j] ← temp
```

La procedura insertionSort() considera, ad ogni iterazione, la "carta" $temp = A[i]$ da inserire, per $i = 2, 3, \ldots, n$. La porzione $A[1 \ldots i-1]$ del vettore è già ordinata ed è scandita a ritroso con l'indice j, traslando di una posizione a destra (da $A[j-1]$ ad $A[j]$) ogni elemento maggiore di *temp*, creando così uno spazio per esso. Al termine di ciascuna iterazione, *temp* è inserito in $A[j]$ nello spazio che si è creato.

2.5 ESEMPIO – [Insertion sort]
La Fig. 2.2 mostra l'ordinamento della sequenza di elementi 7, 4, 2, 1, 8, 3, 5 con la procedura insertionSort().

La complessità della procedura dipende dalla disposizione iniziale degli elementi. Il caso peggiore si ha quando il vettore A è ordinato alla rovescia; in tal caso, ad ogni iterazione del **for**, l'elemento *temp* deve essere sempre trasferito in prima posizione, e questo richiederà i spostamenti e la complessità della procedura è $O(n^2)$. Per sequenze già ordinate, però, la procedura insertionSort() è $\Theta(n)$, poiché la condizione "$A[j-1] > temp$" è sempre falsa.

Possiamo fare di meglio, diminuendo la complessità in tutti i casi, e non solo nel caso ottimo? La risposta è positiva. L'algoritmo *Merge Sort* è basato sull'idea seguente (Fig. 2.2):

> Dividi il vettore in due sequenze di $n/2$ elementi ciascuno, ordina ciascuna sequenza, e poi "fondi" le due metà ordinate in un'unica sequenza ordinata.

1	2	3	4	5	6	7
7	(4)	2	1	8	3	5
4	7	(2)	1	8	3	5
2	4	7	(1)	8	3	5
1	2	4	7	(8)	3	5
1	2	4	7	8	(3)	5
1	2	3	4	7	8	(5)
1	2	3	4	5	7	8

Fig. 2.2 Esecuzione della procedura insertionSort().

La procedura ricorsiva MergeSort(), da chiamare con l'istruzione MergeSort(A, 1, n), realizza tale algoritmo.

```
MergeSort(ITEM[] A, integer primo, integer ultimo)
if primo < ultimo then
    integer mezzo ← ⌊(primo + ultimo)/2⌋
    MergeSort(A, primo, mezzo)
    MergeSort(A, mezzo + 1, ultimo)
    Merge(A, primo, ultimo, mezzo)
```

Per fondere le due metà ordinate, la MergeSort() chiama la procedura Merge(), che si avvale di un vettore di appoggio B, utilizzato come parametro globale. La procedura Merge() usa tre indici i, j e k per scandire, rispettivamente, $A[primo \ldots mezzo]$, $A[mezzo + 1 \ldots ultimo]$ e $B[primo \ldots ultimo]$. Ad ogni passo, sono confrontati gli elementi $A[i]$ e $A[j]$, il minore dei due è copiato in $B[k]$, e sono incrementati k e l'indice dell'elemento minore. Il procedimento è iterato finché una delle due metà $A[primo \ldots mezzo]$ e $A[mezzo + 1 \ldots ultimo]$ è esaurita. Se la prima metà è stata esaurita (cioè $i > mezzo$), gli eventuali elementi non scanditi $A[j \ldots ultimo]$ della seconda metà si trovano già nelle posizioni che competono loro nell'ordinamento finale di $A[primo \ldots ultimo]$. Se invece la prima metà non è stata esaurita (cioè $i \leq mezzo$), gli elementi non scanditi $A[i \ldots mezzo]$ della prima metà vengono subito spostati nelle ultime posizioni $A[k \ldots ultimo]$ che competono loro nell'ordinamento finale. Infine, la porzione $B[primo \ldots k-1]$ è ricopiata in $A[primo \ldots k-1]$, ottenendo così $A[primo \ldots ultimo]$ ordinato.

```
Merge(A[], integer primo, integer ultimo, integer mezzo)
  integer i, j, k, h
  i ← primo; j ← mezzo + 1; k ← primo
  while i ≤ mezzo and j ≤ ultimo do
  |   if A[i] ≤ A[j] then
  |   |   B[k] ← A[i]
  |   |   i ← i + 1
  |   else
  |   |   B[k] ← A[j]
  |   |   j ← j + 1
  |   k ← k + 1
  j ← ultimo
  for h ← mezzo downto i do
  |   A[j] ← A[h]
  |   j ← j − 1
  for j ← primo to k − 1 do A[j] ← B[j]
```

2.6 ESEMPIO – [Partizione dei dati con Merge Sort]
Si supponga di eseguire la procedura Mergesort per i dati d'ingresso: A[1] = 33, A[2] = 21, A[3] = 7, A[4] = 48, A[5] = 28, A[6] = 13, A[7] = 65, A[8] = 17. Il partizionamento dei dati corrispondente è visualizzato nella Fig. 2.3.

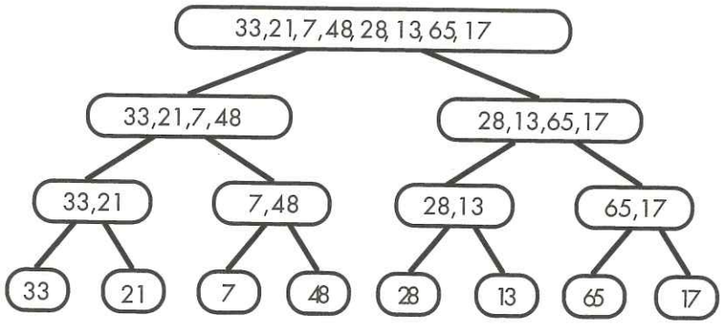

Fig. 2.3 Partizionamento dei dati con la procedura MergeSort.

2.7 ESEMPIO – [Computazione Merge Sort]
L'ordine in cui vengono ricostruiti gli insiemi è visualizzato nella Fig. 2.4. Al livello più basso, tutti i sottovettori costituiti da un unico elemento sono già ordinati. Una coppia di sottovettori viene fusa nel sottovettore superiore, indicato dalle frecce. Al termine, il vettore completo è ordinato.

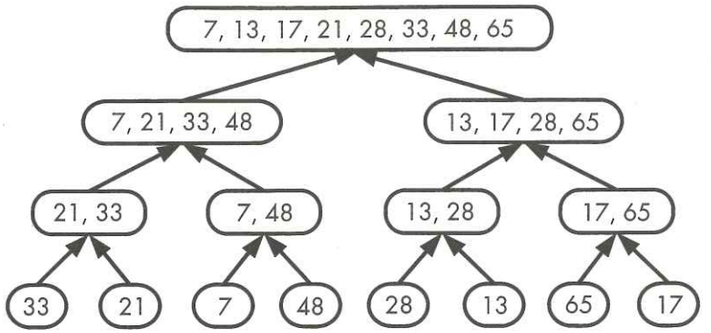

Fig. 2.4 Fusione dei dati con la procedura MergeSort.

Poiché la procedura Merge() è $O(n)$, la complessità $T(n)$ di Mergesort() è $O(n \log n)$. Infatti, assumendo per semplicità $n = 2^m$ e indicando con c e d due costanti positive, si ottiene:

$$\begin{aligned} T(n) &= c && \text{per } n = 1 \\ T(n) &\le 2T(n/2) + dn && \text{per } n \le 2 \end{aligned}$$

Sostituendo successivamente, si ricava:

$$\begin{aligned} T(n) &\le 2T(n/2) + dn \\ &\le 2(2T(n/2^2) + dn/2) + dn \\ &\le 2(2(2T(n/2^3) + dn/2^2) + dn/2) + dn \\ &\vdots \\ &\le 2^m T(n/2^m) + 2^{m-1} dn/2^{m-1} + \ldots + 2^2 dn/2^2 + 2dn/2 + dn \\ &= 2^m c + dmn \end{aligned}$$

Essendo $m = \log n$, $T(n)$ risulta essere $O(n \log n)$.

A questo punto, abbiamo progettato un discreto numero di algoritmi per l'ordinamento (ma non è finita qui: altri esempi verranno discussi in seguito). Per avere una visualizzazione tangibile della differenza di comportamento fra questi algoritmi, si osservi la Fig. 2.5. Assumendo che un'istruzione elementare sia eseguita ogni nanosecondo, come accade in un calcolatore elettronico attuale, il tempo necessario per ordinare un milione di interi per le procedure suggerite finora è il seguente:

- l'algoritmo banale basato su tutte le permutazioni richiede $\approx 2^{10^6}$ nanosecondi, un numero che ha oltre 300 000 cifre decimali. Come termine di paragone, si consideri che il numero di nanosecondi dal big-bang ad ora ha "solo" 27 cifre (il big-bang sarebbe avvenuto circa 15 miliardi di anni fa);

- selectionSort() e insertionSort() nel caso pessimo richiedono $\approx (10^6)^2$ nanosecondi, ovvero più di 16 minuti;
- Mergesort() richiede $\approx 10^6 \log_2 10^6$ nanosecondi, ovvero circa 2 centesimi di secondo;
- insertionSort() nel caso ottimo richiede 10^6 nanosecondi, ovvero un millesimo di secondo; ma questo avviene solo se il vettore è già ordinato.

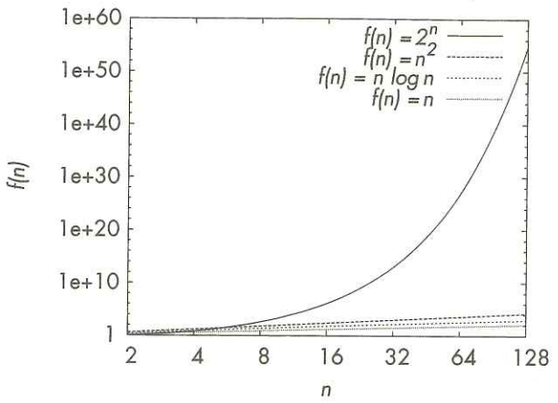

Fig. 2.5 Crescita di alcune funzioni (in scala logaritmica). Le tre rette scarsamente distanziate rappresentano le tre funzioni polinomiali n, $n \log n$, n^2 (in ordine, dal basso verso l'alto). La linea curva rappresenta la funzione esponenziale.

La lezione che ne possiamo trarre è che gli algoritmi superpolinomiali sono ovviamente da scartare e i polinomi di grado inferiore sono sempre da preferire.

La situazione non diverrebbe più rosea neanche se nei prossimi anni fossero realizzati calcolatori elettronici 100 o anche 1000 volte più veloci: per funzioni di complessità superpolinomiale la dimensione massima di un problema risolubile in un dato lasso di tempo non aumenterebbe che di pochissime unità. Per esempio, se D è la dimensione massima di un problema risolubile in un giorno con un algoritmo di complessità $\Theta(2^n)$, con un calcolatore 1000 volte più veloce la nuova dimensione massima D' non supera $D + 10$, perché da $2^{D'} = 1000 \cdot 2^D$ segue passando ai logaritmi che $D' = D + \log 1000$. In altri termini, moltiplicando la funzione di complessità per una costante, la dimensione massima risolvibile praticamente non cambia (questo spiega intuitivamente perché si possono trascurare le costanti nel valutare la complessità). La conclusione che se ne trae è univoca: algoritmi superpolinomiali sono e resteranno sempre estremamente inefficienti e non utilizzabili in pratica. Naturalmente, un algoritmo con complessità polinomiale di grado elevato, per esempio uno che richieda tempo $\Theta(n^{1000})$, pur essendo più veloce di uno con complessità $\Theta(2^n)$ per n che tende all'infinito, sarebbe in pratica inefficiente quanto uno superpolinomiale. Con l'eccezione di appositi casi costruiti artificiosamente, però, i problemi risolvibili con algoritmi di complessità polinomiale richiedono un grado basso del polinomio (raramente più di 3) per cui è sensato considerare efficienti gli algoritmi polinomiali.

2.4 COMPLESSITÀ DI PROBLEMI E ALGORITMI OTTIMI

Per risolvere un problema, si cerca di scoprire algoritmi di complessità sempre più bassa. Fino a quale punto questa ricerca può essere spinta? In altri termini, esiste una limitazione inferiore alla complessità che dipende solo dal problema in esame, che stabilisca una soglia invalicabile alla complessità di ogni algoritmo, anche non noto, che risolva il problema? La risposta è sì. Se si riesce a dimostrare che qualunque algoritmo per il problema in esame deve avere complessità $\Omega(f(n))$, allora si è stabilita una limitazione inferiore alla complessità del problema. Se invece si trova un particolare algoritmo di complessità $O(g(n))$, allora si è stabilita una limitazione superiore alla complessità del problema. Se $f(n) = g(n)$, allora l'algoritmo è detto *ottimo*, perché la sua complessità è, in ordine di grandezza, la migliore possibile. In generale, trovare limitazioni inferiori significative è molto più difficile che trovare algoritmi efficienti. Questo perché non si deve progettare in modo costruttivo un particolare algoritmo che risolva il problema, ma occorre invece fornire una prova matematica generale (cioè dimostrare un teorema) che valga per qualsiasi algoritmo che possa mai venire ideato anche in futuro. Purtroppo, sono noti pochissimi metodi generali di dimostrazione per limitazioni inferiori, per lo più poco potenti. Vediamone tre molto semplici (altri due più potenti, basati su alberi di decisione e riduzioni, saranno discussi in seguito):

(1) *dimensione dei dati*: se un problema ha in ingresso n dati e richiede di esaminarli tutti, allora una limitazione inferiore della complessità è $\Omega(n)$;

(2) *eventi contabili*: se un problema richiede che un certo evento sia ripetuto almeno n volte, allora una limitazione inferiore della complessità è $\Omega(n)$;

(3) *oracolo*: se un oracolo (o avversario), utilizzando una certa regola che l'algoritmo non conosce e che vale solo per certi dati d'ingresso, "divina" ad ogni opportunità la situazione più sfavorevole e fa lavorare l'algoritmo il più possibile, allora combattendo contro di esso si può individuare una limitazione inferiore della complessità.

2.8 ESEMPIO – [Dimensione dei dati]
Si consideri il problema di sommare due interi, composti ciascuno da n bit e memorizzati in due vettori di lunghezza n. In questo caso, è necessario considerare tutti i $2n$ bit per effettuare la somma. Questa affermazione può essere dimostrata per assurdo. Supponiamo di possedere un algoritmo "magico" in grado di calcolare correttamente la somma senza esaminare tutti i bit. Si esegua l'algoritmo su una coppia di numeri x e y. Deve esistere qualche bit in posizione i che non viene esaminato in uno dei due numeri, diciamo x. Cambiamo il valore del bit i-esimo di x e diamo all'algoritmo la nuova coppia di numeri. Poiché il bit i-esimo non viene esaminato, la risposta sarà la stessa di prima, ma avendo cambiato gli addendi, o era sbagliata prima o è sbagliata adesso. Questo è assurdo perché abbiamo assunto che l'algoritmo fosse corretto.

2.9 ESEMPIO – [Eventi contabili]
Il problema della ricerca del minimo elemento in un insieme di n interi richiede almeno $n - 1$ confronti. Infatti, il minimo può essere uno qualsiasi tra gli n elementi. Ciascuno dei rimanenti $n - 1$ elementi deve essere scartato utilizzando almeno un confronto, poiché altrimenti non si potrebbe dire che tale elemento è il minimo. Una limitazione inferiore al numero di confronti necessari per trovare il minimo è quindi $\Omega(n)$. La procedura min(), che usa $\Theta(n)$ confronti, è dunque ottima.

2.10 ESEMPIO – [Oracolo]
Si consideri il problema di "fondere" due sequenze ordinate, ciascuna di n elementi interi, in un'unica sequenza ordinata di $2n$ elementi. Siano $X[1], X[2], \ldots, X[n]$ ed $Y[1], Y[2], \ldots, Y[n]$ le due sequenze, con $X[1] < X[2] < \cdots < X[n]$ ed $Y[1] < Y[2] < \cdots < Y[n]$, e siano i $2n$ elementi tutti distinti. Si consideri la seguente regola dell'oracolo:

$$X[i] < Y[j] \quad \text{se e solo se} \quad i < j$$

Tale regola ovviamente non vale per tutti i dati di ingresso, ma solo per alcuni. Un qualsiasi algoritmo che risolva il problema per tutti i possibili dati di ingresso deve ovviamente poterlo risolvere anche nel caso speciale nel quale valga la regola dell'oracolo. Se l'oracolo risolve il problema, la soluzione *deve* essere:

$$Y[1], X[1], Y[2], X[2], \ldots, Y[n], X[n].$$

Se un qualsiasi algoritmo, nel risolvere il problema, non eseguisse un confronto, ad es. quello tra $X[1]$ e $Y[2]$, allora potrebbe produrre la seguente sequenza, che sarebbe così indistinguibile da quella prodotta dall'oracolo:

$$Y[1], Y[2], X[1], X[2], Y[3], X[3], \ldots, Y[n], X[n].$$

Per produrre la sequenza dell'oracolo, l'algoritmo deve effettuare tutti i $2n - 1$ confronti tra elementi adiacenti della sequenza stessa. Pertanto $\Omega(n)$ è una limitazione inferiore al numero di confronti necessari per risolvere il problema, e quindi la procedura Merge() è ottima.

Le precedenti tecniche di dimostrazione, ancorché apparentemente semplici, nascondono sempre insidie e sottigliezze. Per esempio, con la dimensione dei dati occorre fare molta attenzione, perché per molti problemi non è affatto necessario esaminare tutti i dati in ingresso. Inoltre, come sempre quando si dimostra un teorema, bisogna aver ben chiare quali siano le ipotesi sotto le quali si agisce. Chiariamo questa affermazione discutendo della limitazione inferiore del problema dell'ORDINAMENTO.

Per ordinare un vettore di n elementi, è ovviamente necessario esaminarli tutti. Infatti, anche se il vettore di ingresso fosse già ordinato, occorrerebbe comunque verificare l'ordinamento con un confronto tra ogni coppia di elementi adiacenti. Tale problema ha quindi complessità $\Omega(n)$.

Abbiamo quindi una limitazione inferiore $\Omega(n)$ e un algoritmo che opera in tempo $O(n \log n)$. Di fronte a questo scarto ci sono due possibilità: o esiste una limitazione inferiore più stretta, oppure esiste un algoritmo più efficiente. Nel capitolo 5, una volta introdotto il concetto di albero, saremo in grado di dimostrare che il problema dell'ordinamento è $\Omega(n \log n)$ se si utilizzano algoritmi che confrontano i valori, e quindi Mergesort() è ottimo. Potremmo allora interrompere qui la nostra ricerca del miglior algoritmo di ordinamento, perché fatto salvo per le costanti moltiplicative, lo abbiamo già trovato!

Tuttavia, supponiamo che gli elementi da ordinare, contenuti nel vettore A, siano interi compresi fra 1 e k. L'algoritmo *Counting Sort* inizialmente scandisce A e, avvalendosi di un vettore di appoggio $B[1 \ldots k]$, registra in $B[A[j]]$ il numero di occorrenze del valore $A[j]$, per $j = 1, 2, \ldots, n$. Successivamente, la procedura scandisce il vettore B, per $i = 1, 2, \ldots, k$, e scrive il valore i nel vettore A per $B[i]$ volte.

countingSort(ITEM[] A, **integer** n, **integer** k)

```
integer i, j
integer[] B ← new integer[1 ... k]
for i ← 1 to k do B[i] = 0
for i ← 1 to n do B[A[j]] ← B[A[j]] + 1
j ← 1
for i ← 1 to k do
    while B[i] > 0 do
        A[j] ← i
        j ← j + 1
        B[i] ← B[i] – 1
```

Poiché la procedura scandisce vettori di k ed n elementi, il tempo richiesto è $O(n + k)$. Se k è $O(n)$, Counting Sort ha complessità $O(n)$ e risulta più vantaggioso di Merge Sort.

È bene notare che la limitazione superiore $O(n)$ non è in contrasto con la limitazione inferiore $\Omega(n \log n)$ sul numero di confronti tra elementi da ordinare. Infatti, Counting Sort non effettua confronti tra elementi di A. Inoltre, sono cambiate le ipotesi di base. Merge Sort è in grado di operare su tutti i possibili insiemi di valori, mentre Counting Sort è in grado di operare solo in un ristretto insieme di casi. Se k è $\Omega(n \log n)$, allora Counting Sort non è più conveniente degli algoritmi di ordinamento che hanno complessità $O(n \log n)$ ed anzi può essere molto svantaggioso: per esempio, se k è $\Theta(2^n)$ allora la complessità è superpolinomiale!

2.5 TECNICHE DI ANALISI

Analizzare la complessità di un algoritmo non sempre è un compito agevole. Le maggiori difficoltà si incontrano generalmente nell'analisi di algoritmi ricorsivi, nell'analisi del caso medio, e nell'analisi "ammortizzata" di una sequenza di operazioni. Fortunatamente, sono state individuate alcune tecniche che risultano di grandissima utilità nell'analisi di algoritmi.