1. Analisi di algoritmi

Per risolvere un problema, spesso sono disponibili più algoritmi. Come stabilire quale sia il "migliore"? Un criterio ampiamente adottato consiste nel valutarne l'*efficienza*, ovvero la *quantità di risorse* richieste in funzione della *dimensione dell'input*. La risorsa più significativa è il *tempo* necessario per completare l'esecuzione, ma in molti casi si considera anche lo *spazio* utilizzato per memorizzare input, output e dati intermedi. Per poter effettuare tali valutazioni, è necessario definire con precisione cosa si intenda per *tempo di calcolo* e *dimensione dell'input*.

1.1 Tempo di calcolo

L'approccio più immediato per valutare il tempo di esecuzione di un algoritmo potrebbe sembrare quello di misurare i secondi impiegati da un calcolatore nell'eseguire una sua implementazione. Tuttavia, questo metodo è poco significativo, poiché dipende da fattori esterni al progetto dell'algoritmo: il linguaggio di programmazione utilizzato, l'efficienza del compilatore, e soprattutto le caratteristiche dell'hardware su cui il programma viene eseguito.

Per questo motivo si preferisce adottare una *misura astratta*, che descriva in modo più fedele il comportamento dell'algoritmo, indipendentemente dai dettagli di implementazione e dalla piattaforma di esecuzione. Per questo è necessario introdurre il concetto di *modello di calcolo*, ovvero una rappresentazione semplificata ma rigorosa di un calcolatore che definisca quali operazioni sono ammesse e quanto costano.

Un buon modello di calcolo deve soddisfare tre requisiti fondamentali:

- *Astrazione*: deve permettere di ignorare i dettagli irrilevanti dell'implementazione;
- *Realismo*: deve rappresentare in modo fedele il comportamento delle macchine reali;
- *Potenza matematica*: deve consentire di trarre conclusioni formali e generalizzabili sul costo computazionale.

Il modello più astratto è la *Macchina di Turing*, un dispositivo ideale dotato di un nastro infinito e una testina mobile in grado di leggere, scrivere e spostarsi. Questo modello è fondamentale nello studio della *calcolabilità*, cioè per determinare quali problemi sono risolvibili in linea di principio. Tuttavia, il suo livello di dettaglio è eccessivo e inadatto a stimare il tempo di esecuzione in modo utile per l'analisi pratica degli algoritmi.

Per questo motivo, nel contesto dell'analisi dell'efficienza, si adotta il modello della *Random Access Machine (RAM)*, che cattura in modo più realistico il funzionamento dei moderni calcolatori:

- dispone di una memoria costituita da un numero illimitato di celle, ciascuna di dimensione finita, accessibili in tempo costante (indipendentemente dalla posizione);
- ha un processore singolo che esegue un insieme limitato di *istruzioni elementari* (come somma, sottrazione, moltiplicazione, operazioni logiche, salti condizionati, ecc.);

Questo modello consente di valutare il tempo di esecuzione di un algoritmo assegnando a ogni istruzione elementare un *costo costante*, che rappresenta il tempo richiesto per eseguirla.

Definizione Il *tempo totale di esecuzione* di un algoritmo sarà quindi la somma dei costi delle singole istruzioni effettivamente eseguite.

1.2 Dimensione dell'input

Definizione La *dimensione dell'input* è un parametro numerico che misura la quantità di dati forniti a un algoritmo.

La scelta di come misurarla dipende dal problema considerato. Ad esempio:

- per un problema su vettore, la dimensione è il numero di elementi n del vettore;
- per un problema su grafi, può essere il numero di nodi, il numero di archi, o entrambi;
- per problemi su numeri interi molto grandi, si può considerare il numero di bit necessari per rappresentarli.

La dimensione dell'input è quindi una misura *astratta ma specifica* del problema, che consente di valutare la quantità di lavoro richiesta all'algoritmo.

1.3 Complessità

Definizione La *complessità* di un algoritmo è una funzione matematica che descrive l'andamento del tempo di calcolo (o dello spazio utilizzato) in relazione alla dimensione dell'input:

$$T : \text{"Dimensione dell'input"} \to \text{"Tempo di calcolo"}$$

La complessità descrive come crescono le risorse necessarie all'aumentare della dimensione del problema. In particolare, distinguiamo tra *complessità temporale*, cioè la quantità di tempo richiesta dall'algoritmo, e *complessità spaziale*, cioè la quantità di memoria necessaria durante l'esecuzione. Nel seguito, useremo il termine *complessità* in breve per riferirci alla complessità temporale.

Lo studio della complessità ci permetterà di stimare in anticipo il tempo necessario per risolvere un problema di dimensione nota; ci aiuterà a determinare la dimensione massima dell'input che può essere gestita entro limiti di tempo ragionevoli; ci darà un metro per confrontare oggettivamente algoritmi diversi per lo stesso problema.

1.4 Caso pessimo, medio, ottimo

Il costo delle operazioni è in genere valutato nel *caso pessimo*, cioè sul dato di ingresso più sfavorevole tra tutti quelli di dimensione n. In alternativa, si può considerare anche il *caso medio*, calcolando la media dei costi su tutti i possibili input di dimensione n, pesata in base alla probabilità con cui ciascun dato può verificarsi.

Talvolta si introduce anche il *caso ottimo*, che rappresenta il costo minimo dell'algoritmo su input di dimensione n. Tuttavia, questa misura è raramente utile: un buon comportamento nel caso ottimo non garantisce prestazioni accettabili negli altri casi e può dare un'illusione di efficienza non rappresentativa del comportamento complessivo dell'algoritmo.

Esempio (Tempo di calcolo di min() iterativa) Per stimare il tempo di calcolo della funzione iterativa min(), notiamo che è composta da un numero finito di istruzioni elementari. Ogni istruzione richiede un tempo costante di esecuzione, che può essere diverso da istruzione a istruzione. Indichiamo con c_h il tempo richiesto 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.

item min(**item**[] A, **int** n)

	Costo	# Volte
item $min = A[0]$	c_1	1
for $i = 1$ **to** $n-1$ **do**	c_2	n
if $A[i] < min$ **then**	c_3	$n-1$
$min = A[i]$	c_4	$n-1$
return min	c_5	1

Si osservi che l'indice i viene incrementato n volte, non $n-1$, poiché il ciclo **for** equivale a un **while** in cui la condizione $i \leq n$ è verificata anche all'uscita.

Il tempo di calcolo $T(n)$ si ottiene sommando, per ogni istruzione, il prodotto tra il suo costo unitario e il numero di esecuzioni.

$$\begin{aligned} T(n) &= c_1 + c_2 n + c_3(n-1) + c_4(n-1) + c_5 = \\ &= (c_2 + c_3 + c_4)n + (c_1 + c_5 - c_3 - c_4) \end{aligned}$$

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.

Esempio (Tempo di calcolo di binarySearch() ricorsiva) Valutare il tempo di esecuzione di un algoritmo ricorsivo come binarySearch() è un po' più complesso. Anzitutto, l'elemento cercato potrebbe trovarsi subito nella posizione mediana: questo rappresenta il *caso ottimo*. Noi invece ci concentreremo sul *caso pessimo*, in cui l'elemento non è presente nel vettore.

Come mostrato nel codice seguente, l'algoritmo esegue porzioni diverse del codice a seconda dei valori di *start* e *end*: se $start > end$, la ricorsione si arresta; altrimenti viene effettuata una nuova chiamata ricorsiva. Per tenere conto di queste differenze, nel codice di binarySearch inseriamo due colonne etichettate con "#", che indicano quante volte ciascuna riga viene eseguita nei due possibili casi.

Si indichi di nuovo con $T(n)$ il tempo richiesto per un dato d'ingresso di dimensione n. Si assuma dapprima che $n = 0$ (cioè che $start > end$). 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.

int binarySearch(**item**[] A, **item** v, **int** *start*, **int** *end*)

	Costo	# $(s > e)$	# $(s \leq e)$
if $start > end$ **then**	c_1	1	1
return -1	c_2	1	0
else			
int $m = \lfloor (start + end)/2 \rfloor$	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$(A, v, m+1, end)$	$c_7 + T(\lfloor (n-1)/2 \rfloor)$	0	$0/1$
else			
return binarySearch$(A, v, start, m-1)$	$c_7 + T(\lfloor n/2 \rfloor)$	0	$1/0$

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, entrambe hanno dimensione $\lfloor n/2 \rfloor$). Per semplificare i calcoli, assumiamo che n sia una potenza di 2, cioè $n = 2^k$ per qualche intero $k > 0$.

Nel caso pessimo, supponiamo che l'algoritmo scelga sempre il lato più grande (quello di dimensione $n/2$, essendo n pari). Questo accade, ad esempio, quando si cerca un valore maggiore del massimo contenuto nel vettore: a ogni passo si esplora la metà destra, fino a esaurire gli elementi. 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*:

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

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:

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

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.

Reality check Nella pratica, non è necessario considerare esplicitamente il costo delle singole operazioni. Ad esempio, ha poca importanza che una moltiplicazione sia, poniamo, dieci volte più lenta di un'addizione, se entrambe le operazioni richiedono comunque un tempo costante (sebbene diverso). Di fatto, tutte queste differenze possono essere assorbite in poche costanti — ad esempio a e b — quando si conclude che il tempo di calcolo $T(n)$ cresce linearmente con n, ossia $T(n) = an + b$.

Pertanto, il tempo di calcolo può essere valutato semplicemente contando il numero di operazioni elementari eseguite nel caso pessimo (o medio) su tutti gli input di dimensione n.

Un'obiezione comune alla valutazione nel caso pessimo è che risulti troppo "pessimistica", poiché il caso peggiore potrebbe verificarsi raramente. Tuttavia, questa valutazione offre un chiaro vantaggio: garantisce che l'algoritmo non richiederà mai un tempo superiore, per nessun input di dimensione n. Inoltre, in molti problemi il caso pessimo si verifica frequentemente. Ad esempio, nel controllare se un elemento appartiene a un insieme, il caso pessimo si presenta ogni volta che si verifica se un elemento non è già presente, prima di inserirlo.

Benché la valutazione nel caso medio possa apparire più realistica, spesso non è chiaro quali ipotesi di distribuzione di probabilità adottare. Di norma si assume una distribuzione uniforme (in cui tutti i casi sono equiprobabili), che per molti problemi risulta irrealistica o porta, al più, a risultati simili a quelli del caso pessimo, a meno di costanti.

Ad esempio, se si suppone che un elemento abbia uguale probabilità $1/n$ di comparire in ciascuna delle n posizioni di un vettore, allora il tempo necessario per trovarlo è proporzionale a n sia nel caso medio (circa $n/2$ confronti) sia in quello pessimo (n confronti). Inoltre, l'analisi del caso medio richiede spesso formule matematiche complesse e di difficile risoluzione.

Per tutti questi motivi, salvo poche eccezioni, la valutazione del tempo di calcolo che considereremo nel seguito è quindi quella riferita al caso pessimo.

1.5 Ordine di grandezza e classi di 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 *classe di complessità computazionale asintotica* in ordine di grandezza (o, brevemente, *classe di complessità*) di una procedura. A tal fine, si usano le notazioni "O", "Ω" e "Θ" definite come segue:

Definizione (Notazione O) Sia $g(n)$ una funzione di costo. Indichiamo con $O(g(n))$ l'insieme delle funzioni $f(n)$ tali che:

$$\exists c > 0, \exists m \geq 0 \quad \text{tale che} \quad f(n) \leq cg(n), \quad \forall n \geq m$$

Definizione (Notazione Ω) Sia $g(n)$ una funzione di costo. Indichiamo con $\Omega(g(n))$ l'insieme delle funzioni $f(n)$ tali che:

$$\exists c > 0, \exists m \geq 0 \quad \text{tale che} \quad f(n) \geq cg(n), \quad \forall n \geq m$$

Definizione (Notazione Θ) Sia $g(n)$ una funzione di costo. Indichiamo con $\Theta(g(n))$ l'insieme delle funzioni $f(n)$ tali che:

$$\exists c_1 > 0, \exists c_2 > 0, \exists m \geq 0 \quad \text{tale che} \quad c_1 g(n) \leq f(n) \leq c_2 g(n), \quad \forall n \geq m$$

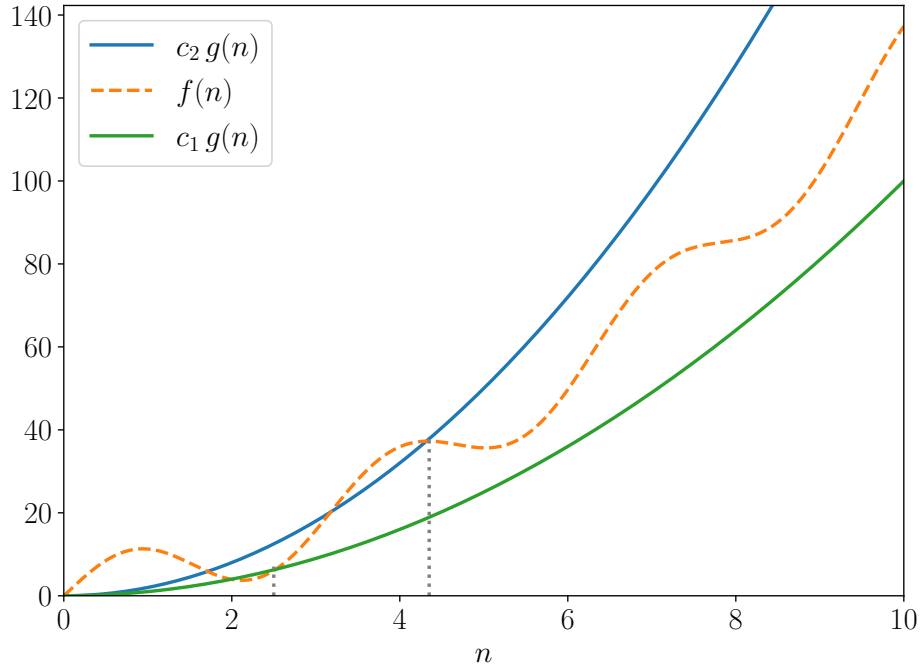

Figura 1.1: Una funzione $f(n)$ che è $\Theta(g(n))$.

Le tre notazioni appena introdotte descrivono l'andamento asintotico della funzione di costo $f(n)$ al crescere della dimensione dell'input n. La notazione $O(g(n))$ rappresenta un *limite asintotico superiore* per $f(n)$: significa che, a partire da un certo punto, $f(n)$ cresce al più quanto $g(n)$ (a meno di una costante moltiplicativa). Simmetricamente, $\Omega(g(n))$ indica un *limite asintotico inferiore*: da un certo punto in poi, $f(n)$ cresce almeno quanto $g(n)$. Infine, $\Theta(g(n))$ rappresenta una stima asintoticamente *esatta*: $f(n)$ cresce nello stesso ordine di grandezza di $g(n)$, entro fattori costanti superiori e inferiori.

Nella Figura 1.1, la funzione $f(n)$ si trova sempre compresa tra due curve, $c_1 g(n)$ e $c_2 g(n)$, che rappresentano rispettivamente un limite inferiore e uno superiore. Questo significa che $f(n)$ appartiene alla classe $\Theta(g(n))$, ovvero ha *crescita asintotica esattamente pari* a $g(n)$. In termini informali, possiamo leggere:

- "$f(n)$ è O grande di $g(n)$" $\rightarrow$ $g(n)$ è un limite superiore per $f(n)$;
- "$f(n)$ è Omega grande di $g(n)$" $\rightarrow$ $g(n)$ è un limite inferiore per $f(n)$;
- "$f(n)$ è Theta di $g(n)$" $\rightarrow$ $g(n)$ è sia limite inferiore che superiore per $f(n)$.

Esempio Consideriamo la funzione $f(n) = 10n^3 + 2n^2 + 7$. Vogliamo dimostrare che appartiene alla classe $O(n^3)$. Dobbiamo provare che $\exists c > 0, \exists m \geq 0 : f(n) \leq cn^3, \forall n \geq m$

$$
\begin{aligned}
f(n) &= 10n^3 + 2n^2 + 7 \\
&\leq 10n^3 + 2n^3 + 7 && \forall n \geq 1 \\
&\leq 10n^3 + 2n^3 + 7n^3 && \forall n \geq 1 \\
&= 19n^3 \\
&\overset{?}{\leq} cn^3
\end{aligned}
$$

che è vera per ogni $c \geq 19$ e per ogni $n \geq 1$. Quindi, scegliendo $c = 19$ e $m = 1$, otteniamo che

$f(n) \in \mathrm{O}(n^3)$.

Esempio La dimostrazione della stessa appartenenza può essere fatta anche in altri modi. Ad esempio, si può scrivere:

$$\begin{aligned} f(n) &= 10n^3 + 2n^2 + 7 \\ &\leq 10n^3 + 2n^3 + n^3 = 13n^3 \qquad \text{per ogni } n \geq \sqrt[3]{7}. \end{aligned}$$

In questo caso, è sufficiente scegliere $c = 13$ e $m = 2$.

Esempio Consideriamo $f(n) = 3n^2 + 7n$. Vogliamo verificare se $f(n) \in \Theta(n^2)$.
Limite inferiore:

$$f(n) = 3n^2 + 7n \geq 3n^2, \quad \forall n \geq 0,$$

quindi possiamo scegliere $c_1 = 3$, $m_1 = 0$.
Limite superiore:

$$f(n) = 3n^2 + 7n \leq 3n^2 + 7n^2 = 10n^2, \quad \forall n \geq 1,$$

quindi possiamo scegliere $c_2 = 10$, $m_2 = 1$.
Ponendo $m = \max(m_1, m_2) = 1$, otteniamo che $f(n) \in \Theta(n^2)$.

Esempio Supponiamo di voler verificare se $n^2 \in \mathrm{O}(n)$.
Questo richiederebbe che esistano $c > 0$ e $m \geq 0$ tali che:

$$n^2 \leq cn, \quad \forall n \geq m.$$

Dividendo entrambi i membri per n (per $n > 0$), otteniamo:

$$n \leq c,$$

ma questa disuguaglianza non può valere per tutti i n. Quindi, non esistono costanti adatte, e concludiamo che:

$$n^2 \notin \mathrm{O}(n).$$

Esempio Verifichiamo ora se $n^2 \in \mathrm{O}(n^3)$.
Dobbiamo trovare $c > 0$ e $m \geq 0$ tali che:

$$n^2 \leq cn^3, \quad \forall n \geq m.$$

Dividendo per n^2 (per $n > 0$), otteniamo $1 \leq cn$ che è vera per ogni $n \geq 1$ se scegliamo $c \geq 1$.

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).

Esempio La funzione $g(n) = 4n^2 + 4n - 1$ è $O(n^2)$, poiché $4n$ è $O(n)$ per la regola (1), $4n^2 = 4n^2$ è $O(n^2)$ 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*).

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 input 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) menzionate sopra).

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 $i = 0$ **to** $n-1$ **do**
    qualcosa che richiede $O(f(n))$ tempo;
for $j = 0$ **to** $n-1$ **do**
    for $k = 0$ **to** $m-1$ **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? Per 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.

1.6 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 $A[i] < A[i+1]$".

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(int[] A, int n)
    for i = 0 to n − 2 do
        int min = min(A, i, n − 1)
        A[i], A[min] = A[min], A[i]

    int argmin(item[] A, int start, int end)
        % Posizione del minimo parziale
        int min = start
        for j = start + 1 to end do
            if A[j] < A[min] then
                % Nuovo minimo parziale
                min = j
        return min
```

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

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

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

Vediamo ora l'algoritmo *Insertion Sort*, ispirato a quello utilizzato per ordinare una mano di scala quaranta

"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, int n)
    for i = 1 to n − 1 do
        int temp = A[i]
        int j = i
        while j > 0 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.

Esempio (Insertion sort) La Figura 1.2 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

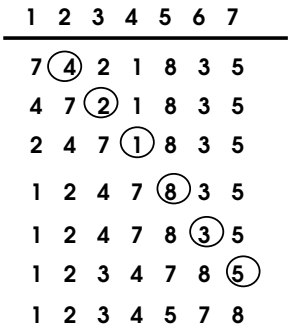

Figura 1.2: Esecuzione della procedura insertionSort()

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 (Figg. 1.3, 1.4)):

"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."

La procedura ricorsiva MergeSort(), da chiamare con l'istruzione MergeSort$(A, 0, n-1)$, realizza tale algoritmo.

MergeSort(**item** $A[\,]$, **int** *start* , **int** *end*)

if $start < end$ **then**
  int $mid = \lfloor (start + end)/2 \rfloor$
  MergeSort$(A, start, mid)$
  MergeSort$(A, mid+1, end)$
  Merge$(A, start, end, mid)$

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, $A[start \ldots mid]$, $A[mid+1 \ldots end]$ e $B[start \ldots end]$. 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 $A[start \ldots mid]$ e $A[mid+1 \ldots end]$ è esaurita. Se la prima metà è stata esaurita (cioè $i > mid$), gli eventuali elementi non scanditi $A[j \ldots end]$ della seconda metà si trovano già nelle posizioni che competono loro nell'ordinamento finale di $A[start \ldots end]$. Se invece la prima metà non è stata esaurita (cioè $i \leq mid$), gli elementi non scanditi $A[i \ldots mid]$ della prima metà vengono subito spostati nelle ultime posizioni $A[k \ldots end]$ che competono loro nell'ordinamento finale. Infine, la porzione $B[start \ldots k-1]$ è ricopiata in $A[start \ldots k-1]$, ottenendo così $A[start \ldots end]$ ordinato.

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 Figura 1.3.

```
Merge(item A[ ], int start , int end , int mid )
int i, j, k, h
i = start; j = mid + 1; k = start
while i ≤ mid and j ≤ end 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 = end
for h = mid downto i do
    A[j] = A[h]
    j = j − 1
for j = start to k − 1 do A[j] = B[j]
```

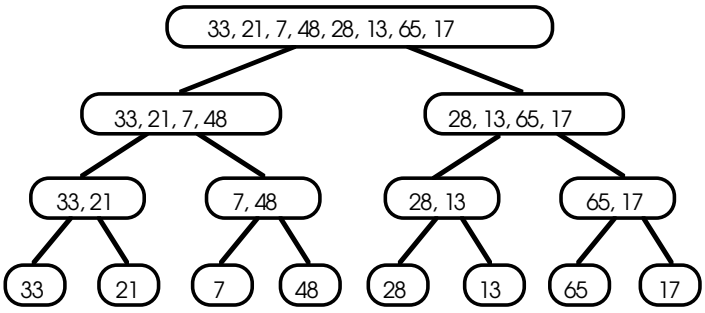

Figura 1.3: Partizionamento dei dati con la procedura MergeSort.

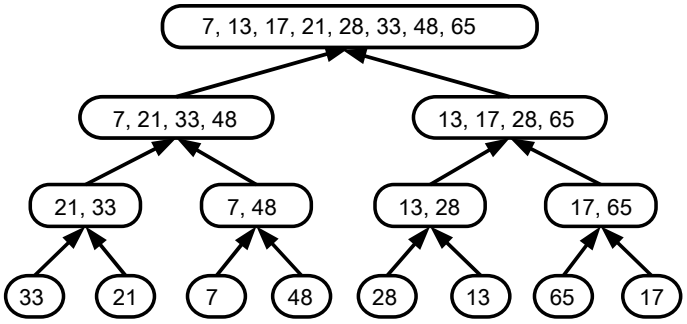

Figura 1.4: Fusione dei dati con la procedura MergeSort.

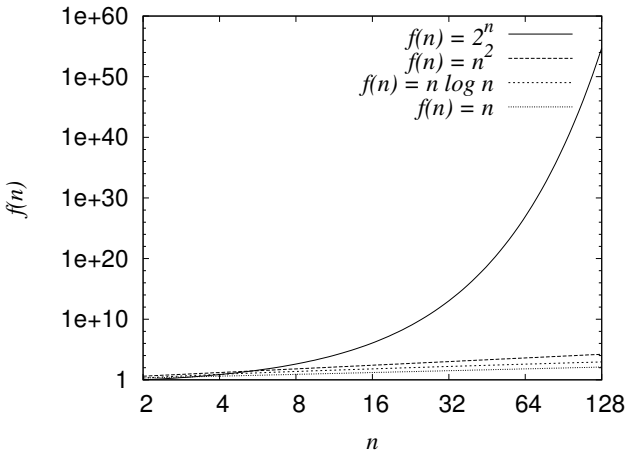

Figura 1.5: Crescita di alcune funzioni (in scala logaritmica). Le tre rette scarsamente distanziate rappresentano le tre funzioni polinomiali n, $n\log n$, n^2 (in ordine, dal basso verso l'alto). La linea curva rappresenta la funzione esponenziale.

Esempio (Computazione Merge Sort) L'ordine in cui vengono ricostruiti gli insiemi è visualizzato nella Figura 1.4. 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.

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:

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

Sostituendo successivamente, si ricava:

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

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 Figura 1.5. 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.

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.

1.7 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.

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.

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.

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 gli input, ma solo per alcuni. Un qualsiasi algoritmo che risolva il problema per tutti i possibili input 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, per esempio 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(int[] A, int n, int k)
    int[] B = new int[1...k]
    for i = 0 to k − 1 do B[i] = 0
    for j = 0 to n − 1 do  B[A[j]] = B[A[j]] + 1
    j = 0
    for i = 0 to k − 1 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!

1.8 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.

Per analizzare la complessità di un algoritmo ricorsivo, occorre impostare relazioni di ricorrenza e individuarne l'ordine di grandezza. Ci sono tre metodi diversi per risolvere tali relazioni:

(1) espandere le relazioni stesse, sostituendo successivamente nella relazione di partenza scritta per $T(n)$ quella riscritta, per esempio, per $T(n-1)$, poi quella per $T(n-2)$, e così via finché, chiarita la struttura della somma così ottenuta, non resta che una formula "chiusa" per $T(n)$ che comprende termini in n e costanti;
(2) usare la soluzione generale di alcune relazioni di ricorrenza "comuni" che sono espresse in forma parametrica, e ricavare la soluzione di $T(n)$ semplicemente sostituendo al posto dei parametri i valori costanti che appaiono nella relazione per $T(n)$;
(3) "tentare" con una soluzione $f(n)$, lasciando qualche parametro non specificato, e dimostrare, generalmente per induzione su n, che $T(n)$ è $O(f(n))$ oppure $\Omega(f(n))$.

In mancanza di metodi più generali, si può sempre applicare la tecnica (1), che in linea di principio garantisce di trovare una soluzione, ma che in pratica può richiedere di risolvere o maggiorare sommatorie molto complicate. Nel seguito verrà trattata una versione particolare della tecnica (1), detta *per livelli*, e verrano introdotte le altre due.

Per l'analisi del caso medio, purtroppo, sono note poche tecniche generali e le formule che vengono impostate per analizzare la complessità sono in genere di difficilissima risoluzione. Vedremo due esempi di analisi "trattabili" introducendo una struttura di dati chiamata *tabella hash* [cfr. Capitolo 7] ed un ulteriore algoritmo per l'ordinamento chiamato Quicksort [cfr. Capitolo 12].

Infine, per l'analisi ammortizzata di una sequenza di operazioni su una struttura di dati, vedremo alcune tecniche molto semplici, quali il metodo di *aggregazione*, quello degli *accantonamenti* e quello del *potenziale*.

1.9 Analisi per livelli

In questa tecnica di risoluzione di ricorrenze, l'espressione viene via via espansa per "livelli": al livello 0, l'espressione originale; al livello 1, abbiamo l'espressione a cui è stata applicata un'espansione; al livello 2, l'espressione a cui sono state applicate due espansioni e così via fino al caso base. Per illustrare questo procedimento, si consideri la ricorrenza seguente:

$$T(n) = \begin{cases} 1 & \text{per } n = 1 \\ 4T(n/2) + n^2 & \text{per } n = 2^h,\ h > 0 \end{cases}$$

Ad ogni livello (escluso l'ultimo) il costo da pagare effettivamente è dovuto alla componente n^2 della formula ricorsiva; varia tuttavia la dimensione di n, che viene via via ridotta.

Al livello 0, il costo della ricorrenza è pari a n^2, dovuto al termine non ricorsivo. Al livello 1, la dimensione n è stata divisa in 4 parti di dimensione $n/2$; con la ricorsione, il termine n^2 applicato a $n/2$ dà origine a $(n/2)^2 = n^2/2^2$. Al livello 2, la dimensione n è stata divisa in 16 parti di dimensione $(n/4)^2$. In generale, al livello i-esimo la dimensione è divisa in 4^i parti di costo $(n/2^i)^2 = n^2/2^{2i}$. La ricorsione termina al livello $h = \log_2 n$, ovvero quando $n/2^h = 1$. Graficamente, l'espressione può essere visualizzata nel modo seguente:

Livello	Espressione	Costo
0	n^2	$4^0 n^2$
1	$\frac{n^2}{2^2}\ \frac{n^2}{2^2}\ \frac{n^2}{2^2}\ \frac{n^2}{2^2}$	$4^1 \frac{n^2}{2^2}$
2	$\frac{n^2}{2^4}\ \frac{n^2}{2^4}\ \dots\ \frac{n^2}{2^4}\ \frac{n^2}{2^4}$	$4^2 \frac{n^2}{2^4}$
	$\dots$	
i	$\frac{n^2}{2^{2i}}\ \frac{n^2}{2^{2i}}\ \dots\ \frac{n^2}{2^{2i}}\ \frac{n^2}{2^{2i}}$	$4^i \frac{n^2}{2^{2i}}$
	$\dots$	
h	$T(1)\ \ T(1)\ \ \dots\ \ T(1)\ \ T(1)$	4^h

L'ultima riga costa $4^h = 4^{\log n} T(1) = \Theta(n^2)$. Sommando le righe precedenti si ottiene:

$$\sum_{i=0}^{\log n-1} n^2/2^{2i} \cdot 4^i = n^2 \sum_{i=0}^{\log n-1} \frac{2^{2i}}{2^{2i}} = n^2 \sum_{i=0}^{\log n-1} 1 = n^2 \log n$$

da cui si ricava:

$$T(n) = n^2 + n^2 \log n = \mathrm{O}(n^2 \log n)$$

1.10 Risoluzione di relazioni di ricorrenza "comuni"

In molti algoritmi ricorsivi accade spesso che l'algoritmo richiami se stesso un numero costante a_i di volte su dati di dimensione $n-i$, con i costante. Le chiamate ricorsive possono essere effettuate prima o dopo aver eseguito un numero polinomiale di operazioni. In tal caso, la funzione di complessità $T(n)$ dell'algoritmo può essere espressa in termini di $a_i T(n-i) + cn^\beta$. I parametri a_i e i sono costanti intere tali che $a_i \geq 1$, perché viene fatta almeno una chiamata ricorsiva, e $i \geq 1$, perché la dimensione del problema deve essere diminuita di almeno uno. Invece, i parametri c e β sono costanti reali tali che $c > 0$ e $\beta \geq 0$, in quanto è eseguita almeno un'operazione oltre alle chiamate ricorsive.

Generalizzando, date $a_1, a_2, \dots, a_h$ costanti intere non negative, con h costante positiva, si considerano relazioni di ricorrenza nelle quali $T(n)$ dipende da $\sum_{1 \leq i \leq h} a_i T(n-i) + cn^\beta$, con c e β costanti reali tali che $c > 0$ e $\beta \geq 0$. Una relazione di ricorrenza di questo tipo è detta *lineare* (perché n compare nei $T(n-i)$ con grado 1), a *coefficienti costanti* (essendo $a_1, a_2, \dots, a_h$ costanti), di *ordine costante* (poiché anche h è una costante), e con *lavoro polinomiale di suddivisione e/o ricombinazione* (in quanto cn^β è un polinomio). Si ottiene il seguente risultato.

Teorema 1.1 (Ricorrenze lineari di ordine costante) Siano $a_1, a_2, \dots, a_h$ costanti intere non negative, con h costante positiva, c e β costanti reali tali che $c > 0$ e $\beta \geq 0$, e sia $T(n)$ definita dalla relazione di ricorrenza:

$$T(n) = \begin{cases} \text{costante} & \text{per } n \leq m \geq h \\ \sum_{1 \leq i \leq h} a_i T(n-i) + cn^\beta & \text{per } n > m \end{cases}$$

Posto $a = \sum_{1 \leq i \leq h} a_i$, allora:
(1) $T(n)$ è $\mathrm{O}(n^{\beta+1})$, se $a = 1$,
(2) $T(n)$ è $\mathrm{O}(a^n n^\beta)$, se $a \geq 2$.

Dimostrazione. Si supponga dapprima che ci sia un solo coefficiente non nullo, ovvero esista j, $1 \leq j \leq h$, tale che $a_j = a$, mentre $a_i = 0$, per $1 \leq i \leq h$ ed $i \neq j$. Si assuma per semplicità che n sia uguale ad $m + kj$, per un k intero. Sostituendo successivamente si ottiene:

$$\begin{aligned}
T(n) &= aT(n-j) + cn^\beta = \\
&= a(aT(n-2j) + c(n-j)^\beta) + cn^\beta = \\
&\quad \vdots \\
&= a(a \ldots a(a(T(m) + c(m+j)^\beta) + c(m+2j)^\beta) + \cdots + \\
&\quad + c(n-j)^\beta) + cn^\beta = \\
&= a^k T(m) + \sum_{0 \leq v \leq k-1} a^v c(n - vj)^\beta \leq \\
&\leq a^k T(m) + cn^\beta \sum_{0 \leq v \leq k-1} a^v.
\end{aligned}$$

Si individuano due casi possibili:

(1) Se $a = 1$, allora:

$$\begin{aligned}
T(n) &\leq T(m) + cn^\beta [1 + 1 + \cdots + 1] = \\
&= T(m) + cn^\beta k = \\
&= T(m) + cn^\beta (n-m)/j,
\end{aligned}$$

e quindi $T(n)$ è $O(n^{\beta+1})$.

(2) Se $a \geq 2$, allora:

$$\begin{aligned}
T(n) &\leq a^k T(m) + cn^\beta [(a^k - 1)/(a-1)] \leq \\
&\leq a^k [T(m) + cn^\beta] = \\
&= a^{(n-m)/j} [T(m) + cn^\beta],
\end{aligned}$$

e quindi $T(n)$ è $O(a^n n^\beta)$.

Per completare la dimostrazione, si consideri il caso più generale in cui $\prod_{i=1}^{h} a_i \neq 0$, cioè tutti gli h coefficienti sono non nulli, con $h > 1$. In tal caso risulta necessariamente $\sum_{i=1}^{h} a_i = a \geq 2$. Pertanto, si ha:

$$\begin{aligned}
T(n) &= \sum_{1 \leq i \leq h} a_i T(n-i) + cn^\beta \leq \\
&\leq \sum_{1 \leq i \leq h} a_i T(n-1) + cn^\beta = \\
&= aT(n-1) + cn^\beta,
\end{aligned}$$

e ci si riconduce al caso precedente con $j = h = 1$ e $a_j = a \geq 2$. Infine, la dimostrazione è analoga anche per $n \neq m + kj$. ■

Esempio (Quicksort) Per la procedura QuickSort() che vedremo nel Capitolo 12 le relazioni di ricorrenza, nel caso pessimo, sono del tipo: $T(n) = T(n-1) + cn$. Poiché $a = 1$ e $\beta = 1$, si ha che $T(n)$ è $O(n^2)$.

Esempio (Torri di Hanoi) Per la procedura hanoi-ricorsiva(), che vedremo nel Capitolo 4, si ottengono relazioni del tipo: $T(n) \leq 2T(n-1)+c$. Poiché $a=2$ e $\beta=0$, ne segue che $T(n)$ è $O(2^n)$.

Per algoritmi di tipo *divide-et-impera* [cfr. Capitolo 12] in cui si partizionino i dati in modo bilanciato, il problema originario di dimensione n è diviso in a sottoproblemi di dimensione n/b ciascuno. Se si dividono i dati e/o si ricombinano i risultati in tempo polinomiale, la funzione di complessità $T(n)$ può essere espressa in termini di $aT(n/b)+cn^\beta$. I parametri a e b sono costanti intere tali che $a \geq 1$, perché è fatta almeno una chiamata, e $b \geq 2$, in quanto il problema è suddiviso in almeno due sottoproblemi, mentre c e β sono, come prima, costanti reali tali che $c > 0$ e $\beta \geq 0$. Una relazione di ricorrenza di questo tipo è ancora lineare, a coefficienti costanti, e con lavoro polinomiale di suddivisione e ricombinazione, ma non è più di ordine costante, bensì con *partizione bilanciata* (perché n è diviso per una costante b). In questo caso, si ottiene il seguente risultato.

Teorema 1.2 (Ricorrenze lineari con partizione bilanciata) Siano a e b costanti intere tali che $a \geq 1$ e $b \geq 2$, e c, d e β costanti reali tali che $c > 0$, $d \geq 0$, e $\beta \geq 0$. Sia $T(n)$ data dalla relazione di ricorrenza:

$$T(n) = \begin{cases} d & \text{per } n = 1 \\ aT(n/b)+cn^\beta & \text{per n} > 1 \end{cases}$$

Posto $\alpha = \log a / \log b$, allora:

(1) $T(n)$ è $O(n^\alpha)$, se $\alpha > \beta$,
(2) $T(n)$ è $O(n^\alpha \log n)$, se $\alpha = \beta$,
(3) $T(n)$ è $O(n^\beta)$, se $\alpha < \beta$,

Dimostrazione. Si supponga dapprima per semplicità che n sia una potenza di b, cioè $n = b^k$ per un k intero. Sostituendo successivamente si ottiene la seguente tabella:

Liv.	Dim.	Costo chiam.	N. chiamate	Costo livello
0	b^k	$cb^{k\beta}$	1	$cb^{k\beta}$
1	b^{k-1}	$cb^{(k-1)\beta}$	a	$acb^{(k-1)\beta}$
2	b^{k-2}	$cb^{(k-2)\beta}$	a^2	$a^2cb^{(k-2)\beta}$
...	...	...	...	...
i	b^{k-i}	$cb^{(k-i)\beta}$	a^i	$a^icb^{(k-i)\beta}$
...	...	...	...	...
$k-1$	b	cb^β	a^{k-1}	$a^{k-1}cb^\beta$
k	1	d	a^k	da^k

Si osservi che $a^k = a^{\log n/\log b} = 2^{\log a \log n/\log b} = n^{\log a/\log b} = n^\alpha$, e che $a = b^\alpha$, poiché $\alpha = \log a/\log b$. Ponendo $q = a/b^\beta = b^\alpha/b^\beta = b^{\alpha-\beta}$, la relazione può essere scritta come:

$$T(n) = n^\alpha d + cb^{k\beta}\left[q^{k-1}+q^{k-2}+\cdots+q^2+q+1\right].$$

Si individuano tre casi, a seconda che α sia maggiore, uguale, o minore di β, e che quindi q risulti, rispettivamente, maggiore, uguale, o minore di 1.

(1) Se $\alpha > \beta$, allora:

$$\begin{aligned} T(n) &= n^\alpha d + cb^{k\beta}[(q^k-1)/(q-1)] \leq \\ &\leq n^\alpha d + cb^{k\beta} q^k/(q-1) = \\ &= n^\alpha d + ca^k/(q-1) = \\ &= n^\alpha [d + c/(q-1)], \end{aligned}$$

e quindi $T(n)$ è $O(n^\alpha)$;

(2) Se $\alpha = \beta$, allora:

$$\begin{aligned} T(n) &= n^\alpha d + cb^{k\beta}[1+1+\cdots+1] = \\ &= n^\alpha d + cn^\beta k = \\ &= n^\alpha [d + c\log n/\log b], \end{aligned}$$

e pertanto $T(n)$ è $O(n^\alpha \log n)$;

(3) Se $\alpha < \beta$, allora:

$$\begin{aligned} T(n) &= n^\alpha d + cb^{k\beta}[(1-q^k)/(1-q)] \leq \\ &\leq n^\alpha d + cb^{k\beta}[1/(1-q)] = \\ &= n^\alpha d + cn^\beta/(1-q), \end{aligned}$$

e quindi $T(n)$ è $O(n^\beta)$.

Nel caso n non sia una potenza di b, si può considerare un problema di dimensione più grande, per es. n', tale che n' sia la più piccola potenza di b maggiore di n. Poiché $n' \leq bn$ e b è una costante, la dimensione aumenta solo di un fattore costante e la soluzione rimane valida in ordine di grandezza. Inoltre, poiché si è interessati all'analisi del caso pessimo, la soluzione della relazione di ricorrenza si applica anche al caso in cui valga il "$\leq$" anziché l'"=". Infine, la soluzione resta valida anche se $T(n)$ è uguale ad una costante per $n \leq h$, con h costante intera non necessariamente uguale ad 1, oppure se al posto di n/b si ha $\lfloor n/b \rfloor$, oppure $\lceil n/b \rceil$. ■

Esempio (Merge Sort) Per la procedura MergeSort() le relazioni di ricorrenza sono del tipo: $T(n) \leq 2T(n/2) + cn$. Poiché $a = b = 2$ ed $\alpha = \log 2/\log 2 = 1 = \beta$, si ha che $T(n)$ è $O(n \log n)$.

Esempio (Ricerca binaria) Per la procedura binarySearch() si ha: $T(n) \leq T(n/2) + c$. Poiché $a = 1$, $b = 2$, $\alpha = \log 1/\log 2 = 0 = \beta$, ne segue che $T(n)$ è $O(\log n)$.

1.11 Analisi per tentativi

Consideriamo adesso un metodo per individuare il comportamento asintotico della soluzione di una relazione di ricorrenza qualsiasi, che non sia necessariamente né lineare, né a coefficienti costanti, né di ordine costante. Tale metodo procede "per tentativi": si suppone che la soluzione sia di un certo tipo, per esempio $T(n) \leq cn^\beta$ con c e β costanti, la si sostituisce nella relazione di ricorrenza e si cerca di dimostrare per induzione su n se è effettivamente una soluzione della ricorrenza. Purtroppo, non c'è una regola generale per individuare la soluzione corretta al primo tentativo ed in questi casi occorre un po' di esperienza. Spesso si inizia tentando con soluzioni che

sono sicuramente limitazioni superiori oppure inferiori "lasche", restringendo successivamente con tentativi di limitazioni "più strette". Per dimostrare correttamente un ordine di grandezza tramite induzione, però, bisogna prestare particolare attenzione. Infatti, per non cadere miseramente in errori banali, occorre provare che la relazione vale per ogni $n \geq m$ (con m costante) ma sempre per le medesime costanti m e c.

Esempio (Tentativi) Si consideri la relazione di ricorrenza

$$T(n) = \begin{cases} 1 & \text{per } n = 1 \\ 2T(n/2) + n & \text{per } n = 2^h,\ h > 0 \end{cases}$$

Si tenti di dimostrare per induzione che $T(n) \leq cn$, per qualche costante c. Chiaramente, $T(1) = 1$ verifica la base dell'induzione per $c \geq 1$. Assumendo per ipotesi induttiva che valga $T(n/2) \leq cn/2$, si ricava che $T(n) \leq 2(cn/2) + n \leq cn + n = cn(1 + 1/c)$. Ciò non autorizza a concludere che, poiché $(1 + 1/c)$ è $O(1)$, allora $T(n)$ è $O(n)$. Infatti, $(1 + 1/c) > 1$ e quindi non si si è trovato che $T(n) \leq cn$, ma che $T(n) \leq (c + 1)n$, dove $c + 1$ è una costante più grande di c. Pertanto, non è vero che $T(n/2) \leq cn/2$ e $T(n) \leq cn$ per la stessa costante c, e quindi $T(n)$ non è $O(n)$. Si tenti adesso con $T(n) \leq cn^2$. Dopo passaggi analoghi si ottiene che $T(n) \leq 2c(n/2)^2 + n = cn^2(1/2 + 1/(cn))$. Poiché la quantità $(1/2 + 1/(cn))$ è minore di 1 per n sufficientemente grande, la relazione $T(n) \leq cn^2$ vale per ogni $n \geq m$ per le stesse costanti c ed m, e quindi $T(n)$ è $O(n^2)$. Con qualche altro tentativo si può dimostrare che $T(n)$ è $O(n \log n)$, risultato che d'altronde sapevamo già dimostrare per altra via (sia per sostituzione che applicando il teorema sulle ricorrenze lineari con partizione bilanciata).

1.12 Criterio di costo uniforme e logaritmico

La valutazione del tempo di calcolo basato sull'equipollenza delle operazioni primitive prende il nome di *criterio di costo uniforme*. Un modello alternativo è quello chiamato *criterio di costo logaritmico*, in cui il tempo richiesto da ciascuna istruzione elementare di un algoritmo dipende dal numero di bit necessari a rappresentare i suoi operandi. In generale, possiamo assumere che gli operandi siano rappresentati da un numero costante di bit, e che quindi le due misure coincidano a meno di una costante moltiplicativa. In alcuni casi, tuttavia, il criterio di costo uniforme può trarre in inganno sulla reale complessità di un algoritmo.

Esempio (Calcolo del fattoriale) Si consideri l'algoritmo iterativo per calcolare il fattoriale di n:

```
int fattoriale(int n)
    int tot = 1
    for i = 2 to n do
        tot = tot · i
    return tot
```

Dal punto di vista del criterio di costo uniforme, l'algoritmo richiede n moltiplicazioni, quindi il suo tempo di calcolo cresce linearmente con n. Questo vale, ad esempio, se il tipo **int** è rappresentato con un numero fisso di bit, come i 64 bit degli interi standard: in tal caso, è possibile calcolare il fattoriale solo fino a 20!, poiché 21! supera il massimo valore rappresentabile.

Se invece consideriamo interi a precisione arbitraria (come `BigInteger` in Java o `int`

in Python), il numero di bit necessario per rappresentare n è $d = \lceil \log n \rceil$. In questo caso, esprimendo il tempo di calcolo in funzione di d anziché di n, osserviamo che il numero di moltiplicazioni richieste è pari a $n = 2^d$, cioè un valore esponenziale rispetto a d. Torneremo su queste considerazioni nella Sezione 19.1, parlando di algoritmi pseudo-polinomiali.

1.13 Analisi ammortizzata

Quando si lavora con strutture dati dinamiche, l'interesse non è tanto il costo di una singola operazione, ma il costo complessivo di una *sequenza* di operazioni. In questo contesto entra in gioco un potente strumento teorico: l'*analisi ammortizzata*.

A differenza dell'analisi nel caso medio, che si basa su considerazioni probabilistiche, l'analisi ammortizzata è puramente deterministica. Si considera una sequenza di m operazioni e si valuta il tempo complessivo $T(m)$ richiesto per eseguirle, anche nel caso peggiore. Si ottiene così un *tempo ammortizzato* per operazione pari a $T(m)/m$, che rappresenta una media distribuita sul lungo periodo.

Questo approccio è particolarmente utile per strutture dati che mantengono uno *stato* interno e in cui operazioni costose possono essere bilanciate da molte operazioni economiche. Ad esempio, una singola operazione potrebbe richiedere tempo lineare, ma se tale evento raro è compensato da molte operazioni a costo costante, il tempo ammortizzato rimane basso.

L'analisi ammortizzata permette di modellare il comportamento *globale* delle strutture dati, ottenendo stime realistiche ed efficaci, laddove un'analisi superficiale fornirebbe solo limiti teorici troppo pessimisti.

Esempio (Contatore binario) Si consideri un contatore binario di k bit che conta "circolarmente" a partire da 0, realizzato con un vettore A di k valori interi pari a 0 o 1, dove il bit meno significativo è memorizzato in $A[0]$ e il bit più significativo è memorizzato in $A[k-1]$. Si consideri un'operazione di incremento add() del vettore realizzata secondo il ben noto schema di addizione di 1 (modulo 2^k) con propagazione del riporto. Qual è il costo di n operazioni di incremento sul contatore?

```
add(int[] A, int k)
    int i = 0
    while i < k and A[i] == 1 do
        A[i] = 0
        i = i + 1
    if i < k then A[i] = 1
```

Una risposta superficiale si basa sull'osservazione che un'operazione di incremento costa $O(k)$ nel caso pessimo (se tutti i bit sono pari a 1, sommare 1 li porta tutti a 0). Quindi n operazioni vengono eseguite in $O(kn)$ tempo, e il costo medio è $O(k)$.

Tuttavia, osserviamo cosa succede al contatore per $k = 8$ e $n = 16$:

n	A	Costo	n	A	Costo
0	00000000	-			
1	0000000<u>1</u>	1	9	0000100<u>1</u>	1
2	000000<u>10</u>	2	10	000010<u>10</u>	2
3	0000001<u>1</u>	1	11	0000101<u>1</u>	1
4	00000<u>100</u>	3	12	00001<u>100</u>	3
5	0000010<u>1</u>	1	13	0000110<u>1</u>	1
6	000001<u>10</u>	2	14	000011<u>10</u>	2
7	0000011<u>1</u>	1	15	0000111<u>1</u>	1
8	0000<u>1000</u>	4	16	000<u>10000</u>	5

I bit che sono cambiati ad ogni chiamata di add() sono sottolineati. Come si può vedere, il numero totale di bit cambiati è sicuramente più basso di nk. Cerchiamo quindi di calcolare esattamente quanti sono, osservando che il bit $A[0]$ cambia ad ogni incremento; il bit $A[1]$ cambia ogni 2 incrementi; il bit $A[2]$ ogni 4, e così via. In altre parole, il bit i-esimo cambia ogni 2^i incrementi; il numero totale di cambiamenti è quindi:

$$\sum_{i=0}^{k-1} \lfloor \frac{n}{2^i} \rfloor \leq \sum_{i=0}^{k-1} \frac{n}{2^i} = n \sum_{i=0}^{k-1} \frac{1}{2^i} \leq n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$$

Pertanto, per n operazioni, il numero di bit cambiati è limitato superiormente da $2n$; il costo ammortizzato di ogni operazione è quindi $O(1)$. Questo approccio all'analisi ammortizzata viene detto *metodo dell'aggregazione*, in quanto si calcola il costo aggregato di tutte le n operazioni, per poi dividere per n.

Esistono approcci alternativi al metodo dell'aggregazione. Nel *metodo degli accantonamenti*, ad ogni operazione i si assegna un costo ammortizzato a_i che può essere superiore o inferiore al costo effettivo c_i. In questo modo, le operazioni meno costose vengono caricate di un costo aggiuntivo detto *credito*; i crediti accumulati vengono utilizzati per pagare le operazioni più costose. Lo scopo è quello di dimostrare che la somma dei costi effettivi per n operazioni è sempre maggiorata dalla somma dei costi ammortizzati, che quindi costituisce una limitazione superiore alla somma:

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} a_i$$

Riconsideriamo quindi il nostro contatore. Il costo effettivo di un'operazione di inserimento è $d+1$, dove d è il numero di bit a uno da cambiare (che devono essere sostituiti con zeri) e 1 è il costo per cambiare un bit da zero ad uno (tranne nel caso di overflow, quando tutti i bit sono a uno). L'idea è porre il costo ammortizzato di un incremento pari a $2 = 1+1$; 1 per cambiare un bit da zero ad uno, ed uno per il futuro cambio da uno a zero. In questo modo, il credito disponibile ($\sum_{i=1}^{n} a_i - \sum_{i=1}^{n} c_i$) è esattamente pari al numero di bit a uno presenti in tutto il vettore, che è maggiore o uguale a 0, e quindi il costo effettivo di n operazioni è sicuramente limitato superiormente da $2n$.

L'ultima tecnica usata nell'analisi ammortizzata è il *metodo del potenziale* perché impiega una funzione potenziale, che in un certo senso mantiene l'"energia potenziale" della struttura di dati.

Formalmente, una funzione potenziale Φ associa ciascuna configurazione della struttura di dati ad un numero reale. Sia dunque Φ_i il valore della funzione potenziale dopo che è stata eseguita l'i-esima operazione, per $i = 1, 2, \ldots, n$.

Il costo ammortizzato a_i dell'i-esima operazione è il costo effettivo dell'operazione c_i più l'incremento di potenziale dovuto all'operazione:

$$a_i = c_i + \Phi_i - \Phi_{i-1}.$$

Il costo ammortizzato della sequenza di n operazioni è pertanto:

$$\sum_{i=1}^{n} a_i = \sum_{i=1}^{n} (c_i + \Phi_i - \Phi_{i-1}) = \sum_{i=1}^{n} c_i + \Phi_n - \Phi_0,$$

dove Φ_n e Φ_0 sono il potenziale della configurazione finale ed iniziale, rispettivamente. Di solito, è opportuno che $\Phi_0 = 0$. Se $\Phi_i - \Phi_{i-1} > 0$, allora il costo ammortizzato a_i dell'i-esima operazione è più del costo effettivo c_i, e l'eccedenza va ad accrescere il potenziale. Se invece $\Phi_i - \Phi_{i-1} < 0$, il costo ammortizzato è meno di quello effettivo ed il potenziale accumulato in precedenza viene usato per compensare il debito. Se $\Phi_n - \Phi_0 \geq 0$, allora il costo ammortizzato della sequenza di operazioni rappresenta una limitazione superiore al costo effettivo della sequenza. Si osservi che, se $\Phi_i - \Phi_0 \geq 0$ per ogni i, allora si ha la garanzia di pagare sempre in anticipo il costo di ciascuna operazione.

Nel caso del nostro contatore, la funzione potenziale Φ è definita come il numero di bit a uno presenti nel vettore.

Reality check Il numero di marzo 2006 della rivista *Nature* si intitolava *2020 – Future of Computing*. L'intero numero è una lettura importante per uno studente di Informatica; ma uno degli articoli, *Science in an exponential world* di Alexander Szalay e Jim Gray è particolarmente interessante per chi ha appena iniziato a valutare l'efficienza degli algoritmi.

L'articolo si apre con una breve introduzione storica al lavoro dello scienziato naturalista; fino a non molto tempo fa, le osservazioni scientifiche venivano registrate nei "registri di laboratorio", su carta. Oggi lo scenario è cambiato: la scienza moderna si basa sugli elaboratori, e la quantità di dati che è possibile ottenere e registrare cresce in maniera esponenziale. Già oggi *dataset* misurabili in terabyte (mille gigabyte) sono comuni; ma già si vedono all'orizzonte *dataset* misurabili in petabyte (un milione di gigabyte). Ecco alcuni esempi:

- GenBank è un database aperto che ospita sequenze di DNA (per una breve introduzione ad un algoritmo di bio-informatica si veda il Capitolo 13). Al momento in cui scriviamo, ospita il DNA di oltre 100 000 organismi, per un totale di 110 miliardi di basi;
- il *Large Synoptic Survey Telescope*, che verrà completato nel 2015, produrrà 1.28 petabyte di dati all'anno;
- il *Large Hadron Collider* del Cern, che ha iniziato a funzionare alla fine del 2009, produrrà 15 Petabyte di dati all'anno quando la sua operatività sarà completa.

Grandi sfide si aprono quindi per lo studente di Informatica che voglia abbracciare una doppia carriera, nell'Informatica e nelle Scienze naturali. È facile capire che per simili quantità di dati non c'è spazio per algoritmi esponenziali e anche molti degli algoritmi polinomiali più complessi dovranno essere evitati.

1.14 Esercizi

Esercizio 1.1. Si ordinino le seguenti funzioni per ordine di grandezza crescente: $\log^2 n$, 3^{n-2}, π^n, $n^5 - 5n^2$, $n^4 - 7n^3$, $n^{\log n}$, $\log^n n$, $n/\log n$, $n^{1/2}$, 19.

Esercizio 1.2. Per dimostrare che l'ordine di grandezza di una funzione $g(n)$ è $O(f(n))$ si può calcolare $\lim_{n\to\infty} g(n)/f(n)$. Se tale limite esiste ed è una costante c (reale), allora $g(n)$ è $O(f(n))$, mentre se il limite è ∞ allora $g(n)$ cresce più velocemente di $f(n)$. Si dimostri che $n \log n$ è $O(n^a)$ per ogni $a > 1$.

Esercizio 1.3. Si mostri che, per stabilire se una funzione $g(n)$ è $\Omega(f(n))$, si può calcolare $\lim_{n\to\infty} g(n)/f(n)$ ed affermare che se il limite esiste ed è una costante $c > 0$ oppure ∞, allora $g(n)$ è $\Omega(f(n))$. Come si può inoltre stabilire se $g(n)$ è $\Theta(f(n))$?

Esercizio 1.4. Si dimostri che tutti i polinomi crescono meno di tutti gli esponenziali, cioè che n^k è $O(b^n)$, per qualsiasi $k > 0$ e $b > 1$.

Esercizio 1.5. Si provi se le funzioni $2^{n+1}, 2^{2n}$ e 4^n sono $O(2^n)$.

Esercizio 1.6. Si dimostri che $\log n!$ è $\Theta(n \log n)$.

Esercizio 1.7. Si dimostri, per induzione, che $\sum_{i=1}^{n} i^h$ è $O(n^{h+1})$.

Esercizio 1.8. Si considerino le funzioni $f(n) = n$ e $g(n) = n^{1+\sin n}$. Si dimostri che le due funzioni non sono confrontabili in ordine di grandezza, cioè che non vale né che $f(n)$ è $O(g(n))$, né che $f(n)$ è $\Omega(g(n))$.

Esercizio 1.9. Dato un vettore di n interi, si vuole decidere se esiste un elemento che compare due volte. Determinare una limitazione inferiore e una superiore alla complessità di questo problema.

Esercizio 1.10. Sia dato un vettore di n elementi che possono assumere solo i tre valori VERDE, BIANCO e ROSSO. Si vuole ordinare il vettore in $\Theta(n)$ tempo in modo che tutti i "verdi" precedano tutti i "bianchi" e tutti i "bianchi" precedano tutti i "rossi". Le uniche operazioni ammesse sono l'esame di un elemento e lo scambio di due elementi, dati i loro indici.

Esercizio 1.11. Dato un vettore V di n interi positivi distinti, si vuole decidere se esistono due elementi che danno per somma 17. Determinare una limitazione inferiore $\Omega(n)$ e una superiore $O(n)$ alla complessità di questo problema.

Esercizio 1.12 (Analisi di algoritmi ricorsivi)**.** Per ciascuna delle seguenti funzioni si valuti l'ordine di grandezza del tempo $T(n)$.

```
int pippo(int n)
    if n ≤ 2 then
        return 1
    else if n > 99 then
        int i = ⌊n/2⌋
        return pippo(i) + n · i
    else
        return pippo(n − 2)+5
```

```
int follia(int n)
    if n > 10 then
        for i = n downto n − 3 do
            int k = ⌊n/2⌋
        return follia(⌊n/2⌋) · follia(⌊n/2⌋) + n · 5
    else if n > 20 then
        return follia(n + 1)
    else
        return n
```

Esercizio 1.13. Scrivere un algoritmo ricorsivo che riceve come parametro un vettore di n elementi tale che il suo tempo di esecuzione $T(n)$ verifichi la relazione

$$T(n) = \begin{cases} d & \text{per } n \leq k \\ 5T(n/2) + 2T(n-1) + cn^2 & \text{per } n > k \end{cases}$$

con c, d e k costanti opportune. L'ordine di grandezza di $T(n)$ è polinomiale oppure no?

Esercizio 1.14 (Analisi con cambio di variabile)**.** Si dimostri che la ricorrenza

$$T(n) = \begin{cases} d & \text{per } n \leq k \\ 2T(\sqrt{n}) + \log n & \text{per } n > k \end{cases}$$

con d e k costanti opportune, ha soluzione $O(\log n \log\log n)$.

Esercizio 1.15 (Analisi per livelli di ricorsione)**.** Si dimostri che la ricorrenza

$$T(n) = \begin{cases} d & \text{per } n \leq k \\ T(n/3) + T(2n/3) + cn & \text{per } n > k, \end{cases}$$

con d, k e c costanti opportune, ha soluzione $O(n \log n)$.

Esercizio 1.16 (Analisi per tentativi)**.** Si dimostri per tentativi che la ricorrenza

$$T(n) = \begin{cases} d & \text{per } n \leq k \\ 3T(n/4) + n \log n & \text{per } n > k \end{cases}$$

con d e k costanti opportune, ha soluzione $O(n \log n)$.

1.15 Soluzioni

Soluzione Esercizio 1.1
L'ordine è il seguente:

$$19, \log^2 n, n^{1/2}, n/\log n, n^4 - 7n^3, n^5 - 5n^2, n^{\log n}, 3^{n-2}, \pi^n, \log^n n.$$

Soluzione Esercizio 1.2

$$\lim_{n\to\infty} \frac{n\log n}{n^a} = \lim_{n\to\infty} \frac{\log n}{n^{a-1}} = \lim_{n\to\infty} \frac{n^{-1}}{(a-1)n^{a-2}} = \lim_{n\to\infty} \frac{1}{(a-1)n^{a-1}} = 0,$$

dove si è applicata la regola de L'Hôpital, derivando sia il numeratore che il denominatore, poiché sia il limite di $\log n$ che quello di n^{a-1} sono ∞. Pertanto si ha che $n\log n$ è $O(n^a)$ per ogni $a > 1$ (questo spiega perché l'ordine $O(n\log n)$ sia detto pseudolineare).

Soluzione Esercizio 1.5
Essendo $2^{n+1} = 2 \cdot 2^n$, si ha che 2^{n+1} è $O(2^n)$. Poiché invece $2^{2n} = 2^{n+n} = 2^n 2^n$, non è possibile che $2^{2n} < c2^n$ per ogni $n \geq m$, per nessuna costante c, e quindi 2^{2n} non è $O(2^n)$. Lo stesso vale per 4^n, perché $4^n = (2^2)^n = 2^{2n}$.

Soluzione Esercizio 1.6
Dato che $(n/2)^{n/2} < n! < n^n$, si ha che $\frac{n}{2}\log\frac{n}{2} < \log n! < n\log n$. Pertanto $\log n!$ è $\Theta(n\log n)$. Da questo segue anche che $\log n!$ è $O(n^a)$ per ogni $a > 1$ (si veda l'Esercizio 1.2).

Soluzione Esercizio 1.7
Procediamo per induzione su h. La base dell'induzione è facilmente verificabile per $h = 0$, perché $\sum_{i=1}^n i^0 = \sum_{i=1}^n 1 = n$. Si assuma che la limitazione sia vera per un generico h, e si consideri la sommatoria per $h+1$:

$$\sum_{i=1}^n i^{h+1} = \sum_{i=1}^n i^h i \leq \sum_{i=1}^n i^h n = n\sum_{i=1}^n i^h$$

Poiché per ipotesi induttiva $\sum_{i=1}^n i^h$ è $O(n^{h+1})$, segue che $\sum_{i=1}^n i^{h+1}$ è $O(n^{h+2})$.

Soluzione Esercizio 1.10
Una limitazione inferiore alla complessità del problema è $\Omega(n)$, poiché un qualsiasi algoritmo deve almeno esaminare tutti gli elementi. La seguente procedura ordinaBandiera() inizialmente scandisce il vettore $B[1..n]$ in avanti alla ricerca del primo elemento $B[k]$ non "verde", ed a ritroso alla ricerca dell'ultimo elemento $B[j]$ non "rosso". Successivamente, viene scandita in avanti (con l'indice i) la porzione $B[k \dots j]$ del vettore. Ogni elemento $B[i]$ "verde" incontrato è spostato indietro, scambiandolo con $B[k]$ e incrementando poi l'indice k, mentre ogni elemento $B[i]$ "rosso" è spostato in avanti scambiandolo con $B[j]$ e decrementando poi l'indice j. La scansione termina quando l'indice i scavalca j. La complessità della procedura, da richiamarsi con l'istruzione ordinaBandiera(B,n), è ovviamente $\Theta(n)$.

ordinaBandiera(**int**[] B, **int** n)

```
int k = 1
int j = n
while k ≤ n and B[k] = VERDE do k = k + 1
while j ≥ 1 and B[j] = ROSSO do j = j − 1
int i = k
while i ≤ j do
    if B[i] = ROSSO then
        B[i] ↔ B[j]
        j = j − 1
    else if B[i] = VERDE then
        B[i] ↔ B[k]
        k = k + 1
        i = i + 1
    if B[i] = BIANCO then
        i = i + 1
```

Soluzione Esercizio 1.11
Una limitazione inferiore $\Omega(n)$ si determina con la tecnica della dimensione dei dati. Infatti V potrebbe contenere una sola coppia di elementi la cui somma dà 17 e se non si esaminasse un elemento di tale coppia, che può trovarsi in una posizione qualsiasi di V, si fornirebbe la risposta sbagliata. Per ottenere una limitazione superiore $O(n)$, scandiamo tutti gli elementi di V e memorizziamo in un secondo vettore $P[1 \dots 16]$ tutti gli elementi di V minori o uguali a 16. Quindi controlliamo se in P esistono due elementi che danno per somma 17. Dato che la dimensione di P è costante, ci sono $O(1)$ coppie di elementi da controllare. Quindi il tempo richiesto è $O(n)$ ed è dovuto alla scansione di V.

Soluzione Esercizio 1.12
Per la funzione pippo() si ha:

$$T(n) = \begin{cases} d & \text{per } n \leq 99 \\ T(n/2) + c & \text{per } n > 99 \end{cases}$$

con c e d costanti. Applicando il teorema delle ricorrenze lineari con partizione bilanciata, si ha: $a = 1$, $b = 2$, $\alpha = 0$, $\beta = 0$, ed essendo $\alpha = \beta = 0$, $T(n)$ risulta essere $O(\log n)$.

Per la funzione follia() si ricava:

$$T(n) = \begin{cases} d & \text{per } n \leq 10 \\ 2T(n/2) + c & \text{per } n > 10 \end{cases}$$

con c e d costanti, poiché "**then return** follia$(n+1)$" non è mai eseguito. Per il il teorema delle ricorrenze lineari con partizione bilanciata, si ottiene: $a = 2$, $b = 2$, $\alpha = 1$, $\beta = 0$, ed essendo $\alpha > \beta$, $T(n)$ è $O(n)$.

Soluzione Esercizio 1.14
Effettuiamo il cambio di variabile $m = \log n$, ovvero $n = 2^m$. Allora $\sqrt{n} = 2^{m/2}$ e quindi si ottiene $T(2^m) = 2T(2^{m/2}) + m$. Ponendo adesso $T(2^m) = S(m)$ si ricava la relazione di ricorrenza $S(m) = 2S(m/2) + m$ la cui soluzione è $O(m \log m)$. Dato che $m = \log n$, si ottiene infine che $T(n)$ è $O(\log n \log\log n)$.

Soluzione Esercizio 1.15
Si noti che non è possibile applicare i teoremi sulle ricorrenze lineari e che il metodo di sostituzione porta a calcoli molto complicati. Allora vediamo cosa accade in ciascun livello della ricorsione sul valore complessivo della funzione. Al livello 0 della ricorsione, $T(n)$ contribuisce con cn al valore della funzione, al quale vanno aggiunti i contributi delle due chiamate ricorsive. Al livello 1 della ricorsione, $T(n/3)$ contribuisce con $cn/3$ e $T(2n/3)$ con $2cn/3$, per un totale ancora di cn, ai quali vanno aggiunti i contributi delle quattro chiamate ricorsive. Al livello 2, le quattro chiamate ricorsive $T(n/9)$, $T(2n/9)$, $T(2n/9)$ e $T(4n/9)$ contribuiscono ancora con $cn/9 + 2cn/9 + 2cn/9 + 4cn/9 = cn$. In sostanza, per ogni livello di ricorsione abbiamo un contributo di al più cn. Il numero di livelli di ricorsione è massimo seguendo le chiamate del tipo 2/3 e cresce come $O(\log_{3/2} n)$. Pertanto, dato che ad ogni livello di ricorsione si paga $O(n)$, la funzione $T(n)$ cresce come $O(n \log n)$.