```
function MIN(var B: insieme; j, k: integer): integer;
    var m: integer;
    begin
        if j = k then
            MIN := B[k]
        else begin
            m := MIN(B, j + 1, k);
            if B[j] < m then MIN := B[j] else MIN := m
        end
    end;
```

La funzione, che è richiamata come prima con l'istruzione MIN(A, 1, n), suddivide l'array in due porzioni, quella contenente soltanto il primo elemento $B[j]$, e quella contenente i rimanenti elementi $B[j+1], \ldots, B[k]$, se ci sono. Se j è uguale a k, allora vi è un solo elemento, che è anche il minimo; altrimenti, la procedura richiama se stessa sulla seconda porzione, e decide successivamente se il minimo complessivo era proprio $B[j]$ o il minimo elemento della seconda porzione. □

I.7 Tempo di calcolo

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.

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.

Esempio I.6 (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 con indicato per ogni istruzione il costo per eseguirla una volta ed il numero di volte che l'istruzione viene eseguita complessivamente. Tenendo conto

che la dimensione del problema è n, numero di elementi dell'array A, poiché MIN è richiamata con $j = 1$ e $k = n$, si ottiene:

	costo	*volte*
function MIN(**var** B: insieme; j, k: integer): integer;	c_1	1
var m, i: integer;	0	1
begin		
$m := B[j]$;	c_2	1
for $i := j + 1$ **to** k **do**	c_3	n
if $B[i] < m$ **then**	c_4	$n-1$
$m := B[i]$;	c_5	$n-1$
MIN $:= m$	c_6	1
end;		

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 $i \leq k$ deve essere verificata una volta in più per poter uscire dal ciclo quando i supera k. Il tempo di calcolo $T(n)$ di MIN si ottiene sommando su tutte le istruzioni il prodotto del costo di ciascuna istruzione per il numero di volte che tale istruzione è eseguita:

$$\begin{aligned} T(n) &= c_1 + c_2 + c_3 n + c_4(n-1) + c_5(n-1) + c_6 = \\ &= (c_3 + c_4 + c_5)n + (c_1 + c_2 + c_6 - c_4 - c_5). \end{aligned}$$

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 I.7 (Tempo di calcolo di MIN ricorsiva). Valutare il tempo della versione ricorsiva di MIN è un po' più complicato. Si indichi ancora con $T(n)$ il tempo richiesto per un dato d'ingresso di dimensione n. Si assuma dapprima che $n = 1$ (cioè che $j = k$). In tal caso la funzione esegue direttamente la condizione di chiusura:

	costo	*volte*
function MIN(**var** B: insieme; j, k: integer): integer;	c_1	1
var m: integer;	0	1
begin		
if $j = k$ **then**	c_2	1
MIN $:= B[k]$	c_3	1
else		
...		0
end;		

Se vi è un solo elemento, il tempo di calcolo è $c_1 + c_2 + c_3$ e quindi:

$$T(1) = c$$

dove c è una costante. Altrimenti, la funzione suddivide l'array in due porzioni, la prima contenente un solo elemento, e la seconda contenente i rimanenti $n-1$ elementi, e richiama se stessa sulla seconda porzione, la cui dimensione è $n-1$:

```
                                                        costo      volte
function MIN(var B: insieme; j, k: integer): integer;    c1          1
    var m: integer;                                      0           1
    begin
        if j = k then                                    c2          1
            MIN := B[k]                                  c3          0
        else begin
            m := MIN(B, j + 1, k);                       c4 + T(n-1) 1
            if B[j] < m                                  c5          1
                then MIN := B[j]                         c6          1
                else MIN := m                            c7          1
        end
    end;
```

dove la costante c_4 include sia il costo dell'incremento del parametro j che quello dell'assegnamento ad m, mentre $T(n-1)$ è il tempo di calcolo dovuto alla chiamata ricorsiva. Tenendo inoltre conto che solo uno dei due assegnamenti finali su MIN è eseguito, si ha:

$$\begin{aligned} T(n) &= T(n-1) + c_1 + c_2 + c_4 + c_5 + \max\{c_6, c_7\} = \\ &= T(n-1) + d \end{aligned}$$

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

$$\begin{aligned} T(n) &= c && \text{se } n = 1, \\ T(n) &= T(n-1) + d && \text{se } n > 1. \end{aligned}$$

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

$$\begin{aligned} T(n) &= T(n-1) + d \\ &= T(n-2) + 2d \\ &= T(n-3) + 3d \\ &\ldots \\ &= T(1) + (n-1)d \\ &= dn + (c-d). \end{aligned}$$

Quindi la versione ricorsiva di MIN richiede un tempo di calcolo che, a meno dei valori delle costanti d e $c-d$ che possono differire da a e b, cresce linearmente in n come il tempo di calcolo della versione iterativa di MIN. □

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.

Una 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. Per esempio, nel verificare se un dato elemento appartiene oppure no ad un insieme, il caso pessimo occorre ogni volta che l'elemento non è presente nell'insieme, perché è necessario esaminarne tutti gli elementi prima di accorgersene! 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 difficilissima 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 (per 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.

I.8 Ordine di grandezza e complessità

Nel valutare il tempo di calcolo $T(n)$ di una procedura, è in genere assai difficile quantificare con esattezza il numero di operazioni elementari eseguite. 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.

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].

Esempio I.8 (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 le funzioni $T(n)$ degli Esempi I.6 e I.7 sono $\Theta(n)$: le due versioni della procedura Pascal MIN hanno quindi complessità $\Theta(n)$. □

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).

Esempio I.8 (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).

Esempio I.9 (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), $\Theta(n!)$ (fattoriale), $\Theta(n^n)$ (esponenziale in base n). In generale, un ordine $\Theta(n^k)$, con k costante positiva, viene detto polinomiale, mentre $\Theta(a^n)$, con a costante maggiore di uno, 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).

[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 $\Theta(g(n)) = \Theta(f(n))$.

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 $O(max\{f(n), g(n), h(n)\})$ 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 $O(max\{nf(n), nmg(m)\})$ per le regole (5) e (6):

for i := 1 **to** n **do**
  qualcosa che richiede $O(f(n))$ tempo;
for j := 1 **to** n **do**
  for k := 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.

I.9 Algoritmi efficienti e inefficienti

Ricapitolando, per stabilire la complessità di un algoritmo, se ne studia l'ordine di grandezza O oppure Θ del tempo di calcolo $T(n)$, inteso come numero di operazioni elementari eseguite nel caso pessimo (o medio) in funzione della dimensione n dei dati di ingresso. In particolare, si dice che un algoritmo è efficiente se la sua complessità è di ordine polinomiale, mentre è inefficiente se la sua complessità è superpolinomiale. Gli algoritmi superpolinomiali sono in genere molto facili da trovare, poiché si basano sul semplice principio di "provare tutte le possibilità", ma hanno scarsa utilità pratica a causa dell'irragionevole quantità di tempo di calcolo richiesta.

Esempio I.10 (Ordinamento di un array). Si consideri il problema di ordinare un insieme di n interi, memorizzati in un array. Un banale algoritmo superpolinomiale 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** in Pascal) e il numero di permutazioni possibili è $n!$, la complessità dell'algoritmo è $\Theta(nn!)$. Un semplice algoritmo polinomiale invece è:

"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 può essere descritto con la seguente procedura Pascal, la quale richiama al suo interno una nuova versione iterativa di MIN che restituisce nel parametro *pos* la posizione dell'elemento minimo nella porzione di array $A[j..n]$, per $j = 1, 2, \ldots, n$:

```
procedure ORDINA(var A: insieme; n: integer);
    var j, pos, temp: integer;
    begin
        for j := 1 to n do begin
            temp := A[j];
            A[j] := MIN(A, j, n, pos);
            A[pos] := temp
        end
    end;

function MIN(var B: insieme; j, k: integer; var pos: integer): integer;
    var m, i: integer;
    begin
        m := B[j]; pos := j
        for i := j + 1 to k do
            if B[i] < m then begin
                m := B[i];
                pos := i
            end;
        MIN := m
    end;
```

La procedura ORDINA esegue il ciclo **for** n volte. Alla generica iterazione j, viene chiamata la funzione MIN che a sua volta esegue un altro **for** $n-j$ volte. Il tempo complessivo di calcolo della ORDINA cresce, a meno di costanti, come

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

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

Come illustrato nella Figura I.1, la crescita vertiginosa delle funzioni superpolinomiali rende di fatto inutilizzabili algoritmi aventi tale ordine di complessità. Ad esempio, si assuma che un'istruzione elementare sia eseguita ogni microsecondo, come accade in un calcolatore elettronico attuale. Allora la procedura ORDINA richiede

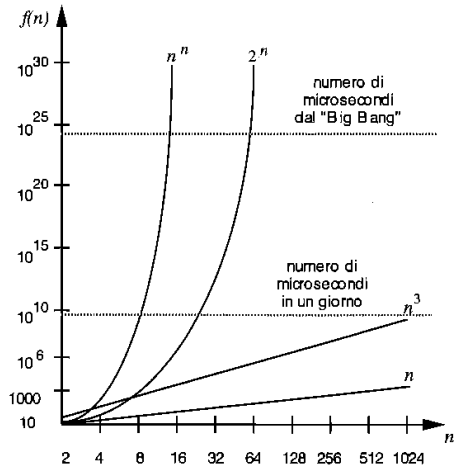

Figura I.1: Crescita di alcune funzioni (in scala logaritmica).

pressappoco un minuto per ordinare un insieme di 10.000 elementi, mentre l'algoritmo di ordinamento che generi tutte le permutazioni richiederebbe, per ordinare un insieme di soli 50 elementi, un numero di operazioni elementari avente oltre 66 cifre. Come termine di paragone si consideri che il numero di microsecondi dal "Big-Bang" ad ora ha "solo" 24 cifre (il Big-Bang sarebbe avvenuto circa 15 miliardi di anni fa). La conclusione è che gli algoritmi superpolinomiali non possono risolvere in tempo ragionevole che problemi di dimensioni estremamente piccole. Per esempio, un algoritmo di complessità $\Theta(2^n)$ richiede un numero di secoli con oltre 75 cifre per risolvere un problema di dimensione $n = 300$, quando uno $\Theta(n)$ impiega invece circa un decimo di secondo per $n = 100000$. Infine, 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.

I.10 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 I.11 (Dimensione dei dati). Il problema dell'ordinamento di un array, visto nell'Esempio I.10, richiede di esaminare tutti gli n elementi dell'array. Infatti, anche se l'array in ingresso fosse già ordinato, occorrerebbe comunque verificare l'ordinamento con un confronto tra ogni coppia di elementi adiacenti. Tale problema ha quindi complessità $\Omega(n)$. Questa limitazione non autorizza, per ora, ad affermare che la procedura ORDINA, che ha complessità $\Theta(n^2)$, non sia ottima. Infatti potrebbe essere dimostrabile una limitazione inferiore più elevata di $\Omega(n)$, così come potrebbe essere possibile trovare un algoritmo con complessità inferiore a $\Theta(n^2)$ (si dimostrerà in seguito per altra via una limitazione inferiore $\Omega(n \log n)$ e saranno individuati algoritmi ottimi di complessità $\Theta(n \log n)$). □

Esempio I.12 (Eventi contabili). Il problema della ricerca del minimo elemento in un insieme di n interi, visto nell'Esempio I.5, 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 $\Omega(n)$. Le due versioni iterativa e ricorsiva di MIN, che usano entrambe $\Theta(n)$ confronti, sono dunque ottime. □

Esempio I.13 (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, p.e. 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 (si individui per esercizio una procedura Pascal ottima che usi $\Theta(n)$ confronti). □

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. Per esempio, se un insieme è ordinato sequenzialmente, allora il minimo è il primo elemento che può essere reperito in tempo $O(1)$. Questo risultato non è in contrasto con la limitazione inferiore $\Omega(n)$, perché si assume un'ipotesi aggiuntiva (l'ordinamento dell'insieme, appunto).

I.11 Sommario

Lo scopo di questa "Introduzione" è stato quello di presentare alcune nozioni di base dell'informatica, quali algoritmi, procedure, tipi di dato, ricorsione, ordini di grandezza e complessità. Tali nozioni saranno oggetto di uno studio più approfondito nelle successive quattro parti in cui questo testo è suddiviso.

Nella prima saranno trattati metodi di specifica e realizzazione di strutture di dati, cioè particolari tipi di dato formati da aggregati di oggetti elementari (e non) che hanno rilevante interesse in informatica (ad esempio, sono strutture di dati gli array, le liste e gli insiemi). Si vedrà come la stessa struttura di dati possa essere rappresentata in vari modi che influenzano direttamente l'efficienza e la complessità delle operazioni ammesse sui componenti della struttura. Sarà anche illustrato come

l'esecuzione di procedure ricorsive possa essere gestita agevolmente utilizzando una particolare struttura, nota col nome di pila.

Nella seconda parte sarà dapprima data una classificazione dei problemi computazionali più comuni nei quali ci si imbatte nella pratica. Saranno successivamente trattate tecniche per il progetto di algoritmi efficienti per risolvere problemi appartenenti alle suddette classi, quali il divide et impera, il backtrack, la programmazione dinamica, il greedy e la ricerca locale. Sarà anche mostrato come la scelta di una struttura di dati, rispetto ad un'altra, influenzi direttamente l'efficienza di un algoritmo che la usi.

Nella terza parte si introdurranno classi di problemi "intrattabili" di rilevante interesse pratico. Tali problemi, detti NP-completi, sono tra loro computazionalmente correlati: per essi sono conosciuti soltanto algoritmi del tipo "prova tutte le possibilità" e si dubita fortemente che anche uno solo di essi sia risolubile con un algoritmo polinomiale! Si vedranno tecniche per trattare problemi di questa classe, quali la randomizzazione, l'approssimazione ed il branch-&-bound. Si accennerà infine anche all'esistenza di classi di problemi ancora più difficili, per i quali è possibile dimostrare limitazioni inferiori del tempo di calcolo che sono addirittura infinite!

Nella quarta parte si considereranno algoritmi paralleli. Al contrario degli algoritmi sequenziali, trattati nelle prime tre parti, dove è possibile eseguire una singola istruzione per volta, negli algoritmi paralleli è possibile eseguire più istruzioni contemporaneamente, permettendo così di abbassare drasticamente il tempo di risoluzione di molti problemi. Il progetto di algoritmi paralleli, purtroppo, dipende dal modello di calcolatore parallelo a disposizione e, in particolare, dal fatto che ci sia o no sincronismo tra le istruzioni eseguite contemporaneamente e che la memoria sia o no condivisa tra tutti gli esecutori. Verranno tra gli altri considerati algoritmi per modelli PRAM (sincroni con memoria condivisa), reti a grado limitato (sincroni senza memoria condivisa), concorrenti (asincroni con memoria condivisa), e distribuiti (asincroni senza memoria condivisa).

I.12 Esercizi

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

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

Esercizio I.2. Per dimostrare che una funzione $g(n)$ è $O(f(n))$ si può calcolare il limite per n che tende all'infinito del rapporto $g(n)/f(n)$. Se il 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 $a > 1$.

Il rapporto $g(n)/f(n)$ è $(n \log n)/n^a = (\log n)/n^{a-1}$. Poiché per n che tende all'infinito sia il limite di $\log n$ che quello di n^{a-1} sono ∞, si applica la regola de L'Hôpital, derivando entrambe le funzioni, ottenendo rispettivamente n^{-1} e $(a-1)n^{a-2}$. Il rapporto tra n^{-1} e $(a-1)n^{a-2}$ è $(a-1)n^{a-1}$ il cui limite per n che tende all'infinito è 0. Pertanto si ha che $n \log n$ è $O(n^a)$ per ogni $a > 1$ (questo spiega perché l'ordine $O(n \log n)$ è detto pseudolineare).

Esercizio I.3. Si mostri che, per stabilire se una funzione $g(n)$ è $\Omega(f(n))$, si può calcolare il limite per n che tende all'infinito del rapporto $g(n)/f(n)$ 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 I.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 I.5. Si provi se le funzioni 2^{n+1}, 2^{2n} e 4^n sono $O(2^n)$.

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 $4^n = (2^2)^n = 2^{2n}$.

Esercizio I.6. Si dimostri che $\log(n!)$ è $\Theta(n \log n)$.

Dato che $(n/2)^{n/2} < n! < n^n$, si ha che $n/2 \log(n/2) < \log(n!) < n \log n$. 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 I.2).

Esercizio I.7. Si dimostri, per induzione, che $\sum_{i=1}^{n} i^h$ è $O(n^{h+1})$.

Si proceda 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 $O(n^{h+2})$.

Esercizio I.8. Si considerino le funzioni $f(n) = n$ e $g(n) = n^{1+\operatorname{sen} n}$. 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 I.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 I.10. Dato un vettore di n interi positivi, 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 I.11. Dato un intero n, si vogliono stampare tutte le potenze di 5 minori o uguali di n. Si scriva una procedura Pascal di complessità ottima.

Una limitazione inferiore alla complessità del problema si determina con la tecnica degli eventi contabili, dove l'evento è la generazione di una nuova potenza di 5. Poiché bisogna elencare tutte le potenze di 5, la limitazione inferiore è data dal numero di tali potenze, che è $\Omega(\log_5 n)$.

La seguente procedura utilizza un ciclo **while** con una variabile *potenza* che, partendo da $5^0 = 1$, assume via via i valori delle potenze di 5, ciascuna ottenuta dalla precedente moltiplicando *potenza* per 5. Poiché il ciclo **while** è ripetuto $\Theta(\log_5 n)$ volte, la procedura ha complessità ottima.

```
procedure POTENZE5(n: integer);
    var potenza: integer;
    begin
        potenza := 1;
        while potenza ≤ n do begin
            write(potenza);
            potenza := potenza * 5
        end;
    end;
```

Esercizio I.12. 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.

Una limitazione inferiore alla complessità del problema è $\Omega(n)$, poiché un qualsiasi algoritmo deve almeno esaminare tutti gli elementi. La seguente procedura ORDINA 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 $B[k..j]$ 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 ORDINA(B, n), è ovviamente $\Theta(n)$.

```
type
    colore = (verde, bianco, rosso);
    bandiera = array[1..maxlung] of colore;

procedure ORDINA(var B: bandiera; n: integer);
    var i, j, k: integer; temp: colore;
    begin
        k := 1; j := n;
        while k ≤ n and B[k] = verde do k := k + 1;
        while j ≥ 0 and B[j] = rosso do j := j − 1;
        i := k;
        while i ≤ j do begin
            if B[i] = rosso then begin
                temp := B[i]; B[i] := B[j]; B[j] := temp;
                j := j − 1;
            end else
            if B[i] = verde then begin
                temp := B[k]; B[k] := B[i]; B[i] := temp;
                k := k + 1
            end;
            if B[i] = bianco then i := i + 1;
        end;
    end;
```

Parte prima

Strutture di dati