1. Divide et impera

La tecnica più importante e più usata per progettare algoritmi efficienti si basa su una massima antica dei tiranni, enunciata anche dal Machiavelli a proposito della politica dell'antico senato romano:

"Per regnare, occorre tenere disuniti i nemici e trarne vantaggio".

Applicata alla risoluzione di un problema computazionale, questa tecnica consiste nel partizionare il problema in sottoproblemi più piccoli dello stesso tipo, risolverli, e successivamente ricombinare, con poco sforzo, le soluzioni parziali per ottenere la soluzione del problema originale. Lo "schema" di una procedura ricorsiva divideEtImpera() per risolvere un problema P di dimensione n è il seguente, dove k è una costante intera prefissata:

```
divideEtImpera(P, int n)
    if n ≤ k then
        risolvi P direttamente
    else
        dividi P in h sottoproblemi P_1, ..., P_h di dimensione n_1, ..., n_h
        for i = 1 to h do divideEtImpera(P_i, n_i)
        combina i risultati di P_1, ..., P_h per ottenere quello di P
```

La tecnica *divide-et-impera* (*divide-et-conquer* in inglese) permette di provare agevolmente la correttezza dell'algoritmo usando il principio di induzione e di impostarne facilmente le relazioni ricorrenti della funzione $T(n)$ di complessità:

$$T(n) = \begin{cases} c, & \text{per } n \leq k, \\ D(n) + C(n) + \sum_{i=1}^{h} T(n_i), & \text{per } n > k, \end{cases}$$

dove c è una costante, mentre $D(n)$ e $C(n)$ sono il numero di operazioni per dividere il problema e per combinare i risultati, rispettivamente. Se i dati sono partizionati in maniera bilanciata, cioè tutti gli n_i sono all'incirca uguali, allora l'algoritmo può risultare molto efficiente.

Abbiamo già incontrato algoritmi basati sulla tecnica *divide-et-impera* durante l'introduzione all'analisi degli algoritmi [cfr. Capitolo 2]. In particolare, abbiamo introdotto la ricerca binaria e l'algoritmo Merge Sort per l'ordinamento. In questo capitolo, raffineremo le nostre conoscenze su questa tecnica studiando le problematiche relative al rapporto fra la parte *divide* e la parte *impera* e a come dividere i problemi in maniera efficiente.

1.1 Ricerca binaria e ricerca per interpolazione

La procedura binarySearch() introdotta nel Capitolo 1 è un esempio particolare di tecnica *divide-et-impera*. Il problema viene suddiviso in due sottoproblemi ($h = 2$) approssimativamente uguali ($n_2 \leq n_1 \leq n_2 + 1$), mentre i costi $C(n)$ e $D(n)$ sono costanti. La particolarità sta nel fatto che la chiamata ricorsiva avviene in solo uno dei sottoproblemi, invece che su entrambi.

Per questo motivo, alcuni autori considerano la ricerca binaria come un cattivo esempio per introdurre il *divide-et-impera*; noi riteniamo invece rilevante discuterne qui perché ci permette di enunciare una regola importante in questi casi: se non tutti i sottoproblemi devono essere analizzati, è buona norma affrontare ricorsivamente i problemi più piccoli possibili.

La complessità della ricerca può essere ulteriormente abbassata tenendo conto dei valori delle chiavi memorizzate nel vettore e della loro distribuzione di probabilità.

> **Esempio (Elenco telefonico)** Volendo ricercare il cognome Cherubini nell'elenco telefonico, non si apre l'elenco a metà, ma all'incirca ad un quarto del numero delle pagine. ■

Siano le n chiavi presenti nel vettore numeriche ed uniformemente distribuite nell'intervallo $[k_{min}, k_{max}]$. Dovendo ricercare la chiave k nel vettore $A[1 \ldots n]$, anziché provare nella posizione centrale è ragionevole tentare in quella più vicina a:

$$n \frac{(k - k_{min})}{(k_{max} - k_{min})}.$$

Infatti, $k - k_{min}$ fornisce lo scarto tra i valori della chiave da ricercare e di quella più piccola, mentre $k_{max} - k_{min}$ dà lo scarto tra i valori della chiave più grande e quella più piccola. Il loro rapporto indica di quanto il valore di k si discosta dai valori estremi. Se $k = k_{min}$, allora il rapporto è 0 e conviene cercare nella prima posizione del vettore, mentre se $k = k_{max}$, il rapporto è 1 e conviene cercare nell'ultima posizione del vettore (la ricerca binaria, ignorando il valore di k, assume tale rapporto $1/2$, e tenta in una posizione centrale). Il metodo di ricerca risultante è detto di *interpolazione* .

La procedura di ricerca binaria binarySearch() può essere trasformata in una di interpolazione, ricercaInterpolata(), semplicemente sostituendo l'istruzione

$$m = \lfloor (i + j)/2 \rfloor$$

con

$$m = i + \left\lfloor (j - i) \cdot \frac{(k - A[i])}{(A[j] - A[i])} \right\rfloor.$$

Infatti la ricerca viene effettuata in generale sulla porzione $A[i \ldots j]$ del vettore, che contiene la più piccola chiave in $A[i]$ e la più grande in $A[j]$. La formula per m si ricava da quella della ricerca binaria, che era $i + \lfloor (j - i)/2 \rfloor$, sostituendo $(k - A[i])/(A[j] - A[i])$ ad $1/2$.

Esempio (Interpolazione) Applicando la procedura ricercaInterpolata() al vettore dell'Esempio 1.2, sempre con $v = 13$, si ottiene la seguente lista di chiamate; si noti come, rispetto alla ricerca binaria, siano sufficienti 2 sole chiamate ricorsive anziché 4.

CHIAMATA	i	j	$A[i]$	$A[j]$	m	$A[m]$
1	1	12	1	42	4	8
2	5	12	13	42	5	13

■

È possibile dimostrare che, sotto l'ipotesi di uniforme distribuzione delle chiavi, la complessità di ricercaInterpolata() è $O(\log\log n)$. La dimostrazione è piuttosto complicata e non viene qui riportata. La funzione $\log\log n$ cresce molto lentamente (per esempio, vale meno di 5 per n uguale a un miliardo), ma è paragonabile a $\log n$ per n piccolo. Pertanto se ci sono poche chiavi oppure se le chiavi non sono uniformemente distribuite è più conveniente usare la ricerca binaria.

1.2 Quicksort

L'algoritmo praticamente più efficiente per ordinare gli elementi di un vettore è stato proposto da Hoare (1962) e, non a caso, è stato chiamato Quicksort (quick significa "svelto"). Questo algoritmo è basato sulla tecnica *divide-et-impera*, ma differisce dal Merge Sort nel modo in cui divide il problema e combina i risultati. La strategia consiste nel selezionare un elemento del vettore, detto *perno*, attorno al quale riarrangiare gli altri elementi. Tutti gli elementi più piccoli del perno vengono spostati in posizioni del vettore che precedono quella del perno, mentre gli elementi più grandi del perno sono spostati in posizioni che la seguono. Lo stesso algoritmo è poi riapplicato alle due porzioni di elementi minori e maggiori del perno.

Sia $A[\ldots]$ la porzione di A che l'algoritmo, ad una generica chiamata, deve ordinare. La seguente procedura QuickSort() richiama al suo interno una procedura perno() che sceglie un elemento perno x e permuta gli elementi trovando un indice j tale che $A[i] \leq x$ per $\leq i < j$, $A[j] = x$, e $A[i] \geq x$ per $j < i \leq$:

```
QuickSort(item[] A, int , int )
  if < then
      int j = perno(A, , )
      QuickSort(A, , j − 1)
      QuickSort(A, j + 1, )
```

La procedura perno() può scegliere l'elemento x in vari modi. Supponiamo per semplicità che x sia uguale ad $A[]$. Gli altri elementi del vettore vengono via via suddivisi in tre sequenze di elementi: quelli minori di x, quelli maggiori o uguali ad x, e quelli non ancora esaminati. La procedura utilizza due cursori: il cursore i punta al primo elemento y non ancora esaminato e il cursore j dà la posizione finale in cui va spostato il perno. Ad ogni iterazione, se $y \geq x$ allora y non viene spostato, mentre se $y < x$ allora y è scambiato con il primo elemento tra quelli maggiori o uguali ad x. Al termine, x è scambiato con $A[j]$.

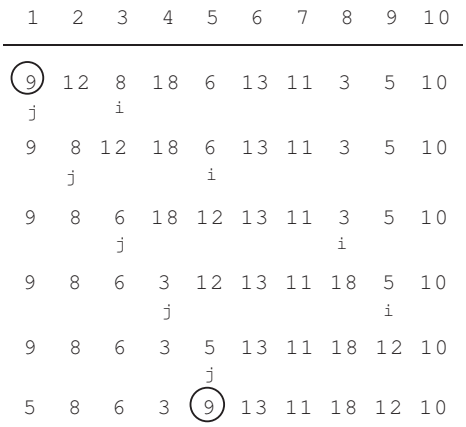

Figura 1.1: Esecuzione della procedura perno().

int perno(**item**[] A, **int** , **int**)

```
item x = A[]
int j =
for i = to do
    if A[i] < x then
        j = j + 1
        A[i] ↔ A[j]
A[] = A[j]
A[j] = x
return j
```

Esempio (perno()) La Figura 1.1 mostra l'esecuzione di perno$(A,1,10)$ per i dati d'ingresso: $A[1]=9$, $A[2]=12$, $A[3]=8$, $A[4]=18$, $A[5]=6$, $A[6]=13$, $A[7]=11$, $A[8]=3$, $A[9]=5$ e $A[10]=10$. Al termine dell'esecuzione, $j=5$. ■

Esempio (QuickSort()) La Figura 1.2 mostra la partizione dei dati indotta dall'esecuzione della procedura QuickSort() sul vettore A dell'Esempio 1.2. ■

Per valutare la complessità della procedura QuickSort(), impostiamone le relazioni ricorrenti. Siano $T(n)$ e $P(n)$ il numero di confronti tra elementi di A eseguiti, rispettivamente, dalle procedure QuickSort() e perno(). Poiché $P(n)=n-1$, e le due porzioni di A sulle quali avviene la chiamata ricorsiva hanno $j-1$ e $n-j$ elementi, si ottiene:

$$T(n)=\begin{cases}=0 & \text{per } n=0,1\\ n-1+T(j-1)+T(n-j) & \text{per } n\geq 2\end{cases}$$

Nel caso pessimo, $A[]$ è sempre l'elemento più piccolo, e quindi la porzione di A con elementi minori di x è vuota, mentre quella con elementi maggiori o uguali ne contiene $n-1$. Pertanto, $T(n)=n-1+T(n-1)$ e, sostituendo successivamente, si ha: $T(n)=\sum_{2\leq k\leq n}(k-1)$, che è $O(n^2)$. Nel caso pessimo, la procedura QuickSort() è lenta proprio come l'Insertion Sort. Paradossalmente, ciò avviene proprio quando il vettore A è già ordinato e non sarebbe necessaria alcuna elaborazione,

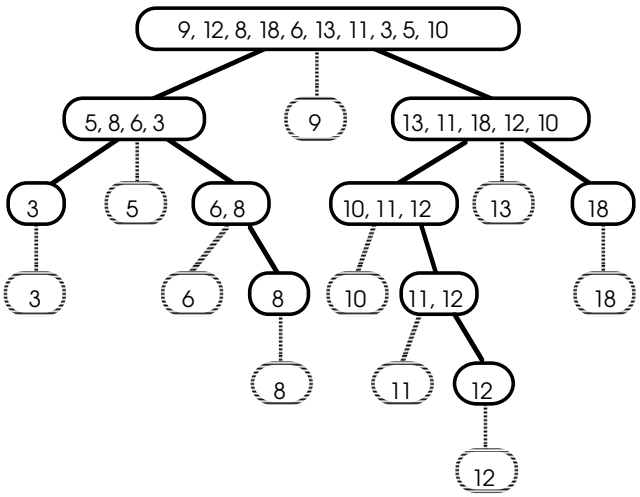

Figura 1.2: Ordinamento con la procedura QuickSort().

tranne una verifica del corretto posizionamento degli elementi che richiede solo $O(n)$ tempo (esattamente quello che fa Insertion Sort in questo caso). Questo cattivo comportamento è dovuto ad un forte sbilanciamento dei due insiemi sui quali avvengono le chiamate ricorsive!

Esempio (Caso pessimo QuickSort()) La Figura 1.3 mostra la partizione dei dati fortemente sbilanciata indotta eseguendo QuickSort() su un vettore già ordinato. ■

Il termine "Quicksort", comunque, non è immeritato per questo algoritmo, poiché la sua velocità è elevata non nel caso pessimo, ma in quello medio. Per analizzarne la complessità nel caso medio, si supponga per semplicità che tutti gli elementi di A siano distinti, e che tutte le permutazioni degli elementi $A[1\ldots n]$ siano equiprobabili. Allora, il valore j prodotto dalla procedura perno() può essere con la stessa probabilità $1/n$ in ciascuna posizione compresa tra 1 ed n, e quindi:

$$\begin{aligned} T(n) &= 0, && \text{per } n = 0,1, \\ T(n) &= n-1+\frac{1}{n}\sum_{j=1}^{n}(T(j-1)+T(n-j)) \\ &= n-1+\frac{2}{n}\sum_{j=1}^{n-1}T(j), && \text{per } n \geq 2. \end{aligned}$$

Per risolvere la relazione, la si riscriva dapprima per $n-1$, e successivamente si calcoli $T(n)-(n-1)T(n-1)/n$, ottenendo:

$$\begin{aligned} T(n-1) &= n-2+\frac{2}{n-1}\sum_{j=1}^{n-2}T(j), \\ \frac{(n-1)T(n-1)}{n} &= \frac{(n-2)(n-1)}{n}+\frac{2}{n}\sum_{j=1}^{n-2}T(j), \\ T(n)-\frac{(n-1)T(n-1)}{n} &= n-1-\frac{(n-2)(n-1)}{n}+\frac{2T(n-1)}{n}. \end{aligned}$$

Semplificando, dall'ultima equazione si ricava:

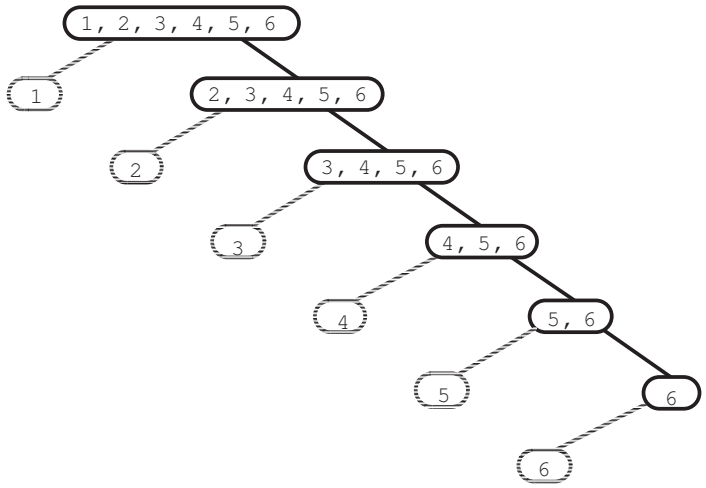

Figura 1.3: Caso pessimo della procedura QuickSort().

$$
\begin{aligned}
T(n) &= \frac{2(n-1)}{n} + \frac{(n+1)T(n-1)}{n} \leq 2 + \frac{(n+1)T(n-1)}{n} \\
&\leq 2 + \left[2 + \frac{nT(n-2)}{n-1}\right]\frac{n+1}{n} \\
&= 2 + \frac{2(n+1)}{n} + \frac{(n+1)T(n-2)}{n-1} \\
&\leq 2 + \frac{2(n+1)}{n} + \left[2 + \frac{(n-1)T(n-3)}{n-2}\right]\frac{n+1}{n-1} \\
&= 2 + \frac{2(n+1)}{n} + \frac{2(n+1)}{n-1} + \frac{(n+1)T(n-3)}{n-2} \\
&\;\;\vdots \\
&\leq 2 + 2(n+1)\left[\frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{3}\right] + \frac{(n+1)T(1)}{2} \\
&= \frac{2(n+1)}{n+1} + 2(n+1)\left[\frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{3}\right] \\
&= 2(n+1)\sum_{k=3}^{n+1}\frac{1}{k}.
\end{aligned}
$$

Poiché, come illustrato nella Figura 1.4, si ha che

$$\sum_{k=3}^{n+1}\frac{1}{k} < \int_2^{n+1}\frac{1}{x}\,\mathrm{d}x = \ln(n+1) - \ln 2,$$

$T(n)$ risulta essere $O(n\log n)$ e la procedura QuickSort() è quindi ottima nel caso medio!

Intuitivamente, il perno avrà in media un valore centrale tra $A[], A[+1], \ldots, A[]$ e la procedura sarà richiamata ricorsivamente su due sottoinsiemi all'incirca della stessa dimensione. In pratica, se si adottano alcuni "accorgimenti", la procedura QuickSort() non solo è veloce, ma è addirittura la più veloce tra tutti gli algoritmi di ordinamento noti finora per n abbastanza "grande" (per un fattore

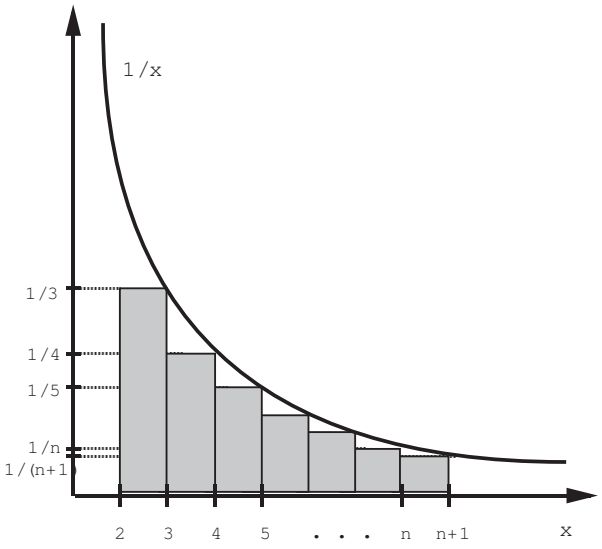

Figura 1.4: Maggiorazione di $1/3+1/4+\cdots+1/n+1/(n+1)$ (area dei rettangoli grigi) con $\int_2^{n+1}\frac{1}{x}\mathrm{d}x$ (area sottesa dalla curva $1/x$ per $2 \leq x \leq n+1$).

costante, ovviamente, non in ordine di grandezza, ché MergeSort() e heapsort() sono anch'esse asintoticamente ottime)! I principali accorgimenti, che possono essere combinati tutti insieme per esercizio ottenendo una nuova versione della procedura, sono i seguenti:

- *scelta del perno*. un buon criterio di scelta del perno x consiste nel considerare i tre elementi $A[]$, $A[(+)/2]$ e $A[]$, e sceglierne il mediano, cioè quello che non è né il massimo né il minimo dei tre (dopo averlo scambiato con $A[]$, naturalmente);
- *procedura* perno(): un metodo più veloce, ma meno intuitivo, di realizzare la procedura perno() consiste nell'utilizzare due cursori i e j per scorrere gli elementi di A, rispettivamente, da sinistra verso destra e da destra verso sinistra: i e j sono spostati finché non valgono entrambe $x > A[i]$ e $x < A[j]$, e a questo punto $A[i]$ e $A[j]$ sono scambiati tra loro; inoltre, il tempo perso per la chiamata della procedura può essere risparmiato scrivendo le istruzioni della perno() direttamente nella procedura QuickSort();
- *condizione di terminazione*: poiché il tempo perso nella gestione della ricorsione è elevato quando il numero di elementi da ordinare è piccolo, è opportuno scegliere una dimensione minima di A, per esempio 10, al di sotto della quale l'ordinamento è effettuato con una semplice procedura non ricorsiva (per esempio con shellSort(), che sarà illustrata nel Capitolo 15);
- *gestione esplicita della ricorsione*: per abbassare il tempo necessario alla gestione della ricorsione, è opportuno gestire la ricorsione esplicitamente con una pila, come illustrato nel Capitolo 4.4, avendo cura di salvare nella pila la minima informazione necessaria; per far questo, si possono salvare nella pila solo gli "estremi" di una porzione di A, ordinando direttamente l'altra porzione;
- *occupazione di memoria*: poiché nel caso pessimo la partizione di A può essere sbilanciata, la porzione di A i cui estremi sono salvati nella pila può essere sempre la più piccola e la pila richiedere così $O(n)$ spazio; per ridurre la dimensione della pila, basta salvare nella pila gli estremi della porzione più grande, e ordinare direttamente quella più piccola; in questo modo, l'occupazione di memoria della pila diviene $O(\log n)$.

1.3 Moltiplicazione di polinomi

Siano dati due polinomi $p(x) = p_{n-1}x^{n-1} + p_{n-2}x^{n-2} + \ldots + p_1x + p_0$ e $q(x) = q_{n-1}x^{n-1} + q_{n-2}x^{n-2} + \ldots + q_1x + q_0$, ciascuno avente n coefficienti (e quindi grado $n-1$). Tenendo conto che il polinomio prodotto $r(x)$ ha un numero doppio di coefficienti, è agevole scrivere un algoritmo banale di moltiplicazione che ha complessità $\Theta(n^2)$.

```
moltiplicaPolinomi(real[] p, real[] q, real[] r, int n)
for h = 0 to 2·(n−1) do r[h] = 0
for h = 0 to n−1 do
    for k = 0 to n−1 do
        r[h+k] = r[h+k] + p[h]·q[k]
```

Per abbassare la complessità, è opportuno impiegare la tecnica *divide-et-impera*, grazie alla quale si riescono ad utilizzare soltanto tre prodotti di polinomi di grado $n/2-1$ nella ricombinazione dei risultati dei sottoproblemi.

Si supponga per semplicità che il numero n di coefficienti sia una potenza di 2. Si può dividere il polinomio $p(x)$ di grado $n-1$ in due polinomi, ciascuno avente $n/2$ coefficienti:

$$p^S(x) = p_{n-1}x^{n/2-1} + p_{n-2}x^{n/2-2} + \cdots + p_{n/2+1}x + p_{n/2}$$
$$p^D(x) = p_{n/2-1}x^{n/2-1} + p_{n/2-2}x^{n/2-2} + \cdots + p_1x + p_0$$

Dividendo anche $q(x)$ nello stesso modo, si ha:

$$p(x) = x^{n/2}p^S(x) + p^D(x)$$
$$q(x) = x^{n/2}q^S(x) + q^D(x)$$

e il prodotto $r(x) = p(x)q(x)$ può essere scritto come:

$$r(x) = x^n p^S(x)q^S(x) + x^{n/2}[p^D(x)q^S(x) + q^D(x)p^S(x)] + p^D(x)q^D(x).$$

Calcolando una volta per tutte, con solo tre prodotti, i polinomi:

$$r^S(x) = p^S(x)q^S(x)$$
$$r^D(x) = p^D(x)q^D(x)$$
$$r^M(x) = [p^S(x) + p^D(x)][q^S(x) + q^D(x)]$$

il prodotto $r(x)$ si può ottenere nel modo seguente:

$$r(x) = x^n r^S(x) + x^{n/2}[r^M(x) - r^D(x) - r^S(x)] + r^D(x).$$

Questa formula permette di calcolare il prodotto tra due polinomi di dimensione n risolvendo tre sottoproblemi di dimensione $n/2$, e addizionando tra loro polinomi per "preparare" i sottoproblemi e "combinare" le soluzioni.

```
moltiplicaPolinomiD&I(real[] p, real[] q, real[] r, int n)
if n = 1 then
    r[0] = p[0]·q[0]
else
    ripartisci p in p^S e p^D
    ripartisci q in q^S e q^d
    calcola r^S, r^M ed r^D richiamando moltiplicaPolinomiD&I()
    ricava r in funzione di r^S, r^M ed r^D
```

Osservando che la somma di polinomi ed il prodotto per $x^{n/2}$ e x^n richiedono $O(n)$ operazioni e che per $n = 1$ il problema si riduce ad una singola moltiplicazione tra due coefficienti costanti, il tempo $T(n)$ richiesto dall'algoritmo verifica le seguenti relazioni di ricorrenza:

$$T(n) = \begin{cases} c & \text{per } n = 1 \\ 3T(n/2) + dn & \text{per } n = 2^h \text{ con } h > 0 \end{cases}$$

dove c e d sono costanti. Essendo $\alpha = \log 3/\log 2 \approx 1.59 > 1 = \beta$, la complessità $T(n)$ dell'algoritmo è $O(n^{1.59})$.

È possibile adattare questo algoritmo per moltiplicare due numeri interi di n bit, con la stessa complessità $O(n^{1.59})$ invece che $O(n^2)$. L'algoritmo è dovuto a Karatsuba (1962). Tanto per curiosità, l'algoritmo descritto per la moltiplicazione di polinomi non è il migliore possibile. Si possono infatti moltiplicare i due polinomi in tempo ottimo $O(n \log n)$ usando trasformate ed antitrasformate di Fourier, che a loro volta possono essere calcolate con lo stesso tempo utilizzando algoritmi basati anch'essi sulla tecnica *divide-et-impera*.

1.4 Moltiplicazione di matrici

Si consideri il problema di moltiplicare tra loro due matrici quadrate $n \times n$ $A = [a_{ij}]$ e $B = [b_{ij}]$. Per definizione, la matrice prodotto $C = AB$ ha il generico elemento $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$, per $1 \le i \le n$ ed $1 \le j \le n$. Il problema può essere risolto facilmente con la procedura moltiplicaMatrici().

```
moltiplicaMatrici(real[][] A, real[][] B, real[][] C, int n)
for i = 1 to n do
    for j = 1 to n do
        C[i, j] = 0
        for k = 1 to n do
            C[i, j] = C[i, j] + A[i, k] · B[k, j]
```

Questo banale algoritmo ha complessità $O(n^3)$. Vediamo come ridurre la complessità utilizzando la tecnica *divide-et-impera*. Assumendo per semplicità $n = 2^h$, la matrice A può essere ripartita in quattro sottomatrici A_{11}, A_{12}, A_{21} e A_{22}, ciascuna di dimensione $n/2 \times n/2$. Ripartendo in modo analogo anche B e C, poiché

$$c_{ij} = \sum_{k=1}^{n/2} a_{ik} b_{kj} + \sum_{k=n/2+1}^{n} a_{ik} b_{kj},$$

ogni sottomatrice C_{ij} della matrice prodotto può essere espressa come somma delle sottomatrici seguenti:

$$C_{ij} = A_{i1} B_{1j} + A_{i2} B_{2j} \qquad \text{per } 1 \le i \le 2 \text{ ed } 1 \le j \le 2.$$

Si definiscano adesso le sette matrici $X_1, \ldots, X_7$, ciascuna delle quali ha dimensione $n/2 \times n/2$ e può essere calcolata con un solo prodotto tra matrici:

$$\begin{aligned} X_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ X_2 &= (A_{21} + A_{22}) B_{11} \\ X_3 &= A_{11}(B_{12} - B_{22}) \\ X_4 &= A_{22}(B_{21} - B_{11}) \\ X_5 &= (A_{11} + A_{12}) B_{22} \\ X_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\ X_7 &= (A_{12} - A_{22})(B_{21} + B_{22}). \end{aligned}$$

Le matrici C_{ij} possono essere ricavate in funzione delle matrici $X_1, \ldots, X_7$ come segue:

$$C_{11} = X_1 + X_4 - X_5 + X_7$$
$$C_{12} = X_3 + X_5$$
$$C_{21} = X_2 + X_4$$
$$C_{22} = X_1 + X_3 - X_2 + X_6$$

L'algoritmo risultante, proposto da Strassen (1969), può essere riassunto informalmente con la procedura strassen().

strassen(**real**[][] A, **real**[][] B, **real**[][] C, **int** n)

if $n = 1$ **then**
  $C[1,1] = A[1,1] \cdot B[1,1]$
else
  ripartisci A in A_{11}, A_{12}, A_{21} e A_{22}
  ripartisci B in B_{11}, B_{12}, B_{21} e B_{22}
  calcola $X_1, \ldots, X_7$ richiamando strassen()
  ricava $C_{11}, C_{12}, C_{21}, C_{22}$ in funzione di $X_1, \ldots, X_7$

Poiché le somme di matrici di dimensione $n/2 \times n/2$ richiedono $O(n^2)$ operazioni, e la procedura richiama se stessa sette volte per calcolare $X_1, \ldots, X_7$, il tempo $T(n)$ richiesto dall'algoritmo verifica le seguenti relazioni di ricorrenza, dove c e d sono costanti positive:

$$T(n) = \begin{cases} d & \text{per } n = 1 \\ 7T(n/2) + cn^2 & \text{per } n = 2^h,\ h > 0 \end{cases}$$

Applicando il teorema delle ricorrenze lineari con partizione bilanciata, si ottiene $\alpha = \log 7 / \log 2 \approx 2.81 > 2 = \beta$. Pertanto, $T(n)$ risulta essere $O(n^{2.81})$.

L'algoritmo di Strassen è valido per n qualsiasi, non necessariamente per $n = 2^h$. Basta infatti bordare le matrici A e B con righe e colonne di zeri fino ad ottenere $n = 2^h$ e l'ordine di grandezza di $T(n)$ non cambia. Inoltre, il metodo può essere adattato per calcolare, con la stessa complessità, il determinante e l'inversa di una matrice. L'algoritmo è efficiente per n grande, mentre per n piccolo l'algoritmo banale di complessità $O(n^3)$ può risultare più veloce. Ciò è dovuto alle costanti "nascoste" nell'ordine di grandezza della complessità, che sono elevate per l'algoritmo di Strassen. Sono noti persino algoritmi asintoticamente più efficienti di quello di Strassen (il più veloce ha complessità $O(n^{2.376})$, mentre una limitazione inferiore alla complessità del problema è ovviamente $\Omega(n^2)$), ma tali algoritmi non sono di utilità pratica a causa delle enormi costanti moltiplicative coinvolte negli ordini di grandezza delle funzioni di complessità.

1.5 Reality check

Poiché una limitazione inferiore al problema dell'ordinamento è $\Omega(n \log n)$, MergeSort() è ottimo sempre, mentre QuickSort() è ottimo nel caso medio, ma è $O(n^2)$ nel caso pessimo. Tuttavia, fra i due algoritmi si preferisce spesso QuickSort(), perché le sue costanti moltiplicative sono più basse.

Oltre all'efficienza, esistono ulteriori parametri per valutare gli algoritmi di ordinamento. Uno di questi è la *stabilità* dell'ordinamento: supponendo di ordinare coppie (chiave, valore), e che le chiavi possano apparire più di una volta, un algoritmo di ordinamento è stabile se, date due coppie con chiave uguale (k, v_1) e (k, v_2), tali che (k, v_1) appare prima di (k, v_2) nella sequenza da ordinare, allora quest'ordine viene rispettato nella sequenza ordinata. Fra gli algoritmi discussi in questo testo, Selection Sort, Heapsort, Quicksort e Shell Sort [cfr. Capitolo 15] non sono stabili, mentre lo sono Insertion Sort, Merge Sort e Pigeonhole Sort [cfr. Esercizio 4.11].

Un altro parametro di valutazione è l'utilizzo di memoria aggiuntiva, oltre a quella necessaria per memorizzare l'input. Selection Sort, Insertion Sort, Heapsort e Shell Sort sono algoritmi iterativi e utilizzano $O(1)$ memoria aggiuntiva (solo lo spazio necessario per le variabili della procedura); Quicksort richiede in media $O(\log n)$ spazio aggiuntivo (lo spazio necessario per le chiamate ricorsive); mentre Merge Sort e Pigeonhole Sort richiedono spazio $O(n)$, anche se è possibile realizzare versioni iterative di Merge Sort. Gli algoritmi che non necessitano $O(n)$ spazio aggiuntivo sono comunemente detti *in-place*.

Comunque, algoritmi di ordinamento basati sulla tecnica Merge Sort risultano vantaggiosi qualora gli elementi da ordinare risiedano in memoria secondaria (per esempio su disco). In tal caso, il tempo per trasferire avanti e indietro dati tra la memoria principale, che è veloce ma poco capiente, e la memoria secondaria, che viceversa è capiente ma lenta, supera di gran lunga il tempo richiesto per effettuare l'ordinamento nella memoria principale. È quindi conveniente elaborare i dati in blocchi sequenziali, proprio come viene fatto dalla procedura Merge(). Pertanto, la maggior parte degli algoritmi di ordinamento su memoria secondaria si basa su una tecnica "Merge Sort polifase", distribuendo dapprima gli elementi da ordinare in molte sequenze ordinate (dette *corse*) memorizzate in memoria secondaria e poi eseguendo ripetutamente operazioni di fusioni tra corse finché non rimane che un'unica corsa.

Esistono competizioni per stabilire il più veloce algoritmo di ordinamento per grandi (enormi!) quantità di dati[1]. Per esempio, nel 2009 un programma sviluppato dall'Università di Padova è stato in grado di ordinare oltre 2 miliardi di chiavi da 100 byte, per un totale di 223 Gigabyte, in meno di un minuto, su un elaboratore "normale" (2.6 Ghz AMD Athlon, 4 GB RAM, 5 dischi da 160 GB - 7200 RPM Sata).

1.6 Esercizi

Esercizio 1.1 (Ordinamento su lista)**.** Si riscrivano le procedure MergeSort() e QuickSort() nel caso in cui la sequenza da ordinare sia contenuta in una lista realizzata con puntatori, mantenendo in entrambi i casi la stessa complessità $O(n \log n)$.

Esercizio 1.2 (Massima somma di sottovettori)**.** Progettare un algoritmo che, preso un vettore A di n interi (positivi o negativi), trovi un sottovettore (una sequenza di elementi consecutivi del vettore) la somma dei cui elementi sia massima.

Esercizio 1.3 (Inversioni)**.** Data una sequenza di n interi memorizzata in un vettore A, si ha un'inversione tra gli elementi $A[i]$ e $A[j]$ se $i < j$ e $A[i] > A[j]$. Si modifichi la procedura MergeSort() per contare il numero totale di inversioni presenti in A.

Esercizio 1.4 (Analisi Quicksort)**.** Si analizzi la complessità di QuickSort() nel caso in cui tutti gli elementi del vettore da ordinare abbiano lo stesso valore.

Esercizio 1.5 (Merge Sort iterativo)**.** Si fornisca una versione iterativa della procedura MergeSort() senza usare una pila e mantenendone la complessità di $O(n \log n)$.

Esercizio 1.6 (Ricerca in matrice ordinata)**.** Sia data una matrice $n \times n$ di interi in cui gli elementi in ogni riga ed ogni colonna sono ordinati in modo crescente. Si progetti un algoritmo di tipo *divide-et-impera* per determinare se un dato intero k è contenuto nella matrice e se ne discuta la complessità.

Esercizio 1.7 (Torneo di tennis)**.** Si progetti un algoritmo *divide-et-impera* per determinare il calendario di un torneo di tennis, cui partecipano $n = 2^k$ giocatori, in modo che ciascun giocatore incontri una ed una sola volta tutti gli altri, disputi un incontro al giorno, e la durata del torneo sia di $n-1$ giorni

[1] `http://sortbenchmark.org`

Esercizio 1.8 (Prodotto di numeri complessi)**.** Si individui un algoritmo per moltiplicare due numeri complessi usando solo tre operazioni aritmetiche di moltiplicazione anziché quattro.

Esercizio 1.9 (Conta gli zeri)**.** Un vettore V di n numeri binari contiene tutti gli zeri nelle prime posizioni, seguiti da tutti gli uni. Si scrivano due funzioni, basate sulla ricerca binaria, per determinare il numero h di zeri in tempo $O(\log n)$ e $O(\log h)$, rispettivamente.

Esercizio 1.10 (Minimo in una matrice)**.** Data una matrice $n \times n$ di interi con n potenza di 2, si vuole determinarne l'elemento minimo. Si scriva una funzione utilizzando la tecnica *divide-et-impera*.

Esercizio 1.11 (Chi manca?)**.** Sia dato un vettore ordinato $V[1 \ldots n]$ contenente n elementi interi distinti appartenenti all'intervallo $1 \ldots n+1$. Si scriva una procedura, basata sulla ricerca binaria, per individuare in tempo $O(\log n)$ l'unico intero dell'intervallo $1 \ldots n+1$ che non compare in V.

Esercizio 1.12 (Massimo in sequenza unimodale)**.** Una sequenza a_1, a_2, …, a_n di n elementi distinti è unimodale se esiste un indice i, $1 \leq i \leq n$, tale che $a_1 < a_2 < \cdots < a_i$ e $a_i > a_{i+1} > \cdots > a_n$ (per $i = 1$ oppure $i = n$, allora la sequenza unimodale è anche ordinata). Si scriva una funzione che trovi in tempo $O(\log n)$ il massimo di una sequenza unimodale memorizzata in un vettore.

Esercizio 1.13 (Occorrenze)**.** Dato in input un vettore ordinato di n interi e un intero k, scrivere un algoritmo che restituisca in tempo $O(\log n)$ il numero di occorrenze di k in A.

1.7 Soluzioni

Soluzione Esercizio 1.2

Innanzitutto, è importante notare che il vettore di input deve contenere valori negativi, altrimenti la risposta è banalmente data dal vettore completo.

Un algoritmo banale e poco efficiente per risolvere questo problema consiste nel considerare tutti gli $n(n+1)/2$ sottovettori, calcolare la loro sommatoria e ritornare il valore più alto. Un algoritmo del genere ha costo $O(n^3)$. Si noti che il massimo viene inizializzato a zero; infatti, se tutti i valori sono negativi un sottovettore vuoto ha somma zero ed è preferibile.

```
int maxsum(int[] A, int n)
    int max = 0                                  % Massimo valore trovato
    for i = 1 to n do
        for j = i to n do
            int sum = 0                          % Somma sottovettore
            for k = i to j do
                sum = sum + A[k]
            if sum > max then max = sum
    return max
```

Un algoritmo migliore si può ottenere notando che la somma di $A[i \ldots j]$ è uguale alla somma di $A[i \ldots j-1]$ più $A[j]$. L'algoritmo corrispondente ha costo $O(n^2)$. Questa semplice ottimizzazione ha il notevole effetto di ridurre la complessità di un ordine di grandezza; sebbene sia possibile dire che è un'applicazione della tecnica della programmazione dinamica, che vedremo nel capitolo successivo, in realtà questa ottimizzazione è una semplice applicazione del buon senso.

```
int maxsum(int[] A, int n)
    int max = 0                                  % Massimo valore trovato
    for i = 1 to n do
        int sum = 0                              % Somma sottovettore
        for k = i to n do
            sum = sum + A[k]
            if sum > max then max = sum
    return max
```

Un approccio basato su divide-et-impera divide il vettore $A[i \dots j]$ in due parti $A[i \dots m]$, $A[m+1 \dots j]$, con $m = \lfloor (i+j)/2 \rfloor$. Ricorsivamente, viene calcolato:

- max_s, il valore massimo fra tutti i sottovettori contenuti nella parte sinistra $A[i \dots m]$
- max_d, il valore massimo fra tutti i sottovettori contenuti nella parte destra $A[m+1 \dots j]$

Inoltre, viene calcolato iterativamente il massimo sottovettore che sta a cavallo fra la parte sinistra e la parte destra, ottenuto calcolando:

- max'_s, il valore massimo fra tutti i sottovettori contenuti nella parte sinistra e confinanti con la parte destra (ovvero che terminano in $A[m]$);
- max'_d, il valore massimo fra tutti i sottovettori contenuti nella parte destra e confinanti con la parte sinistra (ovvero che iniziano in $A[m+1]$);

A questo punto, il valore finale è dato dal massimo fra max_d, max_s e $max'_d + max'_s$. Come caso base, si considerano vettori costituiti da 0 elementi (il cui valore è 0) o da un elemento (il cui valore è il massimo fra l'elemento stesso e 0, in modo da escludere valori negativi).

La chiamata iniziale è $\mathsf{maxsumRic}(A, 1, n)$; il costo computazionale è $O(n \log n)$.

```
int maxsumRic(int[] A, int i, int j)
    int max_s, max_d, max'_s, max'_d, s, m, k
    if i > j then return 0
    if i = j then return max(0, A[i])
    m = ⌊(i + j)/2⌋
    max_s = maxsumRic(A, i, m)
    max_d = maxsumRic(A, m + 1, j)
    sum = 0
    max'_s = 0
    for k = m downto i do
        sum = sum + A[k]
        if sum > max'_s then max'_s = sum
    sum = 0
    max'_d = 0
    for k = m + 1 to j do
        sum = sum + A[k]
        if sum > max'_d then max'_d = sum
    return max(max_s, max_d, max'_s + max'_d)
```

E' possibile trovare una soluzione ancora più efficiente, che opera in tempo $O(n)$, ma rimandiamo la soluzione al capitolo successivo, quando introdurremo la programmazione dinamica.

Soluzione Esercizio 1.3

Osservando la procedura Merge(), si nota che ogniqualvolta $A[i] > A[j]$ nel primo **if**, allora tutti gli elementi $A[i]$, $A[i+1]$, ..., $A[]$ provocano un'inversione con $A[j]$, per un totale di $-i+1$ inversioni. Pertanto, si può scrivere una funzione Contamerge(), ottenuta dalla Merge() aggiungendo un nuovo

parametro intero *count*, inizializzato a 0 e restituito come risultato della funzione, e modificando il primo **if** come segue:

if $A[i] \leq A[j]$ **then**
  $B[k] = A[i]$
  $i = i + 1$
else
  $B[k] = A[j]$
  $j = j + 1$
  $count = count + -i + 1$

La risultante funzione Inversioni(), che conta il numero totale di inversioni di A, è ottenuta banalmente dalla procedura MergeSort() ed ha anch'essa complessità $O(n \log n)$.

int Inversioni(**item** $A[\,]$, **int** ,**int**)

if $\geq$ **then**
  return 0
else
  $= \lfloor (+)/2 \rfloor$
  return Inversioni$(A,,)$ + Inversioni$(A,+1,)$ + Contamerge$(A,,,)$

Soluzione Esercizio 1.6

Una limitazione inferiore di $\Omega(n)$ si può dimostrare applicando la tecnica dell'oracolo. Si supponga di avere una matrice $M[1 \ldots n, 1 \ldots n]$ con righe e colonne ordinate in modo crescente con elementi uguali a 0 oppure 1 sulla diagonale secondaria, con elementi negativi nel triangolo sopra la diagonale secondaria, ed elementi maggiori di 1 nel triangolo sotto la diagonale secondaria:

$$M[i,j] \begin{cases} < 0 & \text{per } i < n-\text{j}+1, \\ = 0 \text{ oppure } 1 & \text{per } i = n-j+1, \\ > 1 & \text{per } i > n-j+1. \end{cases}$$

Se l'elemento k da cercare è 1 e c'è un solo 1 nella matrice, tale elemento può occupare una posizione qualsiasi sulla diagonale secondaria, pertanto un qualsiasi algoritmo deve necessariamente scandirla tutta.

La funzione ricerca() è una generalizzazione sulle matrici della ricerca binaria per i vettori ordinati. Si comincia la ricerca di k a partire dalla posizione in alto a destra $M[1,n]$. Si esclude dalla ricerca successiva la colonna n, qualora $M[1,n] > k$, oppure la riga 1, se $M[1,n] < k$. La ricerca viene poi riapplicata ricorsivamente alla sottomatrice rettangolare $n \times (n-1)$ (oppure $(n-1) \times n$) privata dell'ultima colonna (oppure della prima riga). La chiamata iniziale è ricerca$(M,1,n,k,n)$. Ad ogni chiamata si esclude o una riga o una colonna di M. Poiché M ha n righe ed n colonne, ricerca() ha complessità ottima $O(n)$.

```
boolean ricerca(item[][] M, int i, int j, int k, int n)
if M[i, j] = k then
    return true
else if M[i, j] < k then
    if i < n then
        return ricerca(M, i + 1, j, k, n)
    else
        return false
else
    if j > 1 then
        return ricerca(M, i, j − 1, k, n)
    return false
```

Soluzione Esercizio 1.7

Si assuma che i giocatori siano indicati con numeri interi da 1 ad n. Si costruisce una matrice $n \times n$, dove la prima colonna contiene i giocatori in ordine crescente e nelle rimanenti $n-1$ colonne si trovano tutti i giocatori che ognuno di essi dovrà incontrare in ciascuna giornata del calendario. In altri termini, la generica riga i della matrice contiene proprio i nella prima colonna e il giocatore che i deve incontrare nella giornata $j-1$ nella colonna $j > 1$. Per esempio, per $n = 2$ si ottiene la matrice:

1	2
2	1

mentre per $n = 4$ si ha la seguente matrice, ottenuta combinando opportunamente quattro matrici per $n = 2$:

1	2	3	4
2	1	4	3
3	4	1	2
4	3	2	1

In generale, denotata con $M_{i,j}$ la matrice per i giocatori compresi tra i e j, la matrice $M_{1,n}$ si ottiene combinando $M_{1,\frac{n}{2}}$ e $M_{\frac{n}{2}+1,n}$ nel modo seguente:

$M_{1,\frac{n}{2}}$	$M_{\frac{n}{2}+1,n}$
$M_{\frac{n}{2}+1,n}$	$M_{1,\frac{n}{2}}$

A questo punto viene spontaneo progettare la seguente procedura torneo() che costruisce la matrice $M_{1,n}$ riapplicando ricorsivamente la tecnica alle quattro sottomatrici, fino a quando non viene raggiunto il caso di due soli giocatori. Oltre alla matrice M, la procedura necessita di altri sei parametri, gli indici minimi e massimi di riga e colonna, che individuano la porzione di M sulla quale avviene la generica chiamata ricorsiva, e gli indici minimi e massimi dei giocatori. La procedura va chiamata nel programma principale con torneo$(M, 1, n, 1, n, 1, n)$.

```
torneo(item[][] M, int imin,imax,jmin,jmax,gmin,gmax)
if gmin = gmax − 1 then
    M[imin,jmin] = gmin
    M[imin,jmax] = gmax
    M[imax,jmin] = gmax
    M[imax,jmax] = gmin
else
    int imed = ⌊(imin + imax)/2⌋
    int jmed = ⌊(jmin + jmax)/2⌋
    int gmed = ⌊(gmin + gmax)/2⌋
    torneo(M, imin, imed, jmin, jmed, gmin, gmed)
    torneo(M, imed + 1, imax, jmed + 1, jmax, gmin, gmed)
    torneo(M, imed + 1, imax, jmin, jmed, gmed + 1, gmax)
    torneo(M, imin, imed, jmed + 1, jmax, gmed + 1, gmax)
```

Soluzione Esercizio 1.9

La funzione di complessità $O(\log n)$ esegue una ricerca binaria considerando il valore $V[m]$ dell'elemento centrale. Se $V[m] = 1$ viene fatta una chiamata ricorsiva sulla prima metà $V[i \dots m-1]$, mentre se $V[m] = 0$ è effettuata una chiamata sulla seconda metà $V[m+1 \dots j]$. Per evitare lo "scavalcamento" degli indici i e j, la ricorsione è arrestata quando la porzione ancora da esaminare comprende o un solo elemento (caso $i = j$) oppure due elementi (caso $i = j-1$). In entrambi i casi, se l'elemento sinistro $V[i]$ è 1, allora l'ultimo 0 si trova in posizione $h = i-1$ (se non c'è alcuno 0 allora deve essere $i = 1$ e correttamente $h = i-1 = 0$); se invece l'elemento destro $V[j]$ è 0, allora $h = j$ (se ci sono tutti zeri, allora deve essere $j = n$ e quindi $h = j = n$). Infine, la terza possibilità occorre solo se ci sono esattamente due elementi (cioè quando $i = j-1$) dei quali il sinistro non è 1 ed il destro non è 0; ma in tal caso deve essere $V[i] = 0$ e $V[j] = 1$, ed il numero h di zeri è i. La chiamata nel programma principale è conta0$(V, 1, n)$.

Per contare il numero h di zeri in tempo $O(\log h)$, si scandisce con un ciclo **while** il vettore V da sinistra verso destra per potenze di due. Se l'elemento considerato $V[i]$ è uguale a 0, si procede nella scansione, raddoppiando i, ma evitando di superare la dimensione n del vettore. Se al termine della scansione l'ultimo elemento $V[i]$ esaminato vale 0, allora si chiama la conta0() tra tale elemento e l'ultimo elemento $V[n]$ del vettore. Altrimenti si chiama la conta0() tra l'ultimo elemento $V[i]$ esaminato (che vale 1) e l'elemento $V[i/2]$ esaminato in precedenza, se $i/2$ non è inferiore ad 1, oppure $V[1]$. La chiamata è trovaH(V, n).

```
int conta0(int[] V, int i, j)
if i = j or i = j − 1 then
    while V[i] ≠ 1 and i ≤ j do i = i + 1
    return i − 1
else
    m = ⌊(i + j)/2⌋
    if V[m] = 1 then
        return conta0(V, i, m − 1)
    else
        return conta0(V, m + 1, j)
```

```
int trovaH(int[] V, int n)
    int i = 1
    while V[i] = 0 and 2·i ≤ n do
        i = 2·i
    if V[i] = 0 then
        return conta0(V, i, n)
    else if i/2 > 1 then
        return conta0(V, i/2, i)
    else
        return conta0(V, 1, i)
```

Poiché la scansione procede raddoppiando ogni volta i finché non è individuato un intervallo tra due potenze di 2 che include h (diciamo $2^s \leq h \leq 2^{s+1}$), il ciclo **while** costa $O(\log 2^s) = O(\log h)$. La conta0() viene poi eseguita sugli elementi appartenenti a tale intervallo, il cui numero è $O(2^s) = O(h)$, e quindi la sua complessità è $O(\log 2^s) = O(\log h)$. Ovviamente, questo metodo è vantaggioso quando h è molto minore di n.

Soluzione Esercizio 1.10

La funzione min() divide la matrice in 4 sottomatrici di dimensioni dimezzate, richiama ricorsivamente se stessa sulle 4 sottomatrici, e poi trova il minimo tra i 4 minimi delle sottomatrici. Per semplicità, si assume di disporre di una funzione min4(a,b,c,d), che restituisce il minimo tra i quattro interi a, b, c, d.

```
int min(item[][] M, int imin, int imax, int jmin, int jmax)
    if imin > imax or jmin > jmax then
        return +∞
    else if imin = imax and jmin = jmax then
        return M[imin, imax]
    else
        int imed = ⌊(imin + imax)/2⌋
        int jmed = ⌊(jmin + jmax)/2⌋
        int a = min(imin, imed, jmin, jmed)
        int b = min(imed + 1, imax, jmin, jmed)
        int c = min(imin, imed, jmed + 1, jmax)
        int d = min(imin + 1, imax, jmed + 1, jmax)
        return min4(a, b, c, d)
```

Soluzione Esercizio 1.12

Se ci sono almeno due elementi, basta considerare l'elemento centrale del vettore, confrontarlo col successivo, e continuare la ricerca nella prima o nella seconda metà del vettore:

```
int maxUnimodale(item[] A, int i, j)
    int m = ⌊(i + j)/2⌋
    if i = j then
        return A[m]
    else
        if A[m] < A[m + 1] then
            return maxUnimodale(A, m + 1, j)
        else
            return maxUnimodale(A, i, m)
```

Soluzione Esercizio 1.13

È possibile modificare la ricerca binaria rimuovendo il caso in cui si verifica l'identità con k; in questo modo tutte le ricerche diventano ricerche con insuccesso e terminano sul primo valore maggiore o minore di k (a seconda se si verifica prima se un numero è maggiore di k o viceversa). Eseguendo quindi due ricerche binarie, con gli **if** scambiati, si ottengono in tempo $O(\log n)$ le posizioni dei valori immediatamente minore e maggiore. Successivamente si può calcolare il numero di occorrenze di k in tempo $O(1)$.