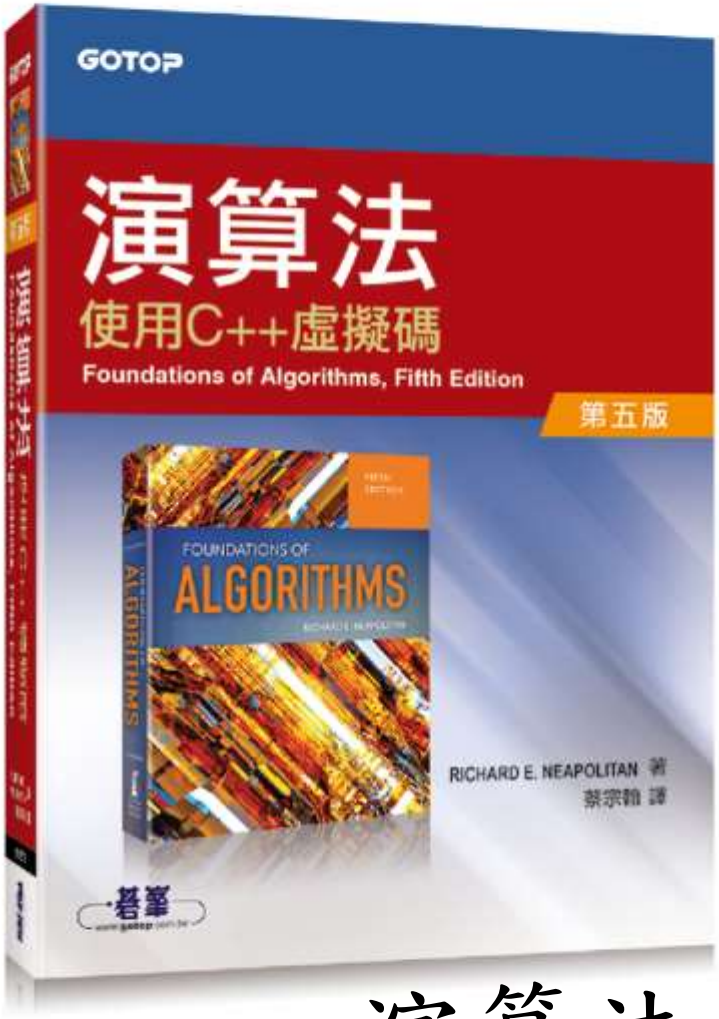


演算法
使用C++虛擬碼
Foundations of Algorithms, Fifth Edition
第五版

第一章
演算法：效率、分析與量級

第一章 演算法：效率、分析與量級

1.1 演算法

- 問題
  - Ex(範例1.1):將由n個數字構成的串列S，依非遞減的順序進行排序
  - S=[10, 7, 11, 5, 13, 8]
- 答案
  - 就是排序後串列中的數字
  - [5, 7, 8, 10, 11, 13]

1.1 演算法

- 問題
  - 如何設計個別的模組以便完成特定的工作。這些特定的工作被稱為“問題”
    - Ex(範例1.2): 試回答某數字x是否在由n個數字的串列S中。當x在S中時，回答“是”，否則答“否”。
- 參數
  - “問題”可能含有一些在該問題的敘述中，未被指定為特定數值的變數。這些變數即被稱為該問題的“參數”(parameter)
    - Ex: 範例1.2的問題有三個參數：S、n及x。

- 由於“問題”帶有參數，因此。我們可以給定不同的參數值以便產生問題的“實例”(instance)。問題實例的解(solution)就是該實例中所問題目的答案。
- **範例1.4**
  - 範例1.2問題的某實例為：
    $S = [10, 7, 11, 5, 13, 8]$，$n = 6$，且$x = 5$
  - 這個實例的解答為“是的，x在S中”。

- 演算法
  - 一個通用而逐步執行的程序以便求出每個問題實例的解答。這個逐步執行的程序即稱為“演算法”(algorithm)
  - “演算法”是用來解“問題”的
- **範例1.5**
  - 某個解範例1.2問題的演算法如下：
  - 由S中的第一個項目開始，循序地比較x與S中的項目，直到找到x，或S中的所有項目都已經被比對完畢。若找到x，答案就是“是”，否則答案就是“否”。

- 使用人類的自然語言描述演算法的缺點
  1. 這種方式很難用來撰寫複雜的演算法，即使寫出，別人也很難讀懂
  2. 根據自然語言描述的演算法來撰寫電腦程式的過程並不是很清楚。
- 由於C++是同學們目前相當熟悉的一種程式語言，因此我們使用類似C++的虛擬碼來撰寫演算法

演算法1.1 循序搜尋法(Sequential Search)

問題：判斷 x 這個 key 是否位於含有 n 個 key 的陣列 S 中。

輸入（參數）：正整數 n，由 n 個 key 所構成的陣列（索引值由 1 到 n），以及 key x。

輸出：$location$，x 在 S 中的位置（如果 x 不在 S 中，將傳回 0）。

```
void seqsearch ( int n ,
                 const keytype S[ ] ,
                 keytype x ,
                 index& location )
{
   location = 1;
   while ( location <= n && S [location] != x )
      location ++;
   if (location > n)
      location =0;
}
```

演算法1.2 加總陣列中的項目

問題：加總所有在陣列 S 中的 n 個數字。

輸入：正整數 n，索引由 1 到 n 的陣列 S。

輸出：sum，S 中所有數字的總和。

```
number sum (int n, const number S[])
{
    index i;
    number result;

    result = 0;
    for (i = 1; i <= n; i++)
        result = result + S[i];
    return result;
}
```

演算法1.3　交換排序法(Exchange Sort)

問題：以非遞減的順序對 n 個 key 進行排序。

輸入：正整數 n，含有 n 個 key 的陣列 S（索引由 1 到 n）。

輸出：陣列 S，S 中的 n 個 key 已依非遞減順序排列。

```
void exchangesort (int n, keytype S[])
{
    index i, j;
    for (i=1; i<=n; i++)
        for (j=i+1; j<=n; j++)
            if (S[j] < S[i])
                交換 S[i] 與 S[j] 的值;
}
```

演算法1.4 矩陣乘法

(1) 定義

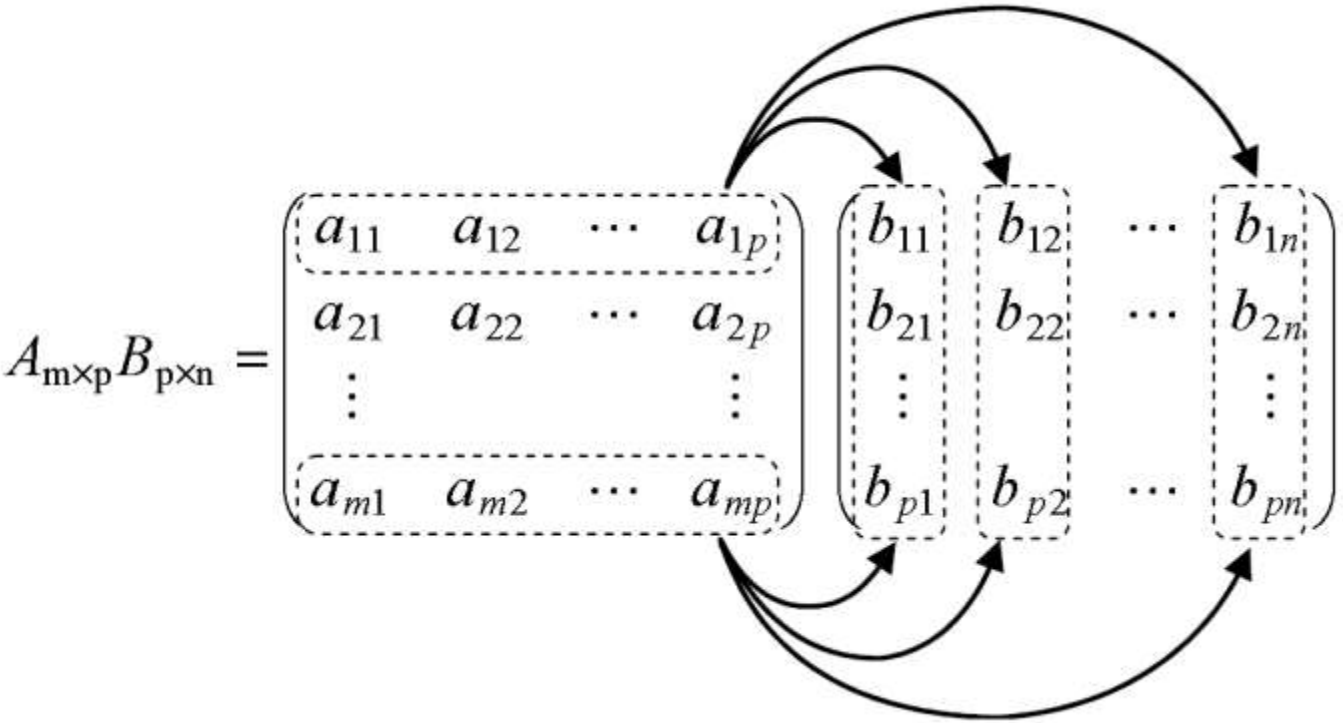
設 $\boldsymbol{A}=[a_{ij}]_{m\times n}$ ，$\boldsymbol{B}=[b_{ij}]_{n\times l}$ ，則

$\boldsymbol{A}\times\boldsymbol{B}=\boldsymbol{C}=[c_{ij}]_{m\times l}$ ，其中：$c_{ij}=\sum_{k=1}^{n}a_{ik}b_{kj}$ 。

Note

A的行數$=B$的列數時，$A\times B$才有意義矩陣乘法參考下方示意圖：

演算法1.4 矩陣乘法


$A_{m\times p}B_{p\times n} =$
a_{11} a_{12} ⋯ a_{1p}
a_{21} a_{22} ⋯ a_{2p}
a_{m1} a_{m2} ⋯ a_{mp}
b_{11} b_{12} ⋯ b_{1n}
b_{21} b_{22} ⋯ b_{2n}
b_{p1} b_{p2} ⋯ b_{pn}

演算法1.4 矩陣乘法

$$= \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} + \cdots a_{1p}b_{p1} & \cdots & a_{11}b_{1n} + a_{12}b_{2n} + \cdots a_{1p}b_{pn} \\ a_{21}b_{11} + a_{22}b_{21} + \cdots a_{2p}b_{p1} & \cdots & a_{21}b_{1n} + a_{22}b_{2n} + \cdots a_{2p}b_{pn} \\ \vdots & & \vdots \\ a_{m1}b_{11} + a_{m2}b_{21} + \cdots a_{mp}b_{p1} & \cdots & a_{m1}b_{1n} + a_{m2}b_{2n} + \cdots a_{mp}b_{pn} \end{pmatrix}$$

$$= \left(\sum_{k=1}^{p} a_{ik}b_{kj} \right)_{m \times n}$$

求下列兩矩陣之乘積AB。

(1) $\boldsymbol{A}=\begin{bmatrix}1 & -2\\ 3 & 4\end{bmatrix}$，$\boldsymbol{B}=\begin{bmatrix}3 & 1\\ -2 & 0\end{bmatrix}$

(2) $\boldsymbol{A}=\begin{bmatrix}5 & 3\\ -2 & 1\\ 0 & 7\end{bmatrix}$，$\boldsymbol{B}=\begin{bmatrix}3 & 1\\ -2 & 0\end{bmatrix}$

(1) $\boldsymbol{AB}=\begin{bmatrix}1\cdot 3+(-2)\cdot(-2) & 1\cdot 1+(-2)\cdot 0\\ 3\cdot 3+4\cdot(-2) & 3\cdot 1+4\cdot 0\end{bmatrix}=\begin{bmatrix}7 & 1\\ 1 & 3\end{bmatrix}$

(2) $\boldsymbol{AB}=\begin{bmatrix}5\cdot 3+(3)\cdot(-2) & 5\cdot 1+(3)\cdot 0\\ (-2)\cdot 3+1\cdot(-2) & (-2)\cdot 1+1\cdot 0\\ 0\cdot 3+(7)\cdot(-2) & 0\cdot 1+(7)\cdot(0)\end{bmatrix}$

$=\begin{bmatrix}9 & 5\\ -8 & -2\\ -14 & 0\end{bmatrix}$。

演算法1.4 矩陣乘法

問題：求得兩個 $n \times n$ 矩陣的乘積。

輸入：正整數 n，二維數字陣列 A 與 B，這兩個矩陣的行索引及列索引都是由 1 到 n。

輸出：二維數字陣列 C，這個矩陣的行索引及列索引都是由 1 到 n，並含有 A 與 B 的乘積。

```
void matrixmult (int n,
                 const number A[][] ,
                 const number B[][] ,
                 number C[][])
{
    index i, j, k;

    for (i=1; i<=n; i++)
        for (j=1; j<=n; j++){
            C[i][j] = 0;
            for (k=1; k<=n; k++)
                C[i][j] = C[i][j] + A[i][k] * B[k][j];
        }
}
```

1.2 發展有效率演算法的重要性
1.2.1 循序搜尋法與二元搜尋法

- 二元搜尋法

  假設我們要搜尋x

  – 先比較x與陣列的中間項。若這兩數相等，演算法就執行完畢。

  – 若x比中間項小，那麼x一定位於陣列的前半段，並且二元演算法會對陣列的前半段重複剛才的搜尋

  – 如果x比陣列的中間項大，二元演算法就會對陣列的後半段重複剛才的搜尋

  – 這個procedure會一直重複執行，直到找到x，或確定x不在陣列中才會停止。

演算法1.5 二元搜尋法(Binary Search)

問題：判斷 x 這個 key 是否位於含有 n 個 key 的已排序陣列 S 中。

輸入：正整數 n，由 n 個 key 所構成的已排序陣列（依非遞減順序排列，索引值由 1 到 n），以及 x 這個 key。

輸出：$location$，x 在 S 中的位置（如果 x 不在 S 中，將傳回 0）。

```
void binsearch (int n,
                const keytype S[],
                keytype x,
                index& location)
```

```
{
    index low, high ,mid;

    low = 1; high = n;
    location = 0;
    while (low <= high && location == 0) {
        mid = ⌊(low+hight)/2⌋;
        if (x == S[mid])
            location = mid;
        else if (x < S[mid])
            high = mid - 1;
        else
            low = mid + 1;
    }
}
```

循序搜尋法 vs. 二元搜尋法

- 計算每個演算法所執行的比較次數
  - 演算法1.1(循序搜尋法)
    - 若陣列S含有32個項目，且x並不在S中，在得知x不在S中之前，會把x與所有32個項目比較
    - 若x在陣列中，所執行的比較次數將不會超過n
  - 演算法1.5(二元演算法)
    - 每執行一遍迴圈只會執行一次x與$S[mid]$的比較
    - 當x比一個大小為32之陣列中的所有項目都大時，二元演算法將會執行六次比較(請注意：$6 = \lg 32 + 1$)
    - 如果x在陣列中，或x小於陣列中的所有項目，或是x位於兩個項目之間，所需執行的比較次數將不會超過當x大於陣列中所有項目時所需執行的比較次數
    - 在一般的情況下，每當我們倍增陣列的大小，我們只多執行了一次比較。因此，若n是2的乘冪且x大於陣列中的所有n個項目時，二元搜尋法所執行的比較次數為$\lg n + 1$。

- 表 1.1　當 x 大於所有陣列中的項目時，循序搜尋法與二元搜尋法各自使用的比較次數

陣列大小	循序搜尋法使用的比較次數	二元搜尋法使用的比較次數
128	128	8
1,024	1,024	11
1,048,576	1,048,576	21
4,294,967,296	4,294,967,296	33

1.2.2 費布那西數列 (Fibonacci Sequence)

- 遞迴式定義
  - $f_0 = 0$
  - $f_1 = 1$
  - $f_n = f_{n-1} + f_{n-2}$ for $n>=2$

演算法1.6
費布那西數列的第n項(遞迴版)

問題：求得費布那西數列的第 n 項。

輸入：非負正整數 n。

輸出：*fib*，費布那西數列的第 n 項。

```
int fib (int n)
{
   if (n <= 1)
      return n;
   else
      return fib(n-1) + fib(n-2);
}
```

n	計算的項目數
0	1
1	1
2	3
3	5
4	9
5	15
6	25

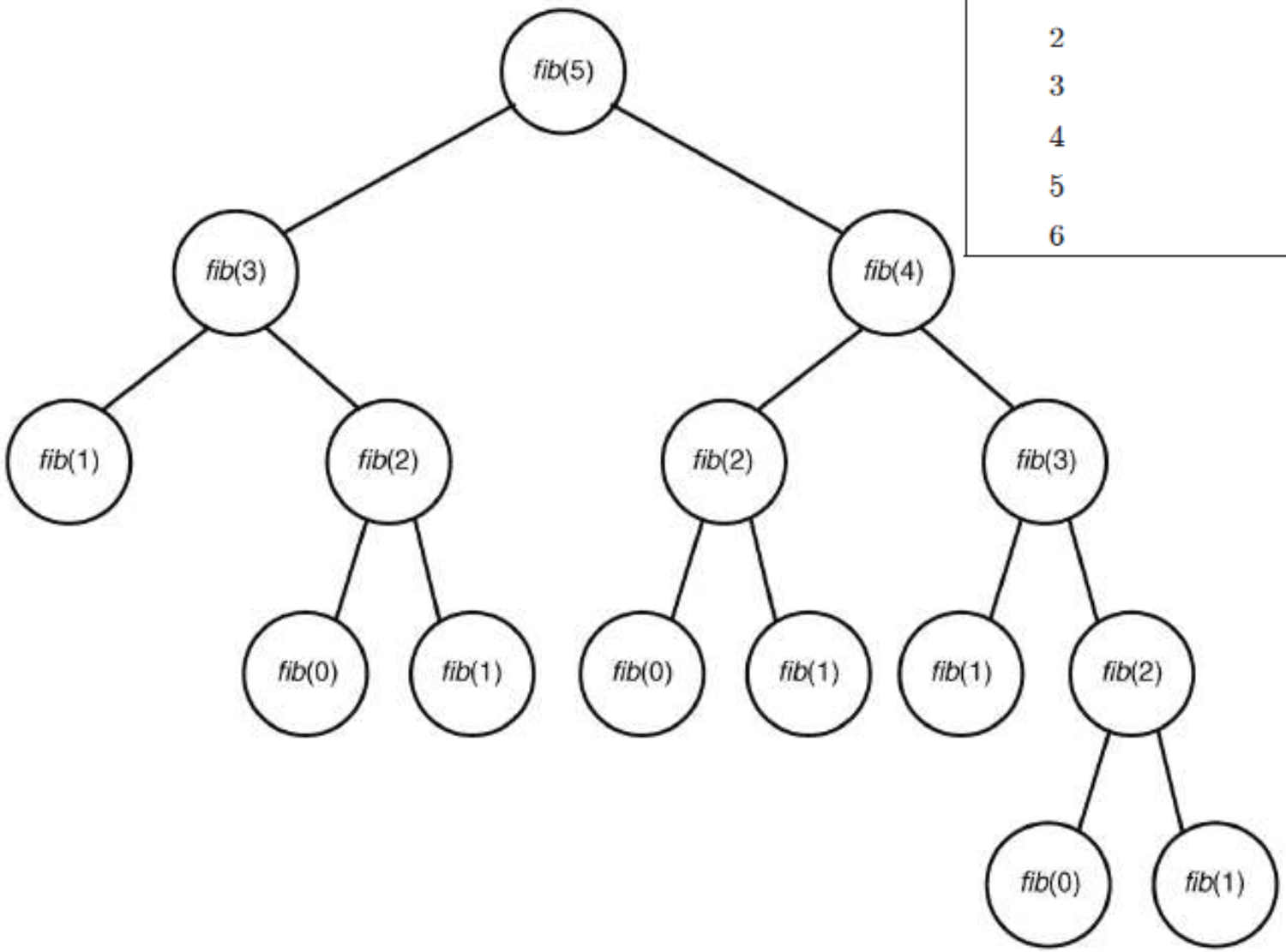


圖 1.2　用演算法 1.6 計算費布那西數列的第五項時，相對應產生的遞迴樹。

目錄

定理 1.1

若 $T(n)$ 代表演算法 1.6 相對應的遞迴樹含有的項目數。則對於 $n \geq 2$，

$$T(n) > 2^{n/2}$$

證明：我們利用在 n 上執行歸納法來證明。

歸納基底：我們需要兩個基礎的例子，因為歸納步驟中需要用到前兩個例子的結果。對於 $n=2$ 及 $n=3$，圖 1.2 中的遞迴樹顯示

$$T(2) = 3 > 2 = 2^{2/2}$$

$$T(3) = 5 > 2.8323 \approx 2^{3/2}$$

歸納假設：產生歸納假設的其中一個方式是，假設對於所有 $m<n$，這個敘述都成立。接著，在歸納步驟中，證明就 n 而言，此表示同樣的敘述也必須是成立的。我們在這個證明中將使用這個方式。假設對於所有的 m，且 $2 \leq m < n$

$$T(m) > 2^{m/2}$$

歸納步驟：我們必須證明 $T(n)>2^{n/2}$。$T(n)$ 的值等於 $T(n-1)$ 與 $T(n-2)$ 的總和加上一個根節點，因此，

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + 1 \\ &> 2^{(n-1)/2} + 2^{(n-2)/2} + 1 \qquad \text{（根據歸納假設）} \\ &> 2^{(n-2)/2} + 2^{(n-2)/2} = 2 \times 2^{(\frac{n}{2})-1} = 2^{n/2} \end{aligned}$$

- 演算法1.6(費布那西數列第n項)的所計算的項目數大於$2^{n/2}$。這個結果已證明此演算法是多麼沒效率。
- 演算法1.6屬於divide-and-conquer的一種
  - Binary search→有效率
  - 演算法1.6→無效率

- 接著，讓我們發展一種有效率的演算法來計算費布那西數列的第*n*項 。
- 若在算出一個值以後，我們就把這個值存放在一個陣列中，那麼之後如果我們需要這個值，我們就不必重新計算 。
- 下面的iterative(重複式)演算法使用了這個策略
- 屬於Dynamic programming的一種

演算法1.7
費布那西數列的第n項(iterative版)

問題：求得費布那西數列的第 n 項。

輸入：非負正整數。

輸出：fib2，費布那西數列的第 n 項。

```
int fib2 (int n)
{
    index i;
    int f[0..n];
    f[0]=0;
    if (n > 0)
        f[1]=1;
        for (i=2; i<=n; i++)
            f[i] = f[i-1] + f[i-2];
    }
    return f[n];
}
```

- 表 1.2　演算法 1.6 與 1.7 的比較

n	$n+1$	$2^{n/2}$	使用演算法 1.7 的執行時間	使用演算法 1.6 的執行時間的下限
40	41	1,048,576	41 奈秒 *	1048 微秒 †
60	61	1.1×10^{9}	61 奈秒	1 秒
80	81	1.1×10^{12}	81 奈秒	18 分
100	101	1.1×10^{15}	101 奈秒	13 天
120	121	1.2×10^{18}	121 奈秒	36 年
160	161	1.2×10^{24}	161 奈秒	3.8×10^{7} 年
200	201	1.3×10^{30}	201 奈秒	4×10^{13} 年

*1 奈秒 $=10^{-9}$ 秒。

†1 微秒 $=10^{-6}$ 秒。

1.3 演算法的分析
1.3.1 複雜度分析

- 演算法的執行時間與下列有關：
  - 輸入量的大小
  - 基本運算
    - 某些指令或指令群組使得演算法做的工作總和約略與這些指令或指令群組被執行的次數成正比。
    - 我們稱這些指令或指令群組為該演算法中的基本運算。
- 計算某些基本運算的執行次數，將比執行次數當作是與輸入大小有關的函數，來分析演算法的效率。

時間複雜度分析

- 求得每個不同的輸入大小，該演算法所執行的基本運算次數
- 在某些情況(Ex:演算法1.1)中，基本運算執行的次數與輸入大小及輸入的值均有關
- 在其他情況(如演算法1.2)中，對於所有大小為*n*的個體，基本運算都是執行同樣的次數
  - **T(n)**: 對於一個大小為*n*的個體，該演算法所執行的基本運算次數
    - *T*(*n*)即被稱為演算法的所有情況時間複雜度

分析演算法1.2

所有情況的時間複雜度(加總陣列中的項目)

除控制指令外，在迴圈中唯一的指令就是把陣列中的一個項目加總到*sum*這個變數中。因此，我們稱這個指令為此演算法的基本運算。

- 基本運算：將陣列中的一個項目加總到*sum*這個變數中。
- 輸入大小：n，陣列中的項目個數。

無論陣列中所含的數值是什麼，for迴圈會執行n次。因此，基本運算總是被執行n次，故可得

$$T(n) = n$$

分析演算法 1.3

所有情況的時間複雜度（交換排序法）

如同先前所提及，對於利用 key 的比較來進行排序的演算法，我們可以把比較指令或是指派值指令當作是基本運算來看。我們將在這裡分析比較的次數。

基本運算：比較 $S[j]$ 與 $S[i]$。

輸入大小：n，被排序的項目個數。

我們必須求得 **for-j** 迴圈執行的次數。給定 n，**for-i** 迴圈執行的次數恆為 $n-1$。在 **for-i** 迴圈第一次執行過後，**for-j** 迴圈執行 $n-1$ 次；在 **for-i** 迴圈第二次執行過後，**for-j** 迴圈執行 $n-2$ 次；在 **for-i** 迴圈第三次執行過後，**for-j** 迴圈執行 $n-3$ 次，…，在 **for-i** 迴圈最後一次執行過後，**for-j** 迴圈執行 1 次。因此，**for-j** 迴圈總共執行的次數可由下式求得

$$T(n) = (n-1) + (n-2) + (n-3) + \cdots + 1 = \frac{(n-1)\,n}{2}$$

最後的等式是根據附錄 A 的範例 A.1 所推導出。

分析演算法 1.4

所有情況的時間複雜度（矩陣乘法）

在最內層的 **for** 迴圈中只有一個由一個乘法與一個加法組成的指令。這個演算法可以被實作成執行的加法個數遠少於執行的乘法個數。因此，我們將只設定乘法指令為基本運算。

基本運算：在最內層的 **for** 迴圈中的乘法指令。

輸入大小：n，行數與列數。

for-i 迴圈的執行次數恆為 n。**for**-i 迴圈每執行一次，**for**-j 迴圈就執行 n 次；**for**-j 迴圈每執行一次，**for**-k 迴圈就執行 n 次。因為基本運算在 **for**-k 迴圈的內部，因此

$$T(n) = n \times n \times n = n^3$$

三種其他的時間複雜度分析技巧

一. 考慮基本運算被執行次數的最大值

- 對於一個給定的演算法，**W(n)**被定義為在輸入大小為*n*的情況下，該演算法執行基本運算次數的最大值
  - **W(n)**被稱為該演算法的最差情況時間複雜度

分析演算法1.1 (一)

最差情況的時間複雜度 (循序搜尋法)

- 基本運算：將x與陣列中的一個項目進行比較
- 輸入大小：n，陣列中的項目個數
- 基本運算最多執行n次，也就是當x位於陣列的尾端或x不在陣列中時。因此，

$$W(n) = n$$

三種其他的時間複雜度分析技巧

二. 對於一個給定的演算法

- **A(n)**被定義為在輸入大小為*n*的情況下，該演算法執行基本運算次數的平均值(請見附錄A中的A.8.2節對於平均值的討論)。
- 因此**A(n)**被稱為該演算法的平均情況時間複雜度
- 若要計算*A*(*n*)，我們必須對大小為*n*的所有可能輸入都指定機率值
- 進行平均情況的分析通常要比最差情況的分析更困難。

分析演算法1.1 (二)

平均情況的時間複雜度(循序搜尋法)

基本運算：將x與陣列中的一個項目進行比較。

輸入大小：n，陣列中的項目個數。

- 已知x在S中的情況
  - 所有在S中的項目都是相異的，因此沒有理由認為x出現在某個位置的機率大於出現在另一個位置的機率。根據這個資訊，對於$1 \le k \le n$，x位於第k個位置的機率為$1/n$。若x位於第k個位置，找到x所需執行的基本運算次數為k。這代表的是平均時間複雜度可以由下式得到：

$$A(n) = \sum_{k=1}^{n} \left(k \times \frac{1}{n} \right) = \frac{1}{n} \times \sum_{k=1}^{n} k = \frac{1}{n} \times \frac{n(n+1)}{2} = \frac{n+1}{2}$$

- x可能不在陣列中的情況
  - 我們必須指定x位於從1到n的每個位置之機率都是相同的。x位於第k個位置的機率為p/n，x不在陣列中的機率為$1-p$。回想若x在第k個位置被找到，迴圈就跑了k次，若x沒有在陣列中被找到，則迴圈則跑了n次。因此平均的時間複雜度可由下式得到：

$$A(n) = \sum_{k=1}^{n} \left(k \times \frac{P}{n}\right) + n(1-p)$$

$$= \frac{p}{n} \times \frac{n(n+1)}{2} + n(1-p) = n\left(1 - \frac{p}{2}\right) + \frac{p}{2}$$

  - 若$p = 1$，$A(n) = (n+1)/2$，如同前面所求得的；而當$p = 1/2$，$A(n) = 3n/4+1/4$。這表示平均約有3/4的陣列項目會被搜尋到。

三種其他的時間複雜度分析技巧

三. 求得基本運算執行次數的最小值。

- 對於一個給定的演算法，**B(n)**被定義為在輸入大小為*n*的情況下，該演算法執行基本運算次數的最小值。
- 因此**B(n)**被稱為該演算法的最佳情況時間複雜度

分析演算法1.1(三)

最佳情況的時間複雜度(循序搜尋法)

- 基本運算：把x與陣列中的一個項目進行比較。
- 輸入大小：n，陣列中的項目個數。

因為 ，因此迴圈至少會走一次，如果$x = S[1]$，那麼無論n的大小為何，迴圈都是只走一次。因此，

$$B(n) = 1$$

複雜度函數

- 任一個將正整數映射到非負實數的函數
- **範例1.6**

下列的函數

$$f(n) = n$$
$$f(n) = n^2$$
$$f(n) = \lg n$$
$$f(n) = 3n^2 + 4n$$

都可以做為複雜度函數，因為它們都將正整數映射到非負實數。

1.3.2演算法分析原理的應用

- 同一個問題經常有兩種不同的演算法做為解法
- 回想我們曾經假設處理額外負擔指令的時間是可以被忽略不計。但如果處理額外負擔指令的時間無法被忽略不計，那麼我們必須在決定第一種演算法何時較有效率時，同時也將額外負擔指令的因素考慮進去。

1.3.3正確性分析

- 在本文中，“演算法分析”表示以時間或記憶體為基準對該演算法進行效率的分析。
- 例如，我們可發展證明方法來確定該演算法真的做到它原本預期要做的事情，也就是分析該演算法正確性(correctness)。

1.4 量級(Order)

- 在n夠大的情況下，不論執行基本運算所費的時間多寡，時間複雜度為n的演算法會比另一個時間複雜度為n^2的演算法更有效率
  - 擁有n或$100n$這種時間複雜度的演算法稱為**線性時間演算法(linear-time algorithm)**
  - 擁有像是n^2或$0.01n^2$這種時間複雜度的演算法則稱為**平方時間演算法 (quadratic-time algorithm)**

1.4.1 以直觀的方式介紹量級的概念

- 直觀上，當進行複雜度函數分類時，我們應能將低次項予以捨棄
  - 所有與純平方函數分在一起的複雜度函數構成的集合稱為 $\Theta(n^2)$
    - Ex: $g(n) = 5n^2 + 100n + 20 \in \Theta(n^2)$
  - 若某個演算法的時間複雜度在$\boldsymbol{\Theta(n^2)}$中，該演算法就被稱為平方時間演算法 (quadratic-time algorithm)或 $\boldsymbol{\Theta(n^2)}$ 演算法

- 最常見的複雜度類別：
  $\Theta(lgn)$ $\Theta(n)$ $\Theta(nlgn)$ $\Theta(n^2)$ $\Theta(n^3)$ $\Theta(2^n)$

- 通常，一個演算法的複雜度必須為 $\Theta(nlgn)$ 或比 $\Theta(nlgn)$ 更好，我們才會認為這個演算法能在可以忍受的時間內處理輸入大小非常大的個體

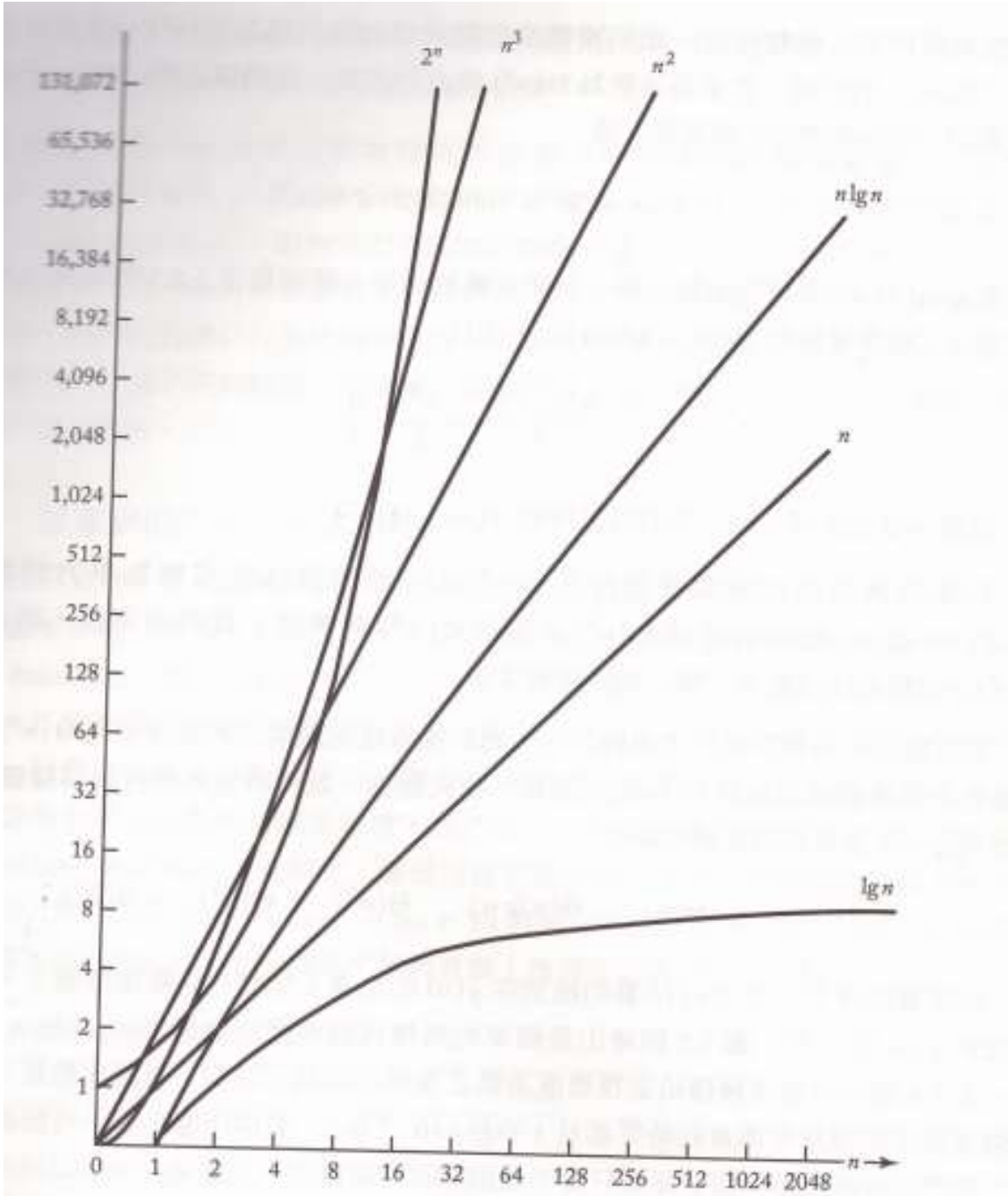


圖 1.3 常用複雜度函數的成長率。

• 表 1.4 給定時間複雜度下，各類演算法的執行時間

n	$f(n)=\lg n$	$f(n)=n$	$f(n)=n\lg n$	$f(n)=n^2$	$f(n)=n^3$	$f(n)=2^n$
10	0.003 微秒 *	0.01 微秒	0.033 微秒	0.10 微秒	1.0 微秒	1 微秒
20	0.004 微秒	0.02 微秒	0.086 微秒	0.40 微秒	8.0 微秒	1 毫秒 †
30	0.005 微秒	0.03 微秒	0.147 微秒	0.90 微秒	27.0 微秒	1 秒
40	0.005 微秒	0.04 微秒	0.213 微秒	1.60 微秒	64.0 微秒	18.3 分
50	0.006 微秒	0.05 微秒	0.282 微秒	2.50 微秒	125.0 微秒	13 天
10^2	0.010 微秒	0.1 微秒	0.664 微秒	10 微秒	1 毫秒	4×10^{13} 年
10^3	0.010 微秒	1 微秒	9.966 微秒	1 毫秒	1 秒	
10^4	0.013 微秒	10 微秒	130 微秒	100. 毫秒	16.70 分	
10^5	0.017 微秒	0.1 毫秒	1.670 毫秒	10 秒	11.6 天	
10^6	0.020 微秒	1.0 毫秒	19.930 毫秒	16.70 分	31.70 年	
10^7	0.023 微秒	0.01 秒	2.660 秒	1.16 天	31,709 年	
10^8	0.027 微秒	0.1 秒	2.660 秒	115.7 天	3.17×10^7 年	
10^9	0.030 微秒	1.00 秒	29.900 秒	31.70 年		

*1 微秒 $=10^{-6}$ 秒。 †1 毫秒 $=10^{-3}$ 秒。

1.4.2 以嚴謹的方式介紹量級的概念

- **big *O***

定義 給定一複雜度函數$f(n)$，O($f(n)$)就是由一些複雜度函數$g(n)$構成的集合。其中，對於每一個$g(n)$，必存在某個正實數常數c與某個非負整數N，使得對於所有$n \geq N$

$$g(n) \leq c \times f(n)$$

若$g(n) \in O(f(n))$，稱$g(n)$為$f(n)$的**big O**

- **big O在一個函數上設置了一個漸近上限**

- 下列的例子解釋如何找到“Big O”。
- **範例1.7**

  我們將證明$5n^2 \leq O(n^2)$。因為，對於$n \geq 0$，

  $$5n^2 \leq 5n^2$$

  我們可以取$c = 5$以及$N = 0$以獲得我們想要的結果

- **Omega Ω**

  －放置一個漸近下限在複雜度函數上

定義　對於一個給定的複雜度函數$f(n)$，$\Omega(f(n))$就是由一些複雜度函數$g(n)$構成的集合。其中，對於每一個$g(n)$，必存在某個正實數常數c與某個非負整數N，使得對於所有$n \geq N$，

$$g(n) \geq c \times f(n)$$

- 若$g(n) \in \Omega(f(n))$，我們稱$g(n)$為$f(n)$的**omega**

- **範例1.12**

我們將證明$5n^2 \in \Omega(n^2)$。因為，對於 n ≥ 0，

$$5n^2 \geq 1 \times n^2$$

我們可以取$c = 1$ 以及 $N = 0$ 以獲得我們想要的結果。

- 如果一個函數既在$O(n^2)$中也在$\Omega(n^2)$中，我們可以推斷，最終這個函數會在某個純平方函數之下且在另一個平方函數之上。也就是說，最終這個函數會跟某個純平方函數一樣好，並與另一純平方函數一樣差。我們可以由此推斷出這個函數的成長率應該接近一個純平方函數的成長率

嚴謹量級標示法 $\Theta(f(n))$

定義 對於一個給定的複雜度函數 $f(n)$，

$$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$$

這代表的是$\Theta(f(n))$就是由一些複雜度函數 $g(n)$構成的集合。其中，對於每一個 $g(n)$，必存在正實數常數 c 與 d 及某個非負整數 N，使得對於所有 $n \geq N$，

$$c \times f(n) \leq g(n) \leq d \times f(n)$$

若$g(n) \in \Theta(f(n))$，我們稱$g(n)$為$f(n)$量級。

- **範例1.16**

再回想演算法 1.3 的時間複雜度（交換排序法）。由範例 1.8 與 1.14 可得到

$$T(n)=\frac{n(n-1)}{2} \quad \text{既在 } O(n^2) \text{ 中也在 } \Omega(n^2) \text{ 中。}$$

因此我們可得到 $T(n)\in O(n^2)\cap\Omega(n^2)=\Theta(n^2)$

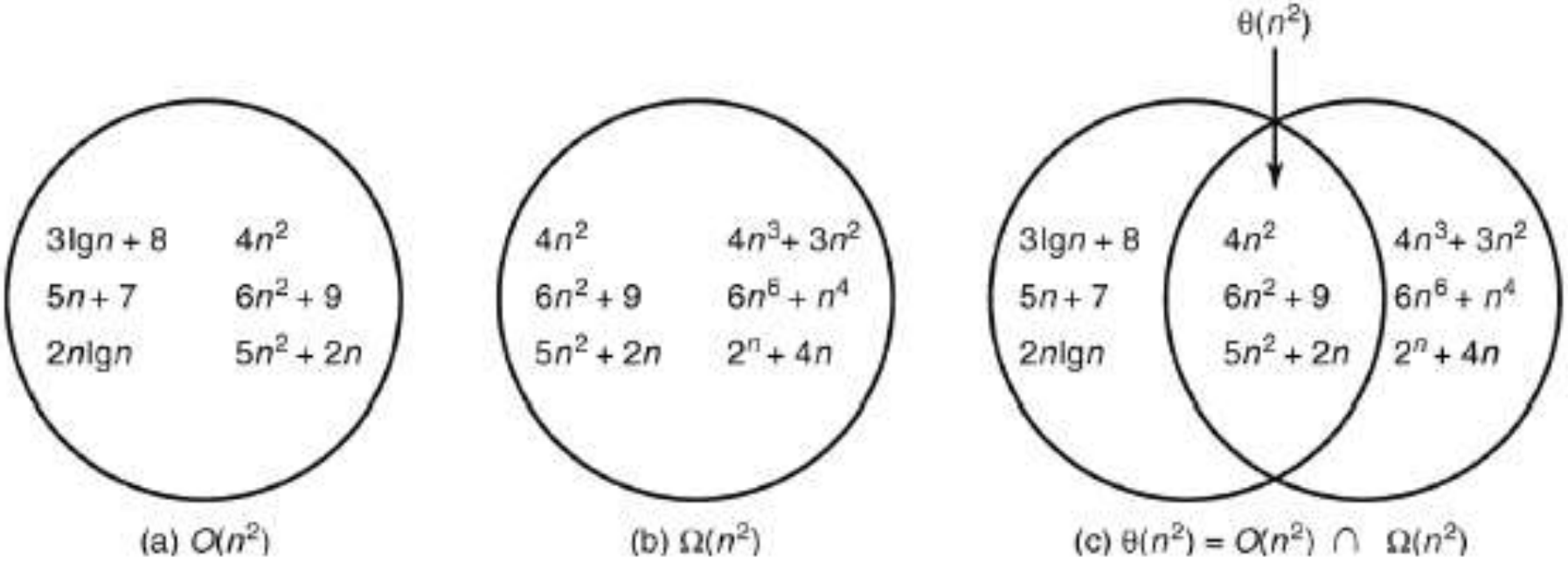


圖 1.6 這裡顯示的是 $\Omega(n^2)$、$O(n^2)$、$\Theta(n^2)$ 三個集合，某些集合中的成員被標示在上面。

不在某一量級中的證明

- 在圖1.6(c)中，函數$5n+7$並不在$\Omega(n^2)$中，而$4n^3+3n^2$也不在$O(n^2)$中，因此，這兩個函數都不在$\Theta(n^2)$中。雖然直觀上來看，這樣的說法是對的，但是我們尚未證明它。下列的例子顯示證明的過程。

- **範例1.12**

我們將使用反證法來證明n不在$\Omega(n^2)$中

假設$n \in \Omega(n^2)$—接著，我們做一些運算，推導出這件事情是錯誤的。

假定n在$\Omega(n^2)$中代表存在著某個正值常數c，以及某個非負整數N，使得對於所有的$n \geq N$，

$$n \geq cn^2$$

如果把不等式的兩邊同除以cn，可以得到，對於$n \geq N$

$$\frac{1}{c} \geq n$$

然而，對於任一個$n > 1/c$，這個不等式就不成立了，這代表著這個不等式並不是對所有的$n \geq N$都成立的。這個矛盾證明了n不在$\Omega(n^2)$中。

有關量級的重要性質

1. $g(n) \in O(f(n))$ 若且唯若 $f(n) \in \Omega(g(n))$
2. $g(n) \in \Theta(f(n))$ 若且唯若 $f(n) \in \Theta(f(n))$
3. 若 $b>1$ 且 $a>1$，則 $\log_a n \in \Theta(\log_b n)$

   這表示所有的對數複雜度函數都在同一個複雜度類別中。我們將會用 $\Theta(\lg n)$ 來代表這個類別。

4. 若 $b>a>0$，則

$$a^n \in o(b^n)$$

   此表示所有指數函數並不在同一個複雜度類別中。

有關量級的重要性質

5. 對於所有的 $a>0$

$$a^n \in o(n!)$$

暗指了 $n!$ 比任一個指數複雜度函數都要差。

6. 考慮下列的複雜度類別的量級：

$$\Theta(\lg n) \quad \Theta(n) \quad \Theta(n \lg n) \quad \Theta(n^2) \quad \Theta(n^j) \quad \Theta(n^k) \quad \Theta(a^n) \quad \Theta(b^n) \quad \Theta(n!)$$

其中 $k>j>2$ 且 $b>a>1$。若一個複雜度函數 $g(n)$ 所在的類別在 $f(n)$ 所在類別的左方，那麼

$$g(n) \in o(f(n))$$

7. 若 $c \geq 0$，$d>0$，$g(n) \in O(f(n))$ 且 $h(n) \in \Theta(f(n))$，那麼

$$c \times g(n) + d \times h(n) \in \Theta(f(n))$$

- **範例 1.21**

性質 3 說明所有的對數複雜度函數都在同一個複雜度類別中。例如

$$\Theta(\log_4 n) = \Theta(\lg n)$$

這代表的是 $\log_4 n$ 之於 $\lg n$ 的關係等同於與 $7n^2 + 5n$ 之於 n^2 的關係。

- **範例 1.22**

性質 6 說明任一對數函數最終都會比任一多項式函數為佳，任一個多項式函數最終都會比任一指數函數好，而任一指數函數最終都會比任一階乘函數好，例如：

$$\lg n \in o(n), \qquad n^{10} \in o(2^n), \qquad \text{及} \qquad 2^n \in o(n!)$$

- **範例 1.23**

性質 6 與 7 可被重複地使用。例如，我們可以用下面的方式，證明 $5n+3\lg n+10n\lg n+7n^2 \in \Theta(n^2)$。重複地應用性質 6 與 7，我們得到

$$7n^2 \in \Theta(n^2)$$

此表示

$$10n\lg n + 7n^2 \in \Theta\left(n^2\right)$$

此表示

$$3\lg n + 10n\lg n + 7n^2 \in \Theta\left(n^2\right)$$

此表示

$$5n + 3\lg n + 10n\lg n + 7n^2 \in \Theta\left(n^2\right)$$

總結

- 若我們能夠得到某個演算法的精確時間複雜度，我們就可以把低次項丟棄，以求得該演算法的量級
- 假設對於某個演算法我們無法算出精確的$T(n)$(或$W(n)$、$A(n)$、或$B(n)$)。若我們能利用前面的定義去證明

  $$T(n) \in O(f(n)) \quad \text{且} \quad T(n) \in \Omega(f(n))$$

  便可以推論出$T(n) \in (f(n))$

總結

- 在某些情況下，證明$T(n) \in O(f(n))$極為容易，但是證明$T(n) \in \Omega(f(n))$卻很困難。在這樣的情況下我們只好退而求其次證明$T(n) \in O(f(n))$
- 有些作者會使用

  $f(n) = \Theta(n^2)$來取代$f(n) \in \Theta(n^2)$
- 同理

  $f(n) = O(n^2)$來取代$f(n) \in O(n^2)$