Chapter 1 演算法分析

- 1.1 演算法
- 1.2 Big-O

1.1 演算法

- 演算法（Algorithms）是一解決問題（problems）的有限步驟程序。
- 舉例來說，現有一問題爲：在一已排序的整數陣列s 中，判斷是否有x，其演算法爲：
  從s串列的第一個元素開始，依序比較，直到x被找到或s串列已達盡頭。假使x被找到，則印出Yes；否則，印出No。

1.1 演算法

- 當問題很複雜時，上述敘述性的演算法就難以表達出來。因此，演算法大都以類似的程式語言表達之，繼而利用您所熟悉的程式語言執行之。

1.1 演算法

- 程式的效率(efficiency)如何，一般是利用 Big-O 來評估。
- 如何求得Big-O 呢？首先必須求出函數內主体敘述的執行次數，再將這些執行次數加總起來成爲一多項式，之後取其最高次方項，即爲Big-O。

1.1 演算法

- 陣列元素相加（Add array members）

Java 片段程式：陣列元素相加	執行次數
`public static int sum(int arr[], int n)`	
`{`	
`  int  i, total=0;`	1
`  for (i=0; i<n; i++)`	n+1
`  total += arr[i];`	n
`  return total;`	1
`}`	2n+3

1.1 演算法

- 矩陣相加 (n*n)

Java 片段程式：矩陣相加	執行次數

`public static void add(int a[][],int b[][],int c[][],int m,int n)`
`{`
`  for (int i=0; i < n; i++)`
`  for (int j=0; j < n; j++)  c[i][j] =`
`  a[i][j] + b[i][j]`
`}`
	n+1
	n(n+1)
	n^2
	$2n^2+2n+1$

1.1 演算法

- 矩陣相乘（Matrix Multiplication）

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae+bg & af+bh \\ ce+dg & cf+dh \end{bmatrix}$$

1.1 演算法

Java 片段程式：矩陣相乘	執行次數
`public static void mul(int a[][],int b[][],int c[][],int n)`	
`{`	1
`  int  i, j, k, sum;`	n+1
`  for (i = 0; i < n; i++)`	n(n+1)
`  for (j = 0; j < n; j++ ){`	n^2
`  sum = 0;`	$n^2(n+1)$
`  for ( k = 0; k < n; k++ )`	n^3
`  sum = sum + a[i][k] * b[k][j];`	n^2
`  c[i][j] = sum;`	
`  }`	$2n^3+4n^2+2n+2$
`}`	

1.1 演算法

- 循序搜尋（Sequential search）

Java 片段程式：循序搜尋	執行次數
`public static int search(int data[],int target, int n)`	
`{`	
`  int  i;`	1
`  for (i = 0; i < n; i++)`	n+1
`  if (target == data[i])`	n
`  return i;`	1
`}`	
	2n+3

1.2 Big-O

- 如何去計算演算法所需要的執行時間呢？在程式或演算法中，每一敘述（statement）的執行時間爲：
  - 此敘述執行的次數，
  - 每一次執行所需的時間，兩者相乘即爲此敘述的執行時間。
- 由於每一敘述所須的時間必需實際考慮到機器和編譯器的功能，因此通常只考慮執行的次數而已。

1.2 Big-O

- 算完程式敘述的執行次數後，通常利用Big-O來表示此程式的時間複雜度。

Big-O 的定義如下：

f(n)=O(g(n))，若且唯若存在一正整數 c 及 n_0，使得 $f(n) \le cg(n)$，對所有的 n，$n \ge n_0$。

1.2 Big-O

- 請看下列範例：

(a) $3n+2=O(n)$，∵我們可找到 $c=4$，$n_0=2$，使得 $3n+2 \le 4n$

(b) $10n^2+5n+1=O(n^2)$，∵我們可以找到 $c=11$，$n_0=6$ 使得 $10n^2+5n+1 \le 11n^2$

(c) $7*2^n+n^2+n=O(2^n)$，∵我們可以找到 $c=8$，$n_0=5$ 使得 $7*2^n+n^2+n \le 8*2^n$

(d) $10n^2+5n+1=O(n^3)$，這可以很清楚的看出，原來 $10n^2+5n+1 \in O(n^2)$，而 n^3 又大於 n^2，理所當然 $10n^2+5n+1=O(n^3)$是沒問題的。同理也可以得知 $10n^2+5n+1 \neq O(n)$，∵f(x)沒有小於等於 $c*g(n)$。

1.2 Big-O

- 其實，我們可以加以證明，當

  $f(n)=a_m n^m +...+a_1 n+a_0$ 時，$f(n)=O(n^m)$

▸▸▸【證明】

$$f(n) \le \sum_{i=0}^{m} |a_i| n^i$$

$$\le n^m * \sum_{i=0}^{m} |a_i| n^{i-m}$$

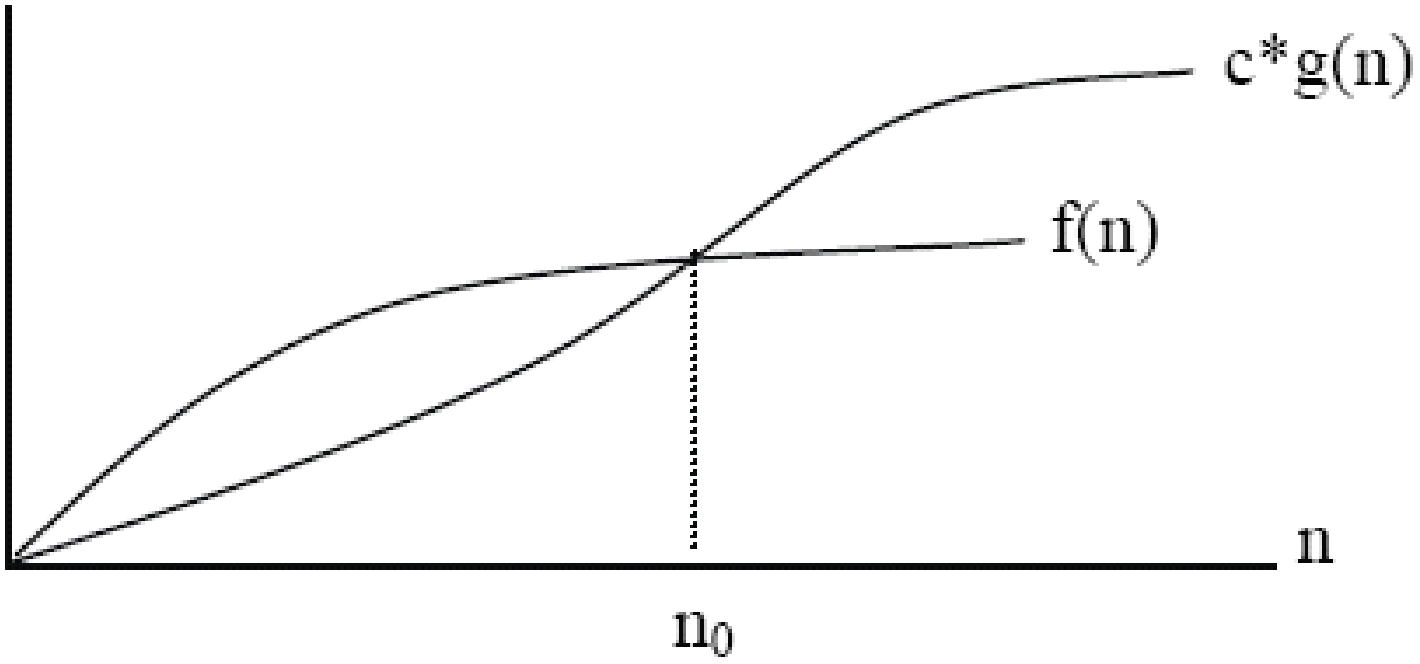
$$\le n^m * \sum_{i=0}^{m} |a_i|$$，對 $n \ge 1$ 而言

⇨ $f(n) \in O(n^m)$，∵可將 $\sum_{i=0}^{m} |a_i|$ 視為 c，而 n^m 為 g(n)

亦即 Big-O 乃取其最大指數的部份即可，因此前述的範例中，陣列元素相加的 Big-O 為 O(n)，矩陣相加 Big-O 為 $O(n^2)$，而矩陣相乘的 Big-O 為 $O(n^3)$。

1.2 Big-O

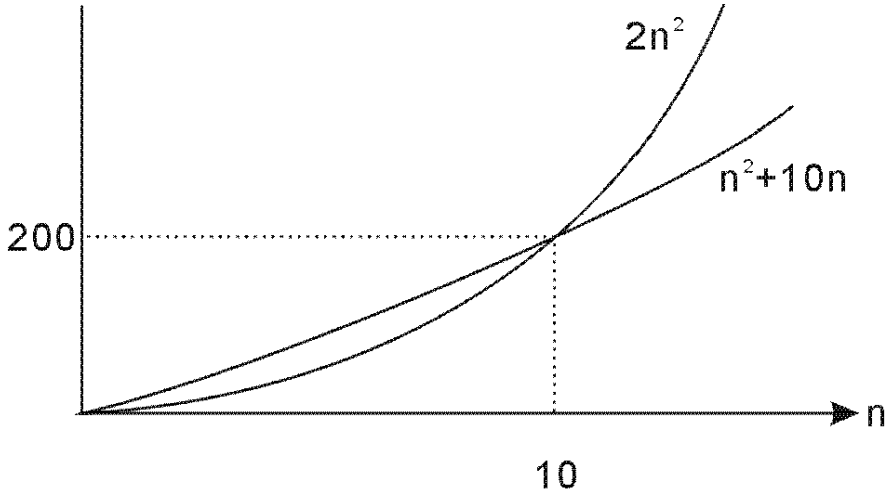
- Big-O的圖形表示



1.2 Big-O

- 例如有一程式的執行次數爲n^2+10n，則其Big-O爲n^2，表示此程式執行的時間最壞的情況下不會超過n^2，因爲

  $n^2+10n \leq 2n^2$，當$c=2$，$n \geq 10$時



1.2 Big-O

- 一般常見的Big-O有幾種類別：

BigO	類別
O(1)	常數時間 (constant)
O(log n)	對數時間 (logarithmic)
O(n)	線性時間 (linear)
O(n log n)	對數線性時間 (log linear)
$O(n^2)$	平方時間 (quadratic)
$O(n^3)$	立方時間 (cubic)
$O(2^n)$	指數時間 (exponential)
O(n!)	階層時間 (factorial)
$O(n^n)$	n 的 n 次方時間

一般而言，這幾種類別由 O(1)，O(log n) ，…，O(n!)，$O(n^n)$之效率按照排列的順序愈來愈差，也可以下一種方式表示。

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) < O(n!) < O(n^n)$

指數

- 在很久的時候，有位勇士救了國王的女兒，國王想獎勵他，就說，你可以提一個願望，我會滿足你。這位勇士對國王說：我的要求很小，請您拿一個棋盤，在第1個格子裡放1粒米，在第2個格子裡放2粒米，第3個格子裡放4粒米…. ，以此類推，每到下一個格子就增加一倍，直到將這六十四個格子全部放完。這就是我的要求。

指數

- 原來仔細一數才知道，米的總數是 $2^{64}-1=18446744073709151616$。
  假設世界上有50億人口，每人每天吃三頓飯，每天吃掉2萬粒米，一年大概吃了7300000粒，而2的64次方的米，足以讓50億人口吃50年。

1.2 Big-O

- 除Big-O之外，用來衡量效率的方法還有

Ω 的定義如下:

$f(n) = \Omega(g(n))$,若且唯若，存在正整數 c 和 n0，使得 $f(n) \geq cg(n)$，對所有的 n，$n \geq n0$。

請看下面幾個範例：

(a) $3n+2=\Omega(n)$，∵我們可找到 $c=3$，$n_0=1$

使得 $3n+2 \geq 3n$

(b) $200n^2+4n+5=\Omega(n^2)$，∵我們可找到 $c=200$，$n_0=1$

使得 $200n^2+4n+5 \geq 200\ n^2$

(c) $10n^2+4n+2=\Omega(n)$，為什麼呢?

∵ 從定義得知 $10n^2+4n+2=\Omega(n^2)$，

由於 $n^2 > n$，∴理所當然 $10n^2+4n+2$ 也可是為$\Omega(n)$。

1.2 Big-O

- 除Big-O之外，用來衡量效率的方法還有

Θ的定義如下：

f(n)= Θ (g(n)),若且唯若，存在正整數 c1，c2 及 n，使得 c1*g(n) ≤ f(n) ≤ c2*g(n)，對所有的 n，n ≥ n0。

我們以下面幾個範例加以說明：

(a) $3n+1=\Theta(n)$，∵我們可以找到 $c_1=3$，$c_2=4$，且 $n_0=2$，

使得 $3n \le 3n+1 \le 4n$

(b) $10n^2+4n+6=\Theta(n^2)$，∵只要 $c_1=10$，$c_2=11$ 且 $n_0=10$

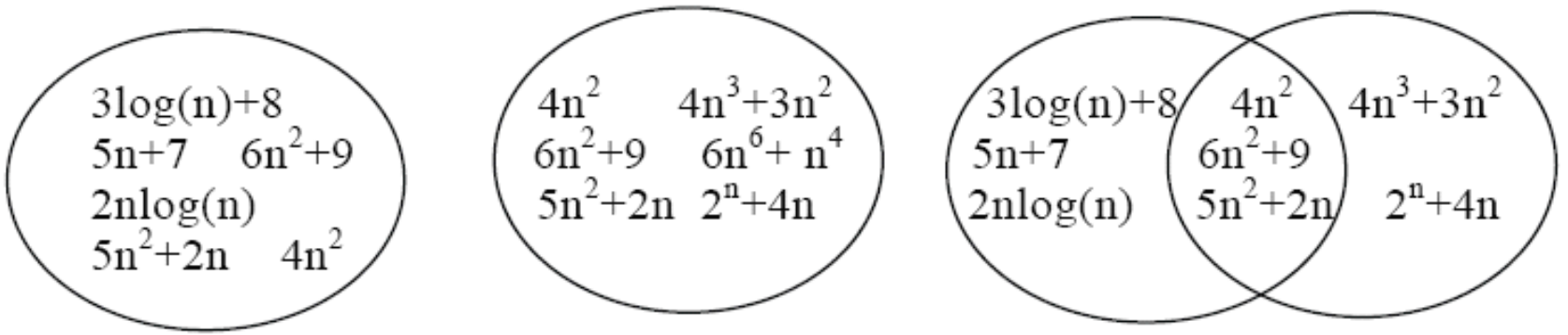
便可得 $10n^2 \le 10n^2+4n+6 \le 11n^2$

(c) 注意! $3n+2 \neq \Theta(n^2)$，$10n^2+n+1 \neq \Theta(n)$

讀者可加以思考一下下。

1.2 Big-O

下圖為 Big-O, Ω, Θ 的表示情形：



(a) $O(n^2)$　　(b) $\Omega(n^2)$　　(c) $\Theta(n^2)$，只有中間交集部份

1.2 Big-O

- 循序搜尋（sequential search）的情形可分爲三種：Worst, Best, Average，其平均搜尋到的次數爲

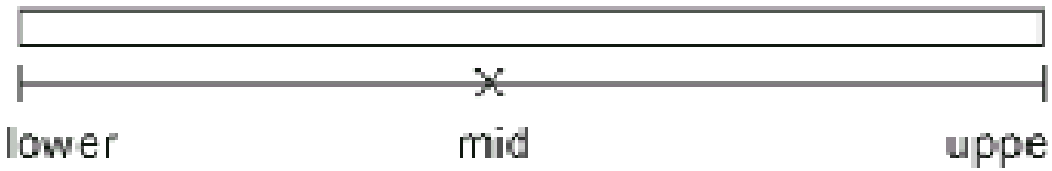
$$\sum_{k=1}^{n}\left(k\times\frac{1}{n}\right)=\frac{1}{n}\times\sum_{k=1}^{n}k=\frac{1}{n}(1+2+\ldots+n)=\frac{1}{n}\times\frac{n(n+1)}{2}=\frac{n+1}{2}$$

因此循序搜尋的 Big-O 爲 O(n)。

1.2 Big-O

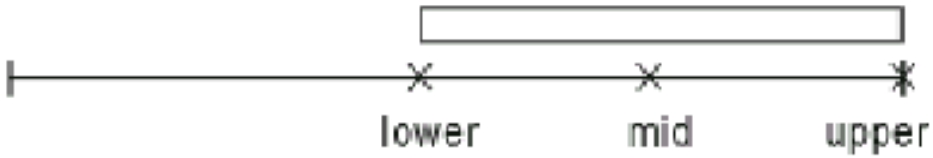
- 二元搜尋法

  乃是資料已經皆排序好，因此由中間（mid）開始比較，便可知欲搜尋的資料（key）落在mid的左邊還是右邊，再將左邊的中間拿出來與key相比，只是每次要調整每個段落的起始位址或最終位址。

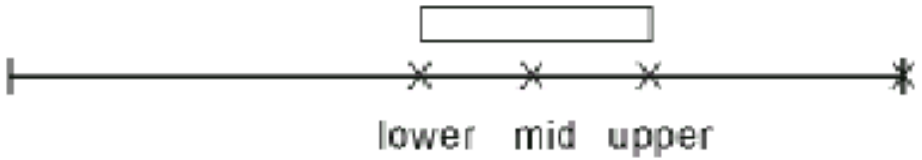


1.2 Big-O

當 key>data[mid]時， mid=(lower+upper)/2 ，則 lower=mid+1 ， upper 不變，如下所示：



當 key<data[mid]時，則 upper=mid-1 ， lower 不變：



1.2 Big-O

Java 片段程式：二元搜尋法

```
public static void binsrch(int A[], int n, int x, int j)
{
   lower = 1;
   upper = n;
   while(lower <= upper) {
       mid = (lower + upper) / 2;
       if(x > A[mid])
          lower = mid + 1;
       else if(x < A[mid])
          upper = mid – 1;
       else {
          j = mid;
          System.out.println("Found, "+ x +" is #"+ mid +"record.");
       }
   }
}
```

1.2 Big-O

- 二元搜尋法搜尋的次數最差爲logn+1，此處的log表示$\log_2$。因此當資料量爲128時，其搜尋的次數爲log128+1=8。

陣列大小	二元搜尋	循序搜尋
128	8	128
1,024	11	1,024
1,048,576	21	1,048,576
4,294,967,296	33	4,294,967,296

1.2 Big-O

- 一個有趣的例子
- 費氏數列（Fibonacci number），其定義如下：

$$f_0=0$$
$$f_1=1$$
$$f_n=f_{n-1}+f_{n-2} \quad \text{for } n \geq 2$$

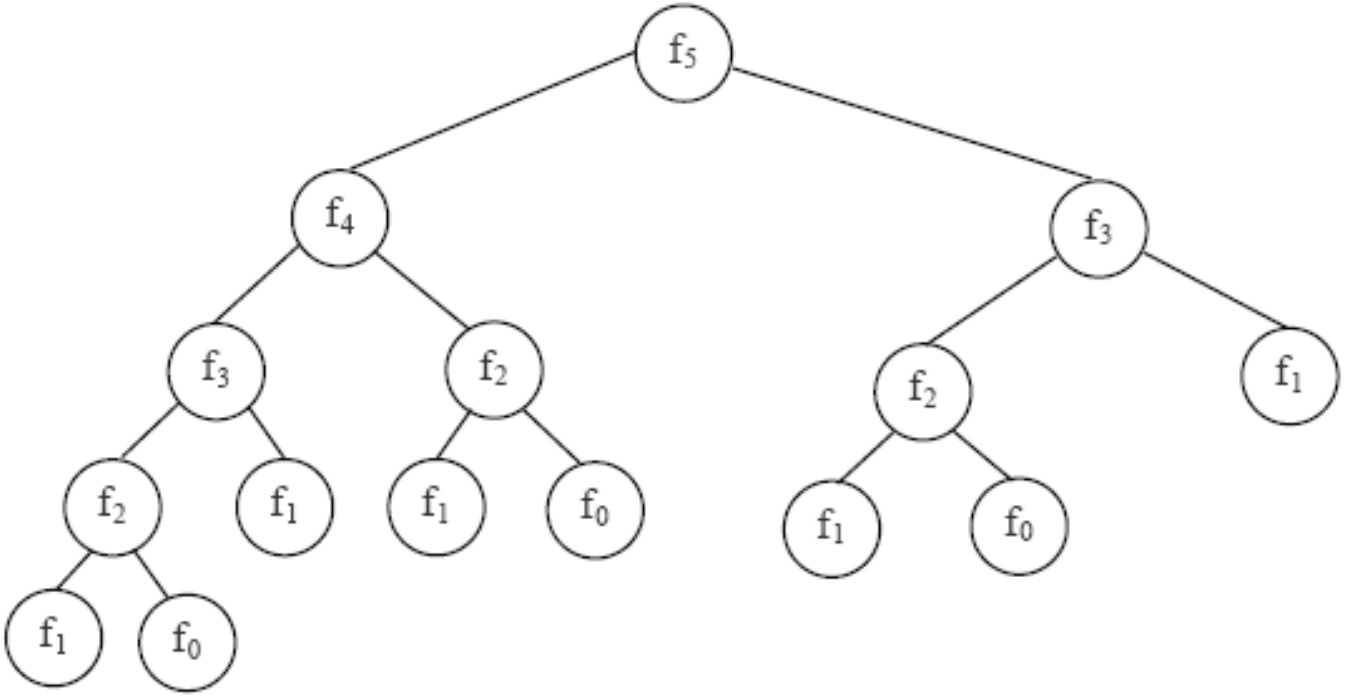
因此

$$f_2=f_1+f_0=1$$
$$f_3=f_2+f_1=1+1=2$$
$$f_4=f_3+f_2=2+1=3$$
$$f_5=f_4+f_3=3+2=5$$
$$\vdots$$

依此類推

1.2 Big-O

若以遞迴的方式進行計算的話，其圖形如下：



1.2 Big-O

因此可得

n（第n項）	需計算的項目數
0	1
1	1
2	3
3	5
4	9
5	15
6	25

1.2 Big-O

- 當n=3(f_3)從上圖可知需計算的項目爲5；n=5時，需計算的項目數爲15個。因此我們可以下列公式表示：

$$T(n) > 2 * T(n-2)$$
$$> 2 * 2 * T(n-4)$$
$$> 2 * 2 * 2 * T(n-6)$$
$$\vdots$$
$$> \underbrace{2 * 2 * 2 * 2 * \ldots * 2}_{n/2 \text{ 次}} * T(0)$$

當 $T(0)=1$ 時，$T(n)>2^{n/2}$，此時的 n 必須大於等於 2，因爲當 n=1

$T(1)=1<2^{n/2}$

1.2 Big-O

Java 片段程式：以遞迴方式計算費氏數列

```
public static int Fibonacci(int n)
{
   if (n==0)
      return 0;
   else if (n==1)
      return 1;
   else
      return (Fibonacci(n-1) + Fibonacci(n-2));
}
```

1.2 Big-O

Java 片段程式：以非遞迴方式計算費氏數列

```
public static int Fibonacci(int n)
{
   int prev1, prev2, item, i;
   if (n == 0)
      return 0;
   else  if (n == 1)
      return 1;
   else {
      prev2 = 0;
      prev1 = 1;
      for ( i = 2; i <= n; i++){
         item = prev1 + prev2;
         prev2 = prev1;
         prev1 = item;
      }
      return item;
   }
}
```

1.2 Big-O

計算第 n 項的費氏數列值	遞　　迴		非　遞　迴	
n	所計算的項目 $(2^{n/2})$	所需執行時間	所計算的項目 (n+1)	所需執行時間
40	1,048,576	1048μs	41	41ns
60	$1.1* 10^{8}$	1s	61	61ns
80	$1.1* 10^{12}$	18min	81	81ns
100	$1.1* 10^{15}$	13 天	101	101ns
200	$1.3* 10^{30}$	$4*10^{13}$ 年	201	201ns
1 ns = 10^{-9} second　　1 μs = 10^{-6} second				

程式練習

- 試比較上述兩種費氏數列程式所需執行時間。