CHAPTER 1

BASIC CONCEPT

龔旭陽　特聘教授

國立屏東科技大學　資訊管理系

資料結構 (Data Structure)

- 如何讓程式執行速度很快？
  - 必須使用一個高執行速度的電腦？
  - 錯！
- 一個好的程式設計才是最具決定性的一個因素
- 資料結構
  - 程式設計的技術與方法
  - 寫出好程式
  - 正確、執行速度較快、使用記憶體較少

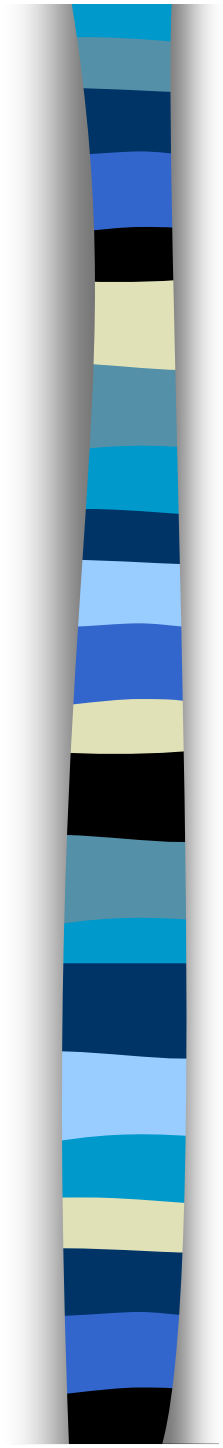
程式設計＝資料表達方式＋程式方法(演算法)

- 好的程式 ＝

  好的資料表達方式 (Data Object) ＋

  好的程式方法(演算法，Algorithm)

- 演算法：處理資料的方法
- 以程式語言實現程式設計
  - 程式語言：C、C++、Java



How to create programs

- Requirements
- Analysis: bottom-up vs. top-down
- Design: data objects and operations
- Refinement and Coding
- Verification
  - Program Proving
  - Testing
  - Debugging

程式設計的過程-階段

- 程式設計是將需要解決的問題轉換成程式碼，程式碼不只能夠在電腦上正確的執行，而且可以驗證程式執行的正確性，程式設計的過程可以分成5個階段，如下所示：
  - **需求（Requirements）**
  - **分析（Analysis）**
  - **設計（Design）**
  - **撰寫程式碼（Coding）**
  - **驗證（Verification）**

程式設計的過程–需求

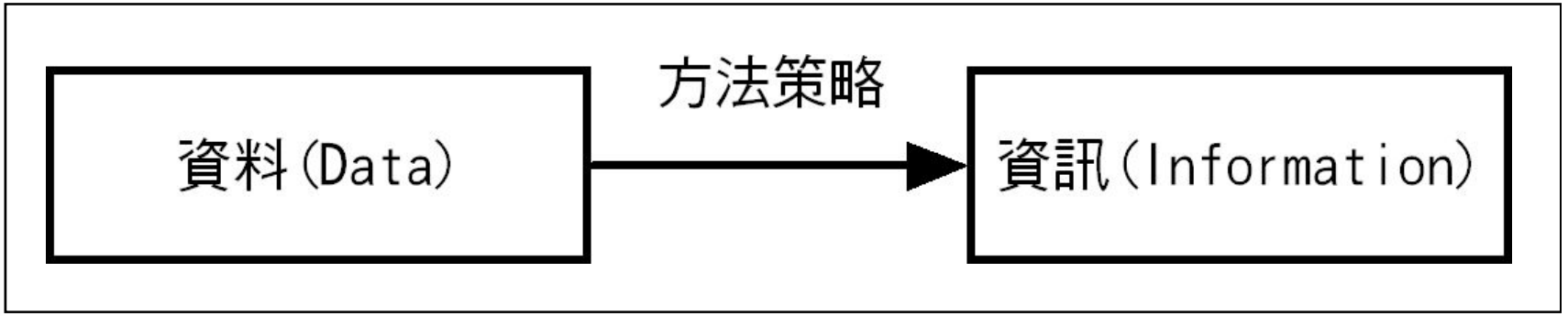
- **需求（Requirements）**：程式設計的需求是在了解問題本身，以便確切獲得程式需要輸入的資料和其產生的結果，如下圖所示：

程式設計的過程-分析

- **分析（Analysis）**：在解決需求時，只有一種解決方法嗎？例如：如果有100個變數，我們可以宣告100個變數來儲存資料，或是使用陣列來儲存，在分析階段是將所有可能解決問題的演算法都寫下來，然後分析比較那一種方法比較好，選擇最好的演算法來撰寫程式。

程式設計的過程-設計

- 設計（**Design**）：在了解程式設計的需求後，我們就可以開始找尋解決問題的方法和策略，簡單的說，設計階段就是找出解決問題的步驟，如下圖所示：



程式設計的過程-撰寫

- **撰寫程式碼（Coding）**：現在就可以開始使用指定的程式語言來撰寫程式碼，以本書為例是使用C程式語言，在實際撰寫程式時，可能發現另一種方法比較好，因為設計歸設計，在實際撰寫程式時才能真正發現其優劣，如果這是一個良好的設計方法，就算改為其它方法也不會太困難。

程式設計的過程–驗證

- **驗證（Verification）**：驗證是證明程式執行的結果符合需求的輸出資料，在這個階段可以再細分成三個部分：
  - **證明**：執行程式時需要證明它的執行結果是正確的，程式符合所有輸入資料的組合，程式規格也都符合演算法的需求。
  - **測試**：程式需要測試各種可能情況、條件和輸入資料，以測試程式執行無誤，如果有錯誤產生，就需要除錯來解決程式問題。
  - **除錯**：如果程式無法輸出正確的結果，除錯是在找出錯誤的地方，我們不但需要找出錯誤，還需要決定如何更正它。

分析 (Analysis)

演算法 (Algorithm) -定義

- 演算法是完成目標工作的一組指令，這組指令的步驟是有限的。除此之外，演算法還必須滿足一些條件，如下所示：
  - **輸入(Input)**：沒有或數個外界的輸入資料。
  - **輸出(Output)**：至少有一個輸出結果。
  - **明確性(Definiteness)**：每一個指令步驟都十分明確，沒有模稜兩可。
  - **有限性(Finiteness)**：這組指令一定結束。
  - **有效性(Effectiveness)**：每一個步驟都可行，可以追蹤其結果。

演算法 - 表達(描述)方法

- 演算法只是將解決問題步驟詳細的寫出來，所以並沒有固定的方式，基本上只要能夠描述這組指令的執行過程即可，常用的方式如下所示：
  - **一般語言文字**：直接使用文字描述來說明執行的步驟。
  - **虛擬碼（Pseudo Code）**：趨近程式語言的描述方法，其每一列約可轉換成一列程式碼。
  - **流程圖（Flow Chart）**：使用結構化的圖表描述執行過程，以各種不同形狀的圖形表示不同的操作。

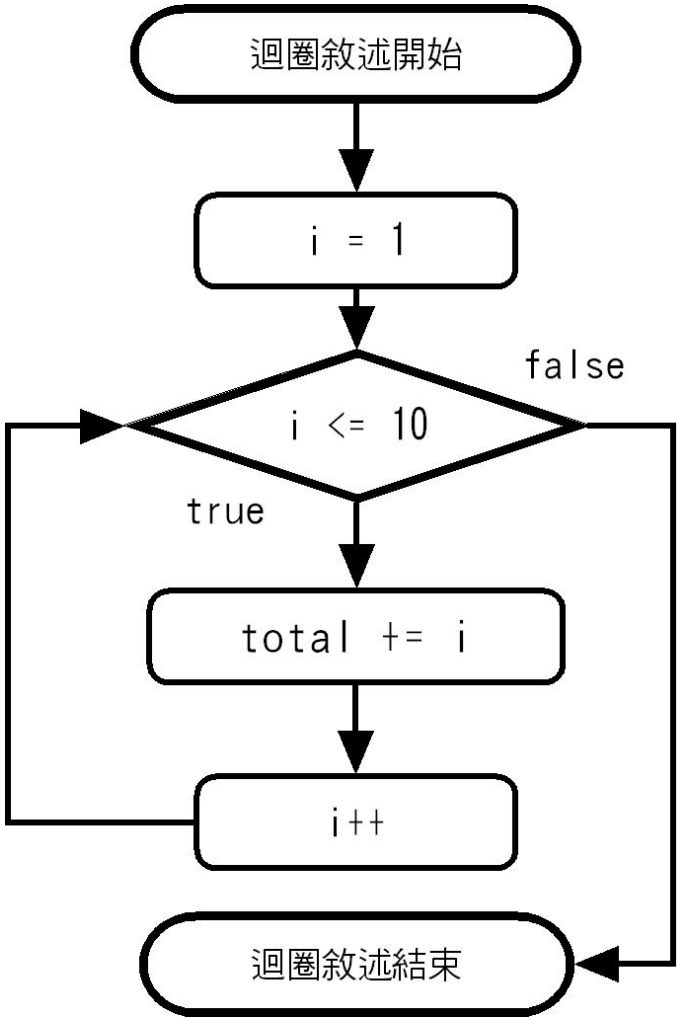
演算法 - 虛擬碼範例

- 從1加到10演算法的虛擬碼，如下所示：

```
/* 計算1加到10 */
Let counter = 1
Let total = 0
while counter <= 10
   total = total + counter
   Add 1 to counter
Output the total    /* 顯示結果 */
```

演算法 - 流程圖範例

- 從1加到10演算法的流程圖，如右圖所示：



Algorithm

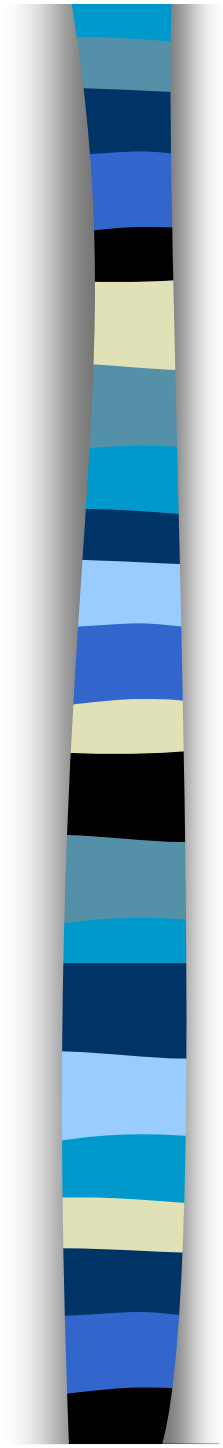
- Definition
  An *algorithm* is a finite set of instructions that accomplishes a particular task.
- Criteria
  - Input: 可以沒有
  - Output: 至少一個
  - definiteness: clear and unambiguous
  - finiteness: terminate after a finite number of steps
  - effectiveness: instruction is basic enough to be carried out

Data Type

- Basic data type of c includes char, int, float and double
- C helps us by providing two mechanisms for grouping data together
- Array：are collections of elements of the same basic data type

  int list[5];
- Structs：are collections of elements whose data type need not to he the same
- Struct students

```
{
        char last_name[40];
        int    student_id;
 }
```



Data Type

- C also provides the pointer data type
  - A pointer is denoted by placing an * before a variables name so

    int i, *pi

    - i is a integer, and pi is a pointer to an integer

Data Type

- Data Type
  A *data type* is a collection of *objects* and a set of *operations* that act on those objects.
- Abstract Data Type
  An *abstract data type(ADT)* is a data type that is organized in such a way that the specification of the objects and the operations on the objects is separated from the representation of the objects and the implementation of the operations.

Specification vs. Implementation

- Operation specification
  - function name
  - the types of arguments
  - the type of the results
- Implementation independent

*Structure 1.1: Abstract data type *Natural_Number* (p.20)

structure Natural_Number is

objects: an ordered subrange of the integers starting at zero and ending at the maximum integer (*INT_MAX*) on the computer

functions:

for all x, y ∈ *Nat_Number*; *TRUE*, *FALSE* ∈ *Boolean*

and where +, -, <, and == are the usual integer operations.

Nat_No Zero ( ) ::= 0

Boolean Is_Zero(x) ::= **if** (x) **return** *FALSE*
else return *TRUE*

Nat_No Add(x, y) ::= **if** ((x+y) <= *INT_MAX*) **return** x+y
else return *INT_MAX*

Boolean Equal(x,y) ::= **if** (x== y) **return** *TRUE*
else return *FALSE*

Nat_No Successor(x) ::= **if** (x == *INT_MAX*) **return** x
else return x+1

Nat_No Subtract(x,y) ::= **if** (x<y) **return** 0
else return x-y

end *Natural_Number*

::= is defined as

作業

- Section 1.4: P21 – ex 1

Measurements (評估)

- How to judge a program? (p18)
  - Does the program meet the original spec. of the task.
  - Does it work correctly?
  - Does the program contain documentation that shows, how to use it and how it works.
  - Does the program effectively use function to create logical units?
  - Does the program's code readable?

Measurements (評估)

- Is the program's running time acceptable for the task?
- Is the program's running memory acceptable for the task?

- Performance Evaluation：Two distinct fields.
  - Performance Analysis：focus on obtaining estimates of time and space that are machine independent
  - use complexity theory.
- Performance Analysis (machine independent)
  - space complexity: storage requirement
  - time complexity: computing time
- Performance Measurement：Focus obtaining machine-dependent running times.

演算法的效率分析

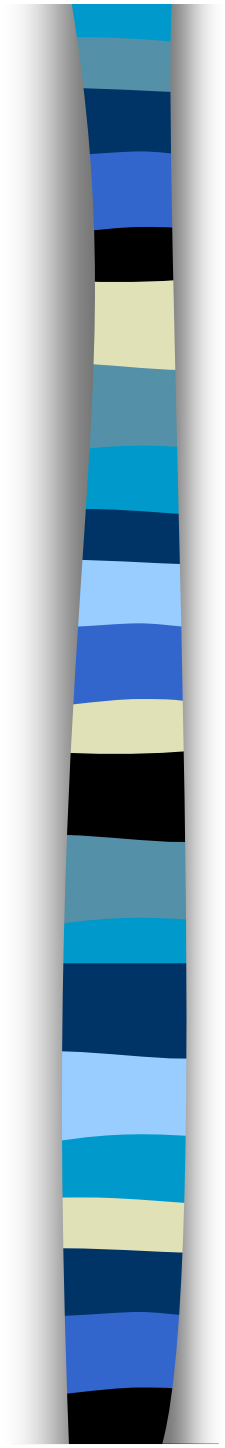
- 什麼是有效率的演算法？電腦學家為此衡量準則提供了客觀的標準—分析演算法的執行時間和記憶體需求。以時間複雜度或空間複雜度來討論演算法的效率
- 解決相同的問題，演算法所用的時間複雜度和空間複雜度愈少愈好。
- 時間複雜度(time complexity)：一個程式或演算法所需的執行時間；
- 空間複雜度(space complexity)：一個程式或演算法所需的記憶體空間。

空間與時間複雜度–
空間複雜度分析

- 空間複雜度是指程式執行時所需的記憶體空間，主要分成兩種，如下所示：
  - **固定記憶體空間**：程式本身、靜態變數和常數等所佔用的記憶體空間，它和程式輸出入的資料量並沒有關係。
  - **動態記憶體空間**：在程式執行過程所需的額外記憶體空間，例如：函數參數、遞迴函數的堆疊空間和程式動態配置的記憶體空間等。它會隨著輸出入的資料量、函數參數個數，遞迴呼叫次數而動態改變所需的記憶體空間。

Measurements

- Criteria
  - Is it correct?
  - Is it readable?
  - …
- Performance Analysis (machine independent)
  - space complexity: storage requirement
  - time complexity: computing time
- Performance Measurement (machine dependent)



Space Complexity
$S(P)=C+S_P(I)$

- Fixed Space Requirements (C)
  Independent of the characteristics of the inputs and outputs
  - instruction space
  - space for simple variables, fixed-size structured variable, constants
- Variable Space Requirements ($S_P(I)$)
  depend on the instance characteristic I
  - number, size, values of inputs and outputs associated with I
  - recursive stack space, formal parameters, local variables, return address

***Program 1.10:** Simple arithmetic function (p.23)

```
float abc(float a, float b, float c)
{
   return a + b + b * c + (a + b - c) / (a + b) + 4.00;
}
```

$S_{abc}(I) = 0$

***Program 1.11:** Iterative function for summing a list of numbers (p.24)

```
float sum(float list[ ], int n)
{
  float tempsum = 0;
  int i;
  for (i = 0; i<n; i++)
     tempsum += list [i];
  return tempsum;
}
```

$S_{sum}(I) = 0$

Recall: pass the address of the first element of the array & pass by value

*__Program 1.12:__ Recursive function for summing a list of numbers (p.24)

```
float rsum(float list[ ], int n)
{
  if (n) return rsum(list, n-1) + list[n-1];
  return 0;
}
```

$S_{sum}(I)=S_{sum}(n)=6n$

Assumptions:

*__Figure 1.1:__ Space needed for one recursive call of Program 1.12 (p.25)

Type	Name	Number of bytes
parameter: float parameter: integer return address:(used internally)	list [ ] n	2 2 2(unless a far address)
TOTAL per recursive call		6

空間與時間複雜度-
時間複雜度分析(時間T_P)

- 至於如何決定時間t，其評量標準如下所示：
  - **執行程式碼使用的電腦種類**：桌上型電腦、工作站或大型電腦。
  - **CPU使用的機器語言指令集**：某些CPU的機器語言指令包含乘法和除法指令，有些沒有，有些以硬體實作，有些是軟體加持。
  - **CPU執行機器語言指令所需的時間**：即CPU的執行速度，每秒可以執行的指令數不同，執行所需的時間當然不同。
  - **使用的編譯程式**：好的編譯程式可以將指令轉譯成為單一機器語言指令，相對於將它轉譯成數個機器語言指令的編譯程式，其執行時間的差異就十分明顯。

空間與時間複雜度–
時間複雜度分析(頻率計數)

- 基於上述原因，執行單位時間t依不同軟硬體，可能造成十分大的差異，因此我們並不會直接計算程式的執行時間，取而代之是計算程式每一列程式碼的執行頻率，也就是「頻率計數」（Frequency Count），以程式執行的次數來代替每一列程式碼實際執行的時間。

頻率計數的計算–說明

- 頻率計數（Frequency Count）是以原始程式碼的每一列可執行指令作為一個執行單位，我們可以計算出程式的執行次數，這個次數就是頻率計數。

頻率計數的計算-種類

種類	頻率計數	說明
註解	0	註解是不可執行的程式碼
宣告程式碼	0	宣告程式碼也是不可執行的程式碼，例如：變數宣告，程式區塊的大括號
指定敘述和運算式	1	不含函數呼叫的指定敘述和運算式
執行函數	1	函數執行次數是 1，遞迴函數需視其遞迴執行的次數而定
條件敘述	視情況	if、switch 條件敘述的頻率計數是條件運算式本身的判斷次數，和程式區塊中程式碼頻率計數的總和
迴圈敘述	視情況	for、while 和 do/while 迴圈敘述的頻率計數分為控制部分（即迴圈條件）和程式區塊，如果迴圈執行 n 次，則控制部分為 n+1 次（多一次結束迴圈的判斷），程式區塊是 n 次

Time Complexity

$$T(P)=C+T_P(I)$$

- Compile time (C)
  independent of instance characteristics
- run (execution) time T_P
- Definition

  $$T_P(n)=c_aADD(n)+c_sSUB(n)+c_lLDA(n)+c_{st}STA(n)$$

  A *program step* is a syntactically or semantically meaningful program segment whose execution time is independent of the instance characteristics.
- Example
  - abc = a + b + b * c + (a + b - c) / (a + b) + 4.0
  - abc = a + b + c

Regard as the same unit
machine independent

Methods to compute the step count

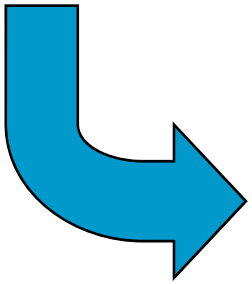
- Introduce variable count into programs
- Tabular method
  - Determine the total number of steps contributed by each statement
    step per execution × frequency
  - add up the contribution of all statements

演算法的執行次數

程式1-4 陣列元素加總

```
int Sum(int data[],int n)
{  int summation=0;
  for (int i=0; i<n; i++) summation += data[i];
   return summation;
}
```

程式1-4將傳入整數陣列data 中的n個元素(data[0]~data[n-1])，總加至整數變數summation中傳出。



在程式1-4中加入一全域變數count，來加總所有指令執行的次數；新的程式將如程式1-5所示。

程式1-5 陣列元素加總並計算所有指令執行的次數

```
int count = 0;                          // 全域變數宣告
int Sum(int data[],int n)
{  int summation=0;
   count++;                             // 計算宣告 int 指令的執行
   for (int i=0; i<n; i++)
   {    count++;                        // 計算 for 指令的執行次數
        summation += data[i];
        count++;                        // 計算 = 指令的執行次數;
   }
   count++;                             // 計算最後一次 for 指令的執行
   count++;                             // 計算 return 指令的執行
   return summation;
}
```

1.5.1 演算法的執行次數(續)

程式1-6 計算陣列元素加總所有指令執行的次數

```
int Sum(int data[],int n)
{ count++;              // 計算宣告 int 指令的
  執行
  for (int i=0; i<n; i++)
     count += 2
  count += 2;
}
```

程式1-6 只保留程式1-5 的主體和加總count 的指令。

此程式的執行總次數為$2n+3$，其中 for 迴圈每執行一次，count會計數兩次；而迴圈共計有n次，所以for迴圈內即執行$2n$ 次。

Iterative summing of a list of numbers

***Program 1.13:** with count statements (p.26)

```
float sum(float list[ ], int n)
{
   float tempsum = 0; count++; /* for assignment */
   int i;
   for (i = 0; i < n; i++) {
        count++;              /*for the for loop */
        tempsum += list[i]; count++;  /* for assignment */
   }
   count++;         /* last execution of for */
   return tempsum;
   count++;         /* for return */
}
```

2n + 3 steps

***Program 1.14:** Simplified version of Program 1.13(p.27)

```
float sum(float list[ ], int n)
{
   float tempsum = 0;
   int i;
   for (i = 0; i < n; i++)
       count += 2;
   count += 3;
   return 0;
}
```

2n + 3 steps

Recursive summing of a list of numbers

***Program 1.15:** Program 1.12 with count statements added (p.27)

```
float rsum(float list[ ], int n)
{
        count++;        /*for if conditional */
        if (n) {
                count++;  /* for return and rsum invocation */
                return rsum(list, n-1) + list[n-1];
        }
        count++;
        return list[0];
}
```

N=0→ 2 次 (if 及 second return)
N>0 → 2n 次 (if 及 first return)
Total = 2n+2

Matrix addition

***Program 1.16:** Matrix addition (p.28)

```
void add( int a[ ] [MAX_SIZE], int b[ ] [MAX_SIZE],
                    int c [ ] [MAX_SIZE], int rows, int cols)
{
   int i, j;
   for (i = 0; i < rows; i++)
      for (j= 0; j < cols; j++)
        c[i][j] = a[i][j] +b[i][j];
 }
```

***Program 1.17:** Matrix addition with count statements (p.29)

```
void add(int a[ ][MAX_SIZE], int b[ ][MAX_SIZE],
                    int c[ ][MAX_SIZE], int row, int cols )
{
  int i, j;
  for (i = 0; i < rows; i++){
      count++; /* for i for loop */
      for (j = 0; j < cols; j++) {
        count++; /* for j for loop */
        c[i][j] = a[i][j] + b[i][j];
        count++; /* for assignment statement */
      }
      count++;    /* last time of j for loop */
  }
  count++;        /* last time of i for loop */
}
```

2rows * cols + 2 rows + 1

***Program 1.18:** Simplification of Program 1.17 (p.29)

```
void add(int a[ ][MAX_SIZE], int b [ ][MAX_SIZE],
                    int c[ ][MAX_SIZE], int rows, int cols)
{
   int i, j;
   for( i = 0; i < rows; i++) {
      for (j = 0; j < cols; j++)
         count += 2;
         count += 2;
   }
   count++;
}
```

時間複雜度： 2rows × cols + 2rows +1

改善建議

Suggestion: Interchange the loops when rows >> cols

Tabular Method

***Figure 1.2:** Step count table for Program 1.10 (p.30)

Iterative function to sum a list of numbers

steps/execution

Statement	s/e	Frequency	Total steps
float sum(float list[ ], int n)	0	0	0
{	0	0	0
float tempsum = 0;	1	1	1
int i;	0	0	0
for(i=0; i <n; i++)	1	n+1	n+1
tempsum += list[i];	1	n	n
return tempsum;	1	1	1
}	0	0	0
Total			2n+3

Recursive Function to sum of a list of numbers

***Figure 1.3:** Step count table for recursive summing function (p.30)

Statement	s/e	Frequency	Total steps
float rsum(float list[ ], int n)	0	0	0
{	0	0	0
if (n)	1	n+1	n+1
return rsum(list, n-1)+list[n-1];	1	n	n
return list[0];	1	1	1
}	0	0	0
Total			2n+2

Matrix Addition

***Figure 1.4:** Step count table for matrix addition (p.31)

Statement	s/e	Frequency	Total steps

Void add (int a[ ][MAX_SIZE] · · ·)
{
int i, j;
for (i = 0; i < row; i++)
for (j=0; j< cols; j++)
c[i][j] = a[i][j] + b[i][j];
}
	0
	0
	0
	1
	1
	1
	0
		0
		0
		0
		rows+1
		rows · (cols+1)
		rows · cols
		0
			0
			0
			0
			rows+1
			rows · cols+rows
			rows · cols
			0

Total			2rows · cols+2rows+1

Exercise 1

***Program 1.18:** Printing out a matrix (p.28)

```
void print_matrix(int matrix[ ][MAX_SIZE], int rows, int cols)
{
  int i, j;
  for (i = 0; i < row; i++) {
    for (j = 0; j < cols; j++)
      printf("%d",  matrix[i][j]);
    printf( "\n");
  }
}
```

2rows * cols + 3rows + 1

Exercise 2

***Program 1.19:Matrix multiplication function**(p.33)

```
void mult(int a[ ][MAX_SIZE], int b[ ][MAX_SIZE], int c[ ][MAX_SIZE])
{
   int i, j, k;
   for (i = 0; i < MAX_SIZE; i++)
      for (j = 0; j< MAX_SIZE;  j++) {
          c[i][j] = 0;
      for (k = 0; k < MAX_SIZE; k++)
            c[i][j]  +=  a[i][k] * b[k][j];
      }
}
```

$2Max\_Size^3+3Max\_Size^2+2Max\_Size+1$

Exercise 3

***Program 1.20:Matrix product function**(p.33)

```
void prod(int a[ ][MAX_SIZE], int b[ ][MAX_SIZE], int c[ ][MAX_SIZE],
                                                    int rowsa, int colsb, int colsa)
{
  int i, j, k;
  for (i = 0; i < rowsa; i++)
    for (j = 0; j< colsb;  j++) {
       c[i][j] = 0;
    for (k = 0; k< colsa; k++)
         c[i][j]  +=  a[i][k] * b[k][j];
    }
}
```

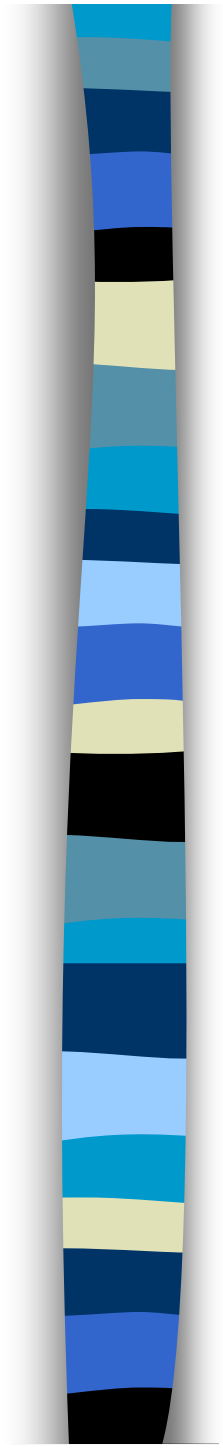
2rowsa * colsb*colsa + 3rowsa * colsb + 2rowsa + 1

Exercise 4

***Program 1.21:Matrix transposition function** (p.34)

```
void transpose(int a[ ][MAX_SIZE])
{
   int i, j, temp;
   for (i = 0; i < MAX_SIZE-1; i++)
      for (j = i+1; j < MAX_SIZE;  j++)
         SWAP (a[i][j], a[j][i], temp);
}
```

$Max\_Size^2+Max\_Size-1$



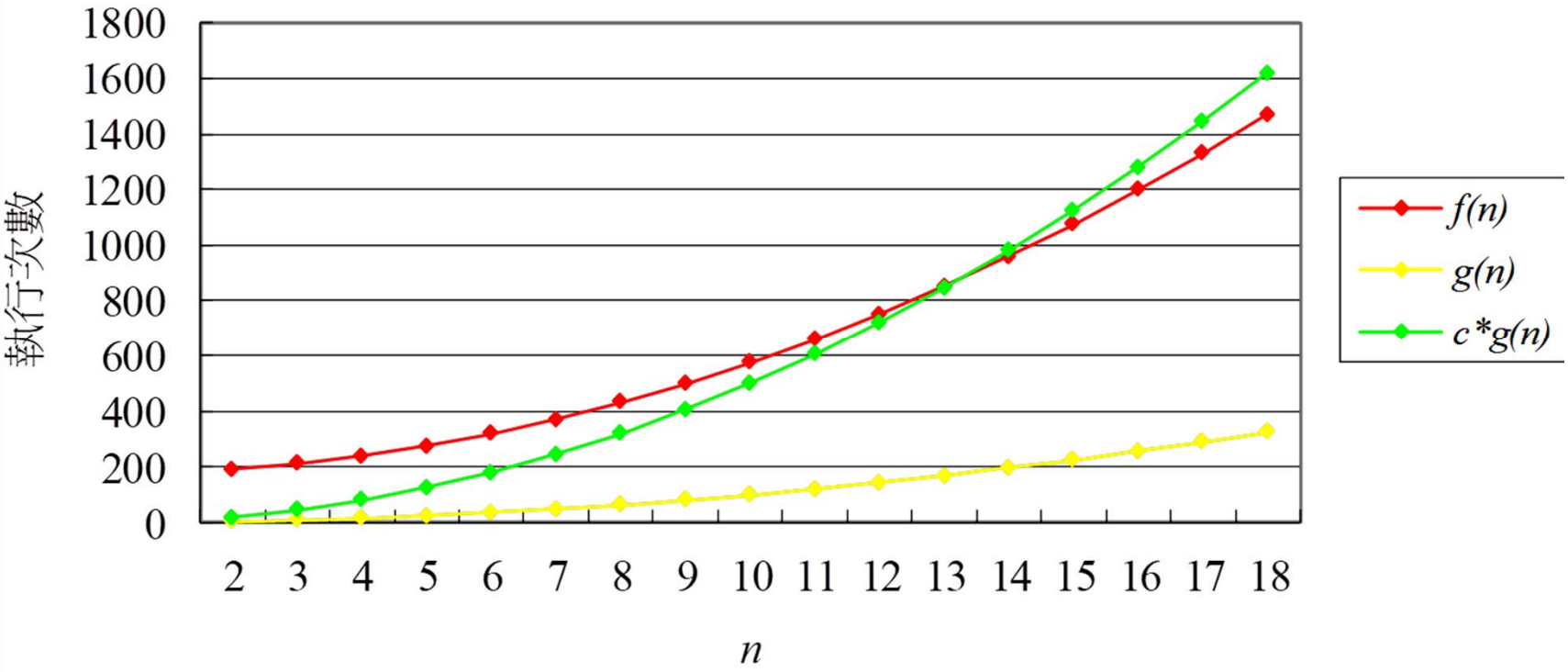
作業

- Section 1.5.2: P32 – ex 4

用大O表示時間複雜度

定義：$f(n)=O(g(n))$ 若且唯若存在兩個正整數c和n_0，當$n \geq n_0$時，$f(n) \leq c \times g(n)$。

- $f(n)$指的是演算法的執行時間（步驟），我們希望能找到$g(n)$；只要在$n \geq n_0$後，$c \times g(n)$一定會大於或等於$f(n)$，那麼就可以用$O(g(n))$來表示$f(n)$。
- 函數O()代表參數頻率計數多項式的級數，唸成Big Oh。



請注意在上圖中，當$n \geq 14$ $(=n_0)$ 之後，$c \times g(n)$ 一定會大於或等於$f(n)$（$5n^2 \geq 4n^2+174$），

所以$O(n^2)$足以代表$4n^2+174$ 。

範例

a) $4n+12 = O(n)$，因為存在$c=5$，$n_0=12$，使得$n \geq 12$後，$4n+12 \leq 5n$（或$c=6$，$n_0=6$，使得$n \geq 6$後，$4n+12 \leq 6n$）；

b) $10n+25 = O(n)$，因為存在$c=11$，$n_0=13$，使得$n \geq 13$後，$10n+25 \leq 11n$；

c) $8n^2+11n+18 = O(n^2)$，因為存在$c=9$，$n_0=13$，使得$n \geq 13$後，$8n^2+11n+18 \leq 9n^2$；

d) $6 \times 2^n+n^2 = O(2^n)$，因為存在$c=7$，$n_0=4$，使得$n \geq 4$後，$6 \times 2^n+n^2 \leq 7 \times 2^n$；

範例 (續)

e) $326 = O(1)$，因為存在c=327，n_0可任取，使得$n \geq n_0$後，$326 \leq 327 \times 1$；

f) $9n^2+n+11 \neq O(n)$，因為找不到適當的c和n_0，使得$n \geq n_0$後，$9n^2+11 \leq cn$；

g) $100n^3 = O(n^3)$，因為存在c=101，n_0=0，使得$n \geq 0$後，$100n^3 \leq 101n^3$。

Big Oh函數的基礎-說明

- 函數O()代表參數頻率計數多項式的級數，唸成Big Oh。
- O(1)表示頻率計數是常數
- O(n)表示頻率計數是an+b
- $O(n^2)$表示頻率計數是an^2+bn+c，a、b和c是常數
- O()函數不計頻率計數的常數，只取其最高次方的項目，所以是O(1)、O(n)和$O(n^2)$。
- 例如：費氏數列fibonacci()函數頻率計數的O()函數，如下所示：

  n=0或1：2　　O(1)

  n>1：4n+1　　O(n)

- 上述頻率計數4n+1是O(n)，因為當n很大時，係數4和1可以不用考慮，O(n)表示程式執行的頻率計數和n成正比。

以大O表示法分辨演算法的優劣

- $O(1)$稱為常數時間，即不論演法的步驟須需要多少指令，只要不像迴圈般重複執行，皆視為常數時間；
- $O(n)$稱為線性時間，取其執行步驟的增加趨勢與n的增加趨勢為線性關係之意；
- $O(n^2)$為平方時間；依此類推，而
- $O(2^n)$則稱為指數時間。
- 如此一來，我們會說$O(\log n)$的演算法比$O(n)$來得有效率，$O(n)$比$O(n^2)$來得有效率...。

Asymptotic Notation (O)

- Definition
  f(n) = O(g(n)) iff there exist positive constants c and n_0 such that $f(n) \leq cg(n)$ for all n, $n \geq n_0$.
- Examples
  - $3n+2=O(n)$ /* $3n+2 \leq 4n$ for $n \geq 2$ */
  - $3n+3=O(n)$ /* $3n+3 \leq 4n$ for $n \geq 3$ */
  - $100n+6=O(n)$ /* $100n+6 \leq 101n$ for $n \geq 10$ */
  - $10n^2+4n+2=O(n^2)$ /* $10n^2+4n+2 \leq 11n^2$ for $n \geq 5$ */
  - $6*2^n+n^2=O(2^n)$ /* $6*2^n+n^2 \leq 7*2^n$ for $n \geq 4$ */

Big Oh函數的基礎-O(1)

O(1)：單行程式碼

```
a = a + 1;
```

- 上述程式碼a = a + 1是一列單行程式碼，不包含在任何迴圈中，頻率計數是1所以是O(1)。

Big Oh函數的基礎-O(n)

O(n)：線性迴圈

```
a = 0;
for ( i = 0; i < n; i++ )
    a=a+1;
```

- 上述程式碼執行n次的a = a + 1，其頻率計數是n+1，包含最後1次的比較，不計常數所以是O(n)。

Big Oh函數的基礎-O(Log n)

O(Log n)：非線性迴圈

```
a = 0;
for ( i = n; i > 0; i = i / 2 )
    a=a+1;
```

- 或

```
a = 0;
for ( i = 0; i < n; i = i * 2 )
    a=a+1;
```

- 上述兩個迴圈的增量是除以2或乘以2，以除來說，如果n = 16，迴圈依序為8、4、2、1，其執行次數是對數Log n，其頻率計數是Log n+1，包含最後1次的比較，不計常數所以是O(Log n)。

Big Oh函數的基礎-O(nLog n)

O(n Log n)：巢狀迴圈擁有內層非線性迴圈

```
a = 0;
for ( i = 0; i < n; i++ )
   for ( j = n; j > 0; j = j / 2 )
      a=a+1;
```

上述巢狀迴圈的外層是線性迴圈的O(n)，內層是非線性迴圈的O(Log n)，所以是：

O(n) * O(Log n) = O(n Log n)

Big Oh函數的基礎-$O(n^2)$

$O(n^2)$：巢狀迴圈

```
a = 0;
for ( i = 0; i < n; i++ )
   for ( j = 0; j < n; j++ )
      a=a+1;
```

- 上述巢狀迴圈的外層是線性迴圈的O(n)，內層線性迴圈的O(n)，所以是：

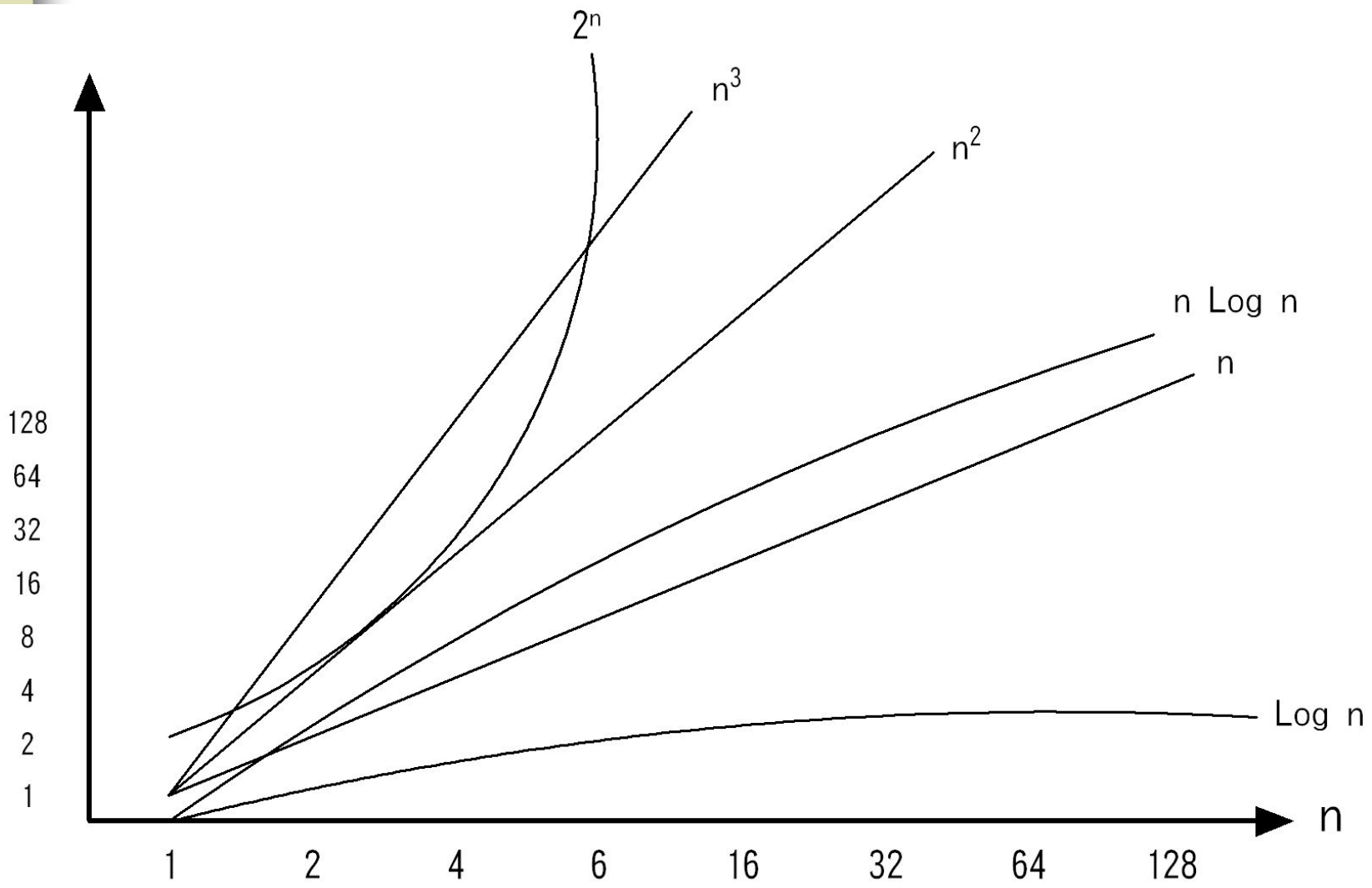
  $O(n) * O(n) = O(n^2)$

Big Oh函數的等級-說明

- 程式執行效率的時間複雜度可以使用O(1)、O(Log n)、O(n)、O(n Log n)、$O(n^2)$、$O(n^3)$和$O(2^n)$的Big Oh函數來表示，如下表所示：

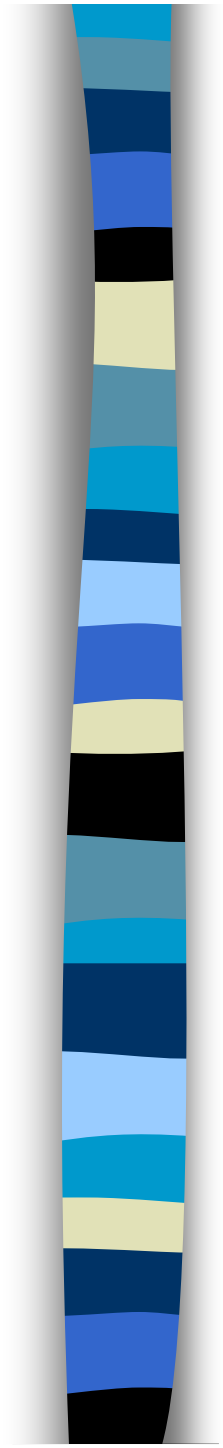
Big-Oh函數	等級	頻率計數範例
O(1)	常數級	2
O(Log n)	對數級	4*Log n+4
O(n)	線性級	5n+1
O(n Log n)	對數線性級	n*(5*Log n)
$O(n^2)$	平方級	$3n^2+5n$
$O(n^3)$	立方級	$n*(3n^2+5)$
$O(2^n)$	指數級	$5*2^n$

Big Oh函數的等級-成長曲線圖



Example

- Complexity of $c_1n^2+c_2n$ and c_3n
  - for sufficiently large of value, c_3n is faster than $c_1n^2+c_2n$
  - for small values of n, either could be faster
    - $c_1=1$, $c_2=2$, $c_3=100$ --> $c_1n^2+c_2n \leq c_3n$ for $n \leq 98$
    - $c_1=1$, $c_2=2$, $c_3=1000$ --> $c_1n^2+c_2n \leq c_3n$ for $n \leq 998$
  - break even point
    - no matter what the values of c1, c2, and c3, the n beyond which c_3n is always faster than $c_1n^2+c_2n$



- O(1): constant
- O(n): linear
- $O(n^2)$: quadratic
- $O(n^3)$: cubic
- $O(2^n)$: exponential
- O(logn)
- O(nlogn)

*Figure 1.7:Function values (p.38)

		Instance characteristic n					
Time	Name	1	2	4	8	16	32
1	Constant	1	1	1	1	1	1
$\log n$	Logarithmic	0	1	2	3	4	5
n	Linear	1	2	4	8	16	32
$n \log n$	Log linear	0	2	8	24	64	160
n^2	Quadratic	1	4	16	64	256	1024
n^3	Cubic	1	8	64	512	4096	32768
2^n	Exponential	2	4	16	256	65536	4294967296
$n!$	Factorial	1	2	24	40326	20922789888000	26313×10^{33}

*Figure 1.8:Plot of function values(p.43)

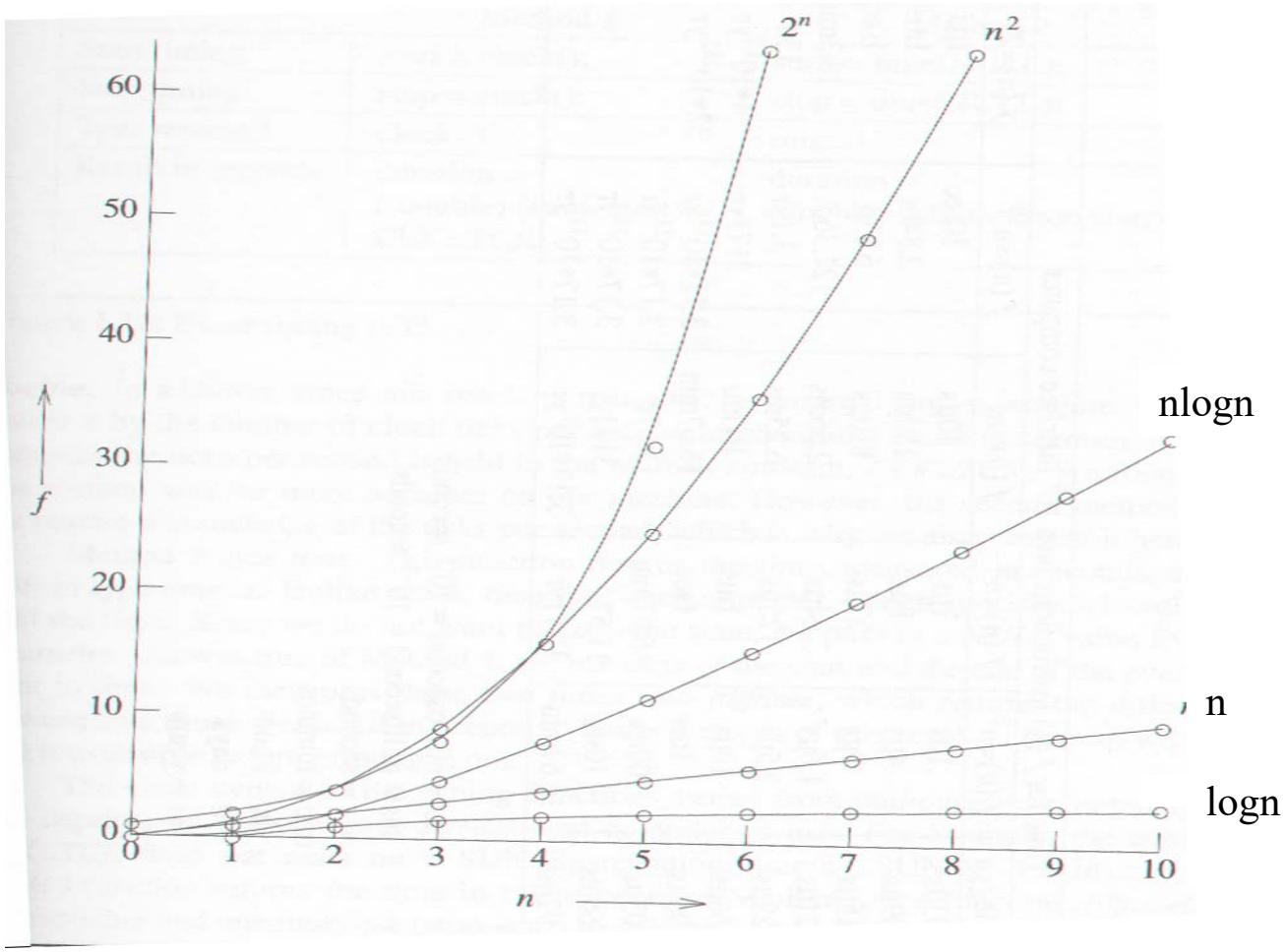


*Figure 1.9: Times on a 1 billion instruction per second computer(p.40)

	Time for $f(n)$ instructions on a 10^9 instr/sec computer						
n	$f(n)=n$	$f(n)=\log_2 n$	$f(n)=n^2$	$f(n)=n^3$	$f(n)=n^4$	$f(n)=n^{10}$	$f(n)=2^n$
10	.01μs	.03μs	.1μs	1μs	10μs	10sec	1μs
20	.02μs	.09μs	.4μs	8μs	160μs	2.84hr	1ms
30	.03μs	.15μs	.9μs	27μs	810μs	6.83d	1sec
40	.04μs	.21μs	1.6μs	64μs	2.56ms	121.36d	18.3min
50	.05μs	.28μs	2.5μs	125μs	6.25ms	3.1yr	13d
100	.10μs	.66μs	10μs	1ms	100ms	3171yr	$4*10^{13}$ yr
1,000	1.00μs	9.96μs	1ms	1sec	16.67min	$3.17*10^{13}$ yr	$32*10^{283}$ yr
10,000	10.00μs	130.03μs	100ms	16.67min	115.7d	$3.17*10^{23}$ yr	
100,000	100.00μs	1.66ms	10sec	11.57d	3171yr	$3.17*10^{33}$ yr	
1,000,000	1.00ms	19.92ms	16.67min	31.71yr	$3.17*10^{7}$ yr	$3.17*10^{43}$ yr	

μs = microsecond = 10^{-6} seconds
ms = millisecond = 10^{-3} seconds
sec = seconds
min = minutes
hr = hours
d = days
yr = years

Performance Analysis

- Omega
  - Def: $f(n)= \Omega(g(n))$ iff
    - there exist positive constants c and n_0
    - such that $f(n) \geq cg(n)$ for all n, $n \geq n_0$
  - e.g.
    - $3n+2 = \Omega(n)$ let $c=3$, $n_o=1$
      $3n+2 \geq 3n$ for all $n \geq 1$
    - $6*2^n + n^2 = \Omega(2^n)$ let $c=1$, $n_o=1$
      $6*2^n + n^2 \geq 2^n$
    - $3n+2 = \Omega(1)$ let $c=1$, $n_o=1$
      » g(n) should be an *upper lower bound*

Performance Analysis

- Theta
  - Def: $f(n)= \Theta(g(n))$ iff
    - there exist positive constants c_1,c_2, and n_0
    - such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all n, $n \geq n_0$
  - e.g.
    - $3n+2 = \Theta(n)$ since $3n \leq 3n+2 \leq 4n$
      $c_1=3$, $c_2=4$, for all $n \geq 1$
    - $10n^2+4n+2= \Theta(n^2)$ since $10n^2 \geq 10n^2+4n+2 \geq 11n^2$
      $c_1=10$, $c_2=11$, for all $n \geq 5$
    - g(n) is in between upper & lower bound

作業

- Section 1.5.3: P41 – ex1.(b), ex1.(k), ex2.(a), ex2.(e), ex 7

C語言回顧

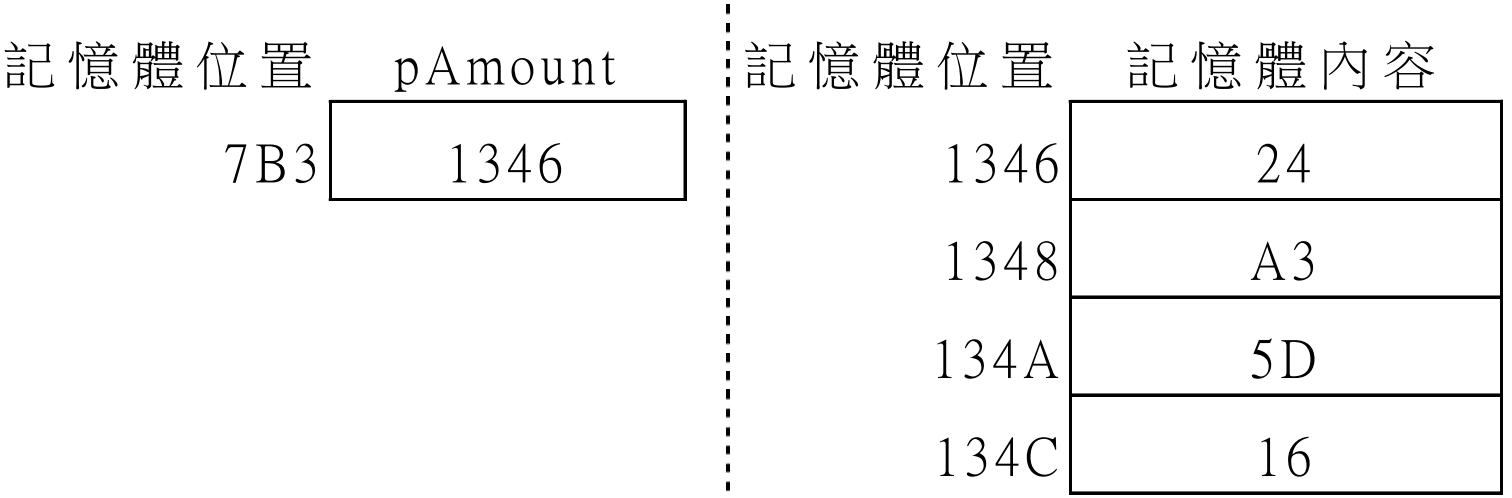
- 變數宣告
- 一個整數變數的宣告方式是：
  - int iYears;
- 其他類型變數的宣告與這個例子相仿

C語言回顧

- 指標變數
- 指標是用來記住資料在記憶體中的位置
- 指向不同資料類型的各種指標屬於不同的資料類型
- 一個指向整數資料的指標的宣告方式：
  - Int *pAmount;
  - 宣告了一個叫做 pAmount 的整數指標，但尚未告訴C語言這個指標的內容應該為何

C語言回顧

- pAmount內的數字代表這個指標正指向記憶體的1346號位置



C語言回顧

- 要取用它所指的記憶體位置之內容，則
  - iYears=*pAmount; /* iYear←24 */
- 要pAmount指向iYears的位置， 則
  - pAmount=&iYears; /* pAmount←iYear在記憶體的位置 */

C語言回顧

- 尋找位置在1346到134C之間的這一段記憶體中，內容為'5D'的記憶體位置

```
for(pAmount=0x1346 ; pAmount<=0x134C; pAmount++)
{
      if (*pAmount==0x5D) break;
}
```

C語言回顧

- 陣列
- 宣告方式：
  - Int iData[5];
  - iData自動被宣告成一個指標，指向這個整數陣列的第一個

Data[0]	Data[1]	Data[2]	Data[3]	Data[4]

iData ↑

C語言回顧

- 自定資料類型

```
typedef  struct    structure_PersonData
{
     char      sName[30];
     int       iAge;
} PersonData;     /* PersonData 為新的資料類型  */
PersonData   Classmates[20];
     /* Classmates[20]的每一個元素皆為 PersonData 格式之變數  */
PersonData   *pClass[5];
     /* pClass[5]的每一個元素皆為指向 PersonData 格式之指標  */
```

C語言回顧

- 宣告了一種新的資料類型，稱為PersonData
- 每一份PersonData資料類型內，都包含有兩個欄位：sName[30]、iAge
- 宣告Classmates為以PersonData這種類型所組成的陣列
- 宣告一個陣列pClass；這個陣列可以存放五筆資料，而每一筆資料都是指向PersonData這種類型的指標

C語言回顧

- 將pClass中的第三個指標所指向的PersonData資料中的iAge欄位內容放入iYears中：
  - iYears = pClass[3]->iAge;