Ch1 資料結構與演算法入門

第一節 資料結構與演算法簡介

1. 簡介資料結構與演算法

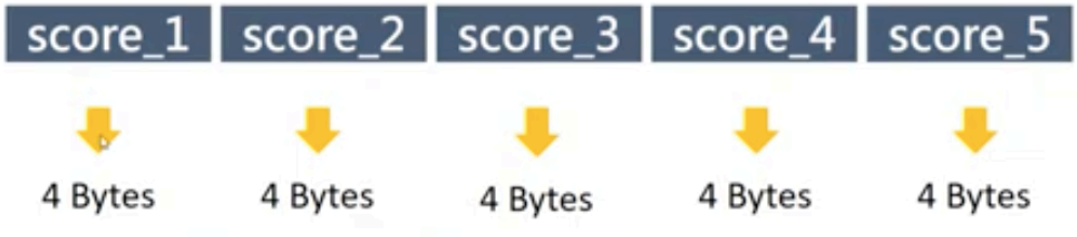
(1) 什麼是資料結構？

資料結構是資料內容加上資料內容間的關聯，比如我們已經學過的陣列，就是不斷把資料放到陣列裡。一個陣列裡的資料關聯就是資料存放的位置，或者看作索引值。

為什麼要有資料結構呢？要是沒有資料結構，我們要處理的資料會散在記憶體裡，難以存取與處理；有好的資料結構，就像圖書館裡的圖書分門別類，擺放得很整齊，方便我們快速查找。

也就是說，有了資料結構，我們就有適當的形式處理資料，可以更快更省時。

我們熟悉的陣列有幾個特點（C/C++ 中的陣列）：



A. 用連續的記憶體空間去裝資料，下一筆資料就在接續的記憶體位置
B. 每個資料佔的空間相同（比如整數陣列每筆資料各占 4 Bytes）
C. 好處
    - 是儲存多筆資料時最省記憶體的方式
    - 取用資料時只要在連續的記憶體位址間移動，方便快速
    - 可以把開頭資料的位置加上索引值，直接算出特定某筆資料的存放位置

(2) 什麼是演算法？

演算法是一步步解決問題的方法，在解決問題時，必須遵守特定的規則，使用電腦的語法來解決這些問題，而演算法並不限於特定程式語言（C/C++、Java、Python 等），可以用任何程式語言撰寫。

當演算法可以解決某一個問題的所有狀況，我們就說演算法「解決」了這個問題。演算法的定義包含下列幾個部分：

A. 一個有限的步驟集合
B. 這些步驟被清楚的定義好
C. 可以被電腦執行

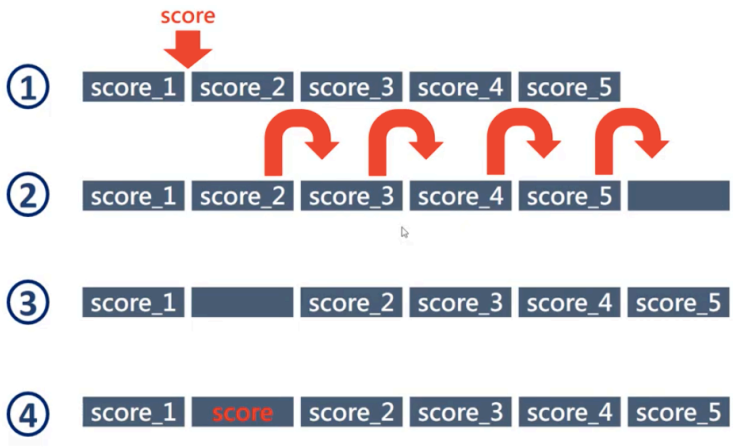
我們所說的「寫程式」其實就是結合了「資料結構」與「演算法」。在寫程式之前，我們要先選定資料結構，演算法則是用這些存好的資料，加上特定的步驟去解決問題。

因此資料結構與演算法是大部分資工系必修的必修課，這是為了培養出合格的工程師，在寫程式時不僅能夠達成目標，還能夠兼顧效能的考量，在不同搭配的效能間選取出最好的並採用。

再以陣列為例：陣列在儲存資料時，資料與資料之間的關係是記憶體位置的關聯，它的好處是儲存空間最小化，但是它也存在許多問題。

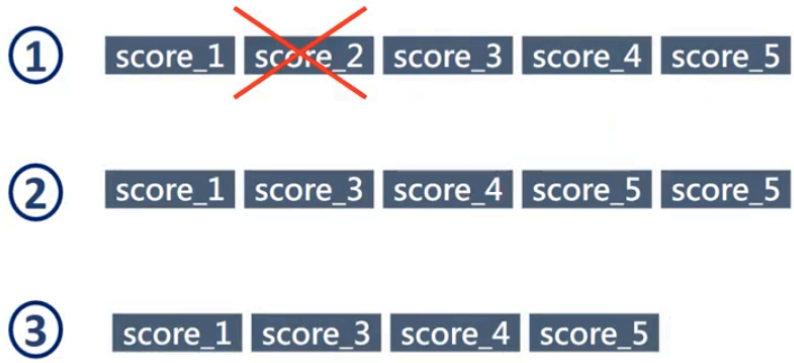
我們考慮幾種常見的陣列操作：

A. 新增資料



新增資料進陣列時，最多需要移動 n 筆資料（在開頭新增資料後，要把原本所有的 n 筆資料往後移一格）

B. 刪除資料



刪除陣列中的資料時，最多需要移動 n-1 筆資料（刪除開頭的資料時，要把後面 n-1 筆資料都往前移）

C. 搜尋資料

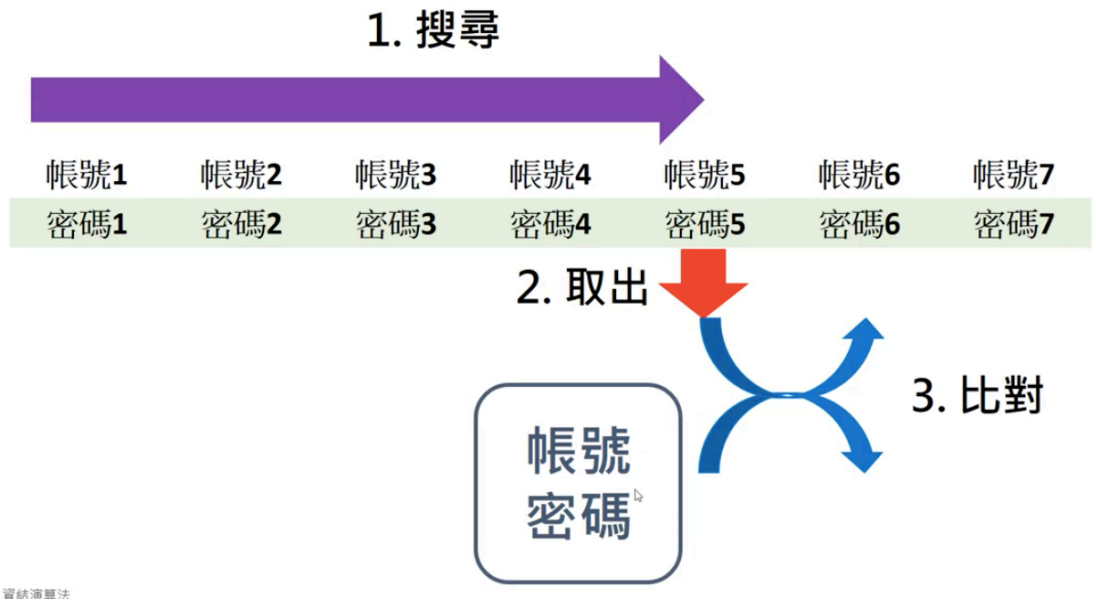
從陣列開頭一個個往後找，叫做循序搜尋，當搜尋的目標在陣列尾端時，需要進行 n 次運算。

舉例而言，如果我們有一個長度為 4 的陣列，為了新增一筆資料在陣列開頭，我們需要將原有的 4 筆資料往後移，如果我們想要刪除第二筆資料，則在刪除後需要將後面的 3 筆資料都往前移。在這個例子裡，移動 3、4 筆資料似乎不是什麼大工程，但當陣列很大時，比如當中有 50,000 筆（甚至 5000 萬、5 億筆）資料的情形，我們每次進行新增與刪除，都得要移動很多筆資料，是相當耗費時間的。

(3) 為什麼要研究資料結構與演算法

你可能會想，真的有研究資料結構與演算法的必要嗎？只要能夠達到目的，使用的資料結構或演算法是不是沒有優劣之分？

事實上，剛剛提到的新增、刪除、查找資料等的操作都很常見。舉我們熟悉的 Facebook 為例，Facebook 在回應每個使用者的需求時，都有進行註冊、登入、搜尋、排序的必要。



比如將新使用者註冊的帳號密碼放到資料庫裡，之後使用者每次登入時，都搜尋資料庫，找到輸入的帳號，再取出對應的密碼，最後把取出的密碼與輸入的密碼加以比對，如果比對結果是一致的，則使用者登入成功，如果不一樣，就會請你重新輸入一次。

像這樣，每個使用者每次登入的時候，都需要經過查找資料的過程。這需要從

第一筆資料開始一路往後搜尋，最多可能要從頭到尾取出所有資料才找到。假設 Facebook 總共有 100 億個帳號（畢竟有些人辦了超過一個帳號），這些帳號密碼的配對都使用陣列儲存，那麼我們平均需要搜尋 50 億筆帳號才能決定一次登入是否是成功的，如果同時間有 10000 個人登入，系統就會無法負荷。

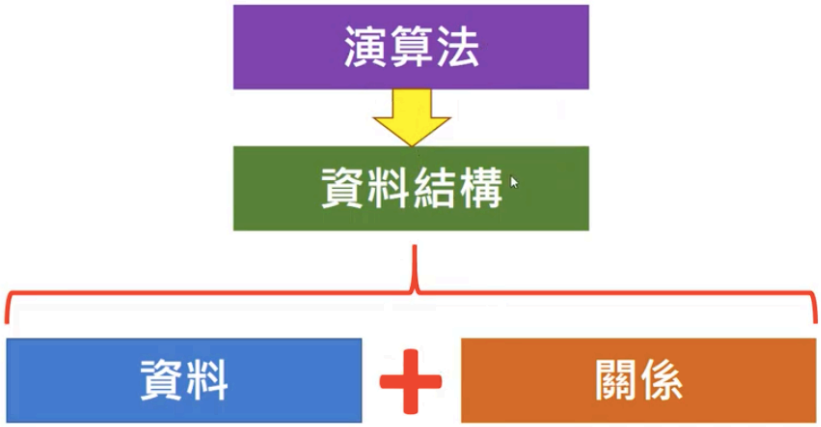
因此，在寫程式的時候需要考慮擴充性，也就是未來業務成長之後能不能承擔增加的流量。

(4) 空間複雜度與時間複雜度

不同的資料結構和對應的演算法有不同的「空間複雜度」和「時間複雜度」。空間複雜度是執行時耗費的記憶體容量，時間複雜度則是運算次數/花費的時間。

2. 常見的資料結構與演算法

1. 資料結構與演算法的關係



演算法操作資料結構，換句話說，演算法就是使用資料的方式。改變我們在程式中使用的資料結構，就會改變資料的使用方式。在使用某個演算法的時候，就要選擇對應的資料結構。

通常這些資料結構我們不會自己發明，因為 C++ STL 裡已經內建了各種資料結構，我們只要瞭解其內容，並能夠熟練使用就好。

以「搜尋」為例，我們知道使用陣列來儲存資料時，要搜尋特定資料就得慢慢從第一筆查找到最後一筆，但是是否有更合適的資料結構可供我們使用呢？C++ 內建的二元樹就是一個更好的選擇，使用二元樹儲存資料時，只要經過 n 次搜尋後，我們就可以覆蓋 2^n 筆資料，因為每筆資料連到兩筆資料，每多進行一次搜尋就可以覆蓋兩倍的資料量。假設有 8 筆資料需要搜尋，只要搜尋 3 次就可以完成。

因為資料結構和演算法之間的關係密切，通常學習程式時會一起學這兩個學科。若你一定要選擇其中一門先學，先選擇資料結構會比較好一點。

(1) 資料結構與演算法的搭配

我們列舉一些常見的資料操作，這些操作我們無時無可都在用，我們可以如何加速這些操作呢？

常見的資料操作

A. sort 排序
B. search 搜尋
C. delete 刪除特定元素
D. insert 插入特定元素
E. push 放入一個元素
F. pop 取出一個元素
G. reversal 反轉
H. query 查詢

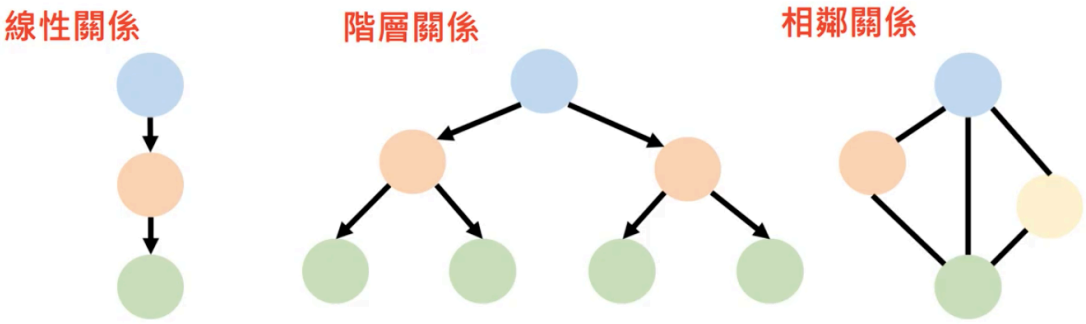
我們常用的資料結構有這些：

A. Array 陣列
B. Linked list 鏈結串列
C. Stack 堆疊
D. Queue 佇列
E. Binary tree 二元樹
F. Undirected graph 無向圖
G. Directed graph 有向圖

對於每一種問題，我們通常會選擇不同的資料結構與不同的演算法。

舉例來說，在「廣度優先搜尋」中，我們通常會使用「佇列」;「深度優先搜尋」中，我們則通常使用「堆疊」; 有新增或刪除資料的需求時，我們使用鏈結串列（Linked list），查找時則優先使用陣列（Array）。

常見的資料結構可以大致分為三種：



A. 線性關係：這種資料結構中，一筆資料只會連結到另外一筆資料。比如陣列、鏈結串列、堆疊、佇列等

B. 階層關係：一筆資料會連結到多筆資料，而且資料間有不同階層的關係，但是不會形成環（比如上面階層關係的圖中，最下方的綠色節點不會往上連回藍色節點），「樹」就是屬於這一類

C. 相鄰關係：資料結構中一筆資料向外連結到其他筆資料後，有可能會再連結回來，比如圖論中的圖

(2) 常見的演算法與資料結構

下面兩個表格中，左邊的演算法和操作可以被運用在右邊的資料結構上：

常見的演算法
讀取
搜尋
遞迴
二分搜尋
排序
動態規劃
貪婪演算法

常見的資料結構
Array 陣列
Linked list 鏈結串列
Stack 堆疊
Queue 佇列
Binary tree 二元樹
Undirected graph 無向圖
Directed graph 有向圖

3. 評估演算法的好壞

(1) 為什麼要評估演算法的好壞？

根本的原因是資源的有限性，雖然電腦的運算速度是人類難以比擬的，但是絕非無限快；同時，記憶體也有容量上限。

(2) 圈圈叉叉與圍棋

圈圈叉叉總共只有 5,478 種下法，對於電腦而言，可以很快運算出所有答案，也因此可以在任何局面上判斷出勝算最大的下法。

圍棋則被稱為「人類最後的堡壘」。一直到前幾年 Alpha Go 出來前，人類在圍棋遊戲上都可以打敗電腦，因為根據圍棋的規則，完整的一局總共有 10^{171} 種下法。10^{171} 是多大的數字呢？我們可以這樣理解：全宇宙的原子數量級在 10^{80}，如果在這每個原子中都放入一個宇宙，而所有被放入的宇宙中，各存在的 10^{80} 個每個原子都可以儲存一種棋局的發展，這也只能儲存 10^{160} 種棋局，仍然無法窮舉完所有圍棋的下法。

這代表即使電腦的運算能力遠遠超過人類，仍然無法很快的解決下圍棋的問題，因為情況太多了。有沒有可能打贏 Alpha Go？有，因為他選擇的並不是最佳解，而是在它能夠搜尋的範圍內最後的解答，叫做「局部最佳解」。即使如此，人類贏 Alpha 的可能性仍然是微乎其微。

從以上的例子我們理解到，如果我們的演算法越好，能夠搜尋的範圍越多，就可以下得越精準。這就像是我們能夠看到的範圍（視野）是有限的，用好的演算法，就能夠用越高的角度來看整個棋局、地圖或環境，從更高的視角來看整個環境，就能夠對環境有更多認識，就可能越早找到路徑走出迷宮。

這裡我們的結論是：當某個問題無法透過窮舉來解答時，運算越快、演算法越好，一次就能夠看更多東西，算出來的解答也會更好。

(3) 演算法的評估方式

在評估演算法時，我們主要會考慮三個方向：

A. 耗用的資源，包括記憶體空間與 CPU 的運算
B. 寫程式時的程式碼的複雜度
C. 其他人理解寫出程式碼的困難程度

在本課程中，我們通常關注的是程式執行時的運算次數，也可以看成所需花費的時間。

(4) 評估方式的平衡

但是「平衡」也是很重要的。

美國為何使用人口抽查代替普查？因為根據估算，想挨家挨戶調查每個家戶，在戶政單位有限的資源下，光是一輪普查就要 7 年時間。這樣一來，普查就失去了意義，因為長達 7 年的時間裡，被調查的對象中很多小孩已經長大，也有許多人已經換工作了。

為了避免耗時費力得到的資料沒有參考性的問題，我們選擇一種折衷方式，也就是每幾個人才抽一個來問，而不要每個人都問，這樣就能夠在比較短的時間內完成一輪。這裡我們看到，因為資源的有限，對於縮短時間的要求有時會勝過對精準度的要求，也就是說我們犧牲了結果的精準度以求更快得出結果。

(5) 時間複雜度與空間複雜度

什麼時候應該更關注時間複雜度，什麼時候則應該更考慮空間複雜度呢？

運算時間和運算的次數大致呈正比。由於 CPU 運算一次如果需要一毫秒，則算 10 次至少需要 10 毫秒，而且事實上 CPU 的運算資源又比記憶體資源來得貴，要增加運算資源是兩者間較困難的：記憶體兩倍只要花兩倍錢買記憶體然後插入插槽內擴充，但 CPU 要擴充成兩倍，成本會比兩倍更高，因此我們多半在意的是時間複雜度。

不過這也不是鐵則，比如醫療影像處理就是一種例外，因為醫療影像講求高解析度，一張影像動輒就需要數 GB 的儲存空間，這時我們就傾向關注空間複雜度。一般而言，高解析度的照片或影片才會有空間複雜度上的考量，像文字小說等 txt 檔，佔用的空間不過幾 MB，本書中，我們主要考慮時間複雜度。

(6) 兩種變數交換方式

交換兩個變數的值是大家已經很熟悉的操作，我們用兩種不同的變數交換方式來直觀比較一下注重時間複雜度與空間複雜度所得到的不同程式寫法。

寫法一	寫法二

int a = 5, b = 3;
a += b;
b = a-b;
a = a-b
	int a = 5, b = 3, tmp;
	tmp = a;
	a = b; // b assign 給 a
	b = tmp;

多耗費運算時間（空間複雜度好）	多耗費記憶體空間（時間複雜度好）

比較寫法一與寫法二，寫法二多使用到了 tmp 這個變數，而寫法一並沒有多宣告變數，因此寫法二的空間複雜度明顯較高（比較不好），但是相較於寫法二只使用到 = （賦值，assign）運算，寫法一使用了加減運算，因為電腦處理賦值運算比處理加減運算快，所以寫法一的時間複雜度較高（亦即較差）。

如同前述，通常我們在意的是時間複雜度，所以實踐上，我們看到多數的交換函式（swap）都採用了寫法二。

(7) 在意資料量大的情形

另外我們還要特別提到，比較不同演算法的複雜度時，有時會因為資料量的大小而得到不同結果。兩種算法 A、B 可能會有資料量小時 A 較有效率，資料量大時 B 較有效率的情形，因為資料量小時，總耗費時間的差異也必定不大，所以我們更在意資料量級很大的時候哪種算法更有效率。

比較「耗費的時間」對「資料量 x」的函數各為 x^2 與 5x 的兩種演算法：
$x^2 < 5x$ for $x < 5$
由上式我們知道 $x<5$ 時，x^2 比較好，但是一旦 $x > 5$，也就是資料量超過 5 筆時，5x 就會比較好（花費的時間較少）。我們當然更在意 $x > 5$ 時花費時間的多寡，因為 5 筆資料一般來說不可能花費到多久，處理上萬、上億筆資料時的效率是我們更在意的，也因此，我們會傾向使用 $t(x) = 5x$ 的演算法，而非 $t(x) = x^2$ 的演算法。

第二節 效能還與哪些有關

1. 另外有幾件事必須注意

(1) 程式碼與程式效能

程式的效能只受到程式碼的直接影響嗎？答案是否定的。

考慮兩個數字相加，當數字小時，基本運算（加減乘除）可以在常數個指令內完成，但是 CPU 單次運算可以處理的位元有限，當數字過大時，就得改用軟體完成，也就是把一筆比較大的資料切割成很多筆小資料再做運算。

這代表同樣一個程式碼，同樣的運算步驟，有時執行的效能會不太一樣（可能受到輸入資料的影響）。我們要記得程式碼越短不等於執行越快，執行上要回到組合語言去看，因為編譯器常默默幫我們完成許多程式碼上無法看到的事情。

(2) strlen 的例子

我們用 strlen 這個函式來看看編譯器自動採用的執行方式是如何影響到程式效能的。

Code Ch1-1		
1	for(int i=0;i<strlen(str);i++){	// int i = 0
2		// check if i<strlen(str), if yes, repeat
3	if(str[i]=='a') counts++;	// check if(str[i]=='a'), if yes, counts++;
4		// i++
5	}	

假設傳入一個長度為 n 的字串 str，在上面的程式碼裡，我們試著算出 str 裡 'a' 這個字元的出現次數。

執行一次 strlen(str) 會需要進行 n 次運算，因為這個函式是直接從開頭字元一個一個數出 str 的字元個數的。再來，因為我們寫了一個 for 迴圈，這個迴圈會執行 n 次，每次呼叫 strlen 又都進行 n 次運算，所以看起來總共需要進行

n^2 次運算。

但是實際測試會發現，當資料量變 n 倍時，需要的時間並沒有真的變成 n^2 倍。這是因為編譯器會默默進行優化，當它發現每次迴圈跑的過程中 strlen 的值不會改變，就將其看作常數，導致程式的執行複雜度與表面上看起來不同。

2. 為什麼我們使用 C++?

(1) C++ 的優缺點

有些同學可能有疑問，為什麼在學習資料結構與演算法時，我們選擇使用 C++？

C++ 主要的好處：

A. C++ 內建的 STL 函式庫有把常見的資料結構與演算法封裝進去

B. 支援指標操作，鏈結串列（linked list）用指標寫比較有感覺

C. Python 的執行效能較差，可能會發生超過時間限制 (TLE, Time Limit Exceeded) 的問題

當然 C++ 也有壞處，最大的一點就是學習與撰寫起來不如 Python 直觀。要提醒同學的是，如果目標是比程式競試或進大公司，因為會考驗寫出來程式的效率，因此要得高分最好使用 C/C++。

(2) 使用 LeetCode 的提交結果有下列幾種：

A. Accepted(AC)：通過

B. Wrong Answer(WA)：答案是錯的

C. Time Limit Exceed(TLE)：線上評分網站擔心有人提交了無窮迴圈造成資源浪費，所以一旦提交的程式碼執行超過時間上限，就會跳出這個結果

D. Runtime Error(RE)：執行 EXE 檔時出錯

E. Compile Error(CE)：編譯成 EXE 檔的過程中出錯

F. Memory Limit Exceed(MLE)：超過可用的記憶體上限，比較少見

(3) C++ 的一些優化方式

在撰寫 C++ 時（尤其是比程式競試的時候），有一些技巧可以使用：

A. 輸入優化：把與 stdio 的同步設為 false，這樣每次用 cin 和 cout 後就不會用 flush 把緩衝器清掉，輸出入會比較快

```
std::ios::sync_with_stdio(false);
std::cin.tie(0);
```

B. 讀檔檢查：讓輸出入的資料顯示在螢幕上，這樣就可以檢視現在存取的的資料究竟有沒有問題。

```
freopen("test.in", "r", stdin);
freopen("test.out", "w", stdout);
```

第三節 小結

1. Take Home Message

（1）資料結構：資料與資料間的關係，把資料先儲存起來，演算法才能夠操作

（2）演算法：操作或運算這些資料的方法與步驟

（3）評估程式效能的方式：主要分成「時間複雜度」與「空間複雜度」

A. 時間複雜度：運算所需的時間，與運算次數成正比

B. 空間複雜度：耗用的記憶體空間

（4）程式碼越短效能一定越好嗎？

不一定，還需要考量運算的次數和組合語言

（5）還有哪些東西會影響效能？

最常見的是編譯器優化，程式實際執行時不一定完全依照我們寫的程式碼，因此電機系和資工系通常會修組合語言，以了解程式實際執行的情況，如果有機會的話，也建議大家去修組合語言這門課