# Ch1 資料結構與演算法入門

## 第一節 資料結構與演算法簡介

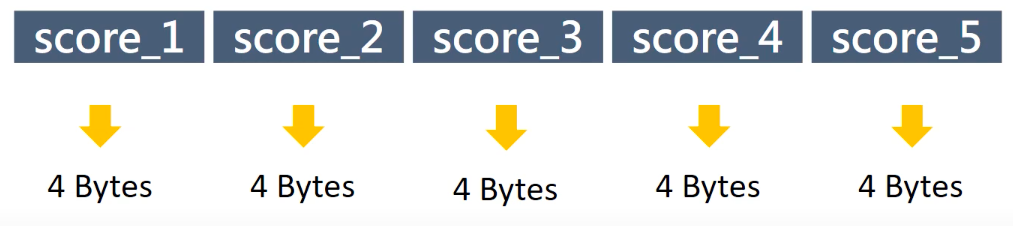
### 1. 什麼是資料結構與演算法

（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」。比如讀者過去可能學過的「陣列」，就是不斷把資料（內容）放到一個「資料結構」裡，陣列裡的「資料關聯」是資料存放的位置，也就是索引值。

為什麼要有資料結構呢？要是沒有資料結構，要處理的資料散在記憶體裡，就難以存取與處理；有好的資料結構，如同圖書館裡的圖書經過分門別類，擺放得很整齊，使得查找方便快速。

也就是說，應用資料結構，能以適當的形式處理資料，變得更快更省時。

C / C++ 中的陣列作為一種資料結構，有幾個特點：

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（比如整數陣列中，每筆資料各占 4 個 Byte）

C. 優點

* 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
* 把「開頭資料的位置」加上特定某筆資料的「索引值」，可以直接算出該筆資料的存放位置

（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，使用電腦的語法來進行。演算法並不限於特定程式語言（C / C++、Java、Python、…），可以任選其一撰寫。

當一個演算法可以解決某一個問題的所有狀況時，就說該演算法「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. 每個步驟被清楚的定義好

C. 每個步驟皆可被電腦執行

一般說的「寫程式」，其實結合了「資料結構」與「演算法」：在寫程式之前，要先選定資料結構，演算法則使用存在資料結構中的資料，經過特定的步驟來解決問題。

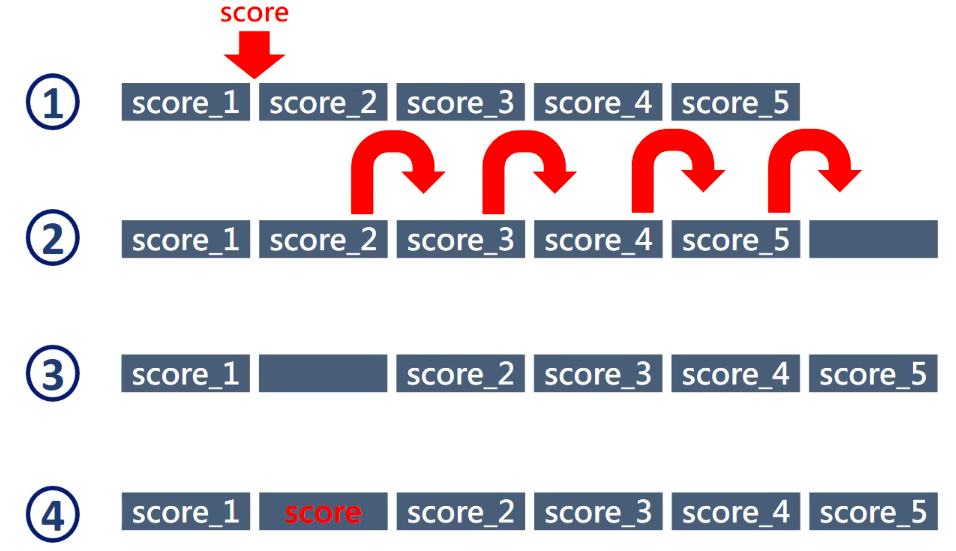
因此，寫程式時必要的「資料結構」與「演算法」是大部分資工系的必修課，這是為了確保培養出合格的工程師，使他們在撰寫程式時，不僅能夠「達成目標」，還能夠「兼顧效能的考量」，在不同資料結構與演算法搭配可得的效能間選取出最好者並採用。

（3）為什麼不全部使用陣列就好？

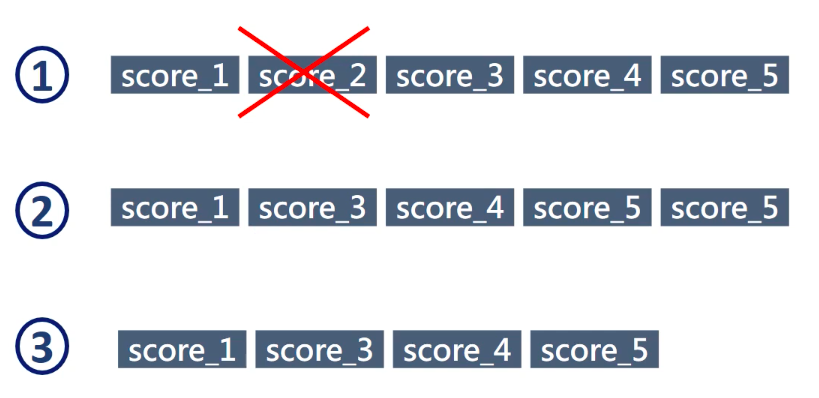
使用陣列儲存資料時，每一筆資料間的關係是「記憶體位置的關聯」。陣列的優點在於儲存空間最小化，但也存在許多問題。

考慮幾種常見的陣列操作，假設陣列長度為 ：

A. 新增資料



新增資料進陣列時，最多需要移動 筆資料（若新增資料位置在陣列開頭，要把原本的 筆資料全部往後移一格）。

B. 刪除資料

刪除陣列中的資料時，最多需要移動 筆資料（刪除開頭的資料時，要把後面 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 次運算。

上面的例子中，為了新增一筆資料在長度為 5 的陣列中開頭資料的「後方」，需要把 4 筆資料往後移，而如果是想刪除第二筆資料，則刪除後需要將後面的 3 筆資料往前移。

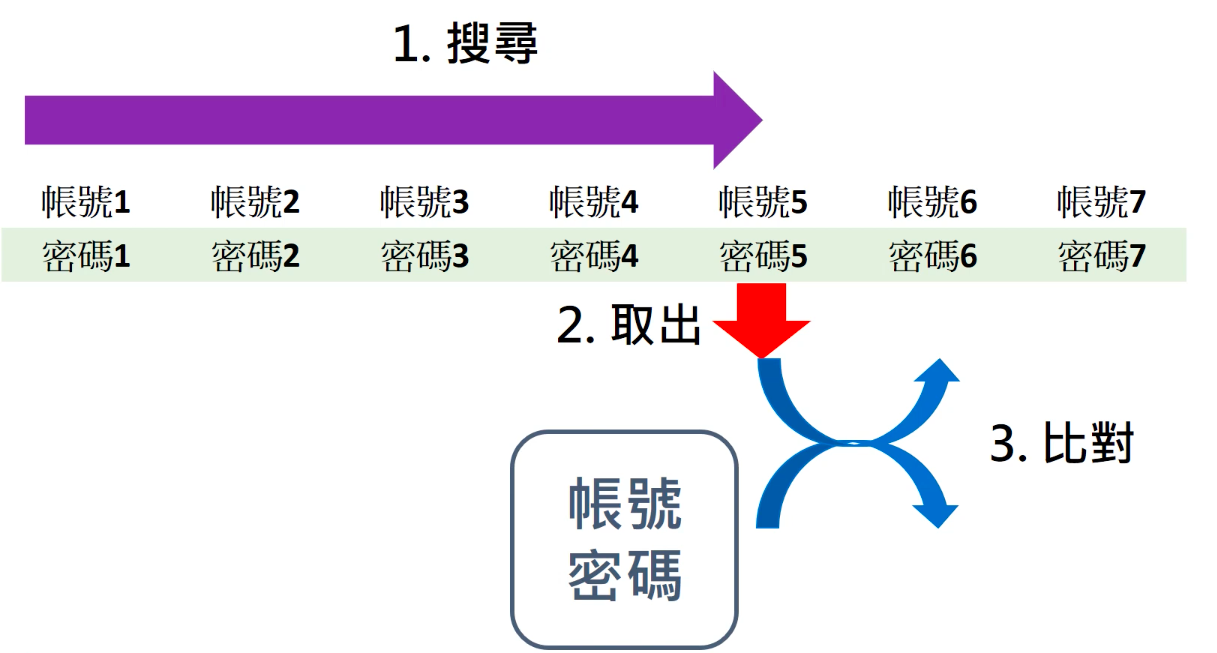
移動 3、4 筆資料似乎不是什麼大工程，但當陣列很大時，比如有 5 萬筆，甚至 5,000 萬、5億筆資料的情形，則每次進行新增與刪除，都得移動很多筆資料，相當耗費時間。

（4）為什麼要研究資料結構與演算法

讀者可能會想，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不是就沒有優劣之分了？



事實上，剛剛提到的新增、刪除、查找資料等操作都相當常見。舉大家熟悉的Facebook 為例，為回應每個使用者的操作，伺服器隨時需要進行受理註冊、確認登入、搜尋、排序等流程。



一開始，新使用者註冊的帳號密碼被放到資料庫中。

之後使用者每次登入時，都會搜尋資料庫，先找到輸入的帳號，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，如果不一致，則會跳出要求重新輸入的訊息。

就像這樣，回應每個使用者的每次登入動作，都需要經過查找資料的過程。若使用「循序搜尋陣列」來進行，要從第一筆資料開始一路往後尋找，最多可能從頭到尾取出所有資料後才找到（即搜尋目標位在陣列尾端）。

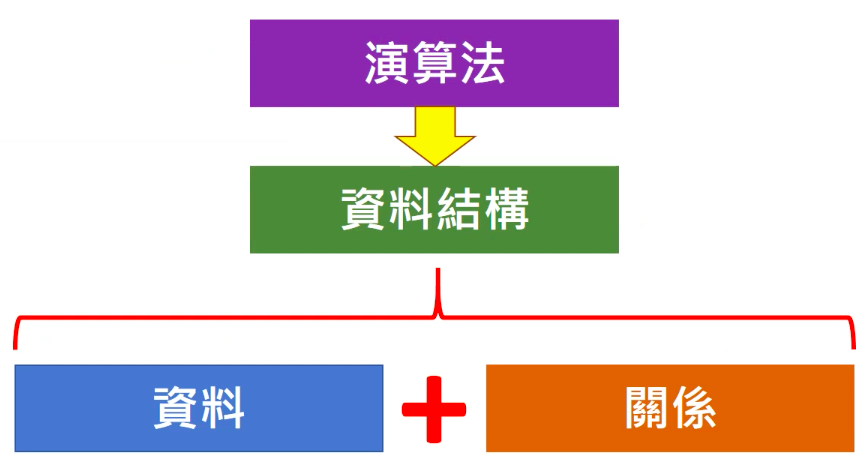
假設 Facebook 總共有 100 億個帳號（畢竟有些人會辦超過一個帳號），且所有帳號密碼的配對都使用陣列儲存，那麼平均來說，要搜尋 50 億筆帳號才能決定一次登入是否成功，如果同時間有數萬名使用者登入，系統就會無法負荷。

因此，在寫程式的時候需要考慮使用的「資料結構與演算法搭配」是否具有充分擴充性，亦即需考慮未來業務成長之後，是否可以負荷增加的流量。

（4）空間複雜度與時間複雜度

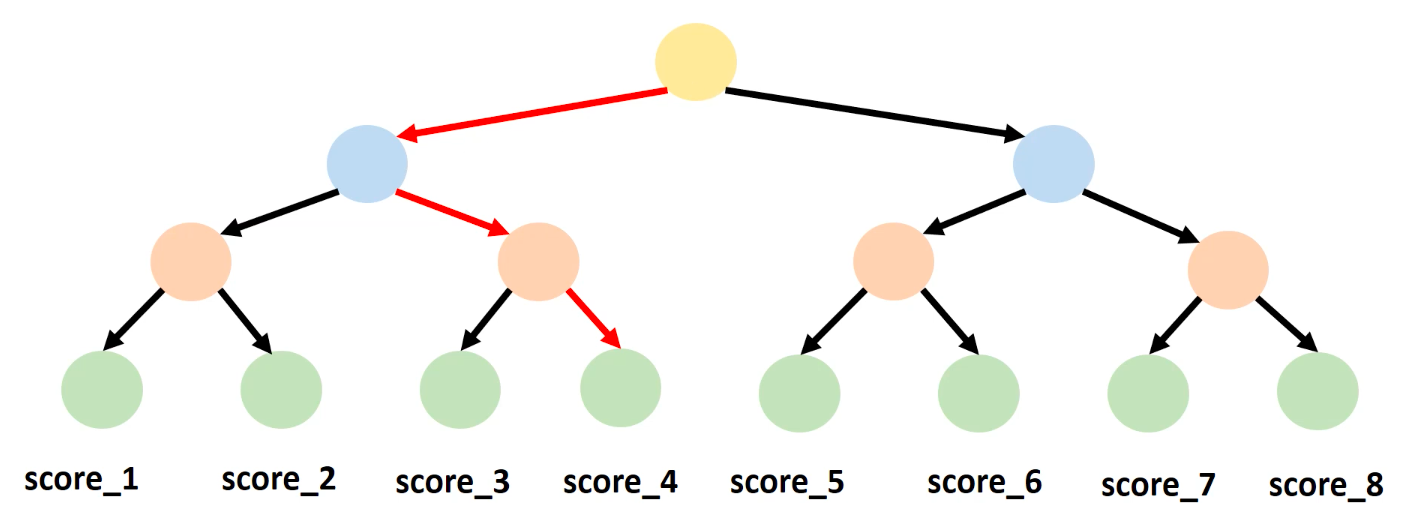
不同的資料結構和演算法有不同的「空間複雜度」和「時間複雜度」。空間複雜度指的是「耗費的記憶體容量」，時間複雜度則是「運算次數 / 花費的時間」。

**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。若改變程式中使用的資料結構，就會連帶改變適當的資料的使用方式；反過來說，採用某種演算法時，也要選擇對應的資料結構。

通常程式的撰寫者不會自己發明資料結構，因為 C++ STL 裡已經內建了各種常用的結構，只需瞭解其內容，並能夠熟練使用就好。

以「搜尋」為例，已知使用陣列來儲存資料時，搜尋特定資料得慢慢從第一筆查找到最後一筆，但是否有更合適的資料結構可供使用呢？

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 次搜尋後，即可覆蓋 筆資料，因為每筆資料都連接到「兩筆」資料上，多進行一次搜尋就覆蓋約兩倍的資料量。如上圖所示，若有 8 筆資料需要搜尋，至多只需搜尋 3 次就可以完成（）。

因為「資料結構」和「演算法」間的關係密切，通常學習程式時會一併學習這兩個學科。若一定要選擇其中一門先學，則先選擇資料結構會好一點。

（2）資料結構與演算法的搭配

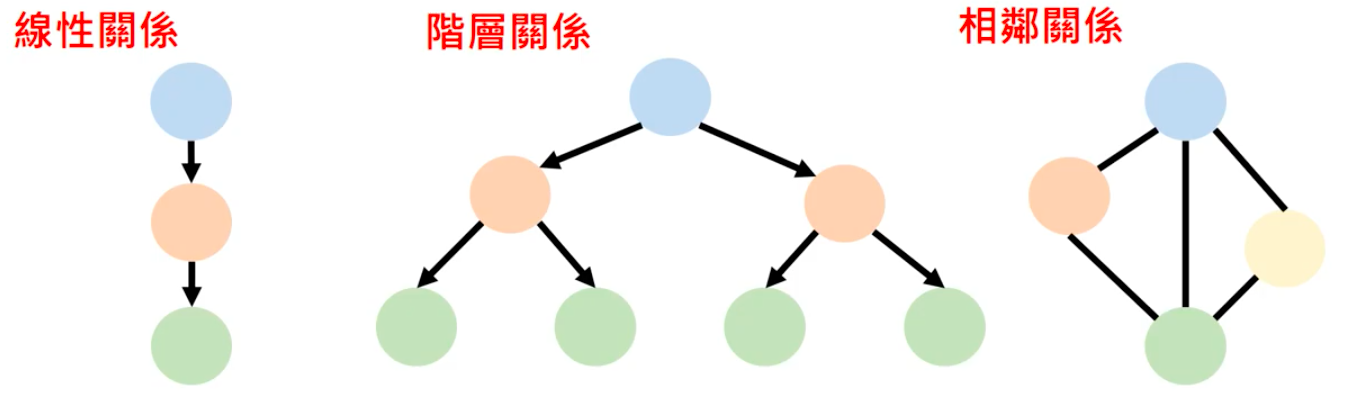
下面列舉一些最常見的資料結構與資料操作，在撰寫程式時隨時會使用到：

|  |  |
| --- | --- |
| 常見的資料操作 | 常用的資料結構 |
| 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 | 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 |

對於不同的問題，通常會選擇不同的資料結構及演算法。

舉例來說，在「廣度優先搜尋 BFS」中，通常使用「佇列 Queue」；在「深度優先搜尋 DFS」中，則通常使用「堆疊 Stack」。

有大量新增或刪除資料的需求時，較常使用「鏈結串列 Linked list」；需要常常存取資料時，則優先使用「陣列 Array」。

常見的資料結構可以大致分為三類：

A. 線性關係：這類資料結構中，一筆資料只會連結到另外「一筆」資料，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而繞成一個圈圈），如各種「樹」。

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（即可能會有環），比如圖論中的「圖」。

（3）常見的演算法

常見的演算法可以被運用在上述的各種資料結構上：

|  |
| --- |
| 常見的演算法 |
| A. 讀取  B. 搜尋（循序、二分、…）  C. 遞迴  D. 排序  E. 動態規劃  F. 貪婪演算法 |

### 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手動計算難以比擬的，但絕非無限快；同時，配置的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，可以很快根據目前局面運算出所有可能發展，也因此可以隨時計算出勝算最大的下法。

圍棋在過去則被稱為「人類最後的堡壘」，直到數年前 Alpha Go 被開發出來前，人類在圍棋遊戲上都能夠打敗電腦。這是因為根據圍棋的規則，完整的一局共有 種可能的進行過程。

是多大的數字呢？可以這樣理解：全宇宙的原子的總數其數量級約在 ，也就是說，如果在目前宇宙的每個原子中都放入一個宇宙，並假定所有被放入的宇宙中各存在的 個原子裡，各自可以儲存一種棋局的發展，這樣也只能儲存 種棋局，仍然無法窮舉完所有的圍棋下法。

這使得雖然電腦的運算能力遠遠超過人類，仍無法很容易的解決下圍棋的問題，因為情況太多了。Alpha Go 當然也不可能透過窮舉的方式來得到最佳解，而僅只於選擇它「能夠搜尋的範圍內」的最佳解答，也叫做「局部最佳解」，因此 Alpha Go 仍可能被打敗，只是人腦要贏 Alpha Go 的可能性微乎其微。

從以上的例子可以理解到：在固定的資源下，演算法越好，能夠搜尋的範圍就越多，可以下得越精準。因為看到的範圍（視野）是有限的，用好的演算法，就像以更高的角度來看一個「棋局」、「地圖」或「環境」，對環境有更多認識，也就可能越早找到路徑走出迷宮。

這裡的結論是：當無法透過窮舉來解答某個問題時，運算越快、演算法越好，就能夠一次看到並考量更多東西，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，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，程式碼的複雜度

C. 其他人理解該程式碼的困難程度

本書中，通常關心程式執行時所需的「運算次數」，也可看成需花費的時間。

（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來代替普查？因為根據估算，在戶政單位有限的資源下，想挨家挨戶調查每個家戶，光是一輪普查就要 7 年時間。這樣一來，普查根本失去了意義，因為長達 7 年的時間裡，許多調查對象中小孩已經長大，許多人也已經換工作了。

為了避免耗時費力得到的資料沒有參考性，只能選擇一種折衷方式，也就是每幾個人中只抽一個來問，而非每個人都問，這樣就能在比較短的時間內完成一輪。

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，也就是犧牲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，CPU 運算一次如果需要一毫秒，則算 10 次至少需要 10 毫秒。

而事實上，CPU 的運算資源比記憶體資源來得貴，「增加運算資源」是兩者間較困難的：要擴充記憶體為兩倍，只需花兩倍錢來買記憶體，並插入插槽內，但要擴充 CPU 為兩倍，則成本會比兩倍更高，因此在意的通常是時間複雜度。

不過這也不是鐵則，比如醫療影像的處理就是一種例外，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，這時就會傾向關注「空間複雜度」，避免所需記憶體空間太大。

一般而言，處理高解析度的照片或影片時，才會有空間複雜度上的考量，文字小說等 txt 檔佔用空間不過幾 MB，只考慮時間複雜度即可。本書中，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相當基礎的操作，下面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「注重空間複雜度」所得的不同程式寫法。

|  |  |
| --- | --- |
| 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; |
| 耗費運算時間多，耗費記憶體空間少  （時間複雜度差，空間複雜度好） | 耗費記憶體空間多，耗費運算時間少  （空間複雜度差，時間複雜度好） |

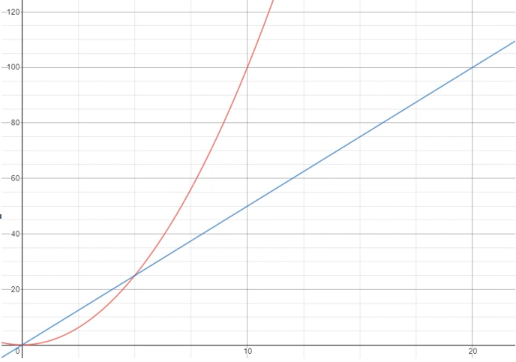
比較寫法一與寫法二，寫法二多使用到了 這個變數，而寫法一並沒有多宣告變數，因此寫法二的空間複雜度明顯較高（比較不好）。

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，所以寫法一花費的時間較多，時間複雜度較高（亦即較差）。

如同前述，通常在意的是時間複雜度，所以實踐上，多數實作交換函式（swap）時都採用寫法二。

（7）只在意資料量大的情形

另外還要特別提到，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。給定兩種算法 A、B，可能會發生「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」必定不大，所以只需在意資料量級很大時哪種算法更有效率。



比較兩種「耗費的時間」對「資料量 」的函數分別為 （上圖紅線）與 （上圖藍線）的演算法：

由上式，知道 時， 比較好，但是一旦 ，也就是資料量超過 5 筆時， 就會比較好（花費的時間較少）。

時花費時間的多寡當然更需要在意，因為 5 筆資料一般來說不可能花費多久時間，處理上萬、上億筆資料時的效率才需要在意，也因此通常會傾向使用 *t* 的演算法，而非 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

同樣的一台電腦上，程式的效能只受到程式碼內容的直接影響嗎？並非如此。

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，數字過大時，就得改用軟體完成，會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（受到輸入資料的影響）。要記得程式碼越短或程式碼上看起來運算次數越少，不等於執行就會越快，這要回到組合語言去看，因為編譯器常默默完成許多程式碼上無法看到的事情。

（2）strlen 的例子

舉 strlen 函式為例，看看編譯器自動採用的執行方式會如何影響到程式效能：

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

假設傳入一個長度為 的字串 ，上面的程式碼可以算出 裡 'a' 這個字元的出現次數。

執行一次 需要進行 次運算，因為這個函式是直接從開頭字元一個一個數出 中字元個數的。再來，因為寫了一個 for 迴圈，迴圈共執行 次，當中每次呼叫 來和 比較時，都需要進行 n 次運算，所以看起來總共需要進行 次運算。

但是實際測試會發現，當資料量變為 倍時（比如 的長度變為 個字元），需要的時間並沒有真的變成 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 的值不會改變，就自動將其看作常數，導致程式的執行複雜度與表面上看起來不同。

### 2. 為什麼使用 C++?

（1）C++ 的優缺點

有些讀者可能有疑問，為什麼本書選擇使用 C++ 來討論資料結構與演算法？

C++ 主要的好處：

A. 內建的 STL 函式庫封裝了常見的資料結構與演算法

B. 支援指標操作，寫鏈結串列 Linked list 時，用指標較有感覺

C. Python 的執行效能較差，比程式競試時可能發生超過時間限制（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。要提醒讀者的是，如果目標是比程式競試或進大公司，因為考驗撰寫出的程式效率，因此要得高分最好使用 C / C++。

（2）本書中許多例子可提交於 LeetCode，提交結果有下列幾種，略作說明：

A. Accepted（AC）：通過

B. Wrong Answer（WA）：執行結果是錯的

C. Time Limit Exceed（TLE）：線上評分網站會擔心使用者提交無窮迴圈，造成資源浪費，所以一旦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（酌參即可，不需了解原理）：

|  |  |
| --- | --- |
| 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，輸出入比較快 | |
| 1  2 | std::ios::sync\_with\_stdio(false);  std::cin.tie(0); |

|  |  |
| --- | --- |
| 讀檔檢查：讓輸出入的資料顯示在螢幕上，就可以檢視現在存取的資料有沒有問題 | |
| 1  2 | freopen("test.in", "r", stdin);  freopen("test.out", "w", stdout); |

## 第三節 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」

* 時間複雜度：運算所需的時間，與運算次數成正比
* 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
* 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，也建議讀者有機會的話可以修習