推薦序

演算法是電腦科學的核心，是當今資訊時代不可或缺的部分。演算法讓搜尋引擎有能力回應每日數十億個網際網路搜尋需求，並為網際網路的通訊保有隱私。從客製化廣告到線上報價，演算法對於無數領域的消費者而言，越來越顯而易見，新聞媒體充斥著演算法是什麼以及演算法能做什麼的論述。

STEM（科學、技術、工程、數學）的蓬勃發展，推動全球經濟持續成長與創新的浪潮。但是，缺乏足夠的電腦科學家探索與應用演算法，以供應醫學、工程、甚至政府的發展所需。我們需要讓更多人知道如何將演算法應用於自己所屬專業領域的問題中。

你不需要有電腦科學學位就可以開始運用演算法。然而，關於這個主題的多數線上教材與教科書都是為科班大學生設計的，重點擺在數學證明與電腦科學概念。演算法教科書可能令人生畏，原因是這些書籍是陳列各種演算法的參考書，內容有無數的變化版本和高度專業的案例。讀者往往很難讀完這些書籍的第 1 章。這些書的用途有點像是試圖閱讀整本字典，來提升自己的英文拼寫能力；不過，若你有一本特別設計的參考書，內容概括 100 個最容易拼錯的單字，並解釋這些單字的掌握規則（和例外），相信你的學習會好很多。同樣的，不同背景和經驗的人要在工作中應用演算法，他們需要更聚焦、更符合所需的參考書。

本書以平易近人的論述，介紹一系列的演算法，你可以立即應用這些演算法來改善你的程式效率。所有演算法都以 Python 實作（Python 是最熱門、易於使用的程式語言之一，從資料科學、生物資訊學到工程學科皆有使用 Python 語言）。這本書仔細說明每個演算法，並以大量圖片輔助讀者掌握基本概念。書中的示例程式是開源的，可從這本書的程式儲存庫免費取得。

本書將教你電腦科學中會用到的基本演算法與資料型別，讓你可以寫出更有效率的程式。若你正在找一份程式設計相關的技術工作，這本書可能會協助你在下次的程式面試中取得好成績。我希望本書能促使你延續學習演算法的旅程。

— *Zvi Galil*
喬治亞理工學院
計算學院 *Frederick G. Storey* 主任暨名譽院長
（*2021* 年 *5* 月於亞特蘭大）

前言

適合閱讀本書的讀者

若你正在閱讀本書，筆者假設你已能運用一種程式語言（譬如：Python）。若你之前從未寫過程式，建議你先學習一種程式設計語言，再來閱讀本書！本書使用 Python，原因是不管是程式設計師或非程式師都可輕易理解這個語言。

演算法的目的是解決軟體程式中常見的問題。筆者教大學生演算法時，試圖弭平學生的背景知識和筆者正在教授的演算法概念兩者之間的差距。許多教科書都認真地描述說明（但總是過於簡要）。若無指南來說明如何瀏覽這些教材，學生往往無法自行學習演算法。

筆者用一段文字與圖 P-1，說明本書的目標。其中將介紹一些資料結構，說明如何使用固定大小的基本型別（譬如：32 位元整數、64 位元浮點數）來組織資訊。某些演算法（譬如二元陣列搜尋）直接運用這些資料結構。較複雜的演算法，尤其是圖演算法，採用許多基本的抽象資料型別（譬如：**堆疊**、**優先佇列**），筆者將依需要來介紹這些型別。這些資料型別會有基本作業，其中可選用正確的資料結構而有效率的實作出這些功能作業。讀完本書，你將了解各種演算法如何達到應有的效能。針對這些演算法，筆者將呈現 Python 的完整實作，或者向你推薦第三方的 Python 套件，引用其中的有效率實作。

若你檢視本書提供的相關程式資源，會看到每一章都有對應一個 `book.py` 的 Python 檔，可以執行該檔案重現本書的所有表格。縱然「結果變化萬千」，但整體趨勢將呈現一致。

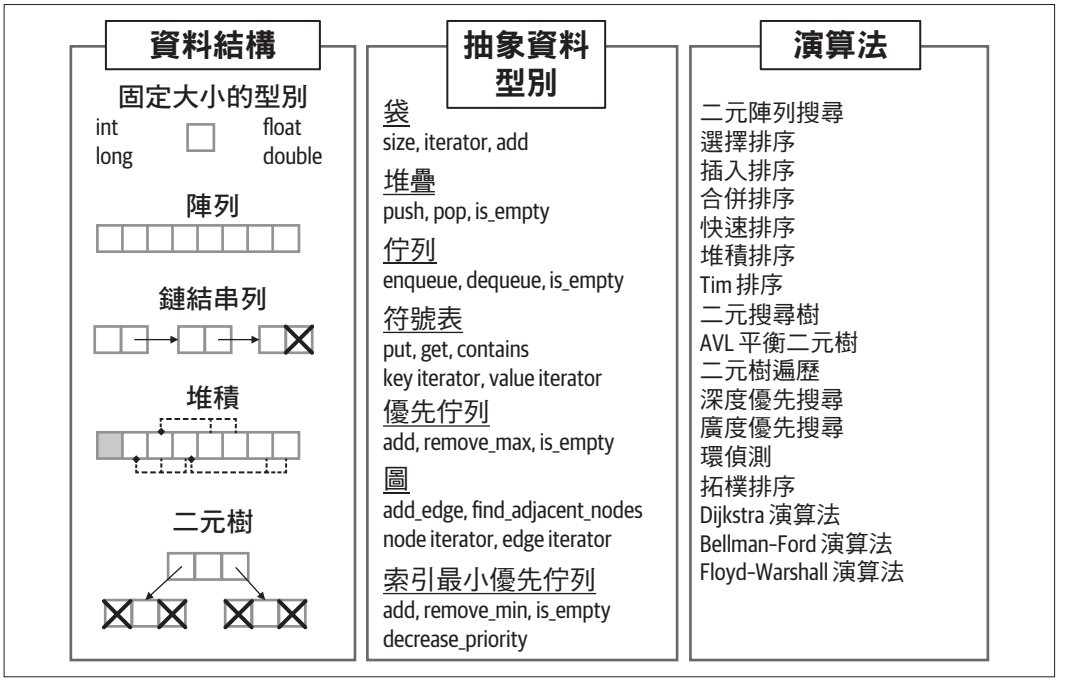


圖 P-1　本書技術內容摘要

本書每一章的尾聲都有挑戰題，讓你有機會應用新學到的知識。筆者建議你，在查看本書附帶的程式儲存庫中相關示例解法之前，先自行試著提出自己的解法。

使用示例程式

本書所有程式都可以在相關的 GitHub 儲存庫中找到（*http://github.com/heineman/LearningAlgorithms*）。這些程式適用於 Python 3.4 以上的版本。本書在適當之處採雙底線做法（例如：`__str__()`、`__len__()`），此為 Python 內建方法的最佳慣例。本書的示例程式會以兩空格為縮排單位降低程式碼印刷所占的頁寬比重；程式儲存庫使用標準的四空格為縮排單位。少數示例程式會使用單行縮寫格式的 `if` 陳述式，譬如：`if j == lo: break`。

本書程式用到三個外來的開源 Python 函式庫：

- NumPy（*https://www.numpy.org*）版本：1.19.5
- SciPy（*https://www.scipy.org*）版本：1.6.0
- NetworkX（*https://networkx.org*）版本：2.5

NumPy、SciPy 是最常用的開源 Python 函式庫，擁有廣泛的使用者。本書採用這些函式庫來分析實證的執行時間效能。NetworkX 則針對圖的運作提供廣泛的高效率演算法，如第 7 章所述；它還有立即可用的圖資料型別實作。使用這些函式庫可確保我們不會非必要地重新創造輪子。若讀者沒有安裝這些函式庫也無妨，本書仍有提供替代方法。

本書呈現的所有計時結果都使用 `timeit` 模組，重複執行一個程式片段做統計。程式片段往往會重複執行多次，以確保能夠準確測量出結果。多次執行之後，取最短時間作為計時結果，**而非**取所有執行作業的平均值。如此做法一般被認為是產生準確計時結果的最高效率方法。當某些效能執行作業受到外部因素（例如作業系統中執行的其他工作）影響時，就數次的執行作業取平均值可能會扭曲計時結果。

當演算法的效能對輸入內容有高度敏感時（例如第 5 章中的插入排序），筆者將明確表示，取其效能執行作業的平均值。

程式儲存庫內有超過 10,000 行的 Python 程式碼，其中包含用於執行所有測試案例與計算書中表格內容的 script；也能重現諸多圖表、圖形。程式採用 Python docstring 慣例註解描述，而程式碼涵蓋率為 95%（以 *https://coverage.readthedocs.io* 計算）。

若有技術問題或使用示例程式的疑問，請利用電子郵件詢問（寄至 *bookquestions@oreilly.com*）。

本書目的是為了幫助讀者完成相關的工作。一般來說，讀者可以把書中提供的示例程式，應用於自己工作相關的程式或文件中。除非要將書中程式的重大內容重製，否則不需要與我們聯繫取得許可。例如：讀者撰寫的程式有使用到書中的數個程式區塊，這樣是不需要經過授權程序。至於散佈或販賣 O'Reilly 出版書籍中的示例程式，則需要取得授權許可。讀者可以自由引用本書內容或示例程式來解決問題。若要將書中大量的示例程式放到自己的產品文件裡，請事先取得授權同意。

第一章

解決問題

你將於本章學到：

- 解決示例問題所用的多種演算法。
- 如何研究演算法效能（就問題實例大小 N 而言）。
- 如何計數特定問題實例的關鍵解決作業（運算）叫用次數。
- 如何找出問題實例大小加倍之際的成長等級。
- 如何以演算法（針對問題實例大小 N 而論）執行關鍵作業次數的計數，估計**時間複雜度**。
- 如何以演算法（針對問題實例大小 N 而論）的記憶空間需求量，估計**空間複雜度**。

開始進入主題吧！

何謂演算法？

解釋演算法（algorithm）的運作方式就像說故事一般。每個演算法會納入新概念或新方法，改進原本的解法。本章針對簡單問題探討數個解法，用以解釋演算法效能的影響因素。其中將介紹演算法效能（performance）分析技術（雖然過程中皆會呈現實作內容，加以實證，不過這些分析技術**與演算法的實作無關**）。

演算法是按部就班（逐步）解決問題的方法，以電腦程式實作，在可預測的時間內傳回正確結果。演算法的研究涉及兩者：正確性（即此演算法可針對所有輸入而正常運作嗎？）與效能（即此乃問題的最有效率解法嗎？）。

讓我們以問題解法範例說明相關實務內容為何。若你想找出無序串列（list）內最大值，該怎麼辦？圖 1-1 的每個 Python 串列皆為**問題實例**（*problem instance*），即由某個演算法（以圓柱表示）處理的輸入資料；正確答案則呈現於右邊。如何實作此演算法？如何針對不同問題實例執行此演算法？你能夠預測從一百萬個內容值串列找出最大值所需的作業時間嗎？

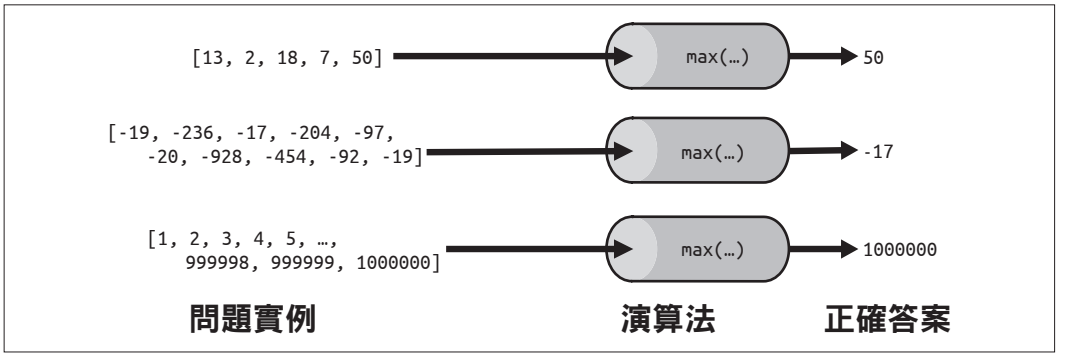


圖 1-1　由某個演算法所處理的三個問題實例

演算法不只是問題的解法；它的實作程式還需要在可預測的時間內執行完成。Python 內建函式 `max()` 已解決上述問題。目前，針對內含隨機資料的問題實例，難以預測演算法的效能，因此得明辨細膩建構的問題實例。

表 1-1 呈現兩種問題實例（大小皆為 N）的 `max()` 執行時間（計時結果）：其中一種是串列內含升序排列的整數，另一種是串列包含降序排列的整數。雖然讀者的執行結果可能與此表內容有所出入，但是基於讀者的運算系統（電腦）組態，我們可以驗證下列兩項陳述：

- 若 N **足夠大**，對於升序排列內容的 `max()` 執行時間總是比降序排列內容的結果慢。
- 當 N 值增加十倍時，`max()` 的對應時間似乎也增加十倍（雖然會有些微偏差），此乃依目前這些效能試驗所預期的結果。

解決此問題，會傳回（輸出）最大值，而輸入內容不變。但某些情況中，演算法直接變更問題實例內容視為結果，而非產出新值——譬如：串列內容的排序（第 5 章所述）。本書以 N 表示問題實例的大小。

表 1-1　兩種問題實例（大小皆為 N）的 max() 執行表現（單位：ms）

N	升序排列值	降序排列值
100	0.001	0.001
1,000	0.013	0.013
10,000	0.135	0.125
100,000	1.367	1.276
1,000,000	14.278	13.419

就執行時間而言：

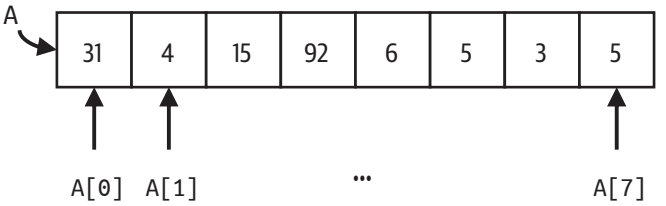
- 因為運算平臺差異、所用的程式語言不同，所以我們不能**事先**預測 T(100,000) 值——即演算法解決該問題實例（大小為 100,000）所需的時間。
- 然而，一旦我們憑實證找出 T(10,000) 的結果，就可以預測 T(100,000)——即解決該問題實例的時間將為十倍之多——不過難免有某種程度的預測不準確。

演算法設計的主要挑戰是確保演算法的正確性，即可**針對所有輸入**正常運作。第 2 章將以更多篇幅說明各種演算法（解決相同問題）的行為分析比較方式。演算法分析領域與現實生活注重的問題研究息息相關。儘管對於演算法的數學理解可能會有難度，但本書將提供具體示例，始終將抽象概念與實際問題相連。

判斷演算法效率的標準方法是計數演算法所需的**運算作業**（*computing operation*）次數，不過此計算窒礙難行！電腦有個中央處理單元（CPU），可執行**機器指令**（*machine instruction*），以用於數學運算（譬如：加法、乘法）、對 CPU 暫存器（register）指派值、兩值相較等等。某些高階程式語言（像 C、C++）程式被編譯成機器指令。其他語言（如 Python、Java）程式被編譯為中間**位元組碼**（*byte code*）。Python **直譯器**（*interpreter*）執行位元組碼（此直譯器本身是以 C 語言所寫的程式），Python 內建函式，譬如：`min()`、`max()`，皆以 C 語言實作，最終編譯成機器指令供 Python 程式執行使用。

全能的陣列

陣列（*array*）以記憶體的連續區塊儲存 N 個值的集合。此為程式設計師用於儲存多值的容器，是最悠久可靠的資料結構之一。下圖表示內含八個整數的陣列。



陣列 A 按其位置索引（index）可存取八個值。例如：A[0] = 31、A[7] = 5。A 的內容值可為任何型別，譬如字串或複雜物件。

下列是陣列相關的重要須知：

- 第一個值 A[0]，其為位置索引 0 之處的內容；最後一個值是 A[N – 1]，其中 N 是陣列的大小。
- 陣列的長度固定，不得隨意變動。Python、Java 程式可於執行期（runtime）決定陣列的長度，C 程式則不允許（譯註：即在編譯期決定長度）。
- 可按位置**索引值** *i* 存取個別位置內容 A[i]，此索引值為整數，範圍是 0 ～ (N – 1)。
- 陣列不能擴大（縮小）；而是配置新需求長度的陣列，複製應保留的舊內容值。

儘管陣列簡單易懂，但是此結構是相當通用而有效率的資料組織方式。雖然 Python 的 `list` 物件功能強大，不過可將它視為陣列（原因是其大小可隨著時間增減）。

計數演算法所執行的機器指令總量幾乎是不可能的事，何況當今 CPU 每秒可以執行數十億個指令！取而代之的是，計數每個演算法**關鍵作業**（*key operation* 或**關鍵運算**）的叫用次數，該次數可能是「陣列兩內容值相較次數」、「函式呼叫次數」。而對於 `max()` 而言，關鍵作業次數是「**小於**（<）運算子叫用次數」。第 2 章將擴展此計數原則。

此刻正是揭開 max() 演算法面紗的好時機，可對其所作所為與來龍去脈一探究竟。

找出任一串列的最大值

以示例 1-1 有缺失的 Python 實作為例，其中試圖找出任一串列的最大值（串列**至少要有一個內容值**），將 A 每個內容與 my_max 相比，若該內容值較大，則需要以此值更新 my_max。

示例 1-1　找出串列中最大值（有缺失的實作）

```
def flawed(A):
  my_max = 0          ❶
  for v in A:         ❷
    if my_max < v:
      my_max = v      ❸
  return my_max
```

❶ my_max 變數儲存最大值；在此 my_max 初始值為 0。

❷ for 迴圈 v（定義變數），疊代處理 A 的每個元素。對於每個內容值 v，if 陳述式將執行一次。

❸ 若 v 值較大，則更新 my_max。

此解決方案的核心是小於運算子（<）：比較兩數值，判斷何者較小。圖 1-2 的 v 從 A 中連續取出內容值，其中 my_max 內容更新三次，進而找出 A 中最大值。flawed() 找出 A 中最大值，「小於」運算叫用六次（對每個內容值執行一次）。以大小為 N 的問題實例而言，flawed() 的「小於」作業會叫用 N 次。

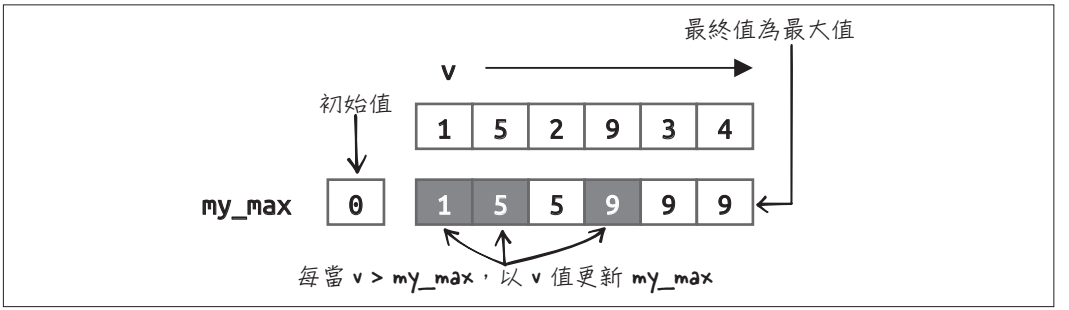


圖 1-2　flawed() 執行過程（視覺化呈現）

上述實作有缺點，因其假設 A 中至少有一個值會大於 0。所以 flawed([-5,-3,-11]) 的運算會傳回 0，這是不正確的結果。常見的修正是將 my_max 以可能的最小值初始化，譬如 my_max = float('-inf')。這種方法仍有缺陷，倘若 A 為空串列 []，則會傳回此初始值。因而要修正此缺陷。

Python 陳述式 range(x,y) 產生從 x 直到 y（不含 y）的整數。另外我們若採用 range(x,y,-1)，則產生的整數內容是從 x 開始以倒數順序排列直到 y（不含 y）。因此 list(range(1,7)) 的結果是 [1,2,3,4,5,6]，而 list(range(5,0,-1)) 則為 [5,4,3,2,1]。我們可用任何遞增值（increment）計數，因此 list(range(1,10,2)) 產生 [1,3,5,7,9]，即此串列中連續兩內容值彼此的差值皆為 2。

計數關鍵作業

實際上，最大值必然位於 A 中，所以示例 1-2 正確的 largest() 函式選擇 A 的第一個內容值作為 my_max 的初始值，與其他內容值相較，確認何者較大。

示例 1-2　找出串列中最大值（正確函式實作）

```
def largest(A):
  my_max = A[0]                 ❶
  for idx in range(1, len(A)):  ❷
    if my_max < A[idx]:
      my_max = A[idx]           ❸
  return my_max
```

❶ 設定 my_max 為 A 的第一個內容值（即位置索引 0 之處的內容）。

❷ idx 取用的整數，範圍從 1 直到 len(A)（不包含 len(A)）。

❸ 若 A 中位置 idx 的內容值較大，則以此內容值更新 my_max。

若叫用 largest()、max() 時傳入空串列，將引發 ValueError: list index out of range（串列索引超出範圍）例外（exception）。這些執行期例外屬於程式設計師造成的錯誤，其中的含意是：並不曉得 largest() 需要使用的 list 至少要有一個內容值。

此刻我們已完成演算法的正確實作（Python 函式），就該新演算法來看，可以確定其「小於」判斷運算的叫用次數嗎？嗯！是 N – 1 次。我們已修正前述演算法的缺陷，改善效能表現（當然，這只是些微的變化）。

為什麼計數「小於」的運用次數很重要？此乃兩值相較所需的關鍵作業（運算）。至於實作內容的其他陳述式（譬如 `for`、`while` 迴圈）可依所用的程式語言任意選擇。下一章將擴展此一概念，而目前計數關鍵作業足矣。

能夠預測演算法效能的模型

要是有人以不同演算法解決同一個問題，會怎麼樣？我們如何決定使用哪個演算法？以示例 1-3 的 `alternate()` 演算法為例，其逐一檢查 `A` 中所有值，確認是否大於或等於相同串列中其他內容值。此演算法會傳回正確結果嗎？對於大小為 N 的問題而言，其「小於」判斷作業的叫用次數為何？

示例 1-3　找出 A 中最大值（不同的實作方法）

```
def alternate(A):
  for v in A:
    v_is_largest = True          ❶
    for x in A:
      if v < x:
        v_is_largest = False     ❷
        break
    if v_is_largest:
      return v                   ❸
  return None                    ❹
```

❶ 疊代處理 `A` 時，假設每次的 `v` 值為最大值。

❷ 若 `v` 小於另一值 `x`，則停止比較，記錄該 `v` 值不是最大值。

❸ 若 `v_is_largest` 為 `true`，則傳回該 `v` 值（該值為 A 中最大值）。

❹ 若 `A` 為空串列，則傳回 `None`。

`alternate()` 試圖找出 `A` 中某個值 `v`，使得 `A` 中其他值 `x` 不會大於該值。此實作使用兩層 `for` 迴圈（巢狀迴圈）。在此並非純粹計數「小於」的叫用次數，當某 `x` 值大於 `v` 時，內層（inner）迴圈 `x` 立即停止作業。此外，一旦找到最大值（`v` 值），外層（outer）迴圈 `v` 會停止作業。圖 1-3 視覺化呈現 `alternate()` 的執行情況（針對前述的串列範例而論）。

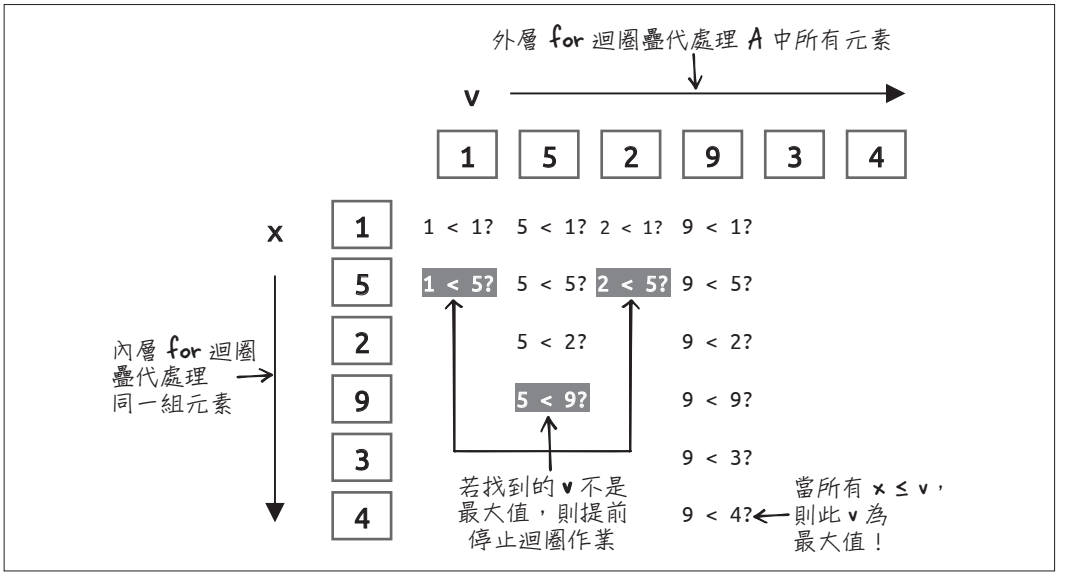


圖 1-3　alternate() 執行過程（視覺化呈現）

針對此問題實例而言，「小於」作業叫用 14 次。不過此演算法的實際總次數取決於串列 A 中特定內容。要是內容值以不同順序排列，會怎麼樣？我們可以想出演算法「小於」作業使用次數需求最少的內容值排列情況嗎？其中會將這樣的問題實例視為 alternate() 的**最佳情況**（*best case*）。例如，若 A 的第一個內容值是 N 個值中最大的，則「小於」作業的使用總數恆為 N。相關概述如下：

最佳情況（*best case*）

演算法執行工作需求量最少的問題實例（其中問題實例大小為 N）

最差情況（*worst case*）

工作需求量最多的問題實例（其中問題實例大小為 N）

設法確認 alternate() **最差情況**（*worst case*）的問題實例，即「小於」作業運用次數需求最大的實例。就 alternate() **最差情況**的問題實例而言，並非只是最大值位於 A 中最後位置，A 的內容值還必然以升序排列呈現。

圖 1-4 的視覺化內容，上半部為**最佳情況**（即：p = [9,5,2,1,3,4]），下半部為**最差情況**（即：p = [1,2,3,4,5,9]）。

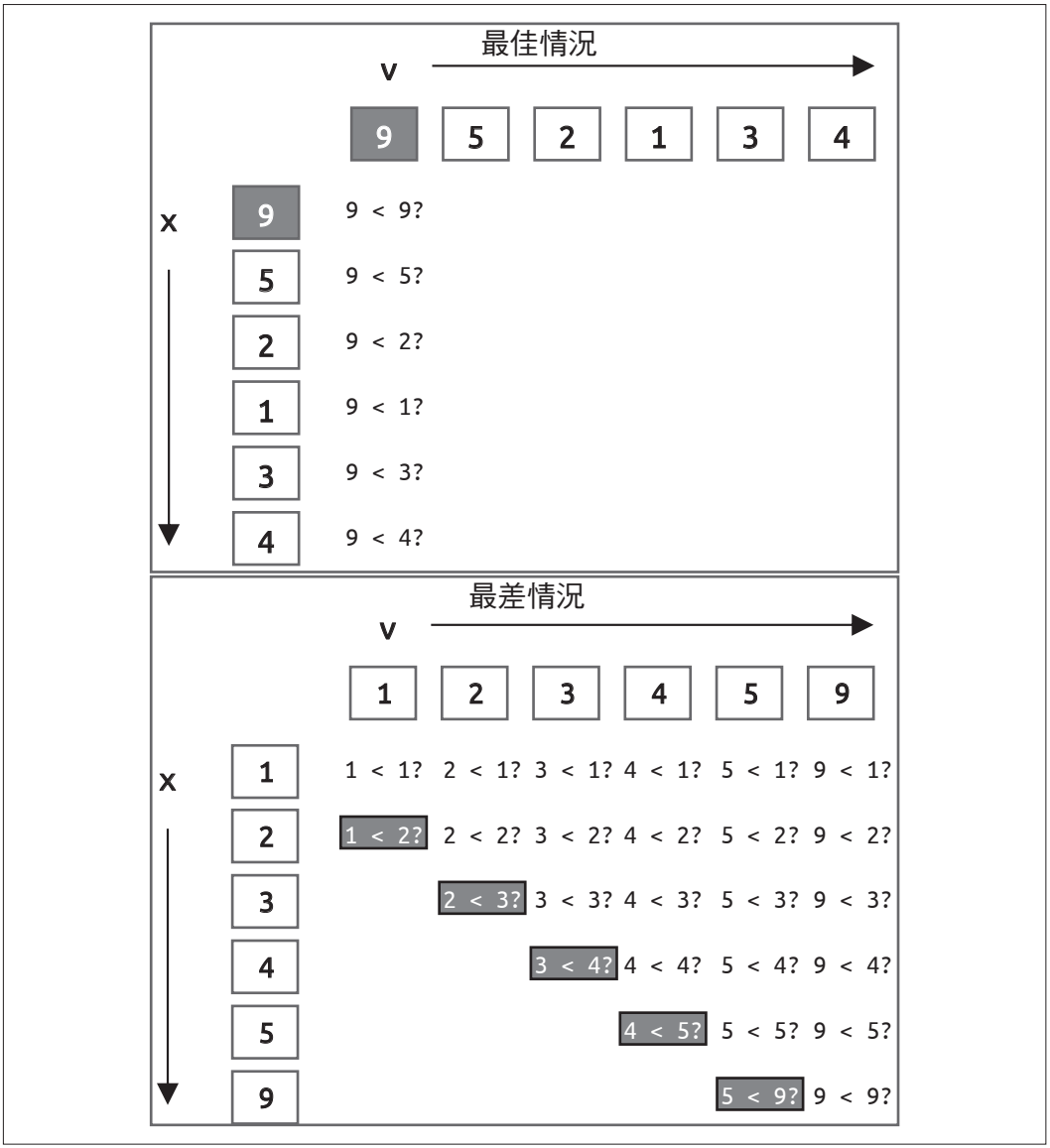


圖 1-4　alternate() 執行過程（視覺化呈現最佳情況與最差情況）

最佳情況有六次「小於」判斷作業；若最佳情況有 N 個值，則所用的「小於」作業總數為 N。針對最差情況而言則有些複雜。圖 1-4 的串列（內含 N 個值）若以降序排列，則總共執行 26 次「小於」運算。以簡單的數學式子而言，N 個值的最差情況作業計數恆為 $(N^2 + 3N - 2)/2$。

表 1-2 針對最差情況的問題實例（大小為 N）呈現 `largest()`、`alternate()` 執行實證。

表 1-2　`largest()` 與 `alternate()` 相較表現（針對最差情況的問題實例而言）

N	largest （#「小於」作業次數）	alternate （#「小於」作業次數）	largest （單位：ms）	alternate （單位：ms）
8	7	43	0.001	0.001
16	15	151	0.001	0.003
32	31	559	0.002	0.011
64	63	2,143	0.003	0.040
128	127	8,383	0.006	0.153
256	255	33,151	0.012	0.599
512	511	131,839	0.026	2.381
1,024	1,023	525,823	0.053	9.512
2,048	2,047	2,100,223	0.108	38.161

對於小量的問題實例來說，兩者的表現似乎不差，然而隨著問題實例大小加倍，`alternate()` 使用的「小於」作業次數大致增為四倍，遠多於 `largest()` 運用的數量。表 1-2 右邊兩行（column）顯示的執行效能，是這兩種實作針對問題實例（大小為 N）執行 100 次隨機試驗的結果。`alternate()` 的執行時間大致也是以四倍成長。

筆者測量演算法處理隨機問題實例（大小為 N）所需的執行時間。其中從整組執行項目中，選擇最快的完成時間（即最短的時間）。比起對所有項目的執行時間直接取總平均，此方式較合適（取平均值可能會扭曲結果）。

本書將呈現像表 1-2 這樣的表格，內容包含關鍵作業（在此為小於運算子）的執行總數以及執行時間。每列（row）資料表示不同大小（N）的問題實例表現，由上到下閱讀表格內容，了解每行內容的變化程度（隨著問題實例大小加倍所呈現的結果）。

計數「小於」運用次數，即詮釋 `largest()`、`alternate()` 的行為。隨著 N 加倍，`largest()` 的「小於」叫用次數加倍，而 `alternate()` 的次數則為**四倍**。上述的行為表現始終如一，我們可以針對較大規模的問題，使用此資訊預測這兩個演算法的執行時間。圖 1-5 描繪 `alternate()` 的「小於」運用計數（左邊 y 軸）及其執行時間（右邊 y 軸），呈現彼此的直接相關程度。

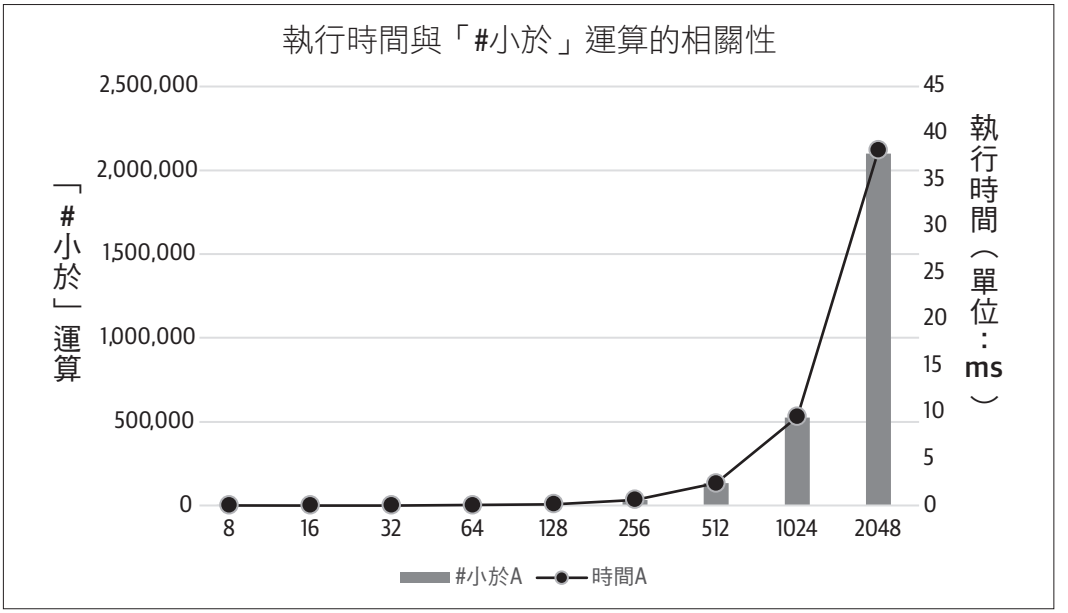


圖 1-5　「小於」運算次數與執行時間的關係

可喜可賀！方才我們執行了演算法分析的關鍵步驟：以關鍵運算的執行次數計數，斷定兩個演算法的相對效能。當然讀者可以持續實作兩者的變動（如同筆者在此所為），而當問題實例大小增為兩倍時，測量各自的執行時間；不過因為該模型已預測此一行為，確認 `largest()` 是兩者之中較有效率的演算法，所以沒有必要如此為之。

`largest()` 與 `max()` 實作同一個演算法，然而如表 1-3 所示，`largest()` 的執行時間始終比 `max()` 慢，通常慢四倍。原因是 Python 為直譯式語言，其中將程式碼編譯成中間位元碼，而以 Python 直譯器（interpreter）執行。因為內建函式（如：`max()`）實作於直譯器之中，所以內建函式始終優於為相同目的而實作的 Python 程式碼。其中值得關注的是，在所有情況下，當 N 加倍，`largest()`、`max()` 的執行時間——對於最佳情況與最差情況來說——也是加倍。

表 1-3 顯示，解決大小增加的問題實例，可以預測其中所需的時間。對於大小為 N 的問題實例而言，一旦我們知道 `largest()`、`max()` 在最佳情況或最差情況的執行時間，就可以預測該演算法的執行時間將隨 N 加倍而加倍。此刻，將問題稍作改變，讓內容更有趣。

表 1-3　largest()、max() 的效能（最佳情況與最差情況）

N	largest() 最差情況	max() 最差情況	largest() 最佳情況	max() 最差情況
4,096	0.20	0.05	0.14	0.05
8,192	0.40	0.11	0.29	0.10
16,384	0.80	0.21	0.57	0.19
32,768	1.60	0.41	1.14	0.39
65,536	3.21	0.85	2.28	0.78
131,072	6.46	1.73	4.59	1.59
262,144	13.06	3.50	9.32	3.24
524,288	26.17	7.00	18.74	6.50

找出任一串列的前兩大值

設計演算法，可找出任何串列中前兩大值。也許我們可以修改現有的 largest() 演算法（只需稍作調整）。讀者何不自行嘗試解決這個修改題，然後將自己的解法呈現於此？示例 1-4 為筆者的解法。

示例 1-4　找出前兩大值（largest() 稍改版）

```
def largest_two(A):
  my_max,second = A[:2]                    ❶
  if my_max < second:
    my_max,second = second,my_max

  for idx in range(2, len(A)):
    if my_max < A[idx]:                    ❷
      my_max,second = A[idx],my_max
    elif second < A[idx]:                  ❸
      second = A[idx]
  return (my_max, second)
```

❶ 確保 my_max、second 為 A 中前兩項值（按降序指派）。

❷ 若 A[idx] 是新出現的最大值，則將 my_max 設為 A[idx]，而將 my_max 原值放入 second 中。

❸ 若 A[idx] 大於 second（而小於 my_max），則僅更新 second 的內容。

largest_two() 與 largest() 似乎雷同：將 A 中前兩項值（以對應順序排列的）指派給 my_max、second。接著對於 A 中其餘值（有多少個？答案是 N – 2 個！），若你遇到某個

A[idx] 大於 my_max，則調整 my_max、second 兩者的內容，否則檢查是否僅要更新 second 的內容。

「小於」叫用次數的計數較為複雜，原因又是**取決於問題實例的內容**。

for 迴圈內的 if 陳述式條件為真時，largest_two() 的「小於」叫用次數最少。若 A 的內容值以升序排列，此「小於」條件始終為真，所以會被叫用 N – 2 次；因為此函式在開頭有用到「小於」，所以不要忘記額外加 1 次。因此，**最佳情況**，我們只需要 N – 1 次的「小於」叫用即可找出前兩大值。**最佳情況**下始終不會用到 elif 條件。

對於 largest_two() 而言，你可以建構出**最差情況**的問題實例嗎？讀者可自行嘗試看看：只要 for 迴圈內的 if 的「小於」條件每次皆為 False，就會發生這種情況。

筆者可以肯定的是，A 的內容值以降序排列時，largest_two() 的「小於」叫用次數最多。特別是，針對**最差情況**而言，for 迴圈的每次作業中「小於」會用到兩次，或總共使用 $1 + 2 \times (N - 2) = 2N - 3$ 次。基於某種原因，這似乎是對的，不是嗎？若我們需要運用 N – 1 次的「小於」才能找到 A 中最大值，也許確實需要額外的 N – 2 次「小於」運算（其中理應排除最大值），才能找到第二大值。

largest_two() 的行為總結如下：

- 對於**最佳情況**，需叫用 N – 1 次「小於」運算找出前兩大值。
- 對於**最差情況**，需叫用 2N – 3 次「小於」運算找出前兩大值。

我們完成演算法了嗎？這是找出任何串列中前兩大值的解題「最佳」演算法嗎？我們可以基於多個因素，決定選擇某個演算法，而不採用另一個演算法：

額外的儲存需求（*required extra storage*）

該演算法是否需要複製原本的問題實例？

程式設計負荷（*programming effort*）

程式設計師必須撰寫幾行程式碼？

可變輸入（*mutable input*）

該演算法會**直接**變更問題實例的輸入內容，或維持原狀？

速度（*speed*）

該演算法有優於其他演算法嗎（效能不受輸入內容所影響）？

接著探討解決相同問題所用的三種不同演算法，如示例 1-5 所示。`sorting_two()` 建立新的串列（內含 A 的降序排列項目），取此串列前兩項內容值，並以元組（tuple）形式傳回該結果。`double_two()` 使用 `max()` 找出 A 中最大值，將其從 A 的副本中移除，針對此副本其餘內容，再以 `max()` 找出 A 中第二大值。`mutable_two()` 找出 A 中最大值的位置，將其從 A 中移除；隨後將 A 中其餘內容的最大值指派給 `second`，接著將 `my_max` 值重新插入其原位置。前兩個演算法有額外的儲存需求，第三個演算法會修改輸入內容：三個演算法僅適用於多個內容值的問題實例。

示例 1-5　三種方法（運用 Python 工具函式）

```
def sorting_two(A):
  return tuple(sorted(A, reverse=True)[:2])    ❶

def double_two(A):
  my_max = max(A)                              ❷
  copy = list(A)
  copy.remove(my_max)                          ❸
  return (my_max, max(copy))                   ❹

def mutable_two(A):
  idx = max(range(len(A)), key=A.__getitem__)  ❺
  my_max = A[idx]                              ❻
  del A[idx]
  second = max(A)                              ❼
  A.insert(idx, my_max)                        ❽
  return (my_max, second)
```

❶ 建立新串列（以降序排列對 A 排序），並傳回串列的前兩大值。

❷ 使用內建函式 `max()` 找出最大值。

❸ 建立 A 的原始副本，將其中 `my_max` 項目移除。

❹ 結果以元組傳回（內含 `my_max` 值與 `copy` 裡的最大值）。

❺ 此 Python 技巧找出 A 中最大值的*索引位置*（*index location*），而非內容值本身。

❻ 將 A 中此位置內容記錄於 `my_max`，接著從 A 中刪除此項內容。

❼ 在其餘內容中用 `max()` 找出結果。

❽ 將 `my_max` 插入其原位以還原 A 內容。

這些方法並無直接運用「小於」，而是使用現有的 Python 函式庫。`sorting_two()`、`double_two()` 兩者皆有 A 陣列副本，這似乎非必要，`largest_two()` 就沒有採用。此外，

僅為了找出前兩大值而**將整個串列排序**，似乎處理過度。如同分析執行時間效能的作業計數一樣，我們將估算演算法所用的**額外儲存**（*extra storage*）空間——對於前述兩種方法，儲存量皆與 N 成正比。第三個方法 `mutable_two()` 先將 A 的最大值刪除，短暫變更內容，稍後再將此值加回去。原串列內容的變更可能會讓呼叫者感到意外。

筆者稍微運用 Python 技巧，以特定的 `RecordedItem` 類別精確計算「小於」的叫用次數[1]。表 1-4 顯示，對於升序排列的內容值，`double_two()` 的「小於」運算叫用次數最多，而對於降序排列的內容值，`largest_two()`（與另外兩個方法）的「小於」運算叫用次數最多。該表最後一行（column），標題為「奇偶交替」，是將 524,288 個值，區分奇偶兩種數值交替陳列，其中偶數部分以升序排列，奇數則以降序排列：對於 N = 8，輸入內容將為 `[0,7,2,5,4,3,6,1]`。

表 1-4　各種方法針對 524,288 個內容值的效能表現（內容以三種順序排列）

演算法	升序排列	降序排列	奇偶交替
largest_two	524,287	1,048,573	1,048,573
sorting_two	524,287	524,287	2,948,953
double_two	1,572,860	1,048,573	1,048,573
mutable_two	1,048,573	1,048,573	1,048,573
tournament_two	524,305	524,305	524,305

接著我們要論述的 `tournament_two()` 演算法，不論輸入內容為何，其中的「小於」叫用次數最少。籃球迷對於該演算法的設計邏輯應該不陌生。

對於解決已知問題的某演算法而言，若我們知道該演算法的最差情況問題實例，則解同一個問題的其他演算法，不見得會受到該問題實例的負面影響。不同演算法可能有不同缺點，透過細膩分析可揭露箇中缺陷。

錦標賽演算法

單淘汰錦標賽乃由多支競賽隊伍組成，爭奪冠軍。理想而言，隊數是 2 的冪，比如 16、64。錦標賽分為數輪（回合），賽程中留下的各個隊伍兩兩相對競賽；比賽落敗者會被淘汰，而獲勝者晉級下一輪。最終留下的那支隊伍即為該錦標賽冠軍。

1　叫用 `__lt__()` 小於運算子（或 `__gt__()` 大於運算子）時，`RecordedItem` 包裹類別（wrapper class）將覆寫（override）運算子的計數。

以 N = 8 個值的問題實例 p = [3,1,4,1,5,9,2,6] 為例。圖 1-6 呈現的單淘汰賽中，首輪就八個值用「小於」比較兩兩相鄰項；較大者獲得比賽晉級 [2]。在八強對決賽中，有四個值被淘汰，而留下 [3,4,9,6]。在四強對決賽中，[4,9] 晉級，最終 9 獲得冠軍。

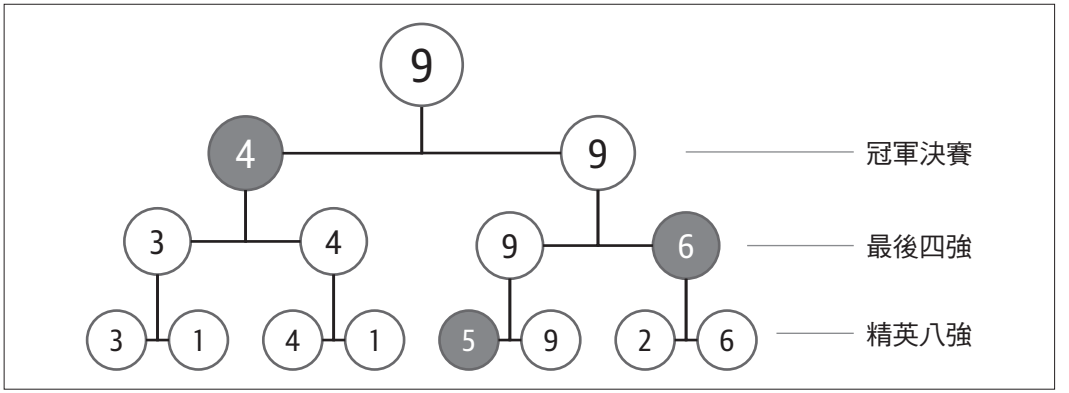


圖 1-6　有八個初始值的錦標賽

此錦標賽中，「小於」叫用七次（即：每場比賽用一次），如此結果應該不用擔心，這表示叫用 N – 1 次「小於」可找到最大值（如之前所述）。若我們儲存 N – 1 場的比賽結果，則可以快速找到第二大值（如此例所示）。

當 9 成為冠軍之際，第二大值「藏」於何處？因為 4 是冠軍賽中落敗值，起初將它視為第二大值可能之選。不過最大值 9 的決賽之前有兩場比賽，因此我們必須檢查另外兩個落敗值——最後四強對決（前一輪）的值 6、精英八強對決（前二輪）的值 5。因此，第二大值為 6。

對於 8 個初始值而言，我們只需要額外 2 次的「小於」叫用——4 < 6？及 6 < 5？——即可確定 6 是第二大值。因 $8 = 2^3$，所以需要 $3 - 1 = 2$ 次的比較，如此並非巧合。結果是，對於 $N = 2^K$，需要額外的 K – 1 次比較，其中 K 為比賽輪數。

若有 $8 = 2^3$ 個初始值，演算法建構的錦標賽將有 3 輪比賽。圖 1-7 視覺化呈現 32 個值所構成的錦標賽（共有五輪比賽）。若錦標賽的內容值數量加倍，則只需要額外增加一輪比賽即可；換句話說，每新增第 K 輪比賽，就可以多加 2^K 個值。想要找到 64 個值中的最大值嗎？因為 $2^6 = 64$，所以只需要 6 輪比賽即可。

2　若某場比賽兩個值相等，則只有其中一個值會晉級。

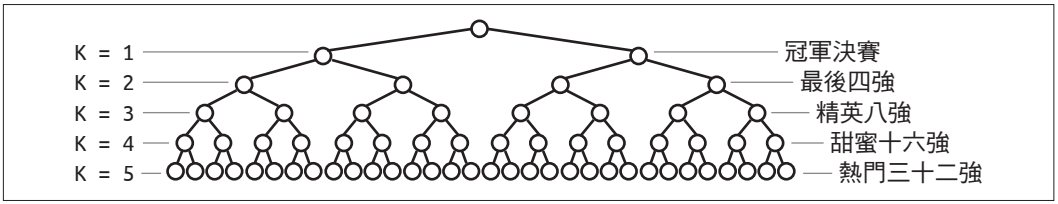


圖 1-7　有 32 個初始值的錦標賽

要得知任何 N 值的比賽輪數，可使用對數（*logarithm*）函數——log()，其為指數函數的反函數。若有 N = 8 個初始值，因為 $2^3 = 8$ 與 $\log_2(8) = 3$，所以該錦標賽需要 3 輪比賽。就本書及傳統電腦科學（computer science 或稱作計算機科學）而言，log() 運算子以 2 為底。

大多數手持式計算機（calculator）對於 log() 的計算是以 10 為底。函數 ln() 則表示以 e（約為 2.718）為底的自然對數。若要使用計算機（或 Microsoft Excel）快速算出（以 2 為底的）log(N)，則可改算 log(N)/log(2) 得到相同結果。

若 N 為 2 的冪（如：64、65,536），則該錦標賽的比賽輪數為 log(N)，其中額外的「小於」叫用次數為 log(N) – 1。示例 1-6 實作的演算法，使用額外儲存空間記錄所有比賽結果，進而讓「小於」叫用次數最小化。

示例 *1-6*　找出 *A* 中前兩大值的演算法（錦標賽法）

```
def tournament_two(A):
  N = len(A)
  winner = [None] * (N-1)          ❶
  loser = [None] * (N-1)
  prior = [-1] * (N-1)             ❷

  idx = 0
  for i in range(0, N, 2):         ❸
    if A[i] < A[i+1]:
      winner[idx] = A[i+1]
      loser[idx] = A[i]
    else:
      winner[idx] = A[i]
      loser[idx] = A[i+1]
    idx += 1

  m = 0                            ❹
```

```
while idx < N-1:
  if winner[m] < winner[m+1]:     ❺
    winner[idx] = winner[m+1]
    loser[idx]  = winner[m]
    prior[idx]  = m+1
  else:
    winner[idx] = winner[m]
    loser[idx]  = winner[m+1]
    prior[idx]  = m
  m += 2                          ❻
  idx += 1

largest = winner[m]
second = loser[m]                 ❼
m = prior[m]
while m >= 0:
  if second < loser[m]:           ❽
    second = loser[m]
  m = prior[m]

return (largest, second)
```

❶ 這些陣列儲存 `idx` 比賽的贏家、輸家；該錦標賽將有 N – 1 場比賽。

❷ 當某個值於 `m` 比賽中晉級，`prior[m]` 記錄該值的前場比賽（若為初場比賽，該值記為 `-1`）。

❸ 使用 N/2 次的「小於」叫用，初始化前 N/2 場贏家 / 輸家組。這些為最初一輪的比賽結果。

❹ 獲勝者兩兩對決，找出新的贏家，記錄 `prior` 比賽索引。

❺ 需要額外 N/2 – 1 次的「小於」叫用。

❻ `m` 加 2，處理下一組贏家對決。若 `idx` 為 N – 1 時，`winner[m]` 為最大值。

❼ 此為初始的第二大值可能之選，不過必須檢查輸給 `largest` 的其他值，以找到真正的第二大值。

❽ 額外的「小於」叫用次數最多為 log(N) – 1 次。

圖 1-8 呈現此演算法的執行情況。進行初始化步驟之後，原陣列 `A` 的 N 個值會分成 N/2 組 `winner`、`loser`；以圖 1-6 的範例而論，會分為四組。`while` 迴圈的每個晉級步驟中，連續兩場比賽（`m`、`m + 1`）的贏家對決之後的獲勝者與落敗者，分別放在 `winner[idx]` 與 `loser[idx]` 中；`prior[idx]` 記錄贏家的前場比賽（以由右往左的箭頭描繪表示）。進行三個晉級步驟之後，儲存所有比賽資訊，演算法檢視先前與該贏家比賽的所有輸家。此視

覺化的內容是依循該贏家的箭頭往回追溯，直到無箭頭可追。如圖可見，第二大值可能之選位於 loser[6] 中：只需兩次的「小於」叫用（針對 loser[5]、loser[2]），即可確定何者最大。

筆者方才已勾勒出一個演算法，用於求出 A 中最大值與第二大值，對於任何 N （其值為 2 的冪），僅使用 N – 1 + log(N) – 1 = N + log(N) – 2 次的「小於」叫用。tournament_two() 實用嗎？其表現優於 largest_two() 嗎？若我們只計數「小於」的叫用次數，tournament_two() 應該比較快。對於 N = 65,536 大小的問題而言，largest_two() 需要 131,069 次的「小於」運算，而 tournament_two() 只需要 65,536 + 16 – 2 = 65,550 次，大約少了一半。但結果並非在此所見的這樣片面。

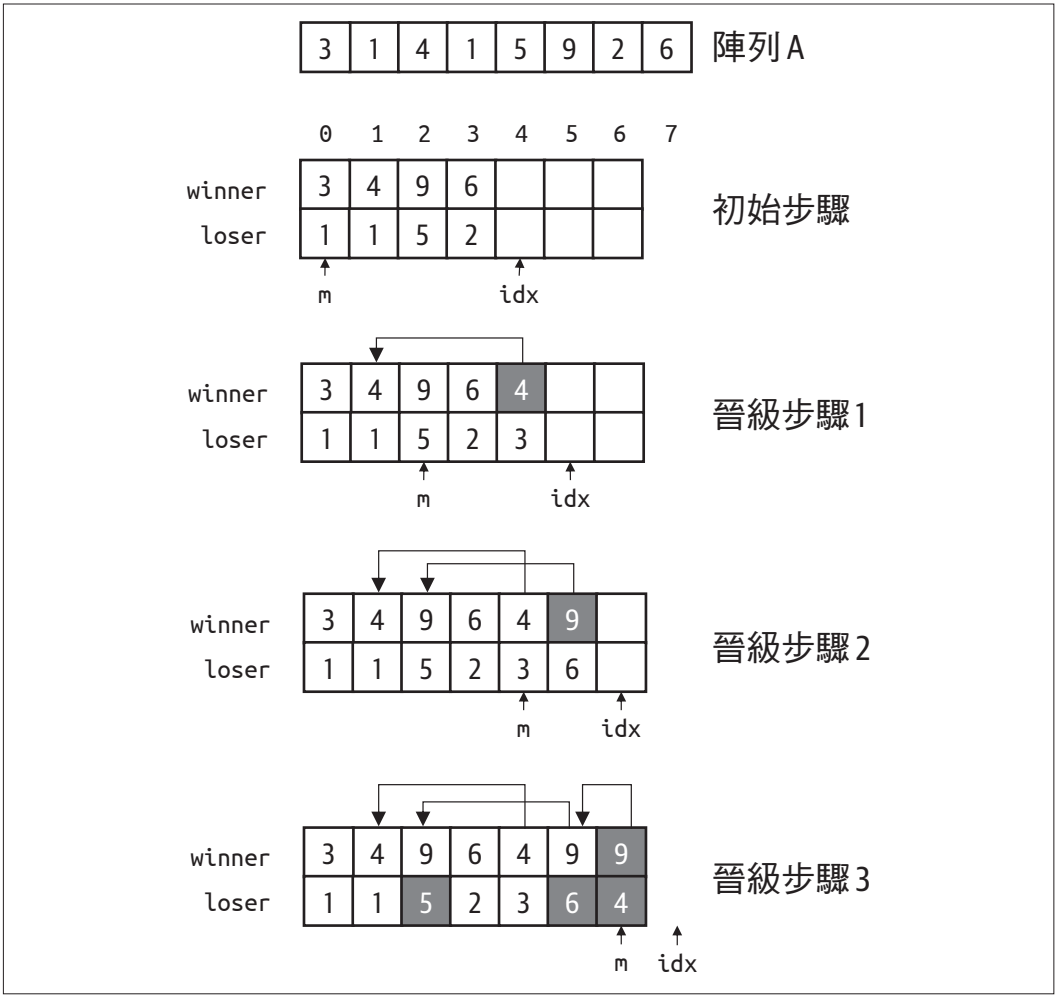


圖 1-8　錦標賽演算法（逐步執行）

表 1-5 顯示，tournament_two() 明顯比其他演算法慢得多！在此記錄 100 組隨機問題實例的解決總時間（問題實例的大小 N 值會從 1,024 成長到 2,097,152）。其中還包括示例 1-5 中各種演算法的效能結果。注意，若讀者於電腦執行此示例程式，則各個結果將有所不同，然而每行的整體趨勢將會雷同。

表 1-5　五種演算法的執行時間比較（單位：秒）

N	double_two	mutable_two	largest_two	sorting_two	tournament_two
1,024	0.00	0.01	0.01	0.01	0.03
2,048	0.01	0.01	0.01	0.02	0.05
4,096	0.01	0.02	0.03	0.03	0.10
8,192	0.03	0.05	0.05	0.08	0.21
16,384	0.06	0.09	0.11	0.18	0.43
32,768	0.12	0.20	0.22	0.40	0.90
65,536	0.30	0.39	0.44	0.89	1.79
131,072	0.55	0.81	0.91	1.94	3.59
262,144	1.42	1.76	1.93	4.36	7.51
524,288	6.79	6.29	5.82	11.44	18.49
1,048,576	16.82	16.69	14.43	29.45	42.55
2,097,152	35.96	38.10	31.71	66.14	…

表 1-5 看起來可能讓人不知所措，內容乍看像是一面數字牆。若讀者以另一台電腦（也許用的是較少記憶體或較慢 CPU）執行這些函式，則結果可能會不太一樣；然而，不論以何種電腦執行，都應該會呈現某些趨勢。大部分來說，我們鎖定任何一行往下檢視，執行時間約莫會隨著問題大小加倍而加倍。

此表中有些特別情況：注意，double_two() 起初為五種解法中最快的一個，而 largest_two() 後來居上（當 N > 262,144 之際）。巧妙設計的 tournament_two() 是目前為止最慢的方法（原因僅是它需要配置持續擴增的儲存陣列來處理資料）。因為太慢了（執行時間冗長），筆者甚至沒有針對表中最大問題實例執行這個演算法。

為了易於了解這些數值含意，圖 1-9 視覺化呈現對應執行時間趨勢（隨著問題實例大小遞增而論）。

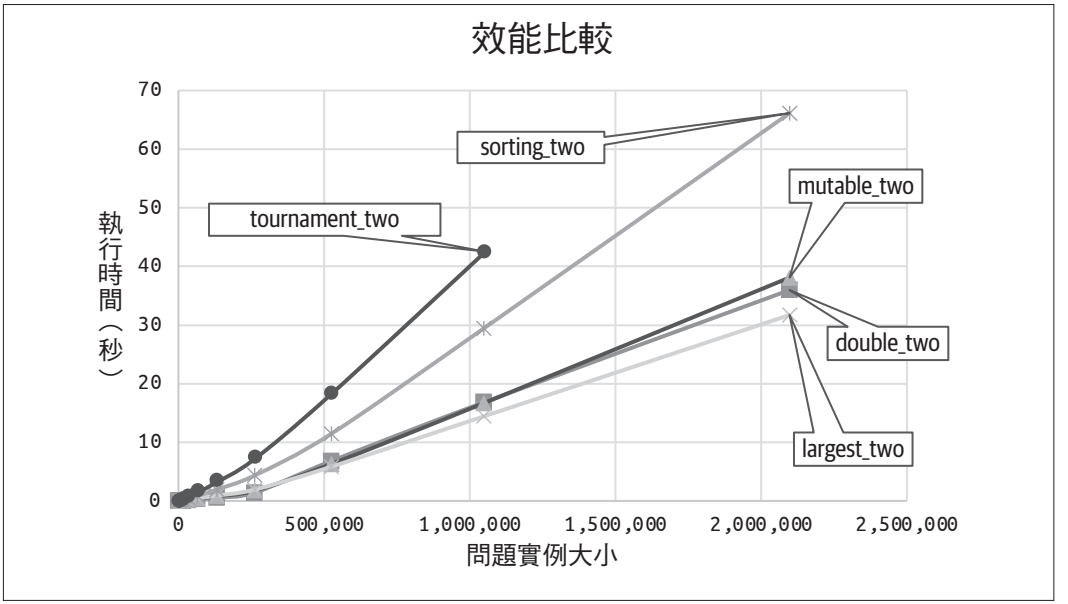


圖 1-9　執行時間效能比較

此圖呈現上述五種方法的執行時間效能細節資訊：

- 其中 `mutable_two()`、`double_two()`、`largest_two()` 三者效能不分軒輊（相較其餘兩種方法而言）。似乎像是源於同一「族群」，皆處於一條（似乎可預測的）直線軌跡上。
- `tournament_two()` 效率最差，其行為顯然與眾不同。基於為數不多的資料點，目前尚不確定是否會往效率更差的「向上彎曲」方向表現，或者也是循著一條直線發展。
- `sorting_two()` 的表現似乎優於 `tournament_two()`，不過比其他三種方法慢。`sorting_two()` 會進一步向上彎曲，還是最終轉直線發展呢？

若要了解這些趨勢線成形原因，我們需要學習兩個基本概念——詮釋演算法固有**複雜度**（*complexity*）的概念。

時間複雜度與空間複雜度

因為程式語言的差異，及某些語言（如：Java、Python）以直譯器運作的事實，可能難以計數基本作業（譬如：加法、變數指派、控制邏輯）數量。不過，若你**能夠**計數演

算法執行的基本作業次數，則可得知作業總數基於問題實例大小的變化。**時間複雜度**（*time complexity*）的目標是提出某個函數 C(N)，用於演算法執行的基本作業數計算（成為 N 的函數，N 是問題實例的大小）。

假設每個基本作業都需要固定時間量 t，基於執行作業的 CPU，我們可以將演算法執行時間建模為 T(N) = t × C(N)。示例 1-7 呈現的見解是：程式的**結構**是關鍵。對於函式 `f0`、`f1`、`f2`、`f3`，我們可以就輸入大小 N，確切計算函式執行 `ct = ct + 1` 作業的次數。表 1-6 為某幾個 N 值的計數結果。

示例 1-7　具有不同效能表現的四種函式

```
def f0(N):
  ct = 0
  ct = ct + 1
  ct = ct + 1
  return ct
```

```
def f1(N):
  ct = 0
  for i in range(N):
    ct = ct + 1
  return ct
```

```
def f2(N):
  ct = 0
  for i in range(N):
    ct = ct + 1
    ct = ct + 1
    ct = ct + 1
    ct = ct + 1
    ct = ct + 1
    ct = ct + 1
    ct = ct + 1
  return ct
```

```
def f3(N):
  ct = 0
  for i in range(N):
    for j in range(N):
      ct = ct + 1
  return ct
```

`f0` 的計數結果固定（與 N 無關）。`f2` 的每樣計數皆為 `f1` 的七倍（其中隨著 N 加倍，兩函式的結果也是加倍）。相較之下，`f3` 的計數成長相當快速；正如之前所述，當 N 加倍時，`f3(N)` 的計數結果增為四倍。在此，`f1`、`f2` 彼此較為相似（與 `f3` 相較而言）。下一章在評估演算法效能的論述中，將解釋 `for` 迴圈、巢狀迴圈（nested loop）的重要性。

表 1-6　四種函式的計數作業

N	f0	f1	f2	f3
512	2	512	3,584	262,144
1,024	2	1,024	7,168	1,048,576
2,048	2	2,048	14,336	4,194,304

我們評估演算法，也須考量**空間複雜度**（*space complexity*），即演算法針對問題實例大小 N 所需的額外記憶空間（extra memory）。**記憶體**（*memory*）[譯註]是於檔案系統或電腦 RAM 中儲存資料的通用術語。`largest_two()` 的空間需求最小：使用兩個變數（`my_max`、`second`）以及疊代器（iterator）變數（`idx`）。無論問題實例大小為何，其額外的空間需求固定不變。如此表示空間複雜度**與問題實例大小無關**（為常數）；

譯註　「memory」此字於多處會被譯成「記憶空間」。

mutable_two() 的行為類似。相較之下，tournament_two() 配置三個陣列 ——winner、loser、prior——大小皆為 N – 1。隨著 N 增加，額外儲存總量的成長幅度將與問題實例大小成正比[3]。相較 largest_two()，錦標賽結構的建置讓 tournament_two() 變慢。double_two()、sorting_two() 皆複製輸入資料 A 的內容，如此表示它們的儲存用法與 tournament_two() 較為類似（而非如同 largest_two()）。本書將評估演算法的時間複雜度與空間複雜度兩者。

表 1-5 中，largest_two 行的計時結果，往後的資料列中約莫為雙倍成長；double_two、mutable_two 的行為類似，正如筆者的觀測結果。總時間似乎與問題實例大小成正比，往後的資料列皆為雙倍增加。這是重要的觀測結果，這些函式比 sorting_two() 更有效率（sorting_two() 似乎處在另一個效率較低的軌跡上）。tournament_two() 依然是效率最差的，執行時間差了一倍之多，時間成長如此快速，讓筆者不願針對大的問題實例執行該演算法。

身為電腦科學家，我們不能直接將 largest_two()、mutable_two() 兩者效能曲線視為一樣。筆者需要以正規理論、符號（形式化）描述這個想法。下一章將介紹演算法行為分析所需的適當數學工具。

本章總結

本章介紹各式各樣的演算法範疇。筆者針對問題實例（大小為 N），以演算法執行的關鍵作業次數計數，為演算法的效能建模。另外還可以根據實證，評估演算法實作的執行時間。基於兩種情況下，隨著問題實例大小 N 加倍，而可以找出演算法的成長等級。

本章介紹數個關鍵概念，其中包括：

- **時間複雜度**的估計是：針對問題實例（大小為 N）計數演算法執行關鍵作業的次數。
- **空間複雜性**的估計是：對於問題實例（大小為 N）統計演算法執行所需的儲存量（記憶空間）。

下一章將介紹**漸近分析**（*asymptotic analysis*）數學工具，對正確分析演算法所需的技術做完整詮釋。

3 即：除對問題實例編碼資料之外的儲存內容；問題實例本身並不屬於演算法空間複雜度的一部分。

挑戰題

1. **回文字檢測工具**：回文字（palindrome word）從前面或從後面閱讀字母皆同，譬如：*madam*。設計相關演算法，檢查某個單字（有 N 個字元）是否為回文。而憑實證確認該演算法可優於示例 1-8 所述的另外兩種方法：

   示例 *1-8*　回文字檢測的兩種實作

   ```
   def is_palindrome1(w):
     """ 以負數 step（步長）建立 slice（切片），確認其與 w 的相等情況。"""
     return w[::-1] == w

   def is_palindrome2(w):
     """ 若頭尾字元相同則去掉頭尾字元，若不同則傳回 false。"""
     while len(w) > 1:
       if w[0] != w[-1]:     # 若不同，則傳回 False
         return False
       w = w[1:-1]           # 將頭尾字元去除；反覆運作

     return True             # 最終結果為回文
   ```

   解決上述問題之後，請讀者修改演算法，支援具有空格、標點符號、大小寫混合內容的回文字串檢測。例如，下列字串應屬於回文：「`A man, a plan, a canal. Panama!`」。

2. **線性時間的中位數演算法**：有個不錯的演算法，能夠有效率的找出任何串列的中位數所在之處（為簡單說明，假設串列大小為奇數）。檢視示例 1-9 的程式碼，使用本章所述的 `RecordedItem` 值計數「小於」的叫用。該實作在處理資料時會對輸入的串列內容重新排列。

   示例 *1-9*　線性時間演算法（用於算無序串列的中位數）

   ```
   def partition(A, lo, hi, idx):
     """ 以 A[idx] 作為劃分使用之值。"""
     if lo == hi: return lo

     A[idx],A[lo] = A[lo],A[idx]    # 互換位置
     i = lo
     j = hi + 1
     while True:
       while True:
         i += 1
         if i == hi: break
         if A[lo] < A[i]: break
   ```

```
    while True:
      j -= 1
      if j == lo: break
      if A[j] < A[lo]: break

    if i >= j: break
    A[i],A[j] = A[j],A[i]

  A[lo],A[j] = A[j],A[lo]
  return j

def linear_median(A):
  """
  傳回串列中位數之有效率實作，
  假設 A 有奇數個內容值。
  注意，該演算法將重新排列 A 的內容。
  """
  lo = 0
  hi = len(A) - 1
  mid = hi // 2
  while lo < hi:
    idx = random.randint(lo, hi)     # 隨機選擇有效索引
    j = partition(A, lo, hi, idx)

    if j == mid:
      return A[j]
    if j < mid:
      lo = j+1
    else:
      hi = j-1
  return A[lo]
```

實作不同的方法（有額外的儲存需求），依輸入內容建立排序串列、選出中間值。產生執行時間表，將其與 `linear_median()` 的執行時間相比。

3. **計數排序**：若已知某個串列 A，其內僅有非負整數（範圍從 0 到 M），則以下演算法僅使用大小為 M 的額外儲存空間即可正確排序 A 的內容。

   示例 1-10 有巢狀迴圈 ——`while` 迴圈中還有 `for` 迴圈。然而，讀者可以驗證 `A[pos+idx] = v` 只會執行 N 次。

   示例 *1-10*　線性時間的計數排序演算法

```
def counting_sort(A, M):
  counts = [0] * M
```

```
for v in A:
  counts[v] += 1

pos = 0
v = 0
while pos < len(A):
  for idx in range(counts[v]):
    A[pos+idx] = v
  pos += counts[v]
  v += 1
```

執行效能分析，驗證 N 個整數（範圍從 0 到 M）的排序時間，會隨著 N 的大小加倍而加倍。

你可以去掉內層 `for` 迴圈，改進此作業效能（其中使用 Python 的功能取代子串列內容，即 `sublist[left:right] = [2,3,4]`）。變更內容，依實證確認其結果變化也是隨著 N 加倍而加倍，以及執行速度提升 30%。

4. 修改錦標賽演算法得以支援奇數個內容值。

5. 示例 1-11 的程式能正確找到 A 中前兩大值的所在之處嗎？

示例 1-11　另類嘗試找出無序串列中的前兩大值

```
def two_largest_attempt(A):
  m1 = max(A[:len(A)//2])
  m2 = max(A[len(A)//2:])
  if m1 < m2:
    return (m2, m1)
  return (m1, m2)
```

說明該程式正常運作與失敗之際的情況。