

演算法基礎與實作

– 使用 Python 語言

江振瑞

April 1, 2019

Contents

| | |
|--------------------------------------|----|
| 1 認識演算法 | |
| — 從食譜到高階程式語言 | 4 |
| 1.1 演算法名稱的由來 | 4 |
| 1.2 什麼是演算法 | 6 |
| 1.3 演算法的例子 | 7 |
| 1.4 演算法的表示 | 9 |
| 1.5 演算法的實作 | 13 |
| 1.5.1 以 C 語言實作歐幾里得演算法 | 13 |
| 1.5.2 以 C++ 語言實作歐幾里得演算法 | 14 |
| 1.5.3 以 Java 語言實作歐幾里得演算法 | 14 |
| 1.5.4 以 Python 語言實作歐幾里得演算法 | 16 |
| 1.6 結語 | 17 |
| 2 演算法的正確性與效能 | |
| — 要馬好也要馬少吃草 | 18 |
| 2.1 演算法的正確性 | 18 |
| 2.2 正確性證明方法 | 19 |
| 2.2.1 演繹法 | 19 |
| 2.2.2 歸納法 | 19 |
| 2.2.3 矛盾法 | 20 |
| 2.3 演算法的效能 | 21 |
| 2.4 演算法的時間複雜度 | 21 |
| 2.5 趨近記號 | 23 |
| 2.6 降低演算法複雜度量級 | 25 |
| 2.7 結語 | 28 |
| 3 演算法的複雜度分析實例 | |
| — 決勝在數大時 | 29 |
| 3.1 演算法複雜度分析 | 29 |
| 3.2 費伯納西數列演算法 | 30 |
| 3.3 遞迴費伯納西數列演算法 | 32 |
| 3.4 氣泡排序演算法 | 34 |
| 3.5 插入排序演算法 | 36 |
| 3.6 堆積排序演算法 | 37 |
| 3.7 結語 | 40 |

| | |
|----------------------------|----|
| 4 分治演算法 | |
| — 凡治眾如治寡，分數是也 | 42 |
| 4.1 分治演算法基本概念 | 42 |
| 4.2 缺陷棋盤填滿演算法 | 43 |
| 4.3 合併排序演算法 | 46 |
| 4.4 快速排序演算法 | 48 |
| 4.5 最大連續子序列和演算法 | 52 |
| 4.6 快速傅立葉變換演算法 | 55 |
| 4.7 結語 | 59 |
| 5 分治計算幾何演算法 | |
| — 各個擊破獲得最後勝利 | 61 |
| 5.1 計算幾何介紹 | 61 |
| 5.2 計算幾何基本演算法 | 61 |
| 5.3 二維極大點問題 | 65 |
| 5.4 二維求秩演算法 | 67 |
| 5.5 二維最近點對問題 | 69 |
| 5.6 二維範諾圖演算法 | 72 |
| 5.7 二維凸包演算法 | 72 |
| 5.8 結語 | 72 |
| 6 刪尋演算法 | |
| — 化整為零蠶食而盡 | 73 |
| 6.1 刪尋演算法基本概念 | 73 |
| 6.2 二元搜尋演算法 | 74 |
| 6.3 選取演算法與中位數演算法 | 75 |
| 6.4 受限最小圓演算法 | 77 |
| 6.5 結語 | 79 |
| Appendix A Jeep7 | 81 |
| Appendix B ACM ICPC | 84 |
| B.1 簡介 | 84 |
| B.2 歷史 | 84 |
| B.3 競賽規則 | 85 |
| B.4 參賽獲勝秘訣 | 86 |
| B.4.1 獲勝秘訣一 | 86 |
| B.4.2 獲勝秘訣二 | 87 |
| B.4.3 獲勝秘訣三 | 87 |
| B.4.4 獲勝秘訣四 | 88 |
| B.5 ICPC 題目範例 | 88 |
| B.5.1 題目內容 | 88 |
| B.5.2 題目中文翻譯 | 89 |
| B.5.3 解題建模 | 90 |
| B.5.4 解題過程 | 91 |

Chapter 1

認識演算法 —— 從食譜到高階程式語言

Contents

| | |
|----------------------------|----|
| 1.1 演算法名稱的由來 | 4 |
| 1.2 什麼是演算法 | 6 |
| 1.3 演算法的例子 | 7 |
| 1.4 演算法的表示 | 9 |
| 1.5 演算法的實作 | 13 |
| 1.5.1 以 C 語言實作歐幾里得演算法 | 13 |
| 1.5.2 以 C++ 語言實作歐幾里得演算法 | 14 |
| 1.5.3 以 Java 語言實作歐幾里得演算法 | 14 |
| 1.5.4 以 Python 語言實作歐幾里得演算法 | 16 |
| 1.6 結語 | 17 |

1.1 演算法名稱的由來

演算法 (algorithm) 的名稱源自於 “al-Khwarizmi” , 這是一個出生於波斯名城巴格達 (Baghdad) 的阿拉伯數學家穆罕默德 · 賓 · 穆薩 · 阿爾-可瓦里茲米 (Muhammad ibn Musa al-Khwarizmi) 名字的最後一部份。阿爾-可瓦里茲米 大約生於西元 780 年，卒於西元 850 年，圖1.1為蘇聯在 1983 年為紀念他的 1200 歲生辰所發行的紀念郵票。阿爾-可瓦里茲米 將印度所發明的十進位數字記號傳入阿拉伯地區，而阿拉伯商人在經商時則將十進位數字記號傳入歐洲成為現今我們使用的數字記號。更重要的是，阿爾-可瓦里茲米 著有一本討論有系統地解決一次方程式 (linear equation) 及一元二次方程式 (quadratic equation) 的書籍，此書被翻譯成名為 “Liber algebrae et almucabala” 的拉丁文書籍，啟發了代數學的萌芽，對人類的現代科技與文明發展有相當深遠的影響。代數學的英文名稱 Algebra 就是源自於此書書名。阿爾-可瓦里茲米 以一步一步 (step by step) 的方式，描述算術 (arithmetic) 運算與一元一次方程式與一元二次方程式的解答步驟。稍後我們會知道，這些一步一步解答問題的步驟就是我們現今所定義的演算法，而這也就是為什麼演算法 (algorithm) 的名稱是源自於阿爾-可瓦里茲米的原因。



圖 1.1: 蘇聯在 1983 年為紀念阿爾-可瓦里茲米1200 歲生辰所發行的紀念郵票 (資料來源: https://en.wikipedia.org/wiki/Muhammad_ibn_Musa_al-Khwarizmi)

以下為一元二次方程式解答步驟的描述，取材自維基百科 (<https://zh.wikipedia.org/wiki/一元二次方程>):

一元二次方程式的解答步驟：

在方程的兩邊同時乘以未知數二次項係數的四倍；在方程式的兩邊同時加上未知數一次項係數的平方；然後在方程的兩邊同時開平方根 (即二次方根)。

例如：關於解開未知數 x 的方程式 $ax^2 + bx = -c$ 的解答步驟如下：

步驟一：在方程式的兩邊同時乘以 $4a$ ，即未知數二次項係數 a 的四倍。我們可得

$$4a^2x^2 + 4abx = -4ac$$

步驟二：在方程式的兩邊同時加上 b^2 ，即未知數一次項係數 b 的平方，我們可得

$$4a^2x^2 + 4abx + b^2 = -4ac + b^2$$

步驟三：然後在方程式的兩邊同時開平方根，得

$$2ax + b = \pm\sqrt{-4ac + b^2}$$

步驟四：經過移項處理及整理之後，我們可得未知數 x 的解答為

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

1.2 什麼是演算法

根據 Merriam-Webster Dictionary(<http://www.merriam-webster.com/dictionary/algorithm/>)，演算法可以廣義定義為「解決某一問題的一步一步程序」(a step-by-step procedure for solving a problem)，或狹義定義為「一個由一些步驟所構成的集合，依循這些步驟得以解決數學問題或完成計算機行程」(a set of steps that are followed in order to solve a mathematical problem or to complete a computer process)。

根據演算法的廣義定義，我們可以說一本食譜書籍中包含許多演算法，因為每一個單一食譜 (recipe) 都是一步一步解決烹調出某一種美食 (例如，冬蟲夏草雞湯) 的程序。企業組織的標準作業程序 (Standard Operating Procedure, SOP) 也是演算法，工作人員面對特定問題時，只要按照步驟指示一步一步進行就能解決問題。設備的使用手冊或故障排除手冊也包含許多演算法，因為它包含許多可以用於解決某一問題 (例如，如何安裝新設備及解決印表機的卡紙問題等) 的一步一步程序。流程圖也是演算法，例如，圖1.2中的流程圖 (flow chart) 可以一步一步解決燈泡不亮的問題。事實上，許多 SOP、使用手冊與故障排除手冊常常借重流程圖用以表達解決問題的一步一步程序，這些都可以視為廣義的演算法的例子。

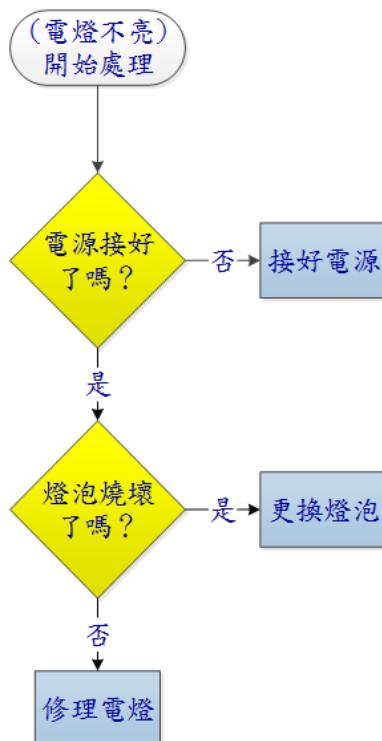


圖 1.2: 解決燈泡不亮問題的流程圖

在廣義與狹義的演算法的定義之外，我們從計算的角度給定演算法的定義，以方便將演算法的概念應用於計算機中。從這裡，我們開始闡述一個有關於演算法的一門學科 (a discipline of algorithms)— 演算法學 (Algorithmics)。我們希望帶領各位入門，探索研究演算法的相關知識，了解演算法的奧妙。

從計算角度給定的演法定義如下：

演算法 (algorithm) :

由有限 (finite) 步驟 (step) 所構成的集合，依照給定的輸入 (input)，依序執行每個明確 (definite) 且有效 (effective) 的步驟 (step)，以便能夠解決特定的問題；而步驟的執行必定會終止 (terminate)，並產生輸出 (output)。

根據上述演算法的計算角度定義，演算法是由解決某一特定問題的一些步驟所組成，具有以下特性：

1. 指定輸入 (input)：演算法必須指定輸入，可以由外界輸入 0 個、1 個或多個資料。
2. 具有輸出 (output)：演算法必定有至少 1 個以上的輸出。
3. 有限性 (finiteness)：演算法步驟的個數必須是有限的，而且步驟的執行最後會終止 (terminate)。
4. 明確性 (definiteness)：演算法的每個步驟都必須是明確 (definite) 而不含糊的 (unambiguous)。
5. 有效性 (effectiveness)：演算法的每個步驟必須是有效的 (effective) 或說是可行的 (feasible)。

演算法必須指定輸入，我們有時會透過介面（例如鍵盤）由外界獲得問題輸入，有時也會將輸入直接寫在演算法中。因此演算法可以由外界輸入 0 個、1 個或多個資料。當然，演算法必須要有一個以上的輸出，如此演算法才能為人們所用。演算法必須滿足有限性 (finiteness)，它的步驟個數必須是有限的，而且步驟的執行最後會終止 (terminate)；如此，演算法才有可能在執行有限步驟之後終止並產生輸出為人們所用。

演算法必須滿足明確性 (definiteness)，也就是說它的每一個步驟必須是明確而不含糊的。例如，若有一個步驟是「將變數 x 加 6 或 7」，則這個步驟是不明確的，因為我們可能加 6 也可能加 7 到變數 x 中；但是，「將變數 x 加入以下的值：亂數生成器 (random number generator) 函數的值乘以 2 取整數後再加 6」則是明確的步驟，因為我們可以很明確地將亂數產生器函數的值加到變數 x ；又例如，「計算 $5/0$ 」是不明確的，因為分母 (除數) 為 0 是沒有明確定義的計算。

最後，演算法必須滿足有效性 (effectiveness)，也就是說演算法的每個步驟必須是有效的 (effective) 或說是可行的 (feasible)；基本上，每個步驟即使由人們拿著紙筆，都可以在有限時間內計算出結果。舉個例子說，步驟「計算出 $\sqrt{2}$ 完全無誤差的值」不滿足有效性，因為它是不可行的，我們需要進行無窮位數的計算才可以得到 $\sqrt{2}$ 完全無誤差的值。相反的，「計算 $\sqrt{2}$ 到小數點以下 10 位，並捨棄其後位數」則滿足有效性，因為它是可行的，人們即使只是藉由紙筆，都可以計算出 $\sqrt{2} = 1.4142135623$ 的結果。

1.3 演算法的例子

圖1.3中的流程圖是一個演算法的例子（我們稍後會說明流程圖中記號的涵義）。這個流程圖包含有限步驟，可以輸入兩個正整數 m 與 n ，並一步一步解決 m 是不是 n 的因數 (factor) 的問題，最後結束執行並輸出結果： m 是 (或不是) n 的因數。這個流程圖不但符合廣義與狹義演算法的定義，也符合我們從計算觀點給定的演算法定義。

另外，我們也可以說大部份的計算機程式 (computer program) 是演算法，因為它可以在計算機上一步一步執行明確且有效的步驟以解決特定的問題。例如，程式列表1.1中的 Python 程式可以輸入兩個正整數 m 與 n ，並一步一步解決 m 是不是 n 的因數的問題，它符合廣義與狹義演算法的定義，也符合我們從計算觀點給定的演算法嚴謹定義。但是，並不是所有的計算機程式都符合我們從計算觀點給定的演算法定義。例如，程式列表1.2及1.3中的 Python 程式就不是演算法。程式列表1.2中的 Python 程式根本沒有輸出，因此它不是演算法。雖然程式列表1.3的 Python 程式有輸出敘述，但是此輸出敘述永遠不會被執行，因為在此敘述之前有一個無窮迴圈，永遠都不會終止，所以這個程式沒有輸出也不會終止，它並不符合我們從計算觀點給定的演算法定義，因此它當然不是演算法。

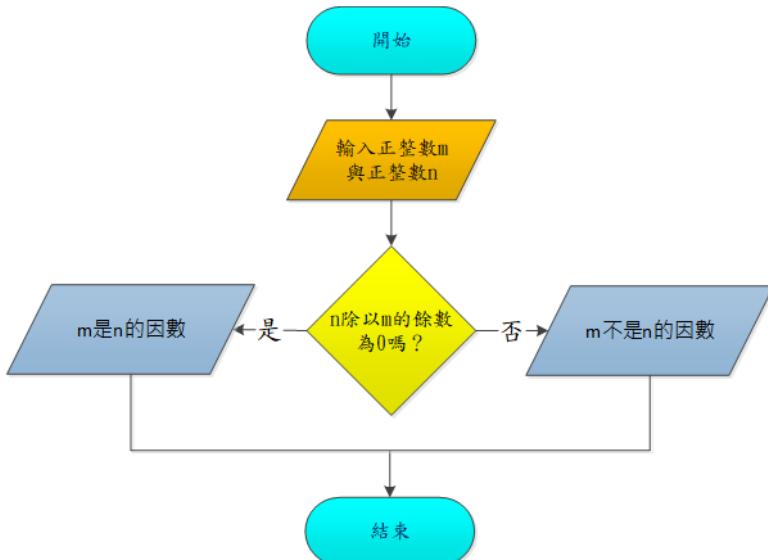


圖 1.3: 解決正整數 m 是不是正整數 n 的因數問題的流程圖

程式列表 1.1: factor.py

```

#File: factor.py
#Note: To input integers m and n and output n is (or is not) a factor of m
m,n=map(int, input('Please enter two integers m (smaller) and n (larger): ').split());
if m%n==0:
    print(f'{m} is a factor of {n}.')
else:
    print(f'{m} is not a factor of {n}.')
  
```

程式列表 1.2: non-algorithm.py

```

#File: non-algorithm1.py
#Note: The program never outputs anything
m, n=300, 800
for i in range(m, n):
    s+=i
  
```

程式列表 1.3: non-algorithm2.py

```

#File: non-algorithm2.py
#Note: The program neither outputs anything nor stops
m, n=300, 800
while m!=n:
    r=n%m
    print(f'{n} mod {m} equals to {r}.')
  
```

1.4 演算法的表示

我們在此單元中說明演算法的表示方法，這裡所謂的演算法，指的是從計算觀點定義的演算法。實際上，在本單元以後，當我們提及演算法時，指的都是從計算觀點所給定的演算法定義。

一般我們使用自然語言 (中文或英文等語言)、流程圖 (flowchart)、虛擬碼 (pseudo code) 或高階程式語言 (high level programming language) 來表示演算法。以下我們舉一個古老的演算法 – 歐幾里得演算法 (Euclid Algorithm) 為例子，說明如何以前三種方式表示演算法；以高階程式語言表示演算法則在下一單元中再介紹。

歐幾里得演算法就是我們熟知的輾轉相除法 (division algorithm)，如圖1.4所示。大約在西元前 300 年由希臘數學家歐幾里得(Euclid, 西元前 325-西元前 265) 在著名的「幾何原本」著作中提出，用於求出二個正整數的最大公因數 (GCD, Greatest Common Divisor)。

以下我們使用中文與英文符號表示歐幾里得演算法。

歐幾里得演算法:

問題: 給定二個正整數 m 及 n ，找出此二數的最大公因數 (GCD)

步驟 1.[找出餘數]: 求出 m 除以 n 的餘數，並記錄於 r 。

步驟 2.[餘數為 0 嗎 ?]: 如果 $r=0$ 則停止，輸出 n 為 GCD。

步驟 3.[更新被除數與除數]: 設定 $m=n$ 及 $n=r$ ，並跳至步驟 1。

| | | | |
|---|-------|-------|---|
| 1 | 75569 | 52317 | 2 |
| | 52317 | 46501 | |
| 4 | 23252 | 5813 | |
| | 23252 | | |
| | 0 | | |

圖 1.4: 輾轉相除法示意圖

我們也可以採用明確定義的符號來繪製流程圖，以清楚表達演算法。圖1.5中所列的符號是美國國家標準學會 (American National Standards Institute, ANSI) 於 1970 年公佈的流程圖符號，是當今通用的流程圖符號 [1] [2]。圖1.6中所顯示的，即是以 ANSI 流程圖符號表示歐幾里得演算法的例子。

| 符號名稱 | 符號 | 意義 |
|-----------------------------------|----|---|
| 終端 (terminal) | | 表示流程圖的開始與結束 |
| 流程線 (flowline) | | 表示流程的進行方向 |
| 輸入/輸出 (input/output) | | 表示資料的輸入或結果的輸出 |
| 處理 (processing) | | 表示進行某些處理或工作 |
| 決策 (decision) | | 表示針對條件進行滿足(是)或不滿足(否)的判斷 |
| 連接 (connector) | | 表示連接流程到另一個流程圖 |
| 已定義好的處理程序 (predefined process) | | 表示已事先定義好的，但是未畫在流程圖中的處理程序(process)或副程式(subroutine) |

圖 1.5: ANSI 通用流程圖符號 [1] [2]

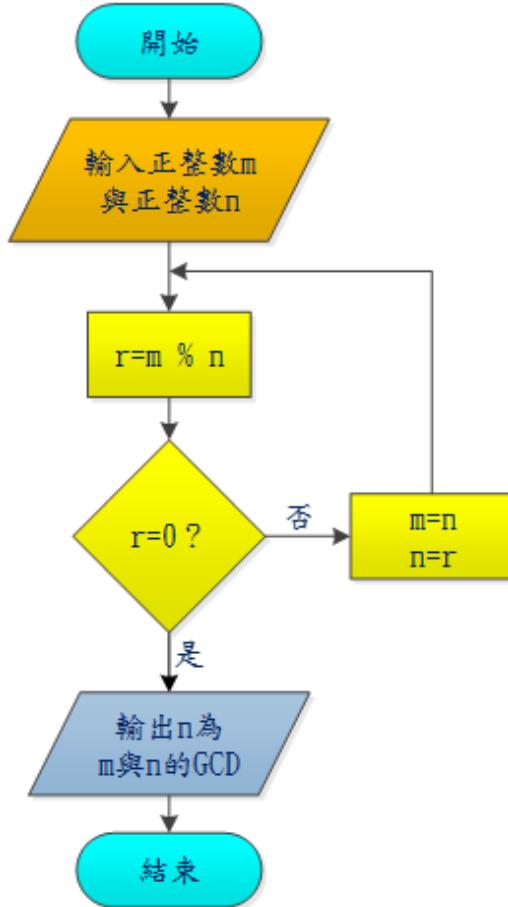


圖 1.6: 以流程圖表示歐幾里得演算法

在本書中，我們雖然有時候直接使用自然語言描述演算法，但是我們主要還是透過虛擬碼來表示演算法。虛擬碼以一種混雜著自然語言與高階程式語言結構的方式來描述演算法，可以達到簡潔易讀、容易分析，而且也容易轉換為高階程式語言的目的。以下我們介紹本書所採用的虛擬碼撰寫規則，因為虛擬碼仍然具有自然語言性質，因此這些撰寫規則有時可以稍稍調整，以方便閱讀者明確理解為原則。虛擬碼撰寫規則如下：

- 演算法名稱及參數：以 **Algorithm** 演算法名稱 (參數 1, 參數 2,...) 來列出演算法名稱並指明其輸入參數。因為在演算法中仍然會描述輸入參數，因此有時可省略而不列出輸入參數。
- 輸入：以 **Input** 輸入描述 來進行輸入說明
- 輸出：以 **Output** 輸出描述 來進行輸出說明
- 設定：以 **\leftarrow** 表示，可以將一個算式 (expression) 的值指定給某一個變數 (置入某變數中)。
- 算術運算：以 **$+ - * / \%$** 表示加、減、乘、除、模 (除法求餘數) 的運算。
- 比較與邏輯運算：以 **$= > < \geq \leq \neq$** 表示等於、大於、小於、大於等於、小於等於及不等於的運算，並使用 **$\wedge \vee \sim$** 表示邏輯的且、或與反向的運算。

- 決策結構：以 **if** 條件 **then** 條件為真的動作 **else** 條件為偽的動作 **end if** 來表示。當條件成立時，演算法執行所有包含在「條件為真的動作」的所有指令（步驟）；反之則執行所有包含在「條件為偽的動作」的所有指令。請注意，有時為了縮減演算法排版空間，有時會省略 **end if**，僅使用不同程度的縮排來表示步驟的區塊關係，而本書也將採用只有縮排的方式來表示步驟的區塊關係。
- while 迴圈：以 **while** 條件 **do** 迴圈動作 **end while** 來表示。當條件成立時，演算法會重複執行所有包含在「迴圈動作」的所有指令；反之則離開迴圈，進入下一個指令。
- for 迴圈：以 **for** 迴圈變數變動之範圍及其變動方式 **do** 迴圈動作 **end for** 來表示。當「迴圈變數」的值在指定的範圍中時，演算法會重複執行所有包含在「迴圈動作」的所有指令；反之則離開迴圈，進入下一個指令。
- 陣列元素索引：以 **陣列名稱 [i]** 代表命名為陣列名稱的陣列索引 (index) 為 i 的元素，一個有 n 個元素的陣列，其元素索引值為 0,1,...,n-1。
- 演算法呼叫：以 **演算法名稱 (參數...)** 來表示演算法的呼叫。
- 演算法返回：以 **return 返回值** 來代表演算法結束執行並輸出返回值。

請注意，有時為了縮減演算法排版空間，有些虛擬碼會省略 **end if**、**end for** 或 **end while** 等表示步驟區塊結束的記號，而僅使用不同程度的縮排來表示步驟的區塊關係。為節省排版空間，本書也採用只有縮排的方式來表示步驟的區塊關係。另外，為了方便理解，本書也在虛擬碼中加入註解（以 \triangleright 為註解開始記號）；為方便說明，也在每個步驟前加上編號。當然，讀者在使用虛擬碼描述演算法時，可以加也可以不加註解與編號。

以下就是使用虛擬碼來描述歐幾里得演算法的範例。其中第一個範例未加入步驟編號與註解；而第二個範例則加上步驟編號與完整的註解。

Algorithm EuclidGCD(m, n)

Input: 二個正整數 m 及 n
Output: m 及 n 的最大公因數 (GCD)

```

 $r \leftarrow m \% n$ 
while  $r \neq 0$  do
     $m \leftarrow n$ 
     $n \leftarrow r$ 
     $r \leftarrow m \% n$ 
return  $n$ 
```

Algorithm EuclidGCD(m, n) \triangleright 歐幾里得 (Euclid) 輾轉相除最大公因數 (GCD) 演算法

Input: 二個正整數 m 及 n \triangleright 演算法輸入描述

Output: m 及 n 的最大公因數 (GCD) \triangleright 演算法輸出描述

| | |
|--------------------------------------|---|
| 1: $r \leftarrow m \% n$ | \triangleright 計算 m 除以 n 的餘數並存入 r 中 |
| 2: while $r \neq 0$ do | \triangleright 當 r 不等於 0 時持續執行 while 迴圈區塊內之步驟 |
| 3: $m \leftarrow n$ | \triangleright 以縮排指出區塊結構；將 n 的值存入 m 中 |
| 4: $n \leftarrow r$ | \triangleright 以縮排指出區塊結構；將 r 的值存入 n 中 |
| 5: $r \leftarrow m \% n$ | \triangleright 以縮排指出區塊結構；計算 m 除以 n 的餘數並存入 r 中 |
| 6: return n | \triangleright 演算法結束並回傳 n 的值作為輸出 |

1.5 演算法的實作

如前所述，我們一般使用自然語言、流程圖、虛擬碼或高階程式語言來表示演算法，而在本書中我們通常使用虛擬碼來表示演算法。使用虛擬碼表示演算法，可以方便演算法的分析，增加演算法的易讀性，並且可以方便的轉為以高階程式語言表示演算法。當我們以高階程式語言表示演算法時，我們可以在電腦上直接執行以高階程式語言編寫而成的程式，並藉此得到執行結果。因此，當我們直接以高階程式語言描述演算法時，我們特別將之稱為「以高階程式語言實作 (implement) 演算法」，或是稱為「以高階程式語言進行演算法的實作 (implementation)」。在本單元中，我們將舉例說明使用 C、C++、Java 與 Python 語言實作歐幾里得演算法，或稱為歐幾里得 GCD(Euclid GCD) 演算法。

本書的範例程式都是使用 Jeep7 軟體編寫的。建議讀者也可以使用 Jeep7 軟體來編輯、編譯與執行 C、C++、Java 與 Python 高階程式語言程式。Jeep7 為 Java Editor for Every Programmer v7.0 的簡稱，是由筆者使用 Java 語言所編寫的整合開發環境 (Integrated Development Environment, IDE)。因為使用 Java 語言編寫，因此 Jeep7 可以在所有的平台上執行，而且 Jeep7 支援 C、C++、Java 與 Python 四種語言，並透過簡潔的中文介面，讓使用者輕易完成四種不同語言程式的編輯、編譯與執行等工作。請讀者參考附錄 A 以獲得 Jeep7 詳細的資訊。

因為演算法有指定輸入的特性，因此演算法僅處理特定的輸入。例如，剛剛提過的歐幾里得演算法指定輸入二個正整數 m 及 n。當然，當我們以高階程式語言實作演算法，讓使用者透過輸入介面由外界傳入演算法輸入時，可能會輸入錯誤的資料 (例如輸入負數)。本書聚焦於演算法解決問題的核心概念，因而假設所有的輸入都符合演算法的指定，所以在實作演算法時不處理使用者輸入錯誤資料的狀況。

在實務上，若我們將符合演算法指定的輸入資料以文字檔案形式儲存，並以作業系統之輸入轉向 (redirect) 方式將文字檔案資料直接輸入高階程式語言程式中，則所有的輸入都會符合演算法的指定。基本上，著名的計算機協會國際大學生程式設計競賽 (Association of Computing Machinery International Collegiate Programming Contest，簡稱 ACM ICPC) 就是採取上述的方法作為高階語言程式的輸入方式。ACM ICPC 是一個試煉各種演算法實作的好場合，國際間也有許多團體提供相關的 ICPC 訓練教學網站 (例如，UVa Online Judge) 與 ICPC 賽事裁判系統 (例如，PC² (Programming Contest Control) 系統)，有興趣的讀者請參考附錄 B 以取得詳細資訊。

1.5.1 以 C 語言實作歐幾里得演算法

下列的 C 語言程式 EuclidGCD.c 以 int EuclidGCD(int m, int n) 函式或函數 (function) 實作歐幾里得演算法。並在主要函式 main() 中加入輸入 (printf(...)) 與輸出 (scanf(...)) 敘述，並呼叫 EuclidGCD 方法，讓演算法可以由外部輸入訊息，並輸出計算結果。

程式列表 1.4: EuclidGCD.c

```
Note: To input two positive integers m and n for running the
      Euclid division algorithm to ouput the greatest common divisor of m and n */
#include <stdio.h>
int EuclidGCD(int m, int n){
    int r=m%n;
    while (r!=0) {
        m=n;
        n=r;
        r=m%n;
    }
    return n;
} /* End of EuclidGCD() */
main() {
```

```

int m,n;
printf("Please enter two positive integers m and n: ");
scanf("%d%d",&m,&n);
printf("The GCD of integers %d and %d is %d.",m,n,EuclidGCD(m,n));
} /* End of main() */

```

執行結果:



1.5.2 以 C++ 語言實作歐幾里得演算法

下列的 C++ 語言程式 EuclidGCD.cpp 以函式 int EuclidGCD(int m, int n) 實作歐幾里得演算法。並在主要函式 int main() 中加入輸入 (cin>>...) 與輸出 (cout<<...) 敘述，並呼叫 EuclidGCD 方法，讓演算法可以由外部輸入訊息，並輸出計算結果。

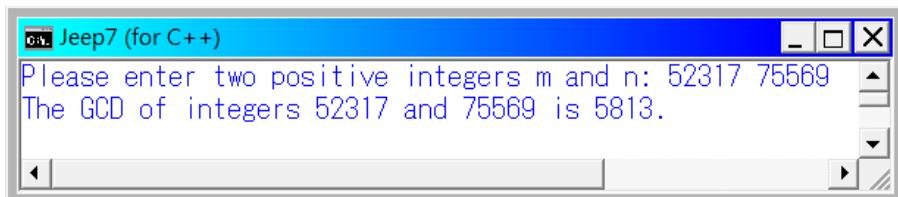
程式列表 1.5: EuclidGCD.cpp

```

//File: EuclidGCD.cpp
//Note: To input two positive integers m and n for running the
//      Euclid division algorithm to ouput the greatest common divisor of m and n
#include <iostream>
using namespace std;
int EuclidGCD(int m, int n){
    int r=m%n;
    while (r!=0) {
        m=n;
        n=r;
        r=m%n;
    }
    return n;
} // End of EuclidGCD()
int main() {
    int m,n;
    cout<<"Please enter two positive integers m and n: ";
    cin>>m>>n;
    cout<<"The GCD of integers "<<m<<" and "<<n<<" is "<<EuclidGCD(m,n)<<".<<endl;
} // End of main()

```

執行結果:



1.5.3 以 Java 語言實作歐幾里得演算法

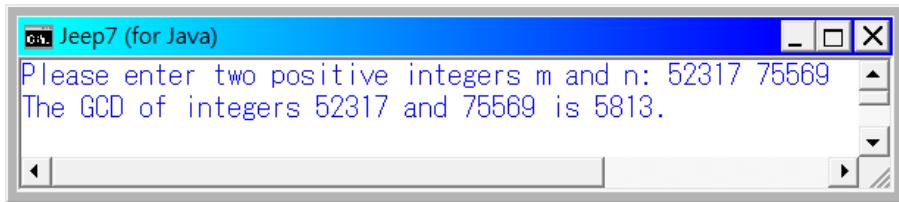
下列的 Java 語言程式 EuclidGCDClass1.java 以方法 (method) static int EuclidGCD(int m, int n) 實作了歐幾里得演算法。並在主類別 EuclidGCDClass1 中的主要方法 main 中加入

標準輸入 (Scanner Cin=new Scanner(System.in);...Cin.nextInt()) 與標準輸出 (System.out.print(...)) 敘述，並呼叫 EuclidGCD 方法，讓演算法可以由外部輸入訊息，並輸出計算結果。

程式列表 1.6: EuclidGCDClass1.java

```
//File: EuclidGCDClass1.java
//Note: To input two positive integers m and n for running the
//      Euclid division algorithm to ouput the greatest common divisor of m and n
import java.util.Scanner;
public class EuclidGCDClass1 {
    public static void main(String[] args) {
        int m,n;
        Scanner Cin=new Scanner(System.in);
        System.out.print("Please enter two positive integers m and n: ");
        m=Cin.nextInt();
        n=Cin.nextInt();
        System.out.println("The GCD of integers "+m+" and "+n+" is "+EuclidGCD(m,n)+".");
    } //End of method: main()
    static int EuclidGCD(int m, int n) {
        int r=m%n;
        while (r!=0) {
            m=n;
            n=r;
            r=m%n;
        }
        return n;
    } //End of method: EuclidGCD()
} //End of class: EuclidGCDClass1
```

執行結果：



下列的 Java 語言程式 EuclidGCDClass2.java 與 EuclidGCDClass1.java 類似，但是採用視窗輸入 (JOptionPane.showInputDialog(...)) 與視窗輸出 (JOptionPane.showMessageDialog(...)) 敘述，讓演算法可以由外部輸入訊息，並輸出計算結果。

程式列表 1.7: EuclidGCDClass1.java

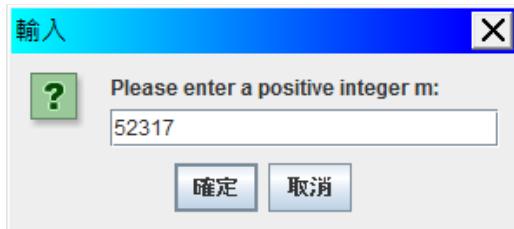
```
//File: EuclidGCDClass2.java
//Note: To input two positive integers m and n for running the
//      Euclid division algorithm to ouput the greatest common divisor of m and n
import javax.swing.JOptionPane;
public class EuclidGCDClass2 {
    public static void main(String[] args) {
        int m,n,gcd;
        String s1,s2,msg;
        s1=JOptionPane.showInputDialog("Please enter a positive integer m: ");
        s2=JOptionPane.showInputDialog("Please enter a positive integer n: ");
        m=Integer.parseInt(s1);
        n=Integer.parseInt(s2);
        gcd=EuclidGCD(m, n);
        msg="The GCD of integers "+m+" and "+n+" is "+gcd+".";
        JOptionPane.showMessageDialog(null, msg);
    } //End of method: main()
    static int EuclidGCD(int m, int n) {
        int r=m%n;
        while (r!=0) {
            m=n;
            n=r;
            r=m%n;
        }
        return n;
    } //End of method: EuclidGCD()
}
```

```

        n=r;
        r=m%n;
    }
    return n;
} //End of method: EuclidGCD()
} //End of class: EuclidGCDClass2

```

執行結果:



1.5.4 以 Python 語言實作歐幾里得演算法

下列的 Python 語言程式 EuclidGCD.py 以 EuclidGCD(m, n) 方法實作歐幾里得演算法，並以標準輸入(input(...)) 與標準輸出(print(...)) 敘述由外部輸入訊息及輸出計算結果。

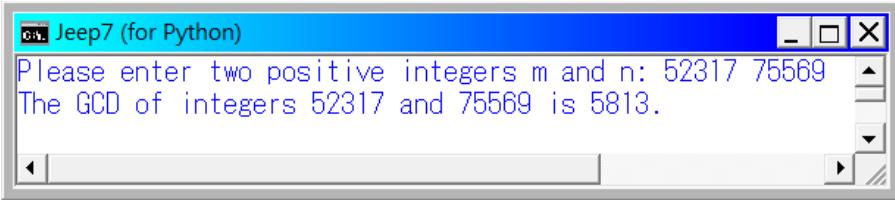
程式列表 1.8: EuclidGCD.py

```

#File: EuclidGCD.py
#Note: To input two positive integers m and n for running the
#      Euclid division algorithm to output the greatest common divisor of m and n
def EuclidGCD(m, n):
    r=m%n
    while (r!=0):
        m=n
        n=r
        r=m%n
    return n
m,n=map(int, input("Please enter two positive integers m and n: ").split())
print(f'The GCD of integers {m} and {n} is {EuclidGCD(m,n)}.')

```

執行結果:



1.6 結語

有本著名的書，書名為「演算法統治世界」(Automate This – How Algorithms Came to Rule the World)，內容闡述由華爾街的股票交易、亞馬遜二手書的拍賣網站、電影票房預測到賭博對弈，完全都由演算法直接掌控，最後並總結未來將屬於演算法及其創造者。演算是否能夠真實統治世界或許留有爭議，但是演算法確實對我們今日的生活有深遠的影響，這可以由另外一本著名的書：「改變世界的九大演算法」(Nine Algorithms That Changed the Future) 得到佐證。

在本章中，我們介紹演算法名稱的由來，描述演算法在廣義、狹義及計算觀點的定義，並藉此說明包括食譜、標準作業程序、流程圖、高階程式語言程式等都是演算法的例子。我們也敘述如何以自然語言、流程圖、特別是虛擬碼描述演算法，並舉例說明如何以高階程式語言，如 C、C++、Java 及 Python 來實作以計算角度定義的演算法，能夠給定某個特定問題的輸入，然後一步一步明確地執行數量有限的，可以有效執行的高階程式語言敘述，最後輸出解決問題的結果並結束。我們希望藉由此章，讓讀者能夠對演算法有最基本的認識，並進而對演算法產生興趣，能夠進一步探索演算法的奧妙，可以設計、分析、比較各種演算法。

Chapter 2

演算法的正確性與效能 — 要馬好也要馬少吃草

Contents

| | |
|--------------------------|----|
| 2.1 演算法的正確性 | 18 |
| 2.2 正確性證明方法 | 19 |
| 2.2.1 演繹法 | 19 |
| 2.2.2 歸納法 | 19 |
| 2.2.3 矛盾法 | 20 |
| 2.3 演算法的效能 | 21 |
| 2.4 演算法的時間複雜度 | 21 |
| 2.5 趨近記號 | 23 |
| 2.6 降低演算法複雜度量級 | 25 |
| 2.7 結語 | 28 |

2.1 演算法的正確性

一個演算法一定要能夠產生正確的結果，也就是滿足正確性 (correctness)，如此演算法才能夠為人們所用，才是好的 (對的) 演算法。我們可以透過一些證明的技巧，如演繹法 (deduction)(也就是直接證明法 (direct proof))、歸納法 (induction) 或矛盾法 (contradiction)(也就是反證法) 來證明演算法的正確性。其中演繹法由法國哲學家笛卡兒(Rene Descartes, 1596-1650) 倡導；歸納法由英國哲學家培根(Francis Bacon, 1561-1626) 提出；而反證法出現在古希臘哲學家亞里斯多德(Aristotle, 西元前 384-西元前 322) 所編纂的題旨篇和謬誤論証篇二書中。

演算法的正確性無疑是演算法最重要的特性，然而因為篇幅關係，本書僅簡單介紹上述三種證明技巧，並各舉一例說明。本書大部份時候不會特別說明演算法的正確性證明，可幸有許多文獻資料針對特定演算法的正確性進行討論，包括一些推論與證明命題 (proposition) 或定理 (theorem) 的研究。因此，我們要建議對演算法正確性證明有興趣的讀者，自行參考眾多文獻資料以獲得本書中大部份演算法正確性證明的相關知識。若有讀者覺得證明方法稍偏艱澀，建議這些讀者可以直接略過本節。

2.2 正確性證明方法

我們在本節中詳細描述演繹法、歸納法及矛盾法的證明技巧，並分別舉以下三種證明實例說明它們的使用方式：

- 歐幾里德輾轉相除最大公因數演算法正確性證明
- 因數 (divisor or factor) 相關定理證明：唯一分解定理
- 質數 (prime or prime number) 與合數 (composite number) 相關定理證明：歐幾里得定理

我們先介紹一些與上述證明實例相關的定義：

定義 2.1. 若 m 和 n 為不全為零的整數，則 m 和 n 的最大公因數 (greatest common divisor)，標示為 $\gcd(m, n)$ ，是同時整除它們的最大整數。若 $\gcd(m, n) = 1$ ，則我們說 m 和 n 互質或互素 (relatively prime, mutually prime, co-prime or coprime)。

定義 2.2. 質數 (prime or prime number)，又稱素數，指的是除了 1 和該數本身之外，無法被其他正整數整除的大於 1 的整數。

根據上述定義，1 不是質數，而 2 是最小的質數。

定義 2.3. 合數 (composite number)：大於 1 的整數若不是質數，則稱為合數。

根據上述定義，除了 2 以外，所有的偶數都是合數；而 0 與 1 既不是質數也不是合數。

2.2.1 演繹法

演繹法 (deduction) 又稱為演繹推理 (deductive reasoning) 或直接證明 (direct proof) 法，由法國哲學家笛卡兒 倡導，引申自古希臘哲學家亞里斯多德 三段論邏輯推理：「結論」(conclusion) 可從兩個 (或更多) 「前提」(proposition) 必然地推理得出。如果前提為真，則結論必然為真。

演繹法證明思維是利用邏輯推理，由已知的命題推導出下一個命題，如此一一推導出欲證明的命題為真。我們利用演繹法證明以下命題 (定理) 為真。

定理 2.1. 歐幾里德輾轉相除最大公因數演算法是正確的

證明：只要證明 $\gcd(m, n) = \gcd(n, r)$ 就可得證，其中 $r = m \% n$ 。令 $\gcd(m, n) = g$ ，則我們可得 $m = ig, n = jg \circ r = m - kn = ig - jkg = (i - jk)g$ ，這表示 g 也是 r 的因數。設 j 與 $i - jk$ 的最大公因數為 d ，則可設 $j = xd, i - jk = yd$ 。我們可得 $i = jk - yd = xdk - yd = (xk - y)d$ 。因而推得 $m = (xk - y)dg, n = xdg$ ，並可推得 $\gcd(m, n) = dg$ 。因為 $\gcd(m, n) = g$ ，我們可推得 $d = 1$ (也就是 $i - jk$ 與 j 互質)。所以 $\gcd(n, r) = \gcd(jg, i - jkg) = g$ ，故得證。□

2.2.2 歸納法

歸納法 (induction) 由英國哲學家培根 提出，通常用於證明命題針對大於某個基礎值的自然數 n 均成立的情況。其證明思維分為以下階段：

- 歸納基底 (induction base):
先證明當 n 等於基礎值 n_0 時命題成立。
- 歸納假設 (induction hypothesis):
假設當 $n = k \leq n_0$ 時命題成立。

- 歸納步驟 (induction step):
基於歸納假設推導 (deduce) $n = k + 1$ 時命題也成立。
- 最後根據歸納原則 (induction principle) 即可證明當 n 大於或等於基礎值 n_0 時命題都成立。

我們利用歸納法證明以下定理為真。

定理 2.2. 唯一分解定理 (unique factorization theorem): 大於 1 的整數 n 必可分解為唯一的一組質數乘積，亦即 $n = p_1^{k_1} p_2^{k_2} \dots p_j^{k_j}$ ，其中 $p_1 < p_2 < \dots < p_j$ 皆為質數，且這個表示式是唯一的。

證明: 我們利用歸納法證明:

- 歸納基底:
由 $2 = 2^1$ ，我們得知 $n = 2$ 時定理命題成立。
- 歸納假設:
假設所有符合 $2 \leq m < n$ 的整數 m 可表示成唯一一組質數乘積。
- 歸納步驟:
如果 n 是質數，則 $n = n^1$ 為唯一一組質數乘積。否則， n 為合數，代表存在整數 $m, h > 1$ 使得 $n = mh$ 。很清楚地， $m, h < n$ 。因此，藉由歸納假設， m 與 h 可表示為唯一一組質數乘積。亦即 $m = p_1^{k_1} p_2^{k_2} \dots p_j^{k_j}$ 且 $h = q_1^{l_1} q_2^{l_2} \dots q_i^{l_i}$ 。由於 $n = mh$ ，我們可得 $n = p_1^{k_1} p_2^{k_2} \dots p_j^{k_j} q_1^{l_1} q_2^{l_2} \dots q_i^{l_i}$ 。聚集相等的質數並根據質數值遞增排列，即可得到唯一一組質數乘積。
- 最後根據歸納原則，即可證明定理命題成立。

□

2.2.3 矛盾法

矛盾法 (contradiction) 也稱反證法，出現在古希臘哲學家亞里斯多德所編纂的題旨篇和謬誤論証篇二書中。矛盾法的證明思維為：欲證明某命題為真，則先假設該命題為假，若能由該命題為假的假設推論到邏輯上的矛盾，則能夠證明最初該命題為假的假設是錯誤的，也就能夠證明原命題為真。

西元前 300 年古希臘哲學家歐幾里德在其著名的「幾何原本」著作中以特殊方法證明質數有無窮多個，這也被稱為歐幾里德定理。除了「幾何原本」中的證明之外，後來流傳許多歐幾里德定理的證明，其中有一個證明方法就是矛盾法。以下我們展示如何利用矛盾法證明歐幾里德定理。

定理 2.3. 歐幾里德定理: 質數有無限多個

證明: 假設只有有限的 $n(n > 0)$ 個質數，令其由小而大為 p_1, p_2, \dots, p_n 。考慮整數 p ($p_1 p_2 \dots p_n$) + 1，因為 p 比所有有限的 n 個質數 p_1, p_2, \dots, p_n 都大，根據假設 p 必定為合數，一定具有一個因數 $q, 1 < q < p$ 。我們可以就以下兩個情況討論：

(情況 1): q 是質數 p_1, p_2, \dots, p_n 中的一個。但是這情況不成立，因為根據 p 的定義， p_1, p_2, \dots, p_n 除 p 都餘 1。

(情況 2): q 是合數。但這情況也不成立，因為連續因數分解 q 可得 q 為質數乘積 $q_1 q_2 \dots q_m$ ，其中 $q_1, q_2, \dots, q_m < q < p$ 。因為 p 可被 q 整除，所以 p 也可被質數 q_1, q_2, \dots, q_m 整除，但是根據 p 的定義，所有比 p 小的質數除 p 都餘 1，都不是 p 的因數，所以這情況也不成立。

因為 (情況 1) 與 (情況 2) 都不成立，因此產生矛盾，所以最初的假設是錯誤的，因此得證質數有無限多個。 \square

2.3 演算法的效能

除了演算法的正確性之外，我們也關心演算法的效能 (efficiency)。實際上，我們利用估算演算法使用的資源以衡量演算法的效能。演算法執行時使用的資源包括：時間資源、記憶體資源、網路頻寬資源（當演算法需要透過網路傳輸資料時）、邏輯閘資源（當演算法使用積體電路邏輯閘實作時）等，在本書中我們聚焦於演算法執行時使用的時間資源與記憶體資源。一般而言，演算法的輸入規模 (input size) 愈大，則演算法會使用愈多的資源。我們需要找到一種方法來呈現這種關係。

若我們能夠將所有的演算法以相同的高階程式語言實作，並在相同條件下於相同的電腦上執行，則我們似乎可以利用高階程式語言程式的執行時間與執行時所佔用的記憶體空間來衡量被實作演算法的效能。不過，這太不實際了，而且藉由這個方式我們也難以直接看出演算法使用的資源與演算法輸入規模的關係。因此，我們需要其他更具學理基礎，而且更容易分析與比較的方式來衡量演算法的效能。

在學理上，我們使用時間複雜度 (time complexity) 及空間複雜度 (space complexity) 來分析演算法佔用的執行時間與佔用的記憶體空間的大小。兩種演算法的複雜度都很重要，但是演算法時間複雜度卻相對地更為重要。在許多文獻中（包括本書），在進行演算法複雜度分析時，有時只有時間複雜度分析而沒有空間複雜度分析。因此，在本章中，我們先說明演算法時間複雜度的概念，並說明如何進行演算法時間複雜度分析。而演算法空間複雜度則是類似概念，我們在下一章展示演算法複雜度分析範例時再一併說明。

2.4 演算法的時間複雜度

演算法的時間複雜度指的是演算法執行時需要的步驟 (step) 數，分為以下三種狀況：

- 最佳狀況 (best case) 時間複雜度：演算法執行時需要的最少執行步驟數。
- 最差狀況 (worst case) 時間複雜度：演算法執行時需要的最多執行步驟數。
- 平均狀況 (average case) 時間複雜度：所有可能狀況下演算法執行時需要的平均步驟數。

在上列的定義中，所謂的一個步驟 (step) 指的是演算法中的基本操作 (basic operation)，如算術運算操作 (arithmatic operation) 及邏輯運算操作 (logic operation) 等。有時候我們也會將幾個連續執行的操作視為一個步驟，我們稍後會有詳細的說明。

在所有的時間複雜度之間，最差狀況時間複雜度是很重要的一項。一般我們會先分析演算法的最差狀況時間複雜度，因為這項複雜度可以讓我們了解演算法在最壞的情況下的執行時間概況。例如，假設我們需要使用一些天氣預測演算法來預測三天後的天氣概況，並在一小時之後發佈預測結果。姑且不論演算法預測天氣的準確度，若我們能夠分析出這些演算法的最壞情況時間複雜度，則大約可以推估出哪些演算法即使在最壞情況下也可以在某個特定的時間之內執行完畢，讓我們來得及發佈天氣預測結果。

有一些平日經常使用的演算法，會面臨各種不同狀況，而執行步驟有時多有時少，且碰到最壞與最好狀況的機率又很低，則此種演算法的平均狀況時間複雜度也很重要。例如，我們經常使用排序 (sorting) 演算法將一連串的雜亂數字依由小到大的次序排好順序，或將一連串的雜亂檔案名稱依照字典順序 (lexical order，如字典將單字由 A 開頭排到 Z 開頭的順序)

排列。因為可能面對不同的一連串數字或檔名，因而此時我們除了關心排序演算法的最差狀況時間複雜度之外，我們可能還更重視排序演算法的平均狀況時間複雜度。

相對於最差狀況時間複雜度與平均狀況時間複雜度，演算法的最佳狀況時間複雜度則顯得較不重要。一方面這是因為有些演算法的最佳狀況時間複雜度出現的機率並不高，而另一方面則是因為有些演算法的最佳狀況時間複雜度是顯而易見的 (trivial)。例如，我們直接利用質數 (prime number or prime) 的定義設計一個演算法，檢查一個輸入大於 2 的正整數 n 是不是質數。依照定義，我們只要由整數 2 開始到 n 為止 (不含 n) 找到任何一個 n 的因數 (factor)，就可以判斷 n 不是質數了。對於這個演算法，若我們輸入任何大於 2 的偶數 n ，都可以馬上找出 2 是 n 的因數而判斷 n 不是質數，這對應演算法的最佳狀況時間複雜度，而這是顯而易見的。

一般而言，演算法的平均狀況時間複雜度最難分析 (求出)，最壞狀況時間複雜度稍微容易些，而最佳狀況時間複雜度最容易分析。我們通常會分析演算法的最壞狀況時間複雜度，若有可能，還會分析演算法的平均狀況時間複雜度，但是常常不分析最佳狀況時間複雜度。

我們舉以下檢查一個正整數是否為質數的演算法為例子來說明演算法時間複雜度分析。針對一個大於 1 的正整數而言，所謂質數 (prime) 是指除了 1 和本身之外沒有其他因數的整數，例如，2 是質數，也是唯一為偶數的質數；3、5、7、11 等奇整數為質數，而 4、6、8、10 等偶整數不是質數 (因為 2 是 4、6、8 等偶整數的因數)。在數學定義中，1 既不是質數也不是非質數，而 2 是最小的質數，因此我們的質數檢查演算法只針對大於 2 的正整數。

針對以下所列的演算法 PrimeTest1，我們可以看出，輸入大於 2 的任意正整數 n ，若 n 是質數，則演算法 PrimeTest1 需要執行整數除法求餘數 (也就是 $n \% i$) 操作與整數比較 (也就是 $(n \% i) = 0$) 操作各 $n - 2$ 次，才可以知道 n 是質數。另外，若 n 不是質數，則演算法 PrimeTest1 只要執行整數除法求餘數操作與整數比較操作 1 次就可以知道 n 不是質數。因此，我們很容易看出，在最壞狀況下，演算法 PrimeTest1 的執行時間與輸入的正整數 n 的大小成正比；而在最佳狀況下，演算法 PrimeTest1 的執行時間大約為執行 1 次整數除法求餘數與整數比較操作，而與輸入的正整數 n 的大小無關。我們可以說，若將整數除法求餘數操作與整數比較操作視為一個步驟，則演算法 PrimeTest1 的最差狀況時間複雜度為 $n - 2$ ，而其最佳狀況時間複雜度為 1。我們可以說，若同時將整數除法求餘數操作與整數比較操作視為個別步驟，則演算法 PrimeTest1 的最差狀況時間複雜度為 $2n - 2$ ，而其最佳狀況時間複雜度為 2。我們可以看出，個別步驟的認定會影響演算法時間複雜度的估算。我們將在稍後說明解決個別步驟認定問題的方法。而演算法 PrimeTest1 的平均狀況時間複雜度分析較複雜，在此我們省略不提。

Algorithm PrimeTest1(n)

Input: 一個大於 2 的正整數 n

Output: true 或 false (表示 n 是質數或不是質數)

```
1: for  $i = 2$  to  $n - 1$  do
2:   if  $(n \% i) = 0$  then return false
3: return true
```

2.5 趨近記號

我們使用大 O、大 Ω 及大 Θ 等趨近記號 (asymptotic notation) 表示演算法的複雜度 (complexity)。趨近記號適合表示演算法在輸入規模 (input size) 足夠大時的複雜度。這是我們採用趨近記號表示演算法複雜度的原因之一，說明如下。

一般而言，在演算法的輸入規模較小時，不管是有效率 (複雜度較低) 或沒有效率 (複雜度較高) 的演算法通常都可以很快的執行完畢。相對的，在演算法的輸入規模非常大時，有效率的演算法還是可以在一定的時間內執行完畢，而沒有效率的演算法則可能需要相當長的時間，甚至於需要經年累月才能結束 (我們稍後會舉例說明這個狀況)。因此，在分析演算法時，我們會聚焦於演算法的輸入規模足夠大的狀況，這是分析演算法複雜度採取趨近記號的原因之一。

分析演算法複雜度使用趨近記號原因之二為簡化個別步驟認定。當我們分析演算法時，不同的個別步驟認定會產生不同的時間複雜度。例如，當我們將上一單元的 PrimeTest1 演算法中的整數除法求餘數操作與整數比較操作視為一個步驟時，我們說演算法 PrimeTest1 的最差狀況時間複雜度為 $n - 2$ 。當我們將整數除法求餘數操作與整數比較操作視為二個個別步驟時，我們說演算法 PrimeTest1 的最差狀況時間複雜度為 $2n - 4$ 。而當我們另外將迴圈控制中的迴圈變數遞增與迴圈變數與迴圈結束值的比較再視為另外二個個別步驟時，則我們可以說演算法 PrimeTest1 的最差狀況時間複雜度為 $4n - 8$ 。以上這些時間複雜度的說法，似乎都正確。以下我們會說明，在我們使用趨近記號來表示演算法複雜度的情況下，這些說法都是完全相通的；更精確地說，它們對應的趨近記號表示是完全相同的。這是我們使用趨近記號來表示演算法複雜度的原因之二。

在演算法輸入規模足夠大時，演算法的時間複雜度會趨近於一個量級 (order)。一般而言，演算法的時間複雜度是輸入規模 n 的多項式，當演算法輸入規模足夠大時，時間複雜度多項式中除了最高次方的項目外，其他的部分都可以被忽略；而同時，最高次方項目的常數係數也同時可以被忽略。例如，若一個演算法的時間複雜度為 $n - 2$ 、 $2n - 4$ 或 $4n - 8$ ，則當 n 足夠大時，此演算法的時間複雜度趨近於 n ，屬於一次方或線性 (linear) 量級；若一個演算法的時間複雜度為 $35n^2 + 12n + 11$ ，則當 n 足夠大時，此演算法的時間複雜度趨近於 n^2 ，屬於平方 (quadratic) 量級；而若一個演算法的時間複雜度為 $28n^3 + 1245n^2 + 162n + 321$ ，則當 n 足夠大時，此演算法的時間複雜度趨近於 n^3 ，屬於立方 (cubic) 量級。

在學理上，若是一個演算法的時間複雜度表示為一個多項式，則我們取這個多項式的最高次方為其時間複雜度的量級，並且將此量級以大 O 記號表示。以下我們詳細介紹大 O 記號。

大 O 記號 (Big-O notation) 為一種趨近記號，我們使用大 O 記號來表示演算法在輸入規模足夠大時，其複雜度的量級趨近情形，以下我們正式定義大 O 記號：

定義 2.4. 大 O 記號 (Big-O notation): (O 代表 order 之意)

令 $f(n)$ 與 $g(n)$ 是由非負整數對應至實數的函數，若存在正實數常數 c 和正整數常數 n_0 ，使得對所有的 $n \geq n_0$ 而言， $f(n) \leq cg(n)$ 成立，則我們說 $f(n) = O(g(n))$ 。

因為 $O(g(n))$ 帶有集合的涵義，因此 $f(n) = O(g(n))$ 有 $f(n) \in O(g(n))$ 的意思，唸作「 $f(n)$ 屬於 Big-O of $g(n)$ 」；而比較比較完整的英文唸法為「 f of n is of Big-O of g of n 」)。

以下我們舉一個實例說明大 O 記號的應用。令 $f(n) = 2n^2 + n + 3$ ， $g(n) = n^2$ 。因為存在 $c = 6$ 和 $n_0 = 1$ ，使得當 $n \geq n_0 = 1$ 時， $f(n) = 2n^2 + n + 3 \leq cn^2 = 6n^2 = 6g(n)$ 成立，所以我們說 $f(n) = 2n^2 + n + 3 = O(g(n)) = O(n^2)$ 。

我們可以由圖2.1看出 $f(n) = 2n^2 + n + 3$ 和 $g(n) = n^2$ 的關係。我們可以由圖2.1看出，當 $n \geq 1$ 時， $f(n) \leq 6g(n)$ 成立，我們稱 $g(n)$ 是 $f(n)$ 的趨近上界 (asymptotic upper bound)。

這表示 $g(n)$ 的成長率比 $f(n)$ 還快或一樣快。

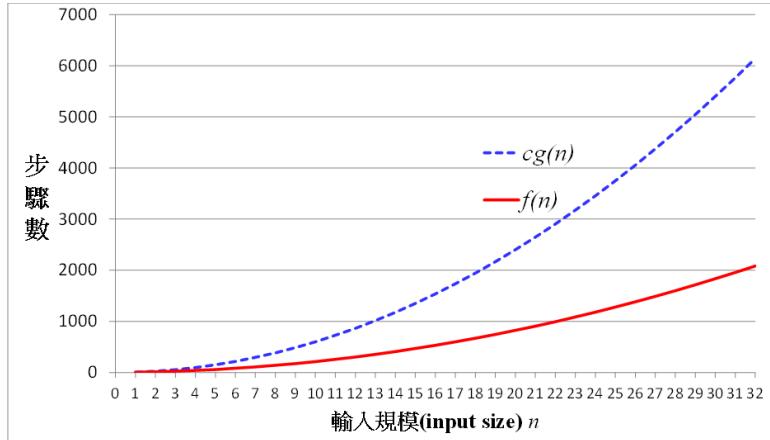


圖 2.1: $f(n) = 2n^2 + n + 3$ 和 $g(n) = n^2$ 的關係圖 ($n_0 = 1, c = 6$)

表2.1舉了一些演算法複雜度的例子，並列出其大 O 記號與量級。常見的演算法複雜度量級有常數 (constant) 量級 ($O(1)$)，也就是 $O(n^0)$ 、次線性 (對數)(sub-linear, logarithmic) 量級 ($O(\log n)$)、線性 (linear) 量級 ($O(n)$)、平方 (quadratic) 量級 ($O(n^2)$)、立方 (cubic) 量級 ($O(n^3)$)、指數 (exponential) 量級 ($O(2^n)$) 等。

| 複雜度 | 以大 O 記號表示 | 量級 |
|---|---------------|--------------------------------------|
| 162 | $O(1)$ | 常數 (constant) 量級 |
| $63 \log n + 4$ | $O(\log n)$ | 次線性 (對數)(sub-linear, logarithmic) 量級 |
| $37\sqrt{n} + 52$ | $O(\sqrt{n})$ | 平方根 (square root) 量級 |
| $n - 2$ | $O(n)$ | 線性 (linear) 量級 |
| $156n + 81$ | $O(n)$ | 線性 (linear) 量級 |
| $35n^2 + 12n + 11$ | $O(n^2)$ | 平方 (quadratic) 量級 |
| $28n^3 + 1245n^2 + 162n + 321$ | $O(n^3)$ | 立方 (cubic) 量級 |
| $14 * 2^n + 457n^3 + 248n^2 - 45n + 81$ | $O(2^n)$ | 指數 (exponential) 量級 |

表 2.1: 一些複雜度的大 O 記號表示及其量級。

除大 O 記號之外，還有兩個類似的趨近記號：大 Ω 記號 (Big-Omega notation) 及 Θ 記號 (Theta notation)，在此我們一併介紹。讀者或許會好奇為何我們要將 O 與 Ω 呃為大 O 與大 Ω 呢？這是因為還有二個類似的記號使用小寫的 o 與 ω ，因此我們要使用大 O 與大 Ω 的稱呼，以便與小 o 與小 ω 區隔。不過在本書中未使用小 o 與小 ω ，因此我們先不介紹它們。

大 Ω 記號的定義恰好與大 O 記號的定義相反，它表示的是一個趨近下界 (asymptotic lower bound) 的概念，以下為其定義：

定義 2.5. 大 Ω 記號 (Big-Omega notation):

令 $f(n)$ 與 $g(n)$ 是由非負整數對應至實數的函數，若存在正實數常數 c 和正整數常數 n_0 ，使得對所有的 $n \geq n_0$ 而言， $f(n) \geq cg(n)$ 成立，則我們說 $f(n) = \Omega(g(n))$ 。

與 $O(g(n))$ 相同， $\Omega(g(n))$ 帶有集合的涵義，因此 $f(n) = \Omega(g(n))$ 有 $f(n) \in \Omega(g(n))$ 的意思，唸作「 $f(n)$ 屬於 Big-Omega of $g(n)$ 」；而比較比較完整的英文唸法為「 f of n is of Big-Omega of g of n 」。

當一個函數 $f(n)$ 同時屬於 Big-O of $g(n)$ (也就是 $f(n)=O(g(n))$) · 也屬於 Big-Omega of $g(n)$ (也就是 $f(n) = \Omega(g(n))$) · 則我們稱「 $f(n)$ 屬於 Theta of $g(n)$ 」· 英文唸法為「 f of n is of Theta of g of n 」· 記為 $f(n) = \Theta(g(n))$ · 同樣的 · $\Theta(g(n))$ 也有集合的涵義。

Θ 記號表示的是一個趨近緊界 (asymptotic tight bound) 的概念 · 以下為其定義：

定義 2.6. Θ 記號 (Theta notation):

令 $f(n)$ 與 $g(n)$ 是由非負整數對應至實數的函數 · 若存在正實數常數 c_1, c_2 和正整數常數 n_0 · 使得對所有的 $n \geq n_0$ 而言 · $f(n) \geq c_1 g(n)$ 與 $f(n) \leq c_2 g(n)$ 同時成立 · 則我們說 $f(n) = \Theta(g(n))$ 。

圖2.2顯示 $f(n) = \Theta(g(n))$ 、 $f(n) = O(g(n))$ 與 $f(n) = \Omega(g(n))$ 的示意圖。

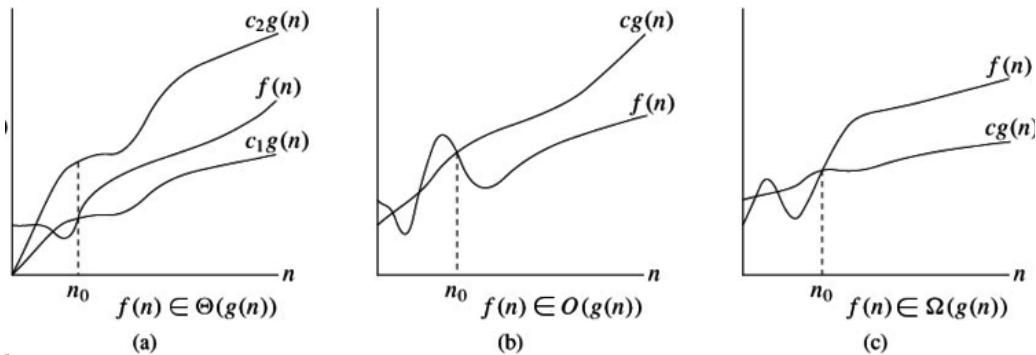


圖 2.2: $f(n) = \Theta(g(n))$ 、 $f(n) = O(g(n))$ 與 $f(n) = \Omega(g(n))$ 示意圖

2.6 降低演算法複雜度量級

在本節中 · 我們希望幫助讀者建立出以下觀念: 「在設計演算法時 · 永遠要想著降低演算法複雜度量級。」我們以時間複雜度量級為例 · 首先比較不同演算法時間複雜度在各種不同量級之下的執行步驟數。由表2.2及圖2.3 · 我們可以發現 · 某些量級在演算法的輸入規模還不是很大的情況下 · 演算法時間複雜度的對照數值(執行步驟數)就已經相當大了 · 這表示演算法需要執行相當久的時間。例如 · 若演算法的一個步驟(例如比較兩個整數的操作)需要一個微秒 (μs , micro second · 百萬分之一秒) 來完成 · 則當輸入規模 n 為 32 時 · 平方量級演算法需要 1,024 微秒 ≈ 0.001 秒來完成 · 而指數量級演算法需要 $2^n = 4,294,967,296$ 微秒 ≈ 143 分鐘來完成。因此 · 如何設計一個複雜度量級比較低的演算法一直是我們必須擺在心中的一個重要目標。以下是演算法複雜度量級高低的次序(靠左邊的量級是比較低的量級) :

$$O(1) \cdot O(\log n) \cdot O(\sqrt{n}) \cdot O(n) \cdot O(n \log n) \cdot O(n^2) \cdot O(n^3) \cdot O(2^n)$$

| $\log n$ | \sqrt{n} | n | $n \log n$ | n^2 | n^3 | 2^n |
|----------|------------|-----|------------|-------|--------|---------------|
| 0 | 1.00 | 1 | 0 | 1 | 1 | 2 |
| 1 | 1.41 | 2 | 2 | 4 | 8 | 4 |
| 2 | 2.00 | 4 | 8 | 16 | 64 | 16 |
| 3 | 2.83 | 8 | 24 | 64 | 512 | 256 |
| 4 | 4.00 | 16 | 64 | 256 | 4,096 | 65,536 |
| 5 | 5.66 | 32 | 160 | 1,024 | 32,768 | 4,294,967,296 |

表 2.2: 演算法各種時間複雜度量級的對照數值。

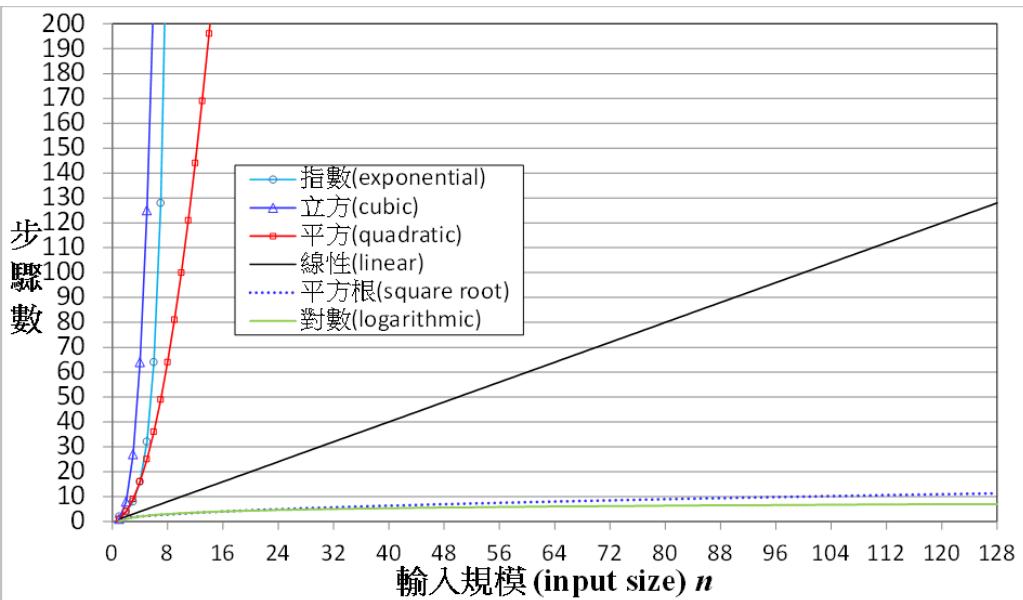


圖 2.3: 演算法各種時間複雜度量級成長圖。

以下，我們再舉檢查一個正整數是否為質數的演算法為例，來看看演算法的時間複雜度量級對執行時間的影響。我們先看以下的觀察，若我們能善加利用它，則我們可以很容易的設計出一個量級比直接使用質數定義設計的演算法的量級更低的演算法。

[觀察]

對於任意的大於 2 的整數 n 而言，若所有小於或等於 \sqrt{n} 的整數 (1 除外) 都無法整除 n ，則 n 是一個質數。

以下我們舉一個實例來說明這個觀察。我們知道每個整數的因數都是成對出現的，例如，16 的因數為 1 與 16($1 \times 16 = 16$)、2 與 8($2 \times 8 = 16$)、4 與 4($4 \times 4 = 16$)。成對出現的因數中，一個會小於等於該數的平方根，而另一個則是大於等於該數的平方根。因此，我們只要檢查小於等於該數的平方根的整數中是否有其因數就可以知道該數是不是質數了。

我們根據上列觀察設計出一個屬於平方根時間複雜度量級的質數檢查演算法 – 演算法 PrimeTest2：

Algorithm PrimeTest2(n)

Input: 一個大於 2 的正整數 n

Output: true 或 false(表示 n 是質數或不是質數)

```

1: for  $i = 2$  to  $\sqrt{n}$  do
2:   if  $(n \% i) = 0$  then return false
3: return true

```

我們可以很輕易的推導出，演算法 PrimeTest2 的最差狀況 (worst case) 時間複雜度為 $\sqrt{n} - 1 = O(\sqrt{n})$ ，屬於平方根量級。而演算法 PrimeTest2 與演算法 PrimeTest1 一樣，二者的最佳狀況 (best case) 時間複雜度都是 $1 = O(1)$ (屬於常數量級)。在最差狀況下，平方根量級的演算法 PrimeTest2 的執行步驟數將比線性量級的演算法 PrimeTest1 的執行步驟數

少，它們的差距是演算法輸入規模的平方根倍。平方根倍看似不大，但是當演算法輸入規模很大時，它的平方根倍也是很驚人的數字。例如，若演算法輸入的 n 為 2147483647，則演算法 PrimeTest1 需要執行 $n - 2 = 2147483645$ 次整數除法求餘數及整數比較敘述才可以得知 2147483647 是質數，而演算法 PrimeTest2 只要執行 $\sqrt{n} - 1 = \sqrt{214783647} - 1 = 46339$ 次整數除法求餘數及整數比較敘述就可以得知 2147483647 是質數了。二個演算法的執行時間差距約為 $\sqrt{n} \approx 46340$ 倍，當然，當所輸入的 n 越大時，這種差距越大。

如程式列表2.1及2.2所示，我們將演算法 PrimeTest1 和演算法 PrimeTest2 以 Python 語言 prime_test1() 函式及 prime_test2() 函式實作，並在其中插入測量執行時間相關敘 time.time() 函式來取得二個演算法以 Python 語言實作的執行時間。當輸入的整數 n 為 2147483647 時，演算法 PrimeTest1(也就是 prime_test1() 函式) 的執行時間為 851.4888181686401 秒，而演算法 PrimeTest2(也就是 prime_test2() 函式) 的執行時間為 0.019001007080078125 秒，二個演算法的執行時間差距為 44812 倍，其倍數約略等於 $\sqrt{n} = 46340$ 。從這二個演算法以 Python 語言函式實作的執行情形，我們可以清楚地的看出演算法的時間複雜度對執行時間的顯著影響：prime_test2() 函式瞬間執行完畢，而 prime_test1() 函式超過 14 分鐘才執行完畢。

程式列表 2.1: prime_test1.py

```
#File: prime_test1.py
#Note: To input an integer n which is larger than or equal to 2 and output if n is a prime
import time
s=input("Please enter an interger which is equal to or larger than 2: ")
n=int(s)
def prime_test1(n):
    for i in range(2,n):
        if n%i == 0:
            return False
    return True
t1=time.time()
is_prime=prime_test1(n)
t2=time.time()
print(f'{n} is prime.' if is_prime else f'{n} is not prime.')
print(f'The time consumed by prime_test1 is {t2-t1} seconds.')
```

執行結果：

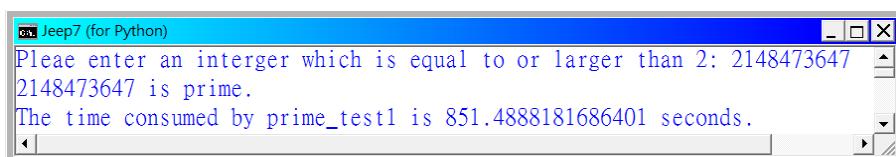


圖 2.4: prime_test1.py 程式執行結果

程式列表 2.2: prime_test2.py

```
#File: prime_test2.py
#Note: To input an integer n which is larger than or equal to 2 and output if n is a prime
import time
s=input("Please enter an interger which is equal to or larger than 2: ")
n=int(s)
def prime_test1(n):
    for i in range(2,n):
        if n%i == 0:
            return False
    return True
t1=time.time()
is_prime=prime_test1(n)
```

```
t2=time.time()
print(f'{n} is prime.' if is_prime else f'{n} is not prime.')
print(f'The time consumed by prime_test1 is {t2-t1} seconds.')
```

圖 2.5: prime_test2.py 程式執行結果

2.7 結語

有句出於清朝和邦額(號霽園主人)所著《夜譚隨錄·卷三·鐵公雞篇》的諺語說：「又要馬兒好，又要馬兒不吃草」，比喻要求過高卻吝於付出。這句形容人苛刻的諺語用於形容演算法確是貼切的，我們必須要求演算法是好的，也就是正確的，能夠產生正確結果的；而卻也必須要求演算法使用最少的資源，包括時間（也就是執行步驟數）、記憶體、網路頻寬、邏輯閘等資源。因此，我們將本章的副標題取為—要馬好也要馬少吃草。

在本章中，我們說明如何證明演算法的正確性，介紹演譯法、歸納法及矛盾法等常用的證明方法。並分別示範如何以這三個方法證明歐幾里德輾轉相除最大公因數演算法、唯一分解定理及歐幾里德定理。本章介紹大 O、大 Ω 及大 Θ 等趨近記號，也說明如何藉由大 O 記號來評估演算法使用的資源以分析其效能，能夠分析出演算法的最佳狀況、最差狀況及平均狀況複雜度，如時間複雜度及空間複雜度等。這些複雜度由低而高可分為常數、次線性（對數）、線性、平方、立方等多項式量級以及指數量級；一般而言，複雜度較低的演算法效能較佳。藉由此章，讀者能夠了解如何證明演算法的正確性與分析演算法在各種狀況下的效能，並建立出以下的觀念：設計出的演算法一定要是正確的，而且最好具有較小量級的複雜度，特別要避免指數量級複雜度，因為這種複雜度會隨著演算法輸入規模的大小非常快速的成長，也就是意味演算法的效能低落。

Chapter 3

演算法的複雜度分析實例 — 決勝在數大時

Contents

| | |
|-----------------|----|
| 3.1 演算法複雜度分析 | 29 |
| 3.2 費伯納西數列演算法 | 30 |
| 3.3 遞迴費伯納西數列演算法 | 32 |
| 3.4 氣泡排序演算法 | 34 |
| 3.5 插入排序演算法 | 36 |
| 3.6 堆積排序演算法 | 37 |
| 3.7 結語 | 40 |

3.1 演算法複雜度分析

我們已經知道在分析演算法時間複雜度時應該著眼於輸入規模比較大的時候演算法需要的步驟 (基本操作數) 的量級，因而採用大 O 趨近記號作為分析工具。因為當輸入規模不大時，許多不同量級的演算法都可以很快地執行結束，只有當輸入規模非常大的時候，不同量級演算法的執行時間或所需要的步驟數就會非常明顯地不同。

演算法的空間複雜度指的是演算法執行時期所佔用的記憶體空間，包括演算法輸入所佔用的空間及演算法執行時所佔用稱為輔助空間 (auxiliary space) 的額外臨時工作記憶體 (working memory)。通常演算法的輸入所佔用的存儲空間是固定的，不隨演算法的不同而改變，因此，我們通常在分析演算法的空間複雜度時只考慮演算法佔用的輔助空間，包括額外的臨時變數空間和程式遞迴呼叫所需的堆疊空間等。

與演算法時間複雜度一樣，演算法空間複雜度也採用大 O 趨近記號作為分析工具。同樣比較重視演算法在輸入規模非常大的時候所佔用的記憶體空間。另外，同樣地，演算法空間複雜度也分為最佳狀況、最差狀況及平均狀況。但是常常這三個狀況的複雜度都一樣，而人們往往只分析最差狀況空間複雜度，因為我們必須先配置好足夠大的記憶體空間以使演算法正確地執行完畢。

在本章中，我們以費伯納西數列 (Fibonacci sequence) 演算法、遞迴費伯納西數列 (recursive Fibonacci sequence) 演算法、氣泡排序 (bubble sort) 演算法、插入排序 (insertion sort) 演算法與堆積排序 (heap sort) 演算法為例，展示演算法的複雜度分析過程。並藉以說

明多項式時間複雜度演算法 (polynomial time-complexity algorithm) 與指數時間複雜度演算法 (exponential time-complexity algorithm) 的分類。

3.2 費伯納西數列演算法

費伯納西數列演算法可以產生費伯納西數列，以下為費伯納西數列的定義：

定義 3.1. 費伯納西數列 (Fibonacci sequence) $F_1, F_2, \dots, F_n, \dots$ 定義如下：

$$F_1 = F_2 = 1; F_n = F_{n-1} + F_{n-2}$$

根據定義，費伯納西數列 (Fibonacci sequence) 為以下數列：1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...。此數列早在西元六世紀就在印度流傳，而義大利數學家李奧納多·費伯納西 (Leonardo Fibonacci, 1175-1250) 是西方第一個研究費伯納西數列的學者，該數列因而得名。

以下一個有趣的例子可以解釋費伯納西數列。假設某種單細胞生物 (如變形蟲) 在理想假設條件下永遠不死，且此種單細胞生物在出生滿一分鐘後成熟，成熟後每滿一分鐘就可以分裂生出一個新的細胞，而分裂過程會在一分鐘內完成。若在第 1 分鐘剛開始就同時放入 1 個單細胞生物，則第 2 分鐘剛開始時，也還是 1 個細胞，因為放入的細胞剛成熟，正要開始分裂。而這個細胞會在第 2 分鐘結束前產出一個新細胞，因此在第 3 分鐘剛開始時總共有 2 個細胞。在第 4 分鐘剛開始時，這 2 個細胞依然存在，而且 2 個細胞中較老的一個已經在第 3 分鐘結束前完成分裂生出一個新的細胞，因此在第 4 分鐘剛開始時一共有 $1+2=3$ 個細胞。一般而言，假設在第 $n-2$ 分鐘剛開始共有 a 個細胞，第 $n-1$ 分鐘剛開始總共有 b 個細胞，則在第 n 分鐘剛開始總共有 $a+b$ 個細胞。這是因為第 $n-2$ 分鐘剛開始時就存在的 a 個細胞，可以在第 $n-1$ 分鐘結束前生出 a 個新細胞，這 a 個新細胞，再加上原來在第 $n-1$ 分鐘剛開始時就存在的 b 個舊細胞總共有 $a+b$ 個。因此在第 n 分鐘剛開始時總共有 $a+b$ 個細胞。

德國天文學家克卜勒 (Johannes Kepler, 1571-1630) 發現費伯納西數列數列前後兩項之比 (後項/前項): $2/1, 2/3, 5/3, 8/5, 13/8, 21/13, 34/21, \dots$ 會形成一個趨近黃金分割比例 (簡稱黃金比例) ϕ 的數列。一般認為按照黃金比例構圖比較具有藝術美感，它其實是兩個數字的比例，類似將一條線分成兩部分，使較長的一段 a 與較短的一段 b 之比等於全長 $a+b$ 與較長 a 的一段之比。我們可以由 $(a+b)/a = a/b = \phi$ 求出黃金分割比例，它的精確值為 $\frac{1+\sqrt{5}}{2}$ ，大約是 1.61803398874989484820...。黃金比例 ϕ 還具有一個非常有趣的性質，也就是 $\frac{1}{\phi} = \phi - 1 = 0.61803398874989484820\dots$ 。圖3.1顯示的費伯納西螺旋 (Fibonacci spiral) 就呈現出接近黃金分割的美麗構圖，自然界中常存在這樣的美麗構圖，如鸚鵡螺及向日葵花等，都可讓人感受到視覺上的美感。

以下為費伯納西數列演算法，可以用以求出費伯納西數列第 n 項的值：

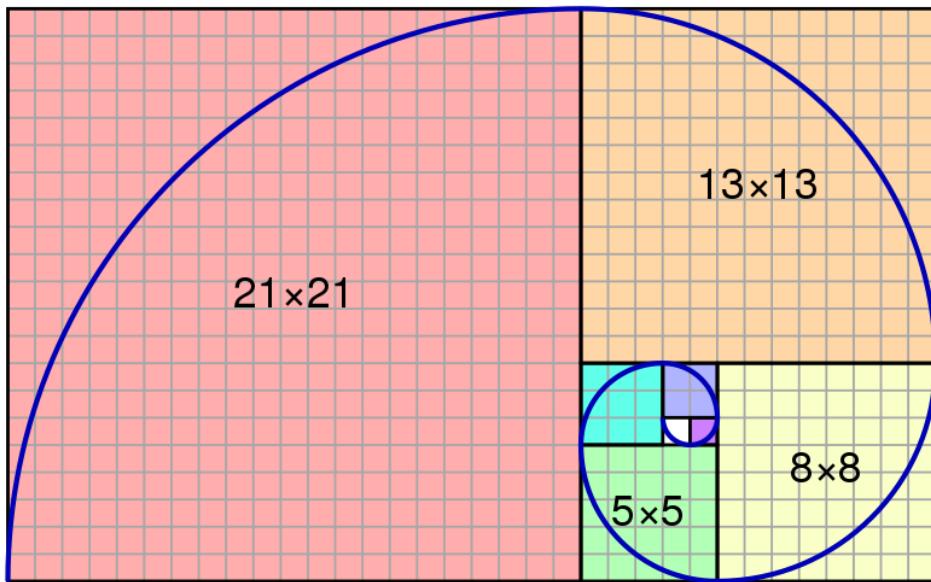


圖 3.1: 顯示費伯納西螺旋

(來源: <https://commons.wikimedia.org/wiki/File:FibonacciSpiral.svg>)

Algorithm Fibonacci(n)

▷ 費伯納西數列演算法

Input: 正整數 n

Output: 費伯納西數列第 n 項

```

1: if  $n = 1$  or  $n = 2$  then
2:     return 1
3: else
4:      $a \leftarrow 1$ 
5:      $b \leftarrow 1$ 
6:     for  $i \leftarrow 3$  to  $n$  do
7:          $c \leftarrow a + b$ 
8:          $a \leftarrow b$ 
9:          $b \leftarrow c$ 
10:    return  $c$ 
```

以下我們分析費伯納西數列演算法的複雜度。費伯納西數列演算法只用到 3 個額外的工作變數 (a 、 b 與 c)，因此它的空間複雜度為 $O(1)$ 。這個演算法最基本的操作為第 7、第 8 及第 9 行的步驟，一共執行 $n - 3$ 次，因此時間複雜度為 $O(n)$ 。

下列的 Python 語言程式 Fibonacci.py 以 $\text{Fibonacci}(n)$ 函數實作費伯納西數列演算法。由執行結果我們可以看出，程式 Fibonacci.py 使用 0.00999990463256836 秒快速算出費伯納西數列第 5000 項的值，是一個非常有效率的演算法。

程式列表 3.1: Fibonacci.py

```

#File: Fibonacci.py
#Note: To input an integer which is equal to or larger than 1
#      to output the nth element of the Fibonacci sequence
import time
def Fibonacci(n):
    if n == 1 or n == 2:
        return(1)
    else:
        return(Fibonacci(n-1) + Fibonacci(n-2))
```

```

else:
    a=1
    b=1
    for i in range(3, n+1):
        c=a+b
        a=b
        b=c
    return(c)
s=input("Please enter an integer which is equal to or larger than 1: ")
n=int(s)
t1=time.time()
Fibo=Fibonacci(n)
t2=time.time()
print(f'The {n}th item of the Fibonacci sequence is {Fibo}.')
print(f'The time consumed by Fibonacci is {t2-t1} seconds.')

```

執行結果:

```

Please enter an integer which is equal to or larger than 1: 5000
The 5000th item of the Fibonacci sequence is 3878964543883256337019163083259053
1208212771464624510610597214895550139044037097010822916462210669479293452858882
97381348310200895498294036143015691147893836421656394410691021450563413370655865
62382546567007125259299038549338139288363783475189087629707120333370529231076930
08518093849801803847813996748881765554653788291644268912980384613778969021502293
08247566634622492307188332480328037503913035290330450584270114763524227021093463
76991040067141748832984228914912731040543287532980442736768229772449877498745556
91907703880637046832794811358973739993110106219308149018570815397854379195305617
51076105307568878376603366735544525884488624161921055345749367589784902798823435
10235998446639348532564119522218595630604753646454707603309024208063825849291564
52876291575759142343809142302917491088984155209854432486594079793571316841692868
03954530954538869811466508206686289742063932343848846524098874239587380197699382
03171742089322654688793640026307977800587591296713896342142525791168727556003603
11370547754724604639987588046985178408674382863125.
The time consumed by Fibonacci is 0.00999990463256836 seconds.

```

費伯納西數列演算法的時間複雜度為 n 的一次方多項式函數，因此它是一個多項式時間複雜度演算法 (polynomial time-complexity algorithm)，簡稱為多項式時間演算法 (polynomial time algorithm)，也有人簡稱為多項式演算法 (polynomial algorithm)。

3.3 遞迴費伯納西數列演算法

伯納西數列的定義是遞迴的形式，因此很自然的我們也可以設計遞迴費伯納西數列演算法來產生費伯納西數列，以下是遞迴費伯納西數列演算法，可以用以求出費伯納西數列第 n 項的值：

Algorithm RecursiveFibonacci(n)

▷ 遞迴費伯納西數列演算

Input: 正整數 n

Output: 費伯納西數列第 n 項

- 1: **if** $n = 1$ or $n = 2$ **then**
- 2: **return** 1
- 3: **else**
- 4: $a \leftarrow \text{RecursiveFibonacci}(n - 2)$
- 5: $b \leftarrow \text{RecursiveFibonacci}(n - 1)$
- 6: **return** $a + b$

以下我們分析費伯納西數列演算法的複雜度，我們先分析空間複雜度。因為在任何時刻遞迴費伯納西數列演算法最多使用到 $n - 2$ 次遞迴呼叫之後，就會碰到 n 是 2 的狀況並返回上一層遞迴呼叫後的位置，會取出遞迴呼叫前的堆疊資料，在釋放放佔用的堆疊空間之後繼續執行後續的步驟。因此，遞迴費伯納西數列演算法最多只會佔用 $O(n)$ 的堆疊空間，空間複雜度為 $O(n)$ 。

遞迴費伯納西數列演算法依照定義編寫，非常簡潔。它可以正確產生費伯納西數列第 n 項，但是卻有量級相當高的時間複雜度，以下是遞迴費伯納西數列演算法的時間複雜度分析：

假設遞迴費伯納西數列演算法的時間複雜度為 $T(n)$ ，則我們有：

$$\begin{aligned} T(1) &= T(2) = 1 \\ T(n) &= T(n-1) + T(n-2) + 1 \leq 2T(n-1) + 1 \\ &= 2(2T(n-2) + 1) + 1 \\ &= 4(2T(n-3) + 1) + 1 + 2 = \dots \\ &= 2^k T(n-k) + (1 + 2 + \dots + 2^{k-1}) = 2^k T(n-k) + (2^k - 1) \end{aligned}$$

令 $n - k = 1$ ，則代入 $k = n - 1$ 我們可得

$$T(n) \leq 2^{n-1} + (2^{n-1} - 1) \leq 2^n \text{ for } n \geq 3$$

因此， $T(n) = O(2^n)$

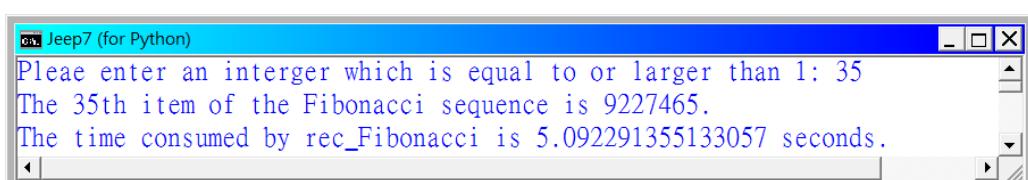
遞迴費伯納西數列演算法的時間複雜度 $O(2^n)$ 為 n 出現在指數的函數，因此它是一個指數時間複雜度演算法 (exponential time-complexity algorithm)，簡稱為指數時間演算法 (exponential time algorithm)，也有人簡稱為指數演算法 (exponential algorithm)。

下列的 Python 語言程式 rec_Fibonacci.py 以 rec_Fibonacci(n) 函數實作遞迴費伯納西數列演算法。由執行結果我們可以看出，程式 rec_Fibonacci.py 使用了 5.092291355133057 秒計算出費伯納西數列第 35 項的值，與費伯納西數列演算法相比，遞迴費伯納西數列演算法顯然是一個效率非常差的演算法。若計算費伯納西數列第 35 項的值需要 5 秒，那麼計算第 50 項的值則需要 $5^{(50-35)} = 30517578125$ 秒，約為 967.7 年才能計算出費伯納西數列第 50 項的值。這是一個不可行的演算法，因為沒有人可以為一個計算結果等那麼久的時間。

程式列表 3.2: rec_Fibonacci.py

```
#File: rec_Fibonacci.py
#Note: To input an integer which is equal to or larger than 1, and
#      to output the nth element of the Fibonacci sequence
import time
def rec_Fibonacci(n):
    if n == 1 or n == 2:
        return(1)
    else:
        a=rec_Fibonacci(n-2)
        b=rec_Fibonacci(n-1)
        return(a+b)
s=input("Please enter an interger which is equal to or larger than 1: ")
n=int(s)
t1=time.time()
Fibo=rec_Fibonacci(n)
t2=time.time()
print(f'The {n}th item of the Fibonacci sequence is {Fibo}.')
print(f'The time consumed by rec_Fibonacci is {t2-t1} seconds.')
```

執行結果：



遞迴費伯納西數列演算法因為採用遞迴方式不斷重複呼叫已經呼叫過的部分，因此時間複雜度為 $O(2^n)$ ，成為指數時間複雜度演算法。我們在上一章中已經學到，指數時間複雜度演算法的執行步驟數會隨著 n 成長得非常快。我們使用 Python 語言實作遞迴費伯納西數列演算法的執行結果也顯示了這樣的現象。若可能的話，我們一定要避免設計與使用指數時間複雜度演算法來解決問題。除非針對某些特定的問題，我們真的沒有辦法找到或設計出多項式時間複雜度演算法來解決它們，我們不得已只好使用指數時間複雜度演算法。事實上，我們還有其他策略可以避開使用指數時間複雜度演算法而還能得到問題的近似解答，我們將在本書稍後再討論這個議題。

3.4 氣泡排序演算法

所謂排序 (sorting) 是將一系列的元素 (資料) 依照某種順序排列的程序。例如，若元素為數值，則排序將依元素數值的大小以由小而大或由大而小的數值順序 (numerical order) 排列；又例如，若元素為字串，則排序將依字串的字典順序 (lexical order) 排列。在節中，我們針對一個元素為數值的陣列，說明氣泡排序演算法如何將陣列元素依由小而大的數值順序排列。然後我們針對氣泡排序演算法的時間複雜度、空間複雜度進行分析。

氣泡排序 (bubble sort) 演算法又稱為下沉 (sinking) 排序演算法或交換 (exchange) 排序演算法。因為它的執行步驟中每回合 (pass) 會找出目前未處理過資料中最大的一個，就如同讓最大的氣泡先由水中冒出，然後再讓第二大的氣泡冒出，...，因此稱為氣泡排序演算法。類似的說法為在執行步驟中，如同每回合會讓最重的東西往下沉，然後再讓第二重的東西往下沉，...，因此稱為下沉排序演算法。又因為它的執行步驟為不斷的比較並交換兩個相鄰的資料，因此又稱為交換排序演算法。

假設我們現在要使用氣泡排序演算法來將陣列中的 n 個元素或資料 (索引為 $0, \dots, n - 1$) 依照其值以由小而大的次序排列，其做法為持續兩兩比較相鄰資料，若前一筆 (左邊) 資料的值大於後一筆 (右邊) 資料的值，則將此二筆資料互相交換，否則資料的位置即維持不動。上述的比較及交換過程一直持續進行到最後一筆資料被比對到為止，這稱為一個回合 (pass)。第一個回合的比對進行到最後一筆資料為止 ($n - 1$ 次比對)，此時具最大值的資料已經調換至最後 (最右邊) 的位置了；而第二回合的比對則進行到倒數第二筆資料為止，此時具第二大值的資料已經調換至倒數第二個位置了；...；其餘依此類推。當進行完第 $n - 1$ 個回合 (比較索引為 0 與 1 的資料) 之後，則所有的資料均已依由小到大的次序排列好了。

以下為氣泡排序演算法。

Algorithm BubbleSort(A, n)

Input: 具 n 個整數的陣列 A

Output: 陣列 A (其中的整數已依其數值由小到大排列)

```
1: for  $i \leftarrow n - 1$  to 1 do           ▷  $i$  代表每回合被比較到的最後一項資料的索引
2:   for  $j \leftarrow 0$  to  $i - 1$  do          ▷  $j$  代表當下正要兩兩比較資料的較小索引
3:     if  $A[j] > A[j + 1]$  then
4:       swap( $A[j], A[j + 1]$ )           ▷ 表示將  $A[j]$  與  $A[j + 1]$  對調
5: return  $A$ 
```

我們舉圖3.2的實例來看看氣泡排序的運作情形：

假設有一個陣列具有 5 個元素 $8, 5, 8', 6, 7$ ，索引為 $0, \dots, 4$ ，其中 8 與 $8'$ 二個元素的值

都是 8，但是為了區別起見，我們將之標示為 8 與 8'。此陣列經由氣泡排序演算法排序的過程如圖3.2所示。在圖3.2中，加垂直線的方格代表索引變數 i 所指的元素，而加水平線的方格代表索引變數 j 所指的元素。

在整個氣泡排序演算法的過程中，8 與 8' 的相對位置一直保持不變，也就是 8 一直排列在 8' 之前，當一個排序演算法能夠讓具有相同值的元素維持原來的相對位置時，我們稱這種排序演算法為穩定 (stable) 排序演算法。若我們所排序的元素只是一個資料庫的鍵值 (key)，則穩定排序演算法在排序之後可以保持所有紀錄原來的次序，這是非常有用的功能。例如，我們可以先針對學生成績資料庫先依國文成績由高到低排序，之後再依總成績由高到低排序。此時資料庫資料的排列次序為總分高的在前，而總分相同的則由國文成績高的排在前面。

另外，除了幾個索引變數之外，氣泡排序演算法不需要額外的記憶體空間來輔助排序的進行，這種不需要額外記憶體空間的排序演算法稱為就地 (in place) 演算法。這也是很好的特定，這表示氣泡排序演算法的空間複雜度可以視為是常數的 (也就是 $O(1)$)，而與所需要排序的陣列的大小無關。

以下我們分析氣泡排序演算法的時間複雜度。如前所述，氣泡排序演算法一共需要 $n - 1$ 個回合才能完成排序工作。第一個回合需要 $n - 1$ 次比較操作 (步驟)，第二個回合需要 $n - 2$ 次比較操作 (步驟)，..., 第 $n - 1$ 個回合需要 1 次比較操作 (步驟)。因此，氣泡排序演算法的時間複雜度為 $\sum_{i=n-1}^1 i = \frac{n(n-1)}{2} = O(n^2)$ 。對所有的狀況 (最佳、最差與平均狀況) 而言，氣泡排序演算法的時間複雜度都是 $O(n^2)$ 。

| 元素索引 回合 | 0 | 1 | 2 | 3 | 4 | 操作情形 |
|--------------|---|---|----|----|----|--------------------------|
| Pass1 | 8 | 5 | 8' | 6 | 7 | $i=4, j=0, 8$ 與 5 對調 |
| | 5 | 8 | 8' | 6 | 7 | $i=4, j=1, 8$ 與 $8'$ 無對調 |
| | 5 | 8 | 8' | 6 | 7 | $i=4, j=2, 8'$ 與 6 對調 |
| | 5 | 8 | 6 | 8' | 7 | $i=4, j=3, 8'$ 與 7 對調 |
| Pass2 | 5 | 8 | 6 | 7 | 8' | $i=3, j=0, 5$ 與 8 無對調 |
| | 5 | 8 | 6 | 7 | 8' | $i=3, j=1, 8$ 與 6 對調 |
| | 5 | 6 | 8 | 7 | 8' | $i=3, j=2, 8$ 與 7 對調 |
| Pass3 | 5 | 6 | 7 | 8 | 8' | $i=2, j=0, 5$ 與 6 無對調 |
| | 5 | 6 | 7 | 8 | 8' | $i=2, j=1, 6$ 與 7 無對調 |
| Pass4 | 5 | 6 | 7 | 8 | 8' | $i=1, j=0, 5$ 與 6 無對調 |

圖 3.2: 氣泡排序演算法執行過程。

3.5 插入排序演算法

在本節中，我們介紹插入排序 (insertion sort) 演算法。假設我們想要使用插入排序演算法將一個陣列中的元素依照其值由小而大排列，其做法是從第二個元素開始，先以其為處理對象，並向前比對每一個元素以找出其適當插入位置；然後以第三個元素為處理對象，並向前比對每一個元素以找出其適當插入位置，...。這類似我們在交考卷時的一種排序概念：每位學生寫完考卷時就交上自己的考卷；而老師為了方便讓考卷依座號的次序由小到大排序，在每個學生交上考卷時即在已經交上的考卷中從頭到尾（或從尾到頭）找出新交考卷的適當位置插入。如此，在老師手上的考卷永遠都是依座號的次序由小到大排列，而當所有學生都交出考卷時，所有的考卷就已經依座號的次序由小到大排序完畢了。以下為插入排序演算法。

Algorithm InsertionSort(A, n)

Input: 具 n 個整數的陣列 A

Output: 陣列 A (其中的整數已依其數值由小到大排列)

```
1: for  $i \leftarrow 1$  to  $n - 1$  do                                ▷  $i$  代表正要被插入資料的索引
2:    $j \leftarrow i - 1$ 
3:    $t \leftarrow A[i]$ 
4:   while ( $t < A[j]$ )  $\wedge (j >= 0)$  do
5:      $A[j + 1] \leftarrow A[j]$ 
6:      $j \leftarrow j - 1$ 
7:    $A[j + 1] = t$ 
8: return  $A$ 
```

我們舉圖3.3中的實例來看插入排序演算法的運作過程：

假設有一個陣列具有 5 個元素 $8, 5, 8', 6, 7$ ，索引 (index) 為 $0, \dots, 4$ ，其中 8 與 $8'$ 二個元素的值都是 8 ，但是為了區別起見，我們將之標示為 8 與 $8'$ 。此陣列經由插入排序演算法排序的過程如圖3.3所示。

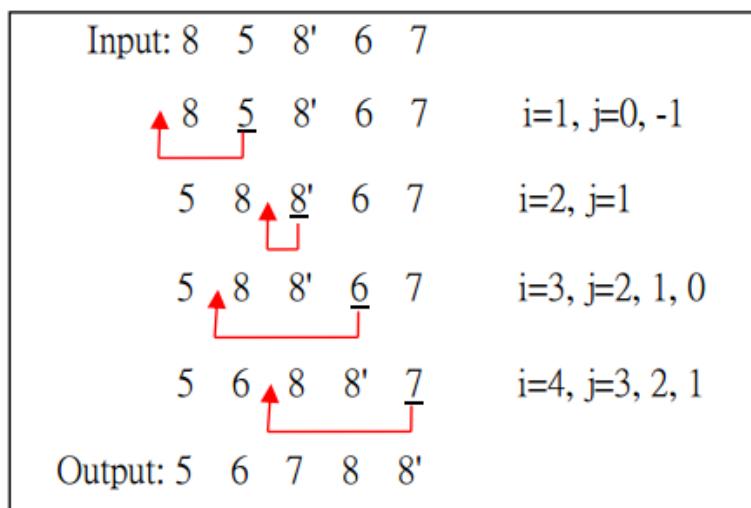


圖 3.3: 插入排序演算法排序過程

在整個插入排序演算法的執行過程中， 8 與 $8'$ 的相對位置一直保持不變，因此插入排序演算法也是一個穩定 (stable) 排序演算法。

除了幾個索引變數與一個陣列數值臨時儲存變數之外，插入排序演算法不需要額外的記憶體空間來輔助排序的進行，因此插入排序演算法也是一個就地 (in place) 演算法，其空間複雜度為 $O(1)$ 。

以下我們分析插入排序演算法的時間複雜度。假設 m_i 代表 $A[i]$ 在插入適當位置前，陣列中資料需要在陣列中移動的次數，而 M 是插入排序演算法所需要的資料總移動次數。那麼，我們可以寫下以下的式子：

$$M = \sum_{i=1}^{n-1} (2 + m_i)$$

其中，因為插入排序演算法在每次插入一個資料 $A[i]$ 時 ($1 \leq i \leq n - 1$)，會將 $A[i]$ 的資料先儲存到 t ，然後最後再將存在 t 中的資料儲存到 $A[j + 1]$ ，因此在加總式子中會有一個加 2 的項目。

資料已經由小到大排好的情況下為最佳狀況，在這種情況下，所有的 $m_i = 0$ ($1 \leq i \leq n - 1$)，因此我們有 $M = 2(n - 1) = 2n - 2 = O(n)$ 。當資料為反向的由大到小排列時為最差狀況，在這種情況下， $m_1 = 1; m_2 = 2; \dots; m_{n-1} = n - 1$ ，或者我們寫為 $m_i = i$ ($1 \leq i \leq n - 1$)，因此我們有 $M = \frac{n(n-1)}{2} + 2(n - 1) = O(n^2)$ 。

以下我們求平均狀況下的時間複雜度。假設當 $A[i]$ 準備插入已經排好次序的 $A[0], \dots, A[i-1]$ 之中時， $A[i]$ 是所有 $i+1$ 個資料中最大的資料的機率為 $\frac{1}{i+1}$ (這個狀況下 $m_i = 0$)， $A[i]$ 是所有 $i+1$ 個資料中第二大的資料的機率也為 $\frac{1}{i+1}$ (這個狀況下 $m_i = 1$)， \dots ， $A[i]$ 是所有 $i+1$ 個資料中最小的 (第 $i+1$ 大的) 資料的機率也為 $\frac{1}{i+1}$ (這個狀況下 $m_i = i$)。以期望值來估算，我們有 $m_i = \frac{(0+1+\dots+i)}{(i+1)} = \frac{\frac{i(i+1)}{2}}{(i+1)} = \frac{i}{2}$ 。因此 $M = \sum_{i=1}^{n-1} (2 + \frac{i}{2}) = 2(n - 1) + \sum_{i=1}^{n-1} (\frac{i}{2}) = 2(n - 1) + \frac{n(n-1)}{4} = O(n^2)$ 。

3.6 堆積排序演算法

堆積排序 (heap sort) [3] 演算法使用堆積 (heap) 資料結構進行排序，它非常的有效率，具有非常低的時間複雜度與空間複雜度。以下我們先介紹堆積資料結構。

堆積是一棵完整二元樹 (complete binary tree)，也就是說除了最深的那一層之外，其他層都完全填滿，而在最深的那一層，所有的節點都盡量向左靠。另外，堆積還必須滿足父節點的值永遠大於等於 (或小於等於) 其子節點的值，因此根節點具有所有元素中的最大值 (或最小值)。其中，最大堆積 (max heap) 滿足父節點的值永遠大於等於其子節點的值，因此根節點具有所有元素中的最大值。而最小堆積 (min heap) 則滿足父節點的值永遠小於等於其子節點的值，因此根節點具有所有元素中的最小值。

圖3.4是一個包含 5 個節點的最大堆積。我們可以看出它是一棵完整二元樹，而且所有的父節點的值都大於或等於子節點的值，因此樹根具有最大的值 8。

堆疊可以很容易使用陣列直接表示。其作法為由樹根開始由左而右由 1 起始為每一個節點加上編號，並作為陣列索引值。因此，樹根的編號為 1，其左子節點編號為 2，右子節點編號為 3， \dots 。以圖3.4的最大堆疊為例，它可以使用圖3.5的陣列表示。一般而言，若一個節點的編號 (或陣列索引值) 為 i ，則我們可以使用以下的 parent、left 及 right 操作，以 i 為參數以存取堆疊的父節點 (parent)、左子節點 (left) 及右子節點 (right)：

- $\text{parent}(i) : \lfloor i/2 \rfloor$

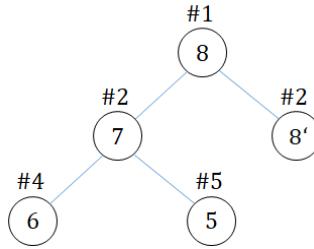


圖 3.4: 一個具有 5 個節點的最大堆積
(# 加數字代表節點編號，圓圈中的數字代表節點的數值)。

| #1 | #2 | #3 | #4 | #5 |
|----|----|----|----|----|
| 8 | 7 | 8' | 6 | 5 |

圖 3.5: 以陣列表示堆疊

- $\text{left}(i) : 2i$
- $\text{right}(i) : 2i + 1$

給定一棵二元樹及一個索引值 i ，若節點 i 的左子樹及右子樹已經是最大堆疊，但是節點 i 的值卻比其左子節點的值及/或右子節點的值小，則我們可以透過以下最大堆疊化 (max_heapify) 演算法的調整使得以節點 i 為根節點的子樹也是最大堆疊。

Algorithm max_heapify (A, i, n) ▷ 最大堆疊化演算法

Input:

A : 一個具 n 個數值且索引值由 1 開始到 n 為止的陣列，用以表示堆疊

i : 某個節點在陣列 A 中的索引值，而該節點 i 的左子樹及右子樹都是最大堆疊

Output: 陣列 A ，其中以節點 i 為樹根的子樹是最大堆疊

```

1: while  $2i \leq n$  do ▷ 確保左子節點索引值  $2i$  小於堆疊最後索引值  $n$ 
2:   if  $2i + 1 \leq n$  and  $A[2i] \leq A[2i + 1]$  then ▷ 找出節點  $i$  左或右子節點中具較大值的子節點  $j$ 
3:      $j \leftarrow 2i + 1$  ▷ 右子節點存在且其值較大，設定  $j$  為右子節點索引值
4:   else
5:      $j \leftarrow 2i + 1$  ▷ 反之則設定  $j$  為左子節點索引值
6:   if  $A[i] \geq A[j]$  then ▷ 判斷節點  $i$  的值是否比左及右子節點的值大
7:     break ▷ 節點  $i$  的值比左及右子節點的值大，跳出迴圈，最大堆疊化調整完畢
8:   else
9:     swap(  $A[i], A[j]$  ) ▷ 反之則對調節點  $i$  與節點  $j$  的值
10:     $i \leftarrow j$  ▷ 將索引值  $j$  存入  $i$  之中，回到迴圈開頭繼續進行最大堆疊化調整
11: return  $A$  ▷ 最大堆疊化調整結束，傳回陣列  $A$ 

```

最大堆疊化 (max_heapify) 演算法的基本概念為將節點 i 與其左右子樹根節點中值較大的節點對調，然後再依同樣的方式去調整因為對調而變成跟節點可能不再是最大值得子樹。這個節點與左右子樹根節點對調的過程會持續進行到對調後的子樹以是最大堆疊或是節點以對調到葉節點為止。

以下的堆積排序 (heapsort) 演算法利用最大堆疊化 (max_heapify) 演算法，可以將陣列 A 中的 n 個數值依由小到大的順序排列。

Algorithm heapsort(A, n)

▷ 堆積排序演算法

Input: A : 一個具 n 個數值且索引值由 1 開始編號到 n 為止的陣列

Output: A : 一個具 n 個由小到大依序排列數值且索引值由 1 開始編號到 n 為止的陣列

- 1: **for** $i \leftarrow \lfloor n/2 \rfloor$ to 1 **do** ▷ 先針對所有的內部節點 i ($\lfloor n/2 \rfloor \geq i \geq 1$)
 - 2: max_heapify(A, i, n) ▷ 將樹根編號為 i 最後點編號為 n 的樹進行最大堆疊化
 - 3: **for** $i \leftarrow n$ downto 2 **do** ▷ 再針對所有的內部節點 i ($n \geq i \geq 2$)
 - 4: swap($A[1], A[i]$) ▷ 將節點 i 的值與樹根節點 (編號為 1) 的值對調
 - 5: max_heapify($A, 1, i - 1$) ▷ 將樹根編號為 1 最後點編號為 $i - 1$ 的樹進行最大堆疊化
調整
 - 6: **return** A ▷ 排序結束，傳回陣列 A
-

堆積排序 (heapsort) 演算法的基本概念為：先由最後一個內節點 (索引值為 $\lfloor n/2 \rfloor$) 到樹根節點為止呼叫 max_heap 演算法，先將整棵樹建構成一顆最大堆疊，此時樹根節點一定是最大值。然後將樹根節點與最後一個節點對調，隔開最後一個節點後呼叫 max_heapify 在樹根節點再建構一顆最大堆疊，此時樹根節點是第二大值。重複相同的動作，將樹根節點與倒數第二個節點對調，隔開倒數第二個節點之後的節點後在樹根節點再建構一顆最大堆疊，...。如此的程序持續進行，陸續找出第 1 大、第 2 大、...、第 $n - 1$ 大的元素，對調倒數第 n 、第 $n - 1$ 、...、第 2 個的位置隔離後，再呼叫 max_heapify 演算法，則可將整個陣列的數值由小到大排列。

我們舉以下的實例來看 heapsort 演算法的運作情形：

給定具有 5 個元素 8、5、8'、6、7，而其索引值為 1,...,5 的陣列 A ，堆積排序 (heapsort) 演算法的執行過程如圖3.6所示。

圖3.6(a) 為陣列原始值。圖3.6(b) 為將整棵樹建構成一個最大堆疊，此時樹根節點具有最大值。圖3.6(c) 為將樹根節點的值 8 與最後節點的值 5 對調並隔離 (以淺灰底色節點表示)，然後再進行最大堆疊建構，此時找出第二大的值。圖3.6(d) 找出第 3 大的值。圖3.6(e) 找出第 4 大的值。圖3.6(f) 將第 4 大的值與索引值倒數第 4(也就是索引值 2) 的節點對調，完成排序。

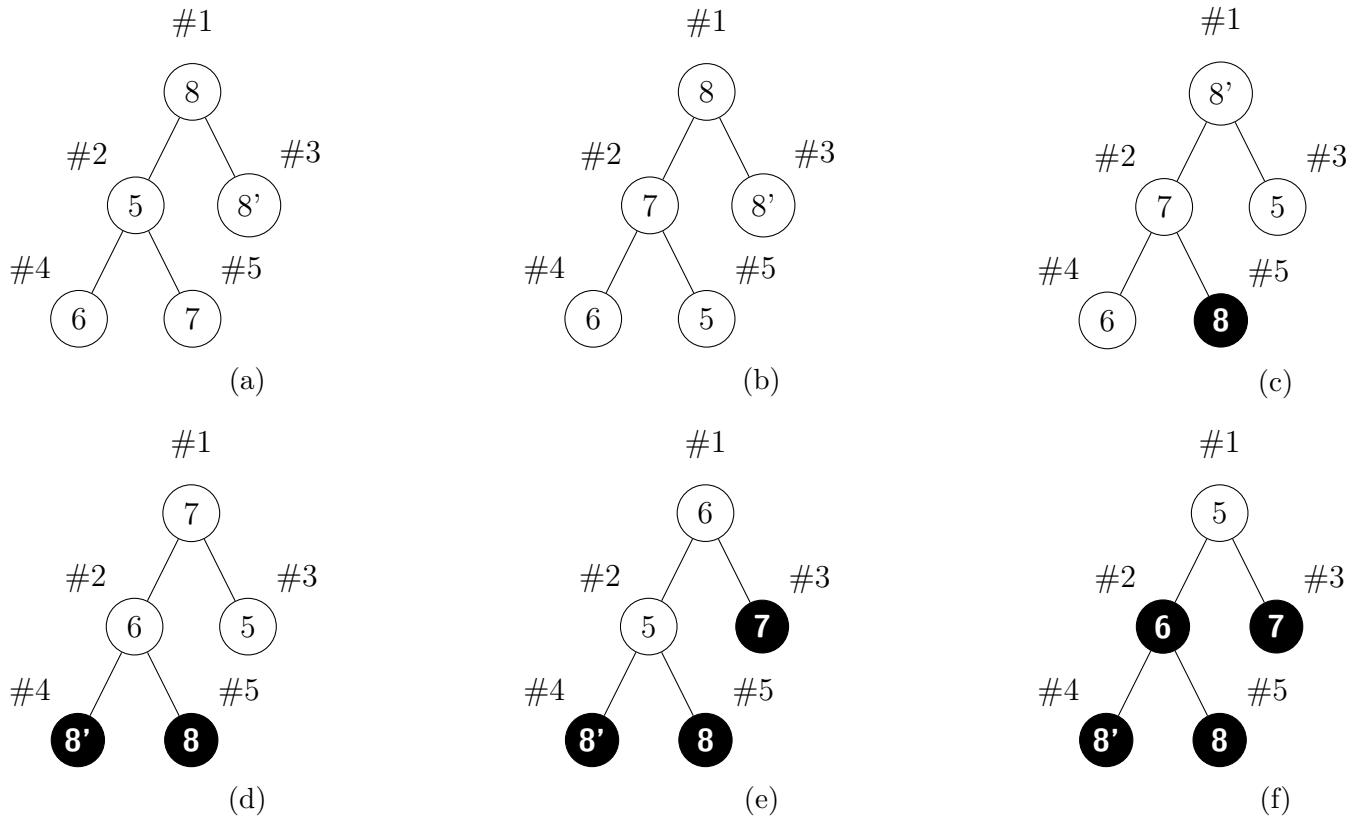


圖 3.6: 堆疊排序 (heapsort) 演算法執行範例。

因為堆疊排序演算法只使用到幾個額外的變數，因此它是就地演算法，它的空間複雜度為 $O(1)$ 。另外，堆疊排序演算法不是穩定排序演算法，因為經過堆疊排序演算法排序之後的數值中，相同的元素可能不會依照原來的順序排列。

以下我們分析堆疊排序演算法的時間複雜度。第 1 及第 2 行由最後的內容節點開始直到跟節點為止，呼叫 `max_heapify` 演算法以建立第一個堆疊。由於 `max_heapify` 演算法的執行步驟樹正比於樹高，而樹高則為 $O(n \log n)$ ，用此建立第一個堆疊需要 $O(n \log n)$ 時間複雜度。另外，第 3 到第 5 行執行 $n - 1$ 次使根節點與尚未排序好的最後節點互調及呼叫 `max_heapify` 演算法在尚未排序好的節點中再建出最大堆疊的動作。同樣的，`max_heapify` 演算法的複雜度正比於樹高而為 $O(\log n)$ ，因此，第 3 第 5 行的執行步驟數為 $O(n \log n)$ 。以上的時間複雜度適用於最佳、平均及最差狀況。因此，綜合的說，堆疊排序演算法的時間複雜度不管在最佳、平均或最差狀況下，都是 $O(n \log n)$ 。

以下，我們將氣泡排序演算法、插入排序演算法以及堆積排序演算法的各項特色比較整理如下：

| 排序演算法 | 最佳狀況 時間複雜度 | 平均狀況 時間複雜度 | 最差狀況 時間複雜度 | 空間複雜度 | 是否 穩定 | 是否 就地 |
|-------|---------------|---------------|---------------|--------|----------|----------|
| 氣泡排序 | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | 是 | 是 |
| 插入排序 | $O(n)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | 是 | 是 |
| 堆積排序 | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ | $O(1)$ | 是 | 是 |

3.7 結語

徐志摩在日記《西湖記》中說：「數大便是美」，因為「數大了似乎按照著一種自然律，自然的會有一種特別的排列，一種特別的節奏，一種特殊的式樣，激動我們審美的本能，激發

我們審美的情緒」。在演算法中，我們可以說：「數大便知甚麼是美」，因為只有在演算法的輸入規模非常大的時候，演算法的效能表現才會明顯地不同。此時，複雜度低的演算法的效能就會非常明顯地超越複雜度高的演算法。

在本章中，我們透過大 O 趨近記號分析費伯納西數列演算法、遞迴費伯納西數列演算法、氣泡排序、插入排序演算法與堆積排序演算法的複雜度。本章並說明多項式時間複雜度演算法與指數時間複雜度演算法的分類。透過本章的學習，讀者將來就可以透過同樣的方式分析演算法的複雜度，並可儘量避開複雜度高的演算法，尤其是指數複雜度演算法。

Chapter 4

分治演算法

— 凡治眾如治寡，分數是也

Contents

| | |
|---------------------------|----|
| 4.1 分治演算法基本概念 | 42 |
| 4.2 缺陷棋盤填滿演算法 | 43 |
| 4.3 合併排序演算法 | 46 |
| 4.4 快速排序演算法 | 48 |
| 4.5 最大連續子序列和演算法 | 52 |
| 4.6 快速傅立葉變換演算法 | 55 |
| 4.7 結語 | 59 |

4.1 分治演算法基本概念

分治 (divide and conquer) 演算法使用分治解題策略解決問題。分治解題策略將難以解決的大問題分割為容易解決的小問題而一一克服，是很好的解題策略，可以很有效率的解決問題，又稱為分割再克服策略或各個擊破策略。

一般而言，分治演算法具有三個階段：

- 分割 (dividing) 階段：如果問題很小，直接將此問題解決；否則，將原本的問題分割 (divide) 成 2 個或多個子問題 (subproblem)。
- 克服 (conquer) 階段：用相同的演算法遞迴地 (recursively) 解決或克服 (conquer) 所有的子問題。
- 合併 (merge) 階段：合併所有子問題的解答成為原本問題的解答。

本章將介紹一些使用分治策略解決問題的分治演算法，包括缺陷的西洋棋盤填滿演算法、合併排序演算法、快速排序演算法、最大連續子序列和演算法及快速傅立葉變換演算法。

4.2 缺陷棋盤填滿演算法

缺陷棋盤填滿演算法使用分治策略解決缺陷棋盤填滿問題，使用三格骨牌填滿缺陷棋盤。以下我們先定義甚麼是棋盤、缺陷棋盤及三格骨牌然後我們定義缺陷棋盤填滿問題，最後我們介紹缺陷棋盤填滿演算法。

- 棋盤的定義：一個棋盤是 $n \times n$ 方格 (grid)，具有 n^2 個單格 (cell)，其中 $n \geq 2$ 而且 n 是 2 的幂 (a power a 2)。圖4.1顯示 2×2 , 4×4 與 8×8 棋盤。
- 缺陷棋盤是一單格 (cell) 無法使用的棋盤。圖4.2顯示 2×2 , 4×4 與 8×8 缺陷棋盤。
- 三格骨牌 (Triomino) 為一 L 型骨牌，可填滿一棋盤上的 3 個單格。三格骨牌有 4 種方向，如圖4.3所示。

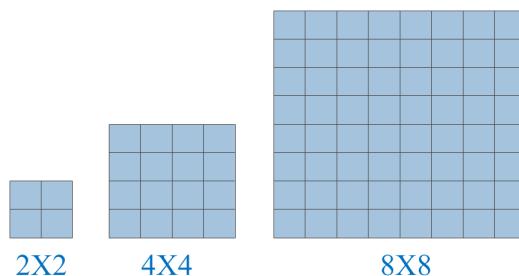


圖 4.1: 2×2 , 4×4 與 8×8 棋盤

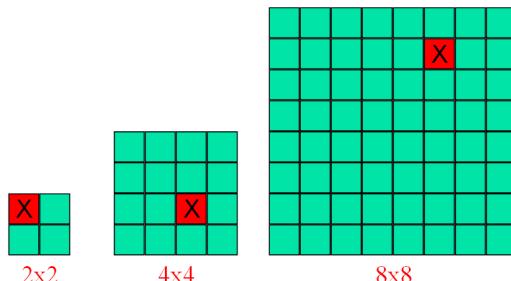


圖 4.2: 2×2 , 4×4 與 8×8 缺陷棋盤。

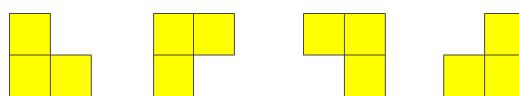


圖 4.3: 三格骨牌的四種方向

以下我們定義缺陷棋盤填滿問題：

缺陷棋盤填滿問題：如何放置 $\frac{(n^2-1)}{3}$ 個三格骨牌在 $n \times n$ 缺陷棋盤上，使得全部 (n^2-1) 個非缺陷單格都被填滿。

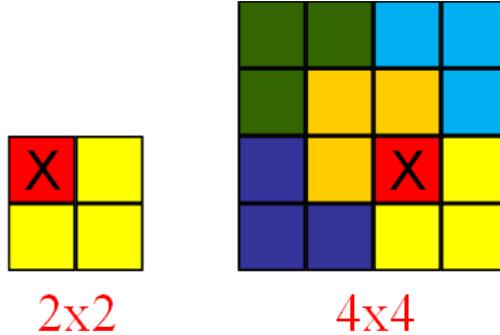


圖 4.4: 2×2 與 4×4 缺陷棋盤填滿問題解答

針對缺陷棋盤填滿問題，圖4.4顯示 2×2 與 4×4 缺陷棋盤填滿問題解答。這個解答可以採用分治策略簡單獲得，以下我們介紹使用分治解題策略解決缺陷棋盤填滿問題的缺陷棋盤填滿演算法。這個演算法主要以自然語言描述，因此我們使用步驟 1、步驟 2 等來區分演算法的步驟。

Algorithm 缺陷棋盤填滿演算法

Input: $n \times n$ 缺陷棋盤， $n \geq 2$ 而且 n 是 2 的幕

Output: 以三格骨牌填滿的 $n \times n$ 缺陷棋盤

步驟 1:

若 $n = 2$ ，則旋轉 1 個三格骨牌直接填滿缺陷棋盤，回傳此 2×2 缺陷棋盤並結束。

步驟 2:

將缺陷棋盤分為 3 個 $(n/2) \times (n/2)$ 棋盤及 1 個 $(n/2) \times (n/2)$ 缺陷棋盤，旋轉 1 個三格骨牌填滿 3 個棋盤中相鄰的單格，可使 3 個棋盤成為缺陷棋盤，我們可得 4 個 $(n/2) \times (n/2)$ 缺陷棋盤。

步驟 3:

遞迴地使用缺陷棋盤填滿演算法以三格骨牌填滿步驟 2 的 4 個 $(n/2) \times (n/2)$ 缺陷棋盤，回傳原始 $n \times n$ 缺陷棋盤並結束。

以下我們舉實例說明缺陷棋盤填滿演算法。其過程如圖4.5、4.6及4.7所示，說明如下：

- 將 16×16 棋盤分割成 4 個更小的 4×4 棋盤。其中 1 個 4×4 缺陷棋盤，其他 3 個為 4×4 一般棋盤（請見圖4.5）。
- 在 3 個一般棋盤的相鄰角落放置 1 個三格骨牌，讓這 3 個棋盤也變成 4×4 缺陷棋盤（請見圖4.6）。
- 再以遞迴方式填滿 4 個 4×4 缺陷棋盤以解決問題（請見圖4.7）。

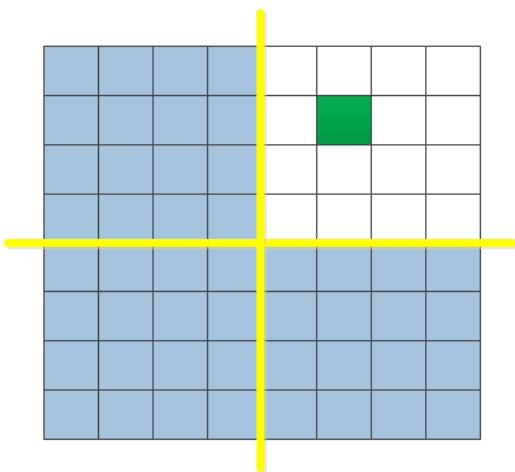


圖 4.5: 將棋盤分割成 4 個更小的棋盤。

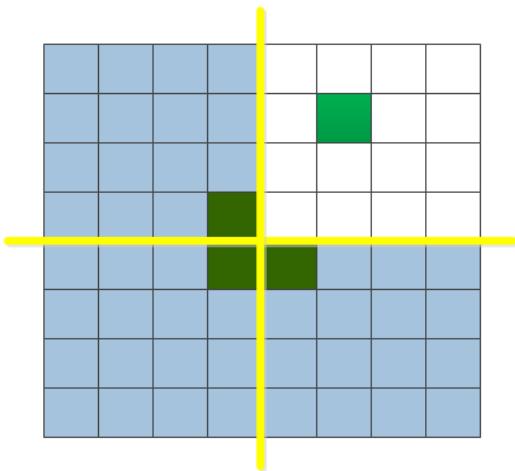


圖 4.6: 在 3 個相鄰正常棋盤角落放置 1 個三格骨牌

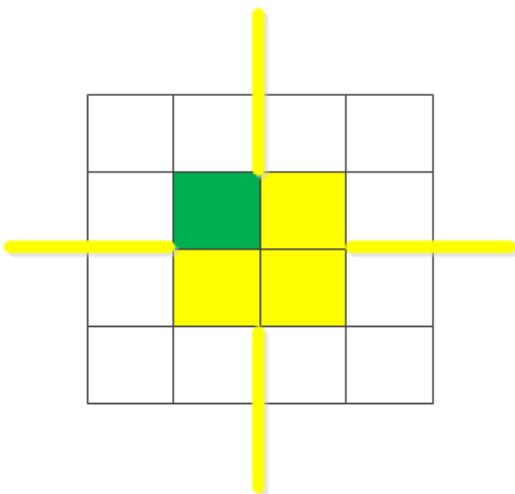


圖 4.7: 以遞迴方式填滿較小的缺陷棋盤

4.3 合併排序演算法

本節介紹使用分治解題策略的合併排序 (merge sort) 演算法。這個演算法由現代電腦之父，內儲程式 (stored program) 電腦架構發明之人之一的紐曼博士 (John von Neumann, 1903-1957)，在西元 1945 年發明。

以下為合併排序演算法使用分治解題策略的概要說明：

假設我們要使用合併排序演算法來將陣列 A 中的 n 個元素或資料 (索引為 $0, \dots, n-1$)，依照其值以由小而大的次序排列，合併排序演算法可依分治策略執行：

- 分割：若陣列 A 只有一個元素，代表陣列已排序完成；否則將陣列分割成兩個大小大約相等的子陣列。
- 克服：遞迴地排序兩個子陣列。
- 合併：最後合併兩個已完成排序的子陣列，即可完成原來陣列的排序。

以下為合併排序演算法 (MergeSort)，而在此演算法中另外使用到如合併演算法 (Merge) 以合併二個已依序排好順序的子陣列，如下所示。

Algorithm MergeSort(A, p, r)

Input: 待排序陣列 A ，與指出排序範圍的索引 p 與 r ($p \leq r$)

Output: 陣列 A ，其中索引 p 至 r 的元素已依由小而大順序排列

- 1: **if** $p = r$ **then return** A ▷ 只有一個元素無須排序
- 2: $q \leftarrow \frac{p+r}{2}$ ▷ 將陣列分割成大小幾乎相同的二個子陣列
- 3: MergeSort(A, p, q) ▷ 遞迴地排序第一個子陣列
- 4: MergeSort($A, q + 1, r$) ▷ 遞迴地排序第二個子陣列
- 5: Merge(A, p, q, r) ▷ 合併兩個已排序子序列
- 6: **return** A ▷ 傳回在索引 p 至 r 元素已依由小而大順序排列的陣列 A

Algorithm Merge(A, p, q, r)

Input: 數值陣列 A · 其中索引 p 至索引 q 之元素已依由小而大順序排列 · 而且索引 $q + 1$ 至索引 r 之元素已依由小而大順序排列

Output: 陣列 A · 其中索引 p 至索引 r 之元素已依由小而大順序排好

```
1:  $n_1 \leftarrow q - p + 1$ 
2:  $n_2 \leftarrow r - q$ 
3: for  $i \leftarrow 1$  to  $n_1$  do                                ▷ 複製  $A[p..q]$  到暫存陣列  $L[1..n_1]$ 
4:    $L[i] \leftarrow A[p + i - 1]$ 
5: for  $j \leftarrow 1$  to  $n_2$  do                                ▷ 複製  $A[q + 1..r]$  到暫存陣列  $R[1..n_2]$ 
6:    $R[j] \leftarrow A[q + j]$ 
7:  $L[n_1 + 1] \leftarrow \infty$                                 ▷ 以人為方式創造兩個終點標記使其不會被複製到  $A$ 
8:  $R[n_2 + 1] \leftarrow \infty$ 
9:  $i \leftarrow 1$ 
10:  $j \leftarrow 1$ 
11: for  $k \leftarrow p$  to  $r$  do                                ▷ 重複地從  $L$  與  $R$  中複製較小的元素到  $A$ 
12:   if  $L[i] \leq R[j]$  then
13:      $A[k] \leftarrow L[i]$ 
14:      $i \leftarrow i + 1$ 
15:   else
16:      $A[k] \leftarrow R[j]$ 
17:      $j \leftarrow j + 1$ 
18: return  $A$ 
```

以下我們舉實例來看合併排序演算法的運作過程。假設有一個陣列具有 8 個元素 85、24、63、50、17、31、96、50'，索引 (index) 為 0,...,7，其中 50 與 50' 二個元素的值都是 50，但是為了區別起見，我們將之標示為 50 與 50'。此陣列經由合併排序演算法排序的過程如圖4.8所示。

在整個合併排序演算法的執行過程中，50 與 50' 的相對位置一直保持不變，因此如同氣泡排序與插入排序演算法一樣，合併排序演算法也是一個穩定 (stable) 排序演算法。

合併排序演算法需要額外的與原來陣列大小相同的記憶體空間來輔助排序的進行，因此合併排序演算法不是就地 (in place) 演算法，它的空間複雜度為 $O(n)$ ， n 為需要排序陣列的元素個數。

以下我們分析合併排序演算法的時間複雜度。假設合併排序演算法將一個具有 n 個元素的陣列排序完畢的時間複雜度為 $T(n)$ ，則我們可以得到以下的式子：

$$T(n) = 2T(n/2) + n$$

這是因為合併排序演算法直接將陣列分割為二個大小相同或幾乎相同的陣列 (大小為或約為原來陣列大小 $1/2$)，並分別遞迴地以相同的演算法將二個子陣列加以排序 (因此有 $2T(n/2)$ 項目)，最後，再使用 n 次比較操作將二個子陣列合併。

我們遞迴地將 $T(n) = 2T(n/2) + n$ 中的 n 取代為 $n/2, n/4, \dots$ ，則我們可以得到：

$$\begin{aligned} T(n) &= 2T(n/2) + n = 2(2T(n/4) + (n/2)) + n = 4T(n/4) + n + n \\ &= 4(2T(n/8) + (n/4)) + 2n = 8T(n/8) + 3n = \dots = 2^kT(n/2^k) + kn \end{aligned}$$

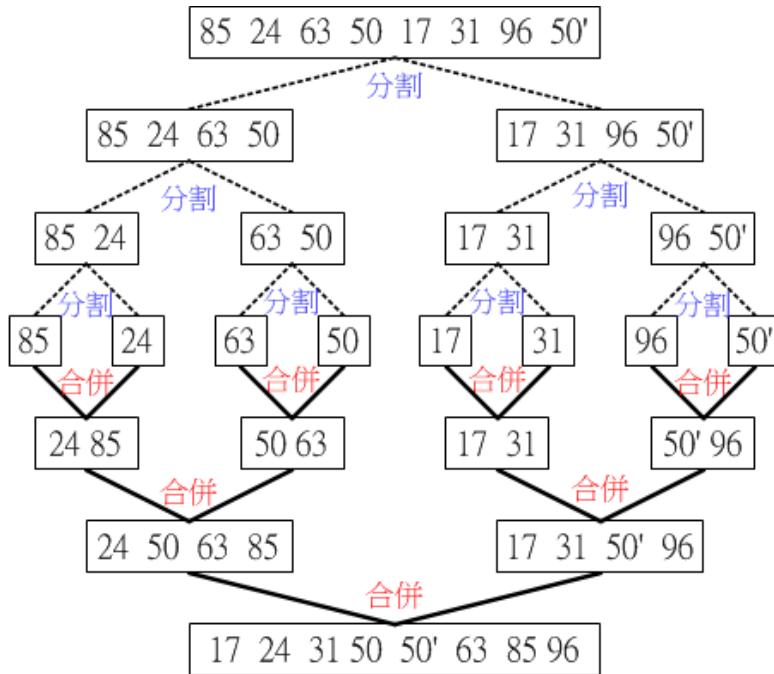


圖 4.8: 合併排序演算法排序過程

當陣列只有一個元素時，合併排序演算法不會再進行遞迴呼叫，會直接回轉 (return)，因此我們可得 $T(1) = 1$ 。假設 $n = 2^k$ ，則 $k = \log_2 n$ 。請注意，未來若我們不特別指定 \log 函數的基底，則代表基底為 2。代入 $k = \log n$ ，我們可得：

$$T(n) = n \log n + 2^k = n \log n + n = O(n \log n)$$

對所有的狀況 (最佳、最差與平均狀況) 而言，合併排序演算法的時間複雜度都是 $O(n \log n)$ 。

4.4 快速排序演算法

本節介紹使用分治解題策略的快速排序 (quick sort) 演算法。此演算法由獲得計算機領域最高榮譽圖靈獎 (Turing Award) 的霍爾博士 (Charles Antony Richard Hoare，縮寫為 C. A. R. Hoare，1934 年 1 月 11 日生) 於 1962 年發表 [4]。如其名稱所示，此排序演算法的排序速度相當快，因此使用相當廣泛。以下為快速排序演算法的概要說明：

假設我們要使用快速排序演算法來將陣列 A 中的 n 個元素或資料 (索引為 $0, \dots, n-1$)，將陣列中索引在「左界」 lb 及「右界」 rb 範圍內的元素依照其值以由小而大的次序排列，快速排序演算法可依分治策略執行：

- 分割：若陣列 A 只有一個元素，代表陣列已排序完成；否則將陣列分割成兩個大小大約相等的子陣列。作法為選一個元素 p 當作中樞 (pivot) 元素，將陣列分為二個分割 (partition)：SP 及 LP，其中 SP (smaller part) 包含所有小於或等於 p 的元素，而 LP (larger part) 則包含所有大於 p 的元素。
- 克服：遞迴地進行 SP 部份與 LP 部份的元素排序。
- 合併：最後再將 SP、 p 及 LP 合併即可完成排序。

以下為快速排序演算法虛擬碼，此演算法使用二個指標 (「左指標」 l 與「右指標」 r) 將陣列中索引在「左界」 lb 及「右界」 rb 範圍內的元素，依照中樞元素 $p = A[rb]$ 分割為二個分

割：SP 及 LP，其中 SP 包含所有小於或等於 p 的元素，而 LP 則包含所有大於 p 的元素；調換元素位置使得 SP 在左， p 在中間，LP 在右。快速排序演算接續著以遞迴方式進行 SP 部份與 LP 部份的元素排序。因為 SP 在左， p 在中間，LP 在右，因此陣列中索引在「左界」 lb 及「右界」 rb 範圍內的元素即已依照由小而大的次序排列。

Algorithm QuickSort(A, lb, rb)

Input: 數值陣列 A ，其中需要排序的第一個（左界）元素索引為 lb ，需要排序的最後一個（右界）元素索引為 rb

Output: 陣列 A ，其中由索引 lb 到索引 rb 的元素由小到大排列

```

1: if  $lb \geq rb$  then return  $A$ 
2:  $p \leftarrow A[rb]$                                  $\triangleright p$  為中樞 (pivot) 元素
3:  $l \leftarrow lb$                                   $\triangleright l$  為左指標
4:  $r \leftarrow rb - 1$                              $\triangleright r$  為右指標
5: while true do
6:   右移  $l$  直到碰到或大於或等於  $p$  的元素
7:   左移  $r$  直到碰到小於  $p$  的元素或  $r$  等於 (到達) $lb$ 
8:   if  $l < r$  then                                 $\triangleright l < r$  成立表示分割尚未完成
9:     對調  $A[l]$  與  $A[r]$ 
10:    else break while-loop                   $\triangleright l < r$  不成立表示分割已完成，因此離開 while 迴圈
11:  對調  $A[rb]$  與  $A[l]$ 
12:  QUICKSORT( $A, lb, l - 1$ )
13:  QUICKSORT( $A, l + 1, rb$ )
14: return  $A$ 

```

以下我們舉實例來看快速排序演算法的運作過程。假設有一個陣列具有 8 個元素 85、24、63、50、17、50'、96、58，索引 (index) 為 0,...,7，其中 50 與 50' 二個元素的值都是 50，但是為了區別起見，我們將之標示為 50 與 50'。圖4.9展示快速排序演算法第一次將陣列分割為二個部份的過程。

| lb | | | | | | | rb |
|----|----|----|-----|----|----|----|------------------------------|
| 85 | 24 | 63 | 50 | 17 | 31 | 96 | 50' |
| 1 | | | | | | r | |
| 85 | 24 | 63 | 50 | 17 | 31 | 96 | 50' |
| 1 | | | | | r | | 將 1 及 r 所指之元素 (85, 31) 對調 |
| 31 | 24 | 63 | 50 | 17 | 85 | 96 | 50' |
| | 1 | | | r | | | 將 1 及 r 所指之元素 (63, 17) 對調 |
| 31 | 24 | 17 | 50 | 63 | 85 | 96 | 50' |
| r | | 1 | | | | | (1 >= r) 表示切割完成 |
| 31 | 24 | 17 | 50' | 63 | 85 | 96 | 50 |
| r | | 1 | | | | | 將 p 調至 1 之位置，元素 (50, 50') 對調 |

圖 4.9: 快速排序演算法將陣列分割為二個部份的過程

在整個快速排序演算法的執行過程中，數值相同的元素會因為元素位置調換而產生相對

位置的變化 (例如，快速排序演算法的運作實例中 50 與 50' 的相對位置在排序後產生變化)，因此快速排序演算法不是一個穩定 (stable) 排序演算法。

以下我們分析快速排序演算法的時間複雜度。在最佳狀況下，快速排序演算法每次都將陣列分割為二個大小相同或幾乎相同的子陣列 (我們可以將分割後的二個子陣列都視為是原陣列的 $1/2$ 大小)。假設利用快速排序演算法針對具有 n 個元素的陣列 (也就是說輸入規模為 n) 進行排序的時間複雜度為 $T(n)$ ，針對最佳狀況，我們可以得到以下的式子：

$$T(n) = n + 2T(n/2)$$

這是因為當指標 l 持續往右移，而同時指標 r 持續往左移而交叉時 (也就是說 $l \leq r$ 時)，代表陣列分割完成。指標每次移動一個位置需要一次的數值比較操作，因此要完成陣列分割需要執行 n 次數值比較操作。而陣列分割完成之後，快速排序演算法就利用遞迴的方式分別完成二個大小相同的 (均為 $n/2$) 子陣列排序。

我們遞迴地將 $T(n)$ 中的輸入規模 n ，取代為 $n/2, n/4, \dots$ ，則我們可以得到：

$$\begin{aligned} T(n) &= n + 2T(n/2) = n + 2(n/2 + 2T(n/4)) = n + n + 4T(n/4) \\ &= 2n + 4(n/4 + 2T(n/8)) = 3n + 8T(n/8) = \dots = kn + 2^k T(n/2^k) \end{aligned}$$

當陣列只有一個元素時，快速排序演算法不會再進行遞迴呼叫，會直接回轉 (return)，因此我們可得 $T(1) = 1$ 。假設 $n = 2^k$ ，則 $k = \log_2 n$ 。請注意，未來若我們不特別指定 \log 函數的基底，則代表基底為 2。代入 $k = \log n$ ，我們可得：

$$T(n) = n \log n + 2^k = n \log n + n = O(n \log n)$$

以下我們分析快速排序演算法的最差狀況時間複雜度。當陣列的 n 個元素已經依照由小而大的方式排列的情況下會產生最差狀況。在此情況下，快速排序演算法首先在經過 n 次數值比較操作之後，將陣列分割為單一一個所有元素都比中樞元素小，具有 $n - 1$ 個元素的子陣列。經過遞迴呼叫，快速排序演算法再利用 $n - 1$ 次數值比較操作將陣列分割為單一一個所有元素都比新中樞元素小，具有 $n - 2$ 個元素的子陣列。如此不斷遞迴執行，直到陣列分割出僅包含一個元素的子陣列為止。同樣假設快速排序演算法的時間複雜度為 $T(n)$ ，針對最差狀況，我們可以得到以下的式子：

$$T(n) = n + (n - 1) + (n - 2) + \dots + 1 = \frac{n(n - 1)}{2} = O(n^2)$$

以下我們分析快速排序演算法的平均狀況時間複雜度。假定陣列有 n 個元素，則在大約經過 n 次數值比對之後 (我們記為 cn ，代表不是剛好是 n ，而是 n 的線性量級)，可以將陣列分割成二個子陣列。在平均狀況下，我們討論陣列分割成二個子陣列後的各種情形 (請注意，我們將中樞元素視為包含在前一個子陣列中)：(狀況 1) 前一個子陣列有 1 個元素，後一個子陣列有 $n - 1$ 個元素；(狀況 2) 前一個子陣列有 2 個元素，後一個子陣列有 $n - 2$ 個元素；…；(狀況 n) 前一個子陣列有 n 個元素，後一個子陣列有 0 個元素。我們假設上述的狀況產生的情形都具有相同的機率，並同樣假設快速排序演算法的時間複雜度為 $T(n)$ ，則我們可以得到以下的式子：

$$\begin{aligned}
T(n) &= \frac{1}{n} \sum_{s=1}^n (T(s) + T(n-s)) + cn \\
&= \frac{1}{n} (T(1) + T(n-1) + T(2) + T(n-2) + \dots + T(n) + T(0)) + cn, T(0) = 0 \\
&= \frac{1}{n} (2T(1) + 2T(2) + \dots + 2T(n-1) + T(n)) + cn
\end{aligned}$$

移項化簡之後我們可得

$$\begin{aligned}
T(n) - \frac{1}{n} T(n) &= \frac{1}{n} (2T(1) + 2T(2) + \dots + 2T(n-1)) + cn \\
\frac{n-1}{n} T(n) &= \frac{1}{n} (2T(1) + 2T(2) + \dots + 2T(n-1)) + cn
\end{aligned}$$

等號兩邊同乘 n 之後，我們可得：

$$(n-1)T(n) = 2T(1) + 2T(2) + \dots + 2T(n-1) + cn^2 \quad (4.1)$$

將 n 以 $n-1$ 代入後我們可得：

$$(n-2)T(n-1) = 2T(1) + 2T(2) + \dots + 2T(n-2) + c(n-1)^2 \quad (4.2)$$

將式4.1減去式4.2，並利用 $cn^2 - c(n-1)^2 = cn^2 - c(n^2 - 2n + 1) = c(2n-1)$ 化簡可得：

$$\begin{aligned}
(n-1)T(n) - (n-2)T(n-1) &= 2T(n-1) + c(2n-1) \\
(n-1)T(n) - nT(n-1) &= c(2n-1)
\end{aligned}$$

等號兩邊同除 $n(n-1)$ 並利用 $\frac{2n-1}{n(n-1)} = \frac{1}{n} + \frac{1}{n-1}$ 化簡後可得：

$$\frac{T(n)}{n} = \frac{T(n-1)}{n-1} + c\left(\frac{1}{n} + \frac{1}{n-1}\right) \quad (4.3)$$

持續地以 $n-1, n-2, \dots, 1$ 等數值代入式4.3中的 n ，我們可得：

$$\begin{aligned}
\frac{T(n)}{n} &= c\left(\frac{1}{n} + \frac{1}{n-1}\right) + c\left(\frac{1}{n-1} + \frac{1}{n-2}\right) + \dots + c\left(\frac{1}{2} + 1\right) + T(1), T(1) = 0 \\
&= c\left(\frac{1}{n} + \frac{1}{n-1} + \dots + \frac{1}{2}\right) + c\left(\frac{1}{n-1} + \frac{1}{n-2} + \dots + 1\right)
\end{aligned} \quad (4.4)$$

以下我們利用調和數 (harmonic number) 來化簡式 (4.4)。調和數 H_n 定義為 $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{3} + \dots + \frac{1}{n}$ 。當 n 足夠大時， H_n 趨近於 $\ln n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + \frac{1}{120n^4} - \varepsilon$ ，其中 $0 < \varepsilon < \frac{1}{252n^6}$ ，而且 $\gamma = 0.5772156649$ 。根據大 O 記號的定義，我們可得 $H_n = c \log n = O(\log n)$ 。

因此，由式4.4與 H_n 的定義我們可得：

$$\frac{T(n)}{n} = c(H_n - 1) + cH_{n-1} = c(2H_n - \frac{1}{n} - 1)$$

最後我們可求得：

$$T(n) = 2cnH_n - c(n+1) = O(n \log n)$$

也就是說，快速排序演算法的平均狀況時間複雜度為 $O(n \log n)$ 。

以下我們分析快速排序演算法的空間複雜度。快速排序演算法使用遞迴呼叫，因此需要使用額外的堆疊記憶體空間來輔助排序的進行，因此不是一個就地 (in place) 演算法，它的空間複雜度為相依於最長的遞迴呼叫。在最佳狀況下，快速排序演算法每次都將陣列分為平均分割為二部分，如此只要經過 $\log n$ 次遞迴呼叫就可將陣列分割到只剩下一個元，因此空間複雜度為 $O(\log n)$ 。但是在最壞情況下，快速排序演算法需要進行 $n - 1$ 次遞迴呼叫，因此最壞狀況時間複雜度為 $O(n)$ 。

以下，我們將快速排序與之前已介紹過的合併排序演算法、氣泡排序與插入排序演算法的各項特色比較整理如下：

| 排序演算法 | 最佳狀況 時間複雜度 | 平均狀況 時間複雜度 | 最差狀況 時間複雜度 | 空間複雜度 | 是否 穩定 | 是否 就地 |
|-------|---------------|---------------|---------------|---|----------|----------|
| 氣泡排序 | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | 是 | 是 |
| 插入排序 | $O(n)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | 是 | 是 |
| 堆積排序 | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ | $O(1)$ | 是 | 是 |
| 合併排序 | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ | $O(n)$ | 是 | 否 |
| 快速排序 | $O(n \log n)$ | $O(n \log n)$ | $O(n^2)$ | $O(\log n)$ (最佳狀況) $O(n)$ (最差狀況) | 否 | 否 |

4.5 最大連續子序列和演算法

在本節中我們介紹使用分治策略解決最大連續子序列和 (Maximum Contiguous Subsequence Sum, MCSS) 問題的最大連續子序列和演算法。以下，我們先描述最大連續子序列和問題：給定一個包含 n 個正或負整數的序列 $S = (s_1, s_2, \dots, s_n)$ ，找出 S 的連續非空 (non-null) 子序列，使得子序列中的元素總和最大。

例如，令 $S = (-2, 1, -3, 4, -1, 2, 1, -5, 4)$ ，則 S 的最大連續子序列和為 $4 + (-1) + 2 + 1 = 6$ 。又例如，令 $S = (-2, -6, -3, -4, -10, -5, -1, -7)$ ，則 S 的最大連續子序列和為 -1 。因為此例中序列 S 的所有元素均為負，因此最大連續子序列和等於最大的元素值 -1 ，而 -1 單一元素就是產生此和的子序列。

請注意，若子序列沒有非空限制，則當序列中所有的數均為負值時，則最大連續子序列和為 0 。例如，令 $S = (-2, -6, -3, -4, -10, -5, -1, -7)$ ，則因為此例中序列 S 的所有元素均為負，所以 S 的最大連續子序列和為 0 。

以下我們介紹兩個可以解決最大連續子序列和問題的窮舉演算法，最大連續子序列和窮舉演算法 1 與最大連續子序列和窮舉演算法 2。我們可以很容易推得他們的時間複雜度分別為 $O(n^3)$ 與 $O(n^2)$ 。但是藉由分治解答策略，我們可以設計出時間複雜度為 $O(n \log n)$ 的演算法：最大連續子序列和分治演算法 (MCSS 演算法)。這個分治演算法的基本概念為：(分割階段) 將序列分為左右兩個子序列；(克服階段) 然後分別遞迴地求出左子序列最大連續子序列和 (存於 $msml$) 及右子序列的最大連續子序列和 (存於 $msmr$)；(合併階段) 最後，求出跨越兩個子序列的最大連續子序列和 (存於 $msml + msmr$)，然後回傳 msl 、 msr 、及 $msml + msmr$ 的最大值。

Algorithm 最大連續子序列和窮舉演算法 1

Input: 包含 n 個正或負整數的序列 $S = (s_1, s_2, \dots, s_n)$

Output: l, r, m · 其中非空子序列 (s_l, \dots, s_r) 可產生最大和 m

```
1:  $m \leftarrow -\infty$ 
2: for  $i \leftarrow 1$  to  $n$  do
3:   for  $j \leftarrow i$  to  $n$  do                                ▷ 以上二迴圈產生所有可能的子序列
4:      $t \leftarrow 0$ 
5:     for  $k \leftarrow i$  to  $j$  do                          ▷ 此迴圈計算所有子序列的和
6:        $t \leftarrow t + s_k$ 
7:     if  $t > m$  then
8:        $m \leftarrow t; l \leftarrow i; r \leftarrow j$ 
9: return  $l, r, m$ 
```

Algorithm 最大連續子序列和窮舉演算法 2

Input: 包含 n 個正或負整數的序列 $S = (s_1, s_2, \dots, s_n)$

Output: l, r, m · 其中非空子序列 (s_l, \dots, s_r) 可產生最大和 m

```
1:  $m \leftarrow -\infty$ 
2: for  $i \leftarrow 1$  to  $n$  do                                ▷ 此迴圈限定子序列的開頭為  $s_i$ 
3:    $t \leftarrow 0$ 
4:   for  $j \leftarrow i$  to  $n$  do                          ▷ 此迴圈限定子序列的結束為  $s_j$ 
5:      $t \leftarrow t + s_j$ 
6:     if  $t > m$  then
7:        $m \leftarrow t; l \leftarrow i; r \leftarrow j$ 
8: return  $l, r, m$ 
```

Algorithm MCSS(S, l, r)

▷ 最大連續子序列和分治演算法

Input: S : a sequence (s_1, s_2, \dots, s_n) of n elements which are negative or positive integers l : an integer indicating that the leftmost element to be considered is s_l r : an integer indicating that the rightmost element to be considered is s_r **Output:** m : the maximum contiguous subsequence sum, where the subsequence may not be empty

```
1: if  $l = r$  then return  $s_l$ 
2:  $mid \leftarrow \lfloor(l + r)/2\rfloor$                                 ▷  $mid$ : middle
3:  $msl \leftarrow \text{MCSS}(S, l, mid)$                                ▷  $msl$ : max sum for left subsequence
4:  $msr \leftarrow \text{MCSS}(S, mid + 1, r)$                          ▷  $msr$ : max sum for right subsequence
5:  $msml \leftarrow -\infty; t \leftarrow 0$                             ▷  $msml$ : max sum from middle to leftmost
6: for  $i \leftarrow mid$  downto  $l$  do
7:    $t \leftarrow t + s_i$ 
8:   if  $t > msml$  then
9:      $msml \leftarrow t$ 
10:   $msmr \leftarrow -\infty; t \leftarrow 0$                            ▷  $msmr$ : max sum from middle+1 to rightmost
11:  for  $i \leftarrow mid + 1$  to  $r$  do
12:     $t \leftarrow t + s_i$ 
13:    if  $t > msmr$  then
14:       $msmr \leftarrow t$ 
15: return  $\max(msl, msr, msml + msmr)$                       ▷  $msml + msmr$ : max sum cross the middle
   (including both mid and mid+1)
```

以下我們分析最大連續子序列和分治演算法 (MCSS 演算法) 的時間複雜度 $T(n)$ ，我們可以得到以下的式子：

$$T(n) = 2T(n/2) + cn$$

遞迴地將 $T(n) = 2T(n/2) + cn$ 中的 n 取代為 $n/2, n/4, \dots$ ，則我們可以得到：

$$\begin{aligned} T(n) &= 2T(n/2) + cn \\ &= 2(2T(n/4) + c(n/2)) + cn \\ &= 4T(n/4) + 2cn \\ &= 4(2T(n/8) + (n/4)) + 2cn \\ &= 8T(n/8) + 3cn = \dots \\ &= 2^k T(n/2^k) + kcn \end{aligned}$$

令 $n = 2^k$ ，則 $k = \log n$ ，我們可得：

$$T(n) = 2^k T(1) + cn \log n = n + cn \log n = O(n \log n) \text{ (因為 } T(1)=1\text{)}$$

因此，最大連續子序列和分治演算法的時間複雜度為 $O(n \log n)$ 。

以上的演算法可以成功的解決最大連續子序列和問題，會找出一個具有最大和的非空子序列。然而，若我們允許演算法找出空的子序列 (其對應的和為 0)。則當序列中所有的整數都是負數時，最大連續子序列和就不再是最大負數，而應該是 0。

若子序列沒有非空限制，則我們要替三個演算法進行一些修改。其中針對「最大連續子序列和窮舉演算法 1」與「最大連續子序列和窮舉演算法 2」，只要將第一行修改為 $m \leftarrow 0$ 即可。針對最大連續子序列和分治 MCSS 演算法，則將其修改為如下所列的可空最大連續子序列和分治演算法，英文名稱為 MCSSE(Maximum Contiguous Subsequence Sum allowing Empty sequence)。

Input:

S : a sequence (s_1, s_2, \dots, s_n) of n elements which are negative or positive integers

l : an integer indicating that the leftmost element to be considered is s_l

r : an integer indicating that the rightmost element to be considered is s_r

Output:

m : the maximum contiguous subsequence sum, where the subsequence may not be empty

```

1: if  $l = r$  then return  $\max(s_l, 0)$ 
2:  $mid \leftarrow \lfloor(l + r)/2\rfloor$                                 ▷  $mid$ : middle
3:  $msl \leftarrow \text{MCSS}(S, l, mid)$                                ▷  $msl$ : max sum for left subsequence
4:  $msr \leftarrow \text{MCSS}(S, mid + 1, r)$                            ▷  $msr$ : max sum for right subsequence
5:  $msml \leftarrow 0; t \leftarrow 0$                                      ▷  $msml$ : max sum from middle to leftmost
6: for  $i \leftarrow mid$  downto  $l$  do
7:    $t \leftarrow t + s_i$ 
8:   if  $t > msml$  then
9:      $msml \leftarrow t$ 
10:   $msmr \leftarrow 0; t \leftarrow 0$                                      ▷  $msmr$ : max sum from middle+1 to rightmost
11:  for  $i \leftarrow mid + 1$  to  $r$  do
12:     $t \leftarrow t + s_i$ 
13:    if  $t > msmr$  then
14:       $msmr \leftarrow t$ 
15: return  $\max(msl, msr, msml + msmr)$                                ▷  $msml + msmr$ : max sum cross the middle
                                         (including both mid and mid+1)

```

以上 MCSS2 演算法相較於 MCSS 演算法的修改只有以下 3 處:

- (1) 第 1 行的 s_l 改為 $\max(s_l, 0)$
- (2) 第 5 行的 $msml \leftarrow -\infty$ 改為 $msml \leftarrow 0$
- (3) 第 10 行的 $msmr \leftarrow -\infty$ 改為 $msmr \leftarrow 0$

4.6 快速傅立葉變換演算法

庫利 (J. W. Cooley) 和圖基 (J. W. Tukey) 在 1965 年共同發表快速傅立葉變換 (fast Fourier transform, FFT) 演算法 [5]，可以快速計算序列的離散傅立葉變換 (discrete Fourier transform, DFT) 及其逆變換，被被 IEEE 科學與工程計算 (Computing in Science Engineering) 期刊 [6] 譽為 20 世紀十大演算法之一。這個演算法源自於法國數學家及物理學家傅立葉 (Joseph Fourier, 1768-1830)。傅立葉熱中於熱傳導的研究，提出傅立葉變換 (Fourier transform) 概念描述如何使用無窮的、具一樣相位 (phase) 的、頻率 (frequency) 為倍數增加的正弦函數及餘弦函數的組合來表示熱能從高溫向低溫部份轉移過程的溫度分佈，而這概念可以延伸用於將函數變換為正弦函數及餘弦函數組合。請注意，上述無窮的、具相同相位的、頻率為倍數增加的正弦函數及餘弦函數的組合，其實可以單獨使用無窮的、具不同相位的、頻率為倍數增加的正弦函數組合來取代。

以下使用傅立葉級數 (Fourier series) 來說明傅立葉變換的原理。令 $f(t)$ 表示實數變數 t 的連續函數，且 f 在 $[t_0, t_0+T]$ 上可積，其中 t_0 和 T 為實數。傅立葉級數是一個無窮級數，使用無窮多個週期 (period) 分別為 $T, T/2, T/3, T/4, \dots$ 的正弦函數及餘弦函數來近似

$[t_0, t_0+T]$ 區間內的 f 函數。另外，若在 $[t_0, t_0+T]$ 區間外， f 也能夠以該級數表示，則級數對 f 的近似在整個實數線上都有效。

上述的無窮多個函數形成所謂的諧波 (harmonic)，其中 T 為基本周期，而週期為 T 的函數為基波 (fundamental wave) 或稱為一次諧波 (first order harmonic)。另外，週期為 $T/2, T/3, \dots, T/k, \dots$ 的函數統稱為高次諧波 (high order harmonic)，它們分別稱為二次諧波 (second order harmonic)、三次諧波 (third order harmonic)、…、 k 次諧波 (k^{th} order harmonic)、…。基波的頻率稱為基頻 (fundamental frequency)，其值為 $1/T$ ， k 次諧波的頻率為基頻的 k 倍，其值為 k/T 。

具體的說，相依於實數變數 t 的連續函數 $f(t)$ ，可以表示為以下的傅立葉級數：

$$f(t) = a_0 + \sum_{k=1}^{\infty} (a_k \cos k\omega t + b_k \sin k\omega t) \quad (4.5)$$

也就是

$$f(t) = a_0 + \sum_{k=1}^{\infty} \left(a_k \cos k \frac{2\pi}{T} t + b_k \sin k \frac{2\pi}{T} t \right) \quad (4.6)$$

在式 (4.5) 及式 (4.6) 中， T 為基本週期，也就是基波的周期，而 ω 為基波的角速度，單位為強度/單位時間。以強度來看，正弦及餘弦函數週期為 2π 強度，因此 $\omega = \frac{2\pi}{T}$ ，而 $k\omega = k \frac{2\pi}{T}$ 則是 k 次諧波的角速度。

另外，我們有以下傅立葉級數係數 a_0 、 a_k 及 $b_k, k = 1, 2, \dots$ 的公式：

$$a_0 = \frac{1}{T} \int_{t_0}^{t_0+T} f(t) dt \quad (4.7)$$

$$a_k = \frac{2}{T} \int_{t_0}^{t_0+T} f(t) \cos k\omega t dt \quad (4.8)$$

$$b_k = \frac{2}{T} \int_{t_0}^{t_0+T} f(t) \sin k\omega t dt \quad (4.9)$$

依照歐拉公式

$$e^{i\theta} = \cos \theta + i \sin \theta \quad (4.10)$$

$$e^{-i\theta} = \cos \theta - i \sin \theta \quad (4.11)$$

式 (4.10) 及式 (4.11) 中， $i = \sqrt{-1}$ 為虛數單位， e 為自然對數底數， θ 為角度 (單位為強度)。因為角速度乘以時間可以得到角度，因此我們可得 $\theta = k\omega t$ 。由式 (4.8)、(4.9)、(4.10) 及 (4.11)，我們可以整理出以下的複數型式傅立葉級數：

$$f(t) = \sum_{n=0}^{\infty} c_n e^{ink\omega t} \quad (4.12)$$

其中 $c_k, k = 1, 2, \dots$ 依照如下的公式計算：

$$c_k = \frac{2}{T} \int_{t_0}^{t_0+T} f(t) e^{-ik\omega t} dt \quad (4.13)$$

給定一個連續函數 $f(t)$ 是，依照式 (4.7)、(4.8) 及 (4.9) 分別求出傅立葉級數係數 $a_0, a_k, b_k, k =$

1, 2, ... 或是依照式 (4.13) 求出傅立葉級數係數 $c_k, k = 0, 1, \dots$ 就是進行函數 $f(t)$ 的傅立葉變換，可以求出相依於時間變數 t 的時域 (time domain) 函數 $f(t)$ 的頻域訊息，也就是傅立葉級數不同頻率諧波的大小，這樣可以讓我們由頻域的觀點來檢視函數 $f(t)$ 。

因為 $f(t)$ 是一個連續函數，在計算機上我們必須先將 $f(t)$ 取樣為離散序列 (discrete series) 才可以進行計算，稱為離散傅立葉變換。一般我們只要在週期函數的一個週期中取樣超過兩次，就可以確實描述週期函數。令 $f(t)$ 取樣為離散序列 $x = x_0, x_1, \dots, x_{n-1}$ ，則以下為複數型的離散傅立葉變換

$$c_k = \sum_{j=0}^{n-1} x_j e^{ijk2\pi/n}, k = 0, 1, \dots, n \quad (4.14)$$

請注意，式 (4.13) 取消 e 的指數的負號並在推導後得到式 (4.13)，取消 e 的指數的負號只會造成角速度旋轉方向的改變，而不會影響公式的正確性。

在式 (4.13) 中， $j = 0, 1, \dots, n - 1$ 代表取樣索引值， $k = 0, 1, \dots, n - 1$ 代表諧波基頻倍數索引值，也是傅立葉級數係數 c_k 的索引值。

令 $\omega = \omega_n = e^{i2\pi/n}$ 為 1 的 n 次方主根 (principle n^{th} root of unity)，我們知道一共有 n 個 n 次方複數根 $e^{ik2\pi/n}, k = 0, 1, \dots, n - 1$ 。我們可將式 (4.14) 改寫為：

$$c_k = \sum_{j=0}^{n-1} x_j \omega^{jk}, k = 0, 1, \dots, n \quad (4.15)$$

在式 (4.14) 中，每個 $c_k (k = 0, 1, \dots, n - 1)$ 的計算需要 $O(n)$ 的時間複雜度，因此計算完所有 n 個 c_k 的時間複雜度為 $O(n^2)$ 。但是由於

$$\omega^n = (e^{i2\pi/n})^n = e^{i2\pi} = \cos 2\pi + i \sin 2\pi = 1 \quad (4.16)$$

$$\omega^{n/2} = (e^{i\pi/n})^{n/2} = e^{i\pi} = \cos \pi + i \sin \pi = -1 \quad (4.17)$$

根據式 (4.16) 及式 (4.17)，我們可以採取分治解題策略以 $O(n \log n)$ 的複雜度完成 $c_k (k = 0, 1, \dots, n - 1)$ 的計算，這就是著名的快速傅立葉轉換演算法。如前所述，這個演算法在 1965 年由庫利及圖基發表，因此又稱為酷利-圖基演算法。其實，早在 1805 年德國數學家高斯 (Friedrich Gauss, 1777-1855) 就已提出相同概念。

我們使用以下的範例來講解快速傅立葉演算法。令 x 為長度為 8 的離散序列， $x = x_0, x_1, \dots, x_7$ 我們有 $n = 8$ ， $\omega = \omega_8 = e^{i2\pi/8}$ ， $\omega^8 = 1$ ， $\omega^4 = -1$ 。以下為傅立葉級數係數 $c_k, k = 0, 1, \dots, 7$ ：

$$\begin{aligned} c_0 &= x_0 + x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 \\ c_1 &= x_0 + x_1\omega + x_2\omega^2 + x_3\omega^3 + x_4\omega^4 + x_5\omega^5 + x_6\omega^6 + x_7\omega^7 \\ c_2 &= x_0 + x_1\omega^2 + x_2\omega^4 + x_3\omega^6 + x_4\omega^8 + x_5\omega^{10} + x_6\omega^{12} + x_7\omega^{14} \\ c_3 &= x_0 + x_1\omega^3 + x_2\omega^6 + x_3\omega^9 + x_4\omega^{12} + x_5\omega^{15} + x_6\omega^{18} + x_7\omega^{21} \\ c_4 &= x_0 + x_1\omega^4 + x_2\omega^8 + x_3\omega^{12} + x_4\omega^{16} + x_5\omega^{20} + x_6\omega^{24} + x_7\omega^{28} \\ c_5 &= x_0 + x_1\omega^5 + x_2\omega^{10} + x_3\omega^{15} + x_4\omega^{20} + x_5\omega^{25} + x_6\omega^{30} + x_7\omega^{35} \\ c_6 &= x_0 + x_1\omega^6 + x_2\omega^{12} + x_3\omega^{18} + x_4\omega^{24} + x_5\omega^{30} + x_6\omega^{36} + x_7\omega^{42} \\ c_7 &= x_0 + x_1\omega^7 + x_2\omega^{14} + x_3\omega^{21} + x_4\omega^{28} + x_5\omega^{35} + x_6\omega^{42} + x_7\omega^{49} \end{aligned}$$

依照奇數與偶數項重新排列之後，我們可得

$$\begin{aligned} c_0 &= (x_0 + x_2 + x_4 + x_6) + (x_1 + x_3 + x_5 + x_7) \\ c_1 &= (x_0 + x_2\omega^2 + x_4\omega^4 + x_6\omega^6) + \omega(x_1 + x_3\omega^2 + x_5\omega^4 + x_7\omega^6) \\ c_2 &= (x_0 + x_2\omega^4 + x_4\omega^8 + x_6\omega^{12}) + \omega^2(x_1 + x_3\omega^4 + x_5\omega^8 + x_7\omega^{12}) \\ c_3 &= (x_0 + x_2\omega^6 + x_4\omega^{12} + x_6\omega^{18}) + \omega^3(x_1 + x_3\omega^6 + x_5\omega^{12} + x_7\omega^{18}) \end{aligned}$$

$$\begin{aligned}
c_4 &= (x_0 + x_2 + x_4 + x_6) - (x_1 + x_3 + x_5 + x_7) \\
c_5 &= (x_0 + x_2\omega^2 + x_4\omega^4 + x_6\omega^6) - \omega(x_1 + x_3\omega^2 + x_5\omega^4 + x_7\omega^6) \\
c_6 &= (x_0 + x_2\omega^4 + x_4\omega^8 + x_6\omega^{12}) - \omega^2(x_1 + x_3\omega^4 + x_5\omega^8 + x_7\omega^{12}) \\
c_7 &= (x_0 + x_2\omega^6 + x_4\omega^{12} + x_6\omega^{18}) - \omega^3(x_1 + x_3\omega^6 + x_5\omega^{12} + x_7\omega^{18})
\end{aligned}$$

我們可將上列的式子重新寫為

$$\begin{aligned}
c_0 &= u_0 + v_0 \\
c_1 &= u_1 + \omega v_1 \\
c_2 &= u_2 + \omega^2 v_2 \\
c_3 &= u_3 + \omega^3 v_3 \\
c_4 &= u_0 - v_0 = u_0 + \omega^4 v_0 \\
c_5 &= u_1 - \omega v_1 = u_1 + \omega^5 v_1 \\
c_6 &= u_2 - \omega^2 v_2 = u_2 + \omega^6 v_2 \\
c_7 &= u_3 - \omega^3 v_3 = u_3 + \omega^7 v_3
\end{aligned}$$

其中

$$\begin{aligned}
u_0 &= x_0 + x_2 + x_4 + x_6 \\
u_1 &= x_0 + x_2\omega^2 + x_4\omega^4 + x_6\omega^6 \\
u_2 &= x_0 + x_2\omega^4 + x_4\omega^8 + x_6\omega^{12} \\
u_3 &= x_0 + x_2\omega^6 + x_4\omega^{12} + x_6\omega^{18}
\end{aligned}$$

圖4.10顯示 1 的 8 次方主根 $\omega_8 = e^{i2\pi/8}$ 及其他 7 個 1 的 8 次方複數根 $\omega_8^1 = e^{i2\pi/8}, \omega_8^2 = e^{i2\pi2/8}, \dots, \omega_8^7 = e^{i2\pi7/8}$ 。

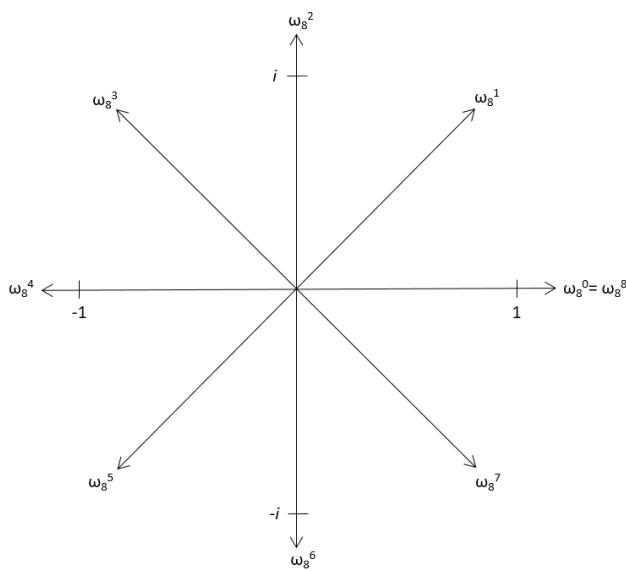


圖 4.10: 1 的 8 次方根

$\Leftrightarrow z = \omega^2 = e^{i2\pi/4}$ ，我們可得

$$\begin{aligned}
u_0 &= x_0 + x_2 + x_4 + x_6 \\
u_1 &= x_0 + x_2z + x_4z^2 + x_6z^3 \\
u_2 &= x_0 + x_2z^2 + x_4z^4 + x_6z^6 \\
u_3 &= x_0 + x_2z^3 + x_4z^6 + x_6z^9
\end{aligned}$$

我們可以看出來 u_0, u_1, u_2, u_3 實際上是偶數項 x_0, x_2, x_4, x_6 的快速傅立葉變換係數。同樣的道理， v_0, v_1, v_2, v_3 是奇數項 x_1, x_3, x_5, x_7 的快速傅立葉變換係數。

一般而言，令 $\omega = e^{i2\pi/n}$ 為 1 的 n 次方主根， n 為 2 的指數次方 ($n = 2^q, q = 1, 2, \dots$)。我們有

$$\begin{aligned}\omega^n &= 1 \\ \omega^{n/2} &= -1\end{aligned}$$

因此，

$$\begin{aligned}c_k &= u_k + \omega^k v_k \\ c_{k+\frac{n}{2}} &= u_k - \omega^k v_k\end{aligned}$$

上式中 u_k 是偶數項的快速傅立葉變換係數，而 v_k 是奇數項的快速傅立葉變換係數。我們可據此寫下採奇數項計算與偶數項計算分治解題策略的 FFT(快速傅立葉變換) 演算法。

Algorithm FFT(x)

▷ 快速傅立葉變換演算法

Input: 離散序列 $x = x_0, x_1, \dots, x_{n-1}, n = 2^q, q = 0, 1, 2, \dots$ (n 為 2 的指數次方)

Output: x 的傅立葉變換係數序列 $c_k = \sum_{j=0}^{n-1} x_j e^{ijk2\pi/n}, k = 0, 1, \dots, n - 1$

```

1:  $n \leftarrow |x|$                                 ▷  $n$  為  $x$  的長度
2: if  $n = 1$  then
3:   return  $x$ 
4:  $\omega_n \leftarrow e^{i2\pi/n}$ 
5:  $\omega \leftarrow 1$                                 ▷  $\omega$  用於臨時儲存  $\omega_n^k$  之值
6:  $x_{even} \leftarrow [x_0, x_2, \dots, x_{n-2}]$       ▷  $x_{even}$  為偶數項序列
7:  $x_{odd} \leftarrow [x_1, x_3, \dots, x_{n-1}]$       ▷  $x_{odd}$  為奇數項序列
8:  $u \leftarrow \text{FFT}(x_{even})$                   ▷  $u$  為偶數項序列之 FFT 係數序列
9:  $v \leftarrow \text{FFT}(x_{odd})$                   ▷  $v$  為奇數項序列之 FFT 係數序列
10: for  $k \leftarrow 1$  to  $\frac{n}{2} - 1$  do
11:    $c_k \leftarrow u_k + \omega v_k$ 
12:    $c_{k+\frac{n}{2}} \leftarrow u_k - \omega v_k$ 
13:    $\omega \leftarrow \omega \omega_n$                       ▷  $\omega$  儲存下個遞迴使用的  $\omega_n^k$  之值
14: return  $x$ 

```

令 $T(n)$ 是 FFT 演算法作用在長度為 n 之離散序列的時間複雜度。FFT 演算法在使用二次遞迴呼叫得到偶數項序列以及奇數項序列之 FFT 係數之後，還需要花 cn 的時間複雜度彙整出整個序列的 FFT 係數，因此我們可得

$$T(n) = 2T(\frac{n}{2}) + cn = O(n \log n)$$

4.7 結語

《孫子兵法》兵勢篇中提到「凡治眾如治寡，分數是也」。意思是說治理人數多的大軍團可以像治理人數少的小部隊一樣有效，只要將整體分為不同的編制，如班、排、連、團、師、軍等，適當配置每個編制的人數，就可以透過健全的層級管理與指揮，統御好整個大軍團。

分治演算法就是使用同樣的概念，將難以解決的大問題分割為容易解決的小問題並一一克服，即便是輸入規模非常大的問題也可以順利解決。本章所介紹的演算法，包括合併排序

演算法、快速排序演算法、最大連續子序列和演算法、缺陷的西洋棋盤填滿演算法、二維求秩演算法、二維求最大點演算法、最近二維點對演算法及快速傅立葉變換演算法，都是典型的分治演算法。

Chapter 5

分治計算幾何演算法 — 各個擊破獲得最後勝利

Contents

| | |
|-------------------------|----|
| 5.1 計算幾何介紹 | 61 |
| 5.2 計算幾何基本演算法 | 61 |
| 5.3 二維極大點問題 | 65 |
| 5.4 二維求秩演算法 | 67 |
| 5.5 二維最近點對問題 | 69 |
| 5.6 二維範諾圖演算法 | 72 |
| 5.7 二維凸包演算法 | 72 |
| 5.8 結語 | 72 |

5.1 計算幾何介紹

計算幾何 (computational geometry) 是一個電腦科學研究領域，主要研究解決幾何問題的演算法 [7]。在 1962 年，美國麻省理工學院使電腦可以直接在電腦顯示器上直接顯示圖形，並且讓人們直接透過游標的移動輸入及修改圖形，造成這個領域的新興發展。其後更發展出電腦圖學 (computer graphics)、電腦輔助設計 (computer-aided design, CAD)、電腦輔助製造 (computer-aided manufacturing, CAM) 等學科，廣泛應用於美學、娛樂、設計及製造等各方面。

在本章中，我們先介紹二個計算幾何基本演算法，包括線段交點演算法、順時針-逆時針演算法。然後我們介紹幾個使用分治解題策略的分治計算幾何演算法，如二維極大點演算法、二維求秩演算法、二維最近點對演算、二維範諾圖演算法及二維凸包演算法。

5.2 計算幾何基本演算法

在本章中，我們先介紹二個計算幾何基本演算法：線段交點演算法及 3 點順時針-逆時針演算法。

以下我們先介紹線段交點 (line segment intersection, LSI) 演算法。如圖5.1所示，給定二維平面上由點 $P_1 = (x_1, y_1)$ 及點 $P_2 = (x_2, y_2)$ 連線的線段 L_a 及由點 $P_3 = (x_3, y_3)$ 及點 $P_4 = (x_4, y_4)$ 連線的線段 L_b ，LSI 演算法可以求出兩個線段的交點 P 。

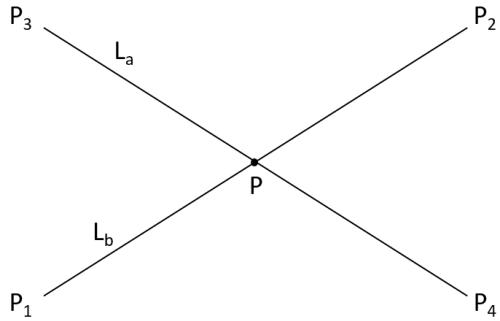


圖 5.1: 兩個線段及其交點示意圖

若點 P_a 在線段 L_a 上，且點 P_b 在線段 L_b 上，則我們可將 P_a 及 P_b 使用以下式子表示：

$$P_a = P_1 + u_a(\overrightarrow{P_1P_2}), 0 \leq u_a \leq 1 \quad (5.1)$$

$$P_b = P_3 + u_b(\overrightarrow{P_3P_4}), 0 \leq u_b \leq 1 \quad (5.2)$$

在式 (5.1) 中， $\overrightarrow{P_1P_2}$ 為起點為 P_1 ，終點為 P_2 的向量可以表示為 $P_2 - P_1$ ；在式 (5.2) 中， $\overrightarrow{P_3P_4}$ 為起點為 P_3 ，終點為 P_4 的向量，可以表示為 $P_4 - P_3$ 。

當 $P = P_a = P_b$ 時，表示 P 是線 L_a 及 L_b 的交點。我們各取出點的 x 軸座標及 y 軸座標則可得：

$$x_1 + u_a(x_2 - x_1) = x_3 + u_b(x_4 - x_3) \quad (5.3)$$

$$y_1 + u_a(y_2 - y_1) = y_3 + u_b(y_4 - y_3) \quad (5.4)$$

利用式 (5.3) 及式 (5.4) 求出 u_a 及 u_b 可得：

$$u_a = \frac{(x_4 - x_3)(y_1 - y_3) - (y_4 - y_3)(x_1 - x_3)}{(y_4 - y_3)(x_2 - x_1) - (x_4 - x_3)(y_2 - y_1)} \quad (5.5)$$

$$u_a = \frac{(x_2 - x_1)(y_1 - y_3) - (y_2 - y_1)(x_1 - x_3)}{(y_4 - y_3)(x_2 - x_1) - (x_4 - x_3)(y_2 - y_1)} \quad (5.6)$$

請注意，式 (5.5) 及式 (5.6) 中的分母相同。我們必須先算出分母，若分母為 0 則兩條線段平行而沒有交點。但是若式 (5.5) 及式 (5.6) 中的分母及分子同時為 0，則兩條線段重疊，有無限多的交點。另外，我們必須檢查 u_a 及 u_b 是否滿足介於 0 與 1 之間的限制（含 0 與 1）。若限制條件滿足，則 $P = P_a = P_b$ 是線段 L_a 及 L_b 的交點；反之，則二線段沒有交點，代表 $P = P_a = P_b$ 在線段之外。

以下為線段交點 (LSI) 演算法的虛擬碼：

Algorithm LSI(P_1, P_2, P_3)

▷ 線段交點演算法

Input: 由點 $P_1 = (x_1, y_1)$ 及點 $P_2 = (x_2, y_2)$ 連線的線段 L_a 及由點 $P_3 = (x_3, y_3)$ 及點 $P_4 = (x_4, y_4)$ 連線的線段 L_b **Output:** 線段 L_a 及 L_b 的交點 P ；若無交點或無窮多交點則傳回 None

```
1:  $deno \leftarrow (y_4 - y_3)(x_2 - x_1) - (x_4 - x_3)(y_2 - y_1)$            ▷ deno 代表分母 (denominator)
2: if  $deno = 0$  then
3:   return None
4:  $u_a \leftarrow \frac{(x_4 - x_3)(y_1 - y_3) - (y_4 - y_3)(x_1 - x_3)}{deno}$ 
5:  $u_b \leftarrow \frac{(x_2 - x_1)(y_1 - y_3) - (y_2 - y_1)(x_1 - x_3)}{deno}$ 
6: if  $(1 \geq u_a \geq 0)$  and  $(0 \geq u_b \geq 1)$  then
7:    $P \leftarrow (x_1 + u_a(x_2 - x_1), y_1 + u_a(y_2 - y_1))$ 
8:   return  $P$ 
9: else
10:  return None
```

以下我們介紹順時針 – 逆時針 (clockwise or counter-clockwise, cw-ccw) 演算法，其輸入為三個二維平面點 P_1, P_2 及 P_3 ；其輸出為 P_1 到 P_2 再到 P_3 的方向旋轉關係，包括輸出為 0 代表三點共線 (collinear)，輸出為 1 代表逆時針方向旋轉 (ccw)，輸出為 -1 代表逆時針方向旋轉 (cw)。圖5.1顯示上述的三種關係。

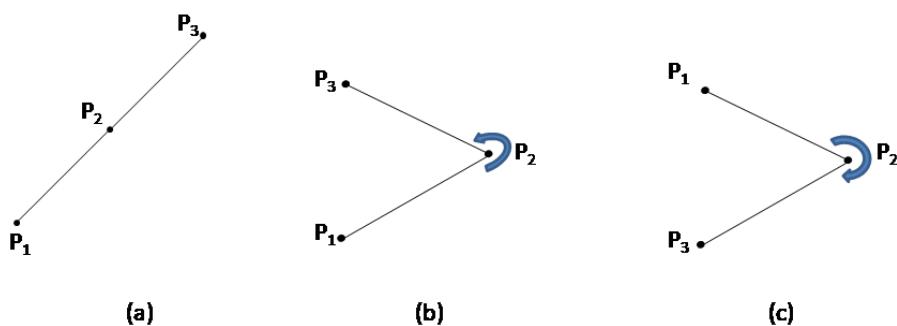


圖 5.2: 三個點的方向旋轉關係：(a) 共線 (collinear)、(b) 逆時針 (counter clockwise, ccw) 及 (c) 順時針 (clockwise, cw)

我們可以藉由向量 $\vec{u} = \overrightarrow{P_1P_2}$ 與向量 $\vec{v} = \overrightarrow{P_1P_3}$ 的外積 (cross product) 的正向與負向來判斷三個點的方向旋轉關係。實際上 \vec{u} 與 \vec{v} 的外積是一個長度為 $|\vec{u}| |\vec{v}| \sin \theta$ ，而且與 \vec{u} 、 \vec{v} 都垂直的向量，其中 $|\cdot|$ 代表向量的長度， $\theta (0 \leq \theta \leq \pi)$ 為 \vec{u} 與 \vec{v} 的夾角。一般我

們可以使用右手法則來決定 $\vec{u} \times \vec{v}$ 的方向：右手四指由 \vec{u} 的方向轉向 \vec{v} ，則右手大拇指所指的方向就是 $\vec{u} \times \vec{v}$ 的方向。舉例而言，若 \vec{u} 、 \vec{v} 都在紙面上，則 $\vec{u} \times \vec{v}$ 的方向為正向的方向是由紙面射出，方向為負向則是射入紙面。

若二維平面向量 \vec{u} 與 \vec{v} 分別為 (x_u, y_u) 及 (x_v, y_v) ，則 $\vec{u} \times \vec{v}$ 為正向若 $x_u y_v - x_v y_u$ 為正，反之為負向。若我們以三維空間向量的角度來看的話，則向量 \vec{u} 為 $(x_u, y_u, 0)$ ，向量 \vec{v} 為 $(x_v, y_v, 0)$ ，向量 $\vec{u} \times \vec{v}$ 為 $(0, 0, x_u y_v - x_v y_u)$ ，其中 $x_u y_v - x_v y_u$ 可以決定 $\vec{u} \times \vec{v}$ 的正向與負向。因為在以下的內容中，向量 \vec{u} 與 \vec{v} 都是二維平面向量，因此我們可能會簡略的使用 $\vec{u} \times \vec{v} = x_u y_v - x_v y_u$ 替代 $\vec{u} \times \vec{v} = (0, 0, x_u y_v - x_v y_u)$ 。

如圖5.3所示，若將 P_1 視為原點，則 $\overrightarrow{P_1 P_2}$ 與 $\overrightarrow{P_1 P_3}$ 的外積向量為正向。而 $\overrightarrow{P_1 P_2}$ 與 $\overrightarrow{P_1 P'_3}$ 的外積為負向。讀者可以代入 $P_1 = (0, 0), P_2 = (6, 2), P_3 = (4, 7), P'_3 = (4, 1)$ ，則 $\overrightarrow{P_1 P_2} = (6, 2), \overrightarrow{P_1 P_3} = (4, 7)$ 且 $\overrightarrow{P_1 P'_3} = (4, 1)$ 。我們可得 $\overrightarrow{P_1 P_2} \times \overrightarrow{P_1 P_3} = 4 \times 7 - 4 \times 2 = 34$ 為正數，而 $\overrightarrow{P_1 P_2} \times \overrightarrow{P_1 P'_3} = 6 \times 1 - 4 \times 2 = -2$ 為負數。

事實上，若點 P_1 到點 P_2 再到點 P_3 是逆時針旋轉關係，則 $\overrightarrow{P_1 P_3}$ 與 $\overrightarrow{P_1 P_2}$ 的外積為正（外積向量為正向）。相反的，若 P_1 到 P_2 再到 P_3 是順時針旋轉關係，則 $\overrightarrow{P_1 P_2}$ 與 $\overrightarrow{P_1 P_3}$ 的外積為負（外積向量為負向）。我們可以據此設計以下的順時針-逆時針 (CW-CCW) 演算法：

Algorithm CW – CCW(P_1, P_2, P_3)

▷ 順時針-逆時針演算法

Input: 三個二維平面點 $P_1 = (x_1, y_1), P_2 = (x_2, y_2), P_3 = (x_3, y_3)$

Output:

- 0: 代表 (P_1, P_2) 及 P_3 共線)
 - 1: 代表 $(P_1$ 到 P_2 再到 P_3 為逆時針旋轉)
 - 1: 代表 $(P_1$ 到 P_2 再到 P_3 為順時針旋轉)
 - 1: $\vec{u} \leftarrow \overrightarrow{P_1 P_2} \leftarrow (x_2 - x_1, y_2 - y_1)$
 - 2: $\vec{v} \leftarrow \overrightarrow{P_1 P_3} \leftarrow (x_3 - x_1, y_3 - y_1)$
 - 3: $cp \leftarrow \vec{u} \times \vec{v} \leftarrow (x_2 - x_1)(y_3 - y_1) - (x_3 - x_1)(y_2 - y_1)$
▷ 實際上 $\vec{u} \times \vec{v} = (0, 0, (x_2 - x_1)(y_3 - y_1) - (x_3 - x_1)(y_2 - y_1))$
 - 4: **if** $cp > 0$ **then**
 - 5: **return** 1
 - 6: **else if** $cp < 0$ **then**
 - 7: **return** -1
 - 8: **else**
 - 9: **return** 0
-

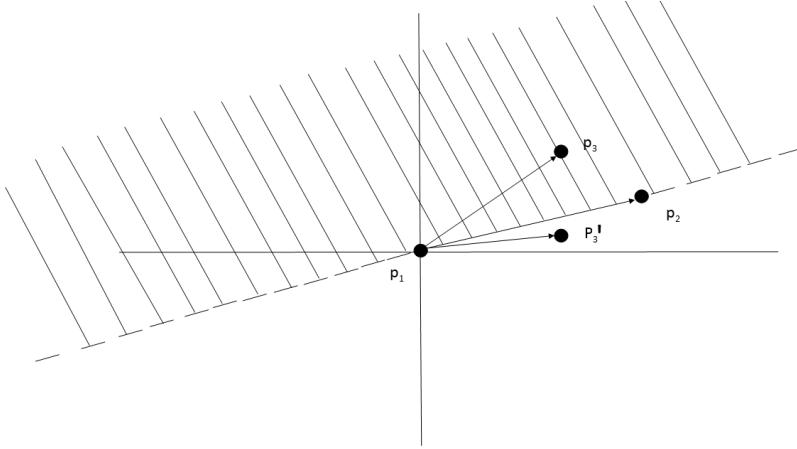


圖 5.3: 向量 $\overrightarrow{P_1P_2}$ 與向量 $\overrightarrow{P_1P_3}$ 外積向量的正向與負向。 P_3 在斜線區域中，則 $\overrightarrow{P_1P_2} \times \overrightarrow{P_1P_3}$ 為正向。 P'_3 不在斜線區域中，則 $\overrightarrow{P_1P_2} \times \overrightarrow{P_1P'_3}$ 為負向。

5.3 二維極大點問題

在本節中我們介紹二維極大點 (2D Maxima Finding) 演算法。以下，我們先描述二維平面點的「支配」定義與「極大點」的定義。

定義 5.1. 支配 (dominate): 令 $A = (a_x, a_y), B = (b_x, b_y)$ 為二維平面上的點，則我們說 A 支配 (dominate) B (記為 $A \succ B$)，若且唯若 $a_x > b_x$ 且 $a_y > b_y$ 。

定義 5.2. 極大點 (maxima):

如果一個點不被其他點所支配，則我們稱此點是不被支配的 (non-dominated)，或稱此點為極大點 (maxima)。

給定一個平面點集合，一般會有不只一個極大點。例如，圖5.4中顯示給定 14 個平面點中有 5 個極大點。

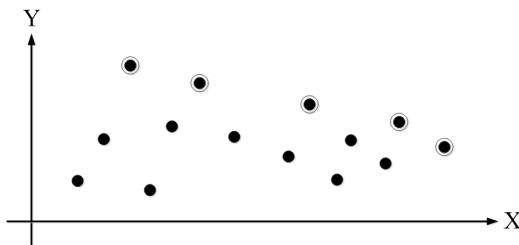


圖 5.4: 紿定的 14 個平面點中有 5 個極大點 (加圓框表示)。

給定一個由 n 個二維平面點所構成的點集合 S ，我們可以使用窮舉式 (exhaustive) 演算法，比較所有的可能點對的「支配」關係來求出 S 中的極大點，這個做法的時間複雜度為 $O(n^2)$ 。

我們也可以使用分治解題策略來設計更有效率的演算法，達到比 $O(n^2)$ 還要低的時間複雜度，這個演算法如下所示：

Algorithm 2DMaximaFinding(S, n)

▷ 二維極大點演算法

Input: n 個二維平面點構成的集合 S **Output:** 在點集合 S 中所有的極大點 (maxima) 集合步驟 1: 若 $n=1$ ，則回傳 S 中唯一一個點為極大點並結束。步驟 2: 找出所有點 X 軸值的中位數 (median)，將 S 中的點分為二個集合 S_L 與 S_R 。步驟 3: 遞迴呼叫 $S_L = 2DMaximaFinding(S_L, \lfloor n/2 \rfloor)$ 及 $M_R = 2DMaximaFinding(S_R, \lceil n/2 \rceil)$ 分別求出 S_L 與 S_R 中的極大點集合 M_L 與 M_R 。步驟 4: 在 M_R 的極大點中找出最大的 Y 軸值 y^* 。對每個在 M_L 中的極大點進行處理，如果該點的 Y 軸值小於 y^* ，則將該點自 M_L 中移除。回傳 $M_L \cup M_R$ 為極大點集合。

圖5.5展示二維求極大點演算法使用分治策略解決二維求極大點問題的基本概念。也就是點集合 S 分為 S_L 與 S_R 之後可利用遞迴的方式分別求出 S_L 與 S_R 中的極大點集合 M_L 與 M_R ，然後再調整 M_R 中的極大點之後，合併 M_L 與 M_R 中的極大點並傳回。

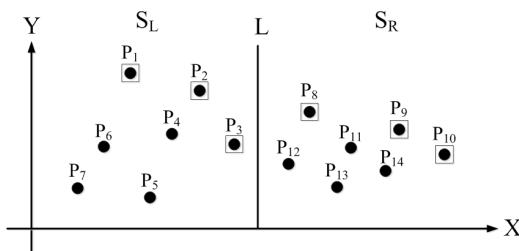


圖 5.5: 二維求極大點演算法使用分治策略解決二維求極大點問題基本概念。

以下我們分析二維求極大點演算法的時間複雜度 $T(n)$ ，在此我們假設以排序演算法找出所有點 X 軸座標的中位數。

- 步驟 1: 1 (這是邊界條件，也就是 $T(1)=1$)
- 步驟 2: $O(n \log n)$ (以 X 軸值的排序尋找 X 軸值的中位數)
- 步驟 3: $T(\frac{n}{2}) + T(\frac{n}{2})$
- 步驟 4: $O(n)$ (合併左半及右半的極大點)

因此總時間複雜度為

$$\begin{aligned}
 T(n) &= 2T\left(\frac{n}{2}\right) + cn \log n \\
 &= 4T\left(\frac{n}{4}\right) + cn \log \frac{n}{2} + cn \log n \quad [\text{註: 除第一項外共有 } \log 4 = 2 \text{ 項}] \\
 &= 8T\left(\frac{n}{8}\right) + cn \log \frac{n}{4} + cn \log \frac{n}{2} + cn \log n \quad [\text{註: 除第一項外共有 } \log 8 = 3 \text{ 項}] \\
 &= nT(1) + cn(\log 2 + \dots + \log \frac{n}{8} + \log \frac{n}{4} + \log \frac{n}{2} + \log n) \quad [\text{註: 除第一項外共有 } \log n \text{ 項}] \\
 &\leq nT(1) + \frac{cn \log n (\log n + \log 2)}{2} \\
 &= O(n \log^2 n)
 \end{aligned}$$

以上介紹的二維求極大點演算法時間複雜度為 $O(n \log^2 n)$ ，展示分治解題策略可以設計出時間複雜度比窮舉式 (exhaustive) 方法還低的演算法。事實上，我們還可以刪尋 (prune and search) 解題策略設計時間複雜度為 $O(n)$ 的中位數演算法，可以進一步降低二維求極大點演算法的時間複雜度。我們將在下章中介紹使用刪尋解題策略的中位數演算法。若二維求極大點演算法使用刪尋解題策略的中位數演算法來求出 X 軸值的中位數，則它的時間複雜度再降為 $O(n \log n)$ ，分析如下：

- 預先排序: $O(n \log n)$
- 步驟 1: 1(這是邊界條件，也就是 $T(1)=1$)
- 步驟 2: $O(n)$ (使用刪尋解題策略的中位數演算法求出 X 軸值的中位數)
- 步驟 3: $T(\frac{n}{2})+T(\frac{n}{2})$
- 步驟 4: $O(n)$ (合併左半及右半的極大點)

因此總時間複雜度為

$$T(n) = 2T(\frac{n}{2}) + cn = O(n \log n)$$

5.4 二維求秩演算法

在本節中我們介紹可以解決二維求秩 (2D rank finding) 問題的二維求秩演算法。以下，我們先介紹比較簡單的數集合求秩 (number set rank finding) 問題。我們先定義數集合的秩 (rank)。

定義 5.3. 秩 (rank): 紿定一個數的集合 S 及一個包含在 S 中的數 a ， a 的秩記為 $\text{rank}(a)$ ，定義為 S 中所有比 a 小的數的總數。

根據上述的定義， S 中最小數的秩為 0；而 S 中最大數的秩為 $|S| - 1$ 。例如，給定 $S = \{2, 18, 25, 41\}$ ，則 $\text{rank}(2)=0$ ， $\text{rank}(18)=1$ ， $\text{rank}(25)=2$ ， $\text{rank}(41)=3$ 。又例如，給定 $S = \{3, 1, 4, 2\}$ ，則 $\text{rank}(1)=0$ ， $\text{rank}(2)=1$ ， $\text{rank}(3)=2$ ， $\text{rank}(4)=3$ 。

以下數集合求秩演算法可以根據定義，求出數 a 在數集合 S 中的秩。其基本概念為：先設定 a 的秩為 0，然後一一比對 S 中的數，若有比 a 小的數則將 a 的秩加 1。

Algorithm RankFinding(a, S)

Input: 一個數集合 S 及一個包含在 S 中的數 a

Output: 數 a 在數集合 S 中的秩 R

```

1:  $R \leftarrow 0$ 
2: for every  $e \in S$  do
3:   if  $e < a$  then
4:      $R \leftarrow R + 1$ 
5: return  $R$ 

```

我們很容易可以看出求秩演算法的時間複雜度為 $O(n)$ ，其中 $n = |S|$ ，因為 S 中的每個元素都要被比較過才能決定 a 的秩。若我們要求出每個數的秩，我們則可以使用排序演算法對 S 中的數排序，然後每個數在排序後的順序位置就是它的秩，這樣的時間複雜度為 $O(n \log n)$ 。

以下我們描述二維求秩問題，這是數集合求秩問題的二維平面點的延伸版本。以下我們先給定二維平面點「秩」的定義，這個定義與二維平面點的「支配」定義有關。這個「支配」定義在前一節中已經介紹過，但是為了方便起見，我們再重複介紹一次。

定義 5.4. 支配 (dominate): 令 $A = (a_x, a_y), B = (b_x, b_y)$ 為二維平面上的點，則我們說 A 支配 (dominate) B (記為 $A \succ B$) 若且唯若 $a_x > b_x$ 且 $a_y > b_y$ 。

定義 5.5. 秩 (rank) 或二維秩 (2D rank): 紿定一個由二維平面點所構成的點集合 S 及一個包含於 S 中的點 A ， A 的秩 (rank) 定義為點集合 S 中被 A 支配的點的總數。

給定一個點集合 S ，我們可以根據上述的定義計算出每個點的秩。例如，圖5.6中顯示 5 個二維平面點及其所對應的秩。

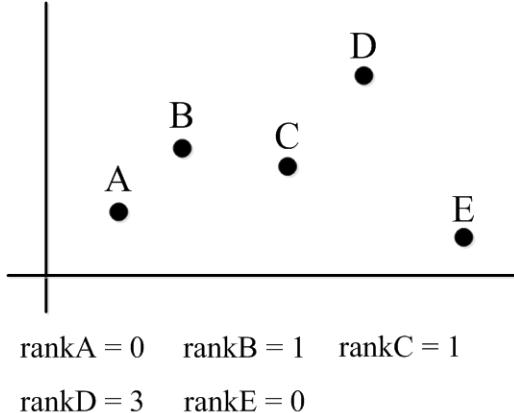


圖 5.6: 5 個二維平面點及其所對應的秩。

給定一個由 n 個二維平面點所構成的點集合 S 及一個包含於 S 中的點 A ，我們可以比照數集合求秩演算法的方式，讓 A 與所有的點比對過「支配」關係之後求出 A 的秩。但是若想要求出 S 中所有點的秩，我們可以使用窮舉式 (exhaustive) 方式，在比較所有可能成對點的「支配」關係之後，可以求出點集合 S 中所有點的秩，這個做法的時間複雜度為 $O(n^2)$ 。但是否如數集合的求秩一樣，數集合的求秩也具有比 $O(n^2)$ 還要低的時間複雜？這問題的答案是肯定的，我們可以使用分治解題策略來設計更有效率的演算法，達到比 $O(n^2)$ 還要低的時間複雜度，這個演算法描述如下：

Algorithm 2DRankFinding(S, n)

Input: 由 n 個二維平面點所構成的點集合 S

Output: 在集合 S 中所有點的二維秩 R

步驟 1: 若 $n=1$ ，則回傳 S 中唯一一個點的秩為 0 並結束。

步驟 2: 找出所有點 X 軸值的中位數 (median)，將 S 中的點分為二個集合 S_L 與 S_R 。

步驟 3: 遞迴呼叫 $2DRankFinding(S_L, \lfloor n/2 \rfloor)$ 及 $2DRankFinding(S_R, \lceil n/2 \rceil)$ 分別求出 S_L 與 S_R 中所有點的秩。

步驟 4: 根據 Y 軸值排序所有在 S 中的點，依序由小而大掃描所有點，利用變數 $counter$ 記錄目前掃描到 S_L 中點的總數。 $counter$ 起始值為 0，若掃描到 S_L 中的點則 $counter$ 加 1，若掃描到 S_R 中的點，則該點的秩加上 $counter$ 。全部的點掃描後，回傳 S 中所有點的秩。

圖5.7展示了二維求秩演算法使用分治策略解決二維求秩問題的基本概念。也就是集合 S 分為 S_L 與 S_R 之後可利用遞迴的方式分別求出每個點的秩，然後在合併時調整 S_R 之點的秩。

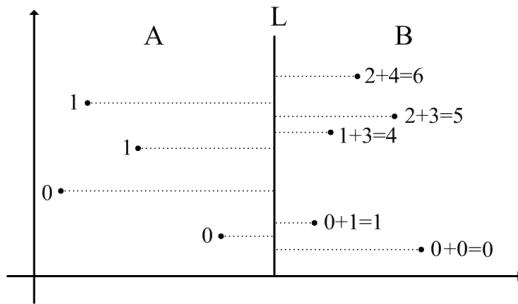


圖 5.7: 二維求秩演算法使用分治策略解決二維求秩問題基本概念。

以下我們分析二維求秩演算法的時間複雜度 $T(n)$ ，在此我們假設以排序演算法找出所有點 X 軸座標的中位數。

- 步驟 1: 1(這是邊界條件，也就是 $T(1)=1$)
- 步驟 2: $O(n \log n)$ (以 X 軸值的排序尋找 X 軸值的中位數)
- 步驟 3: $T(\frac{n}{2})+T(\frac{n}{2})$
- 步驟 4: $O(n \log n)$ (以 Y 軸值的排序進行點的依序掃描)

因此總時間複雜度為

$$\begin{aligned}
 T(n) &= 2T\left(\frac{n}{2}\right) + cn \log n \\
 &= 4T\left(\frac{n}{4}\right) + cn \log \frac{n}{2} + cn \log n \quad [\text{註: 除第一項外共有 } \log 4 = 2 \text{ 項}] \\
 &= 8T\left(\frac{n}{8}\right) + cn \log \frac{n}{4} + cn \log \frac{n}{2} + cn \log n \quad [\text{註: 除第一項外共有 } \log 8 = 3 \text{ 項}] \\
 &= nT(1) + cn(\log 2 + \dots + \log \frac{n}{8} + \log \frac{n}{4} + \log \frac{n}{2} + \log n) \quad [\text{註: 除第一項外共有 } \log n \text{ 項}] \\
 &\leq nT(1) + \frac{cn \log n(\log n + \log 2)}{2} \\
 &= O(n \log^2 n)
 \end{aligned}$$

5.5 二維最近點對問題

在本節中我們介紹可以解決二維最近點對 (closest pair of 2D points) 問題的二維最近點對演算法。我們先介紹比較簡單的最近數對 (closest pair of numbers) 問題：給定一個由 n 個數所構成的數集合 S ，尋找集合中兩個最接近的數（差最小的兩個數），並輸出其差。我們可以使用排序演算法，將 S 中的所有數排序，然後依序檢查排序後相鄰二數的差，在不斷更新並紀錄最小的差之後，即可找出最接近二數的差。這個做法的時間複雜度為 $O(n \log n)$ ，這是因為排序的時間複雜度為 $O(n \log n)$ ，而依序檢查連續二數差的時間複雜度為 $O(n)$ 。

以下我們描述二維最近點對 (closest pair of 2D points) 問題：給定一個由 n 個二維平面點所構成的點集合 S ，尋找兩個最接近的點（距離最小的兩個點），並輸出其距離。

我們可以使用窮舉式 (exhaustive) 演算法，計算所有的二維平面點對 (point pair) 的距離，來找出最接近的兩個點的距離即可解決問題，這樣的做法時間複雜度為 $O(n^2)$ 。而我們也可以使用分治解題策略配合預先排序 (presorting) 的使用來設計更有效率的演算法，達到 $O(n \log n)$ 的時間複雜度，這個演算法如下所示：

Algorithm 2DClosestPointPair(S, n)

▷ 二維最近點對演算法

Input: n 個二維平面點所構成的點集合 $S, n \geq 2$ **Output:** 集合 S 中最點對的距離 d

步驟 1: 進行 X 軸值與 Y 軸值預先排序 (presorting) · 並將排序結果分別儲存以便後續進行 X 軸值中點計算及合併最短點對距離之用。

步驟 2: 若 $n \leq 3$ · 則回傳 S 中唯一點對的距離 d 並結束 (針對 $n = 2$ 的狀況) ; 或是回傳 S 中三個點對中最短的點對距離 d 並結束 (針對 $n = 3$ 的狀況) 。

步驟 3: 找出 S 中所有點的 X 軸值的中位數 (median) $m(X$ 軸值排序第 $\lfloor n/2 \rfloor$ 個 · 畫出垂直於 X 軸的直線 L · 將 S 中的點分為二個集合 S_L 與 S_R 。

步驟 4: 遞迴呼叫 $d_L = 2DClosestPointPair(S_L, \lfloor n/2 \rfloor)$ 及 $d_R = 2DClosestPointPair(S_R, \lceil n/2 \rceil)$ 分別求出 S_L 與 S_R 中最近的點對距離 d_L 與 d_R · 且令 $d = \min(d_L, d_R)$ 。

步驟 5: 針對於每個 X 軸值介於 $m - d$ 到 m 之間 (含 $m - d$ 與 m) 的點 p · 以 y_p 記錄其 Y 軸值 · 並尋找每一個 X 軸值介於 m 到 $m + d$ 之間 (不含 m 但是含 $m + d$) 且 Y 軸值介於 $y_p + d$ 到 $y_p - d$ 之間 (含 $y_p + d$ 與 $y_p - d$ 的) 的點 q · 若點 q 與 p 的距離為 d' 且 $d' < d$ · 則令 $d = d'$ · 當檢查完所有符合上述條件的點 p 與點 q 之後回傳 d 並結束執行。

以下我們說明預先排序 (presorting) 如何使演算法有效率的解決問題。二維最近點對演算法一開始就進行 X 軸值與 Y 軸值預先排序 (presorting) · 期作法為根據 X 軸值與 Y 軸值來預先排序 S 中所有的點 · 並將排序結果分別儲存在屬於全域變數 (global variable) 的陣列 (或列表) 中 · 例如儲存在 x_order 及 y_order 陣列中 · 我們可以將給定的平面點視為物件 · 因此陣列中的第一個元素儲存一個指標指向排序結果 X 軸值或 Y 軸值最小的點的物件 · 第二個元素儲存一個指標指向排序結果 X 軸值或 Y 軸值第二小的點的物件 · 依此類推。

每個二維平面點所對應的物件儲存點的各種屬性 · 包括平面點座標的 X 軸值及 Y 軸值 · 也包括點的 X 軸值及 Y 軸值的排序次序 · 次序最小為 1 而最大為 n · 為方便遞迴呼叫 · 我們可以將 $2DClosestPointPair$ 的輸入參數改為 (S, x_order, l, r) · 其中 l 與 r 是陣列 x_order 的索引值 · 代表本次呼叫僅處理點集合 S 中 X 軸值排序次序為第 l 到第 r 之間的點 · 令這些點的 X 軸值中位數為 m · 則這些點中 X 軸值等於 m 的點在陣列 x_order 的索引值為 $\lfloor (l+r)/2 \rfloor$ · 因此 · 在遞迴呼叫時將呼叫 $d_L = 2DClosestPointPair(S, x_order, l, \lfloor (l+r)/2 \rfloor)$ 與 $d_R = 2DClosestPointPair(S, x_order, \lfloor (l+r)/2 \rfloor + 1, r)$ 。

步驟五是合併最近點對距離的動作 · 可以透過索引值 $\lfloor (l+r)/2 \rfloor$ 依序在陣列 x_order 中找出 X 軸值介於 $m - d$ 與 m 的所有點 p · 令點 p 的 Y 軸值為 y_p · 我們可以利用點 p 所屬的物件屬性找出點 p 在陣列 y_order 中的排序次序 · 也就是點 p 在陣列 y_order 中的相對索引值 · 利用這個索引值依序遞增或遞減 · 就可以找出 Y 軸值介於 $y_p + d$ 與 $y_p - d$ 之間的所有點 · 若有些點的 X 軸值不是介於 m 與 $m + d$ 之間 (不含 m 但是含 $m + d$) · 則略過這些點 · 反之 · 對每一個沒有被略過的點 q 計算點 p 與點 q 的距離 · 以找出其中最小的點 p 與點 q 的距離 d' · 以便可以在 d' 比 d 小時更新 d 的值以完成合併最近點對距離的動作。

圖5.8展示使用分治演算法解決二維最近點對問題的過程。另外 · 圖5.9則展示二維最近點演算法在合併過程中 · 對於一個 X 軸值介於 $m - d$ 與 m 的點 p 而言 · 其所需要比對的點都在垂直長 $2d$ 與水平寬 d 的長方形區域 A 中 · 因為 $d = \min(d_L, d_R)$ · 所以每對點的距離必定大於或等於 d · 因此區域 A 中最多僅包含 6 個點 · 也就是說針對每個點 p 只需要常數個操作就可以完成最短點對距離合併。

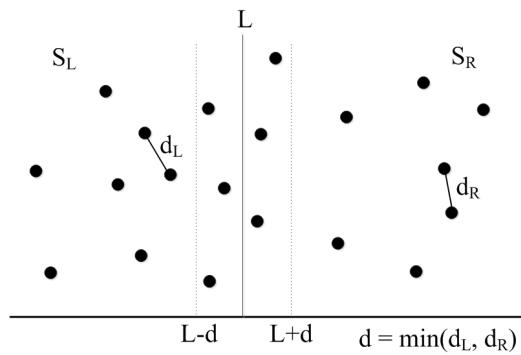


圖 5.8: 使用分治演算法解決二維最近點對問題的過程。

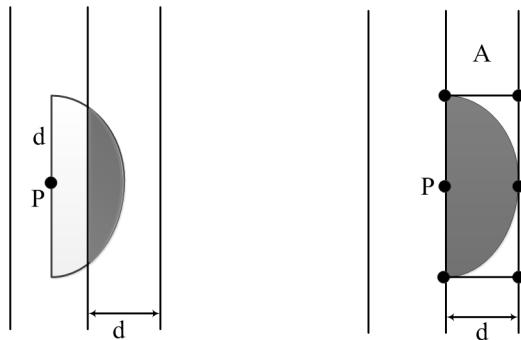


圖 5.9: 在長方形區域 A 中，最多僅 6 個點。

以下我們分析二維最近點對演算法的時間複雜度 $T(n)$ ，我們先列出個步驟的時間複雜度：

- 步驟 1: 預先排序: $O(n \log n)$
- 步驟 2: 1(這是邊界條件，也就是 $T(2)=1$)
- 步驟 3: $O(n)$ (透過預先排序尋找 X 軸值的中位數)
- 步驟 4: $T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right)$
- 步驟 5: $O(n)$ (透過預先排序針對線 L 附近的點計算其距離以完成最短點對距離合併)

如上所列，總時間複雜度為

$$T(n) = 2T\left(\frac{n}{2}\right) + cn = O(n \log n)$$

5.6 二維範諾圖演算法

5.7 二維凸包演算法

5.8 結語

蓋烏斯·尤利烏斯·凱撒 (拉丁文: Gaius Iulius Caesar¹，西元前 100 年 7 月 - 前 44 年 3 月 15 日)，是羅馬共和國體制轉向羅馬帝國的關鍵人物，歐洲史稱凱撒大帝 [8]。在羅馬共和國的內戰期間，凱撒使用所謂的羅馬兵法 [9]，選擇主要攻擊方向，巧妙地分割敵軍兵力並各個擊破，獲得羅馬內戰的勝利。我們可以說凱撒兵法就是使用分治策略來求勝。

本章中所介紹的計算幾何分治演算法也是使用分治策略解決問題，包括二維極大點演算法、二維求秩演算法、二維最近點對演算、二維範諾圖演算法及二維凸包演算法。這些演算法都採取一再將平面一分為二並一一征服的方式有效地解決問題，這與羅馬兵法在軍事上採用的策略，可以說是如出一轍的。

¹Caesar 拉丁文發音為凱撒，但是英文發音已轉為西撒

Chapter 6

刪尋演算法 — 化整為零蠶食而盡

Contents

| | | |
|-----|--------------|----|
| 6.1 | 刪尋演算法基本概念 | 73 |
| 6.2 | 二元搜尋演算法 | 74 |
| 6.3 | 選取演算法與中位數演算法 | 75 |
| 6.4 | 受限最小圓演算法 | 77 |
| 6.5 | 結語 | 79 |

6.1 刪尋演算法基本概念

在本單元中我們介紹刪尋 (prune-and-search) 演算法。刪尋演算法使用刪尋解題策略解決問題，可以很有效率的解決某些問題，又稱為刪除再搜尋策略。刪尋演算法由許多的迭代 (iteration) 所組成，每次迭代都刪除 (prune) 輸入資料的一個固定比例的部份 (假設固定比例為 f , $0 < f < 1$)，而後再採用相同的演算法遞迴地 (recursively) 從剩餘資料中搜尋 (search) 出解答，若仍然無法找出解答則再刪除固定比例的輸入資料。這樣經過幾次迭代後，輸入資料的規模將會小到足以讓問題在常數時間 c 之內直接解決。

一般而言，刪尋演算法的時間複雜度與每次迭代所需要的時間複雜度相同。這是一個很特殊的特性，說明如下：

假設輸入規模為 n ，而刪尋演算法每次迭代都刪除輸入資料固定比例的一部份，令固定比例為 f ($0 < f < 1$)，若每次迭代執行所需的時間複雜度為 $cn^k = O(n^k)$, $k > 0$ ，則在最差狀況下刪尋演算法的時間複雜度也為 $O(n^k)$ 。

以下我們推論上述的特性。假設刪尋演算法的時間複雜度為 $T(n)$ ，則我們可得：

$$\begin{aligned} T(n) &= T((1-f)n) + cn^k \\ &= T((1-f)^2n) + cn^k + c(1-f)^kn^k = \dots \\ &= T((1-f)^pn) + cn^k(1 + (1-f)^k + (1-f)^{2k} + \dots + (1-f)^{pk}) \end{aligned}$$

令 $(1-f)^p n = n'$ ，其中 n' 是一個很小的整數，使得 $T(n')$ 是一個常數 c' ，則我們可得：

$$\begin{aligned} T(n) &= c' + cn^k(1 + (1-f)^k + (1-f)^{2k} + \dots + (1-f)^{pk}) \\ &\leq c' + cn^k(1 + (1-f)^k + (1-f)^{2k} + \dots + (1-f)^{pk} + \dots) \\ &\cong c' + \frac{cn^k}{1-(1-f)} \\ &= c' + \frac{cn^k}{f} = O(n^k) \end{aligned}$$

在以上的推論中，我們用到以下的事實：若 $1 - f < 1$ ，則公比 (common ratio) 為 $1 - f$ 的等比級數 (geometric series) 小於等於公比為 $1 - f$ 的收斂無窮等比級數 (infinite geometric series)。公比為 $r (r < 1)$ 的無窮等比級數會收斂，而其值為 $\frac{a_0}{1-r}$ ，其中 a_0 為級數首項。

由以上的推導可以得到以下的結論：

整個刪尋演算法的時間複雜度與每個迭代的時間複雜度 $cn^k, k > 0$ ，具有相同的量級。請注意，上述的結論適用於 $k > 0$ 的情況，若每個迭代的時間複雜度 $ck^0 = c$ 為常數量級則不適用此結論，我們將在稍後看到這個例子。

本章將介紹幾個刪尋演算法，包括二元搜尋演算法、選取與中位數演算法及限制的一圓心演算法。

6.2 二元搜尋演算法

給定一個元素已經由小而大排好順序的陣列，二元搜尋 (binary search) 演算法可以快速找出給定的元素，也就是找出給定元素的索引值或是回傳找不到給定的元素。二元搜尋演算法可以視為刪尋演算法，因為它在每一迭代都會刪除一半的元素，然後在另外一半未被刪除的元素中繼續搜尋給定的元素。以下為二元搜尋演算法的虛擬碼：

Algorithm BinarySearch(A, l, r, t)

Input: 由小到大排序且索引由 0 開始編號的數值陣列 A 、欲搜尋的目標數值 t 、搜尋範圍索引 l (最左) 與 r (最右)

Output: 目標數值的索引值或-1(代表目標數值不在陣列 A 中)

```

1: while  $l \leq r$  do
2:    $m \leftarrow \lfloor (l + r)/2 \rfloor$ 
3:   if  $t = A[m]$  then
4:     return  $m$ 
5:   else
6:     if  $t > A[m]$  then
7:        $l \leftarrow m + 1$ 
8:     else
9:        $r \leftarrow m - 1$ 
10:  return -1

```

上列的二元搜尋 (BinarySearch) 演算法可以給定一個已依由小到大順序排列的數值陣列 A ，在索引 l 與索引 r 之間找出並回傳目標數值 t 的索引，或是回傳-1 以代表目標數值 t 不在陣列 A 中。以下是一個簡單的二元搜尋演算法範例，給定一個包含 2, 8, 11, 27, 38, 43, 52 等元素的陣列 A ，要在 A 中搜尋目標數值 $t = 43$ 的索引，此時演算法會在迭代 2 回傳索引 5。另外，要在 A 中搜尋目標數值 $t = 1$ 的索引，因為在迭代 4 時演算法 while 迴圈的 $l \leq r$ 條件不滿足，因此演算法會在迭代 4 回傳-1，代表目標數值 $t = 1$ 不在陣列 A 中。

| | | | | | | | | |
|---------------|--------------|--------------|----------------|--------------|--------------|--------------|--------------|---------|
| 索引: | -1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 元素值: | | 2 | 8 | 11 | 27 | 38 | 43 | 52 |
| 搜尋 $t = 43$: | | | | | | | | |
| 迭代 1: | | $l \uparrow$ | | | $m \uparrow$ | | $r \uparrow$ | |
| 迭代 2 | | | | | $l \uparrow$ | $m \uparrow$ | $r \uparrow$ | 回傳索引 5 |
| 搜尋 $t = 1$: | | | | | | | | |
| 迭代 1: | | $l \uparrow$ | | | $m \uparrow$ | | $r \uparrow$ | |
| 迭代 2: | | $l \uparrow$ | $m \uparrow$ | $r \uparrow$ | | | | |
| 迭代 3: | | | $lrm \uparrow$ | | | | | |
| 迭代 4: | $r \uparrow$ | $l \uparrow$ | | | | | | 回傳索引 -1 |

二元搜尋演算法可視為分治演算法，因為將之視為在每一次分割之後，一個分割可能存在解答，另一個分割一定不存在解答。而本節則將二元搜尋演算法視為刪尋演算法，因為將之視為在每一個迭代中一定會刪除一半的資料。以下我們分析二元搜尋演算法的時間複雜度 $T(n)$ ，其中 n 為給定陣列的元素個數。請注意，刪尋演算法每個迭代的時間複雜度為常數量級，因此不適用整個刪尋演算法的時間複雜度與每個迭代的時間複雜度具有相同量級的結論。

令二元搜尋演算法的時間複雜度為 $T(n)$ ，其中 n 為給定陣列的元素個數。我們可得：

$$T(n) = \begin{cases} T\left(\frac{n}{2}\right) + 1, & n > 1 \\ 1, & n \leq 1 \end{cases} \quad (6.1)$$

依序以 $n/2$ 、 $n/4$ 、... 代入上式 n 中，我們可得

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + 1 = T\left(\frac{n}{4}\right) + 1 + 1 = \dots = T\left(\frac{n}{2^k}\right) + k \\ \text{令 } \frac{n}{2^k} &= 1 \cdot \text{ 則 } n = 2^k \text{ 且 } k = \log n \cdot \text{ 我們可得} \\ T(n) &= T(1) + k = 1 + k = 1 + \log n = O(\log n) . \end{aligned}$$

6.3 選取演算法與中位數演算法

給定一個擁有 n 個可重複元素的集合 S 及給定一個整數 k ，選取 (Selection) 演算法可以找出 S 中第 k 小的元素。若將給定的 k 設為 $\lceil \frac{n}{2} \rceil$ ，則形成中位數演算法，可以找出元素集合中的中位數 (median)。例如，給定一個擁有 9 個可重複元素的集合 $S = \{1, 2, 2, 4, 5, 6, 7, 8, 9\}$ ，則選取第 2 小的元素為 2，選取第 3 小的元素也為 2。若選取第 $\lceil \frac{9}{2} \rceil = 5$ 小的元素則為 5，這個元素也是中位數。

我們可以使用時間複雜度為 $O(n \log n)$ 的排序演算法，將 n 個元素排序，然後從排序好的元素中找出第 k 個元素，它就是第 k 小的元素。然而我們也可以使用刪尋解題策略設計選取演算法而得到以下的刪尋選取演算法，可以使用更低的時間複雜度找出第 k 小的元素。

Algorithm Selection(S, n, k)

▷ 刪尋選取演算法

Input: 紿定一個具有 n 個元素的集合 $S = \{a_1, a_2, \dots, a_n\}$ **Output:** 集合 S 中第 k 小的元素步驟 0: 若 $n \leq 5$, 則直接排序所有元素以找出 S 中第 k 小的元素並返回步驟 1: 將 S 分割為 $\lceil \frac{n}{5} \rceil$ 個大小為 5 的子集合。若 n 不能被 5 整除的話則在最後一個子集合內增加值為 ∞ 的元素，使其補滿 5 個元素。

步驟 2: 直接排序每個子集合內的元素，以找出每個子集合的中位數。

步驟 3: 遞迴地找出所有子集合中位數的中位數，令其為 p 步驟 4: 利用 p 將 S 分割成 3 個子集合 S_1, S_2, S_3 ，分別包含小於、等於及大於 p 的元素。步驟 5: 若 $|S_1| > k$ ，則令 $S = S_1$ ，並跳回步驟 1 繼續執行。但是若 $|S_1| + |S_2| > k$ ，則傳回 p 並結束；否則令 $S = S_3$ 與 $k = (k - |S_1| - |S_2|)$ ，並跳回步驟 1 繼續執行。

刪尋選取演算法的步驟 5 是很關鍵的步驟，說明如下：

若 $|S_1| > k$ ，表示 S 中第 k 小的元素在 S_1 中，因此刪除 S_2 和 S_3 (也就是令 $S = S_1$)，並跳回步驟 1 繼續執行。但是若 $|S_1| + |S_2| > k$ ，則 p 就是 S 中第 k 小的元素，則演算法傳回 p 並結束。否則表示 p 在 S_3 中，因此刪除 S_1 和 S_2 (也就是令 $S = S_3$)，並跳回步驟 1 繼續執行尋找 $S = S_3$ 中第 $k' = (k - |S_1| - |S_2|)$ 小的元素。這是因為刪除 S_1 和 S_2 之後， S 中第 k 小的元素，是 S_3 中第 $(k - |S_1| - |S_2|)$ 小的元素。

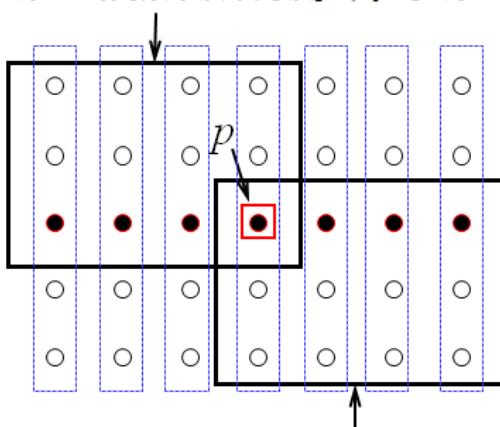
以下，我們以更明確的虛擬碼，不含糊地描述步驟 5:

步驟 5:

```

if  $|S_1| > k$  then
     $S \leftarrow S_1$ ; 跳回步驟 1。
else if  $|S_1| + |S_2| > k$  then
    return  $p$ 
else
     $S \leftarrow S_3$ ;  $k \leftarrow (k - |S_1| - |S_2|)$ ; 跳回步驟 1。

```

S中至少有1/4的元素小於或等於p (S_1 至少有1/4的元素)S中至少有1/4的元素大於或等於p (S_3 至少有1/4的元素)圖 6.1: 找出所有子集合中位數的中位數 p 的示意圖。

以下我們分析刪尋選取演算法的時間複雜度。刪尋選取演算法的步驟 1、步驟 2 及步驟 4 都是 $O(n)$ 複雜度。而圖6.1顯示步驟 3 找出所有子集合中位數的示意圖。在

圖6.1中，虛線代表一個個一 5 個元素組成的子集合，每個子集合各別選出中位數（以實心圓表示），而加上方框的實心圓就是所有子集合中位數的中位數 p 。另外，圖6.1也示意地將 p 所屬的子集合置中，若某個子集合的中位數比 p 小或等於 p ，則此子集合示意地顯示在左側，反之，則示意地顯示在右側。而每個子集合的 5 個元素中有 2 個小於或等於各自的中位數，它們以空心圓表示列在其中位數之上，另 2 個元素則大於或等於各自的中位數，它們也以空心圓表示列在其中位數之下。我們可以由圖6.1看出來，集合 S 中至少有 $1/4$ 的元素（左上粗黑框中的元素）小於或等於 p ，也就是說 S_1 至少包含 $1/4$ 的元素；而 S 中至少有 $1/4$ （右下粗黑框中的元素）的元素大於或等於 p ，也就是說 S_3 也至少包含 $1/4$ 的元素。

因為步驟 5 執行之後，不是回傳 p 為 S 中第 k 小的元素，就是刪除 S_1 和 S_2 或是刪除 S_2 和 S_3 中的元素。因此，刪尋選取演算法在每一次迭代中，都可以刪除至少 $\frac{n}{4}$ 的元素，也就是說在步驟 5 執行之後，最多只會剩下 $\frac{3n}{4}$ 個元素。假設刪尋選取演算法的時間複雜度為 $T(n)$ ，其中 n 是給定集合 S 的元素個數，則每個步驟的時間複雜度如下所列：

- 步驟 1: $O(n)$
- 步驟 2: $O(n)$
- 步驟 3: $T(\frac{n}{5})$
- 步驟 4: $O(n)$
- 步驟 5: $T(\frac{3n}{4})$

因此我們可得: $T(n) = T(\frac{3n}{4}) + T(\frac{n}{5}) + c'n$

令 $T(n) = a_0 + a_1n + a_2n^2 + \dots$, $a_1 \neq 0$ ，則 n 取代為 $\frac{3n}{4}$ ， n 取代為 $\frac{n}{5}$ 及 n 取代為 $\frac{3n}{4} + \frac{n}{5}$ 後分別可得以下三式：

$$T(\frac{3n}{4}) = a_0 + (\frac{3}{4})a_1n + (\frac{9}{16})a_2n^2 + \dots$$

$$T(\frac{n}{5}) = a_0 + (\frac{1}{5})a_1n + (\frac{1}{25})a_2n^2 + \dots$$

$$T(\frac{3n}{4} + \frac{n}{5}) = T(\frac{19n}{20}) = a_0 + (\frac{19}{20})a_1n + (\frac{361}{400})a_2n^2 + \dots$$

我們觀察到：

$$T(\frac{3n}{4}) + T(\frac{n}{5}) \leq a_0 + T(\frac{19n}{20})$$

因此我們可得：

$$T(n) \leq c'n + a_0 + T(\frac{19n}{20}) \leq cn + T(\frac{19n}{20})$$

$$= cn + (\frac{19}{20})cn + T((\frac{19}{20})^2n) = \dots$$

$$= cn + (\frac{19}{20})cn + (\frac{19}{20})^2cn + \dots + (\frac{19}{20})^{p-1}cn + T((\frac{19}{20})^pn)$$

令 $(\frac{19}{20})^pn \leq 5$ ，則 $T((\frac{19}{20})^pn) = c''$ ，我們可得：

$$T(n) \leq cn + (\frac{19}{20})cn + (\frac{19}{20})^2cn + \dots + (\frac{19}{20})^{p-1}cn + c'' \quad (\text{本式含等比級數})$$

$$\leq cn + (\frac{19}{20})cn + (\frac{19}{20})^2cn + \dots + (\frac{19}{20})^{p-1}cn + \dots + c'' \quad (\text{本式含收斂的無窮等比級數})$$

$$\cong cn \frac{1 - (\frac{19}{20})^p}{1 - \frac{19}{20}} + c'' = 20cn + c'' = O(n)$$

6.4 受限最小圓演算法

本節先介紹最小圓 (smallest circle) 問題或稱為最小覆蓋圓 (minimum covering circle) 問題。問題描述如下：給定一個包含 n ($n \geq 2$) 個平面點的集合 S ，找出一個可覆蓋 S 中所有點的最小圓。而受限最小圓 (constrained smallest circle) 問題，則限制圓心 r 必須落在 $y = v$ （例如 $y=3$ ）的直線上，也就是給定一個包含 n ($n \geq 2$) 個平面點的集合 S 與一條直線 $y = v$ ，要找出一個可覆蓋 S 中所有點且圓心在 $y = v$ 上的最小圓，如圖6.2所示。

使用窮舉法 (exhaustive method) 列出每一個可能的候選圓並檢查其是否能夠覆蓋所有的點，可以解決最小圓問題。其作法為任取三點做候選圓（時間複雜度 $O(n^3)$ ）及任取二點為直

徑做候選圓 (時間複雜度 $O(n^2)$)，並針對一個候選圓檢查是否能夠覆蓋所有的點 (時間複雜度 $O(n)$)，藉此找出最小圓。這種做法的總時間複雜度為 $O(n^4)$ 。相同的，窮舉法也可以解決受限最小圓問題，其作法為任選二點做其連線的中垂線並與直線 $y = v$ 求交點，以該交點為圓心，而該交點到二點中任一點距離為半徑作候選圓，再針對每一個候選圓檢查是否能夠覆蓋所有的點，藉此找出受限最小圓。這種做法的總時間複雜度為 $O(n^3)$ 。

以下介紹解決受限最小圓問題的受限最小圓 (constrained smallest circle, CSC) 演算法。它使用刪尋策略解決問題而達到相當低的時間複雜度。

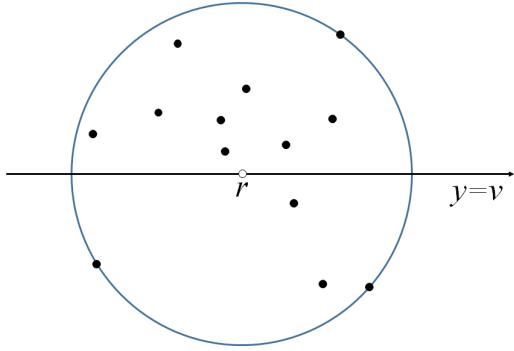


圖 6.2: 受限最小圓問題

Algorithm CSC(S, n, v)

▷ 受限最小圓 (Constrained Smallest Circle) 演算法

Input: $S = \{p_1, p_2, \dots, p_n\}$: 一個包含 n ($n \geq 2$) 個平面點的集合與一直線 $y = v$

Output: 可以覆蓋點集合 S 中所有點的最小圓，其圓心 r 在直線 $y = v$ 上，直徑為 d

步驟 1: 若 $n = 2$ ，則 $S = \{p_1, p_2\}$ ，可以直接求解如下: 若 p_1 及 p_2 都在直線 $y = v$ 上，則 p_1 及 p_2 的中點為圓心 r ； p_1 及 p_2 的距離為直徑 d ；回傳圓心 r 及直徑 d 並結束。否則，求點對 (p_1, p_2) 中垂線 $L_{1,2}$ 與直線 $y = v$ 相交的點 $q_{1,2}$ ，設定 $r = q_{1,2}$ 並求出 p_1 到 $q_{1,2}$ 的兩倍距離 d ；回傳圓心 r 及直徑 d 並結束。

步驟 2: 形成 $\lceil \frac{n}{2} \rceil$ 個點對: 若 n 為偶數，則點對為 $(p_1, p_2), \dots, (p_{n-1}, p_n)$ ；反之，若 n 為奇數，則點對為 $(p_1, p_2), \dots, (p_{n-2}, p_{n-1}), (p_n, p_1)$

步驟 3: 對於每一點對 (p_i, p_{i+1}) ，求其中垂線 $L_{i,j}$ 與直線 $y = v$ 相交的交點 $q_{i,i+1}$ ，共計 $\lceil \frac{n}{2} \rceil$ 個交點。

步驟 4: 尋找這 $\lceil \frac{n}{2} \rceil$ 個交點的 X 座標的中位數 x_m 。

步驟 5: 計算每個點與 x_m 的距離，並令 p_j 為距 x_m 最遠的點。令 q_j 表示 p_j 在直線 $y = v$ 上的投影點，且令其 X 軸值為 x_j 。若 q_j 落在 q_m 左側，也就是 $x_j < x_m$ ，則最佳解 $r = q^*$ 亦必定會落在 q_m 左側，設定 $LR = 0$ 。反之，若 q_j 落在 q_m 右側，也就是 $x_j > x_m$ ，則最佳解 $r = q^*$ 亦必定會落在 q_m 右側，設定 $LR = 1$ 。

步驟 6: 若 $r = q^*$ 在 q_m 的左側 ($LR = 0$)，則針對每個 X 軸值大於 x_m 的交點 $q_{i,i+1}$ ，由 S 中刪除點 p_i 與 p_{i+1} 中靠 q_m 較近的點；反之，若 $r = q^*$ 在 q_m 的右側 ($LR = 1$)，則針對每個 X 軸值小於 x_m 的交點 $q_{i,i+1}$ ，由 S 中刪除點 p_i 與 p_{i+1} 中靠 q_m 較近的點。

步驟 7: 設定 $n = |S|$ 且跳回步驟 1 繼續執行。

以下我們說明刪尋受限最小圓演算法的基本概念。兩個點的到其連線的中垂線等距，我們在步驟一使用這個事實來求出當輸入僅有兩個點時的解答。另外，如圖6.3所示，點對 (p_i, p_{i+1}) 的中垂線 $L_{i,j}$ 將平面一分為二，左半邊離 p_i 較近，而右半邊則是離 p_{i+1} 較近；而因為 $r = q^*$ 在 q_m 的右邊且交點 $q_{i,i+1}$ 在 q_m 的左邊，因此點 p_{i+1} 、 q_m 與 $r = q^*$ 都在中垂線 $L_{i,j}$ 的同一

邊，相較於點 p_i 、點 q_m 與 $r = q^*$ 都離點 p_{i+1} 更近。我們利用這個事實，以 q_m 為基準可以刪除一半點對中離 q_m 比較近的點（也就是離 q^* 比較近的點），因此每個迭代演算法可以刪除 $1/4$ 的點。因為一個以 r 為圓心的圓若已經覆蓋離 r 較遠的點則離 r 較近的點則自然被覆蓋，因此刪除離 r 較近的點不會影響演算法的最後結果。

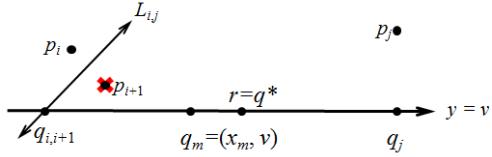


圖 6.3: 受限最小圓演算法刪除一半點對中離 q_m 與 $r = q^*$ 比較近的點

以下分析受限最小圓演算法的時間複雜度。假設刪尋受限最小圓演算法的時間複雜度為 $T(n)$ ，其中 n 是給定點集合 S 中點的個數，則每個步驟的時間複雜度如下所列：

- 步驟 1: $O(1)$
- 步驟 2: $O(n)$
- 步驟 3: $O(n)$
- 步驟 4: $O(n)$
- 步驟 5: $O(n)$
- 步驟 6: $O(n)$
- 步驟 7: $T(\frac{3n}{4})$

步驟 1 為演算法迭代停止條件，其時間複雜度為 $O(1)$ 更具體的可寫為 $T(2)=c_1$ 。步驟 2 到步驟 6，其中包括步驟 4 使用刪尋演算法尋找 $\lceil \frac{n}{2} \rceil$ 個交點的 X 座標的中位數的時間複雜度都是 $O(n)$ ，我們可以將這些步驟的總時間複雜度列為 cn 。另外，在步驟 6 可以 x_m 為基準，刪除一半點對中離 x_m 比較近的點（也就是離 q^* 比較近的點），因此每個迭代演算法可以刪除固定比例的 $1/4$ 的點而剩下 $3/4$ 的點，因此在步驟 7 跳回步驟 1 的遞迴迭代執行的時間複雜度為 $T(\frac{3n}{4})$ 。我們可得：

$$T(n) = T\left(\frac{3n}{4}\right) + cn$$

因為受限最小圓演算法每個迭代都可以刪除固定比例 ($\frac{1}{4} > 0$) 輸入資料而剩下固定比例 ($\frac{3}{4} < 1$) 的輸入資料，因此整個演算法的時間複雜度與每個迭代的時間複雜度具有相同的量級，所以我們可得：

$$T(n) = O(n)$$

6.5 結語

韓非（約西元前 281 生，西元前 233 年卒）是戰國末期的思想家，是中國古代法家思想的代表人物。戰國時期，齊、楚、燕、韓、趙、魏、秦七雄並立。蘇秦曾提出南北合縱六國以抗秦的戰略思想，使秦國十五年無法向西出兵攻擊位於其東邊且南北合縱的六國。但是後來張儀以東西連橫的外交策略，遊說各國由合縱抗秦轉變為連橫親秦，而讓秦國得以遠交近攻，逐漸併吞六國。

韓非在《韓非子·存韓》篇中提到：「諸侯可蠶食而盡」，意思是說秦國可化整為零，也就是將一個整體合縱抗秦的六國化分成許多零散部份，然後像蠶吃桑葉那樣，一點一點的逐步侵佔滅亡六國，而達成一統天下的目的。

本章所介紹的演算法，包括二元搜尋演算法、選取與中位數演算法及限制的一圓心演算法，都採用刪尋解題策略解決問題。這個策略就是類似「化整為零，蠶食而盡」的概念。刪尋演算法包含許多的迭代，每次迭代都嘗試刪除固定比例的輸入資料，而後再遞迴地從剩餘資料中搜尋出解答，若仍然無法找出解答則再刪除固定比例的輸入資料。這樣經過幾次迭代後，輸入資料的規模將會小到足以讓問題在常數時間內直接解決，因而得到問題的解答。一般而言，整個刪尋演算法的時間複雜度與每個迭代的時間複雜度具有相同的量級，因此刪尋演算是一個有效的演算法。

Appendix A

Jeep7

Jeep7 為 Java Editor for Every Programmers v7.0 的簡稱，以 Java 語言所編寫，可以支援 C、C++、Java 與 Python 四種語言，並透過簡潔的中文介面，讓使用者輕易完成四種語言程式的編寫與測試。

以下為 Jeep7 軟體首次執行時的注意事項訊息：

Jeep7 支援 Java/C/C++/Python，請注意以下事項：

1. 進行 Java 程式之編輯、編譯及執行，請先安裝 JDK，並設妥相關 path 參數。
(預設以 JDK 之 javac 進行編譯，path 路徑設定為”JDK 安裝目錄\bin\”)
2. 進行 Python 程式之編輯、編譯及執行，請先安裝 Python 直譯器，並設妥相關 path 參數。
(預設以 Python 直譯器直接執行 python 程式，path 路徑設定為”Python 直譯器安裝目錄”)
3. 進行 C/C++ 程式之編輯、編譯及執行，請先安裝 C/C++ 編譯器，並設妥相關 path 參數。
(預設以 MinGW 之 g++ 進行編譯，path 路徑設定為”MinGW 安裝目錄\bin\”)
4. Jeep7 支援許多快速鍵，請於離開此歡迎對話框後，按下左上方「求助」鈕或鍵盤「F1」鍵以取得詳細資訊。
5. Jeep7 支援「編輯區」對話框滑鼠右鍵彈出選單 (含復原、重作、剪下、複製、貼上、刪除等選項)。
6. Jeep7 支援「尋找」、「取代」、「開舊檔」、「另存檔」等對話框滑鼠右鍵彈出選單 (含複製、貼上選項)。
7. Jeep7 不支援含有空白字元的目錄及檔名，因此，檔案不可儲存於如舊版 Windows 系統之「我的文件夾 (即 My Documents)」或「桌面 (即 Documents and Settings\user\桌面)」等含有空白字元的目錄中，否則於編譯程式時會有錯誤產生。但是 Windows 7 以後的版本則無此問題。
8. 以「新窗執行」功能執行 Java/C/C++/Python 程式時，執行完畢請按新視窗右上角 X 或在新視窗命令提示符號後輸入 exit 並按下 [ENTER] 鍵以關閉該視窗。

Jeep7 下載、安裝與設定:

下載與安裝: 為完整支援 C/C++/Java/Python，使用 Jeep7 必須安裝以下軟體:

1. 安裝最新版 Java SE (Standard Edition) Java Development Kit (JDK): <http://java.sun.com/javase/downloads/index.jsp>
2. 安裝最新版 MinGW C++ Compiler (G++): <http://www.mingw.org>
3. 安裝最新 Python 3 Interpreter: <https://www.anaconda.com>
4. 安裝 Jeep7(Java Editor for Every Programmer v7.0): <http://in2.csie.ncu.edu.tw/jrjiang/Jeep7/Setup&SourceCode.exe>

路徑參數設定:

完成軟體安裝後，還需要設定路徑參數。請注意，這會因為你安裝不同 JDK、MinGW C++ 編譯器及 Python 直譯器軟體的不同版本而不同，也會因為你安裝的作業系統版本不同而不同。以下為針對 Windows 作業系統，安裝上述建議下載軟體的設定:

選擇 [控制台][系統及安全性][系統][進階系統設定][環境變數]，找出 [path] 變數並按下 [編輯]，以在其變數值末端加入:

;C:\Program Files\Java\jdk1.8.0_11\bin;C:\MinGW\bin;C:\Python3;C:\Jeep7
(也就是加入 ;JDK安裝目錄\bin;MinGW安裝目錄\bin;Python3安裝目錄;Jeep7安裝目錄)

測試使用 Jeep7:

以下我們列出四個範例程式 Sample.c、Sample.cpp、Sample.java、Sample.py，讀者可以利用這些範例程式測試使用 Jeep7，若程式均能正常編譯於執行，則代表軟體的安裝與路徑參數均正常。

程式列表 A.1: Sample.c

```
/* File: Sample.c */
#include "stdio.h"
main() {
    printf("Hello, Jeep7!");
} /* End of main() */
```

程式列表 A.2: Sample.cpp

```
//File: Sample.cpp
#include <iostream>
using namespace std;
int main() {
    cout<<"Hello, Jeep7!";
} // End of main()
```

程式列表 A.3: Sample.java

```
//File: Sample.java
class Sample {
    public static void main(String[] args) {
        System.out.println("Hello, Jeep7!");
    } //End of method: main()
} //End of class: Sample
```

程式列表 A.4: Sample.py

```
#File: Sample.py
print('Hello, Jeep7!')
```

Appendix B

ACM ICPC

聲明：本附錄內容部份取材自維基百科 [10]。

B.1 簡介

方法以計算機協會國際大學生程式設計競賽 (Association of Computing Machinery International Collegiate Programming Contest) 簡稱 ACM ICPC 是一個著名的程式設計國際賽事，其參賽就象為大學學生，或精確的說是接受大學教育五年內學生 (students who have had less than five years of university education before the contest)，一般而言，大一到碩一的學生都可參加。

ICPC 競賽可稱為腦力之戰 (Battle of the Brains)，提供學生一個機會，在有限的時間之內，藉由解決精心設計的複雜難題，以鍛鍊和展現其本身解決問題、程式設計，以及團隊合作的能力。除發現和培養計算機科學頂尖學生之外，此競賽亦可促進國際各大學學生之間的交流。

B.2 歷史

競賽的歷史可以上溯到 1970 年，當時在美國德克薩斯農工大學 (The Agricultural and Mechanical College of Texas) 舉辦了首屆比賽，其主辦單位為 the Alpha Chapter of the UPE Computer Science Honor Society。競賽很快得到美國和加拿大各大學的積極響應。1977 年，在計算機學會 (Association of Computing Machinery, 簡稱 ACM) 的計算機科學會議期間舉辦了首次 ACM 主辦的總決賽，並演變成目前的一年一屆的多國參與的國際性比賽。1980 年代，ACM 將競賽的總部設在位於美國德克薩斯州的貝勒大學 (Baylor University)。1989 年建立區域 (regional) 賽制度，區預賽的優勝隊伍才能參加世界總決賽。

1997 年 IBM 開始贊助賽事，自此賽事規模增長迅速，當年總共有來自 560 所大學的 840 支隊伍參加比賽。而到了 2004 年，一共有 840 所大學的 4109 支隊伍參加比賽，並以每年 10-20% 的速度增長。

以下是一些相關的歷史資料：

1. 1991 年亞洲首支隊伍 (台灣交通大學) 參加世界總決賽。
2. 1995 年首度舉辦亞洲區域賽，在台灣舉行，由國立政治大學舉辦。
3. 2002 年上海交通大學獲得中國隊伍首度世界總決賽冠軍。

4. 2005 年上海交通大學第二度獲得世界總決賽冠軍。
5. 2010 年上海交通大學第三度獲得世界總決賽冠軍。
6. 2010 年台灣大學獲得台灣隊伍歷年最好的成績—世界總決賽第三名。
7. 2011 年浙江大學獲得世界總決賽冠軍。
8. 2012 年上海交通大學獲得世界總決賽第四名。
9. 2013 年上海交通大學獲得世界總決賽第二名。
10. 2013 年台灣大學獲得世界總決賽第四名。
11. 2014 年北京大學獲得世界總決賽第三名。
12. 2014 年台灣大學獲得世界總決賽第四名。

B.3 競賽規則

ICPC 分區域賽 (Regional Contest) 與世界賽 (World Final) 兩個階段，區域賽分 6 個賽區，分別為北美 (North America)、南美 (Latin America)、歐洲 (Europe)、非洲 (Africa)、亞洲 (Asia)、南太平洋 (South Pacific)。每年區域賽的日期大約是前一年的九月至十二月，世界賽則是在當年三月至四月舉行。每年各區域賽有若干站區的比賽，根據各賽區規則，每站前若干名的學校自動獲得參加全球總決賽的資格。

ACM-ICPC 以團隊的形式代表各學校參賽，每隊由 3 名隊員組成。一個大學可以有多支隊伍參加區域賽，但只能有一支隊伍參加世界賽。隊員為大學學生，或精確的說是接受大學教育五年內學生 (students who have had less than five years of university education before the contest)，一般而言，大一到碩一的學生都可參加。參加過 5 次區域賽或參加過 2 次世界賽的學生不可再參賽。

比賽期間，每隊使用 1 台電腦，在 5 個小時內使用 C、C++ 或 Java 編寫程式解決 7 到 10 個問題。參賽者可攜帶任何書籍 (包括字典)、手冊、紙本的程式碼。但不可攜帶機器可讀寫的任何軟體或資料，亦不可攜帶自己的電腦、終端機、計算機、電子字典或 PDA，並嚴禁使用行動電話及呼叫器，以免干擾其他隊伍作答。

程式完成之後提交裁判以判定程式運行的結果。每隊在正確完成一題後，將在其位置上升起一隻代表該題顏色的氣球。

判定運行結果可能為：

- 正確: Accepted (AC)
成功解出問題
- 錯誤: Wrong Answer (WA)
程式成功執行結束，但輸出的資料不正確
- 格式錯誤: Presentation Error (PE)
程式輸出資料正確，但格式上有點小誤差，如多了一些空白或跳行等
- 編譯錯誤: Compile Error (CE)
程式編譯時產生錯誤

- 執行時期錯誤: Runtime Error (RE)
程式執行時產生錯誤，如記憶體段錯誤 (segmentation fault) 或浮點數例外 (floating point exception) 等
- 提交錯誤: Submission Error (SE)
提交程式時產生錯誤，如提交程序操作錯誤 (題號、使用者 ID、使用語言沒填好)，或式提交資料毀損等
- 超時: Time Limit Exceeded (TLE)
程式執行超過限定時間 (大部份程式執行時間限定為十秒)
- 超出記憶體使用限制: Memory Limit Exceeded (MLE)
程式使用記憶體超過限制
- 超出輸出限制: Output Limit Exceeded (OLE)
程式產生太多資訊，一般為程式進入無窮迴圈

最後的獲勝隊伍為正確解答題目最多且總用時最少者。每道試題用時將從競賽開始到試題解答被判定為正確為止。其間每一次提交運行結果被判錯誤的話將被加罰 20 分鐘時間，未正確解答的試題不記時。

舉例而言，若有 A 與 B 兩隊都正確完成兩道題目提交。A 隊於比賽開始後 1:00 和 2:45 正確提交兩題 A 隊的總用時為 $1:00+2:45=3:45$ ；B 隊於比賽開始後 1:20 和 2:00 正確提交兩題，但 B 隊有一題提交了 2 次（錯誤一次），B 隊總用時為 $1:20+2:00+0:20=3:40$ 。最後 B 隊以總用時少而獲勝。

B.4 參賽獲勝秘訣

B.4.1 獲勝秘訣一

參賽獲勝秘訣由熟悉比賽使用之各項軟硬體環境開始，如下所列：

- 熟悉比賽使用之作業系統：
2014 World Final 使用的作業系統為 Fedora Core 4 Linux，桌面為 GNOME，編輯器為 vi/vim、gvim、emacs、gedit
- 熟悉比賽使用之語言版本：
2014 World Final 使用的語言為 Java: Version 1.7.0_25; OpenJDK Runtime Environment (IcedTea 2.3.10 - 7u25-2.3.10-1ubuntu0.12.04.2, OpenJDK 64-Bit Server VM (build 23.7-b01, mixed mode
C/C++: GCC 4.6.3 (Ubuntu/Linaro 4.6.3-1ubuntu5)
- 熟悉比賽使用之整合開發環境：
2014 World Final 使用之整合開發環境為 Java: Eclipse 4.3.1
C/C++: CDT 8.2.1 under Eclipse 4.3.1
- 熟悉比賽使用之裁判程式：
Kattis Contest Control System
由瑞典皇家理工學院（英文名 Royal Institute of Technology，瑞典語縮寫為 KTH）所發展與維護。（在 2008 年之前的 world 賽均採用沙加緬度加州州立大學 (California State University, Sacramento) 所開發的 PC² (Programming Contest Control) System，現今仍有許多人以 PC² 做為比賽訓練之用。）

B.4.2 獲勝秘訣二

由於 ICPC 賽是採用團體方式進行，因此應該培養團隊默契，相互了解隊員彼此的長處與短處，定好分工方式。因為 ICPC 賽事均以英文出題，因此至少一位隊員應積極培養英文閱讀能力。

並建議先立定得分策略，分工瀏覽全部試題，挑出有把握的題目，集中全力解決這些題目。若手邊仍有一些可能解出的題目，則可以儘早放棄履遇挫折的題目。

B.4.3 獲勝秘訣三

也建議使用常用的函式庫以簡化程式設計並加速程式執行。例如，C++ 語言除了提供 C 語言的標準函式庫之外，還提供標準樣板函式庫 (Standard Template Library, STL)。STL 包含三個部份：容器函式庫 (containers library)、演算法函式庫 (algorithms library) 及迭代子函式庫 (iterators library)。分別說明如下：

1. 容器函式庫 (containers library): 提供不同的資料結構，方便程式設計者使用，例如 vector、list、stack、set, hash_map, map, queue 及 priority_queue 等。
2. 演算法函式庫 (algorithms library): 提供常用的資料結構操作，方便程式設計者使用，例如 find、sort、count 及 copy 等。
3. 迭代子函式庫 (iterators library): 提供迭代中常用資料結構元素存取操作，方便程式設計者使用，例如 begin 及 end 等。

我們舉以下 C++ 程式片段為例來看 STL 的用法：

程式列表 B.1: STL.cpp

```
//C++ STL用法範例
#include <iostream>
#include <string>
#include <vector>
#include <stack>
#include <set>
#include <hash_map>
#include <map>
#include <queue>
#include <algorithm>
using namespace std;
stack<int> S;
set<string> T;
map<int, string> M;
queue<char> Q;
priority_queue<int> PQ;
vector<float> V; //產生一個空的元素為float型別的vector V
int main() {
V.push_back(3.21); //將浮點數3.21加到V的末端
V.push_back(1.23); //將浮點數1.23加到V的末端
V.push_back(2.31); //將浮點數2.31加到V的末端
cout << V[0] << endl; //將第1個元素(編號為0)的值印出
cout << V[1] << endl; //將第2個元素(編號為1)的值印出
cout << V[2] << endl; //將第3個元素(編號為2)的值印出
vector<float>:: iterator it; //產生一個迭代子it
it = V.begin(); //將it指向第一個元素
cout << *it << endl; //印出迭代子所指向的元素(也就是第一個元素)
cout << V.front() << endl; //印出第一個元素
cout << V.back() << endl; //印出最後一個元素
sort(V.begin(),V.end()); //V.end()代表超過最後一個元素之後的迭代子，代表至此已無元素
//還有stable_sort, partial_sort, partial_sort_copy, nth_element, is_sorted, partition,
//stable_partition等相關的函式可用。
for (it=V.begin();it!=V.end();it++) //此for迴圈可以印出所有元素
```

```

    cout << *it << endl;
V.erase(V.begin()); //將第一個元素刪除
cout << V.size() << endl; //印出元素的個數
//stack: S.push(), S.top(), S.pop(), S.empty()
//注意: top()傳回但不去除堆疊頂端元素;pop()去除但不傳回堆疊頂端元素;stack使用鏈結(linked)方式實作堆
疊因此堆疊永遠不會滿
//queue - Q.front(), Q.back(), Q.push(), Q.pop(), and Q.empty()
//hash_map - H.erase(), H.find(), H.insert() //hash_map使用hash table儲存
//map - M.erase(), M.find(), M.insert() //map使用red-black tree儲存
//priority_queue - PQ.push(), PQ.top(), PQ.pop(), PQ.empty()
//set - T.insert(), T.erase(), T.size(), T.clear(), T.find(), set_union, set_intersection,
      set_difference
} // End of main()

```

B.4.4 獲勝秘訣四

多透過 Online Judge 進行題目練習，以下列出一些不錯的 Online Judge:

- Universidad de Valladolid Online Judge
- Ural State University Online Judge
- Tianjin University Online Judge
- Saratov State University Online Judge
- Sphere Online Judge
- ACM-ICPC Live Archive Around the World
- MIPT Online Judge
- Peking University Online Judge
- Zhejiang University Online Judge
- Harbin Institute of Technology Online Judge
- Fuzhou University Online Judge
- Online Problems Solving System

B.5 ICPC 題目範例

以下我們舉一個 ICPC 比賽中曾經出現過的題目為例，說明題目的形式、如何解答題目與如何透過 Universidad de Valladolid Online Judge (UVa online Judge) 網站練習遞交題目。

B.5.1 題目內容

以下為範例題目內容: Vito' s Family

- **Background**

The world-known gangster Vito Deadstone is moving to New York. He has a very big family there, all of them living in Lamafia Avenue. Since he will visit all his relatives very often, he is trying to find a house close to them.

- **Problem**

Vito wants to minimize the total distance to all of them and has blackmailed you to write a program that solves his problem.

- **Input**

The input consists of several test cases. The first line contains the number of test cases. For each test case you will be given the integer number of relatives $r(0 < r < 500)$ and the street numbers (also integers) where they live ($0 < s_i < 30000$). Note that several relatives could live in the same street number.

- **Output**

For each test case your program must write the minimal sum of distances from the optimal Vito's house to each one of his relatives. The distance between two street numbers s_i and s_j is $d_{ij} = |s_i - s_j|$.

- **Sample Input**

```
2
2 2 4
3 2 4 6
```

- **Sample Output**

```
2 4
```

B.5.2 題目中文翻譯

以下為「Lucky 貓的 UVA (ACM) 園地」[?] 針對以上問題的中文翻譯:

維托家族:

- **背景與問題**

世界聞名的黑社會老大 Vito Deadstone 要搬到紐約來了。在那裡他有一個大家族，並且他們都住在 Lamafia 大道上。因為 Vito 時常要拜訪所有的親戚，他想要找一間離他們最近的房子，也就是說他希望從他的家到所有的親戚的家的距離的和為最小。他恐嚇你寫一個程式來幫助幫助他解決這個問題。

- **Input**

輸入的第一列有一個整數代表以下有多少組測試資料。每組測試資料一列，第一個整數 $r(0 < r < 500)$ 代表他親戚的數目。接下來的 r 個整數 s_1, \dots, s_r 為這些親戚房子的門牌號碼 ($0 < s_i < 30000$)。注意，有些親戚的門牌號碼會相同。

- **Output**

對每一組測試資料，輸出從他的新家到所有的親戚的家的距離的和為最小為多少。2 個門牌號碼 s_i 、 s_j 的距離 d_{ij} 為 $s_i - s_j$ 的絕對值 ($d_{ij} = |s_i - s_j|$)。

- **Sample Input**

```
2
2 2 4
3 2 4 6
```

- **Sample Output**

```
2 4
```

B.5.3 解題建模

Vito's Family 這個題目可以如以下方式建模 (modeling) · 也就是以抽象的方式重新描述:

給定一個具有 n 個元素的陣列 A · 找出 k 使得 $S = \sum_{k=0, \dots, n-1} |A[i] - A[k]|$ 最小 · 最後輸出 S 。

想法: 找出陣列的中位數 (median) $A[k]$ · 再計算 $S = \sum_{k=0, \dots, n-1} |A[i] - A[k]|$ 並輸出 S 即可得到解答。

以下是含中文變數的 Java 程式 · 先將所有陣列中的元素排序 · 然後找出中位數 · 再計算所有元素與中位數的差的總和就可以解決問題。請注意 · 當 n 為偶數時 · 中位數為 $A[n/2]$ · 當 n 為奇數時 · 中位數為 $(A[(n-1)/2] + A[(n+1)/2])/2$ 。不過因為題目的輸出為有元素與所選整數的差的總和 · 因此我們不論選中位數、 $A[(n-1)/2]$ 或選 $A[(n+1)/2]$ 所得到的結果都相同。當 n 為偶數時 · $A[(n+1)/2] = A[n/2]$ · 因此 · 我們在程式中就以 $A[n/2]$ 為所選元素 · 如此不管 n 為奇數或是偶數都可以產生正確解答。

程式列表 B.2: Vito.java

```
import java.util.*;
class Vito {
    static int[] 門牌=new int[500];
    static Scanner 輸入 = new Scanner(System.in);
    public static void main(String[] 參數) {
        int 狀況數 = 輸入.nextInt();
        for(int i=1;i<=狀況數;++i)
            System.out.println(處理狀況並回傳答案());
    } //End of method: main()
    static long 處理狀況並回傳答案() {
        long 總距離=0;
        int 最佳門牌;
        int 人數 = 輸入.nextInt();
        for (int i=0;i<人數;++i)
            門牌[i]= 輸入.nextInt();
        for (int i=人數-1;i>=1;--i)
            for (int j=0;j<=i-1; ++j)
                if(門牌[j]>門牌[j+1]) {
                    int t=門牌[j];
                    門牌[j]=門牌[j+1];
                    門牌[j+1]=t;
                }
        最佳門牌=門牌[人數/2];
        for (int i=0;i<人數;++i)
            總距離+=Math.abs(門牌[i]-最佳門牌);
        return 總距離;
    } //End of method: 處理狀況並回傳答案()
} //End of class: Vito
```

有些 Judge 不能接受使用中文變數的 Java 程式 · 而 UVa Judge 更需要將主 class 名稱改為 Main 才可以正常執行 · 因此我們一併進行以下兩個修改: (1) 使用英文變數名稱。(2) 將主 class 名稱改為 Main 。我們並使用 Java 內建類別 Arrays 的 sort 方法以簡化程式設計並加速程式執行。

程式列表 B.3: Vito1.java

```
import java.util.*;
class Vito1 { //當你使用UVa Judge時，主class名稱要改為Main才可以正常執行
    static int[] House=new int[500];
    static Scanner input = new Scanner(System.in);
    public static void main(String[] args) {
        int cases = input.nextInt();
        for(int i=1;i<=cases;++i)
            System.out.println(result());
    } //End of method: main()
```

```

static long result() {
    long distance=0; //total: the summation of distances from Vito's house to others'
    int best; //best: the best street number for Vito to stay
    int total = input.nextInt(); //total: the total number of relatives
    for (int i=0;i<total;++i)
        House[i]= input.nextInt();
    Arrays.sort(House,0,total); //sort the House array from index 0 to index total (exclusive)
    best=House[total/2];
    for (int i=0;i<total;++i)
        distance+=Math.abs(House[i]-best);
    return distance;
} //End of method: result()
} //End of class: Vito

```

以下我們使用 C++ 語言，搭配 C++ STL 的 vector 集合變數 (collection variable) 解題。我們並使用執行速度比 sort 函式還快的 nth_element 函式來找出中位數。透過使用nth_element(begin, begin+n, end) 函式呼叫，可以使排序為 n 的元素處於位置 n (排序從 0 開始計算)，並且比這個元素小的元素都排在這個元素之前，比這個元素大的元素都排在這個元素之後，但是這個元素前後的元素並未依序排列。而透過使用sort(begin, end) 函式呼叫，可以將所有的元素依序排序，因此排序為 n 的元素自然會處於位置 n 。一般而言，排序演算法的時間複雜度為 $O(s \log s)$ ，而 nth_element 函式的時間複雜度為 $O(s)$ ，其中 s 為輸入規模 (input size)，也就是需要處理的元素總個數。(一般輸入規模使用 n 來表示，在此處我們使用 s 來表示以避免與 nth_element 函式中的 n 產生混淆。)

程式列表 B.4: Vito.cpp

```

#include <iostream>
#include <vector>
#include <algorithm>
#include <math.h>
using namespace std;
vector<int> House; //House: a vector (container) of elements of the int type
long int result();
int main() {
    int cases; //cases: the number of input cases
    cin>>cases;
    for(int i=1;i<=cases;++i)
        cout<<result()<<endl;
    return 0;
} // End of main()
long int result() {
    long int distance=0; //distance: the summation of distances from Vito's house to others'
    int best; //best: the best street number for Vito to stay
    int total; //total: the total number of relatives
    int temp; //temp: a temporary variable for input
    cin>>total;
    House.clear();
    for (int i=0;i<total;++i)
        {cin>>temp; House.push_back(temp);}
    nth_element(House.begin(), House.begin()+total/2, House.end());
    //也可以用 sort(House.begin(),House.end());
    best=House[total/2];
    for (int i=0;i<total;++i)
        distance+=(long int)fabs(House[i]-best);
    return distance;
} // End of result()

```

B.5.4 解題過程

以下我們展示透過 UVa Judge 解答 Vito's Family 題目之過程。

首先進入 UVa Judge 首頁 (<http://uva.onlinejudge.org/>) 並註冊一個帳號，畫面如下：

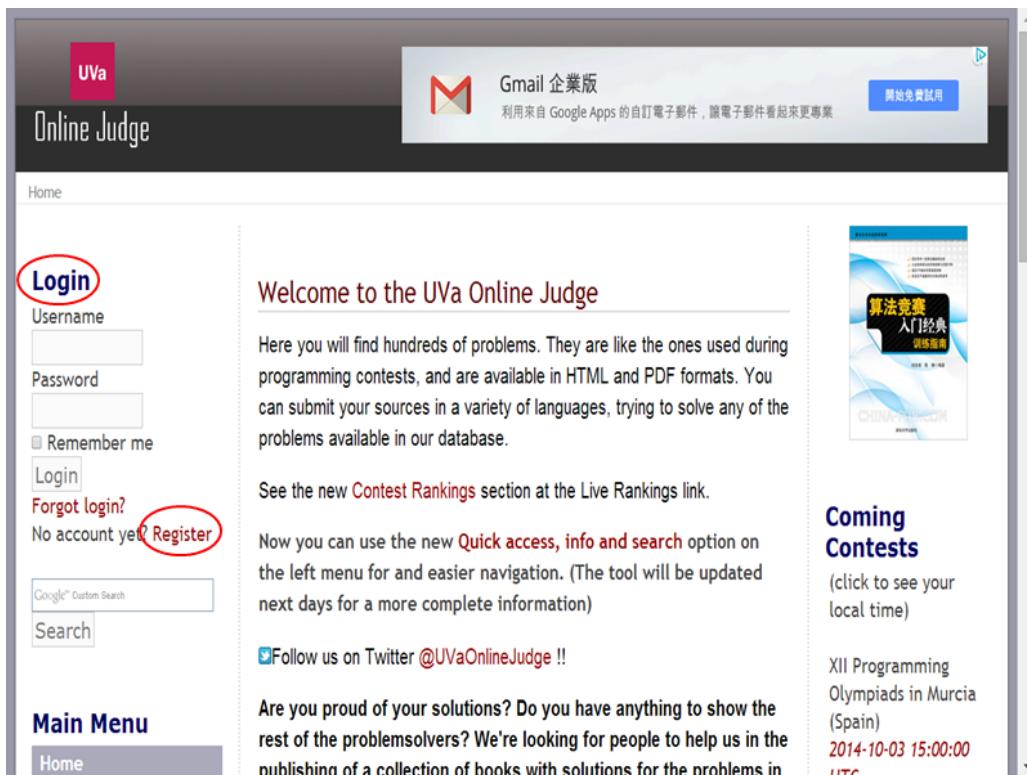


圖 B.1: UVa Online Judge 畫面 (1)

註冊帳號之後都以這個帳號登錄進行程式設計練習，畫面如下：

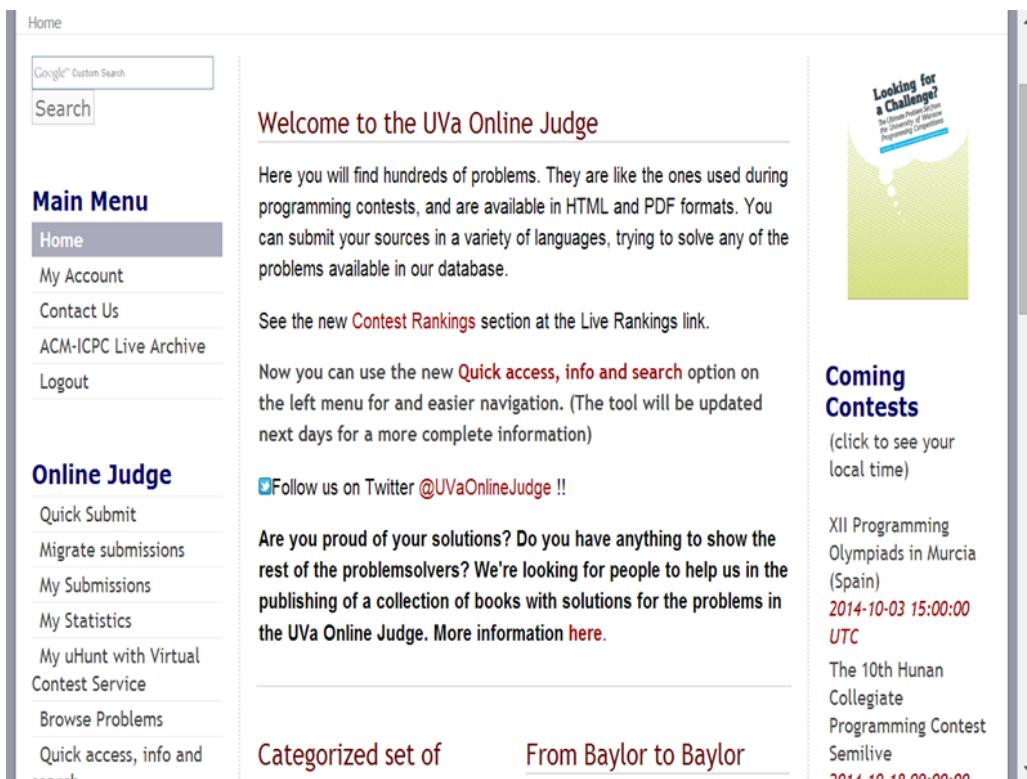


圖 B.2: UVa Online Judge 畫面 (2)

選擇左邊欄位的「Browse Problems」以瀏覽問題，畫面如下：

The screenshot shows the 'Browse Problems' section of the UVa Online Judge. On the left, there's a 'Main Menu' and 'Online Judge' sidebar. The 'Browse Problems' link in the 'Online Judge' menu is highlighted with a red oval. The main content area displays a table titled 'Root' with columns for 'Title', 'Total Submissions / Solving %', and 'Total Users / Solving %'. The table lists various problem sets and contests, such as 'Problem Set Volumes (100...1999)', 'Contest Volumes (10000...)', and 'ACM-ICPC World Finals'. The URL at the bottom is uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8.

圖 B.3: UVa Online Judge 畫面 (3)

選擇 Title 為「Contest Volumes (10000...)」以瀏覽問題，畫面如下：

This screenshot shows the 'Contest Volumes (10000...)' category page. The 'Main Menu' and 'Online Judge' sidebar are visible. The 'Browse Problems' link in the 'Online Judge' menu is highlighted with a red oval. The main content area displays a table titled 'Root :: Contest Volumes (10000...)'. The first item in the table, 'Volume 100 (10000-10099)', is also circled in red. The URL at the bottom is uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=12.

圖 B.4: UVa Online Judge 畫面 (4)

選擇「Volume 100(10000-10099)」瀏覽題目，畫面如下：

Home > Browse Problems

Google Custom Search

Search

Main Menu

- Home
- My Account
- Contact Us
- ACM-ICPC Live Archive
- Logout

Online Judge

- Quick Submit
- Migrate submissions
- My Submissions
- My Statistics
- My uHunt with Virtual Contest Service
- Browse Problems**
- Quick access, info and search

Root :: Contest Volumes (10000...) :: Volume 100 (10000-10099)

| Title | Total Submissions / Solving % | Total Users / Solving % |
|---------------------------------|-------------------------------|-------------------------|
| ✓ 10000 - Longest Paths | 26654 21.84% | 6269 60.34% |
| ✓ 10001 - Garden of Eden | 4696 34.24% | 1437 63.87% |
| ✓ 10002 - Center of Masses | 9634 25.76% | 1980 66.01% |
| ✓ 10003 - Cutting Sticks | 26058 33.45% | 7640 87.97% |
| ✓ 10004 - Bicoloring | 43216 45.28% | 12029 88.39% |
| ✓ 10005 - Packing polygons | 7997 16.41% | 1888 47.56% |
| ✓ 10006 - Carmichael Numbers | 28895 36.68% | 7406 83.28% |
| ✓ 10007 - Count the Trees | 6520 39.91% | 2561 72.43% |
| ✓ 10008 - What's Cryptanalysis? | 24674 56.63% | 11411 93.49% |
| ✓ 10009 - All Roads Lead Where? | 10343 30.42% | 2925 76.72% |
| ✓ 10010 - Where's Waldorf? | 29093 28.01% | 7327 77.55% |
| ✓ 10011 - Where Can You Hide? | 3944 10.14% | 538 39.22% |
| ✓ 10012 - How Big Is It? | 8306 27.40% | 1961 66.24% |
| ✓ 10013 - Super long sums | 42999 25.87% | 8145 62.69% |
| ✓ 10014 - Simple calculations | 9459 30.51% | 3192 68.80% |

圖 B.5: UVa Online Judge 畫面 (5)

向下捲動頁面瀏覽 Contest Volumes (10000...) 題目，畫面如下：

search

Problemsetters' Credits

Live Rankings

Site Statistics

Contests

Electronic Board

Additional Information

Other Links

With the collaboration of

Universidad de Valladolid

acm icpc

Parque Científico Universidad de Valladolid

Fundación General Universidad de Valladolid

EduJUDGE

Root :: Contest Volumes (10000...) :: Volume 100 (10000-10099)

| | | |
|--|--------------|--------------|
| ✓ 10015 - Joseph's Cousin | 7130 25.06% | 2853 45.99% |
| ✓ 10016 - Flip-Flop the Squarelotron | 2325 12.45% | 955 75.08% |
| ✓ 10017 - The Never Ending Towers of Hanoi | 4698 19.80% | 1433 46.27% |
| ✓ 10018 - Reverse and Add | 68521 41.24% | 16951 86.14% |
| ✓ 10019 - Funny Encryption Method | 12568 69.36% | 7783 95.55% |
| ✓ 10020 - Minimal coverage | 15068 28.01% | 3015 77.15% |
| ✓ 10021 - Cube in the labirint | 1543 21.71% | 428 65.12% |
| ✓ 10022 - Delta-wave | 2266 28.55% | 866 62.36% |
| ✓ 10023 - Square root | 23454 8.44% | 4440 23.31% |
| ✓ 10024 - Curling up the cube | 1908 22.48% | 586 52.73% |
| ✓ 10025 - The ?1?2?...?n=k problem | 12662 25.18% | 3378 72.94% |
| ✓ 10026 - Shoemaker's Problem | 21017 30.43% | 5137 78.33% |
| ✓ 10027 - Language Cardinality | 2351 12.76% | 456 51.54% |
| ✓ 10028 - Demerit Points | 1272 22.96% | 331 55.59% |
| ✓ 10029 - Edit Step Ladders | 11706 17.92% | 1736 55.01% |
| ✓ 10030 - Computer Dialogue | 827 27.57% | 305 41.54% |
| ✓ 10031 - Saskatchewan | 1134 6.53% | 234 23.08% |
| ✓ 10032 - Tug of War | 28085 9.94% | 3239 27.82% |

圖 B.6: UVa Online Judge 畫面 (6)

持續向下瀏覽 Contest Volumes (10000...) 題目，畫面如下：

The screenshot shows a table of UVa Online Judge problems. Each row contains a problem ID, name, number of submissions, acceptance rate, and a progress bar. The progress bar consists of a green segment followed by a red segment. The last row, '10041 - Vito's Family', is highlighted in green.

| | | | |
|---------|----------------------------|--------|--------|
| ✓ 10033 | - Interpreter | 29455 | 21.89% |
| ✓ 10034 | - Freckles | 29205 | 29.80% |
| ✓ 10035 | - Primary Arithmetic | 74226 | 28.42% |
| ✓ 10036 | - Divisibility | 9106 | 35.84% |
| ✗ 10037 | - Bridge | 16812 | 20.44% |
| ✓ 10038 | - Jolly Jumpers | 110976 | 25.94% |
| ✓ 10039 | - Railroads | 3974 | 22.62% |
| ✓ 10040 | - Ouroboros Snake | 3072 | 36.72% |
| ✓ 10041 | - Vito's Family | 42858 | 39.14% |
| ✓ 10042 | - Smith Numbers | 13890 | 32.81% |
| ✓ 10043 | - Chainsaw Massacre | 4958 | 26.66% |
| ✓ 10044 | - Erdos Numbers | 22801 | 15.65% |
| ✓ 10045 | - Echo | 2096 | 23.52% |
| ✓ 10046 | - Fold-up Patterns | 121 | 21.49% |
| ✓ 10047 | - The Monocycle | 5788 | 10.24% |
| ✓ 10048 | - Audiophobia | 11106 | 42.41% |
| ✓ 10049 | - Self-describing Sequence | 6025 | 32.08% |
| ✓ 10050 | - Hartals | 19845 | 57.37% |
| ✗ 10051 | - Tower of Cubes | 6250 | 33.09% |
| ✗ 10052 | - Inviting Politicians | 2107 | 8.21% |

圖 B.7: UVa Online Judge 畫面 (7)

選擇題目「10041 - Vito's Family」，畫面如下：

The screenshot shows the details for problem 10041 - Vito's Family. It includes a main menu, search bar, and navigation links. The problem title is '10041 - Vito's Family' with a time limit of 3.000 seconds. A large blue button labeled 'Problem C: Vito's family' is prominent. Below it are sections for 'Background' and 'Problem'. The 'Background' section describes Vito Deadstone's move to New York and his desire to find a house close to his relatives. The 'Problem' section states that Vito wants to minimize the total distance to all relatives and has blackmailed the user to write a program for him. There is also an 'Input' section.

圖 B.8: UVa Online Judge 畫面 (8)

按下右上角「Submit」準備上傳程式，畫面如下：



圖 B.9: UVa Online Judge 畫面 (9)

將 Vito.java 剪貼至文字區中，選擇程式語言為 Java，畫面如下：



圖 B.10: UVa Online Judge 畫面 (10)

按下「Submit」上傳程式，我們可以看到一個 Submission ID，畫面如下：

Submission received with ID 14105841

Root :: Contest Volumes (10000...) :: Volume 100 (10000-10099)

10041 - Vito's Family

Time limit: 3.000 seconds

Problem C: Vito's family

Background

The world-known gangster Vito Deadstone is moving to New York. He has a very big family there, all of them living in Lamafia Avenue. Since he will visit all his relatives very often, he is trying to find a house close to them.

Problem

Vito wants to minimize the total distance to all of them and has blackmailed you to write a program that solves his problem.

Input

圖 B.11: UVa Online Judge 畫面 (11)

選擇左邊欄位的「My Submissions」可以看到所有上傳程式提交後的判斷結果，畫面如下：

My Submissions

| # | Problem | Verdict | Language | Run Time | Submission Date |
|----------|---------------------|---------------|----------|----------|---------------------|
| 14105841 | 10041 Vito's Family | Runtime error | JAVA | 0.000 | 2014-08-28 02:31:38 |
| 14105823 | 10041 Vito's Family | Runtime error | JAVA | 0.000 | 2014-08-28 02:26:04 |
| 14105734 | 10041 Vito's Family | Accepted | C++11 | 0.049 | 2014-08-28 01:35:31 |
| 14105723 | 10041 Vito's Family | Accepted | C++11 | 0.052 | 2014-08-28 01:28:42 |
| 14105704 | 10041 Vito's Family | Wrong answer | C++11 | 0.075 | 2014-08-28 01:22:22 |
| 14105672 | 10041 Vito's Family | Wrong answer | C++11 | 0.076 | 2014-08-28 01:14:23 |
| 14105667 | 10048 Audiophobia | Wrong answer | C++11 | 0.015 | 2014-08-28 01:13:11 |
| 14105648 | 10041 Vito's Family | Wrong answer | C++ | 0.079 | 2014-08-28 01:08:43 |
| 14105538 | 10041 Vito's Family | Wrong answer | C++11 | 0.082 | 2014-08-28 00:37:19 |
| 14105532 | 10041 Vito's Family | Accepted | JAVA | 0.998 | 2014-08-28 00:35:31 |

圖 B.12: UVa Online Judge 畫面 (12)

剛剛提交的 Vito.java 程式的提交判斷結果為”Runtime errors”，其原因是我們沒有將主要 class 的名稱改為 Main。另外，有些 online judge(包括早期的 UVa Online Judge) 不能接受 Java 程式中含有以中文命名的變數，因此我們最好使用以英文命名的變數。

Vito1.java 程式使用以英文命名的變數，並在上傳前將主要 class 的名稱改為 Main，而其提交判斷結果為”Accepted”。另外，Vito.cpp 的提交判斷結果也是”Accepted”，而且其執行時間相對的短很多。以下畫面的最上面三項，由上而下分別為 Vito.cpp、Vito1.java、Vito.java 的提交的判斷結果：

The screenshot shows the 'My Submissions' section of the UVa Online Judge. The table lists 13 submissions, with the first three highlighted by a red rounded rectangle. The columns are: #, Problem, Verdict, Language, Run Time, and Submission Date.

| # | Problem | Verdict | Language | Run Time | Submission Date |
|----------|---------------------|---------------|----------|----------|---------------------|
| 14105872 | 10041 Vito's Family | Accepted | C++11 | 0.045 | 2014-08-28 02:39:13 |
| 14105866 | 10041 Vito's Family | Accepted | JAVA | 0.956 | 2014-08-28 02:38:44 |
| 14105841 | 10041 Vito's Family | Runtime error | JAVA | 0.000 | 2014-08-28 02:31:38 |
| 14105823 | 10041 Vito's Family | Runtime error | JAVA | 0.000 | 2014-08-28 02:26:04 |
| 14105734 | 10041 Vito's Family | Accepted | C++11 | 0.049 | 2014-08-28 01:35:31 |
| 14105723 | 10041 Vito's Family | Accepted | C++11 | 0.052 | 2014-08-28 01:28:42 |
| 14105704 | 10041 Vito's Family | Wrong answer | C++11 | 0.075 | 2014-08-28 01:22:22 |
| 14105672 | 10041 Vito's Family | Wrong answer | C++11 | 0.076 | 2014-08-28 01:14:23 |
| 14105667 | 10048 Audiophobia | Wrong answer | C++11 | 0.015 | 2014-08-28 01:13:11 |
| 14105648 | 10041 Vito's Family | Wrong answer | C++ | 0.079 | 2014-08-28 01:08:43 |

圖 B.13: UVa Online Judge 畫面 (13)

Bibliography

- [1] American National Standards Institute and Business Equipment Manufacturers Association (U.S.). *Flowchart Symbols and Their Usage in Information Processing*. ANSI X3.5-1970. ANSI, 1970.
- [2] Ned Chapin. Flowchart. In *Encyclopedia of Computer Science*, pages 714–716. John Wiley and Sons Ltd., Chichester, UK.
- [3] John William Joseph Williams. Algorithm 232: heapsort. *Commun. ACM*, 7:347–348, 1964.
- [4] Charles AR Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.
- [5] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.
- [6] J. Dongarra and F. Sullivan. Guest editors introduction to the top 10 algorithms. *Computing in Science Engineering*, 2(1):22–23, Jan 2000.
- [7] 維基百科. 計算幾何—維基百科, 自由的百科全書, 2019.
- [8] 維基百科. 尤利烏斯·凱撒—維基百科, 自由的百科全書, 2019.
- [9] 王海林. 軍事趣味閱讀. 崑博出版, 2018.
- [10] 維基百科. 國際大學生程序設計競賽 — 維基百科, 自由的百科全書, 2019.