第 2 章　演算法簡介

能對電腦解釋清楚的東西叫做科學，其他的東西叫做藝術。

– Donald Knuth (高德納)

2.1.　演算法起源

演算法是數學裡隨著電腦的興起被擴大研究的領域， 而演算法本身卻可以追溯到古代。早在古希臘時期， 爲了解決各種算術上的問題就已經有許多不同的演算法， 包括至今在數論上仍然重要的 Euclid 演算法、即輾轉相除法。 演算法在中國古代文獻中稱爲「術」， 最早出現在《周髀算經》及《九章算術》， 例如《九章算術》給出四則運算、最大公約數、 最小公倍數、 開平方根、 開立方根、 求質數的篩法、 線性方程組求解的演算法。 三國時代的劉徽給出求圓周率的演算法： 劉徽割圓術。 自唐代以來， 歷代更有許多專門論述「算法」的專著： 唐代的《一位算法》一卷、《算法》一卷， 宋代的《算法緒論》一卷、《算法秘訣》一卷、 楊輝的《楊輝算法》， 元代的《丁巨算法》， 明代程大位的《算法統宗》， 清代的《開平算法》、《算法一得》、《算法全書》。 而英文名稱 Algorithm 來自於 9 世紀波斯數學家花拉子米 （比阿勒·霍瓦里松， 拉丁轉寫 al-Khwarizmi）， 因爲比阿勒·霍瓦里松在數學上提出了演算法這個概念； al-Khwarizmi 後來音轉爲 algorism， 在 18 世紀演變成 algorithm。

簡單來說， 演算法就是一個有限而明確定義的指令清單， 用途是要針對一個給定的輸入， 在有限的步驟內輸出解決特定問題的答案或特定工作的結果。 第 1 章有幾個跟圖論相關的演算法的例子， 例如用來尋找 Euler 迴路的演算法、 尋找 de Bruijn 序列的演算法、 以及判斷圖序列的演算法等。 圖論是一門具有高度應用性的數學分支， 爲了能夠將理論拿來解決實際的問題， 必須設計出各種對應的演算法， 其難度絕不亞於理論本

身。 演算法除了要求正確之外， 更希望它能快速運作而且節省記憶體空間。 有很多圖論問題至今仍然沒有辦法找到充分有效率的演算法。

不僅圖論的理論可以用來產生演算法， 反過來演算法也經常被用來證明各種圖論的定理， Euler 定理的證明就是一個例子， 在往後的章節裡將會看到更多這樣的例子。 由於這是一本談論圖論以及相關演算法的書籍， 在更進一步介紹各種相關理論之前， 這一章要先介紹一些關於演算法的基本概念、 以及當我們要將演算法實際應用在圖論問題上時所需要注意的課題。

2.2. 演算法的複雜度

演算法最主要關切的課題有兩個， 一個是問題的**演算法可解性**（computablility）， 另一個是演算法的**計算複雜度分析**（computational complexity analysis）。

第一個問題所探究的是， 給定一類型的問題， 是否存在一個演算法可以解決這個類型當中的所有問題？ 早期的數學家很快就意識到這並非永遠可能。 最基本的例子就是所謂的**停機問題**（halting problem）， 摘要而言， 這個問題問的是， 存不存在一個演算法能夠判斷一個任意給定的程式是否總是會終止 （有些程式基於設計上的錯誤， 可能永遠不會停止）， 可以證明這樣的演算法是不可能存在的， 細節請參閱介紹演算法的書籍。 另外一個經典的例子是 1970 年的 Matiyasevich 定理， 他證明了沒有一個演算法可以判斷任意的 Diophantus 方程 （即數論中的整係數方程求整數解）是否有解。 在這兩個例子中， 這些結果並不意味著原問題無解， 只表示不存在可以解它們的演算法， 也就是說， 就算它們可解， 也會需要無限多的方法才能夠解決該類問題中的每一個問題； 而這是演算法辦不到的， 因爲根據定義， 一個演算法只能含有有限條指令。 通常，要證明一個問題是演算法可解， 辦法就是實際構造出一個演算法、 並且證明它能解決該問題（不過偶爾也有純粹只是存在性的證明）。

至於複雜度分析， 著重的在於一個演算法需要用多少時間和空間（記憶體用量）才能解決對應的問題。 演算法除了確實能夠解決問題之外， 還必須要在能夠被接受的時間及空間的限制下解決問題。 一般比較在著重時間複雜度的分析， 因爲空間複雜度的分析相對比較容易。 複雜度一般必須要從三個層次來討論， 一個是問題本身、 一個是演算法、 最後是演算法的**實作**（implementation）。

一個**問題**（problem）可能包含一個待回答的疑問， 一個待滿足的要求， 或是一個待求得的最佳可能情況或架構、 即問題的**解答**（solution）。 問題本身可能會含有一些尚未定值的**參數**（parameter）或**變數**（variable）， 藉由設定這些參數以得到問題的不同**範例**（instance）； 舉例來說， 假設要解決的問題是「判斷一個程式 P 會不會停止」， 設定其中的參數 P =「無窮迴圈程式」的時候， 就得到了問題的一個範例是「判斷無窮迴圈程式會不會停止」， 而這個範例的對應解答是「不會」。 對於一個給定的問題， 一個針對此問題的演算法就是指根據問題的範例來產生解答的逐步程序， 而演算法的實作則是指將演算法轉化爲可以實際讓電腦運作的程式。 不同的演算法和不同的實作都可能導致效率的差異。

圖論相關的演算法問題可以粗分成兩類： 一種稱爲**決定性問題**（decision problem）， 這種問題只要針對範例回答「是」或「否」即可； 另一種稱爲**最佳化問題**（optimization problem）， 這種問題必須針對範例找出一個滿足條件的最好解答， 至於什麼是最好則是視問題的要求而定。 乍看之下， 最佳化問題似乎比決定性問題難，但是其實只要後者會做、 前者就一定會做。 第 1 章曾經談過分解的概念， 底下就以這個來舉這兩種問題的例子：

完全圖分解問題（最佳化）

參數： 一個給定的圖 G。

問題： 最少要用幾個完全圖， 才可以把 G 分解成這些完全圖？

完全圖分解問題（決定性）

參數： 一個給定的圖 G 和一個給定的正整數 k。

問題： 是否可以把 G 分解爲最多 k 個完全圖？

一個最佳化版的完全圖分解問題的演算法， 顯然可以導出 一個決定性版的完全圖分解問題的演算法。 反過來， 假設我們有一個決定性版的完全圖分解問題的演算法， 那麼對於最佳化問題， 我們只要逐一代入 $k = 1, 2, 3, \ldots$ 去解決定性問題， 直到第一次出現「是」的時候， 當時的 k 值就是最佳化問題的答案了。 如果 G 有 m 條邊， 那麼顯然 G 可以分解成 m 個 K_2， 因此我們最多只要執行 m 次決定性版的演算法就能解決最佳化問題。 利用二分法， 可以最多只要執行 $\log_2 m$ 次決定性版的演算法就能解決最佳

化問題。 而如果我們有辦法平行地進行這 m 次的執行， 那麼解決最佳化問題的時間就變得僅跟決定性問題一樣而已。

習慣上， 會用範例當中設定的參數之大小的函數來表示演算法的複雜度。 假設某一個問題 Π 的範例參數之大小 （也稱爲是此問題的**輸入大小**（input size））爲 n， 而且存在一個和 n 無關的常數 $c > 0$， 使得對應的演算法總是能夠在不超過 $cf(n)$ 的計算步驟之內得到解答（其中 f 是某一個函數）， 那麼就會說這個演算法能夠在 $O(f(n))$ 的時間內執行完畢； 而這個演算法的時間複雜度， 則是取滿足前敍述的最小函數 f （這邊所謂最小， 粗略而言， 是指成長的速度最慢）； 空間複雜度的表示法也是類似。 而問題本身的複雜度， 則是指所有能夠解決該問題的演算法當中複雜度最小的演算法的複雜度。

以第 1 章求一個圖的 Euler 迴路的問題爲例， 如果圖 G 有 n 個頂點及 m 條邊， 則此問題的輸入大小就是 $n + m$； 第 1 章給出的是一個需時爲 $O(n + m)$ 的演算法。 再以求 de Bruijn 序列的演算法來說， 輸入大小是 $1 + \lfloor \log_2 \sigma \rfloor + 1 + \lfloor \log_2 n \rfloor$， 因爲一般而言， 需要用 $1 + \lfloor \log_2 m \rfloor$ 的記憶體空間來儲存一個正整數 m； 第 1 章所提供的方法中， 光是圖 $G_{\sigma,n}$ 就有 σ^{n-1} 個點和 σ^n 條邊， 所以這個方法基本上是一個 $O(\sigma^n)$ 的演算法， 是一個「超指數型」時間的方法。

爲了讓讀者有大略的感覺， 這裡列出一個參考表如下所示； 從這個表中可以看出來， 只要 $n = 100$， 2^n 就是一個大得可怕的數字。 舉例來說， 如果電腦每秒鐘可以執行 10^{20} 個運算， 以一年約有 3×10^7 秒估計的話， 一個花費 2^{100} 步運算的演算法就得用上 300 年的時間， 這並不是一個人在有生之年可以看到答案的計算。 相對的， 如果演算法的時間是 $O(n^3)$， 對於輸入大小 $n = 100$ 的問題， 不到一秒鐘就能算出解答。

n	$n \log n$	n^2	n^3	2^n
10	2.3×10	10^2	10^3	10^3
10^2	4.6×10^2	10^4	10^6	10^{30}
10^3	6.9×10^3	10^6	10^9	10^{300}
10^4	9.2×10^4	10^8	10^{12}	10^{3000}
10^5	1.2×10^6	10^{10}	10^{15}	10^{30000}

圖 2.1: 數字成長比較表。

近年來， 研究學者致力的， 是針對各式各樣問題， 尋求更快、更有效率的演算法。 許多問題本來只有**指數時間演算法**（exponential-time algorithm）、 即複雜度爲 $O(a^n)$ 的演算法、 其中 $a > 1$ 爲常數， 後來改進成爲**多項式時間演算法**（polynomial-time algorithm）、 即複雜度爲 $O(n^k)$ 的演算法、 其中 $k \geq 1$ 爲常數， 最後可能還可以改進成爲**線性時間演算法**（linear-time algorithm）、 即複雜度僅爲 $O(n)$ 的演算法。 這三種程度的演算法的效率， 如圖 2.1 所顯示， 隨著輸入參數 n 的大小的成長而有極爲顯著的差異。

首先， 如果是指數時間演算法， 只要參數 n 的大小稍微增加， 需要的運算時間會立刻變得過份龐大， 一般認爲這樣的演算法是不切實際的。 相對地， 多項式時間演算法比較能夠把運算時間控制在一個合理的成長幅度之內， 一般我們會認爲這樣的演算法才具有實作的價值。 當然啦， 如果一個演算法需要的時間是 $O(n^{100})$， 就算這確實是多項式演算法， 我們也不會覺得這是一個堪用的演算法； 一個演算法究竟有沒有實用價值不能只看複雜度的層級， 也必須考慮在目前的電腦能力所及的範圍內是誰比較快的問題。 有些時候， 指數演算法反而會比多項式演算法管用， 因爲雖然當 n 比較大的時候它們總是遠遠不如多項式演算法， 但是在 n 比較小的時候卻可能快很多。 最後， 如果問題存在線性時間演算法那當然最好， 但是這種情況並不是常常有的。

一個貼近日常生活的例子是數字的**排序**（sorting）， 也就是給定一實數列 $x_1, x_2, \ldots, x_n$， 要將它從大到小重排爲 $x_{[1]} \geq x_{[2]} \geq \ldots \geq x_{[n]}$。 一個簡單的方法是， 先從這 n 個數中取出最大的當做 $x_{[1]}$， 接著由剩下的 $n-1$ 個數中取出最大的當做 $x_{[2]}$， 依此類推， 就能順利排序， 其所用的時間是 $n+(n-1)+\ldots+1 = n(n+1)/2$， 即爲 $O(n^2)$。 其實有比較複雜、 但是更有效率的方法叫做**堆排序**（heap sort）， 所花的時間爲 $O(n \log n)$。 當 $n = 10^5$ 時， 這兩種方法相差上千倍， 或著至少幾百倍。 想想看， 臺灣每年十萬高中學生入大學的考試， 考完之後， 就要先將所有學生的成績排序， 才能分發志願。 如果用到好的方法， 排序的工作可能是一天， 如果用 $O(n^2)$ 的方法， 光是排序可能就要一年， 那就是說， 七月大考完了之後， 要等上一年才能分發， 如果是這樣的話， 最好的方法就是考完之後大家先去當一年兵， 退伍之後就知道考上那間大學了。 當然， 臺灣的七月大考從來沒發生過這個現象， 依靠的就是演算法的技術不差。

再舉一個例子來說明， 對一個問題精益求精、 尋找更有效率演算法的過程。 給定

實數列 $x_1, x_2, \ldots, x_n$， 考慮從這個數列中找出連續的一段 $x_i, x_{i+1}, \ldots, x_j$、 使其和爲最大的問題。 一個從題目敍述直接來的簡單演算法是， 對所有滿足 $1 \le i \le j \le n$ 的 i 和 j， 求 $s_{i,j} = x_i + x_{i+1} + \ldots + x_j$、 然後再從所有的 $s_{i,j}$ 裡面去找出最大的一個。 由於所有的這樣的 i, j 共有 $O(n^2)$ 種選擇、 而每次計算 $s_{i,j}$ 要做 $O(n)$ 次加法， 因此用這種演算法需用 $O(n^3)$ 個步驟， 這是最直觀的想法所導致的結果。 但稍微改進的話， 可以用如下的方法： 對於一個固定的 i， 先將 $s_{i,i} = x_i$ 存起來， 然後當要求 $s_{i,j}$ 的時候， 改用 $s_{i,j} = s_{i,j-1} + x_j$ 這個式子來做遞迴計算， 如此一來總共只需要 $O(n)$ 次加法就可以把 $s_{i,i}, s_{i,i+1}, \ldots, s_{i,n}$ 都算出來， 於是就把總共的需時改進成爲 $O(n^2)$。 這個方法用到的空間也是 $O(n^2)$。 是否能有更省空間的方法？ 更進一步， 是否還有更快的方法呢？ 請參見習題 2.5。

當一個演算法被構造出來時， 就得到了對應問題的複雜度的一個上界。 而透過一些數學方法， 能給出問題複雜度的下界， 也就是去證明不管用什麼方法、 都至少需要一定的步驟才有可能解決該問題。 假如得到的上界和下界一致， 那就表示找到的已經是最佳的演算法， 而且問題的複雜度也跟著確定。 但是這樣的例子並不多， 對於大部分的問題而言， 能求得的上界和下界都是相差很遠的， 所以一般僅能知道問題複雜度落在那一個範圍而已。

在圖論領域中， 有許多問題至今都還沒有一個充分有效率的演算法， 因此會想知道究竟只是那樣的演算法還沒被找到、 或者說不定其實根本不存在。 這會牽涉到所謂 P 和 NP 的概念。 關於這個部分， 之後在第 **??** 章會有更完整的探討， 不過這裡先給一個簡單的介紹。 基本上， 給定一個決定性問題， 如果這個問題的答案可以在多項式時間內被確定出來， 那就稱爲 P 問題。 NP 的完整定義比較複雜， 概括而言， 針對一個給定的決定性問題， 假如當問題的答案爲「是」的時候我們有辦法在多項式時間內證明答案的確爲「是」， 那就稱爲 NP 問題[1]。 容易看出 P 問題都是 NP 問題（即 $\mathrm{P} \subseteq \mathrm{NP}$ ）， 此外， 因爲答案一般來說都只有有限種可能， 因此只要有辦法弄到充分多台的

[1]但是， 如果問題的答案爲「否」， 那就不見得有辦法一樣輕易地證明答案的正確性。 如果一個決定性問題當答案爲「否」的時候可以在多項式時間內被證明， 就稱爲是一個**餘-NP**（co-NP）問題。 有些問題同時屬於 NP 和餘-NP， 例如整數分解問題： 判斷一個給定的整數 m 是否含有不超過 n 的眞因數（這個問題屬於 NP 是容易的， 屬於餘-NP 則困難許多）。

P 和 NP 分別是指「Polynomial time」和「Non-deterministic Polynomial time」， 即「多項式時間」和「不定型多項式時間」之意， 後者的由來是因爲 NP 最初的定義是指可以在不定型 Turing 機上以多項式時間解決的問題。 詳情請見第 **??** 章。

電腦同時檢查那所有可能的答案， 就有辦法在多項式時間內解出一個 NP 問題 （這是另外一種 NP 的簡單說法）。 但是好奇的是， 有沒有可能只用一台電腦就在多項式時間內解出任何的 NP 問題？ 這就是演算法中至今尚未能解答的「P = NP 是否成立？」這個有名的問題[2]。

要如何證明「任何的 NP 問題」都可以在多項式時間內解決呢？ 這需要用到 **NP-困難**（NP-hard）以及 **NP-完全**（NP-complete）的概念。 假設有一個問題具有這樣的特性：「只要它存在多項式演算法、 那麼任何 NP 問題都可以藉助該演算法在多項式時間內解決」， 那麼就說這是一個 NP-困難問題。 如果一個問題同時是 NP-困難問題、也是 NP 問題， 那就說它是一個 NP-完全問題。

NP-完全的理論是由 Cook 在 1971 年建立 [2]， 他透過舉出實例證明了 NP-完全問題存在。 在那之後， Karp [7] 也陸續提出了許多其他 NP-完全問題的例子。 而以圖論的例子來說， 可以證明， 如果隨便給予兩個圖、 要判斷是否其中一個是另一個的子圖， 那麼這會是一個 NP-完全問題 （參見習題 **??**）。 在演算法和圖論領域中都存在有許許多多的 NP-完全問題， 只要有人能夠證明那當中任何一個可以在多項式時間內解決， 那麼就證明了 P = NP， 且必定會帶來重大的震撼衝擊。

只是， 到目前爲止沒有人能做到這一點， 因此現在大部分的人都相信應該是 P $\neq$ NP 才對、 雖然這也沒人能證明。 換句話說， 假設已經知道某個問題是 NP-完全問題， 那麼大致是不用指望它能存在一個充分有效率的演算法了。

等到在這本書沿途當中累積了足夠多的圖論基礎之後， 第 **??** 章會將介紹圖論中的 NP-完全問題的例子， 屆時，也會利用類似 Karp 的化簡手法去證明它們的 NP-完全性。

---

[2]千禧年大獎難題（Millennium Prize Problems）， 是七個由美國克雷數學研究所（Clay Mathematics Institute, CMI）於 2000 年 5 月 24 日公布的數學難題，「P = NP 是否成立？」是其中之一。根據克雷數學研究所訂定的規則， 所有難題的解答必須發表在數學期刊上， 並經過各方驗證， 只要通過兩年驗證期， 每解破一題的解答者， 會頒發獎金 100 萬美元。

這些難題是呼應 1900 年德國數學家 David Hilbert 在巴黎提出的 23 個歷史性數學難題， 經過一百年， 許多難題已獲得解答。 而千禧年大獎難題的破解， 極有可能爲密碼學以及航天、 通訊等領域帶來突破性進展。

2.3. 資料結構

演算法的目標是要讓電腦處理圖論的問題， 而電腦是被設計來處理數字的， 可是現在要處理的物件是頂點和邊， 爲了能夠讓設計出來的演算法被電腦執行， 需要用一個好的方式來組織那些會用到的資料。 在演算法學中， 探討如何將程式中的資料整理得有系統、 使得程式運作起來更有效率的學問就叫做**資料結構**（data structure）。

人們最熟知的資料結構是**陣列**（array）， 在數學中就相當於有下標的變數。 0-維陣列就是單一的變數， 也就是電腦中的一個記憶位置； 遞迴來說， d-維陣列就是一個有限序列的、 大小相同的 $(d-1)$-維陣列。 於是， 一般我們以 1-維陣列表示數學中的向量、 而以 2-維陣列表示矩陣， 依此類推。

陣列最大的意義在於可以利用下標來存取許多個不同的記憶體位置。 一般來說， 陣列中的每個元素都佔有同樣大小的空間， 例如都是 s 位元。 因此， 對於 1-維陣列 $A_1, A_2, \ldots, A_{m_1}$， 如果 A_1 的記憶體位置爲 B， 那麼 A_i 的位置就是 $B+(i-1)s$。 而高維度的陣列可以視爲是許多一維陣列的串連， 例如一個 $m_1 \times m_2$ 矩陣可以說成是一個 1-維陣列 （依照列的次序排列）

$$A_{1,1}, A_{1,2}, \ldots, A_{1,m_2}, A_{2,1}, A_{2,2}, \ldots, A_{2,m_2}, \ldots, A_{m_1,1}, A_{m_1,2}, \ldots, A_{m_1,m_2}，$$

因此如果 $A_{1,1}$ 的位置是在 B， 那 $A_{i,j}$ 的位置就會是在 $B+(i-1)m_2s+(j-1)s$。 類似地也可以考慮以行的次序優先的排法、 以及更高維度的陣列。

而以陣列爲基礎， 可以演化出其他各種不同的資料結構。 仔細說起來的話， 這些衍生的資料結構在本質上也是以陣列方式存放， 只是它們額外附加了一些演算法來進行特定操作， 使得它們在概念上具備了不同於陣列的型態。

堆疊（stack）和**佇列**（queue）是最簡單的兩種衍生資料結構， 它們最常見的用途是用來做工作的排程。 假定現在櫃臺陸陸續續來了一些人要辦理業務， 我們應該要以什麼順序來接待這些人呢？ 現實中約定俗成的辦法就是叫他們排隊、 然後先到的先辦， 這樣的安排方式基本上就是佇列； 每次有新的工作進來的時候就排在最後面， 而每次都是從最前面的資料開始處理， 處理完了就移除。 現實中似乎就只有這種作法是合理的， 但是在演算法中， 另外一種逆其道而行的作法也一樣重要； 堆疊採取的是後到先贏的作法， 每次有新的工作進來時也是排在最後面， 但是每次卻都是從最後面的資料先處理。

要實現堆疊和佇列的基本辦法就是用 1-維陣列。 堆疊就照字面上的意思， 每次新的資料進來就附加在陣列的末端、 然後每次都從末端先取出資料（並從陣列中移除）即可。 佇列同樣是以陣列的形式存在於記憶體當中， 每次新的資料也是附加在陣列的末端， 取出的時候則是從陣列最前面開始取走。

在堆疊當中， 通常會把資料的末端稱爲**頂端**（top）， 並且用一個變數「TOP」來記錄頂端的位置， 而將資料存在一個（例如叫 DATA 的）陣列當中。 於是， 每次新的資料就是放在 DATA[TOP+1] 的位置， 並且將 TOP 更新爲 TOP+1； 而移出資料時則是取走 DATA[TOP] 的資料， 並將 TOP 更新爲 TOP−1。 在這個操作時，不用眞的把移出的資料從記憶體中清除沒關係， 畢竟下次有新的資料進來時它就會自動被覆蓋掉。 圖 2.2 展示了堆疊資料型態的結構。

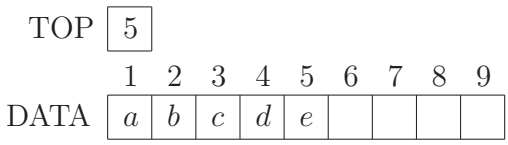

圖 2.2: 一個堆疊的例子。

而在佇列當中， 除了要知道**尾端**（rear）在那裡之外， 也需要知道**開頭**（front）在那裡， 因此會用兩個變數「REAR」和「FRONT」來儲存。 此時資料的分佈就是從開頭位置到尾端位置所標示的範圍內。 每次有新的資料進來就是放在 DATA[REAR+1] 的位置， 並將 REAR 更新爲 REAR+1， 而移出資料時則是取走 DATA[FRONT+1]的資料， 並將 FRONT 更新爲 FRONT+1。 圖 2.3 展示佇列資料型態的結構。

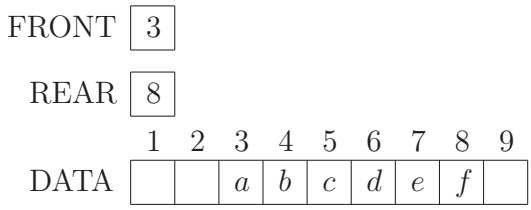

圖 2.3: 一個佇列的例子。

不過， 這樣的作法有一個小問題是， 整個佇列在記憶體空間中會不斷地往後移動，

如果 DATA 預留的空間有限， 當 FRONT 和 REAR 一直增加時， 空間很快就會被用完， 而 FRONT 前面的空間則閒置沒用。 因此， 如果可以事先知道這個佇列最多只會有幾項， 例如至多 n 項， 那麼就可以宣告一個大小爲 n 的 1-維陣列、 然後循環地使用這個記憶體空間； 也就是說， 當 FRONT（或是 REAR）是 n 時 FRONT+1（或是 REAR+1）就設爲 1。 因爲已經事先預估好資料最多只會有 n 項， 因此這樣的作法不會導致資料相撞。

基本上， 堆疊和佇列都只是陣列的稍微變形， 接下來要看的概念就有很大的不同。

表列（list）是由一些大小相同的**記錄**（record）線性連接而成， 其中每個記錄含有一個或多個**欄位**（field）， 有些存的是**資料**（data）、 有些則是存**指標**（pointer）。 圖 2.4 所示的是兩個**單連表列**（singly linked list）。

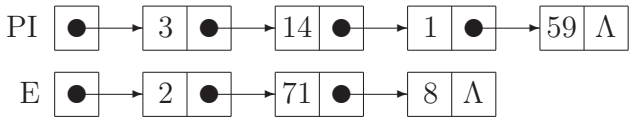

圖 2.4: 單連表列 PI 和 E。

表列和陣列最大的不同在於， 陣列的資料是依序存在電腦的記憶體中， 但表列的資料可以到處放置、 並以指標將它們串連。 舉例來說， 上面的兩個單連表列中， 我們可以利用圖 2.5 的陣列來存放它們， 其中 Λ 表示未定義的符號。

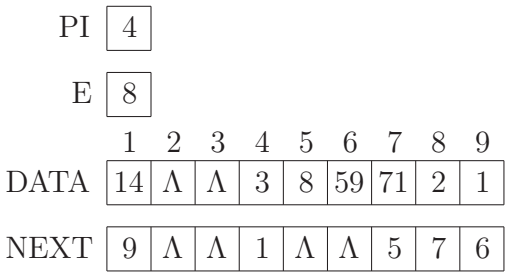

圖 2.5: 用陣列表達表列 PI 和 E。

這個例子的解讀這樣的。 首先有兩個變數 PI 和 E， 裡面的值分別代表對應表列在

記憶體中的第一個位置在那裡。 例如 PI 的值是 4， 就表示 PI 這個表列的開頭放在記憶體的第 4 個位置， 在上面的例子中是 DATA 這個陣列的第 4 個元素， 也就是 3。 如果想要存取下一個元素， 就去看對應的位置在 NEXT 陣列中是多少 （這個陣列就是代表指標）， 此例中爲 1， 就表示下一個資料被存放在 DATA 的第 1 個位置， 其值爲 14。 如此就可以依序讀出 PI 的內容爲 3、14、1、59， 直到讀到 NEXT 爲 Λ 的時候， 就知道 PI 表列到此結束了。 單連表列 E 也可以類似地讀出來。

任何資料結構都有好有壞， 要看需求來決定要用那一種資料結構。 比方說， 表列有一個明顯的缺點在於讀取資料的過程比較複雜而沒有效率， 例如若要讀取 PI 的第三個元素， 那就無法直接知道它被放在那裡， 必須從第一個元素開始逐一地找到第三個才知道； 相對地， 因爲陣列是依序排放， 那一個元素放那裡只要計算一下就知道， 所以要找資料就比較容易。 但是， 表列相較於陣列的好處在於它可以很容易地插入資料。 以 PI 來說， 假設想要在 14 跟 1 中間插入一個新的資料爲「5」， 那首先我隨便找一個地方存放「5」這個資料 （以上面的例子來說， DATA 的第二個位置是空的， 所以可以選來用）， 然後將「14」指向它、 並由「5」繼承「14」原本的指標， 就完成插入的動作了。 處理之後的記憶體結構如下。

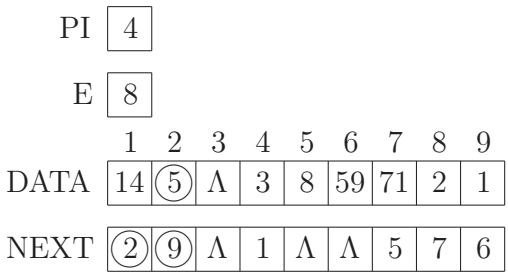

圖 2.6: 一個插入的動作，只需更動三個（圈起來的）記憶體中的資料。

在這個操作過程中只更動了三個記憶體位置的資料， 如圖 2.6 中圈起來的位置所示。 然而， 假如在陣列中要做同樣的事情， 想像一個有上千個元素的陣列， 若要在第 1 個位置插入資料， 就必須將陣列所有的元素全部往後搬 s 位元， 才有辦法進行插入。

如果希望表列這樣的好處也可以發揮在刪除資料的功能上， 那比較好的作法是再多設一個指標 PREV， 用來表示每一個記錄的「前一個」是在何處。 這樣一來， 如果要

刪除一個記錄， 只要將它對應的 PREV 和 NEXT 連接、 並將自己清除即可。 這樣的表列稱爲**雙連表列**（doubly linked list）； 圖 2.7 和圖 2.8 是分別將圖 2.4 和圖 2.5 改爲雙連表列後的樣子。 有時候， 也可以視需要，使用更複雜的結構。

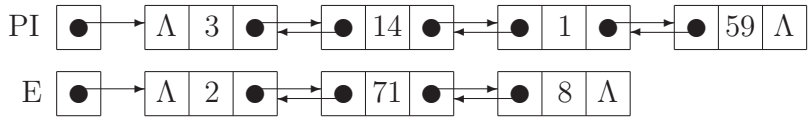

圖 2.7: 雙連表列 PI 和 E。

PI | 4

E | 8

	1	2	3	4	5	6	7	8	9
DATA	14	Λ	Λ	3	8	59	71	2	1
NEXT	9	Λ	Λ	1	Λ	Λ	5	7	6
PREV	4	Λ	Λ	Λ	7	9	8	Λ	1

圖 2.8: 雙連表列 PI 和 E 的陣列表達。

2.4. 圖的表示法

有了上一節關於資料結構的基本概念之後， 接著來要來看怎麼用資料結構表示一個圖。

假設 $G = (V, E)$ 是一個圖， 有 n 個頂點 $1, 2, \ldots, n$。 我們定義 G 的**相鄰矩陣**（adjacent matrix）$A = [a_{ij}]$ 是一個 $n \times n$ 矩陣， 其中

$$a_{ij} = \begin{cases} 1, & \text{若 } ij \in E\text{；} \\ 0, & \text{若 } ij \notin E\text{。} \end{cases}$$

由定義可知， A 是一個主對角線均爲 0 的對稱 (0,1)-矩陣。 利用這個矩陣對應的陣列結構， 可以將一個圖的所有資訊儲存起來。 如果要表示一個重圖， 可以用 a_{ij} 表示點 i 和

點 j 之間的重邊個數。 在近圖的情況， 如果點 i 上面有 s 條迴邊， 則將 a_{ii} 設爲 $2s$。 最後， 在有向圖的情況， a_{ij} 只代表從點 i 連向點 j 的邊， 所以 a_{ij} 可能不等於 a_{ji}。

這樣的表達方法有好有壞。 好處是， 可以很簡單地判斷兩點是否相鄰、 且新增邊或刪除邊的操作也都很容易做。 不過， 如果要問點 i 的全體鄰居有那些點， 那就要將其他各點都看過一遍， 得花 $O(n)$ 的時間， 而如果要看這個圖總共有多少邊， 那就得用 $O(n^2)$ 的時間。 當 $\Delta(G)$ 遠比 n 小、 或者邊數遠比 n^2 小的時候， 這種表示法很沒效率。 而相鄰矩陣需要的儲存空間是 $O(n^2)$， 這一般來說當 n 比較大的時候也是大了點。

當演算法需要常常檢驗某一頂點的所有鄰居時， 最好選擇以**相鄰表列**（adjacent list）來表示圖。 相鄰表列是針對每一個頂點建立一個表列， 裡面存放的是該點各個鄰居的編號。 圖 2.9 展示了這兩種不同的表示法。

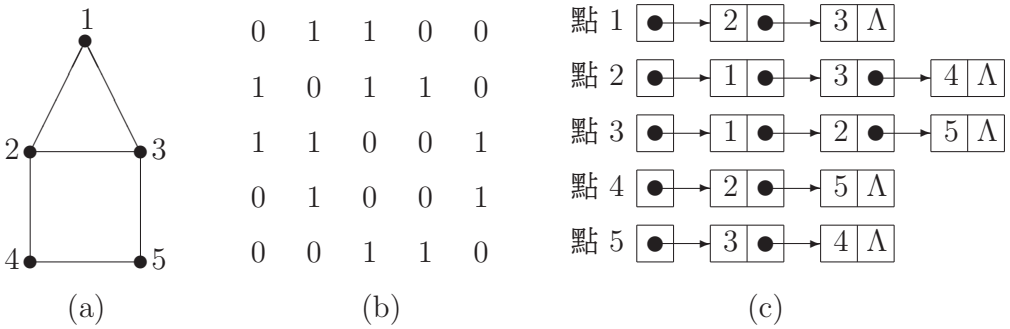

圖 2.9: (a) 有 5 點 6 邊的圖 G。(b) 圖 G 的相鄰矩陣。(c) 圖 G 的相鄰表列。

利用相鄰表列的話， 設 G 的邊數爲 m， 則需要的儲存空間會是 $O(n+m)$。 因此就空間的考量上相鄰表列的表現較好。 而如果要將一個點 v 的鄰居都選出、 或者將所有的邊都選出， 分別也只需要 $O(\deg(v))$ 和 $O(m)$ 的時間。 不過， 相對地它要判斷兩點是否相鄰就需要花上 $O(\Delta(G))$ 的時間， 刪除邊也不像用相鄰矩陣那麼快速。 這些效能上的差異， 都是在實作的時候必須審愼考慮的。

2.5. Euler 迴路的案例

這一節將用第 1 章找 Euler 迴路的方法爲例子， 詳細寫出圖的表示法、 以及電腦程式。 在往後的章節中， 談論演算法的時候則以摘要式的程式爲主。 這裡是用相鄰表列表示一

個圖。 但是爲了表示圖的某一邊是否已經被走過， 額外需要用一個陣列「MARK」來註記： MARK = 0 表示該邊尚未被走過， 而爲 1 則表示該邊已經被走過。 但是， 一條邊 ij 出現兩次， 一次在點 i 的相鄰表列中， 另一次在點 j 的相鄰表列中； 當 Euler 迴路由點 i 經由此邊走到點 j 的時候， 自然需要把點 i 的相鄰表列中的 ij 邊註記、 但同時也需要把點 j 的相鄰表列中的 ji 邊註記。 爲達到這個目的， 需要在 ij 和 ji 之間設置雙連指標， 以便很容易就能找到對方。 爲此， 再多設一個指標「SAME」。

圖 2.10 是一個有 4 個點與 9 條邊的重圖， 演算法輸入的資料爲「$V = 4$」以及其 9 條邊： 「12, 12, 12, 14, 23, 23, 24, 34, 34」， 在這些邊後面，會以 00 表示輸入完畢， 並利用如下程式 input-graph 得到相鄰陣列的內容：

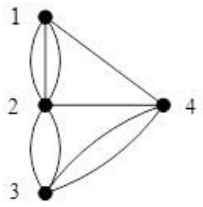

圖 2.10: 有 4 個點與 9 條邊的重圖 G。

```
input-graph :
    input V ;
    for i = 1 to V do ADJ[i] ← 0 ;
    NEW ← 1 ;
    input i, j ;
    while ( i ≠ 0 )
    {   DATA[NEW] ← j ;          DATA[NEW+1] ← i ;
        NEXT[NEW] ← ADJ[i] ;     NEXT[NEW+1] ← ADJ[j] ;
        ADJ[i] ← NEW ;           ADJ[j] ← NEW+1 ;
        SAME[NEW] ← NEW+1 ;      SAME[NEW+1] ← NEW ;
        MARK[NEW] ← 0 ;          MARK[NEW+1] ← 0 ;
        input i, j ;             NEW ← NEW+2 ;
    }
```

透過程式 input-graph 輸入圖 G 的資料， 會得到其相鄰表列的結構如圖 2.11 所示， 爲了畫圖的方便起見，指標 SAME 只畫出 12, 12, 12, 34, 34 這五條邊。 圖 2.12 是其陣列表示。

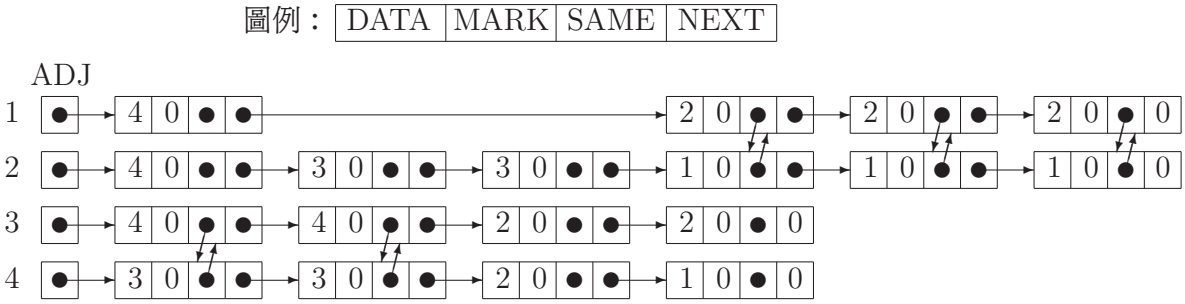

圖 2.11: 圖 G 的相鄰表列，方便起見指標 SAME 只畫出 12, 12, 12, 34, 34 這五條邊。

ADJ	7	13	17	18															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
DATA	2	1	2	1	2	1	4	1	3	2	3	2	4	2	4	3	4	3		
NEXT	0	0	1	2	3	4	5	0	6	0	9	10	11	8	12	14	15	16		
SAME	2	1	4	3	6	5	8	7	10	9	12	11	14	13	16	15	18	17		
MARK	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		

圖 2.12: 圖 G 的相鄰表列之陣列表示。

在上述的表示法中， 對於重邊我們並未分別命名， 只用 12, 12, 12 表示三條重邊， 事實上， 因爲它們的資料分別存在不同的位置， 就已經達到分別命名的功能。 另外， 雖然這裡所舉的例子沒有迴邊， 但是若圖中有迴邊 ii 也是可以表示清楚的， 只是這時候， 點 i 的相鄰表列中會出現兩次 i。

以下的程式 Euler-tour 是將第 1 章找一個圖的一條 Euler 迴路的方法、 也就是「能走就走、不能走則回頭」的方法， 仔細寫出來的的結果， 它將 Euler 迴路存在一個雙連表列 TOUR 中。

```
Euler-tour (V, ADJ, DATA, NEXT, SAME, MARK, TOUR, TOUR_NEXT, TOUR_PREV) :
    for i = 1 to V do A[i] ← ADJ[i] ;
    NEW ← 1 ;
    TOUR[NEW] ← 1 ; TOUR_NEXT[NEW] ← 0 ; TOUR_PREV[NEW] ← 0 ;
    CUR ← NEW ;
    while (CUR ≠ 0)
    {     i ← TOUR[CUR] ;
          if (A[i] ≠ 0 then
                NEW ← NEW+1 ;
                TOUR[NEW] ← DATA[A[i]] ;
                TOUR_NEXT[NEW] ← TOUR_NEXT[CUR] ;
                TOUR_PREV[NEW] ← CUR ;
                CUR ← NEW ;
                MARK[A[i]] ← 1 ; MARK[SAME[A[i]]] ← 1 ;
                j ← NEXT[A[i]] ;
                while (j ≠ 0 and MARK[j] = 1) j ← MARK[j] ;
                A[i] ← j ;
          else CUR ← TOUR_PREV[CUR] ;
    }
```

2.6. 聯集尋找問題

利用指標除了可以達到前面所述的好處之外， 還有更多的應用。 底下來看其中一個經典的用法， 是演算法學當中的「**聯集尋找**（Union-Find）」問題， 這在演算法的設計中有許多用途， 本書主要是在第 **??** 節求最小生成樹時可以用。

假設有一個宇集合 $U = \{1, 2, \ldots, n\}$、 這個集合在我們處理演算法的過程中被分割成一些互斥的非空子集合； 一般來說， 初始情況的分割一般是 $\{1\}, \{2\}, \ldots, \{n\}$， 此時以 i 來當作集合 $\{i\}$ 的名稱。 在演算過程， 經常需要做兩件事： 將兩個既有的集合聯集而形成一個新的集合、 以及判斷一個特定的元素在那一個集合裡面。 這樣的一種處

理機制在很多演算法當中都看得到。

一種最單純的作法是， 建立一個長度爲 n 的陣列 NAME， 其中 NAME[i] 的值就表示 i 這個元素所屬的集合的名稱。 這麼一來， 要判斷一個元素 i 在那個集合， 只要看 NAME[i] 就立即可以完成， 但是若我們要將集合 a 和集合 b 做聯集並合併並以 a 命名， 我們就必須一一看過 NAME[1], NAME[2], ..., NAME[n]、 如果發現值爲 b 的就換成 a。 於是， 每做一次聯集的動作就需要 n 個步驟， 這麼一來， 假如在演算法的過程當中總共需要做 $O(n)$ 次聯集的動作， 就會需要 $O(n^2)$ 的時間， 這樣的時間略嫌長一點。

一種稍微改進的方法是， 除了建立上述的陣列 NAME 以外， 對每個集合我們再用一個單連表列來表示其所有的元素。 這個時候， 假如我們要將集合 a 與集合 b 聯集起來並以 a 命名， 我們只要把 b 對應的單連表列看一次、 將裡面的 i 都把對應的 NAME[i] 都改成 a， 然後把兩個表列串在一起即可。 這麼一來， 每次聯集所需的時間就從 n 改進成爲 b 的大小。 雖然這樣的改進， 大部分時候都很管用， 但是可能出現一種極端的情況， 那就是， 第一次 b 的大小是 1、 第二次是 2、 ... 一直到第 $n-1$ 次時是 $n-1$， 這樣的話， 整體用的時間還是 $O(n^2)$。

可以再做一點點小小的改進， 那就是， 再定義一個陣列 SIZE、 其中 SIZE[a] 用來記錄每個集合 a 的大小。 在上述聯集的過程中， 如果發現 SIZE[a] $<$ SIZE[b]， 那我們就改將集合 a「併入」集合 b 中， 也就是說每次我們都只去改比較小的那個集合的元素所對應的 NAME。 這個看起來很小的修改， 就可以使得整體的演算法複雜度降低到 $O(n \log n)$ 了 （見習題 2.17）。

由以上的討論可以看出來， 聯集尋找問題經典之處就在於， 聯集和尋找這兩種動作剛好有點衝突， 要是資料結構設計得使得其中一邊的執行速度快， 另一邊往往就會慢。 下面介紹一個更有效率的方法。

假設先著眼於聯集這個動作， 那麼一個好的想法是採用指標結構。 把每個集合的元素用指標關係串連成一個樹狀的結構， 其頂端就作爲這整個集合的代表。 也就是， 有一個長度是 n 的 1-維陣列 P， P[i] 表示 i 在樹中的父親， 而當 P[i] $= 0$ 時就表示 i 是某一個樹的頂端， 這個樹所代表的集合的名子就是 i。 這時候， 如果想判斷一個點屬於那一個集合， 只要沿著指標一直走到代表就知道了； 而做聯集動作就更簡單了， 只要把其中一個集合的代表指向另外一個集合的代表， 一個動作就完成了， 參見圖 2.13 的

例子。 這樣的指標結構比之前的單連表列要複雜一點， 但是利用同樣的想法， 也不難用陣列實際將它表現出來。

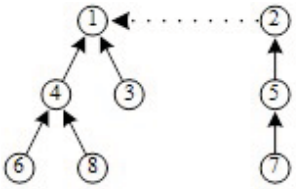

圖 2.13: 一個聯集的動作。

在圖 2.13 中， 根據這個結構， 6 跟 3 都是屬於「1」所代表的集合當中， 而 5 跟 7 則是在 2 所代表的集合裡。 如果我們想把這兩個集合做聯集， 只要把 2 再次指向 1， 一切就完成了， 也就是， 將 P[2] 由原來的 0 變成 1。

這個想法是很不錯， 但是有個小缺點， 那就是這些集合的樹狀結構可能會拉得很長； 也就是說， 每個資料離代表元素的距離有可能會非常遠， 這就導致尋找的速度變慢， 每次都要走很遠才能確定一個元素到底屬於那一個集合。

不過這個問題是有辦法解決的， 只要有辦法維持這些結構不要太長就行了。 這邊有兩個小技巧： 第一個和前面一樣， 我們用陣列記錄每一個集合的大小， 而當我們要做聯集時， 永遠把比較小的集合指向大的集合的代表， 這樣出來的樹狀結構會好一點； 另外一個技巧稱爲**路徑壓縮**（path compression）。 路徑壓縮的想法是， 既然尋找的動作會花上較多時間， 那就來想辦法減少尋找的時間。 辦法是這樣， 以圖 2.13 中合併後的樹爲例， 當我做「尋找 7」這個動作時， 它會循著 7、5、2 最後找到 1， 而在這個過程當中， 我們就順便把 7、5 也一起指向 1：

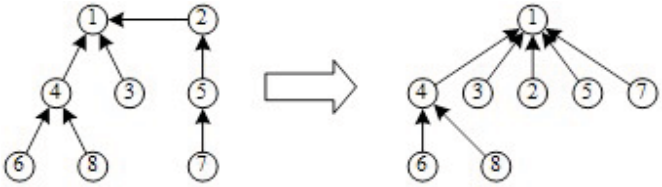

圖 2.14: 進行尋找時順便做路徑壓縮。

圖 2.14 顯示出經過尋找之後順便改變了指標的結構。 這麼一來， 要是下次再有機

會尋找路過 7 的時候， 它離代表元素就只有一步之遙， 大大地節省了時間。 下面是相關的程式：

Union-Find-Initialization :
    input n ;
    for i = 1 **to** n **do** { P[i] ← 0 ; SIZE[i] ← 1 ; }
Union(i, j) :
    if (SIZE[i] > SIZE[j])
    then { SIZE[i] ← SIZE[i] + SIZE[j] ; P[j] ← i ; }
    else { SIZE[j] ← SIZE[i] + SIZE[j] ; P[i] ← j ; }
Find(i) :
    r ← i ; **while** (P[r] ≠ 0) r ← P[r] ;
    s ← i ; **while** (s ≠ 0) { P[s] ← r ; s ← P[s] ; }
    return r ;

把這些概念全部整合的演算法稱爲**高速互斥集聯集法**（fast disjoint-set union algorithm）， 這個方法相當地快， 演算法上可以證明（有興趣的讀者可參見 [1]），如果我們要用這個方法處理 n 個元素的集合之聯集與尋找動作各 $O(n)$ 次， 則所需的時間是 $O(nG(n))$。 $G(n)$ 是什麼呢？ 它是這樣定義的： 考慮一個遞迴定義的函數 $F(n)$， 其中 $F(0) = 1$、 $F(i) = 2^{F(i-1)}$； 於是 $F(1) = 2$、 $F(2) = 4$、 $F(3) = 16$、 $F(4) = 65536$ 到 $F(5) = 2^{65536}$ 已經是個近兩萬位的天文數字了； 而 $G(n)$ 則定義爲使得 $F(k) \geq n$ 的最小 k 值。 這是一個成長得極慢的函數， 因此我們說， 應用這個演算法所需的時間幾乎是線性的。 實際應用上， 我們幾乎永遠沒有機會遇到 n 使得 $G(n) = 6$ 的那一天。

2.7. 習題

2.1. 用 $\gcd(m, n)$ 表示非負整數 m 和 n 的最大公因數。 試證：如果 $m = an + r$， 其中 a 和 r 是整數且 $0 \leq r < n$， 則 $\gcd(m, n) = \gcd(n, r)$。 並證明：這可以用來構造出一個用 $O(\log m + \log n)$ 個除法的演算法。

2.2. 要驗證一個正整數 n 是否爲質數， 依照定義，可以檢驗從 2 到 $n-1$ 的整數 i 是否整除 n， 這是一個用 $O(n)$ 個除法的演算法。 請設計一個用 $O(\sqrt{n})$ 個除法的演算法。

2.3. 針對同一個問題， 假設演算法甲對於參數大小爲 n 的範例需用 n^2 個步驟來執行、而演算法乙則需要 2^n 個步驟。

(1) 如果目前的電腦每秒可執行 10^{20} 步， 那麼一分鐘內上述演算法可以完成大小多少的範例？ 一小時呢？ 一年呢？

(2) 如果電腦大進步， 使得速度增進爲原來的一百萬倍， 那麼上述演算法的容量改進爲何？

2.4. 用 $n-1$ 次比較可以找出 n 個實數中最大（或最小）的數， 所以用 $2n-2$ 次比較可以找出 n 個實數中最大的數及最小的數。 請設計出用更少次比較找出 n 個實數中最大的數及最小的數。 這是可能的，例如，只要用 1 次比較就可以找出 $n=2$ 個實數中最大的數及最小的數， 用 3 次比較就可以找出 $n=3$ 個實數中最大的數及最小的數。

2.5. 給定實數列 $x_1, x_2, \ldots, x_n$， 要從中找出一段連續的若干項使其和爲最大。 第 2.2 節中已經舉出了一個需時 $O(n^2)$ 的方法， 試找一個比這個更有效率的演算法。

2.6. 設 A 是大小爲 $m_1 \times m_2 \times \ldots \times m_d$ 的 d-維陣列， 其中每個元素都佔 s 位元， 而 $A_{1,1,\ldots,1}$ 所在的記憶體位置爲 B， 試求 $A_{i_1,i_2,\ldots,i_d}$ 的位置。

2.7. 單連表列 A 和 B 的資料存在陣列 DATA 中、 而其指標維陣列是 NEXT。 下面這個程式將這兩個單連表列聯合起來放在 A 中。

t = A;

while NEXT[t] $\neq$ 0 **do** t = NEXT[t];

NEXT[t] = B。

請修改資料結構， 使得將兩個單連表列聯合起只需做 $O(1)$ 次動作。

2.8. 雙單連表列 A 的資料存在陣列 DATA 中、 而其指標維陣列是 NEXT 和 PREV。

(1) 請寫出程式，將新資料 x 加入 A 中、放在 i 後面。

(1) 請寫出程式，將 A 中的一項 i 刪除。

2.9. 假設圖 $G=(V,E)$ 有 n 個頂點 $v_1,v_2,\ldots,v_n$ 及 m 條邊 $e_1,e_2,\ldots,e_m$， 定義 G 的**相連矩陣**（incidence matrix） $M=[b_{ij}]$ 是一個 $n\times m$ 矩陣， 其中

$$b_{ij}=\begin{cases}1, & \text{若 } v_i \text{ 和 } e_j \text{ 相連；}\\ 0, & \text{若 } v_i \text{ 和 } e_j \text{ 不相連。}\end{cases}$$

試證： $MM^{\mathrm{T}}=A+I$， 其中 A 爲 G 的相鄰矩陣、 I 爲單位矩陣。

2.10. 假設有向圖 $G=(V,E)$ 有 n 個頂點 $v_1,v_2,\ldots,v_n$ 及 m 條邊 $e_1,e_2,\ldots,e_m$， 定義 G 的**相連矩陣**（incidence matrix） $M=[b_{ij}]$ 是一個 $n\times m$ 矩陣， 其中

$$b_{ij}=\begin{cases}1, & \text{若 } v_iv_{i'}=e_j\text{；}\\ -1, & \text{若 } v_{i'}v_i=e_j\text{；}\\ 0, & \text{若 } v_i \text{ 和 } e_j \text{ 不相連。}\end{cases}$$

試證明：M 是**完全單模**（totally unimodular）， 也就是、M 的任意子矩陣的行列式是 0、1 或 -1。

2.11. (1) 請找出一序列無窮多個圖、其相連矩陣都不是完全單模。

(2) 試證明：二分圖的相連矩陣都是完全單模。

2.12. 令 A 爲圖 G 的相鄰矩陣。 對於非負整數 k， 設 $A^k=[c_{ij}]$ 而 $(A+I)^k=[d_{ij}]$。 請問 c_{ij} 和 d_{ij} 分別代表什麼量？

2.13. 假設圖 $G=(V,E)$ 中 $V=\{1,2,\ldots,n\}$ 而 E 則以 m 個有序對表示。 我們可以用**連續空間**（sequential space）來儲存 G 中各點的鄰居， 也就是在陣列 DATA 中先存 $N(1)$、 然後接續著存 $N(2)$、 ... 依此類推直到放完 $N(n)$ 爲止； 我們以 b_i 表示資料 $N(i)$ 在 DATA 中的起始位置， 於是 $N(i)$ 所在的範圍就是從 DATA$[b_i]$ 到 DATA$[b_{i+1}-1]$ 這一段 （其中定義 b_{n+1} 爲資料結束後的下一個位置）。 b_i 的值存放在陣列 BEG 內。 例如， 設 $n=9$， 而

$$E=\{31,41,59,26,53,58,97,93,23,84\}\text{，}$$

則此時 DATA 和 BEG 的資料分別如下：

試寫一個演算法（或程式）， 能讓使用者輸入 n 和 E， 產生出對應的 DATA 和 BEG。

BEG	1	3	5	9	11	14	15	16	18	21												
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	
DATA	3	4	6	3	1	5	9	2	1	8	9	3	8	2	9	5	4	5	7	3		
	N_1		N_2		N_3				N_4		N_5			N_6	N_7	N_8		N_9				

圖 2.15: 連續空間儲存法，其中 N_i 表示 $N(i)$。

2.14. 在第 2.5 節 Euler 迴路的案例中， 改用連續空間表示圖 2.10 的圖、 但同時要含 SAME 和 MARK 兩個欄位。

2.15. 在第 2.5 節 Euler 迴路的案例中， 利用連續空間表示圖、據以改寫程式 input-graph。

2.16. 在第 2.5 節 Euler 迴路的案例中， 利用連續空間表示圖、據以改寫程式 Euler-tour。

2.17. 在聯集尋找問題中， 我們若採用 NAME、 集合表列與 SIZE 的方法， 試證明， 若演算法總共做了 $n-1$ 次聯集的運算， 那需時會是 $O(n \log n)$。

2.8. 參考文獻

[1] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Massachusetts, 1974.

[2] S. A. Cook, The complexity of theorem-proving procedures, *Proc. 3rd Ann. ACM Symp. on Theory of Computing Machinery*, New York, 1971, pp. 151-158.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction to Algorithms*, Second Edittion, The MIT Press, Cambridge, Massachusetts, 2002.

[4] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, San Francisco, California, 1978.

[5] M. C. Golumbic, *Algorithmic Graph Theory and Perfect Graph*, Academic Press, New York, 1980.

[6] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*, Coumpter Science Press, 1978.

[7] R. M. Karp, Reducibility among combinatorical problems, in R. E. Miller and J. W. Thatcher (eds.), *Complexity of Computer Computations*, Plenum Press, New York, 1972, pp. 85-104.

[8] U. Manber, *Introduction to Algorithms, A Creative Approach*, Addison-Wesley, Reading, Massachusetts, 1989.

[9] D. Hillis 著，林遠志、陳振男譯，《電腦如何思考》，天下文化，1999 年。

[10] N. Macrate 著，楊昌裔譯，《馮紐曼—現代電腦發展的先驅》，牛頓出版公司，民國 85 年。