演算法基礎與實作
– 使用 Python 語言

江振瑞

March 20, 2019

Contents

Chapter 1

1

Chapter 2

2

Chapter 3

3

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

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 (recirsively) 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 的冪 (a power a 2)。圖4.1顯示 2×2, 4×4 與 8×8 棋盤。
- 缺陷棋盤是有一單格 (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。

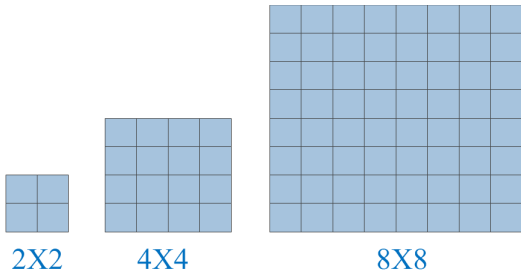

圖 4.1: 2×2, 4×4 與 8×8 棋盤

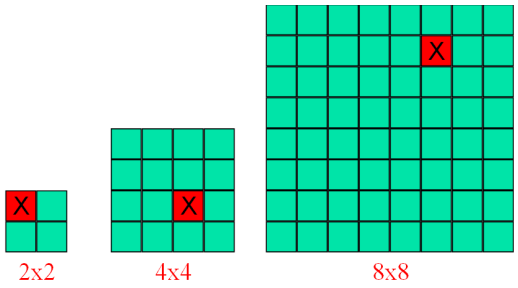

圖 4.2: 2×2, 4×4 與 8×8 缺陷棋盤。

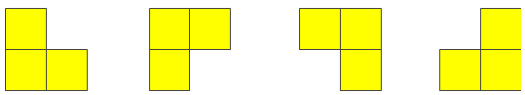

圖 4.3: 三格骨牌的四種方向

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。

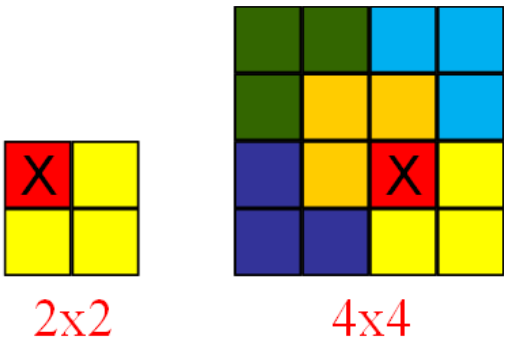

圖 4.4: 2×2 與 4×4 缺陷棋盤填滿問題解答

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 的冪

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)。

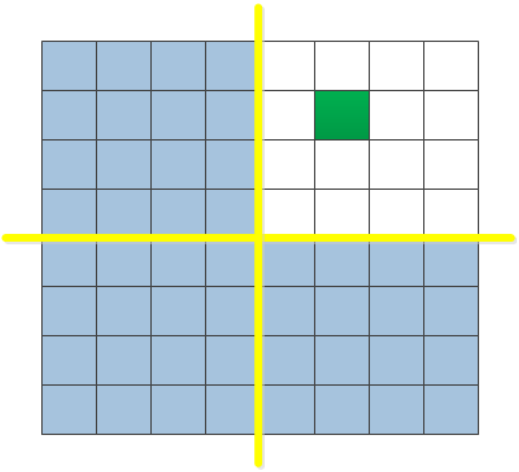

圖 4.5: 將棋盤分割成 4 個更小的棋盤。

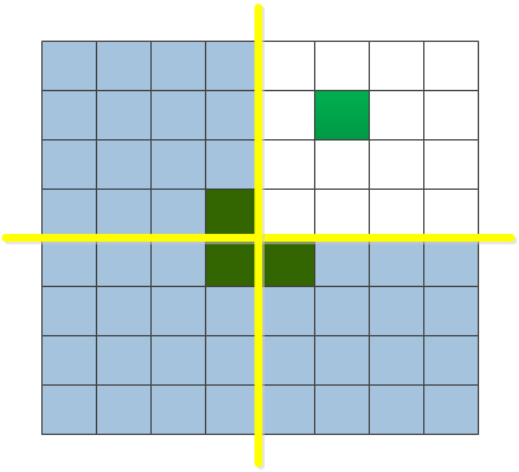

圖 4.6: 在 3 個相鄰正常棋盤角落放置 1 個三格骨牌

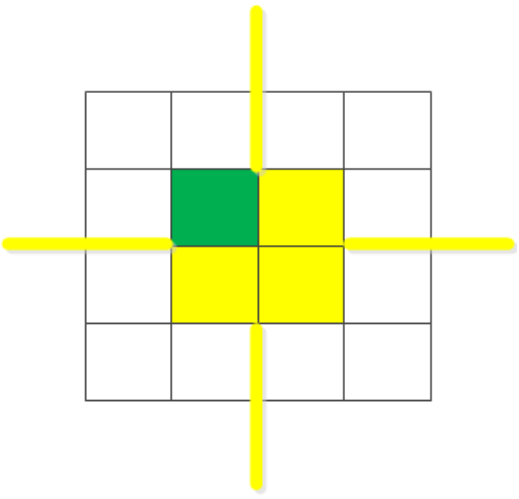

圖 4.7: 以遞迴方式填滿較小的缺陷棋盤

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 $0, ..., n-1$)，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: n1 ← q − p + 1
2: n2 ← r − q
3: for i ← 1 to n1 do                          ▷ 複製 A[p..q] 到暫存陣列 L[1..n1]
4:     L[i] ← A[p + i − 1]
5: for j ← 1 to n2 do                          ▷ 複製 A[q + 1..r] 到暫存陣列 R[1..n2]
6:     R[j] ← A[q + j]
7: L[n1 + 1] ← ∞                               ▷ 以人為方式創造兩個終點標記使其不會被複製到 A
8: R[n2 + 1] ← ∞
9: i ← 1
10: j ← 1
11: for k ← 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] ≤ R[j] then
13:         A[k] ← L[i]
14:         i ← i + 1
15:     else
16:         A[k] ← R[j]
17:         j ← 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 $\mathrm{O}(n)$，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 $\mathrm{T}(n)$，則我們可以得到以下的式子:

$$\mathrm{T}(n) = 2\mathrm{T}(n/2) + n$$

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 $2\mathrm{T}(n/2)$ 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 $\mathrm{T}(n) = 2\mathrm{T}(n/2) + n$ 中的 n 取代為 $n/2, n/4, ...$，則我們可以得到:

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

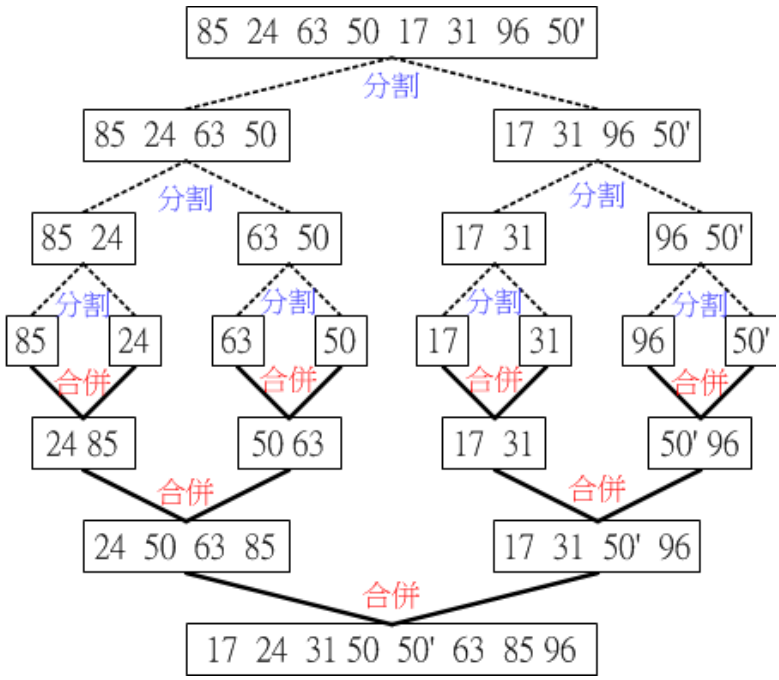

圖 4.8: 合併排序演算法排序過程

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 $\mathrm{T}(1) = 1$。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:

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

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 $\mathrm{O}(n \log n)$。

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 [1]。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 $0, ..., n-1$)，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: $p \leftarrow A[rb]$ ▷ p 為中樞 (pivot) 元素
3: $l \leftarrow lb$ ▷ l 為左指標
4: $r \leftarrow rb - 1$ ▷ 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** ▷ $l < r$ 成立表示分割尚未完成
9: 對調 $A[l]$ 與 $A[r]$
10: **else break** while-loop ▷ $l < r$ 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'	
l						r		
85	24	63	50	17	31	96	50'	
l					r			將l及r所指之元素(85, 31)對調
31	24	63	50	17	85	96	50'	
		l		r				將l及r所指之元素(63, 17)對調
31	24	17	50	63	85	96	50'	
r			l					(l>=r)表示切割完成
31	24	17	50'	63	85	96	50	
r			l					將p調至l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 $\mathrm{T}(n)$，針對最佳狀況，我們可以得到以下的式子:

$$\mathrm{T}(n) = n + 2\mathrm{T}(n/2)$$

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 $\mathrm{n}/2$) 子陣列排序。

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

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

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 $\mathrm{T}(1) = 1$。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:

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

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 $\mathrm{T}(n)$，針對最差狀況，我們可以得到以下的式子:

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

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，我們將中樞元素視同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 $\mathrm{T}(n)$，則我們可以得到以下的式子:

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

移項化簡之後我們可得

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

等號兩邊同乘 n 之後，我們可得:

$$(n-1)\mathrm{T}(n) = 2\mathrm{T}(1)+2\mathrm{T}(2)+\ldots+2\mathrm{T}(n-1)+cn^2 \tag{4.1}$$

將 n 以 $n-1$ 代入後我們可得:

$$(n-2)\mathrm{T}(n-1) = 2\mathrm{T}(1)+2\mathrm{T}(2)+\ldots+2\mathrm{T}(n-2)+c(n-1)^2 \tag{4.2}$$

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:

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

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:

$$\frac{\mathrm{T}(n)}{n} = \frac{\mathrm{T}(n-1)}{n-1}+c(\frac{1}{n}+\frac{1}{n-1}) \tag{4.3}$$

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

$$\begin{aligned}\frac{\mathrm{T}(n)}{n} &= c(\frac{1}{n}+\frac{1}{n-1})+c(\frac{1}{n-1}+\frac{1}{n-2})+\ldots+c(\frac{1}{2}+1)+\mathrm{T}(1), \mathrm{T}(1)=0 \\ &= c(\frac{1}{n}+\frac{1}{n-1}+\ldots+\frac{1}{2})+c(\frac{1}{n-1}+\frac{1}{n-2}+\ldots+1)\end{aligned} \tag{4.4}$$

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 $H_n = 1+\frac{1}{2}+\frac{1}{3}+\frac{1}{3}+\ldots+\frac{1}{n}$。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:

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

最後我們可求得:

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

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 $\mathrm{O}(\log n)$。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 $\mathrm{O}(n)$。

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 (best case time complexity)	平均狀況時間複雜度 (average case time complexity)	最差狀況時間複雜度 (worst case time complexity)	空間複雜度 (space complexity)	是否穩定 (stable)	是否就地 (in place)
氣泡排序 (bubble sort)	$\mathrm{O}(n^2)$	$\mathrm{O}(n^2)$	$\mathrm{O}(n^2)$	$\mathrm{O}(1)$	是	是
插入排序 (insertion sort)	$\mathrm{O}(n)$	$\mathrm{O}(n^2)$	$\mathrm{O}(n^2)$	$\mathrm{O}(1)$	是	是
合併排序 (merge sort)	$\mathrm{O}(n\log n)$	$\mathrm{O}(n\log n)$	$\mathrm{O}(n\log n)$	$\mathrm{O}(n)$	是	否
快速排序 (quick sort)	$\mathrm{O}(n\log n)$	$\mathrm{O}(n\log n)$	$\mathrm{O}(n^2)$	$\mathrm{O}(\log n)$ (best case) $\mathrm{O}(n)$ (worst case)	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 $S = (s_1, s_2, ..., s_n)$，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 一個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 $\mathrm{O}(n^3)$ 與 $\mathrm{O}(n^2)$。但是藉由分治解答策略，我們可以設計出時間複雜度為 $\mathrm{O}(n \log n)$ 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 $S = (s_1, s_2, ..., s_n)$
Output: l, r, m，其中非空子序列 $(s_l, ..., s_r)$ 可產生最大和 m

```
1: m ← −∞
2: for i ← 1 to n do
3:     for j ← i to n do                    ▷ 以上二迴圈產生所有可能的子序列
4:         t ← 0
5:         for k ← i to j do                ▷ 此迴圈計算所有子序列的和
6:             t ← t + s_k
7:         if t > m then
8:             m ← t;l ← i;r ← 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 $S = (s_1, s_2, ..., s_n)$
Output: l, r, m，其中非空子序列 $(s_l, ..., s_r)$ 可產生最大和 m

```
1: m ← −∞
2: for i ← 1 to n do                      ▷ 此迴圈限定子序列的開頭為 s_i
3:     t ← 0
4:     for j ← i to n do                  ▷ 此迴圈限定子序列的結束為 s_j
5:         t ← t + s_j
6:         if t > m then
7:             m ← t;l ← i;r ← 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 $(s_1, s_2, ..., s_n)$ 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 ← ⌊(l + r)/2⌋                                   ▷ mid: middle
3: msl ←MCSS(S, l, mid)                                ▷ msl: max sum for left subsequence
4: msr ←MCSS(S, mid + 1, r)                            ▷ msr: max sum for right subsequence
5: msml ← −∞;t ← 0                                     ▷ msml: max sum from middle to leftmost
6: for i ← mid downto l do
7:     t ← t + s_i
8:     if t > msml then
9:         msml ← t
10: msmr ← −∞;t ← 0                                    ▷ msmr: max sum from middle+1 to rightmost
11: for i ← mid + 1 to r do
12:     t ← t + s_i
13:     if t > msmr then
14:         msmr ← 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 $\mathrm{T}(n)$，我們可以得到以下的式子:

$\mathrm{T}(n)$=$2\mathrm{T}(n/2) + cn$

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

$\mathrm{T}(n)$=$2\mathrm{T}(n/2) + cn$

$= 2(2\mathrm{T}(n/4) + c(n/2)) + cn$

$= 4\mathrm{T}(n/4) + 2cn$

$= 4(2\mathrm{T}(n/8) + (n/4)) + 2cn$

$= 8\mathrm{T}(n/8) + 3cn = ...$

$= 2^k\mathrm{T}(n/2^k) + kcn$

令 $n = 2^k$，則 $k = \log n$，我們可得:

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

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

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)。

Algorithm MCSSE(S, l, r) ▷ 可空最大連續子序列和分治演算法

Input:

S: a sequence $(s_1, s_2, ..., s_n)$ 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 ← ⌊(l + r)/2⌋                          ▷ mid: middle
3: msl ←MCSS(S, l, mid)                       ▷ msl: max sum for left subsequence
4: msr ←MCSS(S, mid + 1, r)                   ▷ msr: max sum for right subsequence
5: msml ← 0;t ← 0                             ▷ msml: max sum from middle to leftmost
6: for i ← mid downto l do
7:     t ← t + s_i
8:     if t > msml then
9:         msml ← t
10: msmr ← 0;t ← 0                            ▷ msmr: max sum from middle+1 to rightmost
11: for i ← mid + 1 to r do
12:     t ← t + s_i
13:     if t > msmr then
14:         msmr ← 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 [2]，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 [3] 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 $[t_0, t_0+\mathrm{T}]$ 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, ... 的正弦函數及餘弦函數來近似

$[t_0, t_0+\mathrm{T}]$ 區間內的 f 函數。另外，若在 $[t_0, t_0+\mathrm{T}]$ 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, ..., T/k, ... 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)，其值為為 $1/\mathrm{T}$，k 次諧波的頻率為基頻的 k 倍，其值為為 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:

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

也就是

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

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，單位為弳度/單位時間。以弳度來看，正弦及餘弦函數週期為 2π 弳度，因此 $\omega = \frac{2\pi}{\mathrm{T}}$，而 $k\omega = k\frac{2\pi}{\mathrm{T}}$ 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 $b_k, k = 1, 2, ...$ 的公式：

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

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

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

依照歐拉公式

$$e^{i\theta} = \cos\theta + i\sin\theta \tag{4.10}$$

$$e^{-i\theta} = \cos\theta - i\sin\theta \tag{4.11}$$

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 (單位為弳度)。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:

$$f(t) = \sum_{n=0}^{\infty} c_k e^{ik\omega t} \tag{4.12}$$

其中 $c_k, k = 1, 2, ...$ 依照如下的公式計算：

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

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 $c_k, k = 0, 2, ...$ 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 $x = x_0, x_1, ..., x_{n-1}$，則以下為複數型的離散傅立葉變換

$$c_k = \sum_{j=0}^{n-1} x_j e^{ijk2\pi/n}, k = 0, 1, ..., n \tag{4.14}$$

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，$j = 0, 1, ..., n-1$ 代表取樣索引值，$k = 0, 1, ..., n-1$ 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 $e^{ik2\pi/n}, k = 0, 1, ..., n-1$。我們可將式 (4.14) 改寫為：

$$c_k = \sum_{j=0}^{n-1} x_j \omega^{jk}, k = 0, 1, ..., n \tag{4.15}$$

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

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

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

根據式 (4.16) 及式 (4.17)，我們可以採取分治解題策略以 $\mathrm{O}(n \log n)$ 的複雜度完成 $c_k (k = 0, 1, ..., n-1)$ 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，$x = x_0, x_1, ..., x_7$ 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 $c_k, k = 0, 1, ..., 7$:

$c_0 = x_0 + x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7$

$c_1 = x_0 + x_1\omega + x_2\omega^2 + x_3\omega^3 + x_4\omega^4 + x_5\omega^5 + x_6\omega^6 + x_7\omega^7$

$c_2 = x_0 + x_1\omega^2 + x_2\omega^4 + x_3\omega^6 + x_4\omega^8 + x_5\omega^{10} + x_6\omega^{12} + x_7\omega^{14}$

$c_3 = x_0 + x_1\omega^3 + x_2\omega^6 + x_3\omega^9 + x_4\omega^{12} + x_5\omega^{15} + x_6\omega^{18} + x_7\omega^{21}$

$c_4 = x_0 + x_1\omega^4 + x_2\omega^8 + x_3\omega^{12} + x_4\omega^{16} + x_5\omega^{20} + x_6\omega^{24} + x_7\omega^{28}$

$c_5 = x_0 + x_1\omega^5 + x_2\omega^{10} + x_3\omega^{15} + x_4\omega^{20} + x_5\omega^{25} + x_6\omega^{30} + x_7\omega^{35}$

$c_6 = x_0 + x_1\omega^6 + x_2\omega^{12} + x_3\omega^{18} + x_4\omega^{24} + x_5\omega^{30} + x_6\omega^{36} + x_7\omega^{42}$

$c_7 = x_0 + x_1\omega^7 + x_2\omega^{14} + x_3\omega^{21} + x_4\omega^{28} + x_5\omega^{35} + x_6\omega^{42} + x_7\omega^{49}$

依照奇數與偶數項重新排列之後，我們可得

$c_o = (x_0 + x_2 + x_4 + x_6) + (x_1 + x_3 + x_5 + x_7)$

$c_1 = (x_0 + x_2\omega^2 + x_4\omega^4 + x_6\omega^6) + \omega(x_1 + x_3\omega^2 + x_5\omega^4 + x_7\omega^6)$

$c_2 = (x_0 + x_2\omega^4 + x_4\omega^8 + x_6\omega^{12}) + \omega^2(x_1 + x_3\omega^4 + x_5\omega^8 + x_7\omega^{12})$

$c_3 = (x_0 + x_2\omega^6 + x_4\omega^{12} + x_6\omega^{18}) + \omega^3(x_1 + x_3\omega^6 + x_5\omega^{12} + x_7\omega^{18})$

$c_4 = (x_0 + x_2 + x_4 + x_6) - (x_1 + x_3 + x_5 + x_7)$
$c_5 = (x_0 + x_2\omega^2 + x_4\omega^4 + x_6\omega^6) - \omega(x_1 + x_3\omega^2 + x_5\omega^4 + x_7\omega^6)$
$c_6 = (x_0 + x_2\omega^4 + x_4\omega^8 + x_6\omega^{12}) - \omega^2(x_1 + x_3\omega^4 + x_5\omega^8 + x_7\omega^{12})$
$c_7 = (x_0 + x_2\omega^6 + x_4\omega^{12} + x_6\omega^{18}) - \omega^3(x_1 + x_3\omega^6 + x_5\omega^{12} + x_7\omega^{18})$
我們可將上列的式子重新寫為
$c_0 = u_0 + v_0$
$c_1 = u_1 + \omega v_1$
$c_2 = u_2 + \omega^2 v_2$
$c_3 = u_3 + \omega^3 v_3$
$c_4 = u_0 - v_0 = u_0 + \omega^4 v_0$
$c_5 = u_1 - \omega v_1 = u_1 + \omega^5 v_1$
$c_6 = u_2 - \omega^2 v_2 = u_2 + \omega^6 v_2$
$c_7 = u_3 - \omega^3 v_3 = u_3 + \omega^7 v_3$
其中
$u_0 = x_0 + x_2 + x_4 + x_6$
$u_1 = x_0 + x_2\omega^2 + x_4\omega^4 + x_6\omega^6$
$u_2 = x_0 + x_2\omega^4 + x_4\omega^8 + x_6\omega^{12}$
$u_3 = x_0 + x_2\omega^6 + x_4\omega^{12} + x_6\omega^{18}$

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 ${\omega_8}^1 = e^{i2\pi/8}, {\omega_8}^2 = e^{i2\pi 2/8}, ..., {\omega_8}^7 = e^{i2\pi 7/8}$ 。

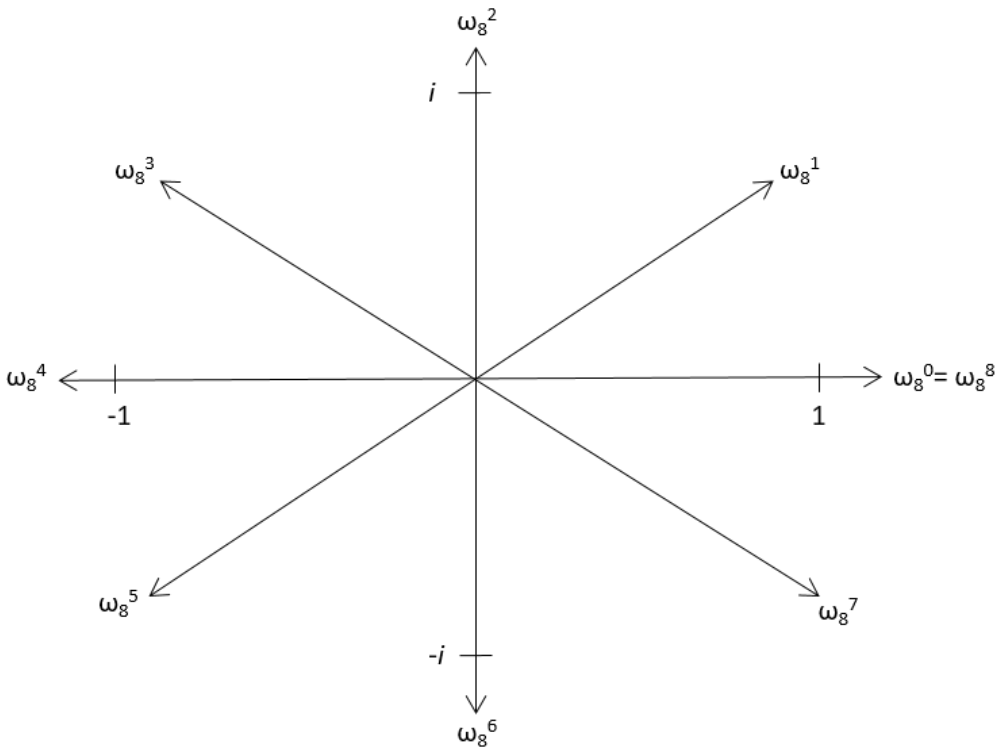

圖 4.10: 1 的 8 次方根

令 $z = \omega^2 = e^{i2\pi/4}$，我們可得

$u_0 = x_0 + x_2 + x_4 + x_6$
$u_1 = x_0 + x_2 z + x_4 z^2 + x_6 z^3$
$u_2 = x_0 + x_2 z^2 + x_4 z^4 + x_6 z^6$
$u_3 = x_0 + x_2 z^3 + x_4 z^6 + x_6 z^9$

我們可以看出來 u_0, u_1, u_2, u_3 其實是偶數項 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 $(n = 2^q, q = 1, 2, ...)$。我們有

$\omega^n = 1$
$\omega^{n/2} = -1$
因此，
$c_k = u_k + \omega^k v_k$
$c_{k+\frac{n}{2}} = u_k - \omega^k v_k$

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 $x = x_0, x_1, ..., x_{n-1}, n = 2^q, q = 0, 1, 2, ...(n$ 為 2 的指數次方)
Output: x 的傅立葉變換係數序列 $c_k = \sum_{j=0}^{n-1} x_j e^{ijk2\pi/n}, k = 0, 1, ..., n-1$

```
n ← |x|                                   ▷ n 為 x 的長度
if n = 1 then
    return x
ω_n ← e^{i2π/n}
ω ← 1                                     ▷ ω 用於臨時儲存 ω_n^k 之值
x_even ← [x_0, x_2, ..., x_{n-2}]         ▷ x_even 為偶數項序列
x_odd ← [x_1, x_3, ..., x_{n-1}]          ▷ x_odd 為奇數項序列
u ← FFT(x_even)                           ▷ u 為偶數項序列之 FFT 係數序列
v ← FFT(x_odd)                            ▷ v 為奇數項序列之 FFT 係數序列
for k ← 1 to n/2 − 1 do
    c_k ← u_k + ωv_k
    c_{k+n/2} ← u_k − ωv_k
    ω ← ωω_n                              ▷ ω 儲存下個遞迴使用的 ω_n^k 之值
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

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

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，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

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 [4]。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 graphiscs)、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、3 點順時針-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。以下先介紹線段交點演算法。

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5.1中顯示給定 14 個平面點中有 5 個極大點。

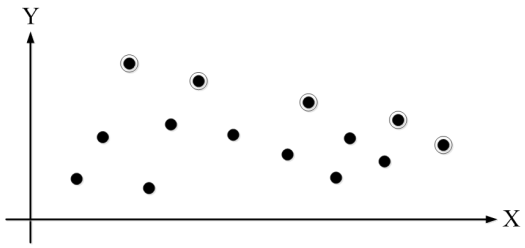

圖 5.1: 給定的 14 個平面點中有 5 個極大點 (加圓框表示)。

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5.2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。

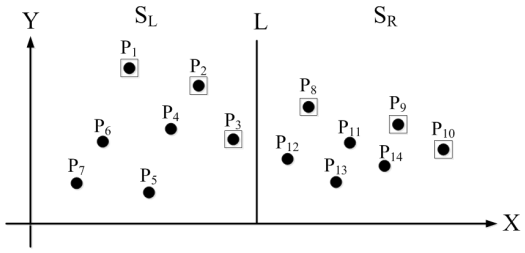

圖 5.2: 二維求極大點演算法使用分治策略解決二維求極大點問題基本概念。

以下我們分析二維求極大點演算法的時間複雜度 $\mathrm{T}(n)$，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 $\mathrm{T}(1)$=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

$\mathrm{T}(n)= 2\mathrm{T}(\frac{n}{2}) + cn \log n$
$= 4\mathrm{T}(\frac{n}{4}) + cn \log \frac{n}{2} + cn \log n$ 【註: 除第一項外共有 $\log 4 = 2$ 項】
$= 8\mathrm{T}(\frac{n}{8}) + cn \log \frac{n}{4} + cn \log \frac{n}{2} + cn \log n$ 【註: 除第一項外共有 $\log 8 = 3$ 項】
$= n\mathrm{T}(1) + cn(\log 2 + ... + \log \frac{n}{8} + \log \frac{n}{4} + \log \frac{n}{2} + \log n)$ 【註: 除第一項外共有 $\log n$ 項】
$\leq n\mathrm{T}(1) + \frac{cn \log n(\log n + \log 2)}{2}$
$= \mathrm{O}(n \log^2 n)$

以上介紹的二維求極大點演算法時間複雜度為 $\mathrm{O}(n \log^2 n)$，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 $\mathrm{O}(n)$ 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 $\mathrm{O}(n \log n)$，分析如下:

- 預先排序: $\mathrm{O}(n \log n)$
- 步驟 1: 1(這是邊界條件，也就是 $\mathrm{T}(1)$=1)
- 步驟 2: $\mathrm{O}(n)$(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: $\mathrm{T}(\frac{n}{2})$+$\mathrm{T}(\frac{n}{2})$
- 步驟 4: $\mathrm{O}(n)$(合併左半及右半的極大點)

因此總時間複雜度為

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

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): 給定一個數的集合 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 ← 0
2: for every e ∈ S do
3:     if e < a then
4:         R ← 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): 給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5.3中顯示 5 個二維平面點及其所對應的秩。

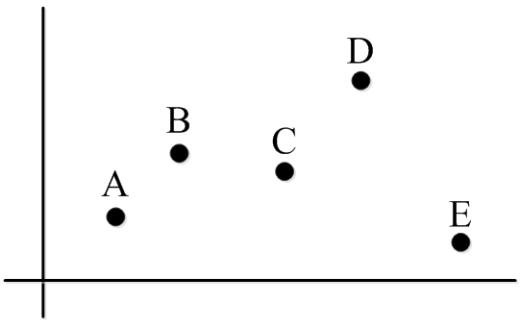

圖 5.3: 5 個二維平面點及其所對應的秩。

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 $\mathrm{O}(n^2)$。但是否如數集合的求秩一樣，數集合的求秩也具有比 $\mathrm{O}(n^2)$ 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 $\mathrm{O}(n^2)$ 還要低的時間複雜度，這個演算法描述如下:

Algorithm $\mathrm{2DRankFinding}(S, n)$

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 $\mathrm{2DRankFinding}(S_L, \lfloor n/2 \rfloor)$ 及 $\mathrm{2DRankFinding}(S_R, \lceil n/2 \rceil)$ 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5.4展示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。

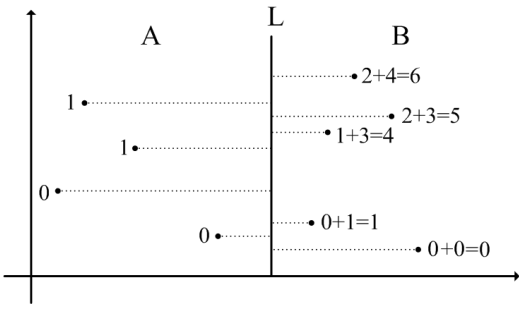

圖 5.4: 二維求秩演算法使用分治策略解決二維求秩問題基本概念。

以下我們分析二維求秩演算法的時間複雜度 $\mathrm{T}(n)$，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 $\mathrm{T}(1)=1$)
- 步驟 2: $\mathrm{O}(n \log n)$(以 X 軸值的排序尋找 X 軸值的中位數)
- 步驟 3: $\mathrm{T}(\frac{n}{2})+\mathrm{T}(\frac{n}{2})$
- 步驟 4: $\mathrm{O}(n \log n)$(以 Y 軸值的排序進行點的依序掃描)

因此總時間複雜度為

$\mathrm{T}(n)= 2\mathrm{T}(\frac{n}{2}) + cn \log n$
$= 4\mathrm{T}(\frac{n}{4}) + cn \log \frac{n}{2} + cn \log n$ 【註: 除第一項外共有 $\log 4 = 2$ 項】
$= 8\mathrm{T}(\frac{n}{8}) + cn \log \frac{n}{4} + cn \log \frac{n}{2} + cn \log n$ 【註: 除第一項外共有 $\log 8 = 3$ 項】
$= n\mathrm{T}(1) + cn(\log 2 + ... + \log \frac{n}{8} + \log \frac{n}{4} + \log \frac{n}{2} + \log n)$ 【註: 除第一項外共有 $\log n$ 項】
$\leq n\mathrm{T}(1) + \frac{cn \log n(\log n + \log 2)}{2}$
$= \mathrm{O}(n \log^2 n)$

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 $\mathrm{O}(n \log n)$，這是因為排序的時間複雜度為 $\mathrm{O}(n \log n)$，而依序檢查連續二數差的時間複雜度為 $\mathrm{O}(n)$。

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 $\mathrm{O}(n^2)$。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 $\mathrm{O}(n \log n)$ 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5.5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5.6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。

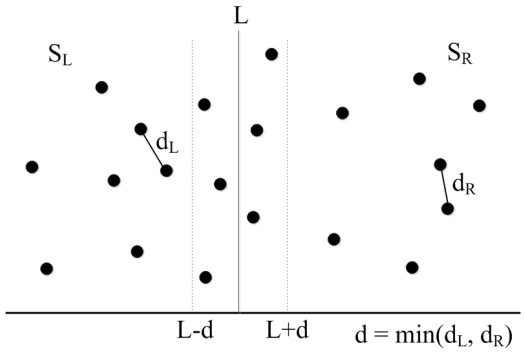

圖 5.5: 使用分治演算法解決二維最近點對問題的過程。

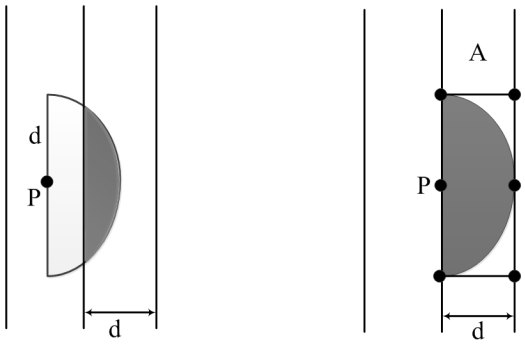

圖 5.6: 在長方形區域 A 中，最多僅 6 個點。

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

- 步驟 1: 預先排序: $\mathrm{O}(n \log n)$
- 步驟 2: 1(這是邊界條件，也就是 $\mathrm{T}(2)=1$)
- 步驟 3: $\mathrm{O}(n)$(透過預先排序尋找 X 軸值的中位數)
- 步驟 4: $\mathrm{T}(\frac{n}{2})+\mathrm{T}(\frac{n}{2})$
- 步驟 5: $\mathrm{O}(n)$(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
$\mathrm{T}(n)= 2\mathrm{T}(\frac{n}{2}) + cn = \mathrm{O}(n \log n)$

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 Caesar[1]，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 [5]。在羅馬共和國的內戰期間，凱撒使用所謂的羅馬兵法 [6]，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。

[1]Caesar 拉丁文發音為凱撒，但是英文發音已轉為西撒

Bibliography

[1] C. A. R. Hoare. Algorithm 65: Find. *Commun. ACM*, 4(7):321–322, July 1961.

[2] 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.

[3] J. Dongarra and F. Sullivan. Guest editors introduction to the top 10 algorithms. *Computing in Science Engineering*, 2(1):22–23, Jan 2000.

[4] 維基百科. 計算幾何— 維基百科, 自由的百科全書, 2019.

[5] 維基百科. 尤利烏斯·凱撒— 維基百科, 自由的百科全書, 2019.

[6] 王海林. 軍事趣味閱讀. 崧博出版, 2018.