第 3 章
Getting Started

本章學習重點

- 證明演算法的正確性
  - 使用 loop invariant，來證明演算法的正確性
- 分析演算法的運行時間 (running time)
  - 遞迴演算法: 使用遞迴樹法 (recursion tree)
  - 非遞迴演算法: 使用加總法

3.1 先行知識

1. 一般來說，我們使用虛擬代碼 (pseudo code) 來描述一個演算法。
   ([原文]: We typically describe algorithms as programs written in a pseudo code.)
2. 虛擬代碼與 “真實” 代碼的區別在於，在虛擬代碼中，我們採用任何最清晰、最簡潔的表達方式來描述給定的演算法。有時，最清晰的方式是語句，所以如果您遇到嵌入在 “真實” 代碼部份中的短語或句子，請不要感到驚訝。
   ([原文]: What separates pseudo code from “real” code is that in pseudo code, we employ whatever expressive method is most clear and concise to specify a given algorithm. Sometimes, the clearest method is English, so do not be surprised if you come across an English phrase or sentence embedded within a section of “real” code.)
3. 虛擬代碼和真實代碼之間的另一個區別是虛擬代碼通常不涉汲軟體工程的問題。為了更簡潔地傳達演算法的本質，通常會忽略數據抽象、模組化和錯誤處理的問題。
   ([原文]: Another difference between pseudo code and real code is that pseudo code is not typically concerned with issues of software engineering. Issues of data abstraction, modularity, and error handling are often ignored in order to convey the essence of the algorithm more concisely.)
4. 我們通常只專注於尋找最壞情況下的運行時間，即 n 為任意輸入大小，的最長運行時間。
   ([原文]: We shall usually concentrate on finding only the worst-case running time, that is, the longest running time for any input of size n.)
5. “平均情況” 通常與最壞情況大致相同。
   ([原文]: The “average case” is often roughly as bad as the worst case.)
6. $\theta(n^2)$ 的念法 “theta of n-squared”。
   ([原文]: $\theta(n^2)$ (pronounced “theta of n-squared”).)
7. 如果一種演算法在最壞情況下的運行時間增長較低，我們通常認為它比另一種算法更有效。

([原文]: We usually consider one algorithm to be more efficient than another if its worst-case running time has a lower order of growth.)

3.2 證明與分析演算法的正確性與運行時間的方法

首先，在 “證明演算法正確性” 方面，我們使用 loop invariant (中譯：迴圈不變量) 方法，來證明一個演算法的正確與否。在 “分析演算法的運行時間” 方面，會根據演算法是否是『遞迴』或『非遞迴』的設計，分別採用『遞迴樹法 (recursion tree)』或『加總法』來分析演算法的運行時間。

接下來，使用 insertion-sort(Algo. 3.1) 與 merge-sort(Algo. 3.3) 兩個演算法，來當示範演算法的 “正確性證明” 與 “運行時間分析” 的例子。

3.2.1 證明與分析 Insertion-Sort 的正確性與運行時間 (running time)

在開始證明前，先來看看如何使用 loop invariant (迴圈不變量) 方法。

定義 3.1 (loop invariant 方法)

使用 loop invariant 方法的 3 個步驟，來證明演算法的正確性:

步驟 1. 初始條件: 在未進入迴圈之前，假如初始屬性條件為真。
([原文] Initialization: It is true prior to the first iteration of the loop.)

步驟 2. 維持條件：假如在進入迴圈前，初始屬性條件為真，等到下次要再進入迴圈時，屬性條件仍須為真。
([原文] Maintenance: If it is true before an iteration of the loop, it remains true before the next iteration.)

步驟 3. 終止條件：當迴圈結束時，屬性條件仍須保持為真。
([原文] Termination: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.)

♣

[證明 Insertion-Sort 演算法的正確性]:

步驟 1. 初始條件: 在一開始，未進入 for loop (Algo. 3.1 的 line 2) 前，$j = 2$。所以初始的屬性條件，子陣列 $A[1..(j-1)] = A[1..(2-1)] = A[1]$，只有一個元素。明顯是已經排序好的陣列，所以初始屬性條件 (陣列已排序) 成立。

步驟 2. 維持條件: 在 for loop 裡，遞增索引 $j = k$，移動 $A[j-1], A[j-2], A[j-3], ...$, 從右到左等元素，找尋 $A[j]$ 適合的插入點 (Algo. 3.1 的 line 10)，最後子陣列 $A[1..j]$ 由原先的元素所構成且已經排序完畢。故得證，經過 for loop 遞增的 j，仍保持屬性條件 (陣列已排序) 的不變性。

步驟 3. 終結條件: 在 for loop 裡，終止條件是 $j = A.length + 1 = n + 1$，將 $j = n + 1$ 代入 Algo. 3.1 line 2 的 for loop。此時，子陣列 $A[1..n]$ 是由原先的陣列元素所構成且已經排序 (屬

Algorithm 3.1: An insertion-sort algorithm.

Input: An Array A.
Result: A sorted array A.

```
1  Insertion-Sort(A)
2  for j = 2 to A.length do
3      key = A[j];
4      i = j − 1;
5      // Insert A[j] into the sorted sequence A[1 .. j-1];
6      while i > 0 and A[i] > key do
7          A[i + 1] = A[i];
8          i = i − 1;
9      end
10     A[i + 1] = key;
11 end
```

性條件成立) 完畢。故得證，insertion sort 的演算法是正確的。

[分析 Insertion-Sort 的運行時間]:

Algorithm 3.2: Time analysis of an insertion-sort algorithm.

Input: An Array A.
Result: A sorted array A.

```
1  Insertion-Sort(A)                          // cost      times
2  for j=2 to A.length do                     //  c1         n
3      key = A[j] ;                           //  c2         n − 1
4      // Insert A[j] into the
5      sorted sequence A[1 .. j-1] ;          //  0          n
6      i = j − 1 ;                            //  c4         n − 1
7      while i > 0 and A[i] > key do          //  c5         sum_{j=2}^{n} t_j
8          A[i + 1] = A[i] ;                  //  c6         sum_{j=2}^{n} (t_j − 1)
9          i = i − 1 ;                        //  c7         sum_{j=2}^{n} (t_j − 1)
10     end
11     A[i + 1] = key ;                       //  c8         n − 1
12 end
```

由於 Insertion-Sort 是非遞迴 (non-recursive) 演算法，所以使用加總法來分析運行時間 (或時間複雜度)。

Algo. 3.2 的 c_i: 表示此行，需花 c_i 個指令步驟來執行，$i = 1, 2, 4, ..., 8$。t_j: 表示 line 7，在 $j = 2, 3, ..., n$ 時，執行測試的次數。

令 $T(n)$: 表示 Insertion-Sort 在輸入 n 個值上的運行時間，我們可將 Algo. 3.2 每列的 cost 和 times 的乘積相加得到。

(原文：$T(n)$ is the running time of Insertion-Sort on an input of n values, we sum the products of the cost and times columns to obtain.)

$\therefore T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$
$= c_1 n + c_2(n-1) + c_4(n-1) + c_5(\frac{n(n+1)}{2} - 1) + c_6(\frac{n(n-1)}{2} - 1) + c_7(\frac{n(n-1)}{2} - 1) + c_8(n-1)$
$= (\frac{c_5+c_6+c_7}{2})n^2 + (c_1 + c_2 + c_4 + (\frac{(c_5-c_6-c_7)}{2}) + c_8)n - (c_2 + c_4 + c_5 + c_8)$
$= an^2 + bn + c$
$= \theta(n^2)$

故，經由 "加總法" 得知，Insertion-Sort 的運行時間 (或時間複雜度) 為 $\theta(n^2)$。

3.2.2 證明與分析 Merge-Sort 的正確性與運行時間 (running time)

在開始使證明與分析 Merge-Sort(Algo. 3.3) 之前，先介紹一下 Merge-Sort 的設計方法。Merge-Sort 是由 divide-and-conquer 方法，設計出來的遞迴 (recursive) 演算法。

定義 3.2 (divide-and-conquer 方法)

使用 divide-and-conquer，來設計遞迴演算法的 3 個步驟:

步驟 1. 將問題切割為多個子問題，這些子問題是原先問題的較小實例。
([原文] Divide the problem into a number of subproblems that are smaller instances of the same problem.)

步驟 2. 通過遞迴解決子問題，來征服子問題。而且，如果子問題的大小足夠小，便能夠以直接的方式解決子問題。
([原文] Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.)

步驟 3. 將子問題的解組合成原始問題的解。
([原文] Combine the solutions to the sub-problems into the solution for the original problem.)

♣

Merge-sort(Algo. 3.3) 透過遞迴的呼叫自己，來將問題切割成較小的子問題 (由 Algo. 3.3 的 line 3，知道每次將問題大小切一半)，直到子問題大小足夠小後 (例如：只剩下 2 個元素比較)，解完後，在呼叫副程式 Merge(Algo. 3.4) 把每個小問題的解組合起來，得到最後已排序結果。

Merge(Algo. 3.4) 與 merge-sort(Algo. 3.3) 可以分別搭配圖 3.1 與 圖 3.2 所舉的例子，來了解 merge 與 merge-sort 的流程。

[證明 Merge-Sort 的正確性]:

∵ Merge-Sort 遞迴呼叫 Merge(Algo. 3.4) 來完成排序。∴ 只要證明 Merge 的 loop invariant 即可。

步驟 1. 初始條件: 在還未進入 Merge(Algo. 3.4) line 15-23 的 for loop 之前 $k = p$，所以子陣列 $A[p..k-1]$ 為空 (空陣列也是已排序好陣列，所以初始屬性條件成立)。此空的子陣列包含 L 與 R ，0 ($k - p = 0$) 個最小元素且 $i = j = 1$，$L[i]$ 與 $R[j]$ 是 L 與 R 陣列尚未複製到 A 的最小的元素。

Algorithm 3.3: A merge-sort algorithm.

Input: An Array A. Two indexes are p and r.
Result: A sorted array A.

```
Merge-Sort(A, p, r)
if p < r then
    q = ⌊(p + r)/2⌋;
    Merge-Sort(A, p, q);
    Merge-Sort(A, q + 1, r);
    Merge(A, p, q, r);
end
```

Algorithm 3.4: A merge algorithm.

Input: An Array A. Three indexes are p, q and r.

```
Merge(A, p, q, r)
n_1 = q − p + 1;
n_2 = r − q;
Let L[1..n_1 + 1] and R[1..n_2 + 1] be new arrays;
for i = 1 to n_1 do
    L[i] = A[p + i − 1];
end
for j = 1 to n_2 do
    R[j] = A[q + j];
end
L[n_1 + 1] = ∞;
R[n_2 + 1] = ∞;
i = 1;
j = 1;
for k = p to r do
    if L[i] ≤ R[j] then
        A[k] = L[i];
        i = i + 1;
    else
        A[k] = R[j];
        j = j + 1;
    end
end
```

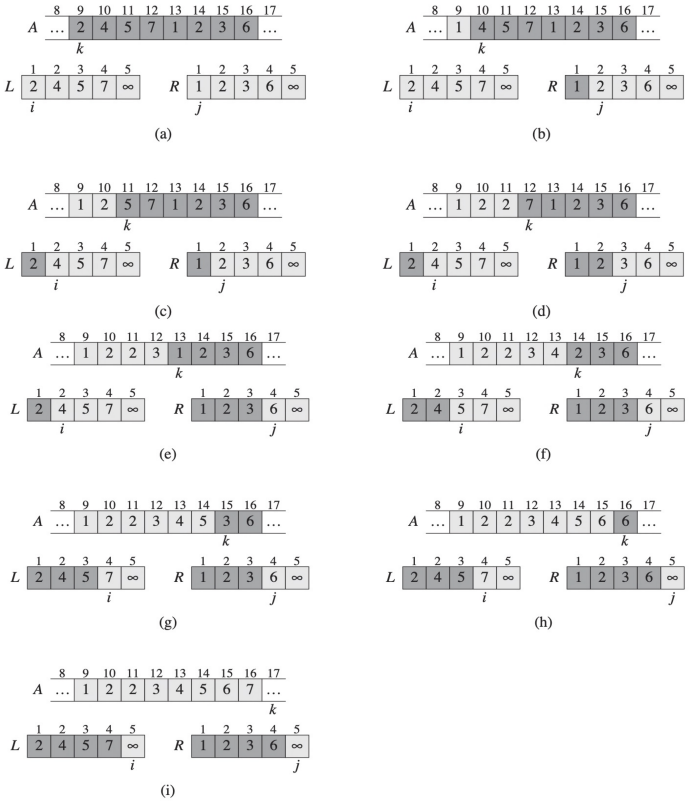

圖 3.1: A merge example. The operation of line 13-23 in the call Merge($A, p = 9, q = 12, r = 16$), when the subarray $A[9..16]$ contains the sequence $(2, 4, 5, 7, 1, 2, 3, 6)$.

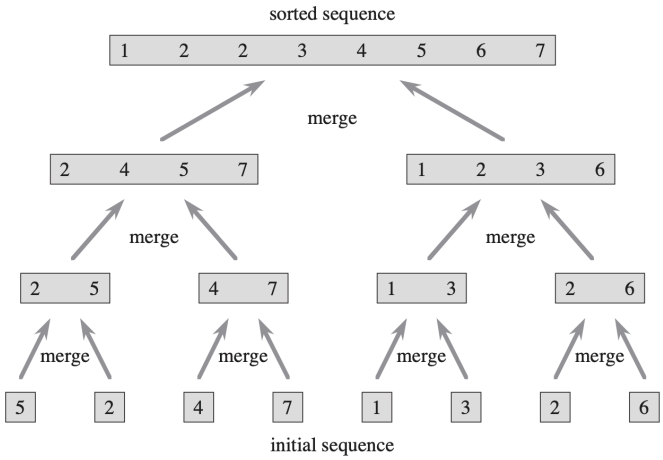

圖 3.2: A merge sort example. The operation of merge sort on the array $A = (5, 2, 4, 7, 1, 3, 2, 6)$. The lengths of the sorted sequences being merged increase as the algorithm progress from bottom to top.

步驟 2. 維持條件: 在進入 Merge 的 for loop(Algo. 3.4 line 15-23) 後，首先，假設 $L[i] \le R[j]$。$L[i]$ 是尚未拷貝回 A 的最小元素。因爲 $A[p..k-1]$ 包含 $k-p$ (= $(k-1)-p+1$) 個最小元素，但在 Merge(Algo. 3.4) line 17 後，拷貝 $L[i]$ 到 $A[k]$ 後，子陣列 $A[p..k]$ 包含 $k-p+1$ 個最小元素，且遞增 k(Merge(Algo. 3.4) line 15) 與 i(Merge(Algo. 3.4) line 18)，子陣列 $A[p..k]$ 維持已排序屬性。假若 $L[i] > R[j]$，則執行 Algo. 3.4 line 20-21，子陣列 $A[p..k]$ 也是維持已排序屬性，故維持條件成立。

Step 3. 終止條件: 終止條件是 $k = r+1$。此時，$A[p..k-1] = A[p..r]$ 包含 $L[1..n_1+1]$ 與 $R[1..n_2+1]$ 中，$k-p = r-p+1$ 個已排序的最小元素。L 與 R 包含 $n_1+n_2+2 = r-p+3$ 個元素。所以，除了 2 個最大的哨兵元素 ($L[n_1+1]$ 與 $R[n_2+1]$)，其餘元素皆已拷貝進 A 且排序完畢。故得證 Merge-Sort 演算法的正確性無誤。

[分析 Merge-Sort 的運行時間]:

Merge-Sort 是採用 divide-and-conquer 方法，設計出的『遞迴』演算法，所以我們使用『遞迴樹』法，來分析 Merge-Sort 的運行時間。在開始分析之前，先定義 $T(n)$、$D(n)$ 與 $C(n)$ 符號，來幫助分析 Merge-Sort 的運行時間分析。

$$T(n) = \begin{cases} \theta(1) & \text{if } n \le c, \\ aT(n/b) + D(n) + C(n) & \text{otherwise.} \end{cases}$$

符號說明:

- $T(n)$: 表示解決問題大小為 n 所需的運行時間。如果問題的規模足夠小 ($n \le$ some constant c)，小到能直接的解決問題 (例如：排序 2 個元素)，並只花常數時間的話，我們將其寫為 $\theta(1)$。
  ([原文] the running time on a problem of size n. If the problem size is small enough, say $n \le c$ for some constant c, the straightforward solution takes constant time, which we write as $\theta(1)$.)
- $D(n)$: 表示將問題分割為子問題，所需的時間。
  ([原文] time to divide the problem into subproblems.)
- $C(n)$: 表示將子問題的解組合成原先問題的解，所需的時間。
  ([原文] time to combine the solutions to the subproblems into the solution to the original problem.)

Merge-Sort 的運行時間 $T(n)$ 分析:

- 分割: 分割的方法，是直接從中間切一刀 (Algo. 3.3 line 3)，一分為二個較小的子問題。所以 $D(n) = \theta(1)$。
  ([原文] Divide: The divide step just computes the middle of the subarray, which takes constant time. Thus, $D(n) = \theta(1)$.)
- 征服: 我們遞迴地解決兩個子問題，每個子問題的大小為 $n/2$，這對運行時間貢獻了 $2T(n/2)$。

([原文] Conquer: We recursively solve two subproblems, each of size $n/2$, which contributes $2T(n/2)$ to the running time.)

- 組合: 我們已經注意到，在 n 個元素的子陣列上，Merge 過程需要花費時間 $\theta(n)$，因此 $C(n) = \theta(n)$。
  ([原文] Combine: We have already noted that the Merge procedure on an n-element subarray takes time $\theta(n)$, and so $C(n) = \theta(n)$.)

所以，Merge-Sort 的運行時間可以寫成，下列數學式子：

$$\therefore T(n) = \begin{cases} \theta(1) = c & \text{if } n = 1, \\ 2T(n/2) + \theta(1) + \theta(n) = 2T(n/2) + \theta(n) = 2T(n) + cn & \text{if } n > 1. \end{cases}$$

最後，使用遞回樹來解 $T(n) = 2T(n/2) + cn$，如圖 3.3 以獲得 Merge-Sort 最後運行時間的結果 $\theta(n \lg n)$。

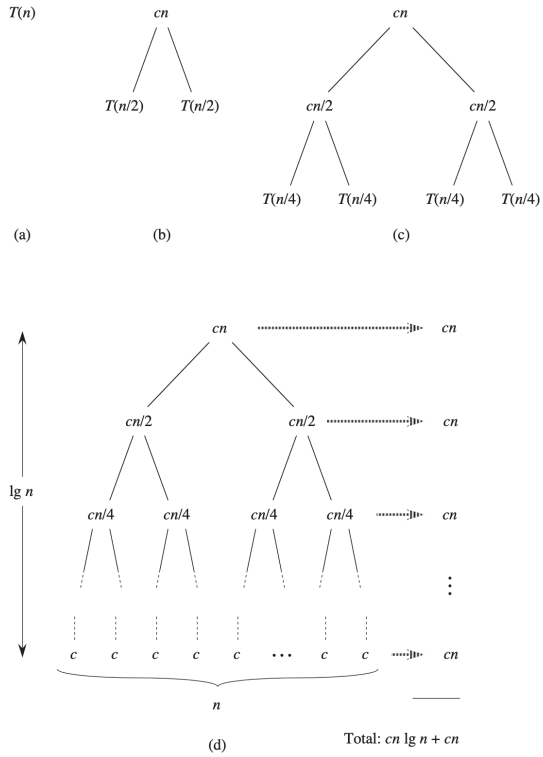

圖 3.3: How to construct a recursion tree for the recurrence $T(n) = 2T(n/2) + cn$. The total cost is $cn \lg n + cn = \theta(n \lg n)$