# 前言

還在不久前，除了電腦科學家或數學家之外，幾乎沒有人聽過「演算法」，然而，隨著電腦在現代生活中日益普及，這個詞已不再如此專業了。環顧家裡可以發現，即使是不起眼的地方都有演算法在執行，例如微波爐、洗衣機，當然還有電腦。你可能會讓演算法推薦音樂，或規劃駕駛路線。我們的社會使用演算法來量刑（姑且不論好壞）。你甚至可能依靠演算法來維持生命，或至少避免喪命，例如，汽車或醫療設備裡的控制系統[1]。新聞報導幾乎每天都會提到「演算法」這個詞。

因此，除了計算機科學學生和從業者需要了解演算法之外，作為世界公民的你也必須了解演算法。一旦你了解演算法，你就可以教別人何謂演算法、它們如何運作，以及它們有什麼限制。

本書將全面介紹電腦演算法的現代研究成果，我們會介紹許多演算法，並對它們進行相當深入的探討，同時讓所有程度的讀者都可以理解本書的內容。我們會進行所有分析，有的很簡單，有的比較複雜。我們試著在不犧牲深度和數學嚴謹性的前提下，盡量清楚地解釋它們。

本書的每一章都會介紹一種演算法、一種設計技術、一種應用領域，或一個相關的主題。為了幫助幾乎沒有程式設計基礎的讀者理解，我們將用英文和虛擬碼（pseudocode）來描述演算法。本書使用 231 張圖表來說明演算法如何運作，其中有些圖表包含多個部分。我們強調**效率**這個設計標準，所以也會仔細地分析演算法的執行時間。

本書主要是讓大學或研究所的演算法或資料結構課程使用的。由於本書探討演算法設計的工程問題和數學層面，所以專業技術人員也可以用來自學。

在第四版中，我們再次更新了全書。我們更新的範圍很廣泛，包括加入新章節、彩色插圖，以及幫助你更專心的寫作風格。

1　若要了解演算法如何在各方面影響日常生活，可參考 Fry [162]。

## 第四版的修改

正如我們在第二版與第三版的修改說明中說過的，取決於你的觀點，你可能認為本書的修改很少，也可能認為很多。從目錄來看，第四版涵蓋了第三版大部分的章節。我們移除了三章與若干小節，但也加入新的三章和若干新的小節。

我們保留前三版採用的混合式架構，同時考慮問題領域和技術，而不是只根據其中一個元素來組織各章。本書有技術型章節，介紹分治法、動態規劃、貪婪演算法、平攤分析、擴大資料結構、NP 完備性，以及近似演算法。但本書也用完整的部分來介紹排序、動態集合的資料結構，以及圖問題演算法。雖然我們在設計演算法和分析演算法時知道該使用哪些技巧，但真正的問題幾乎都不會提示你該採用哪種技巧來處理。

第四版有些全書範圍的修改，有些則針對特定章節。我們將最重要的修改整理如下：

- 我們加入 140 道新習題與 22 個新挑戰。我們也改進了許多舊習題與挑戰，通常源自讀者的回饋（感謝所有提供建議的讀者）。
- 我們使用彩色印刷！我們和 MIT Press 的設計師一起挑選幾種顏色，希望賞心悅目地傳達資訊（很開心可以用紅色與黑色來展示紅黑樹！）。為了更方便閱讀，我們將既定術語、虛擬碼註解，以及索引中的頁數印成彩色的。
- 我們幫虛擬碼加上灰底色背景來突顯它們。虛擬碼不一定放在初次提及它們的那一頁，若是如此，內文會指出它在哪裡。我們也會在引用別頁的數學式、定理、引理和推論時指出頁數。
- 我們刪除比較沒有人教導的主題，移除探討斐波那契堆積、van Emde Boas 樹、計算幾何的章節。此外，也移除了以下的主題：最大子陣列問題、實作指標與物件、完美雜湊、隨機建構二元搜尋樹、擬陣、計算最大流量的 push-relabel 演算法、迭代式快速傅立葉轉換法、線性規劃的單體演算法，以及整數分解。你可以在我們的網站找到被刪除的所有內容：*http://mitpress.mit.edu/algorithms/*。
- 我們重新校閱了整本書，並改寫一些句子、段落和小節，來讓文章更清楚、更個人化、更性別中立。例如，我們將上一版的「traveling-salesman problem」改成「traveling-salesperson problem」。我們認為，工程學和科學（包括我們自己的計算機科學領域）應該一視同仁地歡迎所有人（但是第 13 章有一處讓我們很為難，該章需要用一個詞來代表父母的兄弟姐妹，這在英文裡沒有性別中立的詞，所以我們只能遺憾地使用「uncle」）。

- 我們也修改了各章的說明、參考文獻和索引，以反映演算法領域自第三版問世以來的急劇成長。
- 我們修正了錯誤，按照第三版的勘誤網頁來糾正內容。當我們緊鑼密鼓地準備這一版時也收到一些錯誤回報，雖然它們沒有被貼到第三版的網頁上，但我們直接在這一版修正了（再次感謝協助發現問題的讀者們）。

以下是第四版的具體改變：

- 我們更改了第 3 章的名稱，並在探討正式定義之前，加入介紹漸近表示法的一節。
- 我們大量修改第 4 章，以改善其數學基礎，讓它更穩固和直觀。我們加入演算法遞迴式的概念，並且更嚴謹地處理在遞迴式中忽略向下取整（floor）和向上取整（ceiling）的問題。我們為主定理（master theorem）的第二種情況加入多對數因子，並提供主定理的「連續」版本的嚴謹證明。我們也介紹強大且通用的 Akra-Bazzi 方法（但不證明）。
- 我們稍微修改第 9 章的確定性順序統計演算法，並重新整理關於隨機和確定性順序統計演算法的分析。
- 除了堆疊和佇列之外，第 10.1 節也討論陣列和矩陣的儲存方法。
- 我們在介紹雜湊表的第 11 章加入現代雜湊函數處理方法。本章也強調，當底層的硬體快取適合用來做本地搜尋時，如何用線性探測來有效處理碰撞。
- 為了取代第三版第 15 章中關於擬陣的小節，我們將離線快取的一個挑戰改成完整的小節。
- 現在第 16.4 節更直觀地解釋潛能函數，以分析表格的加倍和減半。
- 將介紹擴充資料結構的第 17 章從第三部分移至第五部分，因為我們認為這種技術不屬於基本教材。
- 我們在新的第 25 章裡介紹二部圖的配對，並介紹一個尋找最多配對的演算法，以解決穩定婚配問題，和尋找最大權重配對組合（稱為「分配問題」）。
- 在介紹任務平行計算的第 26 章，我們改用現代術語，包括該章的名稱。
- 第 27 章是另一個新章節，介紹線上演算法。線上演算法的輸入是逐漸進來的，不是在演算法開始執行時就全部獲得的。本章介紹幾個線上演算法的例子，包括：等了多久電梯之後，就要改走樓梯、透過「移至前面」捷思法（heuristic）來維護鏈接串列，以及評估快取的替換策略。

# 13 紅黑樹

第 12 章提到高度為 $h$ 的二元搜尋樹可以在 $O(h)$ 時間內支援任何基本動態集合操作，例如 SEARCH、PREDECESSOR、SUCCESSOR、MINIMUM、MAXIMUM、INSERT 與 DELETE。因此，如果搜尋樹的高度較低，集合操作很快。然而，如果高度很高，集合操作的執行速度可能不如使用鏈接串列。紅黑樹是許多搜尋樹方案中的一種，它是「平衡」的，以保證基本的動態集合操作在最壞情況下有 $O(\lg n)$ 時間。

## 13.1 紅黑樹的特性

**紅黑樹**是一種二元搜尋樹，它的每一個節點都有一個額外的儲存位元：該節點的**顏色**，可以是 RED 或 BLACK。紅黑樹藉著限制從根節點到葉節點的任何簡單路徑上的節點顏色，來確保沒有任何路徑比其他路徑長兩倍以上，因此樹是接近**平衡**的。事實上，正如我們即將看到的，有 $n$ 個鍵的紅黑樹的高度最多是 $2\lg(n+1)$，也就是 $O(\lg n)$。

這種樹的每個節點都包含 *color*、*key*、*left*、*right* 與 *p* 等屬性。如果一個節點的子節點或父節點不存在，則該節點的相應指標屬性是 NIL 值。你可以把這些 NIL 視為指向二元搜尋樹的葉節點（外圍節點）的指標，而正常的、附帶鍵的節點是樹的內部節點。

紅黑樹是滿足以下**紅黑特性**的二元搜尋樹：

1. 每個節點非紅即黑。
2. 根是黑的。
3. 每一個葉節點（NIL）都是黑的。
4. 如果節點是紅的，它的兩個子節點都是黑的。
5. 對每個節點而言，從該節點到後代葉節點的所有簡單路徑都有相同數量的黑色節點。

圖 13.1(a) 是一個紅黑樹的例子。

為了方便處理紅黑樹程式中的邊界條件（boundary condition），我們用一個哨符來代表 NIL（見第 250 頁）。紅黑樹 $T$ 的哨符 $T.nil$ 是一個物件，它的屬性與樹中的普通節點相同。它的**顏色**屬性是 BLACK，其他屬性（$p$、$left$、$right$、$key$）可以是任意值。如圖 13.1(b) 所示，指向 NIL 的所有指標都換成指向哨符 $T.nil$ 的指標。

為什麼要使用哨符？哨符可讓我們將節點 $x$ 的 NIL 子節點視為「父節點為 $x$ 的普通節點」。另一種設計是讓樹的每個 NIL 使用不同的哨符節點，如此一來，每個 NIL 的父節點都有明確的定義。但是，這種做法毫無必要地浪費空間。我們只要用一個哨符 $T.nil$ 就可以代表所有的 NIL 了，包括所有的葉節點和根的父節點。哨符的 $p$、$left$、$right$ 和 $key$ 屬性的值不重要。紅黑樹程序可以在哨符中放置任何數值，來讓程式更簡單。

通常我們只對紅黑樹的內部節點感興趣，因為它們保存鍵值。本章接下來的紅黑樹圖會省略葉節點，如圖 13.1(c) 所示。

我們用**黑高**（***black-height***）來稱呼從節點 $x$ 往下走到一個葉節點的任何簡單路徑上的黑色節點數量，以 $\text{bh}(x)$ 來表示。根據特性 5，黑高定義完善，因為從節點開始往下走的所有簡單路徑都有相同數量的黑節點。紅黑樹的黑高是它的根節點的黑高。

下面的引理說明為何紅黑樹是優秀的搜尋樹。

## 引理 13.1

有 $n$ 個內部節點的紅黑樹的高最多是 $2\lg(n+1)$。

**證明**　我們先證明，根為 $x$ 的任何子樹至少有 $2^{\text{bh}(x)}-1$ 個內部節點。我們藉著對 $x$ 的高進行歸納來證明這個論點。如果 $x$ 的高度是 0，那麼 $x$ 一定是葉節點（$T.nil$），根為 $x$ 的子樹的確至少有 $2^{\text{bh}(x)}-1=2^0-1=0$ 個內部節點。在歸納步驟，考慮節點 $x$ 有正的高度，而且是內部節點，那麼節點 $x$ 有兩個子節點，其中的一個或兩個可能是葉節點。如果有一個子節點是黑的，那麼它對 $x$ 的黑高貢獻 1，但對它自己的黑高沒有貢獻。如果有一個子節點是紅色的，那麼它對 $x$ 的黑高和它自己的黑高都沒有貢獻。因此，每個子節點的黑高都是 $\text{bh}(x)-1$（如果它是黑的）或 $\text{bh}(x)$（如果它是紅的）。因為 $x$ 的子節點的高小於 $x$ 本身的高，我們可以用歸納假設得出結論，每個子節點至少有 $2^{\text{bh}(x)-1}-1$ 個內部節點。因此，根為 $x$ 的子樹至少有 $(2^{\text{bh}(x)-1}-1)+(2^{\text{bh}(x)-1}-1)+1=2^{\text{bh}(x)}-1$ 個內部節點，這個說法得證。

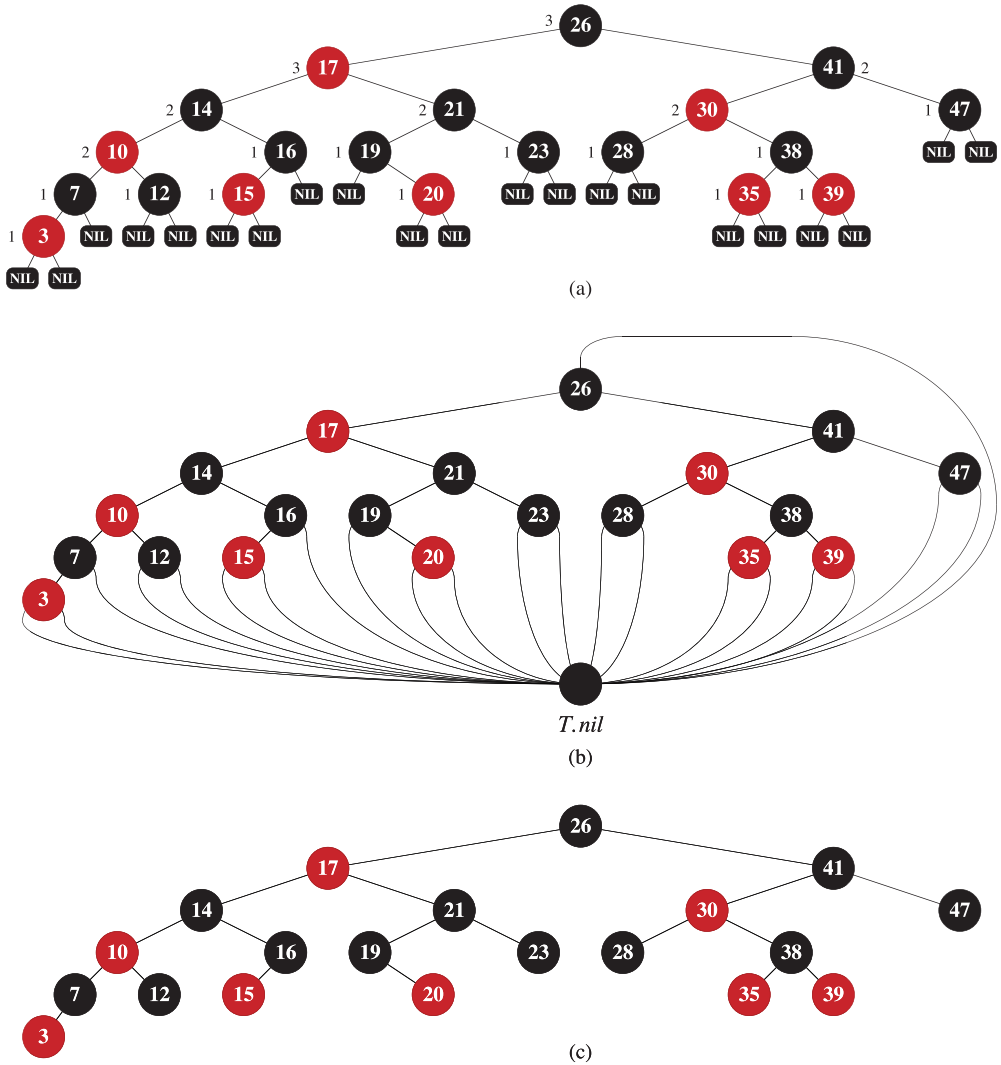


**圖 13.1** 紅黑樹。在紅黑樹裡的每個節點都是紅色或黑色的，紅色節點的子節點都是黑色的，從一個節點到一個後代葉節點的每一條簡單路徑都包含相同數量的黑色節點。**(a)** 標上 NIL 的葉節點都是黑色的。每個非 NIL 節點都標有其黑高，其中 NIL 的黑高為 0。**(b)** 同樣的紅黑樹，但每個 NIL 都被換成單一的哨符 *T.nil*，它一定是黑色的，本圖省略黑高。根節點的父節點也是哨符。**(c)** 同一棵紅黑樹，但完全省略葉節點和根的父節點。本章的其餘內容將使用這種繪圖風格。

為了完成引理的證明，設 $h$ 是樹的高度。根據特性 4，從根節點到葉節點的任何簡單路徑上，至少有一半的節點是黑色，不包括根節點。因此，根節點的黑高至少是 $h/2$，所以，

$$n \geq 2^{h/2} - 1$$

把 1 移到左邊，對兩邊取對數，得到 $\lg(n+1) \geq h/2$ 或 $h \leq 2\lg(n+1)$。 ■

這個引理的直接結論是，動態集合操作 SEARCH、MINIMUM、MAXIMUM、SUCCESSOR 和 PREDECESSOR 在紅黑樹上的執行時間都是 $O(\lg n)$，因為每個操作在高度為 $h$ 的二元搜尋樹上的執行時間都是 $O(h)$（見第 12 章），且 $n$ 個節點的紅黑樹都是高度為 $O(\lg n)$ 的二元搜尋樹（當然，在第 12 章的演算法中，針對 NIL 的參考必須換成 *T.nil*）。儘管第 12 章的程序 TREE-INSERT 和 TREE-DELETE 在接收紅黑樹作為輸入時，執行時間是 $O(\lg n)$，但你不能直接使用它們來實作動態集合操作 INSERT 和 DELETE，它們不一定能維持紅黑特性，所以最終可能不會產生一棵合格的紅黑樹。本章其餘內容將展示如何在 $O(\lg n)$ 時間內對著紅黑樹進行插入和刪除。

## 習題

### 13.1-1
用圖 13.1(a) 的風格，繪製包含鍵 $\{1, 2, \ldots, 15\}$，高度為 3 的完整二元搜尋樹。加入 NIL 葉節點，並以三種不同的方式為節點上色，使得最終的紅黑樹的黑高為 2、3 和 4。

### 13.1-2
畫出以鍵 36 對著圖 13.1 的樹呼叫 TREE-INSERT 之後的紅黑樹。如果插入的節點被畫成紅色，那麼產生的樹是紅黑樹嗎？如果畫成黑色呢？

### 13.1-3
我們將**寬鬆紅黑樹**定義成滿足紅黑特性 1、3、4 和 5 的二元搜尋樹，但它的根可能是紅色或黑色。考慮一棵寬鬆紅黑樹 $T$，它的根是紅色的。如果將 $T$ 的根改成黑色，但不做其他改變，最終的樹是紅黑樹嗎？

### 13.1-4
假設在紅黑樹裡的每個黑節點都「吸收」了它的所有紅色子節點，因此任何紅色節點的子節點都會變成它的黑色父節點的子節點（忽略鍵發生的事情）。在一個黑色節點的所有紅色子節點都被吸收後，黑色節點可能是多少度（degree）？最終的樹的葉節點的深度為何？

## 貪婪選擇特性

第一個關鍵因素是**貪婪選擇特性**：你可以藉著做出局部最佳（貪婪）選擇來建構一個整體最佳解決方案。換句話說，當你考慮該做出哪個選擇時，你會做出對當下的問題看似最佳的選擇，而不考慮子問題產生的結果。

這就是貪婪演算法與動態規劃不同的地方。在動態規劃中，你會在每一步做出選擇，但這個選擇通常取決於子問題的解。因此，我們通常用「由下而上」法來解決動態規劃問題，從較小的子問題處理到較大的子問題（你也可以由上而下地解決問題，但要進行記憶化。當然，即使程式碼是由上而下工作的，你仍然必須在做出選擇之前解決子問題）。在貪婪演算法中，你會做出目前看似最佳的選擇，然後解決剩下的子問題。貪婪演算法做出來的選擇可能取決於迄今為止的選擇，但它不取決於任何未來的選擇或子問題的解。因此，動態規劃在做出第一個選擇之前先解決子問題，貪婪演算法則是在解決任何子問題之前做出第一個選擇。動態規劃演算法是由下而上進行的，貪婪策略通常是由上而下進行的，做出一個又一個貪婪的選擇，將每個問題縮小成更小的問題。

當然，你要證明每一步的貪婪選擇都能產生整體最佳解。做法通常像定理 15.1 的情況一樣，先檢查某個子問題的整體最優解，然後展示如何修改解決方案，將貪婪選擇換成其他選擇，進而得到一個類似但更小的子問題。

做出貪婪的選擇通常比考慮更廣泛的選項更有效率。例如，在活動選擇問題中，假設活動已經按照結束時間單調遞增順序進行排序，那麼每項活動只需要檢查一次。使用適當的資料結構（通常是優先佇列）來預先處理輸入，往往可以讓我們快速地做出貪婪的選擇，進而產生高效的演算法。

## 最佳子結構

正如我們在第 14 章中看到的，如果一個問題的最佳解包含子問題的最佳解，那麼此問題就展現出**最佳子結構**。這個特性是評估動態規劃是否適用的關鍵要素，對貪婪演算法而言也至關重要。舉一個最佳子結構的例子，回顧一下第 15.1 節如何證明：若子問題 $S_{ij}$ 的最佳解包含活動 $a_k$，那麼它一定也包含子問題 $S_{ik}$ 和 $S_{kj}$ 的最佳解。有了這個最佳子結構，我們認為，如果你知道該將哪個活動當成 $a_k$，你就可以藉著選擇 $a_k$ 以及子問題 $S_{ik}$ 與 $S_{kj}$ 的最佳解裡面的所有活動來建構 $S_{ij}$ 的最佳解。這個關於最佳子結構的觀察產生了定義最佳解之值的遞迴式 (15.2)。

當你將最佳子結構用於貪婪演算法時，你通常會採取比較直接的做法。如上所述，你可以假設你在原問題中進行了貪婪選擇，並到達一個子問題，你只需要證明子問題的最佳解，再加上已經做出來的貪婪選擇，就可以得到原問題的最佳解。這種做法暗中對子問題進行歸納，以證明在每一步都做出貪婪選擇可以產生最佳解。

## 貪婪 vs. 動態規劃

由於貪婪和動態規劃策略都利用最佳子結構，你可能會忍不住為貪婪演算法可以解決的問題設計動態規劃解決方案，或者誤以為貪婪法可行，其實需要使用動態規劃解決方案。為了說明這兩種技術之間的微妙差異，我們來研究一個經典的優化問題的兩個變體。

以下是 ***0-1* 背包問題**。有一位入店行竊的小偷想用一個最多可容納 $W$ 磅贓物的背包來竊取最貴的物品。小偷可以選擇拿走 $n$ 個物品的任何子集合。第 $i$ 件物品價值 $v_i$ 美元，重量為 $w_i$ 磅，其中 $v_i$ 和 $w_i$ 是整數。小偷應該拿走哪些物品？（這個問題稱為 0-1 背包問題的原因是，小偷只能選擇拿走或留下每一件物品，不能拿走物件的一部分，或拿走同一項物品兩次）。

**小數背包問題**的規定相似，但小偷可以拿部分的物品，而不是只能對每一項物品做出二元（0-1）選擇。你可以將 0-1 背包問題中的物品想成金塊，而小數背包問題中的物品則比較像金粉。

這兩個背包問題都有最佳子結構特性。如果最有價值負重（已放在背包裡的物品）不超過 $W$ 磅，裡面有物品 $j$，那麼其餘的負重必定是小偷可從不含 $j$ 的 $n-1$ 個原本物品中拿走，而且重量不超過 $W-w_j$ 磅的最有價值負重。對於相似的小數問題，如果最有價值負重不超過 $W$ 磅，裡面有重量為 $w$ 的物品 $j$，那麼其餘的負重必定是小偷可以從 $n-1$ 個原來的物品和重量為 $w_j-w$ 的物品 $j$ 中選出來的、總重量不超過 $W-w$ 磅的最有價值負重。

雖然這兩個問題相似，但貪婪策略可以處理小數背包問題，卻不能解決 0-1 問題。為了處理小數問題，你要先計算每件物品的每磅價值 $v_i/w_i$。按照貪婪策略，小偷一開始要盡可能地裝入每磅價值最高的物品。如果該物品已全部裝入，但小偷還能放入更多物品，那麼小偷就要盡可能地拿走每磅價值最高的下一個物品，以此類推，直到到達重量限制 $W$ 為止。因此，如果將物品按每磅價值的順序排序，貪婪演算法的執行時間為 $O(n \lg n)$。習題 15.2-1 會要求你證明小數背包問題具有貪婪選擇特性。

為了說明這種貪婪策略無法處理 0-1 背包問題，考慮圖 15.3(a) 的問題實例。這個例子有三件物品和一個可以裝 50 磅的背包。物品 1 有 10 磅重，價值 $60。物品 2 有 20 磅重，價值 $100。物品 3 有 30 磅重，價值 $120。因此，物品 1 的每磅價值 $6，高於物品 2（每磅 $5）與物品 3（每磅 $4）的每磅價值。所以，貪婪策略會先拿物品 1。然而，從圖 15.3(b) 的案例分析中可以看出，最佳解是拿物品 2 和 3，將物品 1 留到最後。先拿物品 1 的兩種方案都是次優的。

然而，對於小數問題而言，貪婪策略（即先拿物品 1）確實可產生最佳解，如圖 15.3(c) 所示。在 0-1 背包問題中，選擇物品 1 是不可行的，因為小偷無法將背包裝滿，未裝滿的空間會降低負重的每磅有效價值。在 0-1 背包問題中，當你考慮是否將一個物品放進背包裡時，你必須先比較包含該物品的子問題的解與不包含該物品的子問題的解，然後才能做出選擇。這種形式的問題會產生許多重疊的子問題，這是動態規劃的標誌。而且，正如習題 15.2-2 要求你做的那樣，你可以使用動態規劃來解決 0-1 問題。

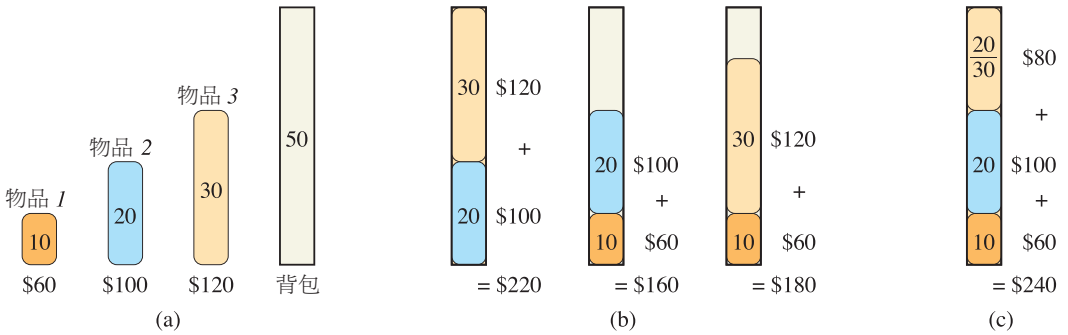


**圖 15.3** 證明貪婪策略不適合處理 0-1 背包問題的例子。**(a)** 小偷必須從所示的三個物品中選擇一個重量不超過 50 磅的子集合。**(b)** 最佳子集合包含物品 2 和 3。包含物品 1 的解決方案都是次優的，儘管物品 1 的每磅價值最大。**(c)** 在處理小數背包問題時，按照每磅最大價值的順序來選擇物品可以得到最佳解。

## 習題

### 15.2-1

證明小數背包問題有貪婪選擇特性。

# 26 平行演算法

在本書中，絕大多數的演算法都是**串行**（***serial***）**演算法**，適合在一次只執行一條指令的單處理器計算機上運行。在這一章，我們要擴展演算模式，加入**平行演算法**，也就是同時執行多個指令。具體來說，我們將探索任務平行演算法的優雅模型，它適合用來進行演算法設計和分析。我們的研究重點是分叉聚合平行演算法，這是最基本且最容易理解的一種任務平行演算法。分叉聚合平行演算法可以用普通串行程式的簡單擴充語法來簡潔地表達，它們在實務上可以有效地實作。

具有多個處理單元的平行計算機隨處可見。掌上型電腦、筆電、桌機和雲端機器都是**多核心計算機**，簡稱**多核處理器**（***multicore***），裡面有多個處理「核心」。每一個處理核心都是一個功能齊全的處理器，可以直接存取**共享記憶體**的任何位置。多核可以用網路來互連，組成更大的系統，例如叢集（cluster）。這些多核叢集通常有一個**分散式記憶體**，裡面的多核的記憶體不能被另一個多核的處理器直接存取。處理器必須明確地透過叢集網路，向遠程多核處理器裡的一個處理器發送一個訊息，以請求它所需要的任何資料。最強大的叢集是超級計算機，由成千上萬個多核處理器組成。但是，由於共享記憶體程式往往比分散式記憶體程式更容易設計，而且多核心機器已經被廣泛使用，所以本章要重點討論多核的平行演算法。

我們使用**執行緒平行化**來設計多核程式。這種以處理器為中心的平行設計模式採用「虛擬處理器」的軟體抽象，或是使用一塊相同記憶體的**執行緒**。每一個執行緒都保存自己的程式計數器，可以單獨執行程式碼。作業系統會將一個執行緒載入一個處理核心來執行，當另一個執行緒需要執行時，再將它換掉。

不幸的是，為共享記憶體的平行計算機撰寫執行緒通常很困難，而且容易出錯。其中一個原因是，在執行緒之間動態地分配工作，讓每個執行緒都承接大致相同的工作量並不容易。除非是最簡單的應用程式，否則程式設計師必須使用複雜的溝通協定來設計調度器，以平衡工作負擔。

## 任務平行程式設計

麻煩的執行緒使得**任務平行平台**應運而生，這種平台在執行緒之上提供一層軟體，來協調、安排和管理多核心處理器。有一些任務平行平台被做成執行期（runtime）程式庫，有些則提供具備編譯器和 runtime 支援的成熟平行語言。

**任務平行規劃**可讓你用「無視處理器」的方式來指定平行機制，程式設計師只需要指明哪些計算任務可以平行執行即可，不必指定該用哪個執行緒或處理器來執行該任務。因此，程式設計師不必煩惱通訊協定、負載平衡和其他五花八門的執行緒設計問題。任務平行平台包含調度器，它可以在各處理器之間自動平衡任務的負載，進而大大簡化程式設計師的工作。**任務平行演算法**為普通的串行演算法提供自然的擴展功能，可讓我們用「工作 / 跨度分析」和數學來推斷性能。

## 分叉聚合平行化

儘管任務平行環境的功能還在不斷發展和擴增，但幾乎所有環境都支援**分叉聚合平行化**，這種機制通常以兩種語言功能來實現：**生產**（***spawning***）和**平行迴圈**（***parallel loop***）。spawn 可以將子程序「分叉」，其執行方式類似子程序呼叫（subroutine call），但呼叫方可以在 spawn 出來的子程序還在計算結果時繼續執行。平行迴圈很像一般的 **for** 迴圈，但是可以同時執行多個迴圈迭代。

**分叉聚合**平行演算法使用 spawn 和 parallel 迴圈來描述平行化。這種平行模式有一個關鍵層面繼承自任務平行模式，但是與執行緒模式不同，那就是程式設計師不需要指定哪些任務**必須**平行執行，只需要指定哪些任務**可以**平行執行。底層的 runtime 系統會使用執行緒來平均分配處理器的工作量。本章將探討以分叉聚合模式來定義的平行演算法，以及底層的 runtime 系統如何高效地安排任務平行計算（其中包括分叉聚合計算）。

分叉聚合平行化提供了幾個重要的好處：

- 分叉聚合模式是本書大部分的內容使用的串行設計模式的簡單擴展版本。本書的虛擬碼只需要增加三個關鍵字，就可以定義分叉聚合平行演算法：**parallel**、**spawn** 和 **sync**。在平行虛擬碼中將這些平行關鍵字刪除就會變成同一個問題的普通串行版本，我們稱之為平行演算法的「串行投影（serial projection）」。
- 底層的任務平行模式提供一種理論上的簡潔方式，可根據「工作」和「跨度」概念來量化平行性。

- spawn 可以用自然的方式將許多分治演算法平行化。此外，正如串行的分治演算法適合使用遞迴式來分析，分叉聚合模式中的平行演算法也是如此。
- 分叉聚合設計模式忠實地反映了多核編程在實踐中的演變。有越來越多的多核環境支援分叉聚合設計的各種變體，包括 Cilk [290, 291, 383, 396]、Habanero-Java [466]、Java Fork-Join Framework [279]、OpenMP [81]、Task Parallel Library [289]、Threading Building Blocks [376] 和 X10 [82]。

第 26.1 節將介紹平行虛擬碼，展示如何將任務平行計算的執行過程畫成有向無迴路圖，並介紹工作、跨度和平行性的測量方法，讓你可以用它們來分析平行演算法。第 26.2 節將研究如何平行地計算矩陣乘法，第 26.3 節將處理一種比較麻煩的問題：設計高效的平行合併排序。

## 26.1 分叉聚合平行化基礎

接下來要開始討論平行設計，首先以平行地遞迴計算斐波那契數為例。我們先看一個簡單的串行斐波那契計算方法，雖然它的效率不高，但可以說明如何用虛擬碼來敘述平行化。

第 63 頁的式 (3.31) 如此定義斐波那契數：

$$F_i = \begin{cases} 0 & \text{若 } i = 0 \\ 1 & \text{若 } i = 1 \\ F_{i-1} + F_{i-2} & \text{若 } i \geq 2 \end{cases}$$

你可以用下面的程序 FIB 中的普通串行演算法來遞迴地計算第 $n$ 個斐波那契數。你應該不會用這種方法來實際計算大的斐波那契數，因為這種方法會做沒必要的重複工作，但將它平行化很有教育意義。

```
FIB(n)
1  if n ≤ 1
2       return n
3  else x = FIB(n − 1)
4       y = FIB(n − 2)
5       return x + y
```

我們來分析這個演算法，設 $T(n)$ 為 FIB($n$) 的執行時間。因為 FIB($n$) 有兩個遞迴呼叫與一個固定的額外工作，我們得到遞迴式

$$T(n) = T(n-1) + T(n-2) + \Theta(1)$$

我們可以用代入法來導出這個遞迴式的解為 $T(n) = \Theta(F_n)$（見第 4.3 節）。為了證明 $T(n) = O(F_n)$，我們採用歸納假設 $T(n) \le aF_n - b$，其中 $a > 1$ 且 $b > 0$ 是常數。代入可得

$$\begin{aligned} T(n) &\le (aF_{n-1} - b) + (aF_{n-2} - b) + \Theta(1) \\ &= a(F_{n-1} + F_{n-2}) - 2b + \Theta(1) \\ &\le aF_n - b \end{aligned}$$

若選擇夠大的 $b$，使得 $\Theta(1)$ 的上限常數可忽略不計。我們可以再選擇一個夠大的 $a$，以決定小 $n$ 時，$\Theta(1)$ 基本情況的上限。我們使用歸納假設 $T(n) \ge aF_n - b$ 來證明 $T(n) = \Omega(F_n)$。我們進行代入，並按照與漸進上限證明類似的推理，選擇比 $\Theta(1)$ 項中的下限常數更小的 $b$，以及足夠小的 $a$，以確保在小的 $n$ 值下，滿足 $\Theta(1)$ 基本情況的下限，以建立這個假設。從第 50 頁的定理 3.1 可得到 $T(n) = \Theta(F_n)$。因為 $F_n = \Theta(\phi^n)$，其中 $\phi = (1 + \sqrt{5})/2$ 是黃金比例，根據第 64 頁的式 (3.34) 可得

$$T(n) = \Theta(\phi^n) \tag{26.1}$$

用這個程序來計算斐波那契數特別緩慢，因為它的執行時間成指數級增長（第 921 頁的挑戰 31-3 有更快速的方法）。

我們來看看為何這個演算法很低效。圖 26.1 是用 FIB 程序來計算 $F_6$ 時建立的遞迴程序實例樹。呼叫 FIB(6) 會遞迴地呼叫 FIB(5) 然後 FIB(4)。然而，呼叫 FIB(5) 也會導致呼叫 FIB(4)。兩次呼叫 FIB(4) 皆回傳相同的結果（$F_4 = 3$）。因為 FIB 程序不進行記憶（第 354 頁有「記憶化」的定義），所以第二次呼叫 FIB(4) 會重複做第一次呼叫所做的工作，浪費資源。

雖然 FIB 很不適合用來計算斐波那契數，但它可以協助我們初步理解並熟悉平行化概念。或許最基本的概念是，如果兩個平行的任務處理完全不同的資料，那麼在沒有其他干擾的情況下，同時執行它們的結果與串行執行它們的結果是相同的。比如說，在 FIB($n$) 中，遞迴呼叫第 3 行的 FIB($n-1$) 兩次和遞迴呼叫第 4 行的 FIB($n-2$) 可以安全地平行執行，因為其中一個的計算不會影響另一個。

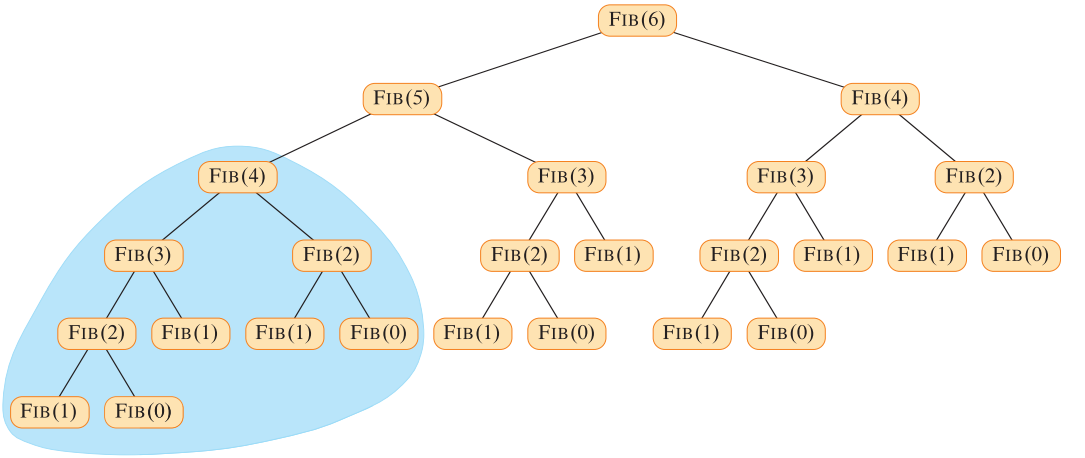


**圖 26.1** FIB(6) 的呼叫樹。樹中的每個節點代表一個程序實例，它們的子節點是它們在執行過程中呼叫的程序實例。因為每一個使用相同引數的 FIB 實例都做相同的工作來產生相同的結果，用這種演算法來計算斐波那契數的效率之低下，可以從大量重複呼叫函式來計算同一個東西看出。圖 26.2 是樹中陰影部分的任務平行形式。

## 平行關鍵字

下一頁的 P-FIB 程序可計算斐波那契數，但使用**平行關鍵字** **spawn** 與 **sync** 來代表虛擬碼中的平行化。

將 P-FIB 中的 **spawn** 與 **sync** 關鍵字刪除後產生的虛擬碼與 FIB 一模一樣（除了程序開頭的名稱與兩個遞迴呼叫中的名稱之外）。將平行演算法的平行指令移除得到的串行演算法稱為平行演算法的**串行投影** [1]，這個例子可以藉著省略關鍵字 **spawn** 與 **sync** 得到。遇到 **parallel for** 迴圈時（等一下會介紹），我們省略關鍵字 **parallel**。事實上，我們的平行虛擬碼有一個優雅的特性：它的串行投影一定是解決相同問題的一般串行虛擬碼。

1 在數學中，投影是個等價函數，它是一個函數 $f$，滿足 $f \circ f = f$。在這個例子裡，函數 $f$ 將分叉聚合程式集合 $\mathcal{P}$ 對映至串行程式集合 $\mathcal{P}_S \subset \mathcal{P}$，串行程式是沒有平行化的分叉聚合程式。對於分叉聚合程式 $x \in \mathcal{P}$，因為 $f(f(x)) = f(x)$，所以正如我們所定義的，串行投影確實是一種數學投影。

```
P-FIB(n)
1  if n ≤ 1
2      return n
3  else x = spawn P-FIB(n − 1)      // 不等待子程序 return
4       y = P-FIB(n − 2)            // 與生產出來的子程序平行執行
5       sync                        // 等待生產出來的子程序完成
6       return x + y
```

## 平行關鍵字的語義

當一個程序呼叫式的前面有關鍵字 **spawn** 時，就會發生**生產**（***spawning***）。spawn 的語義和普通的程序呼叫不同，執行 spawn 的程序實例（**父程序**）可以繼續和 spawn 出來的子程序（其**子程序**）平行執行，而不是像串行執行那樣，等待子程序完成。在這個例子中，當 spawn 出來的子程序正在計算 P-FIB($n-1$) 時，父程序可繼續計算第 4 行的 P-FIB($n-2$)，與 spawn 出來的子程序平行執行。因為 P-FIB 程序是遞迴的，所以這兩個子程序的呼叫本身就產生了嵌套的平行化，它們的子程序也是如此，可能創造很龐大的子計算樹，全部都以平行方式執行。

然而，關鍵字 **spawn** 並未規定程序**必須**和它的子程序平行執行，僅定義它**可以**。平行化的關鍵字表達的是計算的**邏輯平行性**，指出計算的哪些部分可以平行進行。在執行期，究竟哪些子計算可以平行執行是由**調度器**（***scheduler***）決定的，隨著計算的展開，調度器會將它們分配給有空的處理器。等一下會討論任務平行調度器背後的理論（在第 732 頁）。

程序必須執行 **sync** 敘述句才能安全地使用它 spawn 的子程序所回傳的值，像第 5 行。關鍵字 **sync** 代表程序在必要時，必須等待它 spawn 的所有子程序完成工作，才能繼續執行在 **sync** 後面的敘述句，也就是分叉聚合計算中的「聚合」階段。P-FIB 程序必須在第 6 行的 **return** 之前使用 **sync**，以免在 P-FIB($n-1$) 結束之前將 $x$ 和 $y$ 相加，並將其回傳值指派給 $x$，因而發生異常。除了用 **sync** 敘述句來明確提供的聚合同步之外，為了方便起見，我們假設每個過程在 return 之前，都會私下執行一次 **sync**，以確保所有子程序皆在其父程序結束之前完成。

## 平行執行的圖模型

我們可以將平行計算（在平行程式的指示下，處理器所執行的動態執行期指令串流）的執行過程視為有向無迴路圖 $G = (V, E)$，我們稱之為（**平行**）**追跡**（***trace***）[2]。從概念上講，在 $V$ 中的頂點是已執行的指令，在 $E$ 中的邊代表指令之間的依賴關係，其中 $(u, v) \in E$ 代表平行程式要求指令 $u$ 必須在指令 $v$ 之前執行。

一個追跡的頂點僅代表一個執行的指令，有時候這不方便，特別是當我們想要關注計算的平行結構時，因此，只要一連串的指令裡沒有平行或程序控制指令（沒有 **spawn**、**sync**、程序呼叫或 **return**，無論是透過明確的 **return** 敘述句，還是在程序結束時自動 return），我們就將整串指令組成一**串**。例如，圖 26.2 是圖 26.1 的陰影部分計算 P-FIB(4) 產生的追跡。指令串裡面沒有平行或程序控制的指令，在追跡中，這些控制型依賴關係一定用邊來表示。

當父程序呼叫子程序時，追跡有一條邊 $(u, v)$ 從父程序中發出呼叫的 $u$ 串連接到 spawn 出來的子程序的第一串 $v$，例如在圖 26.2 中，有一條邊從 P-FIB(4) 的橘串連接到 P-FIB(2) 的藍串。當子程序的最後一串 $v'$ 返回時，它在追跡裡有一條接到 $u'$ 串的邊 $(v', u')$，其中，$u'$ 是父程序的 $u$ 的下一串，例如從 P-FIB(2) 的白串連接到 P-FIB(4) 的白串的邊。

但是，當父程序 spawn 子程序時，追跡有些不同。與發出呼叫時一樣，此時有一條邊 $(u, v)$ 從父程序接到子程序，例如 P-FIB(4) 的藍串連接到 P-FIB(3) 的藍串，但追跡還有另一條邊 $(u, u')$，這條邊代表 $u$ 的下一串 $u'$ 在 $v$ 執行時可以繼續執行，例如從 P-FIB(4) 的藍串連接到 P-FIB(4) 的橘串的邊。與呼叫時一樣，子程序的最後一串 $v'$ 會有一條出邊，但因為這是 spawn，所以那條邊不是接到 $u$ 的後繼者，而是 $(v', x)$，$x$ 是在父程序裡緊接在 **sync** 之後的指令串，該串會確保子程序已經完成，例如從 P-FIB(3) 的白串連接到 P-FIB(4) 的白串的邊。

你可以判斷特定的追跡是由哪些平行控制指令建立的。如果一個指令串有個後繼者，其中一個一定是 spawn 出來的；如果一個指令串有多個前驅者，那些前驅者一定是用 **sync** 敘述句來聚合的。因此，在一般情況下，$V$ 是指令串集合，而有向邊集合 $E$ 代表平行化和程序控制所造成的指令串之間的依賴關係。如果在 $G$ 中，從串 $u$ 到串 $v$ 有一條有向路徑，我們說這兩串是（**邏輯上**）**串行的**。如果在 $G$ 中，沒有路徑從 $u$ 到 $v$ 或從 $v$ 到 $u$，那麼這兩串是（**邏輯上**）**平行的**。

2 在文獻中也稱為 ***computation dag***。

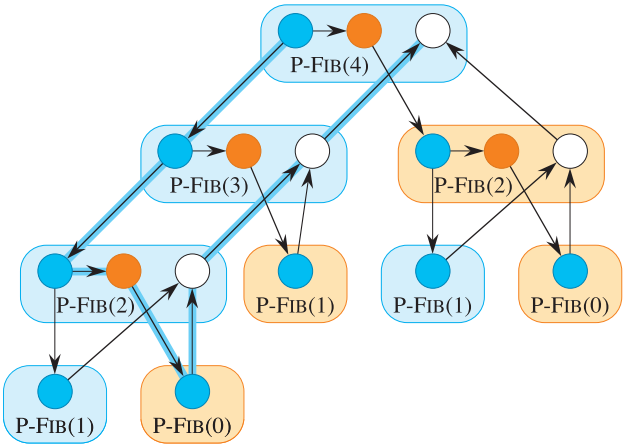


**圖 26.2** 與圖 26.1 對應的 P-FIB(4) 追跡。圖中的每個圓都代表一串指令，藍圓代表程序實例在第 3 行 spawn P-FIB($n-1$) 之前執行的任何指令；橘圓代表程序中，從第 4 行呼叫 P-FIB($n-2$) 到第 5 行的 **sync** 所執行的指令，程序會在第 5 行暫停，直到 P-FIB($n-1$) 的 spawn 返回為止；白圓代表程序在 **sync** 之後（將 $x$ 與 $y$ 相加）一直到回傳結果時執行的指令。屬於同一個程序的指令串都放在圓角矩形內，藍矩形代表 spawn 出來的程序，棕矩形代表被呼叫的程序。假設每一串都花費單位時間，此圖的工作量是 17 單位時間，因為有 17 串，且跨度是 8 單位時間，因為關鍵路徑（藍邊）包含 8 串。

在繪製分叉聚合平行追跡時，我們可以將以指令串構成的 dag 放入程序實例**呼叫樹**中。例如，圖 26.1 是 FIB(6) 的呼叫樹，它也可以當成 P-FIB(6) 的呼叫樹，在程序實例之間的邊代表呼叫或 spawn。圖 26.2 將鏡頭拉近到陰影部分的子樹，展示 P-FIB(4) 的各個程序實例中的指令串。所有連接指令串的有向邊都在程序裡執行，或是沿著圖 26.1 的呼叫樹裡的無向邊執行（較一般的任務平行追跡（不是分叉結合追跡）可能包含一些不沿著無向邊執行的有向邊）。

我們的分析通常假設平行演算法在**理想的平行計算機**上執行，計算機由一組處理器和一個**順序一致的**共享記憶體組成。記憶體是用**載入指令**和**儲存指令**來存取的，前者將資料從記憶體中的某個位置複製到處理器的某個暫存器，後者將資料從處理器的暫存器複製到記憶體的某個位置。一行虛擬碼可能包含幾條這種指令。例如，$x = y + z$ 這一行可能會產生載入指令，將 $y$ 和 $z$ 分別從記憶體抓到處理器中，以及一條加法指令，在處理器中將它們相加，以及一條儲存指令，將結果 $x$ 放回記憶體中。一台平行計算機可能同時使用幾個處理器來進行載入或儲存。順序一致性意味著，即使有多個處理器試圖同時存取記憶體，共享記憶體的行為仍然像每次只執行一個處理器的一條指令一樣，即使實際上可能同時傳輸資料。這就好像指令按照一種總體線性順序被依次執行，且該順序保留了各個處理器執行自己的指令的個別順序。