前言

> 我這個世代的人經常在成長過程中聽到這句話：“撒旦有一些壞事是專門給懶人做的”。天真的我相信它，所以秉持良心努力工作，直到現在。雖然良心約束了我的行為，但我的想法依然經歷一場革命。我認為，人們的工作量太大了，之所以會有這麼痛苦的結果，是因為人們相信工作是良性的，且現代化的工業國家不斷鼓吹這個不良的觀念。
>
> Bertrand Russell, *In Praise of Idleness*（1932）

本書談的是**演算法**（*algorithms*），也就是為了**不做事**而做的事。它是為了避免工作而做的事情。我們善於利用大腦來發明取代勞力的工具，藉由演算法，我們可用大腦來創造大腦。

減少人類的工作是很崇高的任務。我們認為應該盡量使用機器來減少勞動，以減少幾個世紀以來辛勞的工作。更棒的是，我們也可以同時避免體力勞動與腦力勞動。我們應該盡力避免扼殺人類創造力的重複性勞力，演算法可協助我們做到這一點。

此外，現今的數位科技可以完成人性化的工作，而非只有單調的工作。機器可以辨識及產生語音、翻譯文章、分類及歸納文件、預測天氣，以驚人的準確性來預測模式、運行其他機器、算術、在遊戲中擊敗人類，以及幫助我們發明其他機器。這些工作都是用演算法來做的，而且我們可以藉此減少工作量，把時間花在真正想做的事情上，以及獲得更多時間與機會來發明更好的演算法，進一步減少更多勞動。

演算法不是在電腦問世之後才出現的，也不限於電腦科學，它從遠古時代就與我們同在。演算法已經影響絕大多數的學科了。很多人發現演算法已經變成他們的專業領域的主要成分，於是開始學習演算法，試著理解與使用它們，無論它們表面上看起來與電腦的距離有多遙遠。

就算是簡單的事情與日常工作，因為沒有正確思考而浪費的精力也會很驚人。當你做重複的事情時，通常根本不該做那件事。作者的經驗是，在日常工作中，只要我們知道如何避免工作，通常很快就可以完成一系列的操作；我指的不是推卸責任（有些人非常擅長），而是讓電腦為我們工作（我們應該讓更多人擅長這件事）。

本書對象

本書是為第一次學習演算法的人寫的。如果你的本科是電腦科學，你可以從本書學到初階的方法，之後再研究較深的主題；演算法是計算的核心，本書介紹的只是它的皮毛而已。

但是，很多人也會在就業或學習其他學科時，發現演算法是一種重要的工具。許多學科都不太可能完全用不到演算法。本書希望將演算法介紹給工作或求學的過程中需要使用與瞭解演算法的人，無論演算法是不是他們的核心知識。

本書的對象還有希望可以使用演算法（無論多麼小型及簡單）來簡化工作，並避免浪費時間在雜事上的人。你只要寫幾行現代腳本程式就可以執行需要消耗好幾個小時的人力的工作。運用演算法不是內行精英的特權，令人遺憾的是，外行人有時無法瞭解這一點。

要在現代社會中進行有意義的交流，基本的數學與科學知識是不可或缺的元素，同樣的，如果你不瞭解演算法，就不可能在現代社會中具備生產力。它們都是你日常經驗的基礎。

你需要知道的知識

演算法並非只有電腦科學家才能瞭解。演算法是以執行工作的指令組成的，任何人都可以瞭解它。但是，要有效地運用演算法，以及從書中獲得最大的利益，就像這一本書，讀者應該具備一些基本的技術。

你不需要精通數學，但應該瞭解一些基本的數學概念與符號。本書使用的數學不會超過高中等級。你不需要知道更高深的數學，但必須知道如何證明，因為我們會以邏輯步驟來證明演算法的可行性，這與數

學的證明是相同的。我指的不是這本書將會充滿數學證明，而是你必須瞭解我們如何證明。

讀者不需要很會寫程式，但應該**初步**瞭解電腦的運作方式、該怎麼寫程式，以及程式語言的結構。讀者不需要深入瞭解它們；不過，要看懂這本書，你最好也能夠瞭解它們。電腦系統與演算法有密切的關係，它們的理論是互通的。

你應該具備好奇心。演算法談的是解決問題，並確保答案是**有效的**。每次你在想“有沒有更好的做法？”時，其實是在尋找一種演算法。

寫作風格

本書希望以最簡單的方式來說明演算法，但又不至於侮辱讀者的智慧。如果你不明白書中的內容、如果你開始認為這本書或許比較適合程度比你高的人、如果你開始敬畏書的內容，但無法瞭解它，代表這本書讓你感到挫折。我想要盡量避免這種情況，所以會稍微簡化一些東西。也就是說，我們不會充分證明某些內容。

簡化或省略一些內容，不代表讀者不需要積極求知：這就是我們不侮辱讀者智慧之處。我們假設讀者真心想要學習演算法，這確實需要許多時間與精力，而且，投入的時間越多越好。

有些書會讓你深入其中，讓你不知不覺看完整本書，並充分吸收它的內容。我指的不是廉價小說。Albert Camus 寫的《*The Plague*》不是一本難懂的書，但大家都認為它是一本既正式且深入的文學作品。

但有些書會讓人絞盡腦汁，覺得高不可攀，努力看完的人有一種突破自我的感覺，甚至覺得自己與眾不同—並非所有人都喜歡 James Joyce 的《*Ulysses*》、Thomas Pynchon 或 David Foster Wallace 的作品。不過努力看完他們的書之後，覺得後悔的人也不多。

在這兩種書之間，還有許多其他種類的書。或許你覺得《*Gravity's Rainbow*》有點艱深，那你覺得《*The Brothers Karamazov*》或《*Anna Karenina*》如何？

你正在看的這本書試著介於兩者之間；我們不要求你有過人的智慧，但是必須認真研究。作者想要牽著你的手介紹演算法，而不是把你扛在肩上；這本書會協助你瞭解演算法，不過你應該自己走這條路。所以它不會侮辱你的智慧，它會假設你很聰明，樂於學習新知，知道學習是需要積極努力的；你知道天下沒有不勞而獲的事情，而努力終會獲得回報。

虛擬碼

在幾年前，只知道一種程式語言的年輕人就可以進入電腦業界，但現在已經不是如此了。目前有許多很好的程式語言，電腦做的事情比二十年前還要多很多，且各種不同的語言都有各自專精的領域。讓語言彼此競爭是愚蠢且適得其反的。此外，因為電腦可帶來美好的事物，所以人們願意積極尋求新的方法來與電腦合作，發明新的程式語言，並進化舊的語言。

作者確實有偏愛的語言，但將這種偏好強加在讀者身上不太公平。何況電腦語言也有流行期，昨日的寵兒或許會成為今日的敝屣。為了讓這本書的用途更廣泛且長壽，我們不採用實際的語言來編寫範例，而是用虛擬碼來說明演算法。虛擬碼比實際的電腦程式更容易瞭解，因為它沒有真正的程式語言固有的缺點。虛擬碼通常比較容易理解，當你想要深入瞭解演算法時，就必須用手書寫，此時虛擬碼比實際的程式好用，因為使用程式還需要注意語法。

話雖如此，你必須實際編寫電腦程式來實作演算法，否則很難瞭解它們。本書採用虛擬碼，但不希望讀者以傲慢的態度來看待電腦程式碼。可行的話，你應該選擇一種語言來實作書中的演算法。使用電腦程式**正確地**實作演算法，會讓你獲得超乎想像的成就感。

如何閱讀這本書？

本書最佳的閱讀方式是循序漸進，因為前面的章節介紹的是之後會用到的概念。在一開始，你會看到所有演算法都會用到的基本資料結構，之後的章節也會用到它們。但是，奠定基礎之後，如果你有比較想看的章節，也可以選擇想看的部分。

因此，你應該從第 1 章看起，在這一章，你也會看到其餘章節的架構：先描述問題，再展示解決問題的演算法。第 1 章也會介紹本書的虛擬碼規範，以及基本的術語，與我們遇到的第一種資料結構：陣列與堆疊。

第 2 章會初次介紹圖（graph）與探索它們的方式，也會討論遞迴，所以雖然你看過圖，但還不確定是否已瞭解遞迴，就不該跳過它。第 2 章也會展示其他的資料結構，其他章節的演算法會不斷使用它們。接下來，在第 3 章，我們會討論壓縮的問題，與兩種壓縮法的運作方式：我們也會介紹一些更重要的資料結構。

第 4 章與第 5 章討論密碼學。它與圖和壓縮不同，但也是很重要的演算法應用，尤其是近年來，你可以在許多地方與設備上找到個人資料，但各路人馬都想要窺探它們。這兩章可以單獨閱讀，不過有一些重要的部分，例如如何找出大質數，會留到第 16 章說明。

第 6 到 10 章會說明與圖有關的問題：排序工作、走出迷宮、找出彼此聯繫的事物的重要性（例如網路上的網頁）、如何使用圖來選舉。走迷宮有一些應用可能是你想像不到的，包括排版文章段落、Internet 路由與金融套利；它有一種版本是在選舉的情境下使用的，所以你可以將第 7、8 與 10 章視為同一個部分。

第 11 與 12 章會處理兩種最基本的計算問題：搜尋與排序。這兩個主題可用整本書來說明，它們也的確有專屬的書籍。我們只展示一些常用的重要演算法。在討論搜尋時，我們也會討論其他的主題，例如線上搜尋（在你收到的串流中尋找東西，且事後不能更改決定）以及研究員經常看到的無尺度分布。第 13 章會介紹另一種儲存與取回資料的方法，它們相當實用、通用且優雅。

第 14 章會討論分類演算法，這種演算法可根據一組範例來學習分類資料，接著我們可以用它來分類新的、未看過的實例。這是一種機器學習案例，這個領域隨著電腦越來越強大而變得越來越重要。這一章也會討論資訊理論的基本概念，這是另一種與演算法有關的優美領域。第 14 章與本書的其他章節不同，裡面的演算法會呼叫較小型的演算法來完成部分的工作，如同許多小型的組件組成電腦程式，每一個組件都會做一項特定的工作一般。這一章也會說明本書其他地方談過的資料結構如何在分類演算法中扮演重要的角色。想要瞭解高階演

算法細節的讀者會特別喜歡這一章—本章將會介紹將演算法轉換為程式的步驟。

第 15 章會討論符號序列，也就是字串，以及從裡面找出東西的方法。每當我們要求電腦在文章中找出某個東西時，就會執行這種操作，但我們不知道如何有效率地執行它。幸運的是，我們可以快速且優雅地完成它。此外，符號序列也可以用來代表許多其他的東西，所以許多領域都會運用字串比對，例如生物學。

最後，第 16 章將會介紹隨機。隨機化的演算法有大量的應用，這一章只納入其中的一些。它們也可以解決書中談過的其他問題，例如在密碼學中，找到大質數的問題，或是同樣與投票有關的，計算你投下的票造成的影響。

課程使用

本書的教材適合想要探討演算法，而且把重心放在瞭解主要的概念，而不是深入討論技術做法的完整學期課程。商學、經濟學、生活、社會與應用科學等學科，或數學與統計學等形式科學的學生，都可在入門課程中將它當成主要教科書，並且輔以程式設計作業，實作可實際應用的演算法。電腦科學系可以將它當成非正式的介紹，協助你欣賞較具技術性的書籍之中的演算法的深度與美感。

致謝

當我第一次向麻省理工學院出版社提出這本書的構想時，一點都不知道該如何實現它，如果沒有一群優秀的人協助我，這本書就不可能出現。Marie Lufkin Lee 在整個過程中指引我，即使在最後期限之前，他依然採取溫和的態度。Virginia Crossman、Jim Mitchell、Kate Hensley、Nancy Wolfe Kotary、Susan Clark、Janice Miller、Marc Lowenthal 與 Justin Kehoe 曾經在各個階段像匿名評論家一樣協助我。Amy Hendrickson 在我享受 LATEX 奧秘的樂趣時協助過我。

Marios Fragkoulis 針對手稿的部分內容提供詳細的意見，Diomidis Spinellis 特別撥冗建議我該如何改進。Stephanos Androutsellis-Theotokis、George Theodorou、Stephanos Chaliasos、Christina Chaniotaki 與 George Pantelis 都很親切地指出錯誤。如果本書還有任何錯誤或疏漏，都與他們無關。

當然我要對 Eleni、Adrian 與 Hector 表達我的敬意，他們一起經歷了本書最艱困的時刻。

最後的前言

如果你將演算法（algorithm）寫成 *algorhythm*，它其實是一個混合詞，代表“痛苦的節奏（the rhythm of pain）”，*algos* 是痛苦的希臘文。其實，algorithm 這個字來自 al-Khwārizmī，它是一位波斯數學家、天文學家與地理學家的名字（c. 780–c. 850 CE）。希望這本書能夠吸引你，而非讓你感到痛苦。我們開始來討論演算法吧！

1 股價跨幅

假設你有一檔股票每日的股價，也就是說，你有一系列的數字，這些數字都代表該股票在特定日期的收盤價，日期是按照時間順序來排列的，股市休市的那一天就沒有報價。

股票的**跨幅**（*span*）代表在指定的日期之前，價格少於或等於該日股價的連續天數。所以股價跨幅（Stock Span）問題就是，在一系列的逐日股價中，找出每天的股票跨幅。例如，在圖 1.1 中，第一天是零日（day zero），在資料的六日，跨幅是五天，五日是四天，四日是一天。

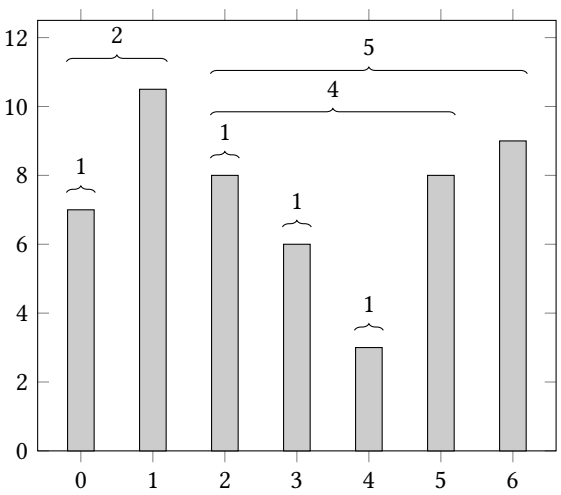

圖 1.1
股價跨幅範例。

在實際的情況下，這系列的數字可能會有上千日，我們也有可能想要計算許多不同系列的跨幅，以每一個系列來描述各種股價的演變。因此，我們想要用電腦來算出解答。

可用電腦來處理的問題通常都有多種方式可得到答案，裡面通常有一些方法比較好。“比較好”本身沒有特定的意思，它只是某方面會比較好，可能指的是速度、記憶體，或某種會影響時間、空間等資源的東西。我們很快就會進一步討論這一點，不過你要記住它，因為有的方法很簡單，但是以我們的限制或標準來看，它可能不是最好的。

假設你在這一系列日期的 m 日。找出 m 日的跨幅的方式之一是回推一天，所以你會在 $m-1$ 日。如果 $m-1$ 日的價格大於 m 日的價格，你就知道 m 日的股價跨幅只有 1。但是如果 $m-1$ 日的價格少於或等於 m 日的價格，那麼該股票在 m 日的價格跨幅就至少是 2，且可能更多，取決於之前的價格。所以我們繼續查看 $m-2$ 日的價格。如果這一天的價格不大於 m 日的價格，我們再繼續查看前一天，以此類推。接下來會發生兩件事。我們可能會看完剩下的日期（也就是到達這一系列的起點），那麼 m 日之前的所有股價都會小於或等於 m 日，所以跨幅剛好是 m。或者，我們發現在 k 日時（$k < m$）股價比 m 日高，則跨幅是 $m-k$。

如果股價序列有 n 日，為了解決問題，你就必須重複剛才談到的程序 n 次，一日的跨幅一次。你可以用圖 1.1 的範例來驗證這個程序是否有效。

不過白話不適合用來描述程序，白話很適合說明世上幾乎所有事項，但不包括要傳送給電腦的程序，因為我們必須準確地描述想要送給電腦的事項。

如果準確程度足以讓電腦瞭解我們的程序，我們就寫出一段**程式**了。人類不容易瞭解電腦程序，因為你必須告訴電腦完成工作需要的任何細節，但它們其實與問題的解法沒有實際的關係。足以讓電腦瞭解的描述對人類來說可能會過於詳細，以致難以理解。

所以我們可以採取折衷方案，使用比一般文字準確且人類比較容易理解的結構化語言來描述程序。電腦無法直接執行這種結構化語言，但它很容易就可以轉換成實際的電腦程式。

1.1 演算法

在處理股價跨幅問題之前，我們要先來認識一些重要的術語。**演算法**是一種程序，不過是一種特殊的程序。它必須用一系列有限的步驟來描述，且必須在有限的時間終止。每一個步驟都必須有良好的定義，讓人類可以用紙筆來執行。演算化會使用我們提供的輸入來辦事，並產生一些結果來反應它處理好的工作。演算法 1.1 實現了上述的程序。

演算法 1.1：簡單的股價跨幅演算法。

```
SimpleStockSpan(quotes) → spans
    輸入：quotes，有 n 筆股價的陣列
    輸出：spans，有 n 筆股價跨幅的陣列
 1  spans ← CreateArray(n)
 2  for i ← 0 to n do
 3      k ← 1
 4      span_end ← FALSE
 5      while i − k ≥ 0 and not span_end do
 6          if quotes[i − k] ≤ quotes[i] then
 7              k ← k + 1
 8          else
 9              span_end ← TRUE
10      spans[i] ← k
11  return spans
```

你可以從演算法 1.1 看到我們描述演算法的方式。使用電腦語言時，我們必須處理與演算法邏輯無關的細節，所以我們改用**虛擬碼**（*pseudocode*）的形式。虛擬碼是一種介於實際的程式碼和非正式敘述之間的東西，它會使用一種結構化的格式，以及一組有特定意義的單字。但是，虛擬碼不是真正的電腦程式碼，它不是要讓電腦執行的，而是要讓人類理解的。順道一提，程式也應該讓人類可以理解，但並不是所有的程式都是如此，世上也有一些運行中的、寫得很差的、令人無法理解的電腦程式。

每一個演算法都有名稱、可接收一些輸入，與產生一些輸出。我們用`CamelCase`（駝峰式大小寫）來表示演算法的名稱，並將它的輸入放在括號內，用→來代表輸出。下一行說明演算法的輸入與輸出。我們可用演算法的名稱與後面的括號內的輸入來**呼叫**演算法。當我們寫好演算法後，可以將它視為黑盒子，傳送一些輸入給它；之後，這個黑盒子會回傳演算法的輸出。當你用程式語言來實作演算法時，它是一段有名稱的程式碼，也就是**函式**（*function*）。在電腦程式中，我們會呼叫實作了演算法的函式。

有些演算法不會明確地回傳輸出，而是會做一些影響環境的行為。例如，我們可能會提供一些儲存空間，來讓演算法寫入它的結果，此時，演算法不會以傳統的方式回傳它的輸出，不過仍然有輸出，也就是影響環境而造成的改變。有些程式語言會將以下兩者視為不同的東西：有名稱而且會明確回傳一些東西的程式稱為**函式**（*functions*）；有名稱但不會回傳東西，不過有其他副作用的程式稱為**程序**（*procedures*）。這個區別來自數學，數學中的函數（也就是function）是必須回傳一個值的東西。對我們來說，演算法被寫成實際的程式時，不是變成函式就是變成程序。

我們的虛擬碼會使用一組以**粗體**表示的關鍵字，如果你瞭解電腦與程式語言的工作方式，從字面應該就可以瞭解它們的意思。我們會使用←字元來代表指派，等號（=）代表相等，五種符號來表示四則運算（+、−、/、×、·），乘法有兩種符號，我們會依美觀的標準來選擇其中一種。我們會使用縮排來代表虛擬碼區塊，而非使用關鍵字或符號。

這個演算法使用**陣列**。陣列是一種保存資料的結構，可讓我們用某些方式來操作它的資料。可以保存資料，並且讓你對這些資料做特定操作的結構稱為**資料結構**。因此陣列是一種資料結構。

陣列之於電腦，就像一系列的物件之於人類。它們是有序的元素序列，元素會被存放在電腦的記憶體內。為了取得空間來保存元素，以及建立可保存 n 個元素的陣列，我們在演算法 1.1 的第 1 行呼叫 `CreateArray` 演算法。如果你熟悉陣列的話，或許會覺得奇怪：陣列為何要用演算法來建立？但事實的確如此。為了取得記憶體區塊來保存資料，你至少必須搜尋電腦中可用的記憶體，並且標記它來讓陣列使用。`CreateArray(`n`)` 呼叫式做的就是上述的所有事情。它會回傳一個可存放 n 個元素的陣列，在一開始，陣列裡面沒有元素，只有可保存元素的空間。演算法必須負責呼叫 `CreateArray(`n`)` 來將實際的資料填入陣列。

就陣列 A 而言，我們以 $A[i]$ 來代表讀取它的第 i 個元素。陣列的元素位置，例如 $A[i]$ 中的 i，稱為它的**索引**（*index*）。有 n 個元素的陣列含有元素 $A[0]$、$A[1]$、…、$A[n-1]$。當你看到它的第一個元素是第零個，最後一個元素是第 $n-1$ 個時可能會覺得很奇怪，你原本可能會認為它們分別是第一個與第 n 個。但是這是多數電腦語言的陣列的工作方式，所以你最好開始適應它。因為它很普遍，所以當我們迭代一個大小為 n 的陣列時，會從第 0 個位置迭代至第 $n-1$ 個位置。如果演算法提到某個東西是從數字 x 到數字 y（假設 x 小於 y），就代表從 x 開始，但不包括 y 的所有值；見演算法的第二行。

我們假設無論 i 是多少，讀取第 i 個元素花的時間都相同。所以讀取 $A[0]$ 需要的時間與 $A[n-1]$ 相同。這是陣列的重要特徵之一：元素可用固定的時間來讀取；當我們用索引來讀取元素時，陣列不需要搜尋它。

關於符號，當我們描述演算法時，會以小寫字母來代表其中的變數，但是如果變數代表的是資料結構，我們會使用大寫來提示它們，例如陣列 A，不過這不是必要的做法。當我們想要用多個單字來構成變數的名稱時，會用底線（`_`）來作為**連接符號**（`a_connector`）；這是必要的做法，因為電腦無法瞭解以空格來分開的多個單字組成的變數名稱。

演算法 1.1 使用了儲存數字的陣列。陣列可以保存任何型態的項目，不過在我們的虛擬碼中，陣列只能保存單一型態的項目，這也是多數程式語言的做法。例如，你或許會有一個十進位數字陣列、一個分數陣列、一個用項目來代表人員的陣列，與一個用項目來代表地址的陣列。但你應該不會有一個同時含有十進位數字與人員項目的陣列。關於“代表人員的項目”究竟是什麼，就要依特定的程式語言而定了。所有程式語言都提供某種手段來表示有意義的東西。

含有字元的陣列是特別實用的陣列種類。字元陣列代表字串，它是一系列的字母、數字、單字、句子，或其他東西。如同所有陣列，陣列內的各個字元都可以用索引來分別引用。如果我們有個字串 *s* = “Hello, World”，則 *s*[0] 是字母“H”，而 *s*[11] 是字母“d”。

總之，陣列是一種資料結構，保存了一系列相同型態的項目。陣列有兩種操作方式：

- `CreateArray(`*n*`)` 會建立一個可以保存 *n* 個元素的陣列。這個陣列並未被初始化，也就是說，它沒有保存任何實際的元素，但是已經保留空間來儲存它們。
- 我們看過，*A*[*i*] 可讀取陣列 *A* 的第 *i* 個元素，且讀取陣列內的任何元素花費的時間都是相同的。用小於 0 的 *i* 來讀取 *A*[*i*] 是錯誤的行為。

回到演算法 1.1。按照上述的講法，這個演算法在第 2 – 10 行有一個迴圈，也就是一段會重複執行的程式。如果我們有 *n* 天的股價，這個迴圈會執行 *n* 次，每次計算一個跨幅。我們正在計算的日子是以變數 *i* 來指定的。最初我們會在第零天，也就是最早的時間點；每當我們經過迴圈的第 2 行時，就會移往第 1、2、…、*n* – 1 天。

我們用**變數**（*variable*）*k* 來代表目前的跨幅的長度；變數就是在虛擬碼中用來代表某塊資料的名稱。準確來說，這些資料的內容，也就是變數的**值**，會隨著演算法的執行而改變，所以稱為變數。當我們開始計算跨幅時，*k* 的值一定是 1，這是在第 3 行設定的。我們也使用一個**指示變數**（*indicator variable*），*span_end*。指示變數可接收 true 與 false 值，來指出某位事成立或不成立。變數 *span_end* 會在我們到達跨幅的終點時變成 true。

在開始計算每一個跨幅時，*span_end* 是 false，見第 4 行。跨幅的長度是在第 5–9 行的內部迴圈中計算的。第 5 行告訴我們，只要跨幅還沒結束，就盡可能地回推時間。盡可能地回推是以條件 $i - k \geq 0$ 來決定的，$i - k$ 是我們往回檢查跨幅是否終止時的日期的索引，這個索引不能為零，因為零相當於第一天。檢查跨幅終止的地方是在第 6 行。如果跨幅沒有終止，我們會在第 7 行遞增它，否則在第 9 行記錄跨幅終止，所以回到第 5 行時，迴圈會終止。每當第 2–10 行的外部迴圈結束迭代時，我們會在第 10 行將 k 值存入陣列 *span* 中適當的地方。我們在第 11 行迴圈結束之後回傳 *spans*，它裡面含有演算法的結果。

注意，當我們開始時，$i = 0$ 且 $k = 1$。這代表第 5 行的條件在一開始必定是失敗的。這是正確的，因為它的跨幅只會等於 1。

之前我們曾經談過關於演算法、筆與紙的事情。要瞭解演算法，最好的方式是你自己手動執行它。如果你覺得演算法很複雜，或不確定是否充分理解它，就用紙筆來操作一些案例，看看它做了些什麼。或許這種做法看起來很老派，但它可以節省許多時間。如果你還不明白演算法 1.1，現在就做這件事，等你明白這個演算法時再回來。

1.2 執行時間與複雜度

演算法 1.1 是股價跨幅問題的解法之一，但還有更好的方法。在這裡，更好代表更快速。當我們談到演算法的速度時，指的其實是演算法的執行步驟的數量。儘管電腦計算每個步驟的速度會愈來愈快，但無論電腦多快，步驟的數量都是相同的，所以使用演算法需要的步驟數量來評估它的效能是合理的方式。我們將步驟數量稱為演算法的**執行時間**（*running time*），儘管這純粹是以數量來衡量的，而不是任何一種時間單位。使用時間單位的話，就會讓執行時間與特定的電腦型號掛勾，這種估算就毫無意義了。

考慮計算 n 筆股價跨幅需要的時間。這個演算法有一個迴圈，從第 2 行開始，它會執行 n 次，每個股價一次。接著從第 5 行開始有一個內部迴圈會試著為每一次外部迴圈迭代尋找股價跨幅。對於每一個股價，它會比較該股價與所有之前的股價。在最壞的情況下，如果該股價是最高的價格，它會檢查所有之前的股價。如果股價 k 比之前所有的股價高，內部迴圈就會執行 k 次。因此，在最壞的情況下，也就是如果股價是愈來愈高的，第 7 行會執行以下的次數：

$$1+2+\cdots+n=\frac{n(n+1)}{2}$$

如果你不瞭解這個方程式，當你將數字 1、2、...、n 相加兩次時，就明白了：

$$\begin{array}{rcccccc} & 1 & + & 2 & +\cdots+ & n \\ + & n & +n-1 & + & \cdots+ & 1 \\ \hline & n+1 & +n+1 & +\cdots+ & n+1 & = n(n+1) \end{array}$$

因為第 6 行是這個演算法執行最多次的步驟，$n(n+1)/2$ 是這個演算法在最壞的情況下的執行時間。

談到演算法的執行時間時，我們感興趣的其實是輸入資料很大時的執行時間（在這個例子就是數字 n）。這是演算法的**漸近**（*asymptotic*）執行時間，因為它處理的是演算法在輸入資料無限增加時的行為。為此，我們會使用一些特殊的符號。對於任何函數 $f(n)$，如果大於某個初始正值的任意 n 都會讓 $f(n)$ 小於或等於另一個函數 $g(n)$ 乘以一個正的常數值 c，也就是 $cg(n)$，我們稱之為 $O(f(n))=g(n)$。更精確地說，如果有正的常數 c 與 n_0，會導致 $0\le f(n)\le cg(n)$ 且所有 $n\ge n_0$ 時，我們就說 $O(f(n))=g(n)$。

$O(f(n))$ 稱為“大 O 表示法”。請記得，我們想知道的是大的輸入值，因為那是可以節省最大資源的情況。圖 1.2 畫出兩個函數，$f_1(n)=20n+1000$ 與 $f_2(n)=n^2$。就小 n 值而言，$f_1(n)$ 的值比較大，但這種情況很快就會改變，之後，n^2 會快速增加。

大 O 表示法可以讓我們簡化函數。如果有一個函數 $f(n)=3n^3+5n^2+2n+1000$，我們只會得到 $O(f(n))=n^3$。為什麼？因為我們必定可以找到一個 c 值來讓 $0\le f(n)\le cn^3$。一般情況下，當

一個函數有多個項時，它的最大項很快就會支配函數的成長，所以我們在取大 O 時，會將最小的項拿掉。所以，$O(a_1n^k + a_2n^{k-1} + \cdots + a_nn + b) = O(n^k)$。

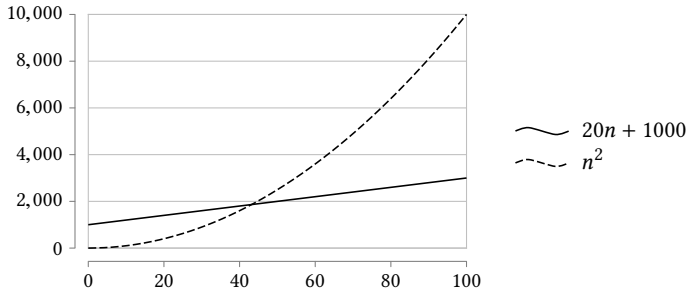

圖 1.2
比較 $O(f(n))$。

之前談到的演算法執行時間通常會被稱為演算法的**計算複雜度**（*computational complexity*），或簡稱為**複雜度**（*complexity*），因為我們會使用簡化版的函數來研究演算法的執行時間，事實上，多數演算法的執行時間函數都屬於少數的幾個簡化函數。也就是說，演算法的複雜度通常都屬於一個或少數的常見種類或族群。

第一種是**常數函數**（*constant function*），$f(n) = c$。它代表無論 n 是什麼，函數的值都是 c。除非 c 是很離譜的高值，否則這是理想中的最佳演算法函數。就大 O 表示法而言，根據定義，我們有個正的常數 c 與 n_0，讓 $0 \le f(n) \le cg(n) = c \cdot 1$。事實上，$c$ 是函數的常數值，且 $n_0 = 1$。因此，$O(c) = O(1)$。有這種行為的演算法稱為**常數時間演算法**（*constant time algorithm*）。其實這種名稱並不正確，因為它的意思不是無論輸入為何，演算法都會花費相同的時間。它代表演算法的執行時間上限與輸入是無關的。舉個簡單的演算法為例，“若 $x > 0$ 則將 x 值加上 y 值”花費的執行時間不一定相同：如果 $x > 0$，它會執行加法，否則就不做任何事情。但是它的上限是固定的，這個上限就是加法所需的時間，所以它屬於 $O(1)$ 族群。不幸的是，以常數時間來執行的演算法很罕見。最常見的常數時間操作之一，就是讀取陣列的一個元素，此時會花費固定的時間，無論想要讀取的元素的索引為何；之前看過，對於有 n 個元素的陣列 A，讀取 $A[0]$ 花費的時間與讀取 $A[n-1]$ 是相同的。

緊接在常數時間演算法之後的是**對數時間**（*logarithmic time*）演算法。對數函數（logarithmic function，或 *logarithm*），是 $\log_a(n)$，它的定義是 a 的幾次方可以得到 n：若 $y = \log_a(n)$，則 $n = a^y$。數字 a 是**對數的底數**。根據對數的定義，我們可以說 $x = a^{\log_a x}$，代表對數是“取一個數字的次方”的相反。事實上，$\log_3 27 = 3$，$3^3 = 27$。如果 $a = 10$，也就是對數的底數是 10 的話，我們只要寫成 $y = \log(n)$ 就可以了。在電腦中，我們很常遇到**底數為二的對數**，稱為二進位對數，所以會使用特殊的表示法，$\lg(n) = \log_2(n)$。它與所謂的**自然對數**不同，自然對數是底數為 e 的對數，$e \approx 2.71828$。自然對數也有特別的表示法，$\ln(n) = \log_e(n)$。

附帶說明一下 e 是怎麼來的。數字 e 有時稱為歐拉數（Euler's number），名稱來自 18 世紀的瑞士數學家 Leonhard Euler，但它也會在許多不同的領域出現。它是 $(1 + 1/n)^n$ 在 n 接近無限大時的上限。雖然它的名稱是歐拉，但它其實是另一位瑞士數學家發現的—17 世紀的 Jacob Bernoulli。當時 Bernoulli 試著找出一個公式來計算複利。

假設當你在銀行存入 d 元時，銀行給你 R% 的利息。如果每年計息一次，經過一年後，你的錢會變成 $d + d(R/100)$。設 $r = R/100$，你的錢會變成 $d(1 + r)$。你可以驗證，若 $R = 50$，$r = 1/2$ 時，你的錢會增加到 $1.5 \times d$。如果每年計息兩次，六個月一期的利率將會是 $r/2$。六個月之後，你會有 $d(1 + r/2)$。再過六個月之後的年末，你會有 $d(1 + r/2)(1 + r/2) = d(1 + r/2)^2$。如果每年計算 n 次利息，行話稱為**複利**，你在年末會有 $d(1 + r/n)^n$。若利率很優渥，$R = 100\%$，你會得到 $r = 1$；如果你不斷計算複利，也就是計算極小期間，n 會變成無限大。所以若 $d = 1$，你的錢在年末會成長為 $(1 + 1/n)^n = e$。附帶說明結束。

對數有一種基本特性：不同底數的對數之間的倍數是固定的，因為 $\log_a(n) = \log_b(n)/\log_b(a)$。例如，$\lg(n) = \log_{10}(n)/\log_{10}(2)$。因此，我們將所有對數函數全部歸類為同一個複雜度族群，通常表示成 $O(\log(n))$，但也會經常使用較具體的 $O(\lg(n))$。當演算法會不斷重複將一個問題分為兩個時，就會出現 $O(\lg(n))$ 複雜度，因為不斷將某個東西分成兩個，其實就是對它套用對數函數。與“搜尋”有關的演算法是很重要的對數時間演算法：最快速的搜尋演算法是以底數為二的對數時間來執行的。

比對數時間演算法更耗時的是以時間 $f(n) = n$ 來執行的**線性時間演算法**（*linear time algorithms*），也就是執行時間會與輸入成正比。這些演算法的複雜度是 $O(n)$。這些演算法可能必須掃描所有的輸入來找出答案。例如，當我們要搜尋未以任何方式來排序的隨機項目集合時，可能就要一一檢查它們才能找出想要的結果，因此這種搜尋是以線性時間來執行的。

比線性時間還要慢的是**對數線性時間演算法**（*loglinear time algorithms*），其 $f(n) = n \log(n)$，所以我們寫成 $O(n \log(n))$。與之前一樣，對數可採用任何底數，不過在實務上，通常採用底數為二的對數。這些演算法就某方面而言，是線性時間演算法與對數時間演算法的結合。它們可能會反覆切割問題，並對每一個切割出來的部分套用線性時間演算法。良好的排序演算法會有對數線性時間複雜度。

當描述演算法執行時間的函數是多項式 $f(n) = (a_1 n^k + a_2 n^{k-1} + \cdots + a_n n + b)$ 時，如前所述，複雜度是 $O(n^k)$，這種演算法是**多項式時間演算法**（*polynomial time algorithm*）。許多演算法都是以多項式時間來執行的；有一種重要的分支是以 $O(n^2)$ 時間執行的演算法，我們稱它為**二次時間演算法**（*quadratic time algorithms*）。有些沒效率的排序方法會以二次時間來執行，這就好像將兩個 n 位數的數字相乘的標準做法。不過，其實我們有更高效的乘法可用，當我們想要採取高效的算術運算時，就會使用這些比較有效率的方式。

比多項式時間演算法還要慢的是**指數時間演算法**（*exponential time algorithms*），其中 $f(n) = c^n$，c 是常數值，所以得到 $O(c^n)$。注意 n^c 與 c^n 的差異。雖然我們將 n 與指數的位置互換了，但產生的函數有很大的差異。如前所述，次方是對數函數的反向運算，它只是將一個常數變大成一個變數。注意，取次方是 c^n；**指數函數**（*exponential function*）是 $c = e$ 時的特例，也就是 $f(n) = e^n$，e 是之前談過的歐拉數。指數會在處理這種問題時出現：有 n 個輸入，且這 n 個輸入中的每一個輸入都有可能接收 c 種值，而且我們必須嘗試所有可能出現的情況。第一個輸入有 c 種值，且第二個輸入對於每一個值會有 c 種值，總共是 $c \times c = c^2$。對於 c^2 的每一個案例，第三個輸入有 c 種值，所以是 $c^2 \times c = c^3$；以此類推，直到最後一個輸入得到 c^n。

比指數時間演算法還要慢的是 $O(n!)$ 的**階乘時間演算法**（*factorial time algorithms*），階乘數的定義是 $n! = 1 \times 2 \times \cdots \times n$，極端的情況是 $0! = 1$。如果我們需要嘗試所有可能的輸入**排列**（*permutations*）來解決問題，就會產生階乘。“排列”指的是以不同的順序來排列一系列的值。例如，值 [1, 2, 3] 的排列有：[1, 2, 3]、[1, 3, 2]、[2, 1, 3]、[2, 3, 1]、[3, 1, 2] 與 [3, 2, 1]。第一個位置有 n 種可能的值，因為我們已經使用一個值了，所以第二個位置有 $n - 1$ 種可能的值，如此一來，前兩個位置會有 $n \times (n - 1)$ 種不同的排列。其餘的位置以此類推，直到最後一個位置，屆時只有一個可能的值。所以我們總共有 $n \times (n - 1) \cdots \times 1 = n!$ 種排列組合。這種階乘數字會在洗牌時出現：洗一副撲克牌可能會出現 52! 種結果，這是個天文數字。

根據經驗，複雜度在多項式時間以下的演算法都是好的演算法，所以我們的目標通常是找出具有這種效能的演算法。不幸的是，就我們所知，所有重要的問題都沒有多項式時間的演算法！你可以從表 1.1 看到，如果一個問題的執行時間是 $O(2^n)$，這種演算法除了處理輸入值極小的小問題之外沒有其他用途。你也可以從圖 1.3 的最底下看到，$O(2^n)$ 與 $O(n!)$ 的值從很小的 n 值就開始飆升了。

表 1.1

函數的增長。

函數	輸入大小				
	1	10	100	1000	1,000,000
$\lg(n)$	0	3.32	6.64	9.97	19.93
n	1	10	100	1000	1,000,000
$n \ln(n)$	0	33.22	664.39	9965.78	1.9×10^{7}
n^2	1	100	10,000	1,000,000	10^{12}
n^3	1	1000	1,000,000	10^{9}	10^{18}
2^n	2	1024	1.3×10^{30}	10^{301}	$10^{10^{5.5}}$
$n!$	1	3,628,800	9.33×10^{157}	4×10^{2567}	$10^{10^{6.7}}$

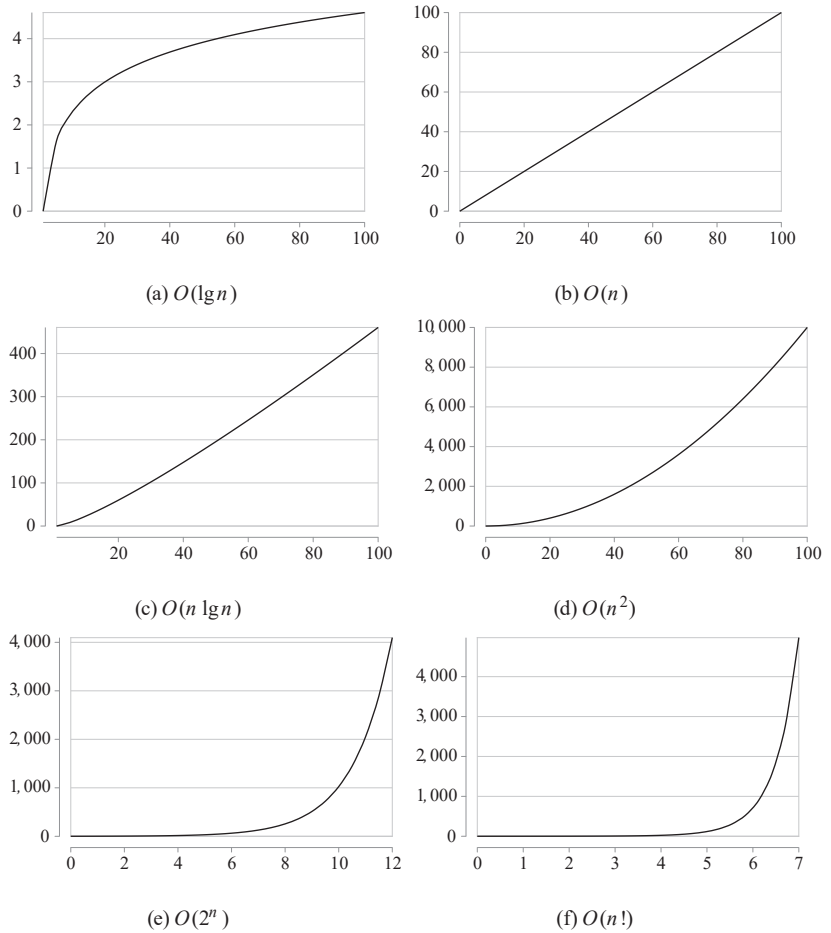

圖 1.3
各種複雜度族群。

圖 1.3 以線條來表示函數，但實際上，在研究演算法時，數字 n 是自然數，所以你看到的應該是散佈圖，顯示的是點，而不是線條。對數、線性、對數線性與多項式函數當然是針對實數來定義的，所以使用一般的函數定義，將它們繪成線條是沒問題的。指數通常是整數，但是有時也會出現指數為有理數的次方，因為 $x^{(a/b)} = (x^a)^{(1/b)} = \sqrt[b]{x^a}$。則指數為實數的次方的定義是 $b^x = (e^{\ln b})^x = e^{x \ln b}$。關於階乘，我們可以藉由一些較進階的數學，完全用實數來定義它們（負階乘視為無限大）。所以用線條來描繪複雜度函數是合理的做法。

為了避免讓你誤以為 $O(2^n)$ 或 $O(n!)$ 複雜度在實際情況下不容易出現，我們來考慮著名的（或臭名遠播的）巡遊推銷員問題。這個問題是，有一位推銷員必須前往一些城市，並且只拜訪每一個城市一次。每一個城市都會直接與其他的城市相連（推銷員可能是搭飛機）。重點是，在過程中，推銷員必須盡量減少旅程的公里數。其中一種直接的解法是嘗試排列所有可能的城市順序。如果有 n 個城市，它就是 $O(n!)$。有一種比較好的演算法可以用 $O(n^2 2^n)$ 來解決這個問題—它的確有稍微改善，但實際的幅度不大。我們該如何處理這個問題（與其他類似的問題）？事實上，我們可能找不到可以提供精準答案的演算法，但或許可以找到可提供近似結果的演算法。

大 O 代表的是演算法效能的**上限**。它的相反是**下限**，代表我們知道經過一些初始值之後，它的複雜度一定不會比某個函數還要好。這稱為“大 Omega”，或 $\Omega(f(n))$，準確的定義是，如果有正的常數 c 與 n_0，且所有 $n \geq n_0$ 都可讓 $f(n) \geq cg(n) \geq 0$ 時，則 $\Omega(f(n)) = g(n)$。定義大 O 與大 Omega 後，我們也要定義同時有上下限的情況。也就是“大 Theta”，即，若且唯若 $O(f(n)) = g(n)$ 且 $\Omega(f(n)) = g(n)$，則 $\Theta(f(n)) = g(n)$。所以我們知道演算法的執行時間的上下限是用同一個函數乘上一個常數來界定的。你可以想像在那個函數上有一個帶狀的演算法執行時間。

1.3 使用堆疊來處理股價跨幅

回到股價跨幅問題。我們發現一個複雜度為 $O(n(n + 1/2))$ 的演算法。之前談過，這相當於 $O(n^2)$。我們可以得到更好的結果嗎？回到圖 1.1，注意，當我們在第六天時，並不需要比較之前直到第一天的每一天，因為我們已經路過第六天之前的每一天了，所以“知道”第二、三與四天的股價都小於或等於第六天，如果我們設法保存這個認知，就只需要比較一天的價格，而不需要比較每一天。

這是一種常見的模式。想像你在 k 日。如果 $k-1$ 日的股價小於或等於 k 日的股價，我們會得到 $quotes[k-1] \leq quotes[k]$，或等效的 $quotes[k] \geq quotes[k-1]$，所以就不需要與 $k-1$ 做比較。為什麼？就未來的日期 $k+j$ 而言，如果 $k+j$ 的股價小於 k 日的股價，$quotes[k+j] < quotes[k]$，我們就不需要與 $k-1$ 比較，因為跨幅是從 $k+j$ 到 k。如果 $k+j$ 的股價大於 k 的股價，我們已經知道 $quotes[k+j] \geq quotes[k-1]$，因為 $quotes[k+j] \geq quotes[k]$ 且 $quotes[k] \geq quotes[k-1]$。所以每當我們往回搜尋跨幅的終點時，可以將值小於跨幅起始日的每一天扔掉，爾後在計算跨幅時，也可以將被扔掉的這幾天排除在外。

以下的比喻或許對你有幫助：想像你坐在圖 1.4 的六日長柱的上面，往前直視，而不是往下看，你只會看到一日的長柱。它是唯一需要與六日的股價比較的對象。一般來說，在每一天，你只需要比較直視看到的那一天就可以了。也就是說，在演算法 1.1 中，當我們在第 5 行開始比較之前的每一天時，是在浪費時間。我們可以使用一些機制來掌握已建立的最大跨幅的上限，省下這些時間。

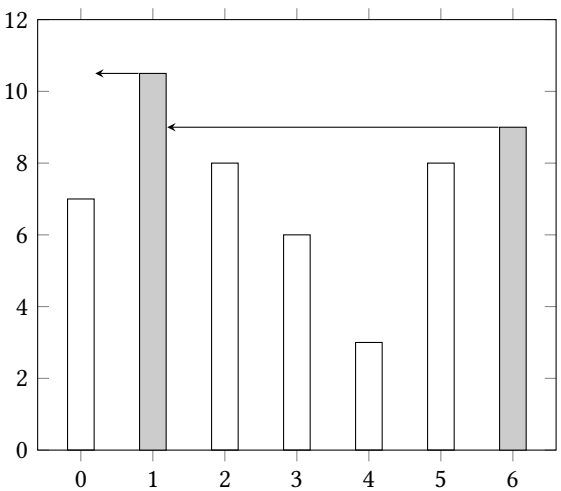

圖 1.4
優化股價跨幅。

為此，我們可以使用一種特殊的資料結構來保存資料，稱為**堆疊**（*stack*）。堆疊是一種簡單的資料結構，我們可以在它裡面放入資料，一個接一個，以及取回它們。我們每次都會取回最後一個放入的資料。堆疊的動作就像餐廳的一疊托盤，每一個托盤都被疊在另一個上面，我們只能拿最上面的托盤，也只能將托盤放在那一疊的最上面。因為被最後一個放上的托盤也是被第一個拿走的托盤，所以我們將堆疊稱為後入先出（Last In First Out，LIFO）結構。圖 1.5 展示類似托盤的行為，在堆疊中加入與移除項目的操作。

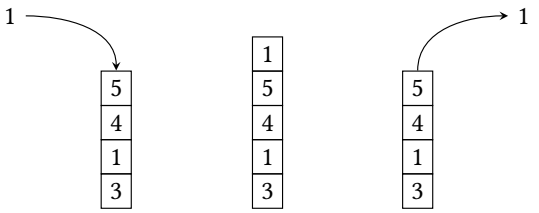

圖 1.5
將項目加入堆疊與移除。

當我們談到資料結構時，也必須說明可對它執行的操作。我們知道與陣列有關的操作有兩種：建立陣列與存取元素。關於堆疊，如前所述，有五種操作，包括：

- `CreateStack()` 可建立一個空堆疊。
- `Push(`*S, i*`)` 可將項目 *i* push 至堆疊 *S* 的最上面。
- `Pop(`*S*`)` 可將堆疊最上面的項目取出（pop）。這項操作會回傳該項目。如果堆疊是空的就不可以執行這項操作（會得到錯誤）。
- `Top(`*S*`)` 可取得堆疊 *S* 最上面的項目的值，但不移除它，堆疊會保持原貌。如果堆疊是空的，一樣不可以執行這項操作，我們會得到錯誤。
- `IsStackEmpty(`*S*`)` 如果堆疊 *S* 是空的則回傳 TRUE，否則 FALSE。

現實世界的堆疊是有限的，我們只能 push 它的上限之內的元素數量，畢竟電腦的記憶體是有限的。真正的堆疊還有一些額外的操作，例如查看堆疊的元素數量（堆疊的大小），以及它是否已滿，那些操作與這些虛擬碼的演算法無關，所以我們不討論它們，爾後使用的其他資料結構的相關操作也一樣。

我們可以藉由堆疊來執行剛才討論出來的概念，用演算法 1.2 來解決股票跨幅問題。一如往常，在一開始的第 1 行，我們先建立一個大小為 n 的陣列。根據定義，第一天的跨幅是一，所以我們在第 2 行將 $spans[0]$ 初始化為這個值。這一次我們使用堆疊來儲存需要比較的日期。所以在第 3 行建立一個新的空堆疊。在一開始，我們知道一個簡單的道理：第一天的股價不會低於第一天的股價，所以我們在第 4 行 push 0 到堆疊內，也就是第一天的索引。

演算法 1.2：堆疊股價跨幅演算法。

```
StackStockSpan(quotes) → spans
    輸入：quotes，有 n 筆股價的陣列
    輸出：spans，有 n 筆股價跨幅的陣列

 1  spans ← CreateArray(n)
 2  spans[0] ← 1
 3  S ← CreateStack()
 4  Push(S, 0)
 5  for i ← 1 to n do
 6      while not IsStackEmpty(S) and quotes[Top(S)] ≤ quotes[i] do
 7          Pop(S)
 8      if IsStackEmpty(S) then
 9          spans[i] ← i + 1
10      else
11          spans[i] ← i − Top(S)
12      Push(S, i)
13  return spans
```

第 5–12 行的迴圈會處理所有後續的日期。第 6–7 行的內部迴圈會往回查看，來找出比目前正在處理的這一天的股價還要高的最近日期。它的做法是：只要堆疊最上面那一天的股價小於或等於我們正在處理的日期的股價（第 6 行），就從堆疊 pop 一個項目（第 7 行）。如果我們因為堆疊耗盡而離開內部迴圈（第 8 行），而且正處於 i 日，那麼在那天之前的每一天的股價都比較低，所以跨幅是 $i + 1$，我們在第 9 行將 *spans*[i] 設為那個值。否則（第 10 行），跨幅會從 i 日延伸到堆疊最上面的那一天，所以我們在第 11 行將 *spans*[i] 設為這兩天的差。在回到迴圈的開頭之前，我們將 i 日 push 至堆疊的最上面。如此一來，在外部迴圈的結束時，堆疊裡面會有股價不低於我們正在檢查的那一天的日期。在下一次迭代迴圈時，我們可用它來與關鍵日期比較，也就是高於我們的視線的日期，這就是我們要的結果。

在演算法的第 6 行有一個值得注意的細節。如果 S 是空的，計算 `Top(`S`)` 就是錯誤的。但這種事情不會發生，因為有一種與估算條件的方法有關的重要性質，稱為**短路估算**（*short circuit evaluation*）。這個特性的意思是，當我們處理一個涉及邏輯布林運算子的運算式時，只要得到運算式的最終結果，計算就會停止，不會繼續費力計算運算式的其餘的部分。舉例而言，有個運算式：**if** $x > 0$ **and** $y > 0$。如果我們知道 $x \leq 0$，整個運算式就是 false，無論 y 的值為何；我們完全不需要估算運算式的第二個部分。同樣的，這個運算式：**if** $x > 0$ **or** $y > 0$，當我們知道 $x > 0$ 時，就不需要計算運算式的第二個部分：y 的地方，因為我們已經知道，當第一個部分是 true 時，整個運算式都是 true。表 1.2 是當我們用 **and** 與 **or** 來計算兩個布林運算式時的各種情況。灰色的兩列代表運算的結果與第二個部分無關，因此只要知道第一個部分的值，計算就會停止。因為短路計算，當 `IsStackEmpty(`S`)` 回傳 TRUE 時，也就是 **not** `IsStackEmpty(`S`)` 是 FALSE 時，我們不會計算 **and** 右邊的 `Top(`S`)` 部分，所以可避免錯誤。

表 1.2
布林短路估算。

運算子	a	b	結果
	T	T	T
and	T	F	F
	F	T/F	F
	T	T/F	T
or	F	T	T
	F	F	F

你可以在圖 1.6 看到演算法的工作方式以及視線比喻。每一張小圖的右邊有每次開始迭代迴圈時的堆疊情形；灰色的長條代表堆疊內的日期，虛線的長條代表尚未處理的日期。小圖下方的黑色圓圈代表目前正在處理的日期。

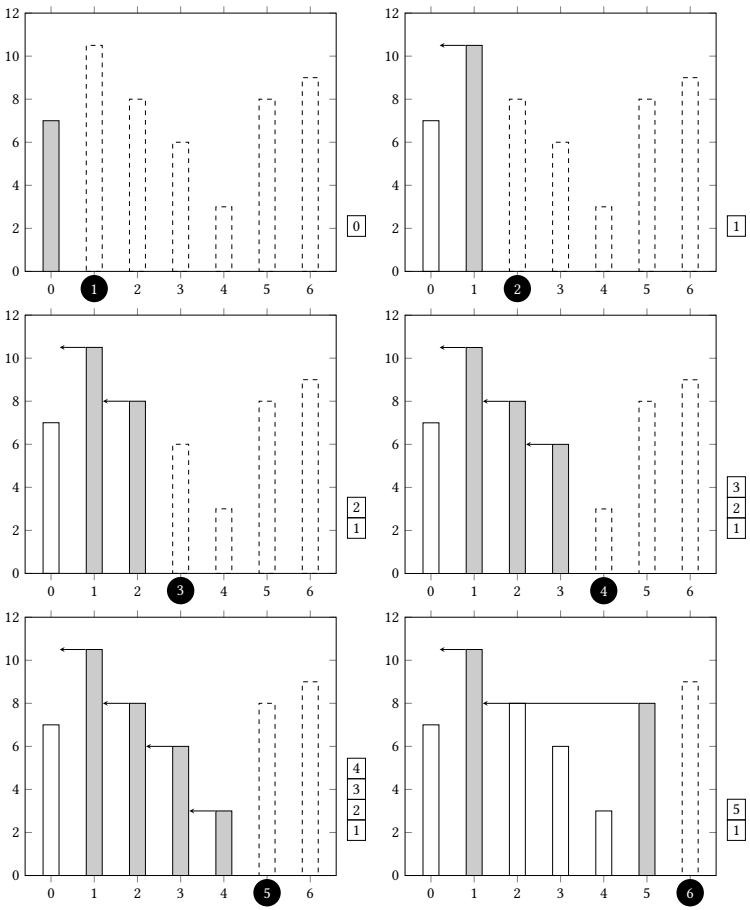

圖 1.6
股價跨幅的視線。

第一張小圖的 $i = 1$，我們必須檢查目前的日期的值與堆疊內的其他日期的值，裡面只有零日。一日的價格比零日高。也就是說，從現在開始，我們不需要比較一日之前的價格；我們的視線只會到達一日；所以在下一次迭代，$i = 2$，堆疊裡面有數字 1。二日的價格低於一日。

這代表如果三日的值小於二日的值，從三日開始的跨幅會在二日結束，或者如果三日的值不小於二日的值，會在一日結束。但是，它不可能會在零日結束，因為零日的價格小於一日。$i = 3$ 與 $i = 4$ 也是類似的情況。但是當我們到達 $i = 5$ 時，發現以後再也不需要與二、三與四日比較了，這幾天會被遮蔽，因為我們在五日。或者，在視線比喻中，我們的視野在一日之前是暢通無阻的。在這兩天之間的每一天都可以從堆疊 pop，所以堆疊只會有 5 與 1，因此在 $i = 6$ 時，我們最多只需要比較這兩天。如果有一天的值大於或等於五日的值，它肯定會跳過四、三、二日的值；我們無法確定的是它的值是否到達一日的值。當我們完成六日時，堆疊只有數字 6 與 1。

這種做法會不會比之前的做法好？在演算法 1.2 中，從第 5 行開始的迴圈會執行 $n - 1$ 次，每一次，稱之為第 i 次迭代，會執行從第 6 行開始的內部迴圈的 Pop 操作 p_i 次。也就是說 Pop 操作總共會執行 $p_1 + p_2 + \cdots + p_{n-1}$ 次，每個外部迴圈迭代 p_i 次。我們不知道數字 p_i 是什麼，但是仔細研究演算法就可以知道每一天只會被 push 至堆疊一次，第一天在第 4 行，之後幾天在第 12 行。因此在第 7 行，演算法最多只會將每一天從堆疊 pop 一次。所以，整個演算法的執行過程，在所有的外部迴圈迭代中，第 7 行的執行次數不會超過 $n - 1$ 次；注意，最後一天被 push 而不是被 pop。換句話說，$p_1 + p_2 + \cdots + p_{n-1} = n$，也就是整個演算法是 $O(n)$；第 7 行是執行最多次的操作，因為它在內部迴圈中，但 5-12 行的其他程式不是。

繼續分析可以知道，演算法 1.1 只能提供最差的估計結果，因此我們的估計是演算法效能的下限—演算法不可能用少於 n 個步驟來完成，因為我們必須遍歷 n 天。所以這個演算法的計算複雜度也是 $\Omega(n)$，因此它是 $\Theta(n)$。

堆疊與我們即將看到的其他資料結構一樣有許多用途。LIFO 行為在電腦中很常見，所以你可以在許多地方看到堆疊，包括用機器語言寫成的低階程式，以及超級電腦執行的巨型問題。它們是人類多年使用電腦來處理問題獲得的經驗結晶，所以資料結構通常會被放在最重要的位置，事實證明，演算法會使用類似的方法來管理它們所處理的資料，人們已經整理好這些方法，因此，當我們尋求解決之道時，會利用它們的功能來開發演算法。