Hello 演算法

動畫圖解、一鍵執行的資料結構與演算法教程

靳宇棟（@krahets） 著

Hello 演算法

Rust 語言版

作者：靳宇棟（@krahets）
程式碼審閱：伍志豪（@night-cruise）、榮怡（@rongyi）

Release 1.2.0
2024-12-06

序

兩年前，我在力扣上分享了“劍指 Offer”系列題解，受到了許多讀者的鼓勵與支持。在與讀者交流期間，我最常被問到的一個問題是“如何入門演算法”。漸漸地，我對這個問題產生了濃厚的興趣。

兩眼一抹黑地刷題似乎是最受歡迎的方法，簡單、直接且有效。然而刷題就如同玩“踩地雷”遊戲，自學能力強的人能夠順利將地雷逐個排掉，而基礎不足的人很可能被炸得焦頭爛額，並在挫折中步步退縮。通讀教材也是一種常見做法，但對於面向求職的人來說，畢業論文、投遞履歷、準備筆試和面試已經消耗了大部分精力，啃厚重的書往往變成了一項艱鉅的挑戰。

如果你也面臨類似的困擾，那麼很幸運這本書“找”到了你。本書是我對這個問題給出的答案，即使不是最佳解，也至少是一種積極的嘗試。本書雖然不足以讓你直接拿到 Offer，但會引導你探索資料結構與演算法的“知識地圖”，帶你了解不同“地雷”的形狀、大小與分布位置，讓你掌握各種“排雷方法”。有了這些本領，相信你可以更加自如地刷題與閱讀文獻，逐步構建起完整的知識體系。

我深深認同費曼教授所言：“Knowledge isn’t free. You have to pay attention.”從這個意義上看，這本書並非完全“免費”。為了不辜負你為本書所付出的寶貴“注意力”，我會竭盡所能，投入最大的“注意力”來完成本書的創作。

本人自知學疏才淺，書中內容雖然已經過一段時間的打磨，但一定仍有許多錯誤，懇請各位老師與同學批評指正。

> 本書中的程式碼附有可一鍵執行的原始檔，託管於 github.com/krahets/hello-algo 倉庫。
> 動畫在 PDF 內的展示效果有限，可訪問 www.hello-algo.com 網頁版以獲得更佳的閱讀體驗。

推薦語

> “一本通俗易懂的資料結構與演算法入門書，引導讀者手腦並用地學習，強烈推薦演算法初學者閱讀！”
> **——鄧俊輝，清華大學計算機系教授**

> “如果我當年學資料結構與演算法時有《Hello 演算法》，學起來應該會簡單 10 倍！”
> **——李沐，亞馬遜資深首席科學家**

電腦的出現為世界帶來了巨大的變革，它憑藉高速的運算能力與卓越的可程式化特性，成為執行演算法與處理資料的理想媒介。無論是電玩遊戲的逼真畫面、自動駕駛的智慧決策，還是 AlphaGo 的精彩棋局、ChatGPT 的自然互動，這些應用都是演算法在電腦上的精妙演繹。

事實上，在電腦問世之前，演算法和資料結構就已經存在於世界的各個角落。早期的演算法相對簡單，例如古代的計數方法與工具製作步驟等。隨著文明的進步，演算法逐漸變得更加精細和複雜。從巧奪天工的匠人技藝、到解放生產力的工業產品、再到宇宙運行的科學規律，幾乎每一件平凡或令人驚嘆的事物背後，都隱藏著精妙的演算法思想。

同樣，資料結構無處不在：大到社會網絡，小到地鐵路線，許多系統都可以建模為“圖”；大到一個國家，小到一個家庭，社會的主要組織形式呈現出“樹”的特徵；冬天的衣服就像“堆疊”，最先穿上的最後才能脫下；羽毛球筒則如同“佇列”，一端放入、一端取出；字典就像一個“雜湊表”，能夠快速查找目標詞條。

本書旨在透過清晰易懂的動畫圖解與可執行的程式碼範例，使讀者理解演算法和資料結構的核心概念，並能夠透過程式設計來實現它們。在此基礎上，本書致力於揭示演算法在複雜世界中的生動體現，展現演算法之美。希望本書能夠幫助到你！

目　錄

第 0 章　前言

Abstract

演算法猶如美妙的交響樂，每一行程式碼都像韻律般流淌。

願這本書在你的腦海中輕輕響起，留下獨特而深刻的旋律。

0.1　關於本書

本專案旨在建立一本開源、免費、對新手友好的資料結構與演算法入門教程。

- 全書採用動畫圖解，內容清晰易懂、學習曲線平滑，引導初學者探索資料結構與演算法的知識地圖。
- 源程式碼可一鍵執行，幫助讀者在練習中提升程式設計技能，瞭解演算法工作原理和資料結構底層實現。
- 提倡讀者互助學習，歡迎大家在評論區提出問題與分享見解，在交流討論中共同進步。

0.1.1　讀者物件

若你是演算法初學者，從未接觸過演算法，或者已經有一些刷題經驗，對資料結構與演算法有模糊的認識，在會與不會之間反覆橫跳，那麼本書正是為你量身定製的！

如果你已經積累一定的刷題量，熟悉大部分題型，那麼本書可助你回顧與梳理演算法知識體系，倉庫源程式碼可以當作“刷題工具庫”或“演算法字典”來使用。

若你是演算法“大神”，我們期待收到你的寶貴建議，或者一起參與創作。

> **前置條件**
> 你需要至少具備任一語言的程式設計基礎，能夠閱讀和編寫簡單程式碼。

0.1.2　內容結構

本書的主要內容如圖 0-1 所示。

- **複雜度分析**：資料結構和演算法的評價維度與方法。時間複雜度和空間複雜度的推算方法、常見型別、示例等。
- **資料結構**：基本資料型別和資料結構的分類方法。陣列、鏈結串列、堆疊、佇列、雜湊表、樹、堆積、圖等資料結構的定義、優缺點、常用操作、常見型別、典型應用、實現方法等。
- **演算法**：搜尋、排序、分治、回溯、動態規劃、貪婪等演算法的定義、優缺點、效率、應用場景、解題步驟和示例問題等。

圖 0-1　本書主要內容

0.1.3　致謝

本書在開源社群眾多貢獻者的共同努力下不斷完善。感謝每一位投入時間與精力的撰稿人，他們是（按照 GitHub 自動生成的順序）：krahets、coderonion、Gonglja、nuomi1、Reanon、justin-tse、hpstory、danielsss、curtishd、night-cruise、S-N-O-R-L-A-X、msk397、gvenusleo、khoaxuantu、RiverTwilight、rongyi、gyt95、zhuoqinyue、K3v123、Zuoxun、mingXta、hello-ikun、FangYuan33、GN-Yu、yuelinxin、longsizhuo、Cathay-Chen、guowei-gong、xBLACKICEx、IsChristina、JoseHung、qualifier1024、QiLOL、pengchzn、Guanngxu、L-Super、WSL0809、Slone123c、lhxsm、yuan0221、what-is-me、theNefelibatas、longranger2、cy-by-side、xiongsp、JeffersonHuang、Transmigration-zhou、magentaqin、Wonderdch、malone6、xiaomiusa87、gaofer、bluebean-cloud、a16su、Shyam-Chen、nanlei、hongyun-robot、Phoenix0415、MolDuM、Nigh、he-weilai、junminhong、mgisr、iron-irax、yd-j、XiaChuerwu、XC-Zero、seven1240、SamJin98、wodray、reeswell、NI-SW、Horbin-Magician、Enlightenus、xjr7670、YangXuanyi、DullSword、boloboloda、iStig、qq909244296、jiaxianhua、wenjianmin、keshida、kilikilikid、lclc6、lwbaptx、liuxjerry、lucaswangdev、lyl625760、hts0000、gledfish、fbigm、echo1937、szu17dmy、dshlstarr、Yucao-cy、coderlef、czruby、bongbongbakudan、beintentional、ZongYangL、ZhongYuuu、luluxia、xb534、bitsmi、ElaBosak233、baagod、zhouLion、yishangzhang、yi427、yabo083、weibk、wangwang105、th1nk3r-ing、tao363、4yDX3906、syd168、steventimes、sslmj2020、smilelsb、siqyka、selear、sdshaoda、Xi-Row、popozhu、nuquist19、noobcodemaker、XiaoK29、chadyi、ZhongGuanbin、shanghai-Jerry、JackYang-hellobobo、Javesun99、lipusheng、BlindTerran、ShiMaRing、FreddieLi、FloranceYeh、iFleey、fanchenggang、gltianwen、goerll、Dr-XYZ、nedchu、curly210102、CuB3y0nd、

KraHsu、CarrotDLaw、youshaoXG、bubble9um、fanenr、eagleanurag、LifeGoesOnionOnionOnion、52coder、foursevenlove、KorsChen、hezhizhen、linzeyan、ZJKung、GaochaoZhu、hopkings2008、yang-le、Evilrabbit520、Turing-1024-Lee、thomasq0、Suremotoo、Allen-Scai、Risuntsy、Richard-Zhang1019、qingpeng9802、primexiao、nidhoggfgg、1ch0、MwumLi、martinx、ZnYang2018、hugtyftg、logan-qiu、psychelzh、Keynman、KeiichiKasai 和 0130w。

本書的程式碼審閱工作由 coderonion、curtishd、Gonglja、gvenusleo、hpstory、justin-tse、khoaxuantu、krahets、night-cruise、nuomi1、Reanon 和 rongyi 完成（按照首字母順序排列）。感謝他們付出的時間與精力，正是他們確保了各語言程式碼的規範與統一。

在本書的創作過程中，我得到了許多人的幫助。

- 感謝我在公司的導師李汐博士，在一次暢談中你鼓勵我“快行動起來”，堅定了我寫這本書的決心；
- 感謝我的女朋友泡泡作為本書的首位讀者，從演算法小白的角度提出許多寶貴建議，使得本書更適合新手閱讀；
- 感謝騰寶、琦寶、飛寶為本書起了一個富有創意的名字，喚起大家寫下第一行程式碼“Hello World!”的美好回憶；
- 感謝校銓在智慧財產權方面提供的專業幫助，這對本開源書的完善起到了重要作用；
- 感謝蘇潼為本書設計了精美的封面和 logo ，並在我的強迫症的驅使下多次耐心修改；
- 感謝 @squidfunk 提供的排版建議，以及他開發的開源文件主題 Material-for-MkDocs 。

在寫作過程中，我閱讀了許多關於資料結構與演算法的教材和文章。這些作品為本書提供了優秀的範本，確保了本書內容的準確性與品質。在此感謝所有老師和前輩的傑出貢獻！

本書倡導手腦並用的學習方式，在這一點上我深受《動手學深度學習》的啟發。在此向各位讀者強烈推薦這本優秀的著作。

衷心感謝我的父母，正是你們一直以來的支持與鼓勵，讓我有機會做這件富有趣味的事。

0.2　如何使用本書

> **Tip**
> 為了獲得最佳的閱讀體驗，建議你通讀本節內容。

0.2.1　行文風格約定

- 標題後標註 `*` 的是選讀章節，內容相對困難。如果你的時間有限，可以先跳過。
- 專業術語會使用黑體（紙質版和 PDF 版）或新增下劃線（網頁版），例如**陣列**（array）。建議記住它們，以便閱讀文獻。
- 重點內容和總結性語句會 **加粗**，這類文字值得特別關注。
- 有特指含義的詞句會使用“引號”標註，以避免歧義。
- 當涉及程式語言之間不一致的名詞時，本書均以 Python 為準，例如使用 `None` 來表示“空”。
- 本書部分放棄了程式語言的註釋規範，以換取更加緊湊的內容排版。註釋主要分為三種類型：標題註釋、內容註釋、多行註釋。

```
/* 標題註釋，用於標註函式、類別、測試樣例等 */

// 內容註釋，用於詳解程式碼

/**
 * 多行
 * 註釋
 */
```

0.2.2　在動畫圖解中高效學習

相較於文字，影片和圖片具有更高的資訊密度和結構化程度，更易於理解。在本書中，**重點和難點知識將主要透過動畫以圖解形式展示**，而文字則作為解釋與補充。

如果你在閱讀本書時，發現某段內容提供瞭如圖 0-2 所示的動畫圖解，**請以圖為主、以文字為輔**，綜合兩者來理解內容。

圖 0-2　動畫圖解示例

0.2.3　在程式碼實踐中加深理解

本書的配套程式碼託管在 GitHub 倉庫。如圖 0-3 所示，**源程式碼附有測試樣例，可一鍵執行**。

如果時間允許，**建議你參照程式碼自行敲一遍**。如果學習時間有限，請至少通讀並執行所有程式碼。

與閱讀程式碼相比，編寫程式碼的過程往往能帶來更多收穫。**動手學，才是真的學**。

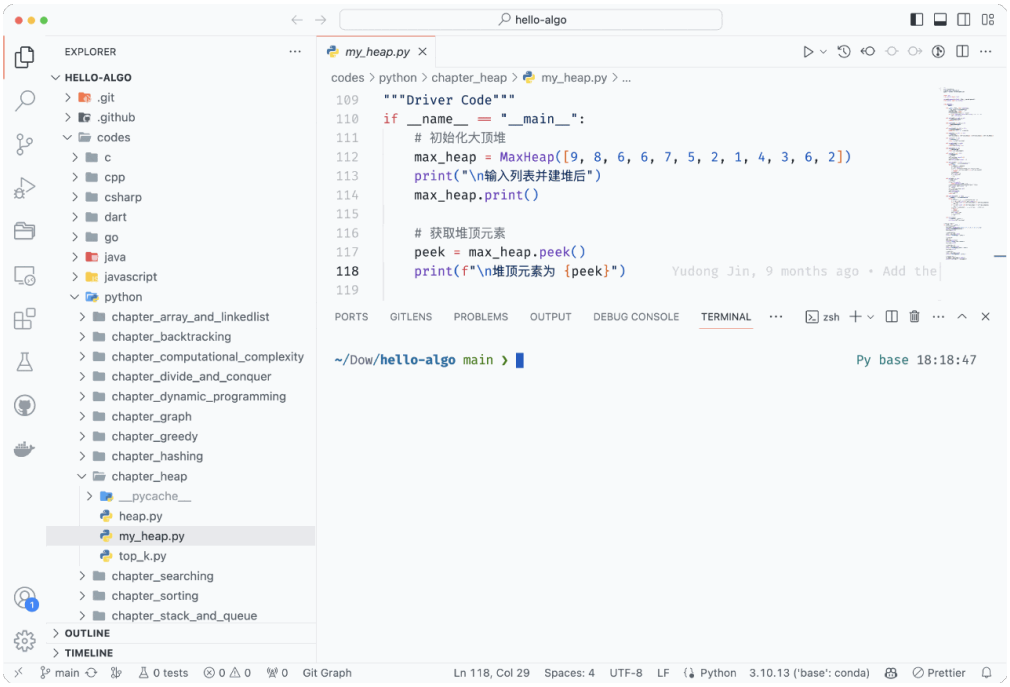


圖 0-3　執行程式碼示例

執行程式碼的前置工作主要分為三步。

第一步：安裝本地程式設計環境。請參照附錄所示的教程進行安裝，如果已安裝，則可跳過此步驟。

第二步：克隆或下載程式碼倉庫。前往 GitHub 倉庫。如果已經安裝 Git ，可以透過以下命令克隆本倉庫：

```
git clone https://github.com/krahets/hello-algo.git
```

當然，你也可以在圖 0-4 所示的位置，點選“Download ZIP”按鈕直接下載程式碼壓縮包，然後在本地解壓即可。

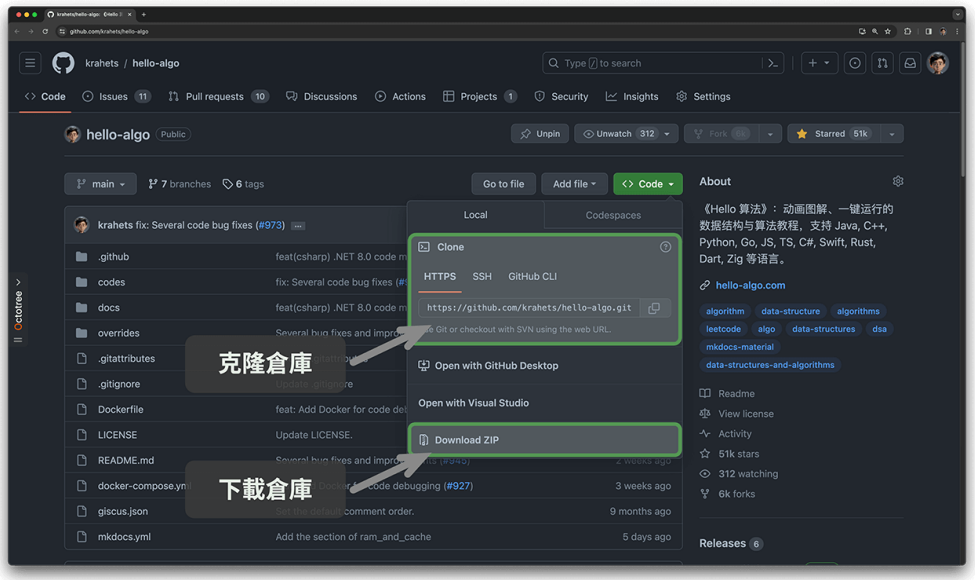


圖 0-4　克隆倉庫與下載程式碼

第三步：執行源程式碼。如圖 0-5 所示，對於頂部標有檔案名稱的程式碼塊，我們可以在倉庫的 `codes` 檔案夾內找到對應的源程式碼檔案。源程式碼檔案可一鍵執行，將幫助你節省不必要的除錯時間，讓你能夠專注於學習內容。

圖 0-5　程式碼塊與對應的源程式碼檔案

除了本地執行程式碼，**網頁版還支持 Python 程式碼的視覺化執行**（基於 pythontutor 實現）。如圖 0-6 所示，你可以點選程式碼塊下方的“視覺化執行”來展開檢視，觀察演算法程式碼的執行過程；也可以點選“全屏觀看”，以獲得更好的閱覽體驗。

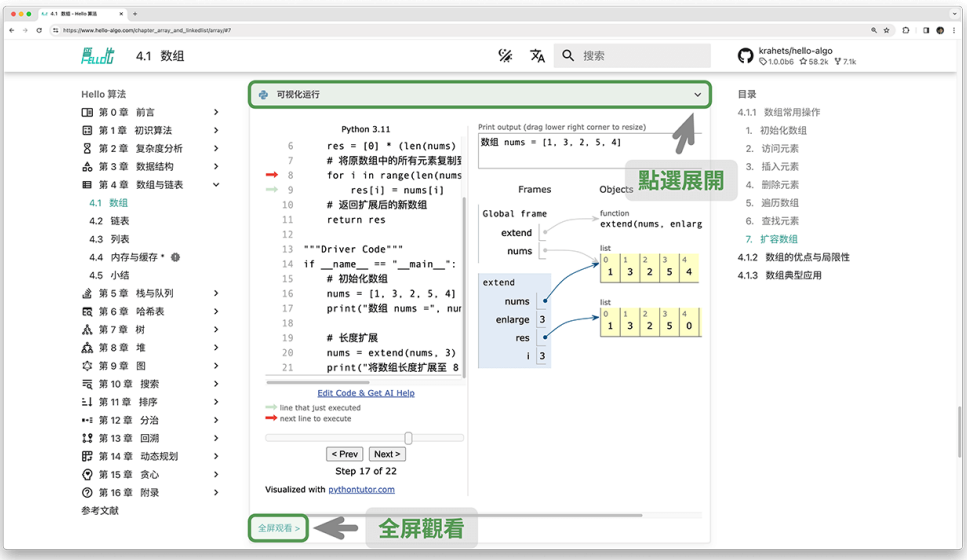


圖 0-6　Python 程式碼的視覺化執行

0.2.4　在提問討論中共同成長

在閱讀本書時，請不要輕易跳過那些沒學明白的知識點。**歡迎在評論區提出你的問題**，我和小夥伴們將竭誠為你解答，一般情況下可在兩天內回覆。

如圖 0-7 所示，網頁版每個章節的底部都配有評論區。希望你能多關注評論區的內容。一方面，你可以瞭解大家遇到的問題，從而查漏補缺，激發更深入的思考。另一方面，期待你能慷慨地回答其他小夥伴的問題，分享你的見解，幫助他人進步。

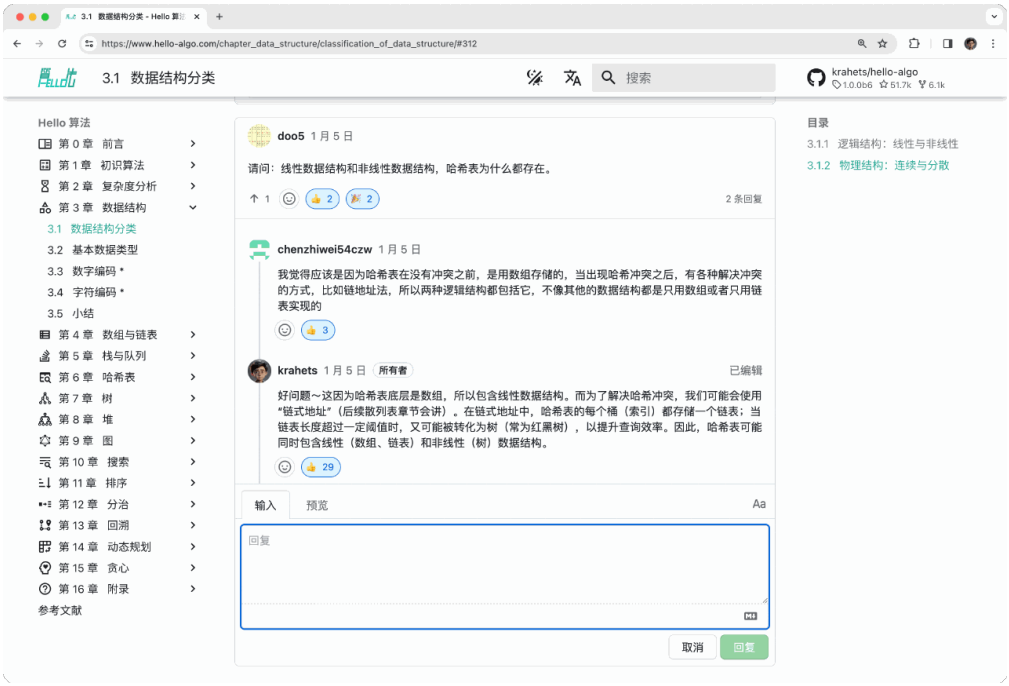


圖 0-7　評論區示例

0.2.5　演算法學習路線

從總體上看，我們可以將學習資料結構與演算法的過程劃分為三個階段。

1. **階段一：演算法入門**。我們需要熟悉各種資料結構的特點和用法，學習不同演算法的原理、流程、用途和效率等方面的內容。
2. **階段二：刷演算法題**。建議從熱門題目開刷，先積累至少 100 道題目，熟悉主流的演算法問題。初次刷題時，“知識遺忘”可能是一個挑戰，但請放心，這是很正常的。我們可以按照“艾賓浩斯遺忘曲線”來複習題目，通常在進行 3～5 輪的重複後，就能將其牢記在心。推薦的題單和刷題計劃請見此 GitHub 倉庫。
3. **階段三：搭建知識體系**。在學習方面，我們可以閱讀演算法專欄文章、解題框架和演算法教材，以不斷豐富知識體系。在刷題方面，可以嘗試採用進階刷題策略，如按專題分類、一題多解、一解多題等，相關的刷題心得可以在各個社群找到。

如圖 0-8 所示，本書內容主要涵蓋“階段一”，旨在幫助你更高效地展開階段二和階段三的學習。

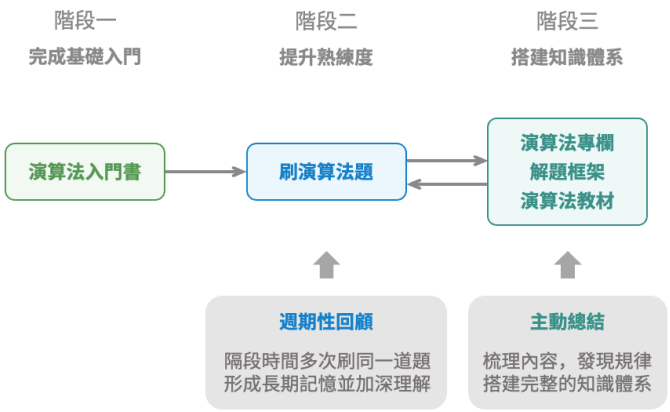


圖 0-8　演算法學習路線

0.3　小結

- 本書的主要受眾是演算法初學者。如果你已有一定基礎，本書能幫助你系統回顧演算法知識，書中源程式碼也可作為“刷題工具庫”使用。
- 書中內容主要包括複雜度分析、資料結構和演算法三部分，涵蓋了該領域的大部分主題。
- 對於演算法新手，在初學階段閱讀一本入門書至關重要，可以少走許多彎路。
- 書中的動畫圖解通常用於介紹重點和難點知識。閱讀本書時，應給予這些內容更多關注。
- 實踐乃學習程式設計之最佳途徑。強烈建議執行源程式碼並親自敲程式碼。
- 本書網頁版的每個章節都設有評論區，歡迎隨時分享你的疑惑與見解。

第 1 章　初識演算法

Abstract

一位少女翩翩起舞，與資料交織在一起，裙襬上飄揚著演算法的旋律。

她邀請你共舞，請緊跟她的步伐，踏入充滿邏輯與美感的演算法世界。

1.1　演算法無處不在

當我們聽到“演算法”這個詞時，很自然地會想到數學。然而實際上，許多演算法並不涉及複雜數學，而是更多地依賴基本邏輯，這些邏輯在我們的日常生活中處處可見。

在正式探討演算法之前，有一個有趣的事實值得分享：**你已經在不知不覺中學會了許多演算法，並習慣將它們應用到日常生活中了**。下面我將舉幾個具體的例子來證實這一點。

例一：查字典。在字典裡，每個漢字都對應一個拼音，而字典是按照拼音字母順序排列的。假設我們需要查詢一個拼音首字母為 r 的字，通常會按照圖 1-1 所示的方式實現。

1. 翻開字典約一半的頁數，檢視該頁的首字母是什麼，假設首字母為 m 。
2. 由於在拼音字母表中 r 位於 m 之後，所以排除字典前半部分，查詢範圍縮小到後半部分。
3. 不斷重複步驟 `1.` 和步驟 `2.` ，直至找到拼音首字母為 r 的頁碼為止。

圖 1-1　查字典步驟

查字典這個小學生必備技能，實際上就是著名的“二分搜尋”演算法。從資料結構的角度，我們可以把字典視為一個已排序的“陣列”；從演算法的角度，我們可以將上述查字典的一系列操作看作“二分搜尋”。

例二：整理撲克。我們在打牌時，每局都需要整理手中的撲克牌，使其從小到大排列，實現流程如圖 1-2 所示。

1. 將撲克牌劃分為“有序”和“無序”兩部分，並假設初始狀態下最左 1 張撲克牌已經有序。
2. 在無序部分抽出一張撲克牌，插入至有序部分的正確位置；完成後最左 2 張撲克已經有序。
3. 不斷迴圈步驟 `2.`，每一輪將一張撲克牌從無序部分插入至有序部分，直至所有撲克牌都有序。

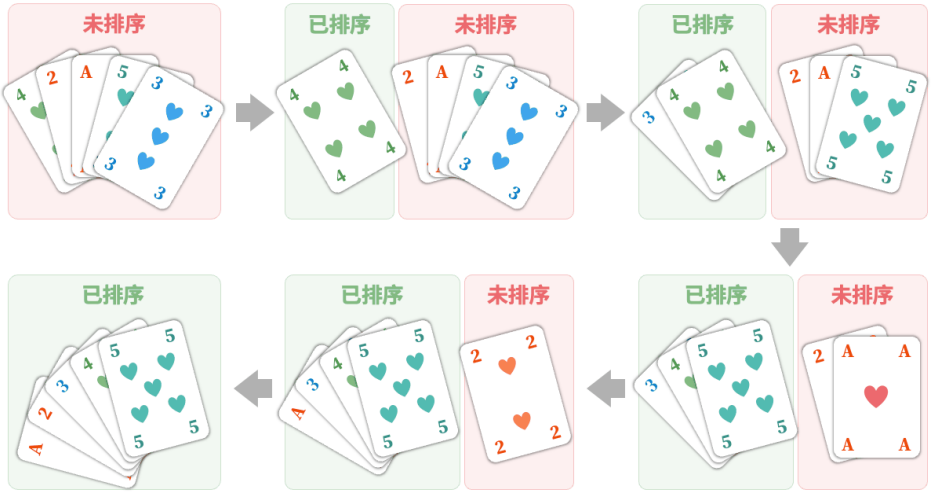


圖 1-2　撲克排序步驟

上述整理撲克牌的方法本質上是“插入排序”演算法，它在處理小型資料集時非常高效。許多程式語言的排序庫函式中都有插入排序的身影。

例三：貨幣找零。假設我們在超市購買了 69 元的商品，給了收銀員 100 元，則收銀員需要找我們 31 元。他會很自然地完成如圖 1-3 所示的思考。

1. 可選項是比 31 元面值更小的貨幣，包括 1 元、5 元、10 元、20 元。
2. 從可選項中拿出最大的 20 元，剩餘 $31 - 20 = 11$ 元。
3. 從剩餘可選項中拿出最大的 10 元，剩餘 $11 - 10 = 1$ 元。
4. 從剩餘可選項中拿出最大的 1 元，剩餘 $1 - 1 = 0$ 元。
5. 完成找零，方案為 $20 + 10 + 1 = 31$ 元。

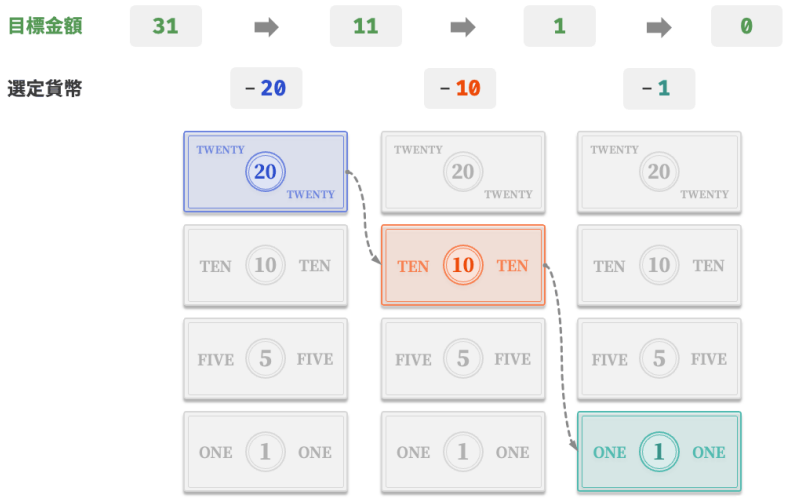


圖 1-3　貨幣找零過程

在以上步驟中，我們每一步都採取當前看來最好的選擇（儘可能用大面額的貨幣），最終得到了可行的找零方案。從資料結構與演算法的角度看，這種方法本質上是“貪婪”演算法。

小到烹飪一道菜，大到星際航行，幾乎所有問題的解決都離不開演算法。計算機的出現使得我們能夠透過程式設計將資料結構儲存在記憶體中，同時編寫程式碼呼叫 CPU 和 GPU 執行演算法。這樣一來，我們就能把生活中的問題轉移到計算機上，以更高效的方式解決各種複雜問題。

> **Tip**
> 如果你對資料結構、演算法、陣列和二分搜尋等概念仍感到一知半解，請繼續往下閱讀，本書將引導你邁入資料結構與演算法的知識殿堂。

1.2　演算法是什麼

1.2.1　演算法定義

演算法（algorithm）是在有限時間內解決特定問題的一組指令或操作步驟，它具有以下特性。

- 問題是明確的，包含清晰的輸入和輸出定義。
- 具有可行性，能夠在有限步驟、時間和記憶體空間下完成。
- 各步驟都有確定的含義，在相同的輸入和執行條件下，輸出始終相同。

1.2.2　資料結構定義

資料結構（data structure）是組織和儲存資料的方式，涵蓋資料內容、資料之間關係和資料操作方法，它具有以下設計目標。

- 空間佔用儘量少，以節省計算機記憶體。
- 資料操作儘可能快速，涵蓋資料訪問、新增、刪除、更新等。
- 提供簡潔的資料表示和邏輯資訊，以便演算法高效執行。

資料結構設計是一個充滿權衡的過程。如果想在某方面取得提升，往往需要在另一方面作出妥協。下面舉兩個例子。

- 鏈結串列相較於陣列，在資料新增和刪除操作上更加便捷，但犧牲了資料訪問速度。
- 圖相較於鏈結串列，提供了更豐富的邏輯資訊，但需要佔用更大的記憶體空間。

1.2.3　資料結構與演算法的關係

如圖 1-4 所示，資料結構與演算法高度相關、緊密結合，具體表現在以下三個方面。

- 資料結構是演算法的基石。資料結構為演算法提供了結構化儲存的資料，以及操作資料的方法。
- 演算法為資料結構注入生命力。資料結構本身僅儲存資料資訊，結合演算法才能解決特定問題。
- 演算法通常可以基於不同的資料結構實現，但執行效率可能相差很大，選擇合適的資料結構是關鍵。

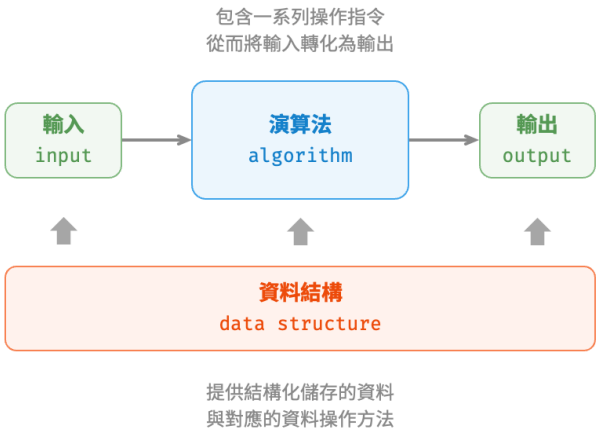


圖 1-4　資料結構與演算法的關係

資料結構與演算法猶如圖 1-5 所示的拼裝積木。一套積木，除了包含許多零件之外，還附有詳細的組裝說明書。我們按照說明書一步步操作，就能組裝出精美的積木模型。

圖 1-5　拼裝積木

兩者的詳細對應關係如表 1-1 所示。

表 1-1　將資料結構與演算法類比為拼裝積木

資料結構與演算法	拼裝積木
輸入資料	未拼裝的積木
資料結構	積木組織形式，包括形狀、大小、連線方式等
演算法	把積木拼成目標形態的一系列操作步驟
輸出資料	積木模型

值得說明的是，資料結構與演算法是獨立於程式語言的。正因如此，本書得以提供基於多種程式語言的實現。

約定俗成的簡稱
在實際討論時，我們通常會將“資料結構與演算法”簡稱為“演算法”。比如眾所周知的 LeetCode 演算法題目，實際上同時考查資料結構和演算法兩方面的知識。

1.3　小結

- 演算法在日常生活中無處不在，並不是遙不可及的高深知識。實際上，我們已經在不知不覺中學會了許多演算法，用以解決生活中的大小問題。
- 查字典的原理與二分搜尋演算法相一致。二分搜尋演算法體現了分而治之的重要演算法思想。

- 整理撲克的過程與插入排序演算法非常類似。插入排序演算法適合排序小型資料集。
- 貨幣找零的步驟本質上是貪婪演算法，每一步都採取當前看來最好的選擇。
- 演算法是在有限時間內解決特定問題的一組指令或操作步驟，而資料結構是計算機中組織和儲存資料的方式。
- 資料結構與演算法緊密相連。資料結構是演算法的基石，而演算法為資料結構注入生命力。
- 我們可以將資料結構與演算法類比為拼裝積木，積木代表資料，積木的形狀和連線方式等代表資料結構，拼裝積木的步驟則對應演算法。

1. Q & A

Q：作為一名程式設計師，我在日常工作中從未用演算法解決過問題，常用演算法都被程式語言封裝好了，直接用就可以了；這是否意味著我們工作中的問題還沒有到達需要演算法的程度？

如果把具體的工作技能比作是武功的“招式”的話，那麼基礎科目應該更像是“內功”。

我認為學演算法（以及其他基礎科目）的意義不是在於在工作中從零實現它，而是基於學到的知識，在解決問題時能夠作出專業的反應和判斷，從而提升工作的整體質量。舉一個簡單例子，每種程式語言都內建了排序函式：

- 如果我們沒有學過資料結構與演算法，那麼給定任何資料，我們可能都塞給這個排序函式去做了。執行順暢、效能不錯，看上去並沒有什麼問題。
- 但如果學過演算法，我們就會知道內建排序函式的時間複雜度是 $O(n \log n)$；而如果給定的資料是固定位數的整數（例如學號），那麼我們就可以用效率更高的“基數排序”來做，將時間複雜度降為 $O(nk)$，其中 k 為位數。當資料體量很大時，節省出來的執行時間就能創造較大價值（成本降低、體驗變好等）。

在工程領域中，大量問題是難以達到最優解的，許多問題只是被“差不多”地解決了。問題的難易程度一方面取決於問題本身的性質，另一方面也取決於觀測問題的人的知識儲備。人的知識越完備、經驗越多，分析問題就會越深入，問題就能被解決得更優雅。

第 2 章　複雜度分析

Abstract

複雜度分析猶如浩瀚的演算法宇宙中的時空嚮導。

它帶領我們在時間與空間這兩個維度上深入探索，尋找更優雅的解決方案。

2.1　演算法效率評估

在演算法設計中，我們先後追求以下兩個層面的目標。

1. **找到問題解法**：演算法需要在規定的輸入範圍內可靠地求得問題的正確解。
2. **尋求最優解法**：同一個問題可能存在多種解法，我們希望找到儘可能高效的演算法。

也就是說，在能夠解決問題的前提下，演算法效率已成為衡量演算法優劣的主要評價指標，它包括以下兩個維度。

- **時間效率**：演算法執行時間的長短。
- **空間效率**：演算法佔用記憶體空間的大小。

簡而言之，**我們的目標是設計“既快又省”的資料結構與演算法**。而有效地評估演算法效率至關重要，因為只有這樣，我們才能將各種演算法進行對比，進而指導演算法設計與最佳化過程。

效率評估方法主要分為兩種：實際測試、理論估算。

2.1.1　實際測試

假設我們現在有演算法 `A` 和演算法 `B` ，它們都能解決同一問題，現在需要對比這兩個演算法的效率。最直接的方法是找一臺計算機，執行這兩個演算法，並監控記錄它們的執行時間和記憶體佔用情況。這種評估方式能夠反映真實情況，但也存在較大的侷限性。

一方面，**難以排除測試環境的干擾因素**。硬體配置會影響演算法的效能表現。比如一個演算法的並行度較高，那麼它就更適合在多核 CPU 上執行，一個演算法的記憶體操作密集，那麼它在高效能記憶體上的表現就會更好。也就是說，演算法在不同的機器上的測試結果可能是不一致的。這意味著我們需要在各種機器上進行測試，統計平均效率，而這是不現實的。

另一方面，**展開完整測試非常耗費資源**。隨著輸入資料量的變化，演算法會表現出不同的效率。例如，在輸入資料量較小時，演算法 `A` 的執行時間比演算法 `B` 短；而在輸入資料量較大時，測試結果可能恰恰相反。因此，為了得到有說服力的結論，我們需要測試各種規模的輸入資料，而這需要耗費大量的計算資源。

2.1.2　理論估算

由於實際測試具有較大的侷限性，因此我們可以考慮僅透過一些計算來評估演算法的效率。這種估算方法被稱為**漸近複雜度分析**（asymptotic complexity analysis），簡稱**複雜度分析**。

複雜度分析能夠體現演算法執行所需的時間和空間資源與輸入資料大小之間的關係。**它描述了隨著輸入資料大小的增加，演算法執行所需時間和空間的增長趨勢**。這個定義有些拗口，我們可以將其分為三個重點來理解。

- “時間和空間資源”分別對應**時間複雜度**（time complexity）和**空間複雜度**（space complexity）。
- “隨著輸入資料大小的增加”意味著複雜度反映了演算法執行效率與輸入資料體量之間的關係。
- “時間和空間的增長趨勢”表示複雜度分析關注的不是執行時間或佔用空間的具體值，而是時間或空間增長的“快慢”。

複雜度分析克服了實際測試方法的弊端，體現在以下幾個方面。

- 它無需實際執行程式碼，更加綠色節能。
- 它獨立於測試環境，分析結果適用於所有執行平臺。
- 它可以體現不同資料量下的演算法效率，尤其是在大資料量下的演算法效能。

> **Tip**
> 如果你仍對複雜度的概念感到困惑，無須擔心，我們會在後續章節中詳細介紹。

複雜度分析為我們提供了一把評估演算法效率的“標尺”，使我們可以衡量執行某個演算法所需的時間和空間資源，對比不同演算法之間的效率。

複雜度是個數學概念，對於初學者可能比較抽象，學習難度相對較高。從這個角度看，複雜度分析可能不太適合作為最先介紹的內容。然而，當我們討論某個資料結構或演算法的特點時，難以避免要分析其執行速度和空間使用情況。

綜上所述，建議你在深入學習資料結構與演算法之前，**先對複雜度分析建立初步的瞭解，以便能夠完成簡單演算法的複雜度分析**。

2.2　迭代與遞迴

在演算法中，重複執行某個任務是很常見的，它與複雜度分析息息相關。因此，在介紹時間複雜度和空間複雜度之前，我們先來了解如何在程式中實現重複執行任務，即兩種基本的程式控制結構：迭代、遞迴。

2.2.1　迭代

迭代（iteration）是一種重複執行某個任務的控制結構。在迭代中，程式會在滿足一定的條件下重複執行某段程式碼，直到這個條件不再滿足。

1.　for 迴圈

`for` 迴圈是最常見的迭代形式之一，**適合在預先知道迭代次數時使用**。

以下函式基於 `for` 迴圈實現了求和 $1 + 2 + \cdots + n$ ，求和結果使用變數 `res` 記錄。需要注意的是，Python 中 `range(a, b)` 對應的區間是“左閉右開”的，對應的走訪範圍為 $a, a + 1, \ldots, b - 1$ ：

```rust
// === File: iteration.rs ===

/* for 迴圈 */
fn for_loop(n: i32) -> i32 {
    let mut res = 0;
    // 迴圈求和 1, 2, ..., n-1, n
    for i in 1..=n {
        res += i;
    }
    res
}
```

圖 2-1 是該求和函式的流程框圖。

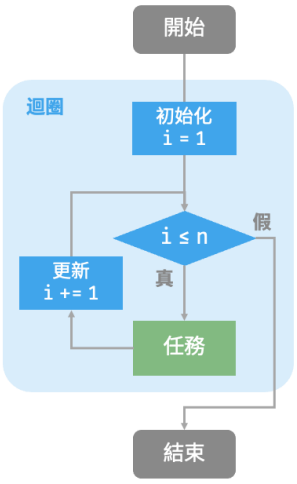


圖 2-1　求和函式的流程框圖

此求和函式的操作數量與輸入資料大小 n 成正比，或者說成“線性關係”。實際上，**時間複雜度描述的就是這個“線性關係”**。相關內容將會在下一節中詳細介紹。

2. while 迴圈

與 `for` 迴圈類似，`while` 迴圈也是一種實現迭代的方法。在 `while` 迴圈中，程式每輪都會先檢查條件，如果條件為真，則繼續執行，否則就結束迴圈。

下面我們用 `while` 迴圈來實現求和 $1 + 2 + \cdots + n$：

```rust
// === File: iteration.rs ===

/* while 迴圈 */
fn while_loop(n: i32) -> i32 {
    let mut res = 0;
    let mut i = 1; // 初始化條件變數

    // 迴圈求和 1, 2, ..., n-1, n
    while i <= n {
        res += i;
        i += 1; // 更新條件變數
    }
    res
}
```

`while` 迴圈比 `for` 迴圈的自由度更高。在 `while` 迴圈中，我們可以自由地設計條件變數的初始化和更新步驟。

例如在以下程式碼中，條件變數 i 每輪進行兩次更新，這種情況就不太方便用 `for` 迴圈實現：

```
// === File: iteration.rs ===

/* while 迴圈（兩次更新） */
fn while_loop_ii(n: i32) -> i32 {
    let mut res = 0;
    let mut i = 1; // 初始化條件變數

    // 迴圈求和 1, 4, 10, ...
    while i <= n {
        res += i;
        // 更新條件變數
        i += 1;
        i *= 2;
    }
    res
}
```

總的來說，**`for` 迴圈的程式碼更加緊湊，`while` 迴圈更加靈活**，兩者都可以實現迭代結構。選擇使用哪一個應該根據特定問題的需求來決定。

3. 巢狀迴圈

我們可以在一個迴圈結構內巢狀另一個迴圈結構，下面以 `for` 迴圈為例：

```
// === File: iteration.rs ===

/* 雙層 for 迴圈 */
fn nested_for_loop(n: i32) -> String {
    let mut res = vec![];
    // 迴圈 i = 1, 2, ..., n-1, n
    for i in 1..=n {
        // 迴圈 j = 1, 2, ..., n-1, n
        for j in 1..=n {
            res.push(format!("({}, {}), ", i, j));
        }
    }
    res.join("")
}
```

圖 2-2 是該巢狀迴圈的流程框圖。

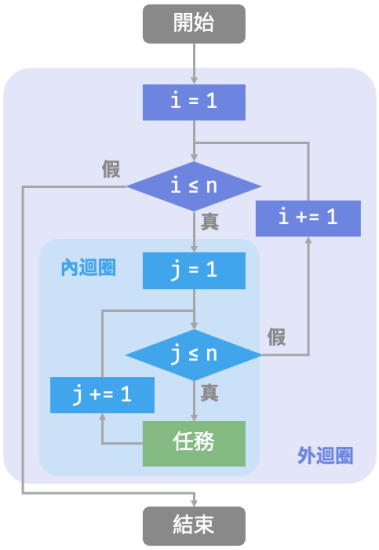


圖 2-2　巢狀迴圈的流程框圖

在這種情況下，函式的操作數量與 n^2 成正比，或者說演算法執行時間和輸入資料大小 n 成“平方關係”。

我們可以繼續新增巢狀迴圈，每一次巢狀都是一次“升維”，將會使時間複雜度提高至“立方關係”“四次方關係”，以此類推。

2.2.2　遞迴

遞迴（recursion）是一種演算法策略，透過函式呼叫自身來解決問題。它主要包含兩個階段。

1. **遞**：程式不斷深入地呼叫自身，通常傳入更小或更簡化的參數，直到達到“終止條件”。
2. **迴**：觸發“終止條件”後，程式從最深層的遞迴函式開始逐層返回，匯聚每一層的結果。

而從實現的角度看，遞迴程式碼主要包含三個要素。

1. **終止條件**：用於決定什麼時候由“遞”轉“迴”。
2. **遞迴呼叫**：對應“遞”，函式呼叫自身，通常輸入更小或更簡化的參數。
3. **返回結果**：對應“迴”，將當前遞迴層級的結果返回至上一層。

觀察以下程式碼，我們只需呼叫函式 `recur(n)`，就可以完成 $1 + 2 + \cdots + n$ 的計算：

```rust
// === File: recursion.rs ===

/* 遞迴 */
fn recur(n: i32) -> i32 {
    // 終止條件
    if n == 1 {
        return 1;
    }
    // 遞：遞迴呼叫
```

```
    let res = recur(n - 1);
    // 迴：返回結果
    n + res
}
```

圖 2-3 展示了該函式的遞迴過程。

圖 2-3　求和函式的遞迴過程

雖然從計算角度看，迭代與遞迴可以得到相同的結果，**但它們代表了兩種完全不同的思考和解決問題的範式**。

- **迭代**：“自下而上”地解決問題。從最基礎的步驟開始，然後不斷重複或累加這些步驟，直到任務完成。
- **遞迴**：“自上而下”地解決問題。將原問題分解為更小的子問題，這些子問題和原問題具有相同的形式。接下來將子問題繼續分解為更小的子問題，直到基本情況時停止（基本情況的解是已知的）。

以上述求和函式為例，設問題 $f(n) = 1 + 2 + \cdots + n$ 。

- **迭代**：在迴圈中模擬求和過程，從 1 走訪到 n ，每輪執行求和操作，即可求得 $f(n)$ 。
- **遞迴**：將問題分解為子問題 $f(n) = n + f(n-1)$ ，不斷（遞迴地）分解下去，直至基本情況 $f(1) = 1$ 時終止。

1. 呼叫堆疊

遞迴函式每次呼叫自身時，系統都會為新開啟的函式分配記憶體，以儲存區域性變數、呼叫位址和其他資訊等。這將導致兩方面的結果。

- 函式的上下文資料都儲存在稱為“堆疊幀空間”的記憶體區域中，直至函式返回後才會被釋放。因此，**遞迴通常比迭代更加耗費記憶體空間**。
- 遞迴呼叫函式會產生額外的開銷。**因此遞迴通常比迴圈的時間效率更低**。

如圖 2-4 所示，在觸發終止條件前，同時存在 n 個未返回的遞迴函式，**遞迴深度為 n**。

圖 2-4　遞迴呼叫深度

在實際中，程式語言允許的遞迴深度通常是有限的，過深的遞迴可能導致堆疊溢位錯誤。

2. 尾遞迴

有趣的是，**如果函式在返回前的最後一步才進行遞迴呼叫**，則該函式可以被編譯器或直譯器最佳化，使其在空間效率上與迭代相當。這種情況被稱為尾遞迴（tail recursion）。

- **普通遞迴**：當函式返回到上一層級的函式後，需要繼續執行程式碼，因此系統需要儲存上一層呼叫的上下文。
- **尾遞迴**：遞迴呼叫是函式返回前的最後一個操作，這意味著函式返回到上一層級後，無須繼續執行其他操作，因此系統無須儲存上一層函式的上下文。

以計算 $1 + 2 + \cdots + n$ 為例，我們可以將結果變數 `res` 設為函式參數，從而實現尾遞迴：

```
// === File: recursion.rs ===

/* 尾遞迴 */
fn tail_recur(n: i32, res: i32) -> i32 {
    // 終止條件
    if n == 0 {
        return res;
    }
    // 尾遞迴呼叫
    tail_recur(n - 1, res + n)
}
```

尾遞迴的執行過程如圖 2-5 所示。對比普通遞迴和尾遞迴，兩者的求和操作的執行點是不同的。

- **普通遞迴**：求和操作是在“迴”的過程中執行的，每層返回後都要再執行一次求和操作。
- **尾遞迴**：求和操作是在“遞”的過程中執行的，“迴”的過程只需層層返回。

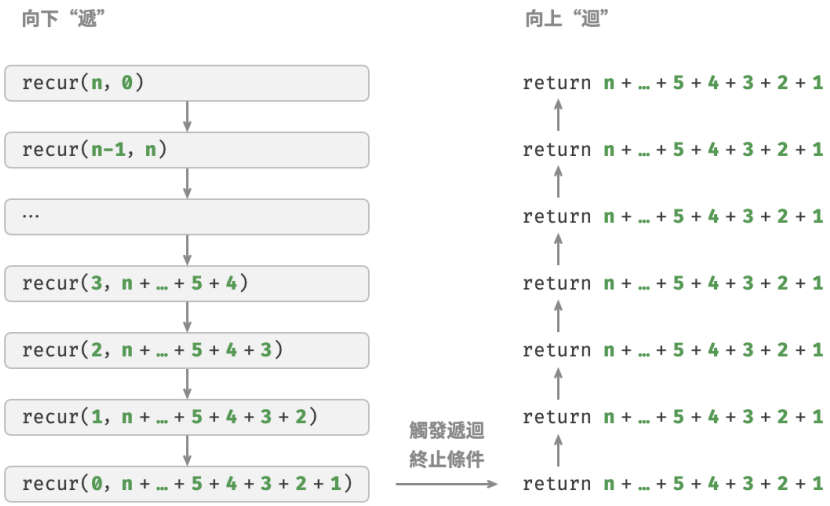


圖 2-5　尾遞迴過程

> **Tip**
> 請注意，許多編譯器或直譯器並不支持尾遞迴最佳化。例如，Python 預設不支持尾遞迴最佳化，因此即使函式是尾遞迴形式，仍然可能會遇到堆疊溢位問題。

3. 遞迴樹

當處理與“分治”相關的演算法問題時，遞迴往往比迭代的思路更加直觀、程式碼更加易讀。以“費波那契數列”為例。

> **Question**
> 給定一個費波那契數列 $0, 1, 1, 2, 3, 5, 8, 13, \ldots$ ，求該數列的第 n 個數字。

設費波那契數列的第 n 個數字為 $f(n)$ ，易得兩個結論。

- 數列的前兩個數字為 $f(1) = 0$ 和 $f(2) = 1$ 。
- 數列中的每個數字是前兩個數字的和，即 $f(n) = f(n-1) + f(n-2)$ 。

按照遞推關係進行遞迴呼叫，將前兩個數字作為終止條件，便可寫出遞迴程式碼。呼叫 `fib(n)` 即可得到費波那契數列的第 n 個數字：

```rust
// === File: recursion.rs ===

/* 費波那契數列：遞迴 */
fn fib(n: i32) -> i32 {
    // 終止條件 f(1) = 0, f(2) = 1
    if n == 1 || n == 2 {
        return n - 1;
    }
    // 遞迴呼叫 f(n) = f(n-1) + f(n-2)
    let res = fib(n - 1) + fib(n - 2);
    // 返回結果
    res
}
```

觀察以上程式碼，我們在函式內遞迴呼叫了兩個函式，**這意味著從一個呼叫產生了兩個呼叫分支**。如圖 2-6 所示，這樣不斷遞迴呼叫下去，最終將產生一棵層數為 n 的遞迴樹（recursion tree）。

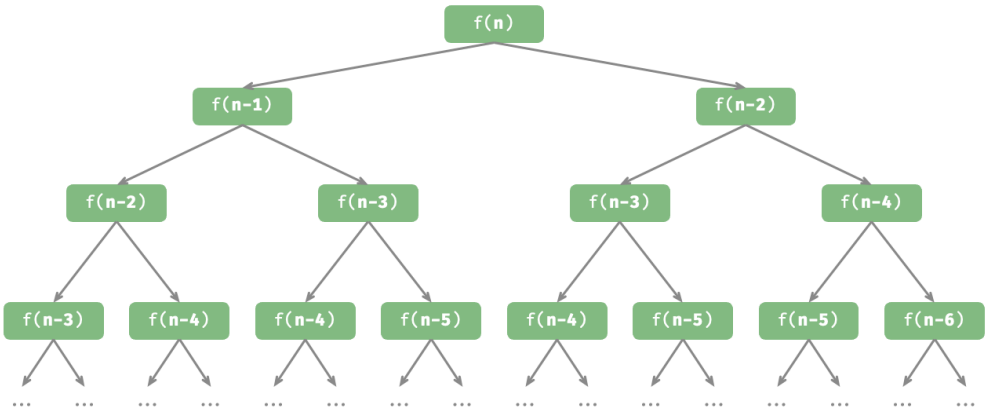


圖 2-6　費波那契數列的遞迴樹

從本質上看，遞迴體現了“將問題分解為更小子問題”的思維範式，這種分治策略至關重要。

- 從演算法角度看，搜尋、排序、回溯、分治、動態規劃等許多重要演算法策略直接或間接地應用了這種思維方式。
- 從資料結構角度看，遞迴天然適合處理鏈結串列、樹和圖的相關問題，因為它們非常適合用分治思想進行分析。

2.2.3　兩者對比

總結以上內容，如表 2-1 所示，迭代和遞迴在實現、效能和適用性上有所不同。

表 2-1　迭代與遞迴特點對比

	迭代	遞迴
實現方式	迴圈結構	函式呼叫自身
時間效率	效率通常較高，無函式呼叫開銷	每次函式呼叫都會產生開銷
記憶體使用	通常使用固定大小的記憶體空間	累積函式呼叫可能使用大量的堆疊幀空間
適用問題	適用於簡單迴圈任務，程式碼直觀、可讀性好	適用於子問題分解，如樹、圖、分治、回溯等，程式碼結構簡潔、清晰

> **Tip**
> 如果感覺以下內容理解困難，可以在讀完“堆疊”章節後再來複習。

那麼，迭代和遞迴具有什麼內在關聯呢？以上述遞迴函式為例，求和操作在遞迴的“迴”階段進行。這意味著最初被呼叫的函式實際上是最後完成其求和操作的，**這種工作機制與堆疊的“先入後出”原則異曲同工**。

事實上，“呼叫堆疊”和“堆疊幀空間”這類遞迴術語已經暗示了遞迴與堆疊之間的密切關係。

1. **遞**：當函式被呼叫時，系統會在“呼叫堆疊”上為該函式分配新的堆疊幀，用於儲存函式的區域性變數、參數、返回位址等資料。
2. **迴**：當函式完成執行並返回時，對應的堆疊幀會被從“呼叫堆疊”上移除，恢復之前函式的執行環境。

因此，**我們可以使用一個顯式的堆疊來模擬呼叫堆疊的行為**，從而將遞迴轉化為迭代形式：

```rust
// === File: recursion.rs ===

/* 使用迭代模擬遞迴 */
fn for_loop_recur(n: i32) -> i32 {
    // 使用一個顯式的堆疊來模擬系統呼叫堆疊
    let mut stack = Vec::new();
    let mut res = 0;
    // 遞：遞迴呼叫
    for i in (1..=n).rev() {
        // 透過“入堆疊操作”模擬“遞”
        stack.push(i);
    }
    // 迴：返回結果
    while !stack.is_empty() {
        // 透過“出堆疊操作”模擬“迴”
        res += stack.pop().unwrap();
    }
    // res = 1+2+3+...+n
    res
}
```

觀察以上程式碼，當遞迴轉化為迭代後，程式碼變得更加複雜了。儘管迭代和遞迴在很多情況下可以互相轉化，但不一定值得這樣做，有以下兩點原因。

- 轉化後的程式碼可能更加難以理解，可讀性更差。
- 對於某些複雜問題，模擬系統呼叫堆疊的行為可能非常困難。

總之，**選擇迭代還是遞迴取決於特定問題的性質**。在程式設計實踐中，權衡兩者的優劣並根據情境選擇合適的方法至關重要。

2.3　時間複雜度

執行時間可以直觀且準確地反映演算法的效率。如果我們想準確預估一段程式碼的執行時間，應該如何操作呢？

1. **確定執行平臺**，包括硬體配置、程式語言、系統環境等，這些因素都會影響程式碼的執行效率。
2. **評估各種計算操作所需的執行時間**，例如加法操作 `+` 需要 1 ns ，乘法操作 `*` 需要 10 ns ，列印操作 `print()` 需要 5 ns 等。
3. **統計程式碼中所有的計算操作**，並將所有操作的執行時間求和，從而得到執行時間。

例如在以下程式碼中，輸入資料大小為 n ：

```
// 在某執行平臺下
fn algorithm(n: i32) {
    let mut a = 2;      // 1 ns
    a = a + 1;          // 1 ns
    a = a * 2;          // 10 ns
    // 迴圈 n 次
    for _ in 0..n {     // 1 ns
        println!("{}", 0);  // 5 ns
    }
}
```

根據以上方法，可以得到演算法的執行時間為 $(6n + 12)$ ns ：

$$1 + 1 + 10 + (1 + 5) \times n = 6n + 12$$

但實際上，**統計演算法的執行時間既不合理也不現實**。首先，我們不希望將預估時間和執行平臺繫結，因為演算法需要在各種不同的平臺上執行。其次，我們很難獲知每種操作的執行時間，這給預估過程帶來了極大的難度。

2.3.1　統計時間增長趨勢

時間複雜度分析統計的不是演算法執行時間，**而是演算法執行時間隨著資料量變大時的增長趨勢**。

“時間增長趨勢”這個概念比較抽象，我們透過一個例子來加以理解。假設輸入資料大小為 n ，給定三個演算法 `A`、`B` 和 `C` ：

```rust
// 演算法 A 的時間複雜度：常數階
fn algorithm_A(n: i32) {
    println!("{}", 0);
}
// 演算法 B 的時間複雜度：線性階
fn algorithm_B(n: i32) {
    for _ in 0..n {
        println!("{}", 0);
    }
}
// 演算法 C 的時間複雜度：常數階
fn algorithm_C(n: i32) {
    for _ in 0..1000000 {
        println!("{}", 0);
    }
}
```

圖 2-7 展示了以上三個演算法函式的時間複雜度。

- 演算法 `A` 只有 1 個列印操作，演算法執行時間不隨著 n 增大而增長。我們稱此演算法的時間複雜度為“常數階”。
- 演算法 `B` 中的列印操作需要迴圈 n 次，演算法執行時間隨著 n 增大呈線性增長。此演算法的時間複雜度被稱為“線性階”。
- 演算法 `C` 中的列印操作需要迴圈 1000000 次，雖然執行時間很長，但它與輸入資料大小 n 無關。因此 `C` 的時間複雜度和 `A` 相同，仍為“常數階”。

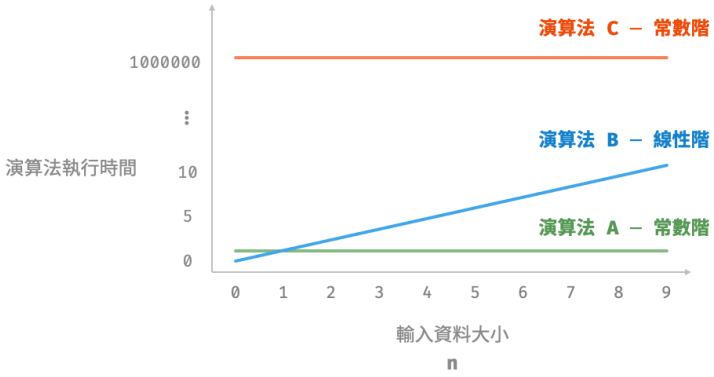


圖 2-7　演算法 A、B 和 C 的時間增長趨勢

相較於直接統計演算法的執行時間，時間複雜度分析有哪些特點呢？

- **時間複雜度能夠有效評估演算法效率**。例如，演算法 `B` 的執行時間呈線性增長，在 $n > 1$ 時比演算法 `A` 更慢，在 $n > 1000000$ 時比演算法 `C` 更慢。事實上，只要輸入資料大小 n 足夠大，複雜度為“常數階”的演算法一定優於“線性階”的演算法，這正是時間增長趨勢的含義。

- **時間複雜度的推算方法更簡便**。顯然，執行平臺和計算操作型別都與演算法執行時間的增長趨勢無關。因此在時間複雜度分析中，我們可以簡單地將所有計算操作的執行時間視為相同的“單位時間”，從而將“計算操作執行時間統計”簡化為“計算操作數量統計”，這樣一來估算難度就大大降低了。
- **時間複雜度也存在一定的侷限性**。例如，儘管演算法 `A` 和 `C` 的時間複雜度相同，但實際執行時間差別很大。同樣，儘管演算法 `B` 的時間複雜度比 `C` 高，但在輸入資料大小 n 較小時，演算法 `B` 明顯優於演算法 `C` 。對於此類情況，我們時常難以僅憑時間複雜度判斷演算法效率的高低。當然，儘管存在上述問題，複雜度分析仍然是評判演算法效率最有效且常用的方法。

2.3.2　函式漸近上界

給定一個輸入大小為 n 的函式：

```rust
fn algorithm(n: i32) {
    let mut a = 1;   // +1
    a = a + 1;       // +1
    a = a * 2;       // +1

    // 迴圈 n 次
    for _ in 0..n { // +1（每輪都執行 i ++）
        println!("{}", 0); // +1
    }
}
```

設演算法的操作數量是一個關於輸入資料大小 n 的函式，記為 $T(n)$ ，則以上函式的操作數量為：

$$T(n) = 3 + 2n$$

$T(n)$ 是一次函式，說明其執行時間的增長趨勢是線性的，因此它的時間複雜度是線性階。

我們將線性階的時間複雜度記為 $O(n)$ ，這個數學符號稱為**大 O 記號**（big-O notation），表示函式 $T(n)$ 的**漸近上界**（asymptotic upper bound）。

時間複雜度分析本質上是計算“操作數量 $T(n)$”的漸近上界，它具有明確的數學定義。

> **函式漸近上界**
> 若存在正實數 c 和實數 n_0 ，使得對於所有的 $n > n_0$ ，均有 $T(n) \leq c \cdot f(n)$ ，則可認為 $f(n)$ 給出了 $T(n)$ 的一個漸近上界，記為 $T(n) = O(f(n))$ 。

如圖 2-8 所示，計算漸近上界就是尋找一個函式 $f(n)$ ，使得當 n 趨向於無窮大時，$T(n)$ 和 $f(n)$ 處於相同的增長級別，僅相差一個常數項 c 的倍數。

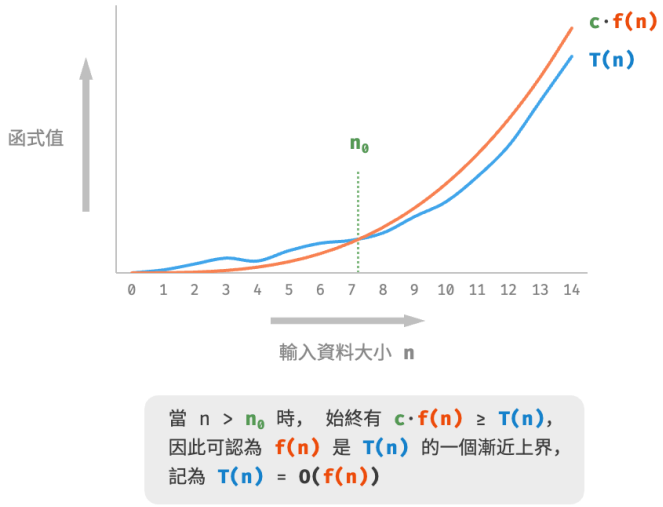


圖 2-8　函式的漸近上界

2.3.3　推算方法

漸近上界的數學味兒有點重，如果你感覺沒有完全理解，也無須擔心。我們可以先掌握推算方法，在不斷的實踐中，就可以逐漸領悟其數學意義。

根據定義，確定 $f(n)$ 之後，我們便可得到時間複雜度 $O(f(n))$ 。那麼如何確定漸近上界 $f(n)$ 呢？總體分為兩步：首先統計操作數量，然後判斷漸近上界。

1.　第一步：統計操作數量

針對程式碼，逐行從上到下計算即可。然而，由於上述 $c \cdot f(n)$ 中的常數項 c 可以取任意大小，**因此操作數量 $T(n)$ 中的各種係數、常數項都可以忽略**。根據此原則，可以總結出以下計數簡化技巧。

1. **忽略 $T(n)$ 中的常數項**。因為它們都與 n 無關，所以對時間複雜度不產生影響。
2. **省略所有係數**。例如，迴圈 $2n$ 次、$5n + 1$ 次等，都可以簡化記為 n 次，因為 n 前面的係數對時間複雜度沒有影響。
3. **迴圈巢狀時使用乘法**。總操作數量等於外層迴圈和內層迴圈操作數量之積，每一層迴圈依然可以分別套用第 `1.` 點和第 `2.` 點的技巧。

給定一個函式，我們可以用上述技巧來統計操作數量：

```
fn algorithm(n: i32) {
    let mut a = 1;      // +0（技巧 1）
    a = a + n;          // +0（技巧 1）

    // +n（技巧 2）
    for i in 0..(5 * n + 1) {
```

```
        println!("{}", 0);
    }

    // +n*n（技巧 3）
    for i in 0..(2 * n) {
        for j in 0..(n + 1) {
            println!("{}", 0);
        }
    }
}
```

以下公式展示了使用上述技巧前後的統計結果，兩者推算出的時間複雜度都為 $O(n^2)$ 。

$$
\begin{aligned}
T(n) &= 2n(n+1) + (5n+1) + 2 \quad \text{完整統計 (-.-|||)} \\
&= 2n^2 + 7n + 3 \\
T(n) &= n^2 + n \quad \text{偷懶統計 (o.O)}
\end{aligned}
$$

2. 第二步：判斷漸近上界

時間複雜度由 $T(n)$ 中最高階的項來決定。這是因為在 n 趨於無窮大時，最高階的項將發揮主導作用，其他項的影響都可以忽略。

表 2-2 展示了一些例子，其中一些誇張的值是為了強調“係數無法撼動階數”這一結論。當 n 趨於無窮大時，這些常數變得無足輕重。

表 2-2　不同操作數量對應的時間複雜度

操作數量 $T(n)$	時間複雜度 $O(f(n))$
100000	$O(1)$
$3n + 2$	$O(n)$
$2n^2 + 3n + 2$	$O(n^2)$
$n^3 + 10000n^2$	$O(n^3)$
$2^n + 10000n^{10000}$	$O(2^n)$

2.3.4　常見型別

設輸入資料大小為 n ，常見的時間複雜度型別如圖 2-9 所示（按照從低到高的順序排列）。

$$
\underbrace{O(1)}_{\text{常數階}} < \underbrace{O(\log n)}_{\text{對數階}} < \underbrace{O(n)}_{\text{線性階}} < \underbrace{O(n \log n)}_{\text{線性對數階}} < \underbrace{O(n^2)}_{\text{平方階}} < \underbrace{O(2^n)}_{\text{指數階}} < \underbrace{O(n!)}_{\text{階乘階}}
$$

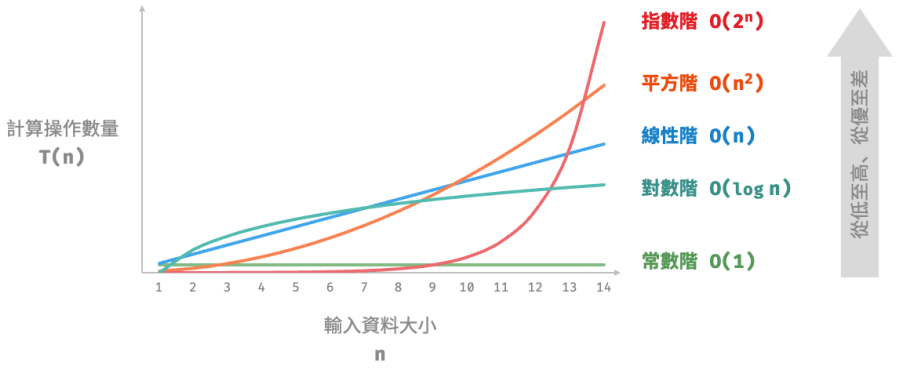


圖 2-9　常見的時間複雜度型別

1. 常數階 $O(1)$

常數階的操作數量與輸入資料大小 n 無關，即不隨著 n 的變化而變化。

在以下函式中，儘管操作數量 `size` 可能很大，但由於其與輸入資料大小 n 無關，因此時間複雜度仍為 $O(1)$ ：

```
// === File: time_complexity.rs ===

/* 常數階 */
fn constant(n: i32) -> i32 {
    _ = n;
    let mut count = 0;
    let size = 100_000;
    for _ in 0..size {
        count += 1;
    }
    count
}
```

2. 線性階 $O(n)$

線性階的操作數量相對於輸入資料大小 n 以線性級別增長。線性階通常出現在單層迴圈中：

```
// === File: time_complexity.rs ===

/* 線性階 */
fn linear(n: i32) -> i32 {
    let mut count = 0;
    for _ in 0..n {
```

```
        count += 1;
    }
    count
}
```

走訪陣列和走訪鏈結串列等操作的時間複雜度均為 $O(n)$，其中 n 為陣列或鏈結串列的長度：

```
// === File: time_complexity.rs ===

/* 線性階（走訪陣列） */
fn array_traversal(nums: &[i32]) -> i32 {
    let mut count = 0;
    // 迴圈次數與陣列長度成正比
    for _ in nums {
        count += 1;
    }
    count
}
```

值得注意的是，**輸入資料大小 n 需根據輸入資料的型別來具體確定**。比如在第一個示例中，變數 n 為輸入資料大小；在第二個示例中，陣列長度 n 為資料大小。

3. 平方階 $O(n^2)$

平方階的操作數量相對於輸入資料大小 n 以平方級別增長。平方階通常出現在巢狀迴圈中，外層迴圈和內層迴圈的時間複雜度都為 $O(n)$，因此總體的時間複雜度為 $O(n^2)$：

```
// === File: time_complexity.rs ===

/* 平方階 */
fn quadratic(n: i32) -> i32 {
    let mut count = 0;
    // 迴圈次數與資料大小 n 成平方關係
    for _ in 0..n {
        for _ in 0..n {
            count += 1;
        }
    }
    count
}
```

圖 2-10 對比了常數階、線性階和平方階三種時間複雜度。

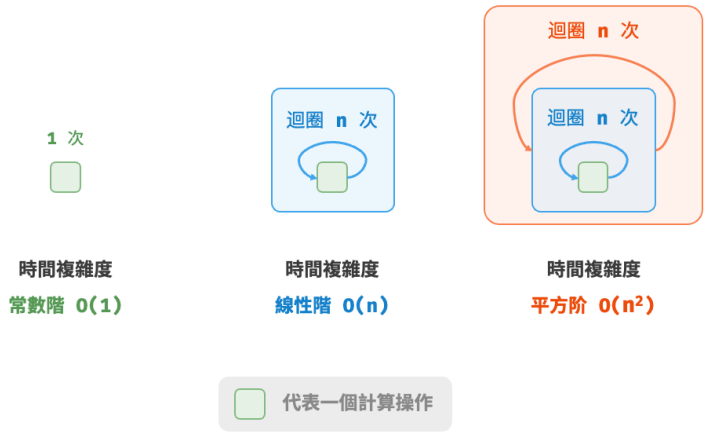


圖 2-10　常數階、線性階和平方階的時間複雜度

以泡沫排序為例，外層迴圈執行 $n-1$ 次，內層迴圈執行 $n-1$、$n-2$、...、2、1 次，平均為 $n/2$ 次，因此時間複雜度為 $O((n-1)n/2) = O(n^2)$ ：

```
// === File: time_complexity.rs ===

/* 平方階（泡沫排序） */
fn bubble_sort(nums: &mut [i32]) -> i32 {
    let mut count = 0; // 計數器

    // 外迴圈：未排序區間為 [0, i]
    for i in (1..nums.len()).rev() {
        // 內迴圈：將未排序區間 [0, i] 中的最大元素交換至該區間的最右端
        for j in 0..i {
            if nums[j] > nums[j + 1] {
                // 交換 nums[j] 與 nums[j + 1]
                let tmp = nums[j];
                nums[j] = nums[j + 1];
                nums[j + 1] = tmp;
                count += 3; // 元素交換包含 3 個單元操作
            }
        }
    }
    count
}
```

4. 指數階 $O(2^n)$

生物學的“細胞分裂”是指數階增長的典型例子：初始狀態為 1 個細胞，分裂一輪後變為 2 個，分裂兩輪後變為 4 個，以此類推，分裂 n 輪後有 2^n 個細胞。

圖 2-11 和以下程式碼模擬了細胞分裂的過程，時間複雜度為 $O(2^n)$ 。請注意，輸入 n 表示分裂輪數，返回值 `count` 表示總分裂次數。

```
// === File: time_complexity.rs ===

/* 指數階（迴圈實現） */
fn exponential(n: i32) -> i32 {
    let mut count = 0;
    let mut base = 1;
    // 細胞每輪一分為二，形成數列 1, 2, 4, 8, ..., 2^(n-1)
    for _ in 0..n {
        for _ in 0..base {
            count += 1
        }
        base *= 2;
    }
    // count = 1 + 2 + 4 + 8 + .. + 2^(n-1) = 2^n - 1
    count
}
```

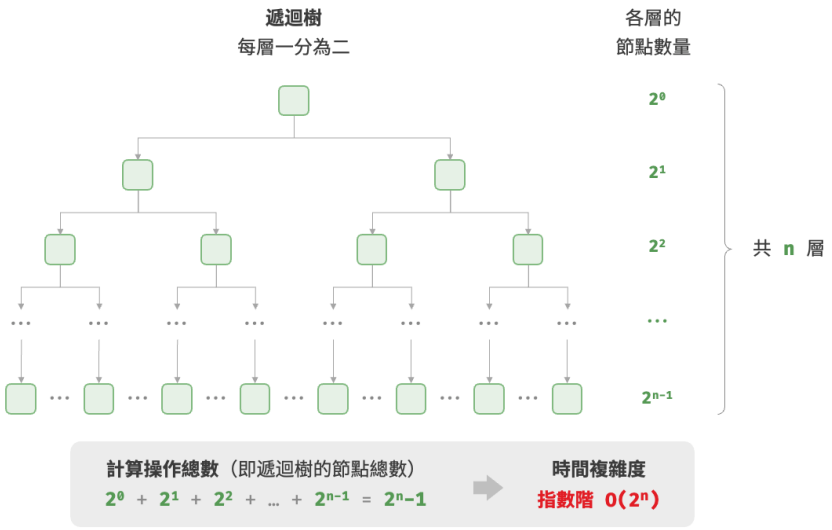


圖 2-11　指數階的時間複雜度

在實際演算法中，指數階常出現於遞迴函式中。例如在以下程式碼中，其遞迴地一分為二，經過 n 次分裂後停止：

```
// === File: time_complexity.rs ===

/* 指數階（遞迴實現） */
fn exp_recur(n: i32) -> i32 {
    if n == 1 {
```

```
        return 1;
    }
    exp_recur(n - 1) + exp_recur(n - 1) + 1
}
```

指數階增長非常迅速，在窮舉法（暴力搜尋、回溯等）中比較常見。對於資料規模較大的問題，指數階是不可接受的，通常需要使用動態規劃或貪婪演算法等來解決。

5. 對數階 $O(\log n)$

與指數階相反，對數階反映了“每輪縮減到一半”的情況。設輸入資料大小為 n，由於每輪縮減到一半，因此迴圈次數是 $\log_2 n$，即 2^n 的反函式。

圖 2-12 和以下程式碼模擬了“每輪縮減到一半”的過程，時間複雜度為 $O(\log_2 n)$，簡記為 $O(\log n)$：

```
// === File: time_complexity.rs ===

/* 對數階（迴圈實現） */
fn logarithmic(mut n: i32) -> i32 {
    let mut count = 0;
    while n > 1 {
        n = n / 2;
        count += 1;
    }
    count
}
```

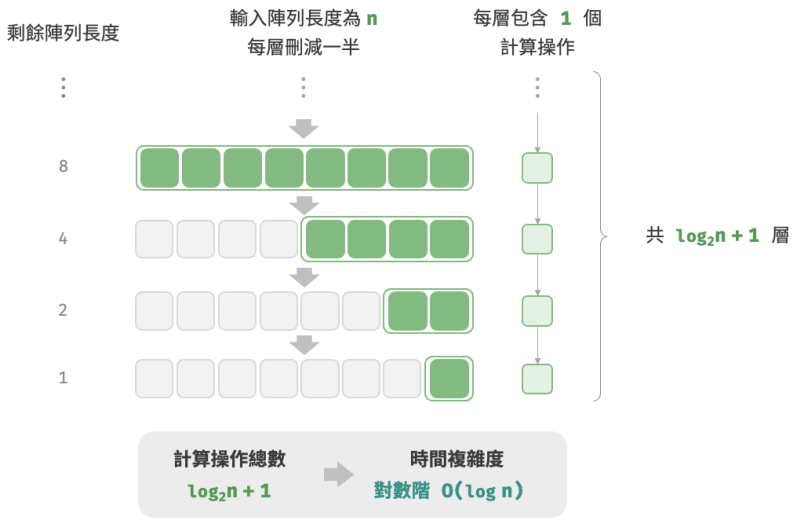


圖 2-12　對數階的時間複雜度

與指數階類似，對數階也常出現於遞迴函式中。以下程式碼形成了一棵高度為 $\log_2 n$ 的遞迴樹：

```
// === File: time_complexity.rs ===

/* 對數階（遞迴實現） */
fn log_recur(n: i32) -> i32 {
    if n <= 1 {
        return 0;
    }
    log_recur(n / 2) + 1
}
```

對數階常出現於基於分治策略的演算法中，體現了“一分為多”和“化繁為簡”的演算法思想。它增長緩慢，是僅次於常數階的理想的時間複雜度。

> **$O(\log n)$ 的底數是多少？**
>
> 準確來說，“一分為 m”對應的時間複雜度是 $O(\log_m n)$ 。而透過對數換底公式，我們可以得到具有不同底數、相等的時間複雜度：
>
> $$O(\log_m n) = O(\log_k n / \log_k m) = O(\log_k n)$$
>
> 也就是說，底數 m 可以在不影響複雜度的前提下轉換。因此我們通常會省略底數 m ，將對數階直接記為 $O(\log n)$ 。

6. 線性對數階 $O(n \log n)$

線性對數階常出現於巢狀迴圈中，兩層迴圈的時間複雜度分別為 $O(\log n)$ 和 $O(n)$ 。相關程式碼如下：

```
// === File: time_complexity.rs ===

/* 線性對數階 */
fn linear_log_recur(n: i32) -> i32 {
    if n <= 1 {
        return 1;
    }
    let mut count = linear_log_recur(n / 2) + linear_log_recur(n / 2);
    for _ in 0..n {
        count += 1;
    }
    return count;
}
```

圖 2-13 展示了線性對數階的生成方式。二元樹的每一層的操作總數都為 n ，樹共有 $\log_2 n + 1$ 層，因此時間複雜度為 $O(n \log n)$ 。

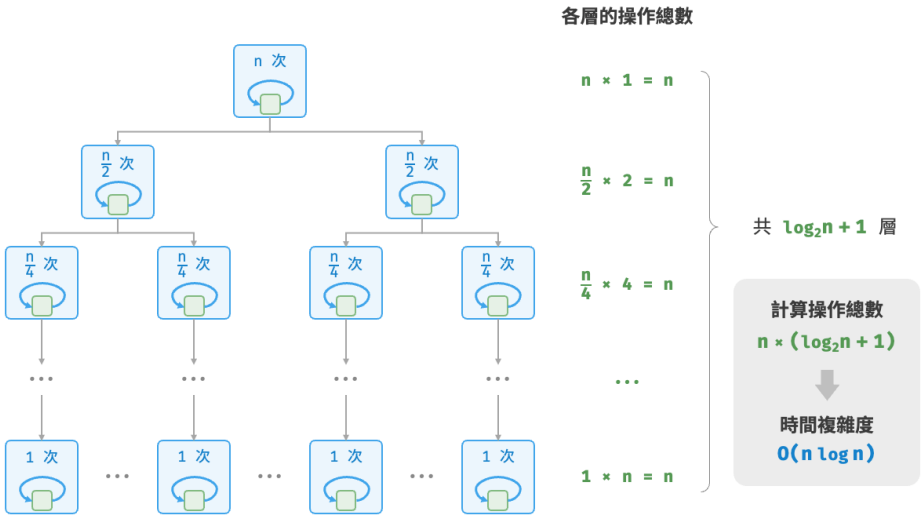


圖 2-13　線性對數階的時間複雜度

主流排序演算法的時間複雜度通常為 $O(n \log n)$，例如快速排序、合併排序、堆積排序等。

7. 階乘階 $O(n!)$

階乘階對應數學上的“全排列”問題。給定 n 個互不重複的元素，求其所有可能的排列方案，方案數量為：

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$$

階乘通常使用遞迴實現。如圖 2-14 和以下程式碼所示，第一層分裂出 n 個，第二層分裂出 $n-1$ 個，以此類推，直至第 n 層時停止分裂：

```rust
// === File: time_complexity.rs ===

/* 階乘階（遞迴實現） */
fn factorial_recur(n: i32) -> i32 {
    if n == 0 {
        return 1;
    }
    let mut count = 0;
    // 從 1 個分裂出 n 個
    for _ in 0..n {
        count += factorial_recur(n - 1);
    }
    count
}
```

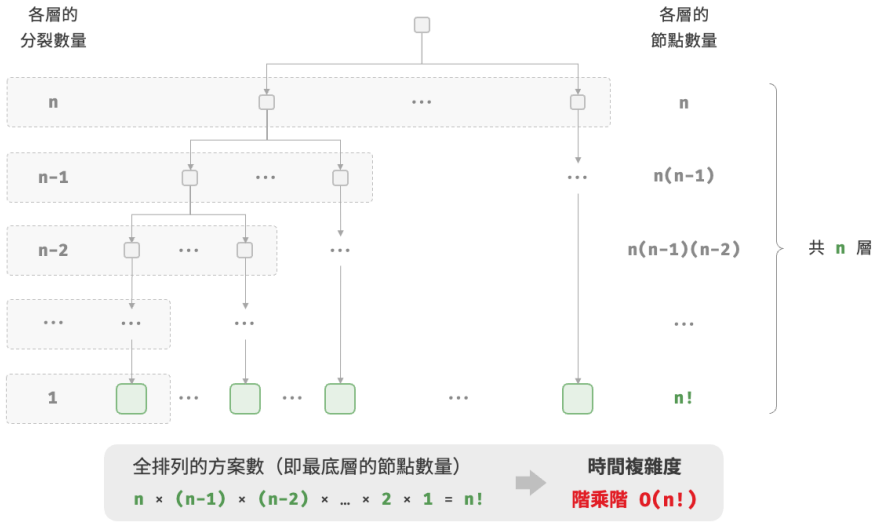


圖 2-14　階乘階的時間複雜度

請注意，因為當 $n \geq 4$ 時恆有 $n! > 2^n$ ，所以階乘階比指數階增長得更快，在 n 較大時也是不可接受的。

2.3.5　最差、最佳、平均時間複雜度

演算法的時間效率往往不是固定的，而是與輸入資料的分佈有關。假設輸入一個長度為 n 的陣列 `nums` ，其中 `nums` 由從 1 至 n 的數字組成，每個數字只出現一次；但元素順序是隨機打亂的，任務目標是返回元素 1 的索引。我們可以得出以下結論。

- 當 `nums = [?, ?, ..., 1]` ，即當末尾元素是 1 時，需要完整走訪陣列，**達到最差時間複雜度** $O(n)$ 。
- 當 `nums = [1, ?, ?, ...]` ，即當首個元素為 1 時，無論陣列多長都不需要繼續走訪，**達到最佳時間複雜度** $\Omega(1)$ 。

“最差時間複雜度”對應函式漸近上界，使用大 O 記號表示。相應地，“最佳時間複雜度”對應函式漸近下界，用 Ω 記號表示：

```
// === File: worst_best_time_complexity.rs ===

/* 生成一個陣列，元素為 { 1, 2, ..., n }，順序被打亂 */
fn random_numbers(n: i32) -> Vec<i32> {
    // 生成陣列 nums = { 1, 2, 3, ..., n }
    let mut nums = (1..=n).collect::<Vec<i32>>();
    // 隨機打亂陣列元素
    nums.shuffle(&mut thread_rng());
    nums
}

/* 查詢陣列 nums 中數字 1 所在索引 */
fn find_one(nums: &[i32]) -> Option<usize> {
```

```
    for i in 0..nums.len() {
        // 當元素 1 在陣列頭部時，達到最佳時間複雜度 O(1)
        // 當元素 1 在陣列尾部時，達到最差時間複雜度 O(n)
        if nums[i] == 1 {
            return Some(i);
        }
    }
    None
}
```

值得說明的是，我們在實際中很少使用最佳時間複雜度，因為通常只有在很小機率下才能達到，可能會帶來一定的誤導性。**而最差時間複雜度更為實用，因為它給出了一個效率安全值**，讓我們可以放心地使用演算法。

從上述示例可以看出，最差時間複雜度和最佳時間複雜度只出現於“特殊的資料分佈”，這些情況的出現機率可能很小，並不能真實地反映演算法執行效率。相比之下，**平均時間複雜度可以體現演算法在隨機輸入資料下的執行效率**，用 Θ 記號來表示。

對於部分演算法，我們可以簡單地推算出隨機資料分佈下的平均情況。比如上述示例，由於輸入陣列是被打亂的，因此元素 1 出現在任意索引的機率都是相等的，那麼演算法的平均迴圈次數就是陣列長度的一半 $n/2$ ，平均時間複雜度為 $\Theta(n/2) = \Theta(n)$ 。

但對於較為複雜的演算法，計算平均時間複雜度往往比較困難，因為很難分析出在資料分佈下的整體數學期望。在這種情況下，我們通常使用最差時間複雜度作為演算法效率的評判標準。

> **為什麼很少看到 Θ 符號？**
> 可能由於 O 符號過於朗朗上口，因此我們常常使用它來表示平均時間複雜度。但從嚴格意義上講，這種做法並不規範。在本書和其他資料中，若遇到類似“平均時間複雜度 $O(n)$”的表述，請將其直接理解為 $\Theta(n)$ 。

2.4　空間複雜度

空間複雜度（space complexity）用於衡量演算法佔用記憶體空間隨著資料量變大時的增長趨勢。這個概念與時間複雜度非常類似，只需將“執行時間”替換為“佔用記憶體空間”。

2.4.1　演算法相關空間

演算法在執行過程中使用的記憶體空間主要包括以下幾種。

- **輸入空間**：用於儲存演算法的輸入資料。
- **暫存空間**：用於儲存演算法在執行過程中的變數、物件、函式上下文等資料。
- **輸出空間**：用於儲存演算法的輸出資料。

一般情況下，空間複雜度的統計範圍是“暫存空間”加上“輸出空間”。

暫存空間可以進一步劃分為三個部分。

- **暫存資料**：用於儲存演算法執行過程中的各種常數、變數、物件等。
- **堆疊幀空間**：用於儲存呼叫函式的上下文資料。系統在每次呼叫函式時都會在堆疊頂部建立一個堆疊幀，函式返回後，堆疊幀空間會被釋放。
- **指令空間**：用於儲存編譯後的程式指令，在實際統計中通常忽略不計。

在分析一段程式的空間複雜度時，**我們通常統計暫存資料、堆疊幀空間和輸出資料三部分**，如圖 2-15 所示。

圖 2-15　演算法使用的相關空間

相關程式碼如下：

```rust
use std::rc::Rc;
use std::cell::RefCell;

/* 結構體 */
struct Node {
    val: i32,
    next: Option<Rc<RefCell<Node>>>,
}

/* 建立 Node 結構體 */
impl Node {
    fn new(val: i32) -> Self {
        Self { val: val, next: None }
    }
}

/* 函式 */
fn function() -> i32 {
    // 執行某些操作...
    return 0;
}
```

```
fn algorithm(n: i32) -> i32 {       // 輸入資料
    const a: i32 = 0;               // 暫存資料（常數）
    let mut b = 0;                  // 暫存資料（變數）
    let node = Node::new(0);        // 暫存資料（物件）
    let c = function();             // 堆疊幀空間（呼叫函式）
    return a + b + c;               // 輸出資料
}
```

2.4.2　推算方法

空間複雜度的推算方法與時間複雜度大致相同，只需將統計物件從“操作數量”轉為“使用空間大小”。

而與時間複雜度不同的是，**我們通常只關注最差空間複雜度**。這是因為記憶體空間是一項硬性要求，我們必須確保在所有輸入資料下都有足夠的記憶體空間預留。

觀察以下程式碼，最差空間複雜度中的“最差”有兩層含義。

1. **以最差輸入資料為準**：當 $n < 10$ 時，空間複雜度為 $O(1)$ ；但當 $n > 10$ 時，初始化的陣列 `nums` 佔用 $O(n)$ 空間，因此最差空間複雜度為 $O(n)$ 。
2. **以演算法執行中的峰值記憶體為準**：例如，程式在執行最後一行之前，佔用 $O(1)$ 空間；當初始化陣列 `nums` 時，程式佔用 $O(n)$ 空間，因此最差空間複雜度為 $O(n)$ 。

```
fn algorithm(n: i32) {
    let a = 0;                          // O(1)
    let b = [0; 10000];                 // O(1)
    if n > 10 {
        let nums = vec![0; n as usize]; // O(n)
    }
}
```

遞迴函式中，需要注意統計堆疊幀空間。觀察以下程式碼：

```
fn function() -> i32 {
    // 執行某些操作
    return 0;
}
/* 迴圈的空間複雜度為 O(1) */
fn loop(n: i32) {
    for i in 0..n {
        function();
    }
}
/* 遞迴的空間複雜度為 O(n) */
fn recur(n: i32) {
    if n == 1 {
```

```
        return;
    }
    recur(n - 1);
}
```

函式 `loop()` 和 `recur()` 的時間複雜度都為 $O(n)$，但空間複雜度不同。

- 函式 `loop()` 在迴圈中呼叫了 n 次 `function()`，每輪中的 `function()` 都返回並釋放了堆疊幀空間，因此空間複雜度仍為 $O(1)$。
- 遞迴函式 `recur()` 在執行過程中會同時存在 n 個未返回的 `recur()`，從而佔用 $O(n)$ 的堆疊幀空間。

2.4.3　常見型別

設輸入資料大小為 n，圖 2-16 展示了常見的空間複雜度型別（從低到高排列）。

$$
\underbrace{O(1)}_{\text{常數階}} < \underbrace{O(\log n)}_{\text{對數階}} < \underbrace{O(n)}_{\text{線性階}} < \underbrace{O(n^2)}_{\text{平方階}} < \underbrace{O(2^n)}_{\text{指數階}}
$$

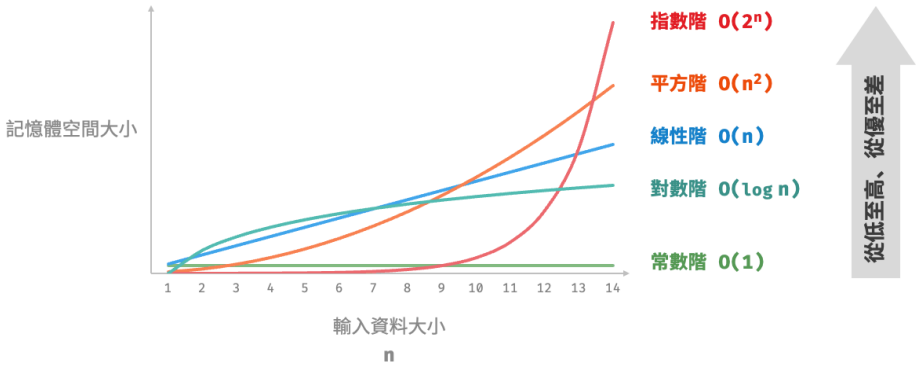


圖 2-16　常見的空間複雜度型別

1.　常數階 $O(1)$

常數階常見於數量與輸入資料大小 n 無關的常數、變數、物件。

需要注意的是，在迴圈中初始化變數或呼叫函式而佔用的記憶體，在進入下一迴圈後就會被釋放，因此不會累積佔用空間，空間複雜度仍為 $O(1)$：

```
// === File: space_complexity.rs ===

/* 函式 */
fn function() -> i32 {
```

```
    // 執行某些操作
    return 0;
}

/* 常數階 */
#[allow(unused)]
fn constant(n: i32) {
    // 常數、變數、物件佔用 O(1) 空間
    const A: i32 = 0;
    let b = 0;
    let nums = vec![0; 10000];
    let node = ListNode::new(0);
    // 迴圈中的變數佔用 O(1) 空間
    for i in 0..n {
        let c = 0;
    }
    // 迴圈中的函式佔用 O(1) 空間
    for i in 0..n {
        function();
    }
}
```

2. 線性階 $O(n)$

線性階常見於元素數量與 n 成正比的陣列、鏈結串列、堆疊、佇列等：

```
// === File: space_complexity.rs ===

/* 線性階 */
#[allow(unused)]
fn linear(n: i32) {
    // 長度為 n 的陣列佔用 O(n) 空間
    let mut nums = vec![0; n as usize];
    // 長度為 n 的串列佔用 O(n) 空間
    let mut nodes = Vec::new();
    for i in 0..n {
        nodes.push(ListNode::new(i))
    }
    // 長度為 n 的雜湊表佔用 O(n) 空間
    let mut map = HashMap::new();
    for i in 0..n {
        map.insert(i, i.to_string());
    }
}
```

如圖 2-17 所示，此函式的遞迴深度為 n ，即同時存在 n 個未返回的 `linear_recur()` 函式，使用 $O(n)$ 大小

的堆疊幀空間：

```
// === File: space_complexity.rs ===

/* 線性階（遞迴實現） */
fn linear_recur(n: i32) {
    println!(" 遞迴 n = {}", n);
    if n == 1 {
        return;
    };
    linear_recur(n - 1);
}
```

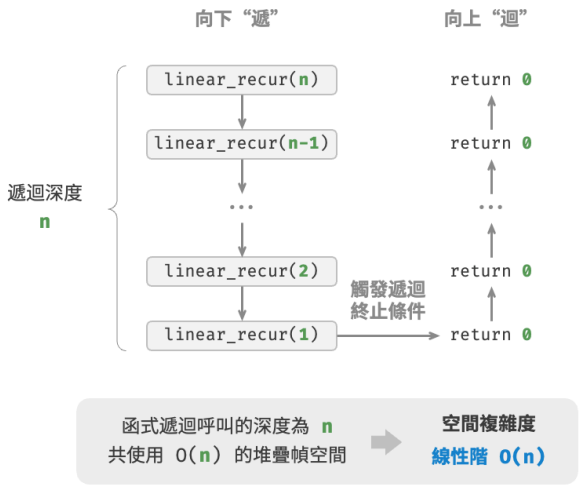


圖 2-17　遞迴函式產生的線性階空間複雜度

3. 平方階 $O(n^2)$

平方階常見於矩陣和圖，元素數量與 n 成平方關係：

```
// === File: space_complexity.rs ===

/* 平方階 */
#[allow(unused)]
fn quadratic(n: i32) {
    // 矩陣佔用 O(n^2) 空間
    let num_matrix = vec![vec![0; n as usize]; n as usize];
    // 二維串列佔用 O(n^2) 空間
    let mut num_list = Vec::new();
    for i in 0..n {
        let mut tmp = Vec::new();
```

```
        for j in 0..n {
            tmp.push(0);
        }
        num_list.push(tmp);
    }
}
```

如圖 2-18 所示，該函式的遞迴深度為 n，在每個遞迴函式中都初始化了一個陣列，長度分別為 n、$n-1$、…、2、1，平均長度為 $n/2$，因此總體佔用 $O(n^2)$ 空間：

```
// === File: space_complexity.rs ===

/* 平方階（遞迴實現） */
fn quadratic_recur(n: i32) -> i32 {
    if n <= 0 {
        return 0;
    };
    // 陣列 nums 長度為 n, n-1, ..., 2, 1
    let nums = vec![0; n as usize];
    println!(" 遞迴 n = {} 中的 nums 長度 = {}", n, nums.len());
    return quadratic_recur(n - 1);
}
```

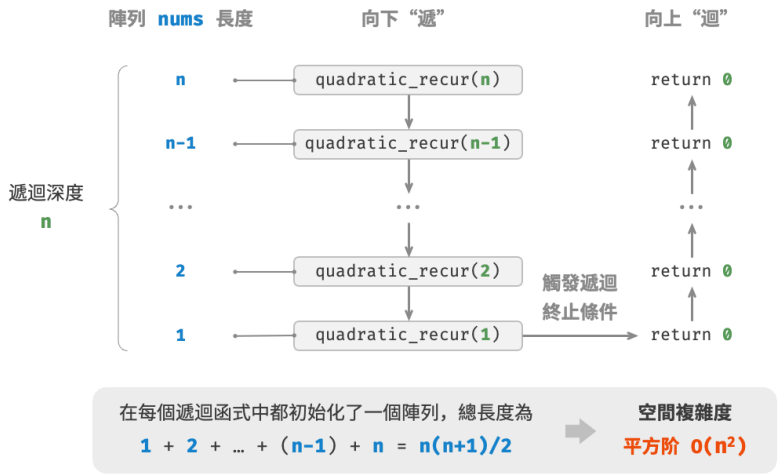


圖 2-18　遞迴函式產生的平方階空間複雜度

4. 指數階 $O(2^n)$

指數階常見於二元樹。觀察圖 2-19，層數為 n 的“滿二元樹”的節點數量為 $2^n - 1$，佔用 $O(2^n)$ 空間：

```rust
// === File: space_complexity.rs ===

/* 指數階（建立滿二元樹） */
fn build_tree(n: i32) -> Option<Rc<RefCell<TreeNode>>> {
    if n == 0 {
        return None;
    };
    let root = TreeNode::new(0);
    root.borrow_mut().left = build_tree(n - 1);
    root.borrow_mut().right = build_tree(n - 1);
    return Some(root);
}
```

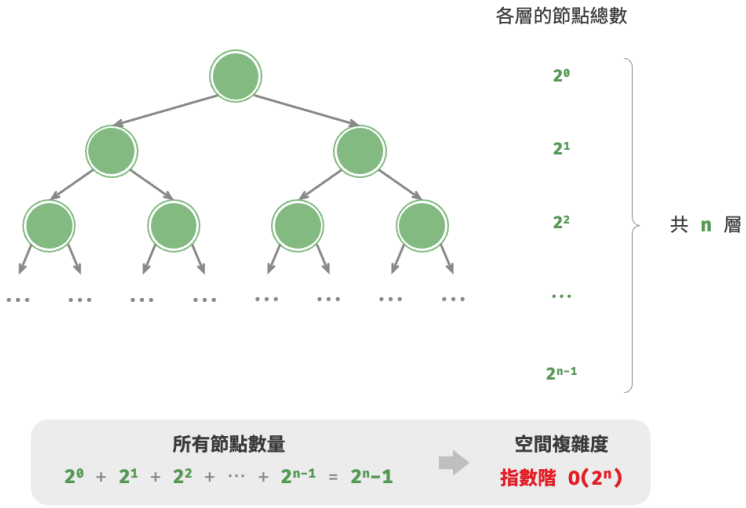


圖 2-19　滿二元樹產生的指數階空間複雜度

5. 對數階 $O(\log n)$

對數階常見於分治演算法。例如合併排序，輸入長度為 n 的陣列，每輪遞迴將陣列從中點處劃分為兩半，形成高度為 $\log n$ 的遞迴樹，使用 $O(\log n)$ 堆疊幀空間。

再例如將數字轉化為字串，輸入一個正整數 n，它的位數為 $\lfloor \log_{10} n \rfloor + 1$，即對應字串長度為 $\lfloor \log_{10} n \rfloor + 1$，因此空間複雜度為 $O(\log_{10} n + 1) = O(\log n)$。

2.4.4　權衡時間與空間

理想情況下，我們希望演算法的時間複雜度和空間複雜度都能達到最優。然而在實際情況中，同時最佳化時間複雜度和空間複雜度通常非常困難。

降低時間複雜度通常需要以提升空間複雜度為代價，反之亦然。我們將犧牲記憶體空間來提升演算法執行速度的思路稱為“以空間換時間”；反之，則稱為“以時間換空間”。

選擇哪種思路取決於我們更看重哪個方面。在大多數情況下，時間比空間更寶貴，因此“以空間換時間”通常是更常用的策略。當然，在資料量很大的情況下，控制空間複雜度也非常重要。

2.5 小結

1. 重點回顧

演算法效率評估

- 時間效率和空間效率是衡量演算法優劣的兩個主要評價指標。
- 我們可以透過實際測試來評估演算法效率，但難以消除測試環境的影響，且會耗費大量計算資源。
- 複雜度分析可以消除實際測試的弊端，分析結果適用於所有執行平臺，並且能夠揭示演算法在不同資料規模下的效率。

時間複雜度

- 時間複雜度用於衡量演算法執行時間隨資料量增長的趨勢，可以有效評估演算法效率，但在某些情況下可能失效，如在輸入的資料量較小或時間複雜度相同時，無法精確對比演算法效率的優劣。
- 最差時間複雜度使用大 O 符號表示，對應函式漸近上界，反映當 n 趨向正無窮時，操作數量 $T(n)$ 的增長級別。
- 推算時間複雜度分為兩步，首先統計操作數量，然後判斷漸近上界。
- 常見時間複雜度從低到高排列有 $O(1)$、$O(\log n)$、$O(n)$、$O(n \log n)$、$O(n^2)$、$O(2^n)$ 和 $O(n!)$ 等。
- 某些演算法的時間複雜度非固定，而是與輸入資料的分佈有關。時間複雜度分為最差、最佳、平均時間複雜度，最佳時間複雜度幾乎不用，因為輸入資料一般需要滿足嚴格條件才能達到最佳情況。
- 平均時間複雜度反映演算法在隨機資料輸入下的執行效率，最接近實際應用中的演算法效能。計算平均時間複雜度需要統計輸入資料分佈以及綜合後的數學期望。

空間複雜度

- 空間複雜度的作用類似於時間複雜度，用於衡量演算法佔用記憶體空間隨資料量增長的趨勢。
- 演算法執行過程中的相關記憶體空間可分為輸入空間、暫存空間、輸出空間。通常情況下，輸入空間不納入空間複雜度計算。暫存空間可分為暫存資料、堆疊幀空間和指令空間，其中堆疊幀空間通常僅在遞迴函式中影響空間複雜度。
- 我們通常只關注最差空間複雜度，即統計演算法在最差輸入資料和最差執行時刻下的空間複雜度。
- 常見空間複雜度從低到高排列有 $O(1)$、$O(\log n)$、$O(n)$、$O(n^2)$ 和 $O(2^n)$ 等。

2. Q & A

Q：尾遞迴的空間複雜度是 $O(1)$ 嗎？

理論上，尾遞迴函式的空間複雜度可以最佳化至 $O(1)$ 。不過絕大多數程式語言（例如 Java、Python、C++、Go、C# 等）不支持自動最佳化尾遞迴，因此通常認為空間複雜度是 $O(n)$ 。

Q：函式和方法這兩個術語的區別是什麼？

函式（function）可以被獨立執行，所有參數都以顯式傳遞。方法（method）與一個物件關聯，被隱式傳遞給呼叫它的物件，能夠對類別的例項中包含的資料進行操作。

下面以幾種常見的程式語言為例來說明。

- C 語言是程序式程式設計語言，沒有物件導向的概念，所以只有函式。但我們可以透過建立結構體（struct）來模擬物件導向程式設計，與結構體相關聯的函式就相當於其他程式語言中的方法。
- Java 和 C# 是物件導向的程式語言，程式碼塊（方法）通常作為某個類別的一部分。靜態方法的行為類似於函式，因為它被繫結在類別上，不能訪問特定的例項變數。
- C++ 和 Python 既支持程序式程式設計（函式），也支持物件導向程式設計（方法）。

Q：圖解“常見的空間複雜度型別”反映的是否是佔用空間的絕對大小？

不是，該圖展示的是空間複雜度，其反映的是增長趨勢，而不是佔用空間的絕對大小。

假設取 $n = 8$ ，你可能會發現每條曲線的值與函式對應不上。這是因為每條曲線都包含一個常數項，用於將取值範圍壓縮到一個視覺舒適的範圍內。

在實際中，因為我們通常不知道每個方法的“常數項”複雜度是多少，所以一般無法僅憑複雜度來選擇 $n = 8$ 之下的最優解法。但對於 $n = 8^5$ 就很好選了，這時增長趨勢已經佔主導了。

第 3 章　資料結構

Abstract

資料結構如同一副穩固而多樣的框架。

它為資料的有序組織提供了藍圖，演算法得以在此基礎上生動起來。

3.1　資料結構分類

常見的資料結構包括陣列、鏈結串列、堆疊、佇列、雜湊表、樹、堆積、圖，它們可以從“邏輯結構”和“物理結構”兩個維度進行分類。

3.1.1　邏輯結構：線性與非線性

邏輯結構揭示了資料元素之間的邏輯關係。在陣列和鏈結串列中，資料按照一定順序排列，體現了資料之間的線性關係；而在樹中，資料從頂部向下按層次排列，表現出“祖先”與“後代”之間的派生關係；圖則由節點和邊構成，反映了複雜的網路關係。

如圖 3-1 所示，邏輯結構可分為“線性”和“非線性”兩大類。線性結構比較直觀，指資料在邏輯關係上呈線性排列；非線性結構則相反，呈非線性排列。

- **線性資料結構**：陣列、鏈結串列、堆疊、佇列、雜湊表，元素之間是一對一的順序關係。
- **非線性資料結構**：樹、堆積、圖、雜湊表。

非線性資料結構可以進一步劃分為樹形結構和網狀結構。

- **樹形結構**：樹、堆積、雜湊表，元素之間是一對多的關係。
- **網狀結構**：圖，元素之間是多對多的關係。

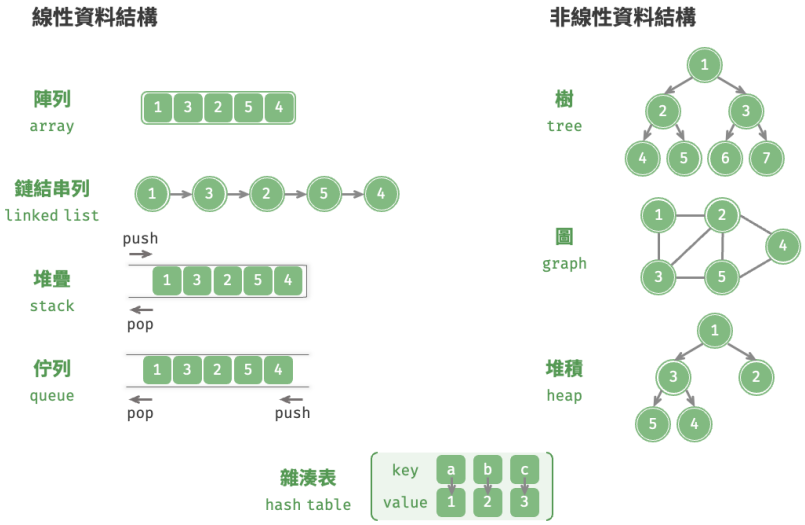


圖 3-1　線性資料結構與非線性資料結構

3.1.2　物理結構：連續與分散

當演算法程式執行時，正在處理的資料主要儲存在記憶體中。圖 3-2 展示了一個計算機記憶體條，其中每個黑色方塊都包含一塊記憶體空間。我們可以將記憶體想象成一個巨大的 Excel 表格，其中每個單元格都可以儲存一定大小的資料。

系統透過記憶體位址來訪問目標位置的資料。如圖 3-2 所示，計算機根據特定規則為表格中的每個單元格分配編號，確保每個記憶體空間都有唯一的記憶體位址。有了這些位址，程式便可以訪問記憶體中的資料。

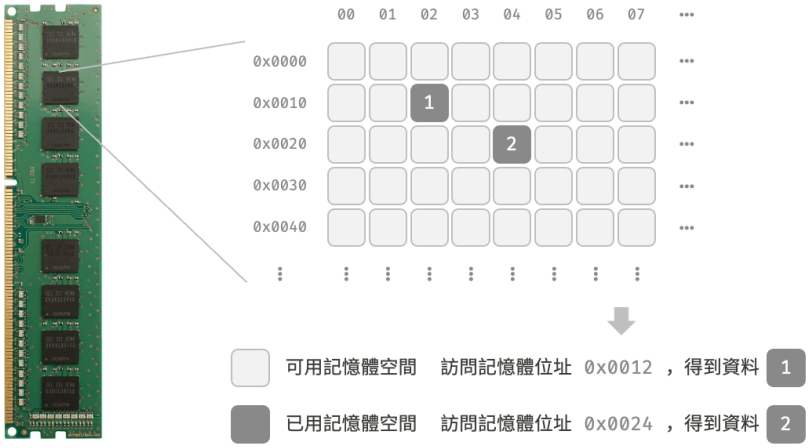


圖 3-2　記憶體條、記憶體空間、記憶體位址

> **Tip**
> 值得說明的是，將記憶體比作 Excel 表格是一個簡化的類比，實際記憶體的工作機制比較複雜，涉及位址空間、記憶體管理、快取機制、虛擬記憶體和物理記憶體等概念。

記憶體是所有程式的共享資源，當某塊記憶體被某個程式佔用時，則通常無法被其他程式同時使用了。**因此在資料結構與演算法的設計中，記憶體資源是一個重要的考慮因素**。比如，演算法所佔用的記憶體峰值不應超過系統剩餘空間記憶體；如果缺少連續大塊的記憶體空間，那麼所選用的資料結構必須能夠儲存在分散的記憶體空間內。

如圖 3-3 所示，**物理結構反映了資料在計算機記憶體中的儲存方式**，可分為連續空間儲存（陣列）和分散空間儲存（鏈結串列）。物理結構從底層決定了資料的訪問、更新、增刪等操作方法，兩種物理結構在時間效率和空間效率方面呈現出互補的特點。

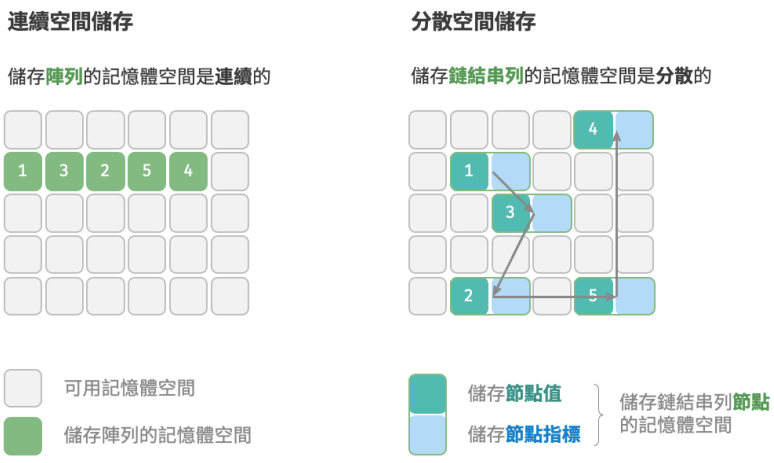


圖 3-3　連續空間儲存與分散空間儲存

值得說明的是，**所有資料結構都是基於陣列、鏈結串列或二者的組合實現的**。例如，堆疊和佇列既可以使用陣列實現，也可以使用鏈結串列實現；而雜湊表的實現可能同時包含陣列和鏈結串列。

- **基於陣列可實現**：堆疊、佇列、雜湊表、樹、堆積、圖、矩陣、張量（維度 ≥ 3 的陣列）等。
- **基於鏈結串列可實現**：堆疊、佇列、雜湊表、樹、堆積、圖等。

鏈結串列在初始化後，仍可以在程式執行過程中對其長度進行調整，因此也稱“動態資料結構”。陣列在初始化後長度不可變，因此也稱“靜態資料結構”。值得注意的是，陣列可透過重新分配記憶體實現長度變化，從而具備一定的“動態性”。

Tip
如果你感覺物理結構理解起來有困難，建議先閱讀下一章，然後再回顧本節內容。

3.2　基本資料型別

當談及計算機中的資料時，我們會想到文字、圖片、影片、語音、3D 模型等各種形式。儘管這些資料的組織形式各異，但它們都由各種基本資料型別構成。

基本資料型別是 CPU 可以直接進行運算的型別，在演算法中直接被使用，主要包括以下幾種。

- 整數型別 `byte`、`short`、`int`、`long` 。
- 浮點數型別 `float`、`double` ，用於表示小數。
- 字元型別 `char` ，用於表示各種語言的字母、標點符號甚至表情符號等。
- 布林型別 `bool` ，用於表示“是”與“否”判斷。

基本資料型別以二進位制的形式儲存在計算機中。一個二進位制位即為 1 位元。在絕大多數現代作業系統中，1 位元組（byte）由 8 位元（bit）組成。

基本資料型別的取值範圍取決於其佔用的空間大小。下面以 Java 為例。

- 整數型別 `byte` 佔用 1 位元組 = 8 位元，可以表示 2^8 個數字。
- 整數型別 `int` 佔用 4 位元組 = 32 位元，可以表示 2^{32} 個數字。

表 3-1 列舉了 Java 中各種基本資料型別的佔用空間、取值範圍和預設值。此表格無須死記硬背，大致理解即可，需要時可以透過查表來回憶。

表 3-1　基本資料型別的佔用空間和取值範圍

型別	符號	佔用空間	最小值	最大值	預設值
整數	`byte`	1 位元組	-2^7 (-128)	$2^7 - 1$ (127)	0
	`short`	2 位元組	-2^{15}	$2^{15} - 1$	0
	`int`	4 位元組	-2^{31}	$2^{31} - 1$	0

型別	符號	佔用空間	最小值	最大值	預設值
	`long`	8 位元組	-2^{63}	$2^{63}-1$	0
浮點數	`float`	4 位元組	1.175×10^{-38}	3.403×10^{38}	0.0f
	`double`	8 位元組	2.225×10^{-308}	1.798×10^{308}	0.0
字元	`char`	2 位元組	0	$2^{16}-1$	0
布林	`bool`	1 位元組	false	true	false

請注意，表 3-1 針對的是 Java 的基本資料型別的情況。每種程式語言都有各自的資料型別定義，它們的佔用空間、取值範圍和預設值可能會有所不同。

- 在 Python 中，整數型別 `int` 可以是任意大小，只受限於可用記憶體；浮點數 `float` 是雙精度 64 位；沒有 `char` 型別，單個字元實際上是長度為 1 的字串 `str` 。
- C 和 C++ 未明確規定基本資料型別的大小，而因實現和平臺各異。表 3-1 遵循 LP64 資料模型，其用於包括 Linux 和 macOS 在內的 Unix 64 位作業系統。
- 字元 `char` 的大小在 C 和 C++ 中為 1 位元組，在大多數程式語言中取決於特定的字元編碼方法，詳見“字元編碼”章節。
- 即使表示布林量僅需 1 位（0 或 1），它在記憶體中通常也儲存為 1 位元組。這是因為現代計算機 CPU 通常將 1 位元組作為最小定址記憶體單元。

那麼，基本資料型別與資料結構之間有什麼關聯呢？我們知道，資料結構是在計算機中組織與儲存資料的方式。這句話的主語是“結構”而非“資料”。

如果想表示“一排數字”，我們自然會想到使用陣列。這是因為陣列的線性結構可以表示數字的相鄰關係和順序關係，但至於儲存的內容是整數 `int`、小數 `float` 還是字元 `char` ，則與“資料結構”無關。

換句話說，**基本資料型別提供了資料的“內容型別”，而資料結構提供了資料的“組織方式”**。例如以下程式碼，我們用相同的資料結構（陣列）來儲存與表示不同的基本資料型別，包括 `int`、`float`、`char`、`bool` 等。

```
// 使用多種基本資料型別來初始化陣列
let numbers: Vec<i32> = vec![0; 5];
let decimals: Vec<f32> = vec![0.0; 5];
let characters: Vec<char> = vec!['0'; 5];
let bools: Vec<bool> = vec![false; 5];
```

3.3　數字編碼 *

> **Tip**
> 在本書中，標題帶有 * 符號的是選讀章節。如果你時間有限或感到理解困難，可以先跳過，等學完必讀章節後再單獨攻克。

3.3.1　原碼、一補數和二補數

在上一節的表格中我們發現，所有整數型別能夠表示的負數都比正數多一個，例如 `byte` 的取值範圍是 $[-128, 127]$ 。這個現象比較反直覺，它的內在原因涉及原碼、一補數、二補數的相關知識。

首先需要指出，**數字是以“二補數”的形式儲存在計算機中的**。在分析這樣做的原因之前，首先給出三者的定義。

- **原碼**：我們將數字的二進位制表示的最高位視為符號位，其中 0 表示正數，1 表示負數，其餘位表示數字的值。
- **一補數**：正數的一補數與其原碼相同，負數的一補數是對其原碼除符號位外的所有位取反。
- **二補數**：正數的二補數與其原碼相同，負數的二補數是在其一補數的基礎上加 1 。

圖 3-4 展示了原碼、一補數和二補數之間的轉換方法。

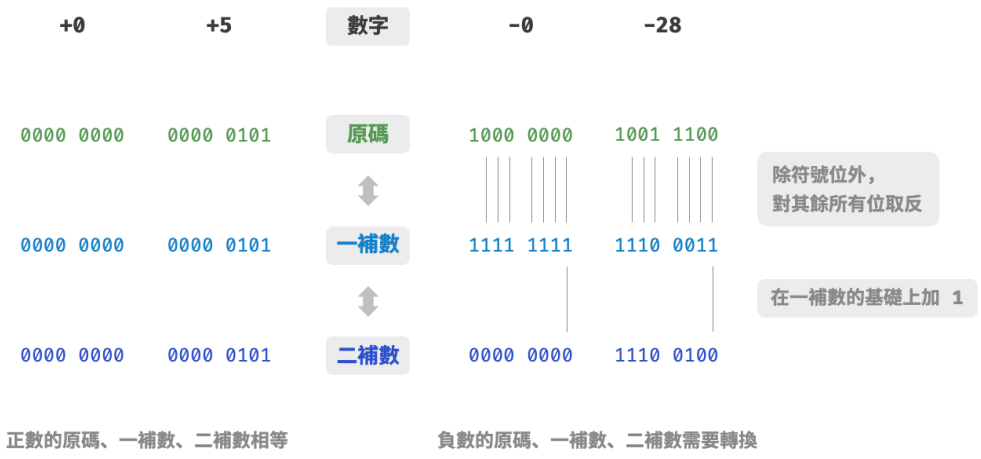


圖 3-4　原碼、一補數與二補數之間的相互轉換

原碼（sign-magnitude）雖然最直觀，但存在一些侷限性。一方面，**負數的原碼不能直接用於運算**。例如在原碼下計算 $1 + (-2)$ ，得到的結果是 -3 ，這顯然是不對的。

$$
\begin{aligned}
& 1 + (-2) \\
& \rightarrow 0000\ 0001 + 1000\ 0010 \\
& = 1000\ 0011 \\
& \rightarrow -3
\end{aligned}
$$

為了解決此問題，計算機引入了一補數（1’s complement）。如果我們先將原碼轉換為一補數，並在一補數下計算 $1 + (-2)$ ，最後將結果從一補數轉換回原碼，則可得到正確結果 -1 。

$$
\begin{aligned}
& 1 + (-2) \\
& \to \text{0000 0001 (原碼)} + \text{1000 0010 (原碼)} \\
& = \text{0000 0001 (一補數)} + \text{1111 1101 (一補數)} \\
& = \text{1111 1110 (一補數)} \\
& = \text{1000 0001 (原碼)} \\
& \to -1
\end{aligned}
$$

另一方面，**數字零的原碼有 $+0$ 和 -0 兩種表示方式**。這意味著數字零對應兩個不同的二進位制編碼，這可能會帶來歧義。比如在條件判斷中，如果沒有區分正零和負零，則可能會導致判斷結果出錯。而如果我們想處理正零和負零歧義，則需要引入額外的判斷操作，這可能會降低計算機的運算效率。

$$
\begin{aligned}
+0 &\to \text{0000 0000} \\
-0 &\to \text{1000 0000}
\end{aligned}
$$

與原碼一樣，一補數也存在正負零歧義問題，因此計算機進一步引入了**二補數**（2's complement）。我們先來觀察一下負零的原碼、一補數、二補數的轉換過程：

$$
\begin{aligned}
-0 &\to \text{1000 0000 (原碼)} \\
&= \text{1111 1111 (一補數)} \\
&= \text{1 0000 0000 (二補數)}
\end{aligned}
$$

在負零的一補數基礎上加 1 會產生進位，但 `byte` 型別的長度只有 8 位，因此溢位到第 9 位的 1 會被捨棄。也就是說，**負零的二補數為 $0000\ 0000$，與正零的二補數相同**。這意味著在二補數表示中只存在一個零，正負零歧義從而得到解決。

還剩最後一個疑惑：`byte` 型別的取值範圍是 $[-128, 127]$，多出來的一個負數 -128 是如何得到的呢？我們注意到，區間 $[-127, +127]$ 內的所有整數都有對應的原碼、一補數和二補數，並且原碼和二補數之間可以互相轉換。

然而，**二補數 $1000\ 0000$ 是一個例外，它並沒有對應的原碼**。根據轉換方法，我們得到該二補數的原碼為 $0000\ 0000$。這顯然是矛盾的，因為該原碼表示數字 0，它的二補數應該是自身。計算機規定這個特殊的二補數 $1000\ 0000$ 代表 -128。實際上，$(-1) + (-127)$ 在二補數下的計算結果就是 -128。

$$
\begin{aligned}
& (-127) + (-1) \\
& \to \text{1111 1111 (原碼)} + \text{1000 0001 (原碼)} \\
& = \text{1000 0000 (一補數)} + \text{1111 1110 (一補數)} \\
& = \text{1000 0001 (二補數)} + \text{1111 1111 (二補數)} \\
& = \text{1000 0000 (二補數)} \\
& \to -128
\end{aligned}
$$

你可能已經發現了，上述所有計算都是加法運算。這暗示著一個重要事實：**計算機內部的硬體電路主要是基於加法運算設計的**。這是因為加法運算相對於其他運算（比如乘法、除法和減法）來說，硬體實現起來更簡

單，更容易進行並行化處理，運算速度更快。

請注意，這並不意味著計算機只能做加法。**透過將加法與一些基本邏輯運算結合，計算機能夠實現各種其他的數學運算**。例如，計算減法 $a - b$ 可以轉換為計算加法 $a + (-b)$；計算乘法和除法可以轉換為計算多次加法或減法。

現在我們可以總結出計算機使用二補數的原因：基於二補數表示，計算機可以用同樣的電路和操作來處理正數和負數的加法，不需要設計特殊的硬體電路來處理減法，並且無須特別處理正負零的歧義問題。這大大簡化了硬體設計，提高了運算效率。

二補數的設計非常精妙，因篇幅關係我們就先介紹到這裡，建議有興趣的讀者進一步深入瞭解。

3.3.2 浮點數編碼

細心的你可能會發現：`int` 和 `float` 長度相同，都是 4 位元組，但為什麼 `float` 的取值範圍遠大於 `int`？這非常反直覺，因為按理說 `float` 需要表示小數，取值範圍應該變小才對。

實際上，**這是因為浮點數 `float` 採用了不同的表示方式**。記一個 32 位元長度的二進位制數為：

$$b_{31}b_{30}b_{29}\ldots b_2b_1b_0$$

根據 IEEE 754 標準，32-bit 長度的 `float` 由以下三個部分構成。

- 符號位 S：佔 1 位，對應 b_{31}。
- 指數位 E：佔 8 位，對應 $b_{30}b_{29}\ldots b_{23}$。
- 分數位 N：佔 23 位，對應 $b_{22}b_{21}\ldots b_0$。

二進位制數 `float` 對應值的計算方法為：

$$\text{val} = (-1)^{b_{31}} \times 2^{(b_{30}b_{29}\ldots b_{23})_2 - 127} \times (1.b_{22}b_{21}\ldots b_0)_2$$

轉化到十進位制下的計算公式為：

$$\text{val} = (-1)^{\text{S}} \times 2^{\text{E} - 127} \times (1 + \text{N})$$

其中各項的取值範圍為：

$$\begin{aligned} \text{S} \in &\{0, 1\}, \quad \text{E} \in \{1, 2, \ldots, 254\} \\ (1 + \text{N}) =&(1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}) \subset [1, 2 - 2^{-23}] \end{aligned}$$

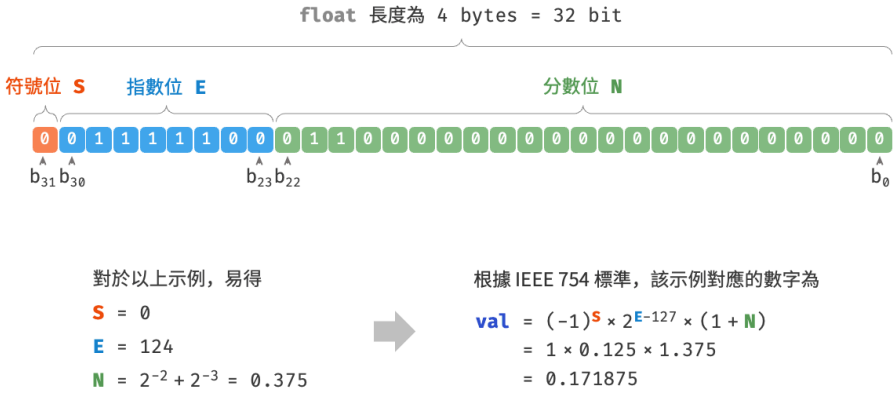


圖 3-5　IEEE 754 標準下的 float 的計算示例

觀察圖 3-5，給定一個示例資料 $\mathrm{S} = 0$，$\mathrm{E} = 124$，$\mathrm{N} = 2^{-2} + 2^{-3} = 0.375$，則有：

$$\text{val} = (-1)^0 \times 2^{124-127} \times (1 + 0.375) = 0.171875$$

現在我們可以回答最初的問題：**`float` 的表示方式包含指數位，導致其取值範圍遠大於 `int`**。根據以上計算，`float` 可表示的最大正數為 $2^{254-127} \times (2 - 2^{-23}) \approx 3.4 \times 10^{38}$，切換符號位便可得到最小負數。

儘管浮點數 `float` 擴展了取值範圍，但其副作用是犧牲了精度。整數型別 `int` 將全部 32 位元用於表示數字，數字是均勻分佈的；而由於指數位的存在，浮點數 `float` 的數值越大，相鄰兩個數字之間的差值就會趨向越大。

如表 3-2 所示，指數位 $\mathrm{E} = 0$ 和 $\mathrm{E} = 255$ 具有特殊含義，**用於表示零、無窮大、NaN 等**。

表 3-2　指數位含義

指數位 E	分數位 $\mathrm{N} = 0$	分數位 $\mathrm{N} \neq 0$	計算公式
0	± 0	次正規數	$(-1)^{\mathrm{S}} \times 2^{-126} \times (0.\mathrm{N})$
$1, 2, \dots, 254$	正規數	正規數	$(-1)^{\mathrm{S}} \times 2^{(\mathrm{E}-127)} \times (1.\mathrm{N})$
255	$\pm\infty$	NaN	

值得說明的是，次正規數顯著提升了浮點數的精度。最小正正規數為 2^{-126}，最小正次正規數為 $2^{-126} \times 2^{-23}$。

雙精度 `double` 也採用類似於 `float` 的表示方法，在此不做贅述。

3.4　字元編碼 *

在計算機中，所有資料都是以二進位制數的形式儲存的，字元 `char` 也不例外。為了表示字元，我們需要建立一套“字符集”，規定每個字元和二進位制數之間的一一對應關係。有了字符集之後，計算機就可以透過查表完成二進位制數到字元的轉換。

3.4.1　ASCII 字符集

ASCII 碼是最早出現的字符集，其全稱為 American Standard Code for Information Interchange（美國標準資訊交換程式碼）。它使用 7 位二進位制數（一個位元組的低 7 位）表示一個字元，最多能夠表示 128 個不同的字元。如圖 3-6 所示，ASCII 碼包括英文字母的大小寫、數字 0 ~ 9、一些標點符號，以及一些控制字元（如換行符和製表符）。

十進位制	二進位制	字元	含義
0	0000 0000	NUL	空字元
1	0000 0001	SOH	標題開始
2	0000 0010	STX	正文開始
3	0000 0011	ETX	正文結束
4	0000 0100	EOT	傳輸結束
5	0000 0101	ENQ	請求
6	0000 0110	ACK	收到通知
7	0000 0111	BEL	響鈴
8	0000 1000	BS	退格
9	0000 1001	HT	水平製表符
10	0000 1010	LF	換行鍵
11	0000 1011	VT	垂直製表符
12	0000 1100	FF	換頁鍵
13	0000 1101	CR	回車鍵
14	0000 1110	SO	不用切換
15	0000 1111	SI	啟用切換
16	0001 0000	DLE	資料鏈路轉義
17	0001 0001	DC1	裝置控制1
18	0001 0010	DC2	裝置控制2
19	0001 0011	DC3	裝置控制3
20	0001 0100	DC4	裝置控制4
21	0001 0101	NAK	拒絕接收
22	0001 0110	SYN	同步空閒
23	0001 0111	ETB	結束傳輸塊
24	0001 1000	CAN	取消
25	0001 1001	EM	媒介結束
26	0001 1010	SUB	代替
27	0001 1011	ESC	換碼(溢位)
28	0001 1100	FS	檔案分隔符
29	0001 1101	GS	分組符
30	0001 1110	RS	記錄分隔符
31	0001 1111	US	單元分隔符
32	0010 0000	SP	空格

十進位制	二進位制	字元
33	0010 0001	!
34	0010 0010	"
35	0010 0011	#
36	0010 0100	$
37	0010 0101	%
38	0010 0110	&
39	0010 0111	'
40	0010 1000	(
41	0010 1001	)
42	0010 1010	*
43	0010 1011	+
44	0010 1100	,
45	0010 1101	-
46	0010 1110	.
47	0010 1111	/
48	0011 0000	0
49	0011 0001	1
50	0011 0010	2
51	0011 0011	3
52	0011 0100	4
53	0011 0101	5
54	0011 0110	6
55	0011 0111	7
56	0011 1000	8
57	0011 1001	9
58	0011 1010	:
59	0011 1011	;
60	0011 1100	<
61	0011 1101	=
62	0011 1110	>
63	0011 1111	?
64	0100 0000	@

十進位制	二進位制	字元
65	0100 0001	A
66	0100 0010	B
67	0100 0011	C
68	0100 0100	D
69	0100 0101	E
70	0100 0110	F
71	0100 0111	G
72	0100 1000	H
73	0100 1001	I
74	0100 1010	J
75	0100 1011	K
76	0100 1100	L
77	0100 1101	M
78	0100 1110	N
79	0100 1111	O
80	0101 0000	P
81	0101 0001	Q
82	0101 0010	R
83	0101 0011	S
84	0101 0100	T
85	0101 0101	U
86	0101 0110	V
87	0101 0111	W
88	0101 1000	X
89	0101 1001	Y
90	0101 1010	Z
91	0101 1011	[
92	0101 1100	\
93	0101 1101	]
94	0101 1110	^
95	0101 1111	_
96	0110 0000	`

十進位制	二進位制	字元
97	0110 0001	a
98	0110 0010	b
99	0110 0011	c
100	0110 0100	d
101	0110 0101	e
102	0110 0110	f
103	0110 0111	g
104	0110 1000	h
105	0110 1001	i
106	0110 1010	j
107	0110 1011	k
108	0110 1100	l
109	0110 1101	m
110	0110 1110	n
111	0110 1111	o
112	0111 0000	p
113	0111 0001	q
114	0111 0010	r
115	0111 0011	s
116	0111 0100	t
117	0111 0101	u
118	0111 0110	v
119	0111 0111	w
120	0111 1000	x
121	0111 1001	y
122	0111 1010	z
123	0111 1011	{
124	0111 1100	|
125	0111 1101	}
126	0111 1110	~
127	0111 1111	DEL

圖 3-6　ASCII 碼

然而，**ASCII 碼僅能夠表示英文**。隨著計算機的全球化，誕生了一種能夠表示更多語言的 EASCII 字符集。它在 ASCII 的 7 位基礎上擴展到 8 位，能夠表示 256 個不同的字元。

在世界範圍內，陸續出現了一批適用於不同地區的 EASCII 字符集。這些字符集的前 128 個字元統一為 ASCII 碼，後 128 個字元定義不同，以適應不同語言的需求。

3.4.2　GBK 字符集

後來人們發現，**EASCII 碼仍然無法滿足許多語言的字元數量要求**。比如漢字有近十萬個，光日常使用的就有幾千個。中國國家標準總局於 1980 年釋出了 GB2312 字符集，其收錄了 6763 個漢字，基本滿足了漢字的計算機處理需要。

然而，GB2312 無法處理部分罕見字和繁體字。GBK 字符集是在 GB2312 的基礎上擴展得到的，它共收錄了 21886 個漢字。在 GBK 的編碼方案中，ASCII 字元使用一個位元組表示，漢字使用兩個位元組表示。

3.4.3　Unicode 字符集

隨著計算機技術的蓬勃發展，字符集與編碼標準百花齊放，而這帶來了許多問題。一方面，這些字符集一般只定義了特定語言的字元，無法在多語言環境下正常工作。另一方面，同一種語言存在多種字符集標準，如果兩臺計算機使用的是不同的編碼標準，則在資訊傳遞時就會出現亂碼。

那個時代的研究人員就在想：**如果推出一個足夠完整的字符集，將世界範圍內的所有語言和符號都收錄其中，不就可以解決跨語言環境和亂碼問題了嗎**？在這種想法的驅動下，一個大而全的字符集 Unicode 應運而生。

Unicode 的中文名稱為“統一碼”，理論上能容納 100 多萬個字元。它致力於將全球範圍內的字元納入統一的字符集之中，提供一種通用的字符集來處理和顯示各種語言文字，減少因為編碼標準不同而產生的亂碼問題。

自 1991 年釋出以來，Unicode 不斷擴充新的語言與字元。截至 2022 年 9 月，Unicode 已經包含 149186 個字元，包括各種語言的字元、符號甚至表情符號等。在龐大的 Unicode 字符集中，常用的字元佔用 2 位元組，有些生僻的字元佔用 3 位元組甚至 4 位元組。

Unicode 是一種通用字符集，本質上是給每個字元分配一個編號（稱為“碼點”），**但它並沒有規定在計算機中如何儲存這些字元碼點**。我們不禁會問：當多種長度的 Unicode 碼點同時出現在一個文字中時，系統如何解析字元？例如給定一個長度為 2 位元組的編碼，系統如何確認它是一個 2 位元組的字元還是兩個 1 位元組的字元？

對於以上問題，**一種直接的解決方案是將所有字元儲存為等長的編碼**。如圖 3-7 所示，“Hello”中的每個字元佔用 1 位元組，“演算法”中的每個字元佔用 2 位元組。我們可以透過高位填 0 將“Hello 演算法”中的所有字元都編碼為 2 位元組長度。這樣系統就可以每隔 2 位元組解析一個字元，恢復這個短語的內容了。

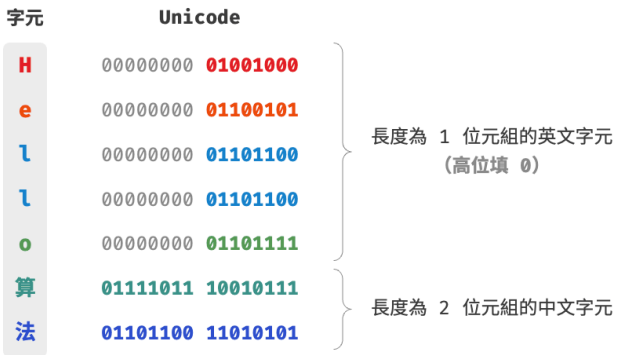


圖 3-7　Unicode 編碼示例

然而 ASCII 碼已經向我們證明，編碼英文只需 1 位元組。若採用上述方案，英文文字佔用空間的大小將會是 ASCII 編碼下的兩倍，非常浪費記憶體空間。因此，我們需要一種更加高效的 Unicode 編碼方法。

3.4.4　UTF-8 編碼

目前，UTF-8 已成為國際上使用最廣泛的 Unicode 編碼方法。**它是一種可變長度的編碼**，使用 1 到 4 位元組來表示一個字元，根據字元的複雜性而變。ASCII 字元只需 1 位元組，拉丁字母和希臘字母需要 2 位元組，

常用的中文字元需要 3 位元組，其他的一些生僻字元需要 4 位元組。

UTF-8 的編碼規則並不複雜，分為以下兩種情況。

- 對於長度為 1 位元組的字元，將最高位設定為 0 ，其餘 7 位設定為 Unicode 碼點。值得注意的是，ASCII 字元在 Unicode 字符集中佔據了前 128 個碼點。也就是說，**UTF-8 編碼可以向下相容 ASCII 碼**。這意味著我們可以使用 UTF-8 來解析年代久遠的 ASCII 碼文字。
- 對於長度為 n 位元組的字元（其中 $n > 1$），將首個位元組的高 n 位都設定為 1 ，第 $n + 1$ 位設定為 0 ；從第二個位元組開始，將每個位元組的高 2 位都設定為 10 ；其餘所有位用於填充字元的 Unicode 碼點。

圖 3-8 展示了“Hello 演算法”對應的 UTF-8 編碼。觀察發現，由於最高 n 位都設定為 1 ，因此系統可以透過讀取最高位 1 的個數來解析出字元的長度為 n 。

但為什麼要將其餘所有位元組的高 2 位都設定為 10 呢？實際上，這個 10 能夠起到校驗符的作用。假設系統從一個錯誤的位元組開始解析文字，位元組頭部的 10 能夠幫助系統快速判斷出異常。

之所以將 10 當作校驗符，是因為在 UTF-8 編碼規則下，不可能有字元的最高兩位是 10 。這個結論可以用反證法來證明：假設一個字元的最高兩位是 10 ，說明該字元的長度為 1 ，對應 ASCII 碼。而 ASCII 碼的最高位應該是 0 ，與假設矛盾。

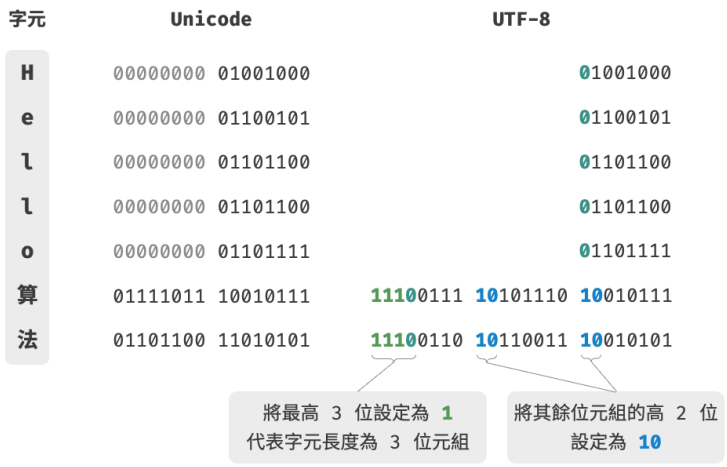


圖 3-8　UTF-8 編碼示例

除了 UTF-8 之外，常見的編碼方式還包括以下兩種。

- **UTF-16 編碼**：使用 2 或 4 位元組來表示一個字元。所有的 ASCII 字元和常用的非英文字元，都用 2 位元組表示；少數字符需要用到 4 位元組表示。對於 2 位元組的字元，UTF-16 編碼與 Unicode 碼點相等。
- **UTF-32 編碼**：每個字元都使用 4 位元組。這意味著 UTF-32 比 UTF-8 和 UTF-16 更佔用空間，特別是對於 ASCII 字元佔比較高的文字。

從儲存空間佔用的角度看，使用 UTF-8 表示英文字元非常高效，因為它僅需 1 位元組；使用 UTF-16 編碼某些非英文字元（例如中文）會更加高效，因為它僅需 2 位元組，而 UTF-8 可能需要 3 位元組。

從相容性的角度看，UTF-8 的通用性最佳，許多工具和庫優先支持 UTF-8 。

3.4.5　程式語言的字元編碼

對於以往的大多數程式語言，程式執行中的字串都採用 UTF-16 或 UTF-32 這類等長編碼。在等長編碼下，我們可以將字串看作陣列來處理，這種做法具有以下優點。

- **隨機訪問**：UTF-16 編碼的字串可以很容易地進行隨機訪問。UTF-8 是一種變長編碼，要想找到第 i 個字元，我們需要從字串的開始處走訪到第 i 個字元，這需要 $O(n)$ 的時間。
- **字元計數**：與隨機訪問類似，計算 UTF-16 編碼的字串的長度也是 $O(1)$ 的操作。但是，計算 UTF-8 編碼的字串的長度需要走訪整個字串。
- **字串操作**：在 UTF-16 編碼的字串上，很多字串操作（如分割、連線、插入、刪除等）更容易進行。在 UTF-8 編碼的字串上，進行這些操作通常需要額外的計算，以確保不會產生無效的 UTF-8 編碼。

實際上，程式語言的字元編碼方案設計是一個很有趣的話題，涉及許多因素。

- Java 的 `String` 型別使用 UTF-16 編碼，每個字元佔用 2 位元組。這是因為 Java 語言設計之初，人們認為 16 位足以表示所有可能的字元。然而，這是一個不正確的判斷。後來 Unicode 規範擴展到了超過 16 位，所以 Java 中的字元現在可能由一對 16 位的值（稱為“代理對”）表示。
- JavaScript 和 TypeScript 的字串使用 UTF-16 編碼的原因與 Java 類似。當 1995 年 Netscape 公司首次推出 JavaScript 語言時，Unicode 還處於發展早期，那時候使用 16 位的編碼就足以表示所有的 Unicode 字元了。
- C# 使用 UTF-16 編碼，主要是因為.NET 平臺是由 Microsoft 設計的，而 Microsoft 的很多技術（包括 Windows 作業系統）都廣泛使用 UTF-16 編碼。

由於以上程式語言對字元數量的低估，它們不得不採取“代理對”的方式來表示超過 16 位長度的 Unicode 字元。這是一個不得已為之的無奈之舉。一方面，包含代理對的字串中，一個字元可能佔用 2 位元組或 4 位元組，從而喪失了等長編碼的優勢。另一方面，處理代理對需要額外增加程式碼，這提高了程式設計的複雜性和除錯難度。

出於以上原因，部分程式語言提出了一些不同的編碼方案。

- Python 中的 `str` 使用 Unicode 編碼，並採用一種靈活的字串表示，儲存的字元長度取決於字串中最大的 Unicode 碼點。若字串中全部是 ASCII 字元，則每個字元佔用 1 位元組；如果有字元超出了 ASCII 範圍，但全部在基本多語言平面（BMP）內，則每個字元佔用 2 位元組；如果有超出 BMP 的字元，則每個字元佔用 4 位元組。
- Go 語言的 `string` 型別在內部使用 UTF-8 編碼。Go 語言還提供了 `rune` 型別，它用於表示單個 Unicode 碼點。
- Rust 語言的 `str` 和 `String` 型別在內部使用 UTF-8 編碼。Rust 也提供了 `char` 型別，用於表示單個 Unicode 碼點。

需要注意的是，以上討論的都是字串在程式語言中的儲存方式，**這和字串如何在檔案中儲存或在網路中傳輸是不同的問題**。在檔案儲存或網路傳輸中，我們通常會將字串編碼為 UTF-8 格式，以達到最優的相容性和空間效率。

3.5 小結

1. 重點回顧

- 資料結構可以從邏輯結構和物理結構兩個角度進行分類。邏輯結構描述了資料元素之間的邏輯關係，而物理結構描述了資料在計算機記憶體中的儲存方式。
- 常見的邏輯結構包括線性、樹狀和網狀等。通常我們根據邏輯結構將資料結構分為線性（陣列、鏈結串列、堆疊、佇列）和非線性（樹、圖、堆積）兩種。雜湊表的實現可能同時包含線性資料結構和非線性資料結構。
- 當程式執行時，資料被儲存在計算機記憶體中。每個記憶體空間都擁有對應的記憶體位址，程式透過這些記憶體位址訪問資料。
- 物理結構主要分為連續空間儲存（陣列）和分散空間儲存（鏈結串列）。所有資料結構都是由陣列、鏈結串列或兩者的組合實現的。
- 計算機中的基本資料型別包括整數 `byte`、`short`、`int`、`long`，浮點數 `float`、`double`，字元 `char` 和布林 `bool`。它們的取值範圍取決於佔用空間大小和表示方式。
- 原碼、一補數和二補數是在計算機中編碼數字的三種方法，它們之間可以相互轉換。整數的原碼的最高位是符號位，其餘位是數字的值。
- 整數在計算機中是以二補數的形式儲存的。在二補數表示下，計算機可以對正數和負數的加法一視同仁，不需要為減法操作單獨設計特殊的硬體電路，並且不存在正負零歧義的問題。
- 浮點數的編碼由 1 位符號位、8 位指數位和 23 位分數位構成。由於存在指數位，因此浮點數的取值範圍遠大於整數，代價是犧牲了精度。
- ASCII 碼是最早出現的英文字符集，長度為 1 位元組，共收錄 127 個字元。GBK 字符集是常用的中文字符集，共收錄兩萬多個漢字。Unicode 致力於提供一個完整的字符集標準，收錄世界上各種語言的字元，從而解決由於字元編碼方法不一致而導致的亂碼問題。
- UTF-8 是最受歡迎的 Unicode 編碼方法，通用性非常好。它是一種變長的編碼方法，具有很好的擴展性，有效提升了儲存空間的使用效率。UTF-16 和 UTF-32 是等長的編碼方法。在編碼中文時，UTF-16 佔用的空間比 UTF-8 更小。Java 和 C# 等程式語言預設使用 UTF-16 編碼。

2. Q & A

Q：為什麼雜湊表同時包含線性資料結構和非線性資料結構？

雜湊表底層是陣列，而為了解決雜湊衝突，我們可能會使用“鏈式位址”（後續“雜湊衝突”章節會講）：陣列中每個桶指向一個鏈結串列，當鏈結串列長度超過一定閾值時，又可能被轉化為樹（通常為紅黑樹）。

從儲存的角度來看，雜湊表的底層是陣列，其中每一個桶槽位可能包含一個值，也可能包含一個鏈結串列或一棵樹。因此，雜湊表可能同時包含線性資料結構（陣列、鏈結串列）和非線性資料結構（樹）。

Q：`char` 型別的長度是 1 位元組嗎？

`char` 型別的長度由程式語言採用的編碼方法決定。例如，Java、JavaScript、TypeScript、C# 都採用 UTF-16 編碼（儲存 Unicode 碼點），因此 `char` 型別的長度為 2 位元組。

Q：基於陣列實現的資料結構也稱“靜態資料結構”是否有歧義？堆疊也可以進行出堆疊和入堆疊等操作，這些操作都是“動態”的。

堆疊確實可以實現動態的資料操作，但資料結構仍然是“靜態”（長度不可變）的。儘管基於陣列的資料結構可以動態地新增或刪除元素，但它們的容量是固定的。如果資料量超出了預分配的大小，就需要建立一個新的更大的陣列，並將舊陣列的內容複製到新陣列中。

Q：在構建堆疊（佇列）的時候，未指定它的大小，為什麼它們是“靜態資料結構”呢？

在高階程式語言中，我們無須人工指定堆疊（佇列）的初始容量，這個工作由類別內部自動完成。例如，Java 的 `ArrayList` 的初始容量通常為 10。另外，擴容操作也是自動實現的。詳見後續的“串列”章節。

Q：原碼轉二補數的方法是“先取反後加 1”，那麼二補數轉原碼應該是逆運算“先減 1 後取反”，而二補數轉原碼也一樣可以透過“先取反後加 1”得到，這是為什麼呢？

這是因為原碼和二補數的相互轉換實際上是計算“補數”的過程。我們先給出補數的定義：假設 $a + b = c$，那麼我們稱 a 是 b 到 c 的補數，反之也稱 b 是 a 到 c 的補數。

給定一個 $n = 4$ 位長度的二進位制數 0010，如果將這個數字看作原碼（不考慮符號位），那麼它的二補數需透過“先取反後加 1”得到：

$$0010 \to 1101 \to 1110$$

我們會發現，原碼和二補數的和是 $0010 + 1110 = 10000$，也就是說，二補數 1110 是原碼 0010 到 10000 的“補數”。**這意味著上述“先取反後加 1”實際上是計算到 10000 的補數的過程**。

那麼，二補數 1110 到 10000 的“補數”是多少呢？我們依然可以用“先取反後加 1”得到它：

$$1110 \to 0001 \to 0010$$

換句話說，原碼和二補數互為對方到 10000 的“補數”，因此“原碼轉二補數”和“二補數轉原碼”可以用相同的操作（先取反後加 1）實現。

當然，我們也可以用逆運算來求二補數 1110 的原碼，即“先減 1 後取反”：

$$1110 \to 1101 \to 0010$$

總結來看，“先取反後加 1”和“先減 1 後取反”這兩種運算都是在計算到 10000 的補數，它們是等價的。

本質上看，“取反”操作實際上是求到 1111 的補數（因為恆有 `原碼 + 一補數 = 1111`）；而在一補數基礎上再加 1 得到的二補數，就是到 10000 的補數。

上述以 $n = 4$ 為例，其可被推廣至任意位數的二進位制數。

第 4 章　陣列與鏈結串列

Abstract

資料結構的世界如同一堵厚實的磚牆。

陣列的磚塊整齊排列，逐個緊貼。鏈結串列的磚塊分散各處，連線的藤蔓自由地穿梭於磚縫之間。

4.1　陣列

陣列（array）是一種線性資料結構，其將相同型別的元素儲存在連續的記憶體空間中。我們將元素在陣列中的位置稱為該元素的索引（index）。圖 4-1 展示了陣列的主要概念和儲存方式。

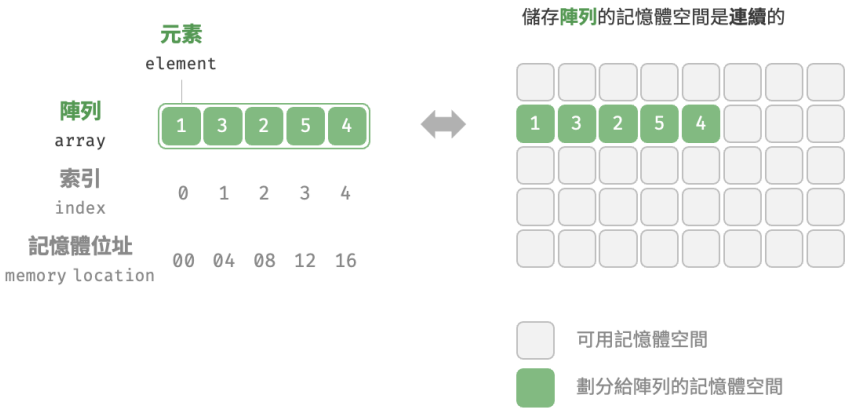


圖 4-1　陣列定義與儲存方式

4.1.1　陣列常用操作

1.　初始化陣列

我們可以根據需求選用陣列的兩種初始化方式：無初始值、給定初始值。在未指定初始值的情況下，大多數程式語言會將陣列元素初始化為 0：

```
// === File: array.rs ===

/* 初始化陣列 */
let arr: [i32; 5] = [0; 5]; // [0, 0, 0, 0, 0]
let slice: &[i32] = &[0; 5];
// 在 Rust 中，指定長度時（[i32; 5]）為陣列，不指定長度時（&[i32]）為切片
// 由於 Rust 的陣列被設計為在編譯期確定長度，因此只能使用常數來指定長度
// Vector 是 Rust 一般情況下用作動態陣列的型別
// 為了方便實現擴容 extend() 方法，以下將 vector 看作陣列（array）
let nums: Vec<i32> = vec![1, 3, 2, 5, 4];
```

2.　訪問元素

陣列元素被儲存在連續的記憶體空間中，這意味著計算陣列元素的記憶體位址非常容易。給定陣列記憶體位址（首元素記憶體位址）和某個元素的索引，我們可以使用圖 4-2 所示的公式計算得到該元素的記憶體位址，從而直接訪問該元素。

圖 4-2　陣列元素的記憶體位址計算

觀察圖 4-2，我們發現陣列首個元素的索引為 0，這似乎有些反直覺，因為從 1 開始計數會更自然。但從位址計算公式的角度看，**索引本質上是記憶體位址的偏移量**。首個元素的位址偏移量是 0，因此它的索引為 0 是合理的。

在陣列中訪問元素非常高效，我們可以在 $O(1)$ 時間內隨機訪問陣列中的任意一個元素。

```rust
// === File: array.rs ===

/* 隨機訪問元素 */
fn random_access(nums: &[i32]) -> i32 {
    // 在區間 [0, nums.len()) 中隨機抽取一個數字
    let random_index = rand::thread_rng().gen_range(0..nums.len());
    // 獲取並返回隨機元素
    let random_num = nums[random_index];
    random_num
}
```

3. 插入元素

陣列元素在記憶體中是“緊挨著的”，它們之間沒有空間再存放任何資料。如圖 4-3 所示，如果想在陣列中間插入一個元素，則需要將該元素之後的所有元素都向後移動一位，之後再把元素賦值給該索引。

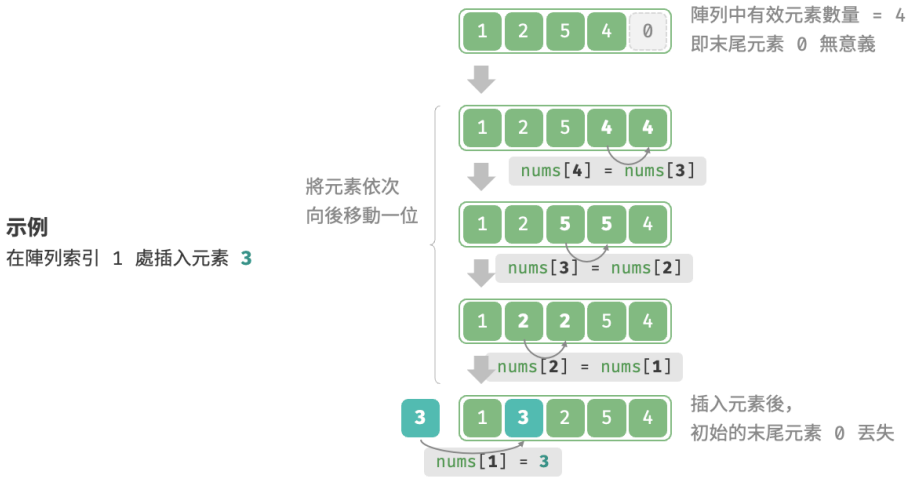


圖 4-3　陣列插入元素示例

值得注意的是，由於陣列的長度是固定的，因此插入一個元素必定會導致陣列尾部元素“丟失”。我們將這個問題的解決方案留在“串列”章節中討論。

```
// === File: array.rs ===

/* 在陣列的索引 index 處插入元素 num */
fn insert(nums: &mut [i32], num: i32, index: usize) {
    // 把索引 index 以及之後的所有元素向後移動一位
    for i in (index + 1..nums.len()).rev() {
        nums[i] = nums[i - 1];
    }
    // 將 num 賦給 index 處的元素
    nums[index] = num;
}
```

4. 刪除元素

同理，如圖 4-4 所示，若想刪除索引 i 處的元素，則需要把索引 i 之後的元素都向前移動一位。

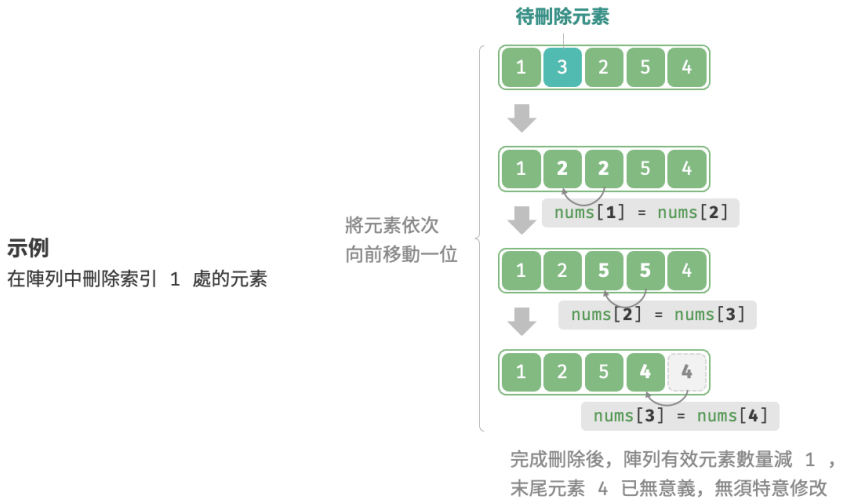


圖 4-4　陣列刪除元素示例

請注意，刪除元素完成後，原先末尾的元素變得“無意義”了，所以我們無須特意去修改它。

```
// === File: array.rs ===

/* 刪除索引 index 處的元素 */
fn remove(nums: &mut [i32], index: usize) {
    // 把索引 index 之後的所有元素向前移動一位
    for i in index..nums.len() - 1 {
        nums[i] = nums[i + 1];
    }
}
```

總的來看，陣列的插入與刪除操作有以下缺點。

- **時間複雜度高**：陣列的插入和刪除的平均時間複雜度均為 $O(n)$ ，其中 n 為陣列長度。
- **丟失元素**：由於陣列的長度不可變，因此在插入元素後，超出陣列長度範圍的元素會丟失。
- **記憶體浪費**：我們可以初始化一個比較長的陣列，只用前面一部分，這樣在插入資料時，丟失的末尾元素都是“無意義”的，但這樣做會造成部分記憶體空間浪費。

5. 走訪陣列

在大多數程式語言中，我們既可以透過索引走訪陣列，也可以直接走訪獲取陣列中的每個元素：

```
// === File: array.rs ===

/* 走訪陣列 */
fn traverse(nums: &[i32]) {
```

```
    let mut _count = 0;
    // 透過索引走訪陣列
    for i in 0..nums.len() {
        _count += nums[i];
    }
    // 直接走訪陣列元素
    for num in nums {
        _count += num;
    }
}
```

6. 查詢元素

在陣列中查詢指定元素需要走訪陣列，每輪判斷元素值是否匹配，若匹配則輸出對應索引。

因為陣列是線性資料結構，所以上述查詢操作被稱為“線性查詢”。

```
// === File: array.rs ===

/* 在陣列中查詢指定元素 */
fn find(nums: &[i32], target: i32) -> Option<usize> {
    for i in 0..nums.len() {
        if nums[i] == target {
            return Some(i);
        }
    }
    None
}
```

7. 擴容陣列

在複雜的系統環境中，程式難以保證陣列之後的記憶體空間是可用的，從而無法安全地擴展陣列容量。因此在大多數程式語言中，**陣列的長度是不可變的**。

如果我們希望擴容陣列，則需重新建立一個更大的陣列，然後把原陣列元素依次複製到新陣列。這是一個 $O(n)$ 的操作，在陣列很大的情況下非常耗時。程式碼如下所示：

```
// === File: array.rs ===

/* 擴展陣列長度 */
fn extend(nums: &[i32], enlarge: usize) -> Vec<i32> {
    // 初始化一個擴展長度後的陣列
    let mut res: Vec<i32> = vec![0; nums.len() + enlarge];
    // 將原陣列中的所有元素複製到新
    for i in 0..nums.len() {
```

```
        res[i] = nums[i];
    }
    // 返回擴展後的新陣列
    res
}
```

4.1.2　陣列的優點與侷限性

陣列儲存在連續的記憶體空間內，且元素型別相同。這種做法包含豐富的先驗資訊，系統可以利用這些資訊來最佳化資料結構的操作效率。

- **空間效率高**：陣列為資料分配了連續的記憶體塊，無須額外的結構開銷。
- **支持隨機訪問**：陣列允許在 $O(1)$ 時間內訪問任何元素。
- **快取區域性**：當訪問陣列元素時，計算機不僅會載入它，還會快取其周圍的其他資料，從而藉助高速快取來提升後續操作的執行速度。

連續空間儲存是一把雙刃劍，其存在以下侷限性。

- **插入與刪除效率低**：當陣列中元素較多時，插入與刪除操作需要移動大量的元素。
- **長度不可變**：陣列在初始化後長度就固定了，擴容陣列需要將所有資料複製到新陣列，開銷很大。
- **空間浪費**：如果陣列分配的大小超過實際所需，那麼多餘的空間就被浪費了。

4.1.3　陣列典型應用

陣列是一種基礎且常見的資料結構，既頻繁應用在各類演算法之中，也可用於實現各種複雜資料結構。

- **隨機訪問**：如果我們想隨機抽取一些樣本，那麼可以用陣列儲存，並生成一個隨機序列，根據索引實現隨機抽樣。
- **排序和搜尋**：陣列是排序和搜尋演算法最常用的資料結構。快速排序、合併排序、二分搜尋等都主要在陣列上進行。
- **查詢表**：當需要快速查詢一個元素或其對應關係時，可以使用陣列作為查詢表。假如我們想實現字元到 ASCII 碼的對映，則可以將字元的 ASCII 碼值作為索引，對應的元素存放在陣列中的對應位置。
- **機器學習**：神經網路中大量使用了向量、矩陣、張量之間的線性代數運算，這些資料都是以陣列的形式構建的。陣列是神經網路程式設計中最常使用的資料結構。
- **資料結構實現**：陣列可以用於實現堆疊、佇列、雜湊表、堆積、圖等資料結構。例如，圖的鄰接矩陣表示實際上是一個二維陣列。

4.2　鏈結串列

記憶體空間是所有程式的公共資源，在一個複雜的系統執行環境下，空閒的記憶體空間可能散落在記憶體各處。我們知道，儲存陣列的記憶體空間必須是連續的，而當陣列非常大時，記憶體可能無法提供如此大的連續空間。此時鏈結串列的靈活性優勢就體現出來了。

鏈結串列（linked list）是一種線性資料結構，其中的每個元素都是一個節點物件，各個節點透過“引用”相連線。引用記錄了下一個節點的記憶體位址，透過它可以從當前節點訪問到下一個節點。

鏈結串列的設計使得各個節點可以分散儲存在記憶體各處，它們的記憶體位址無須連續。

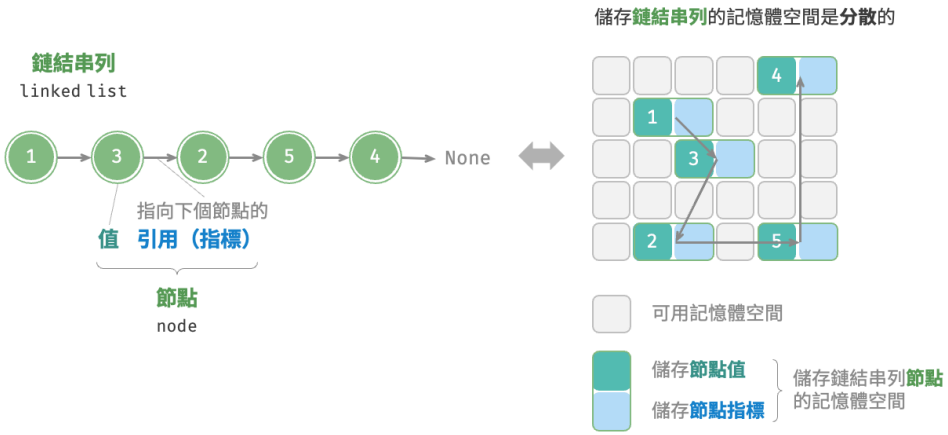


圖 4-5　鏈結串列定義與儲存方式

觀察圖 4-5 ，鏈結串列的組成單位是節點（node）物件。每個節點都包含兩項資料：節點的“值”和指向下一節點的“引用”。

- 鏈結串列的首個節點被稱為“頭節點”，最後一個節點被稱為“尾節點”。
- 尾節點指向的是“空”，它在 Java、C++ 和 Python 中分別被記為 `null`、`nullptr` 和 `None` 。
- 在 C、C++、Go 和 Rust 等支持指標的語言中，上述“引用”應被替換為“指標”。

如以下程式碼所示，鏈結串列節點 `ListNode` 除了包含值，還需額外儲存一個引用（指標）。因此在相同資料量下，**鏈結串列比陣列佔用更多的記憶體空間**。

```rust
use std::rc::Rc;
use std::cell::RefCell;
/* 鏈結串列節點類別 */
#[derive(Debug)]
struct ListNode {
    val: i32, // 節點值
    next: Option<Rc<RefCell<ListNode>>>, // 指向下一節點的指標
}
```

4.2.1　鏈結串列常用操作

1.　初始化鏈結串列

建立鏈結串列分為兩步，第一步是初始化各個節點物件，第二步是構建節點之間的引用關係。初始化完成後，我們就可以從鏈結串列的頭節點出發，透過引用指向 `next` 依次訪問所有節點。

```
// === File: linked_list.rs ===

/* 初始化鏈結串列 1 -> 3 -> 2 -> 5 -> 4 */
// 初始化各個節點
let n0 = Rc::new(RefCell::new(ListNode { val: 1, next: None }));
let n1 = Rc::new(RefCell::new(ListNode { val: 3, next: None }));
let n2 = Rc::new(RefCell::new(ListNode { val: 2, next: None }));
let n3 = Rc::new(RefCell::new(ListNode { val: 5, next: None }));
let n4 = Rc::new(RefCell::new(ListNode { val: 4, next: None }));

// 構建節點之間的引用
n0.borrow_mut().next = Some(n1.clone());
n1.borrow_mut().next = Some(n2.clone());
n2.borrow_mut().next = Some(n3.clone());
n3.borrow_mut().next = Some(n4.clone());
```

陣列整體是一個變數，比如陣列 `nums` 包含元素 `nums[0]` 和 `nums[1]` 等，而鏈結串列是由多個獨立的節點物件組成的。**我們通常將頭節點當作鏈結串列的代稱**，比如以上程式碼中的鏈結串列可記作鏈結串列 `n0` 。

2. 插入節點

在鏈結串列中插入節點非常容易。如圖 4-6 所示，假設我們想在相鄰的兩個節點 `n0` 和 `n1` 之間插入一個新節點 `P` ，**則只需改變兩個節點引用（指標）即可**，時間複雜度為 $O(1)$ 。

相比之下，在陣列中插入元素的時間複雜度為 $O(n)$ ，在大資料量下的效率較低。

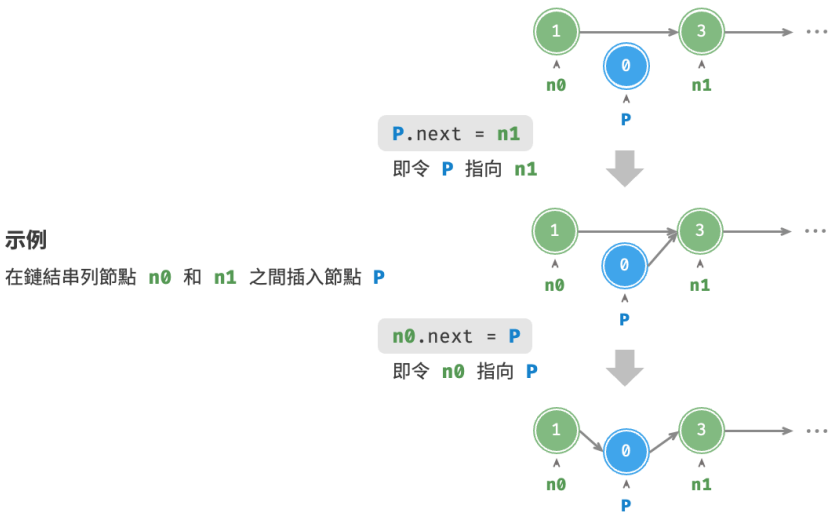


圖 4-6　鏈結串列插入節點示例

```rust
// === File: linked_list.rs ===

/* 在鏈結串列的節點 n0 之後插入節點 P */
#[allow(non_snake_case)]
pub fn insert<T>(n0: &Rc<RefCell<ListNode<T>>>, P: Rc<RefCell<ListNode<T>>>) {
    let n1 = n0.borrow_mut().next.take();
    P.borrow_mut().next = n1;
    n0.borrow_mut().next = Some(P);
}
```

3. 刪除節點

如圖 4-7 所示，在鏈結串列中刪除節點也非常方便，**只需改變一個節點的引用（指標）即可**。

請注意，儘管在刪除操作完成後節點 `P` 仍然指向 `n1`，但實際上走訪此鏈結串列已經無法訪問到 `P`，這意味著 `P` 已經不再屬於該鏈結串列了。

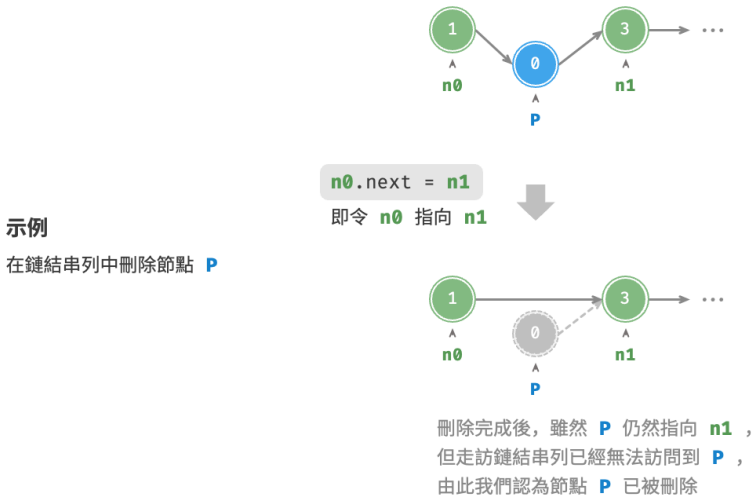


圖 4-7　鏈結串列刪除節點

```rust
// === File: linked_list.rs ===

/* 刪除鏈結串列的節點 n0 之後的首個節點 */
#[allow(non_snake_case)]
pub fn remove<T>(n0: &Rc<RefCell<ListNode<T>>>) {
    if n0.borrow().next.is_none() {
        return;
    };
    // n0 -> P -> n1
    let P = n0.borrow_mut().next.take();
```

```
    if let Some(node) = P {
        let n1 = node.borrow_mut().next.take();
        n0.borrow_mut().next = n1;
    }
}
```

4. 訪問節點

在鏈結串列中訪問節點的效率較低。如上一節所述，我們可以在 $O(1)$ 時間下訪問陣列中的任意元素。鏈結串列則不然，程式需要從頭節點出發，逐個向後走訪，直至找到目標節點。也就是說，訪問鏈結串列的第 i 個節點需要迴圈 $i - 1$ 輪，時間複雜度為 $O(n)$ 。

```
// === File: linked_list.rs ===

/* 訪問鏈結串列中索引為 index 的節點 */
pub fn access<T>(head: Rc<RefCell<ListNode<T>>>, index: i32) -> Rc<RefCell<ListNode<T>>> {
    if index <= 0 {
        return head;
    };
    if let Some(node) = &head.borrow().next {
        return access(node.clone(), index - 1);
    }

    return head;
}
```

5. 查詢節點

走訪鏈結串列，查詢其中值為 `target` 的節點，輸出該節點在鏈結串列中的索引。此過程也屬於線性查詢。程式碼如下所示：

```
// === File: linked_list.rs ===

/* 在鏈結串列中查詢值為 target 的首個節點 */
pub fn find<T: PartialEq>(head: Rc<RefCell<ListNode<T>>>, target: T, index: i32) -> i32 {
    if head.borrow().val == target {
        return index;
    };
    if let Some(node) = &head.borrow_mut().next {
        return find(node.clone(), target, index + 1);
    }
    return -1;
}
```

4.2.2　陣列 vs. 鏈結串列

表 4-1 總結了陣列和鏈結串列的各項特點並對比了操作效率。由於它們採用兩種相反的儲存策略，因此各種性質和操作效率也呈現對立的特點。

表 4-1　陣列與鏈結串列的效率對比

	陣列	鏈結串列
儲存方式	連續記憶體空間	分散記憶體空間
容量擴展	長度不可變	可靈活擴展
記憶體效率	元素佔用記憶體少、但可能浪費空間	元素佔用記憶體多
訪問元素	$O(1)$	$O(n)$
新增元素	$O(n)$	$O(1)$
刪除元素	$O(n)$	$O(1)$

4.2.3　常見鏈結串列型別

如圖 4-8 所示，常見的鏈結串列型別包括三種。

- **單向鏈結串列**：即前面介紹的普通鏈結串列。單向鏈結串列的節點包含值和指向下一節點的引用兩項資料。我們將首個節點稱為頭節點，將最後一個節點稱為尾節點，尾節點指向空 `None` 。
- **環形鏈結串列**：如果我們令單向鏈結串列的尾節點指向頭節點（首尾相接），則得到一個環形鏈結串列。在環形鏈結串列中，任意節點都可以視作頭節點。
- **雙向鏈結串列**：與單向鏈結串列相比，雙向鏈結串列記錄了兩個方向的引用。雙向鏈結串列的節點定義同時包含指向後繼節點（下一個節點）和前驅節點（上一個節點）的引用（指標）。相較於單向鏈結串列，雙向鏈結串列更具靈活性，可以朝兩個方向走訪鏈結串列，但相應地也需要佔用更多的記憶體空間。

```rust
use std::rc::Rc;
use std::cell::RefCell;

/* 雙向鏈結串列節點型別 */
#[derive(Debug)]
struct ListNode {
    val: i32, // 節點值
    next: Option<Rc<RefCell<ListNode>>>, // 指向後繼節點的指標
    prev: Option<Rc<RefCell<ListNode>>>, // 指向前驅節點的指標
}

/* 建構子 */
impl ListNode {
    fn new(val: i32) -> Self {
        ListNode {
```

```
            val,
            next: None,
            prev: None,
        }
    }
}
```

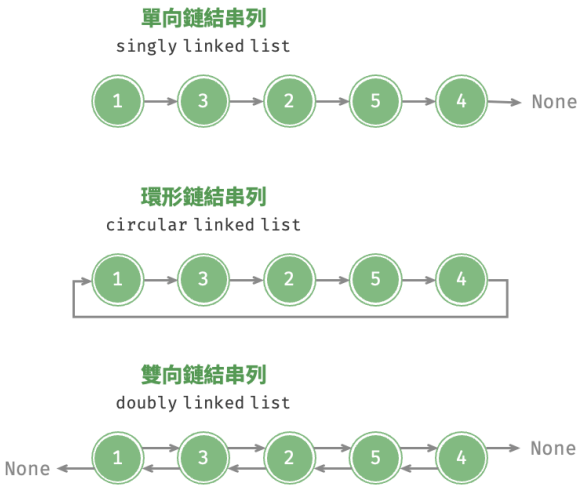


圖 4-8　常見鏈結串列種類

4.2.4　鏈結串列典型應用

單向鏈結串列通常用於實現堆疊、佇列、雜湊表和圖等資料結構。

- **堆疊與佇列**：當插入和刪除操作都在鏈結串列的一端進行時，它表現的特性為先進後出，對應堆疊；當插入操作在鏈結串列的一端進行，刪除操作在鏈結串列的另一端進行，它表現的特性為先進先出，對應佇列。
- **雜湊表**：鏈式位址是解決雜湊衝突的主流方案之一，在該方案中，所有衝突的元素都會被放到一個鏈結串列中。
- **圖**：鄰接表是表示圖的一種常用方式，其中圖的每個頂點都與一個鏈結串列相關聯，鏈結串列中的每個元素都代表與該頂點相連的其他頂點。

雙向鏈結串列常用於需要快速查詢前一個和後一個元素的場景。

- **高階資料結構**：比如在紅黑樹、B 樹中，我們需要訪問節點的父節點，這可以透過在節點中儲存一個指向父節點的引用來實現，類似於雙向鏈結串列。
- **瀏覽器歷史**：在網頁瀏覽器中，當用戶點選前進或後退按鈕時，瀏覽器需要知道使用者訪問過的前一個和後一個網頁。雙向鏈結串列的特性使得這種操作變得簡單。
- **LRU 演算法**：在快取淘汰（LRU）演算法中，我們需要快速找到最近最少使用的資料，以及支持快速新增和刪除節點。這時候使用雙向鏈結串列就非常合適。

環形鏈結串列常用於需要週期性操作的場景，比如作業系統的資源排程。

- **時間片輪轉排程演算法**：在作業系統中，時間片輪轉排程演算法是一種常見的 CPU 排程演算法，它需要對一組程序進行迴圈。每個程序被賦予一個時間片，當時間片用完時，CPU 將切換到下一個程序。這種迴圈操作可以透過環形鏈結串列來實現。
- **資料緩衝區**：在某些資料緩衝區的實現中，也可能會使用環形鏈結串列。比如在音訊、影片播放器中，資料流可能會被分成多個緩衝塊並放入一個環形鏈結串列，以便實現無縫播放。

4.3 串列

串列（list）是一個抽象的資料結構概念，它表示元素的有序集合，支持元素訪問、修改、新增、刪除和走訪等操作，無須使用者考慮容量限制的問題。串列可以基於鏈結串列或陣列實現。

- 鏈結串列天然可以看作一個串列，其支持元素增刪查改操作，並且可以靈活動態擴容。
- 陣列也支持元素增刪查改，但由於其長度不可變，因此只能看作一個具有長度限制的串列。

當使用陣列實現串列時，**長度不可變的性質會導致串列的實用性降低**。這是因為我們通常無法事先確定需要儲存多少資料，從而難以選擇合適的串列長度。若長度過小，則很可能無法滿足使用需求；若長度過大，則會造成記憶體空間浪費。

為解決此問題，我們可以使用動態陣列（dynamic array）來實現串列。它繼承了陣列的各項優點，並且可以在程式執行過程中進行動態擴容。

實際上，**許多程式語言中的標準庫提供的串列是基於動態陣列實現的**，例如 Python 中的 `list`、Java 中的 `ArrayList`、C++ 中的 `vector` 和 C# 中的 `List` 等。在接下來的討論中，我們將把“串列”和“動態陣列”視為等同的概念。

4.3.1 串列常用操作

1. 初始化串列

我們通常使用“無初始值”和“有初始值”這兩種初始化方法：

```
// === File: list.rs ===

/* 初始化串列 */
// 無初始值
let nums1: Vec<i32> = Vec::new();
// 有初始值
let nums: Vec<i32> = vec![1, 3, 2, 5, 4];
```

2. 訪問元素

串列本質上是陣列，因此可以在 $O(1)$ 時間內訪問和更新元素，效率很高。

```
// === File: list.rs ===

/* 訪問元素 */
let num: i32 = nums[1];  // 訪問索引 1 處的元素
/* 更新元素 */
nums[1] = 0;             // 將索引 1 處的元素更新為 0
```

3. 插入與刪除元素

相較於陣列，串列可以自由地新增與刪除元素。在串列尾部新增元素的時間複雜度為 $O(1)$ ，但插入和刪除元素的效率仍與陣列相同，時間複雜度為 $O(n)$ 。

```
// === File: list.rs ===

/* 清空串列 */
nums.clear();

/* 在尾部新增元素 */
nums.push(1);
nums.push(3);
nums.push(2);
nums.push(5);
nums.push(4);

/* 在中間插入元素 */
nums.insert(3, 6);  // 在索引 3 處插入數字 6

/* 刪除元素 */
nums.remove(3);    // 刪除索引 3 處的元素
```

4. 走訪串列

與陣列一樣，串列可以根據索引走訪，也可以直接走訪各元素。

```
// === File: list.rs ===

// 透過索引走訪串列
let mut _count = 0;
for i in 0..nums.len() {
    _count += nums[i];
}

// 直接走訪串列元素
_count = 0;
```

```
for num in &nums {
    _count += num;
}
```

5. 拼接串列

給定一個新串列 `nums1`，我們可以將其拼接到原串列的尾部。

```
// === File: list.rs ===

/* 拼接兩個串列 */
let nums1: Vec<i32> = vec![6, 8, 7, 10, 9];
nums.extend(nums1);
```

6. 排序串列

完成串列排序後，我們便可以使用在陣列類別演算法題中經常考查的“二分搜尋”和“雙指標”演算法。

```
// === File: list.rs ===

/* 排序串列 */
nums.sort(); // 排序後，串列元素從小到大排列
```

4.3.2 串列實現

許多程式語言內建了串列，例如 Java、C++、Python 等。它們的實現比較複雜，各個參數的設定也非常考究，例如初始容量、擴容倍數等。感興趣的讀者可以查閱原始碼進行學習。

為了加深對串列工作原理的理解，我們嘗試實現一個簡易版串列，包括以下三個重點設計。

- **初始容量**：選取一個合理的陣列初始容量。在本示例中，我們選擇 10 作為初始容量。
- **數量記錄**：宣告一個變數 `size`，用於記錄串列當前元素數量，並隨著元素插入和刪除實時更新。根據此變數，我們可以定位串列尾部，以及判斷是否需要擴容。
- **擴容機制**：若插入元素時串列容量已滿，則需要進行擴容。先根據擴容倍數建立一個更大的陣列，再將當前陣列的所有元素依次移動至新陣列。在本示例中，我們規定每次將陣列擴容至之前的 2 倍。

```
// === File: my_list.rs ===

/* 串列類別 */
#[allow(dead_code)]
struct MyList {
    arr: Vec<i32>,       // 陣列（儲存串列元素）
    capacity: usize,     // 串列容量
```

```
    size: usize,         // 串列長度（當前元素數量）
    extend_ratio: usize, // 每次串列擴容的倍數
}

#[allow(unused, unused_comparisons)]
impl MyList {
    /* 建構子 */
    pub fn new(capacity: usize) -> Self {
        let mut vec = Vec::new();
        vec.resize(capacity, 0);
        Self {
            arr: vec,
            capacity,
            size: 0,
            extend_ratio: 2,
        }
    }

    /* 獲取串列長度（當前元素數量）*/
    pub fn size(&self) -> usize {
        return self.size;
    }

    /* 獲取串列容量 */
    pub fn capacity(&self) -> usize {
        return self.capacity;
    }

    /* 訪問元素 */
    pub fn get(&self, index: usize) -> i32 {
        // 索引如果越界，則丟擲異常，下同
        if index >= self.size {
            panic!(" 索引越界")
        };
        return self.arr[index];
    }

    /* 更新元素 */
    pub fn set(&mut self, index: usize, num: i32) {
        if index >= self.size {
            panic!(" 索引越界")
        };
        self.arr[index] = num;
    }

    /* 在尾部新增元素 */
    pub fn add(&mut self, num: i32) {
```

```
        // 元素數量超出容量時，觸發擴容機制
        if self.size == self.capacity() {
            self.extend_capacity();
        }
        self.arr[self.size] = num;
        // 更新元素數量
        self.size += 1;
    }

    /* 在中間插入元素 */
    pub fn insert(&mut self, index: usize, num: i32) {
        if index >= self.size() {
            panic!(" 索引越界")
        };
        // 元素數量超出容量時，觸發擴容機制
        if self.size == self.capacity() {
            self.extend_capacity();
        }
        // 將索引 index 以及之後的元素都向後移動一位
        for j in (index..self.size).rev() {
            self.arr[j + 1] = self.arr[j];
        }
        self.arr[index] = num;
        // 更新元素數量
        self.size += 1;
    }

    /* 刪除元素 */
    pub fn remove(&mut self, index: usize) -> i32 {
        if index >= self.size() {
            panic!(" 索引越界")
        };
        let num = self.arr[index];
        // 將將索引 index 之後的元素都向前移動一位
        for j in (index..self.size - 1) {
            self.arr[j] = self.arr[j + 1];
        }
        // 更新元素數量
        self.size -= 1;
        // 返回被刪除的元素
        return num;
    }

    /* 串列擴容 */
    pub fn extend_capacity(&mut self) {
        // 新建一個長度為原陣列 extend_ratio 倍的新陣列，並將原陣列複製到新陣列
        let new_capacity = self.capacity * self.extend_ratio;
```

```
        self.arr.resize(new_capacity, 0);
        // 更新串列容量
        self.capacity = new_capacity;
    }

    /* 將串列轉換為陣列 */
    pub fn to_array(&mut self) -> Vec<i32> {
        // 僅轉換有效長度範圍內的串列元素
        let mut arr = Vec::new();
        for i in 0..self.size {
            arr.push(self.get(i));
        }
        arr
    }
}
```

4.4　記憶體與快取 *

在本章的前兩節中，我們探討了陣列和鏈結串列這兩種基礎且重要的資料結構，它們分別代表了“連續儲存”和“分散儲存”兩種物理結構。

實際上，**物理結構在很大程度上決定了程式對記憶體和快取的使用效率**，進而影響演算法程式的整體效能。

4.4.1　計算機儲存裝置

計算機中包括三種類型的儲存裝置：硬碟（hard disk）、記憶體（random-access memory, RAM）、快取（cache memory）。表 4-2 展示了它們在計算機系統中的不同角色和效能特點。

表 4-2　計算機的儲存裝置

	硬碟	記憶體	快取
用途	長期儲存資料，包括作業系統、程式、檔案等	臨時儲存當前執行的程式和正在處理的資料	儲存經常訪問的資料和指令，減少 CPU 訪問記憶體的次數
易失性	斷電後資料不會丟失	斷電後資料會丟失	斷電後資料會丟失
容量	較大，TB 級別	較小，GB 級別	非常小，MB 級別
速度	較慢，幾百到幾千 MB/s	較快，幾十 GB/s	非常快，幾十到幾百 GB/s
價格	較便宜，幾毛到幾元 / GB	較貴，幾十到幾百元 / GB	非常貴，隨 CPU 打包計價

我們可以將計算機儲存系統想象為圖 4-9 所示的金字塔結構。越靠近金字塔頂端的儲存裝置的速度越快、容量越小、成本越高。這種多層級的設計並非偶然，而是計算機科學家和工程師們經過深思熟慮的結果。

- **硬碟難以被記憶體取代**。首先，記憶體中的資料在斷電後會丟失，因此它不適合長期儲存資料；其次，記憶體的成本是硬碟的幾十倍，這使得它難以在消費者市場普及。
- **快取的大容量和高速度難以兼得**。隨著 L1、L2、L3 快取的容量逐步增大，其物理尺寸會變大，與 CPU 核心之間的物理距離會變遠，從而導致資料傳輸時間增加，元素訪問延遲變高。在當前技術下，多層級的快取結構是容量、速度和成本之間的最佳平衡點。

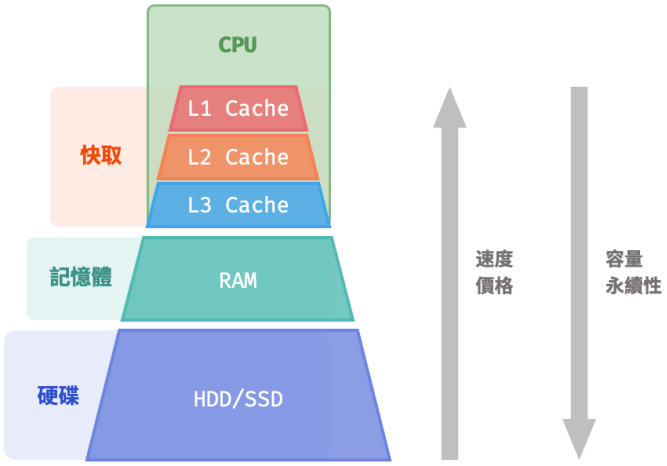


圖 4-9　計算機儲存系統

> **Tip**
> 計算機的儲存層次結構體現了速度、容量和成本三者之間的精妙平衡。實際上，這種權衡普遍存在於所有工業領域，它要求我們在不同的優勢和限制之間找到最佳平衡點。

總的來說，**硬碟用於長期儲存大量資料，記憶體用於臨時儲存程式執行中正在處理的資料，而快取則用於儲存經常訪問的資料和指令**，以提高程式執行效率。三者共同協作，確保計算機系統高效執行。

如圖 4-10 所示，在程式執行時，資料會從硬碟中被讀取到記憶體中，供 CPU 計算使用。快取可以看作 CPU 的一部分，**它透過智慧地從記憶體載入資料**，給 CPU 提供高速的資料讀取，從而顯著提升程式的執行效率，減少對較慢的記憶體的依賴。

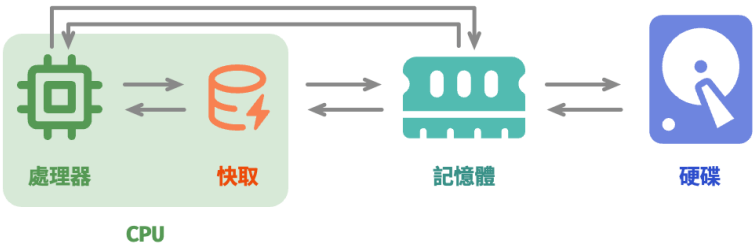


圖 4-10　硬碟、記憶體和快取之間的資料流通

4.4.2　資料結構的記憶體效率

在記憶體空間利用方面，陣列和鏈結串列各自具有優勢和侷限性。

一方面，**記憶體是有限的，且同一塊記憶體不能被多個程式共享**，因此我們希望資料結構能夠儘可能高效地利用空間。陣列的元素緊密排列，不需要額外的空間來儲存鏈結串列節點間的引用（指標），因此空間效率更高。然而，陣列需要一次性分配足夠的連續記憶體空間，這可能導致記憶體浪費，陣列擴容也需要額外的時間和空間成本。相比之下，鏈結串列以“節點”為單位進行動態記憶體分配和回收，提供了更大的靈活性。

另一方面，在程式執行時，**隨著反覆申請與釋放記憶體，空閒記憶體的碎片化程度會越來越高**，從而導致記憶體的利用效率降低。陣列由於其連續的儲存方式，相對不容易導致記憶體碎片化。相反，鏈結串列的元素是分散儲存的，在頻繁的插入與刪除操作中，更容易導致記憶體碎片化。

4.4.3　資料結構的快取效率

快取雖然在空間容量上遠小於記憶體，但它比記憶體快得多，在程式執行速度上起著至關重要的作用。由於快取的容量有限，只能儲存一小部分頻繁訪問的資料，因此當 CPU 嘗試訪問的資料不在快取中時，就會發生**快取未命中**（cache miss），此時 CPU 不得不從速度較慢的記憶體中載入所需資料。

顯然，**“快取未命中”越少，CPU 讀寫資料的效率就越高**，程式效能也就越好。我們將 CPU 從快取中成功獲取資料的比例稱為**快取命中率**（cache hit rate），這個指標通常用來衡量快取效率。

為了儘可能達到更高的效率，快取會採取以下資料載入機制。

- **快取行**：快取不是單個位元組地儲存與載入資料，而是以快取行為單位。相比於單個位元組的傳輸，快取行的傳輸形式更加高效。
- **預取機制**：處理器會嘗試預測資料訪問模式（例如順序訪問、固定步長跳躍訪問等），並根據特定模式將資料載入至快取之中，從而提升命中率。
- **空間區域性**：如果一個數據被訪問，那麼它附近的資料可能近期也會被訪問。因此，快取在載入某一資料時，也會載入其附近的資料，以提高命中率。
- **時間區域性**：如果一個數據被訪問，那麼它在不久的將來很可能再次被訪問。快取利用這一原理，透過保留最近訪問過的資料來提高命中率。

實際上，**陣列和鏈結串列對快取的利用效率是不同的**，主要體現在以下幾個方面。

- **佔用空間**：鏈結串列元素比陣列元素佔用空間更多，導致快取中容納的有效資料量更少。
- **快取行**：鏈結串列資料分散在記憶體各處，而快取是“按行載入”的，因此載入到無效資料的比例更高。
- **預取機制**：陣列比鏈結串列的資料訪問模式更具“可預測性”，即系統更容易猜出即將被載入的資料。
- **空間區域性**：陣列被儲存在集中的記憶體空間中，因此被載入資料附近的資料更有可能即將被訪問。

總體而言，**陣列具有更高的快取命中率，因此它在操作效率上通常優於鏈結串列**。這使得在解決演算法問題時，基於陣列實現的資料結構往往更受歡迎。

需要注意的是，**高快取效率並不意味著陣列在所有情況下都優於鏈結串列**。實際應用中選擇哪種資料結構，應根據具體需求來決定。例如，陣列和鏈結串列都可以實現“堆疊”資料結構（下一章會詳細介紹），但它們適用於不同場景。

- 在做演算法題時，我們會傾向於選擇基於陣列實現的堆疊，因為它提供了更高的操作效率和隨機訪問的能力，代價僅是需要預先為陣列分配一定的記憶體空間。

- 如果資料量非常大、動態性很高、堆疊的預期大小難以估計，那麼基於鏈結串列實現的堆疊更加合適。鏈結串列能夠將大量資料分散儲存於記憶體的不同部分，並且避免了陣列擴容產生的額外開銷。

4.5 小結

1. 重點回顧

- 陣列和鏈結串列是兩種基本的資料結構，分別代表資料在計算機記憶體中的兩種儲存方式：連續空間儲存和分散空間儲存。兩者的特點呈現出互補的特性。
- 陣列支持隨機訪問、佔用記憶體較少；但插入和刪除元素效率低，且初始化後長度不可變。
- 鏈結串列透過更改引用（指標）實現高效的節點插入與刪除，且可以靈活調整長度；但節點訪問效率低、佔用記憶體較多。常見的鏈結串列型別包括單向鏈結串列、環形鏈結串列、雙向鏈結串列。
- 串列是一種支持增刪查改的元素有序集合，通常基於動態陣列實現。它保留了陣列的優勢，同時可以靈活調整長度。
- 串列的出現大幅提高了陣列的實用性，但可能導致部分記憶體空間浪費。
- 程式執行時，資料主要儲存在記憶體中。陣列可提供更高的記憶體空間效率，而鏈結串列則在記憶體使用上更加靈活。
- 快取透過快取行、預取機制以及空間區域性和時間區域性等資料載入機制，為 CPU 提供快速資料訪問，顯著提升程式的執行效率。
- 由於陣列具有更高的快取命中率，因此它通常比鏈結串列更高效。在選擇資料結構時，應根據具體需求和場景做出恰當選擇。

2. Q & A

Q：陣列儲存在堆疊上和儲存在堆積上，對時間效率和空間效率是否有影響？

儲存在堆疊上和堆積上的陣列都被儲存在連續記憶體空間內，資料操作效率基本一致。然而，堆疊和堆積具有各自的特點，從而導致以下不同點。

1. 分配和釋放效率：堆疊是一塊較小的記憶體，分配由編譯器自動完成；而堆積記憶體相對更大，可以在程式碼中動態分配，更容易碎片化。因此，堆積上的分配和釋放操作通常比堆疊上的慢。
2. 大小限制：堆疊記憶體相對較小，堆積的大小一般受限於可用記憶體。因此堆積更加適合儲存大型陣列。
3. 靈活性：堆疊上的陣列的大小需要在編譯時確定，而堆積上的陣列的大小可以在執行時動態確定。

Q：為什麼陣列要求相同型別的元素，而在鏈結串列中卻沒有強調相同型別呢？

鏈結串列由節點組成，節點之間透過引用（指標）連線，各個節點可以儲存不同型別的資料，例如 `int`、`double`、`string`、`object` 等。

相對地，陣列元素則必須是相同型別的，這樣才能透過計算偏移量來獲取對應元素位置。例如，陣列同時包含 `int` 和 `long` 兩種型別，單個元素分別佔用 4 位元組和 8 位元組，此時就不能用以下公式計算偏移量了，因為陣列中包含了兩種“元素長度”。

```
# 元素記憶體位址 = 陣列記憶體位址（首元素記憶體位址） + 元素長度 * 元素索引
```

Q：刪除節點 `P` 後，是否需要把 `P.next` 設為 `None` 呢？

不修改 `P.next` 也可以。從該鏈結串列的角度看，從頭節點走訪到尾節點已經不會遇到 `P` 了。這意味著節點 `P` 已經從鏈結串列中刪除了，此時節點 `P` 指向哪裡都不會對該鏈結串列產生影響。

從資料結構與演算法（做題）的角度看，不斷開沒有關係，只要保證程式的邏輯是正確的就行。從標準庫的角度看，斷開更加安全、邏輯更加清晰。如果不斷開，假設被刪除節點未被正常回收，那麼它會影響後繼節點的記憶體回收。

Q：在鏈結串列中插入和刪除操作的時間複雜度是 $O(1)$ 。但是增刪之前都需要 $O(n)$ 的時間查詢元素，那為什麼時間複雜度不是 $O(n)$ 呢？

如果是先查詢元素、再刪除元素，時間複雜度確實是 $O(n)$ 。然而，鏈結串列的 $O(1)$ 增刪的優勢可以在其他應用上得到體現。例如，雙向佇列適合使用鏈結串列實現，我們維護一個指標變數始終指向頭節點、尾節點，每次插入與刪除操作都是 $O(1)$ 。

Q：圖“鏈結串列定義與儲存方式”中，淺藍色的儲存節點指標是佔用一塊記憶體位址嗎？還是和節點值各佔一半呢？

該示意圖只是定性表示，定量表示需要根據具體情況進行分析。

- 不同型別的節點值佔用的空間是不同的，比如 `int`、`long`、`double` 和例項物件等。
- 指標變數佔用的記憶體空間大小根據所使用的作業系統及編譯環境而定，大多為 8 位元組或 4 位元組。

Q：在串列末尾新增元素是否時時刻刻都為 $O(1)$ ？

如果新增元素時超出串列長度，則需要先擴容串列再新增。系統會申請一塊新的記憶體，並將原串列的所有元素搬運過去，這時候時間複雜度就會是 $O(n)$ 。

Q：“串列的出現極大地提高了陣列的實用性，但可能導致部分記憶體空間浪費”，這裡的空間浪費是指額外增加的變數如容量、長度、擴容倍數所佔的記憶體嗎？

這裡的空間浪費主要有兩方面含義：一方面，串列都會設定一個初始長度，我們不一定需要用這麼多；另一方面，為了防止頻繁擴容，擴容一般會乘以一個係數，比如 $\times 1.5$ 。這樣一來，也會出現很多空位，我們通常不能完全填滿它們。

Q：在 Python 中初始化 `n = [1, 2, 3]` 後，這 3 個元素的位址是相連的，但是初始化 `m = [2, 1, 3]` 會發現它們每個元素的 id 並不是連續的，而是分別跟 `n` 中的相同。這些元素的位址不連續，那麼 `m` 還是陣列嗎？

假如把串列元素換成鏈結串列節點 `n = [n1, n2, n3, n4, n5]` ，通常情況下這 5 個節點物件也分散儲存在記憶體各處。然而，給定一個串列索引，我們仍然可以在 $O(1)$ 時間內獲取節點記憶體位址，從而訪問到對應的節點。這是因為陣列中儲存的是節點的引用，而非節點本身。

與許多語言不同，Python 中的數字也被包裝為物件，串列中儲存的不是數字本身，而是對數字的引用。因此，我們會發現兩個陣列中的相同數字擁有同一個 id ，並且這些數字的記憶體位址無須連續。

Q：C++ STL 裡面的 `std::list` 已經實現了雙向鏈結串列，但好像一些演算法書上不怎麼直接使用它，是不是因為有什麼侷限性呢？

一方面，我們往往更青睞使用陣列實現演算法，而只在必要時才使用鏈結串列，主要有兩個原因。

- 空間開銷：由於每個元素需要兩個額外的指標（一個用於前一個元素，一個用於後一個元素），所以 `std::list` 通常比 `std::vector` 更佔用空間。

- 快取不友好: 由於資料不是連續存放的, 因此 `std::list` 對快取的利用率較低。一般情況下, `std::vector` 的效能會更好。

另一方面，必要使用鏈結串列的情況主要是二元樹和圖。堆疊和佇列往往會使用程式語言提供的 `stack` 和 `queue` ，而非鏈結串列。

Q：操作 `res = [[0]] * n` 生成了一個二維串列，其中每一個 `[0]` 都是獨立的嗎?

不是獨立的。此二維串列中，所有的 `[0]` 實際上是同一個物件的引用。如果我們修改其中一個元素，會發現所有的對應元素都會隨之改變。

如果希望二維串列中的每個 `[0]` 都是獨立的，可以使用 `res = [[0] for _ in range(n)]` 來實現。這種方式的原理是初始化了 n 個獨立的 `[0]` 串列物件。

Q：操作 `res = [0] * n` 生成了一個串列，其中每一個整數 0 都是獨立的嗎?

在該串列中，所有整數 0 都是同一個物件的引用。這是因為 Python 對小整數（通常是 -5 到 256）採用了快取池機制，以便最大化物件複用，從而提升效能。

雖然它們指向同一個物件，但我們仍然可以獨立修改串列中的每個元素，這是因為 Python 的整數是“不可變物件”。當我們修改某個元素時，實際上是切換為另一個物件的引用，而不是改變原有物件本身。

然而，當串列元素是“可變物件”時（例如串列、字典或類別例項等），修改某個元素會直接改變該物件本身，所有引用該物件的元素都會產生相同變化。

第 5 章　堆疊與佇列

Abstract

堆疊如同疊貓貓，而佇列就像貓貓排隊。

兩者分別代表先入後出和先入先出的邏輯關係。

5.1　堆疊

堆疊（stack）是一種遵循先入後出邏輯的線性資料結構。

我們可以將堆疊類比為桌面上的一疊盤子，如果想取出底部的盤子，則需要先將上面的盤子依次移走。我們將盤子替換為各種型別的元素（如整數、字元、物件等），就得到了堆疊這種資料結構。

如圖 5-1 所示，我們把堆積疊元素的頂部稱為“堆疊頂”，底部稱為“堆疊底”。將把元素新增到堆疊頂的操作叫作“入堆疊”，刪除堆疊頂元素的操作叫作“出堆疊”。

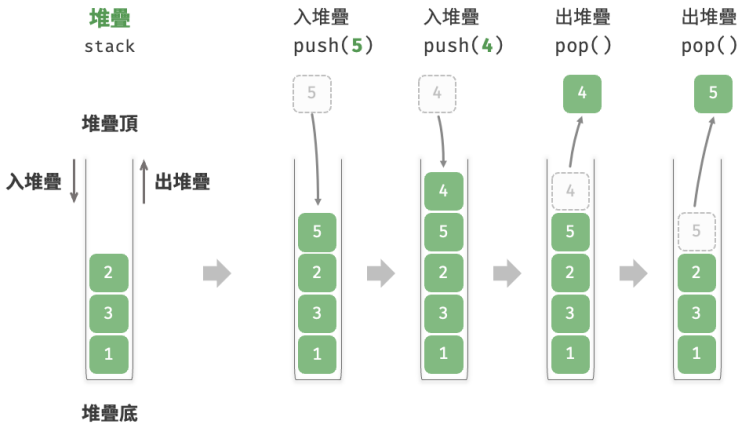


圖 5-1　堆疊的先入後出規則

5.1.1　堆疊的常用操作

堆疊的常用操作如表 5-1 所示，具體的方法名需要根據所使用的程式語言來確定。在此，我們以常見的 `push()`、`pop()`、`peek()` 命名為例。

表 5-1　堆疊的操作效率

方法	描述	時間複雜度
`push()`	元素入堆疊（新增至堆疊頂）	$O(1)$
`pop()`	堆疊頂元素出堆疊	$O(1)$
`peek()`	訪問堆疊頂元素	$O(1)$

通常情況下，我們可以直接使用程式語言內建的堆疊類別。然而，某些語言可能沒有專門提供堆疊類別，這時我們可以將該語言的“陣列”或“鏈結串列”當作堆疊來使用，並在程式邏輯上忽略與堆疊無關的操作。

```rust
// === File: stack.rs ===

/* 初始化堆疊 */
// 把 Vec 當作堆疊來使用
let mut stack: Vec<i32> = Vec::new();

/* 元素入堆疊 */
stack.push(1);
stack.push(3);
stack.push(2);
stack.push(5);
stack.push(4);

/* 訪問堆疊頂元素 */
let top = stack.last().unwrap();

/* 元素出堆疊 */
let pop = stack.pop().unwrap();

/* 獲取堆疊的長度 */
let size = stack.len();

/* 判斷是否為空 */
let is_empty = stack.is_empty();
```

5.1.2　堆疊的實現

為了深入瞭解堆疊的執行機制，我們來嘗試自己實現一個堆疊類別。

堆疊遵循先入後出的原則，因此我們只能在堆疊頂新增或刪除元素。然而，陣列和鏈結串列都可以在任意位置新增和刪除元素，**因此堆疊可以視為一種受限制的陣列或鏈結串列**。換句話說，我們可以“遮蔽”陣列或鏈結串列的部分無關操作，使其對外表現的邏輯符合堆疊的特性。

1.　基於鏈結串列的實現

使用鏈結串列實現堆疊時，我們可以將鏈結串列的頭節點視為堆疊頂，尾節點視為堆疊底。

如圖 5-2 所示，對於入堆疊操作，我們只需將元素插入鏈結串列頭部，這種節點插入方法被稱為“頭插法”。而對於出堆疊操作，只需將頭節點從鏈結串列中刪除即可。

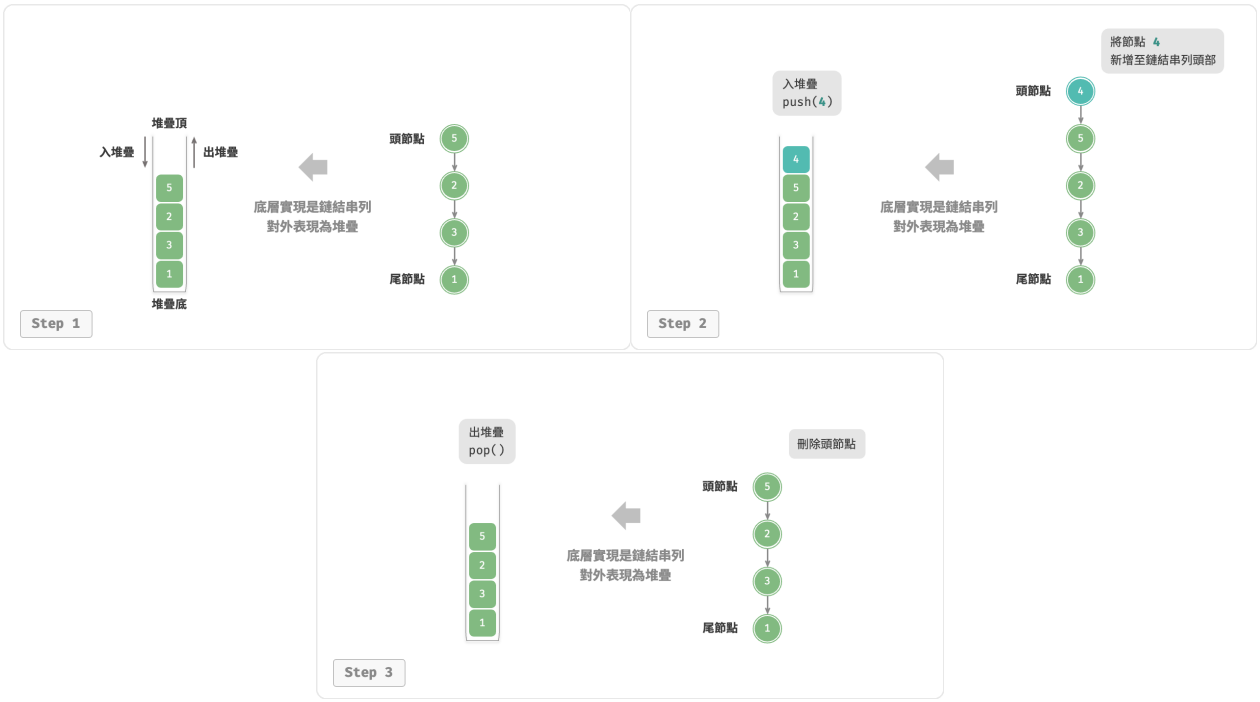


圖 5-2　基於鏈結串列實現堆疊的入堆疊出堆疊操作

以下是基於鏈結串列實現堆疊的示例程式碼：

```
// === File: linkedlist_stack.rs ===

/* 基於鏈結串列實現的堆疊 */
#[allow(dead_code)]
pub struct LinkedListStack<T> {
    stack_peek: Option<Rc<RefCell<ListNode<T>>>>, // 將頭節點作為堆疊頂
    stk_size: usize,                              // 堆疊的長度
}

impl<T: Copy> LinkedListStack<T> {
    pub fn new() -> Self {
        Self {
            stack_peek: None,
            stk_size: 0,
        }
    }

    /* 獲取堆疊的長度 */
    pub fn size(&self) -> usize {
        return self.stk_size;
    }

    /* 判斷堆疊是否為空 */
```

```
    pub fn is_empty(&self) -> bool {
        return self.size() == 0;
    }

    /* 入堆疊 */
    pub fn push(&mut self, num: T) {
        let node = ListNode::new(num);
        node.borrow_mut().next = self.stack_peek.take();
        self.stack_peek = Some(node);
        self.stk_size += 1;
    }

    /* 出堆疊 */
    pub fn pop(&mut self) -> Option<T> {
        self.stack_peek.take().map(|old_head| {
            match old_head.borrow_mut().next.take() {
                Some(new_head) => {
                    self.stack_peek = Some(new_head);
                }
                None => {
                    self.stack_peek = None;
                }
            }
            self.stk_size -= 1;
            Rc::try_unwrap(old_head).ok().unwrap().into_inner().val
        })
    }

    /* 訪問堆疊頂元素 */
    pub fn peek(&self) -> Option<&Rc<RefCell<ListNode<T>>>> {
        self.stack_peek.as_ref()
    }

    /* 將 List 轉化為 Array 並返回 */
    pub fn to_array(&self, head: Option<&Rc<RefCell<ListNode<T>>>>) -> Vec<T> {
        if let Some(node) = head {
            let mut nums = self.to_array(node.borrow().next.as_ref());
            nums.push(node.borrow().val);
            return nums;
        }
        return Vec::new();
    }
}
```

2. 基於陣列的實現

使用陣列實現堆疊時，我們可以將陣列的尾部作為堆疊頂。如圖 5-3 所示，入堆疊與出堆疊操作分別對應在陣列尾部新增元素與刪除元素，時間複雜度都為 $O(1)$ 。

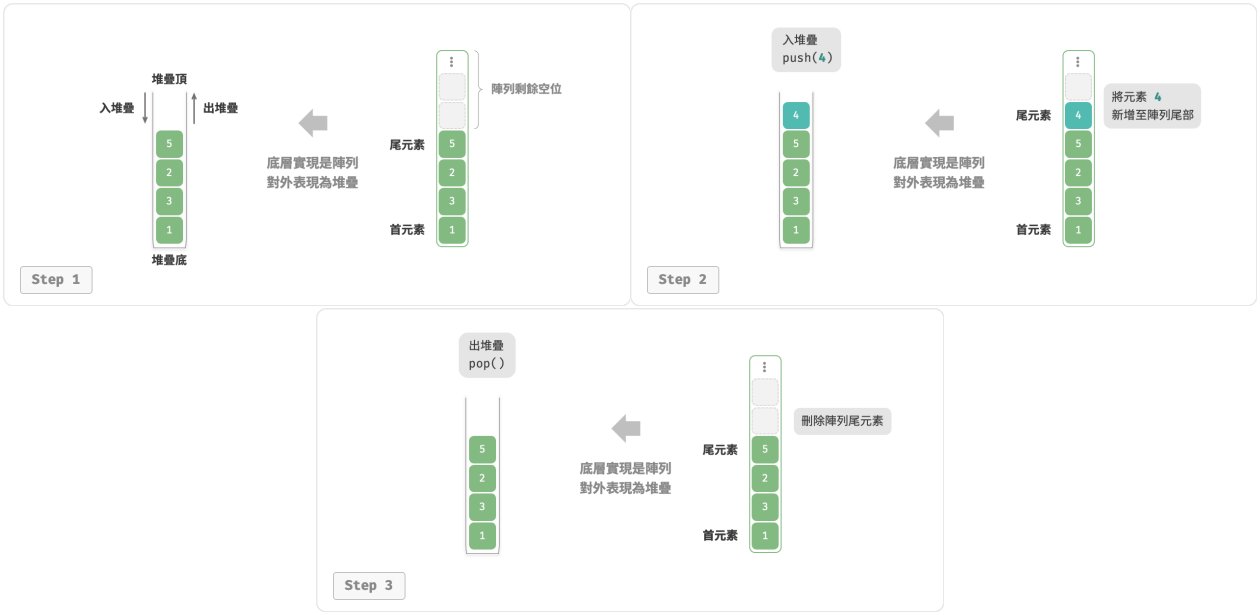


圖 5-3　基於陣列實現堆疊的入堆疊出堆疊操作

由於入堆疊的元素可能會源源不斷地增加，因此我們可以使用動態陣列，這樣就無須自行處理陣列擴容問題。以下為示例程式碼：

```
// === File: array_stack.rs ===

/* 基於陣列實現的堆疊 */
struct ArrayStack<T> {
    stack: Vec<T>,
}

impl<T> ArrayStack<T> {
    /* 初始化堆疊 */
    fn new() -> ArrayStack<T> {
        ArrayStack::<T> {
            stack: Vec::<T>::new(),
        }
    }

    /* 獲取堆疊的長度 */
    fn size(&self) -> usize {
        self.stack.len()
    }
```

```
    /* 判斷堆疊是否為空 */
    fn is_empty(&self) -> bool {
        self.size() == 0
    }

    /* 入堆疊 */
    fn push(&mut self, num: T) {
        self.stack.push(num);
    }

    /* 出堆疊 */
    fn pop(&mut self) -> Option<T> {
        self.stack.pop()
    }

    /* 訪問堆疊頂元素 */
    fn peek(&self) -> Option<&T> {
        if self.is_empty() {
            panic!(" 堆疊為空")
        };
        self.stack.last()
    }

    /* 返回 &Vec */
    fn to_array(&self) -> &Vec<T> {
        &self.stack
    }
}
```

5.1.3　兩種實現對比

支持操作

兩種實現都支持堆疊定義中的各項操作。陣列實現額外支持隨機訪問，但這已超出了堆疊的定義範疇，因此一般不會用到。

時間效率

在基於陣列的實現中，入堆疊和出堆疊操作都在預先分配好的連續記憶體中進行，具有很好的快取本地性，因此效率較高。然而，如果入堆疊時超出陣列容量，會觸發擴容機制，導致該次入堆疊操作的時間複雜度變為 $O(n)$ 。

在基於鏈結串列的實現中，鏈結串列的擴容非常靈活，不存在上述陣列擴容時效率降低的問題。但是，入堆疊操作需要初始化節點物件並修改指標，因此效率相對較低。不過，如果入堆疊元素本身就是節點物件，那麼可以省去初始化步驟，從而提高效率。

綜上所述，當入堆疊與出堆疊操作的元素是基本資料型別時，例如 `int` 或 `double` ，我們可以得出以下結論。

- 基於陣列實現的堆疊在觸發擴容時效率會降低，但由於擴容是低頻操作，因此平均效率更高。
- 基於鏈結串列實現的堆疊可以提供更加穩定的效率表現。

空間效率

在初始化串列時，系統會為串列分配“初始容量”，該容量可能超出實際需求；並且，擴容機制通常是按照特定倍率（例如 2 倍）進行擴容的，擴容後的容量也可能超出實際需求。因此，**基於陣列實現的堆疊可能造成一定的空間浪費**。

然而，由於鏈結串列節點需要額外儲存指標，**因此鏈結串列節點佔用的空間相對較大**。

綜上，我們不能簡單地確定哪種實現更加節省記憶體，需要針對具體情況進行分析。

5.1.4　堆疊的典型應用

- **瀏覽器中的後退與前進、軟體中的撤銷與反撤銷**。每當我們開啟新的網頁，瀏覽器就會對上一個網頁執行入堆疊，這樣我們就可以通過後退操作回到上一個網頁。後退操作實際上是在執行出堆疊。如果要同時支持後退和前進，那麼需要兩個堆疊來配合實現。
- **程式記憶體管理**。每次呼叫函式時，系統都會在堆疊頂新增一個堆疊幀，用於記錄函式的上下文資訊。在遞迴函式中，向下遞推階段會不斷執行入堆疊操作，而向上回溯階段則會不斷執行出堆疊操作。

5.2　佇列

佇列（queue）是一種遵循先入先出規則的線性資料結構。顧名思義，佇列模擬了排隊現象，即新來的人不斷加入佇列尾部，而位於佇列頭部的人逐個離開。

如圖 5-4 所示，我們將佇列頭部稱為“佇列首”，尾部稱為“佇列尾”，將把元素加入列尾的操作稱為“入列”，刪除佇列首元素的操作稱為“出列”。

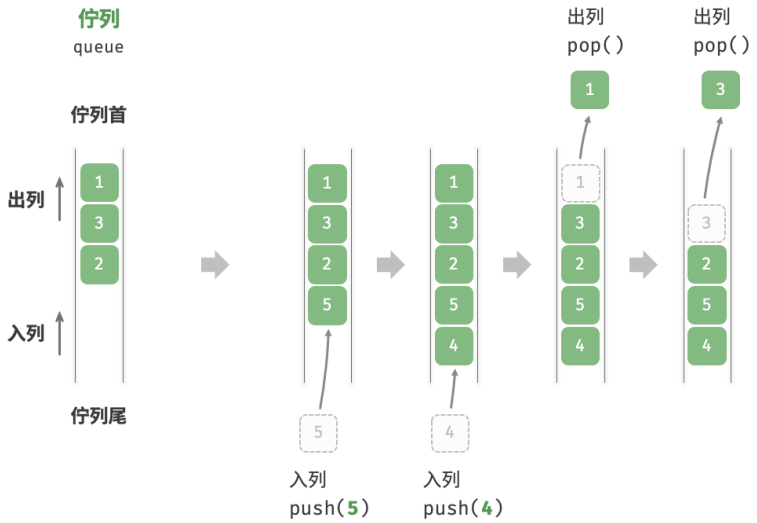


圖 5-4　佇列的先入先出規則

5.2.1　佇列常用操作

佇列的常見操作如表 5-2 所示。需要注意的是，不同程式語言的方法名稱可能會有所不同。我們在此採用與堆疊相同的方法命名。

表 5-2　佇列操作效率

方法名	描述	時間複雜度
`push()`	元素入列，即將元素新增至佇列尾	$O(1)$
`pop()`	佇列首元素出列	$O(1)$
`peek()`	訪問佇列首元素	$O(1)$

我們可以直接使用程式語言中現成的佇列類別：

```rust
// === File: queue.rs ===

/* 初始化雙向佇列 */
// 在 Rust 中使用雙向佇列作為普通佇列來使用
let mut deque: VecDeque<u32> = VecDeque::new();

/* 元素入列 */
deque.push_back(1);
deque.push_back(3);
deque.push_back(2);
deque.push_back(5);
deque.push_back(4);

/* 訪問佇列首元素 */
if let Some(front) = deque.front() {
}

/* 元素出列 */
if let Some(pop) = deque.pop_front() {
}

/* 獲取佇列的長度 */
let size = deque.len();

/* 判斷佇列是否為空 */
let is_empty = deque.is_empty();
```

5.2.2　佇列實現

為了實現佇列，我們需要一種資料結構，可以在一端新增元素，並在另一端刪除元素，鏈結串列和陣列都符合要求。

1.　基於鏈結串列的實現

如圖 5-5 所示，我們可以將鏈結串列的“頭節點”和“尾節點”分別視為“佇列首”和“佇列尾”，規定佇列尾僅可新增節點，佇列首僅可刪除節點。

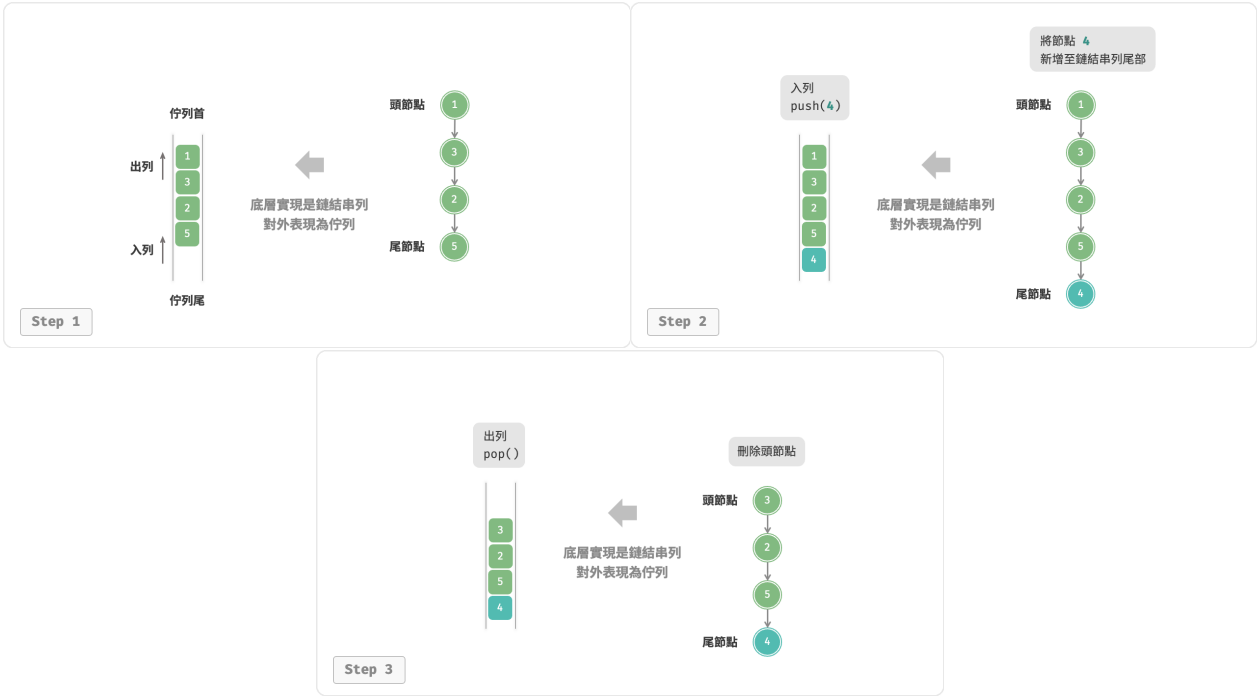


圖 5-5　基於鏈結串列實現佇列的入列出列操作

以下是用鏈結串列實現佇列的程式碼：

```
// === File: linkedlist_queue.rs ===

/* 基於鏈結串列實現的佇列 */
#[allow(dead_code)]
pub struct LinkedListQueue<T> {
    front: Option<Rc<RefCell<ListNode<T>>>>, // 頭節點 front
    rear: Option<Rc<RefCell<ListNode<T>>>>,  // 尾節點 rear
    que_size: usize,                         // 佇列的長度
}

impl<T: Copy> LinkedListQueue<T> {
    pub fn new() -> Self {
```

```
        Self {
            front: None,
            rear: None,
            que_size: 0,
        }
    }

    /* 獲取佇列的長度 */
    pub fn size(&self) -> usize {
        return self.que_size;
    }

    /* 判斷佇列是否為空 */
    pub fn is_empty(&self) -> bool {
        return self.size() == 0;
    }

    /* 入列 */
    pub fn push(&mut self, num: T) {
        // 在尾節點後新增 num
        let new_rear = ListNode::new(num);
        match self.rear.take() {
            // 如果佇列不為空，則將該節點新增到尾節點後
            Some(old_rear) => {
                old_rear.borrow_mut().next = Some(new_rear.clone());
                self.rear = Some(new_rear);
            }
            // 如果佇列為空，則令頭、尾節點都指向該節點
            None => {
                self.front = Some(new_rear.clone());
                self.rear = Some(new_rear);
            }
        }
        self.que_size += 1;
    }

    /* 出列 */
    pub fn pop(&mut self) -> Option<T> {
        self.front.take().map(|old_front| {
            match old_front.borrow_mut().next.take() {
                Some(new_front) => {
                    self.front = Some(new_front);
                }
                None => {
                    self.rear.take();
                }
            }
```

```
            self.que_size -= 1;
            Rc::try_unwrap(old_front).ok().unwrap().into_inner().val
        })
    }

    /* 訪問佇列首元素 */
    pub fn peek(&self) -> Option<&Rc<RefCell<ListNode<T>>>> {
        self.front.as_ref()
    }

    /* 將鏈結串列轉化為 Array 並返回 */
    pub fn to_array(&self, head: Option<&Rc<RefCell<ListNode<T>>>>) -> Vec<T> {
        if let Some(node) = head {
            let mut nums = self.to_array(node.borrow().next.as_ref());
            nums.insert(0, node.borrow().val);
            return nums;
        }
        return Vec::new();
    }
}
```

2. 基於陣列的實現

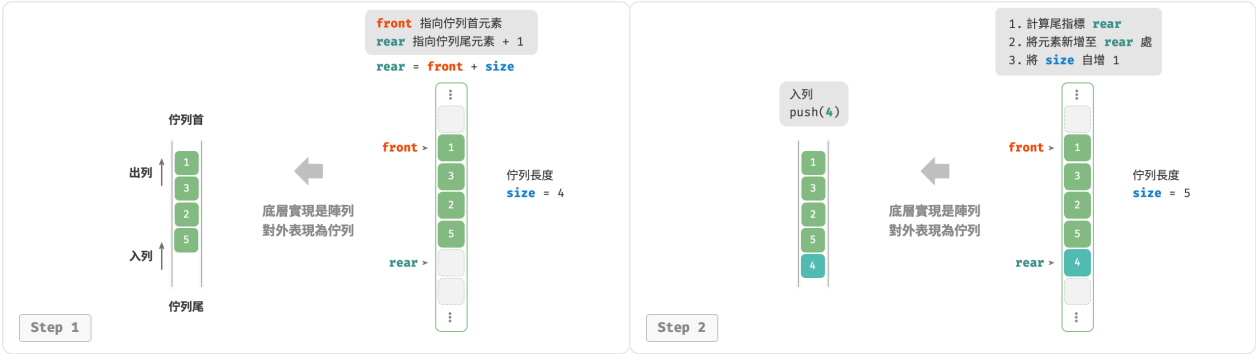
在陣列中刪除首元素的時間複雜度為 $O(n)$，這會導致出列操作效率較低。然而，我們可以採用以下巧妙方法來避免這個問題。

我們可以使用一個變數 `front` 指向佇列首元素的索引，並維護一個變數 `size` 用於記錄佇列長度。定義 `rear = front + size`，這個公式計算出的 `rear` 指向佇列尾元素之後的下一個位置。

基於此設計，**陣列中包含元素的有效區間為 `[front, rear - 1]`**，各種操作的實現方法如圖 5-6 所示。

- 入列操作：將輸入元素賦值給 `rear` 索引處，並將 `size` 增加 1 。
- 出列操作：只需將 `front` 增加 1 ，並將 `size` 減少 1 。

可以看到，入列和出列操作都只需進行一次操作，時間複雜度均為 $O(1)$ 。



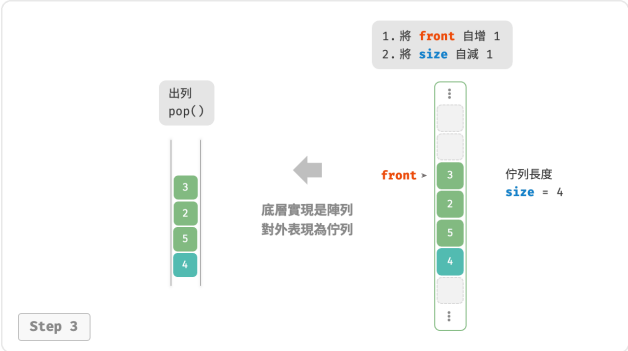


圖 5-6　基於陣列實現佇列的入列出列操作

你可能會發現一個問題：在不斷進行入列和出列的過程中，`front` 和 `rear` 都在向右移動，**當它們到達陣列尾部時就無法繼續移動了**。為了解決此問題，我們可以將陣列視為首尾相接的“環形陣列”。

對於環形陣列，我們需要讓 `front` 或 `rear` 在越過陣列尾部時，直接回到陣列頭部繼續走訪。這種週期性規律可以透過“取餘操作”來實現，程式碼如下所示：

```
// === File: array_queue.rs ===

/* 基於環形陣列實現的佇列 */
struct ArrayQueue {
    nums: Vec<i32>,    // 用於儲存佇列元素的陣列
    front: i32,        // 佇列首指標，指向佇列首元素
    que_size: i32,     // 佇列長度
    que_capacity: i32, // 佇列容量
}

impl ArrayQueue {
    /* 建構子 */
    fn new(capacity: i32) -> ArrayQueue {
        ArrayQueue {
            nums: vec![0; capacity as usize],
            front: 0,
            que_size: 0,
            que_capacity: capacity,
        }
    }

    /* 獲取佇列的容量 */
    fn capacity(&self) -> i32 {
        self.que_capacity
    }

    /* 獲取佇列的長度 */
    fn size(&self) -> i32 {
        self.que_size
```

```rust
}

/* 判斷佇列是否為空 */
fn is_empty(&self) -> bool {
    self.que_size == 0
}

/* 入列 */
fn push(&mut self, num: i32) {
    if self.que_size == self.capacity() {
        println!(" 佇列已滿");
        return;
    }
    // 計算佇列尾指標，指向佇列尾索引 + 1
    // 透過取餘操作實現 rear 越過陣列尾部後回到頭部
    let rear = (self.front + self.que_size) % self.que_capacity;
    // 將 num 新增至佇列尾
    self.nums[rear as usize] = num;
    self.que_size += 1;
}

/* 出列 */
fn pop(&mut self) -> i32 {
    let num = self.peek();
    // 佇列首指標向後移動一位，若越過尾部，則返回到陣列頭部
    self.front = (self.front + 1) % self.que_capacity;
    self.que_size -= 1;
    num
}

/* 訪問佇列首元素 */
fn peek(&self) -> i32 {
    if self.is_empty() {
        panic!("index out of bounds");
    }
    self.nums[self.front as usize]
}

/* 返回陣列 */
fn to_vector(&self) -> Vec<i32> {
    let cap = self.que_capacity;
    let mut j = self.front;
    let mut arr = vec![0; self.que_size as usize];
    for i in 0..self.que_size {
        arr[i as usize] = self.nums[(j % cap) as usize];
        j += 1;
    }
```

```
        arr
    }
}
```

以上實現的佇列仍然具有侷限性：其長度不可變。然而，這個問題不難解決，我們可以將陣列替換為動態陣列，從而引入擴容機制。有興趣的讀者可以嘗試自行實現。

兩種實現的對比結論與堆疊一致，在此不再贅述。

5.2.3　佇列典型應用

- **淘寶訂單**。購物者下單後，訂單將加入列列中，系統隨後會根據順序處理佇列中的訂單。在雙十一期間，短時間內會產生海量訂單，高併發成為工程師們需要重點攻克的問題。
- **各類待辦事項**。任何需要實現“先來後到”功能的場景，例如印表機的任務佇列、餐廳的出餐佇列等，佇列在這些場景中可以有效地維護處理順序。

5.3　雙向佇列

在佇列中，我們僅能刪除頭部元素或在尾部新增元素。如圖 5-7 所示，雙向佇列（double-ended queue）提供了更高的靈活性，允許在頭部和尾部執行元素的新增或刪除操作。

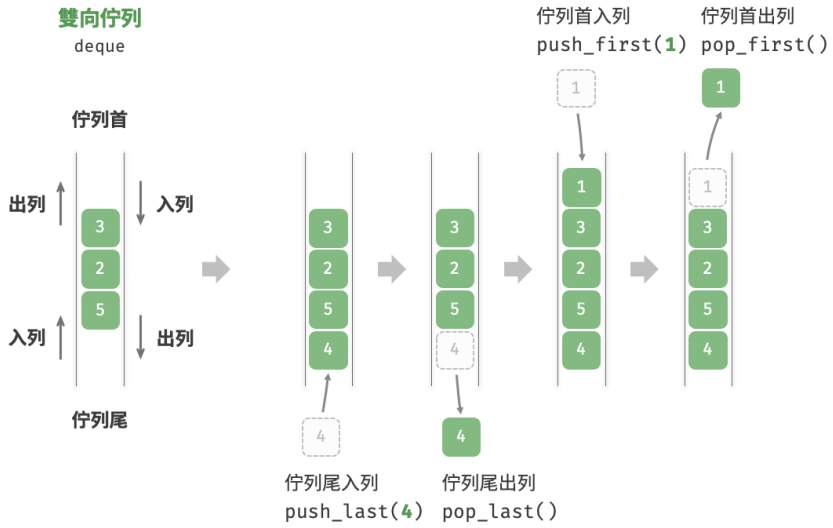


圖 5-7　雙向佇列的操作

5.3.1　雙向佇列常用操作

雙向佇列的常用操作如表 5-3 所示，具體的方法名稱需要根據所使用的程式語言來確定。

表 5-3　雙向佇列操作效率

方法名	描述	時間複雜度
`push_first()`	將元素新增至佇列首	$O(1)$
`push_last()`	將元素新增至佇列尾	$O(1)$
`pop_first()`	刪除佇列首元素	$O(1)$
`pop_last()`	刪除佇列尾元素	$O(1)$
`peek_first()`	訪問佇列首元素	$O(1)$
`peek_last()`	訪問佇列尾元素	$O(1)$

同樣地，我們可以直接使用程式語言中已實現的雙向佇列類別：

```rust
// === File: deque.rs ===

/* 初始化雙向佇列 */
let mut deque: VecDeque<u32> = VecDeque::new();

/* 元素入列 */
deque.push_back(2);  // 新增至佇列尾
deque.push_back(5);
deque.push_back(4);
deque.push_front(3); // 新增至佇列首
deque.push_front(1);

/* 訪問元素 */
if let Some(front) = deque.front() { // 佇列首元素
}
if let Some(rear) = deque.back() {   // 佇列尾元素
}

/* 元素出列 */
if let Some(pop_front) = deque.pop_front() { // 佇列首元素出列
}
if let Some(pop_rear) = deque.pop_back() {   // 佇列尾元素出列
}

/* 獲取雙向佇列的長度 */
let size = deque.len();

/* 判斷雙向佇列是否為空 */
let is_empty = deque.is_empty();
```

5.3.2　雙向佇列實現 *

雙向佇列的實現與佇列類似，可以選擇鏈結串列或陣列作為底層資料結構。

1.　基於雙向鏈結串列的實現

回顧上一節內容，我們使用普通單向鏈結串列來實現佇列，因為它可以方便地刪除頭節點（對應出列操作）和在尾節點後新增新節點（對應入列操作）。

對於雙向佇列而言，頭部和尾部都可以執行入列和出列操作。換句話說，雙向佇列需要實現另一個對稱方向的操作。為此，我們採用“雙向鏈結串列”作為雙向佇列的底層資料結構。

如圖 5-8 所示，我們將雙向鏈結串列的頭節點和尾節點視為雙向佇列的佇列首和佇列尾，同時實現在兩端新增和刪除節點的功能。

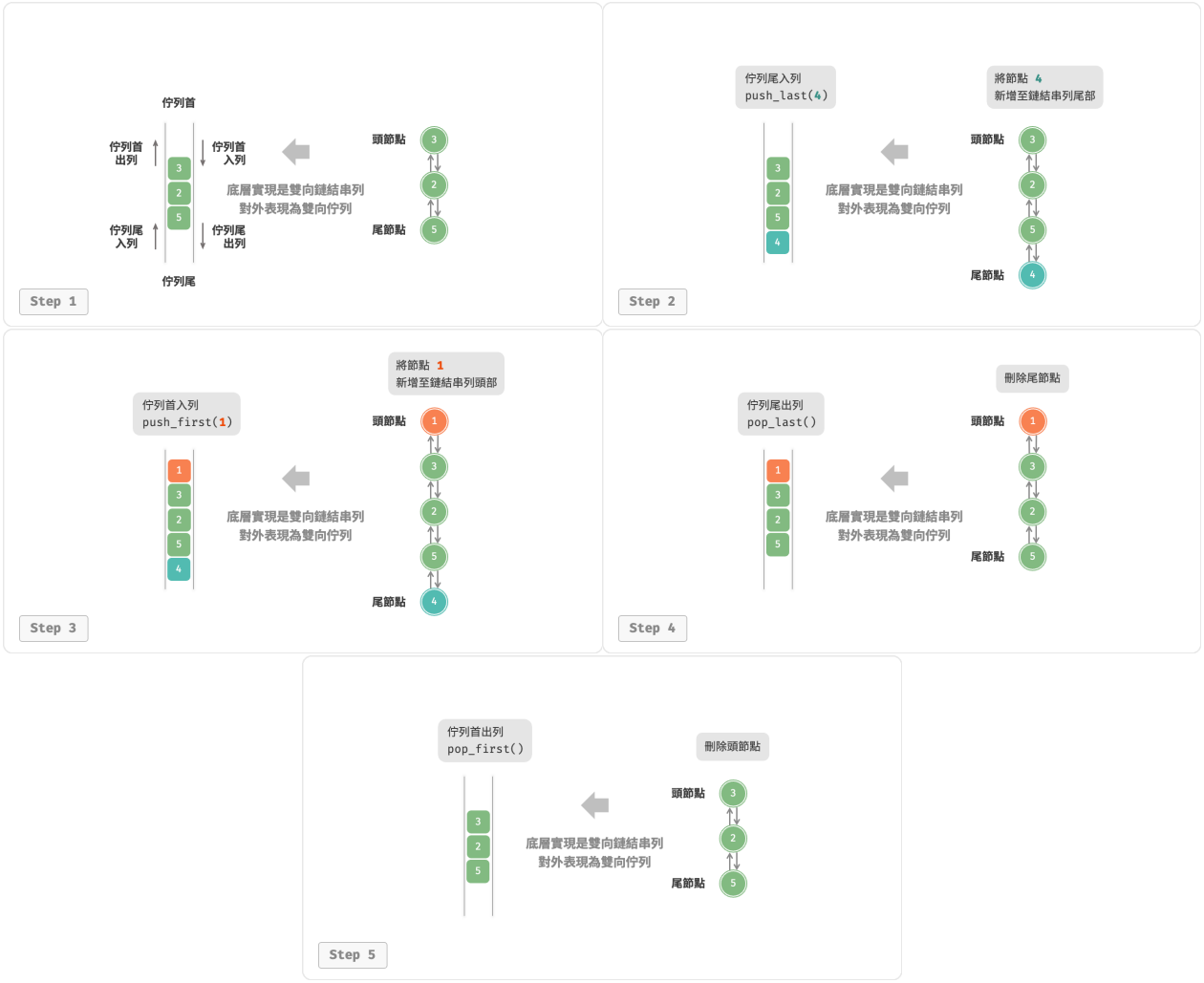


圖 5-8　基於鏈結串列實現雙向佇列的入列出列操作

實現程式碼如下所示：

```rust
// === File: linkedlist_deque.rs ===

/* 雙向鏈結串列節點 */
pub struct ListNode<T> {
    pub val: T,                                     // 節點值
    pub next: Option<Rc<RefCell<ListNode<T>>>>, // 後繼節點指標
    pub prev: Option<Rc<RefCell<ListNode<T>>>>, // 前驅節點指標
}

impl<T> ListNode<T> {
    pub fn new(val: T) -> Rc<RefCell<ListNode<T>>> {
        Rc::new(RefCell::new(ListNode {
            val,
            next: None,
            prev: None,
        }))
    }
}

/* 基於雙向鏈結串列實現的雙向佇列 */
#[allow(dead_code)]
pub struct LinkedListDeque<T> {
    front: Option<Rc<RefCell<ListNode<T>>>>, // 頭節點 front
    rear: Option<Rc<RefCell<ListNode<T>>>>,  // 尾節點 rear
    que_size: usize,                         // 雙向佇列的長度
}

impl<T: Copy> LinkedListDeque<T> {
    pub fn new() -> Self {
        Self {
            front: None,
            rear: None,
            que_size: 0,
        }
    }

    /* 獲取雙向佇列的長度 */
    pub fn size(&self) -> usize {
        return self.que_size;
    }

    /* 判斷雙向佇列是否為空 */
    pub fn is_empty(&self) -> bool {
        return self.size() == 0;
    }

    /* 入列操作 */
```

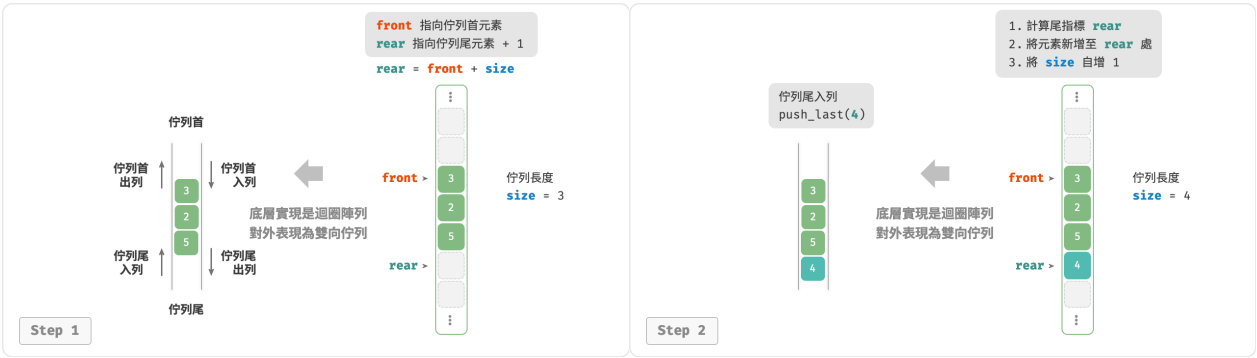
```rust
pub fn push(&mut self, num: T, is_front: bool) {
    let node = ListNode::new(num);
    // 佇列首入列操作
    if is_front {
        match self.front.take() {
            // 若鏈結串列為空，則令 front 和 rear 都指向 node
            None => {
                self.rear = Some(node.clone());
                self.front = Some(node);
            }
            // 將 node 新增至鏈結串列頭部
            Some(old_front) => {
                old_front.borrow_mut().prev = Some(node.clone());
                node.borrow_mut().next = Some(old_front);
                self.front = Some(node); // 更新頭節點
            }
        }
    }
    // 佇列尾入列操作
    else {
        match self.rear.take() {
            // 若鏈結串列為空，則令 front 和 rear 都指向 node
            None => {
                self.front = Some(node.clone());
                self.rear = Some(node);
            }
            // 將 node 新增至鏈結串列尾部
            Some(old_rear) => {
                old_rear.borrow_mut().next = Some(node.clone());
                node.borrow_mut().prev = Some(old_rear);
                self.rear = Some(node); // 更新尾節點
            }
        }
    }
    self.que_size += 1; // 更新佇列長度
}

/* 佇列首入列 */
pub fn push_first(&mut self, num: T) {
    self.push(num, true);
}

/* 佇列尾入列 */
pub fn push_last(&mut self, num: T) {
    self.push(num, false);
}
```

```
    /* 出列操作 */
    pub fn pop(&mut self, is_front: bool) -> Option<T> {
        // 若佇列為空，直接返回 None
        if self.is_empty() {
            return None;
        };
        // 佇列首出列操作
        if is_front {
            self.front.take().map(|old_front| {
                match old_front.borrow_mut().next.take() {
                    Some(new_front) => {
                        new_front.borrow_mut().prev.take();
                        self.front = Some(new_front); // 更新頭節點
                    }
                    None => {
                        self.rear.take();
                    }
                }
                self.que_size -= 1; // 更新佇列長度
                Rc::try_unwrap(old_front).ok().unwrap().into_inner().val
            })
        }
        // 佇列尾出列操作
        else {
            self.rear.take().map(|old_rear| {
                match old_rear.borrow_mut().prev.take() {
                    Some(new_rear) => {
                        new_rear.borrow_mut().next.take();
                        self.rear = Some(new_rear); // 更新尾節點
                    }
                    None => {
                        self.front.take();
                    }
                }
                self.que_size -= 1; // 更新佇列長度
                Rc::try_unwrap(old_rear).ok().unwrap().into_inner().val
            })
        }
    }

    /* 佇列首出列 */
    pub fn pop_first(&mut self) -> Option<T> {
        return self.pop(true);
    }

    /* 佇列尾出列 */
    pub fn pop_last(&mut self) -> Option<T> {
```

```
        return self.pop(false);
    }

    /* 訪問佇列首元素 */
    pub fn peek_first(&self) -> Option<&Rc<RefCell<ListNode<T>>>> {
        self.front.as_ref()
    }

    /* 訪問佇列尾元素 */
    pub fn peek_last(&self) -> Option<&Rc<RefCell<ListNode<T>>>> {
        self.rear.as_ref()
    }

    /* 返回陣列用於列印 */
    pub fn to_array(&self, head: Option<&Rc<RefCell<ListNode<T>>>>) -> Vec<T> {
        if let Some(node) = head {
            let mut nums = self.to_array(node.borrow().next.as_ref());
            nums.insert(0, node.borrow().val);
            return nums;
        }
        return Vec::new();
    }
}
```

2. 基於陣列的實現

如圖 5-9 所示，與基於陣列實現佇列類似，我們也可以使用環形陣列來實現雙向佇列。



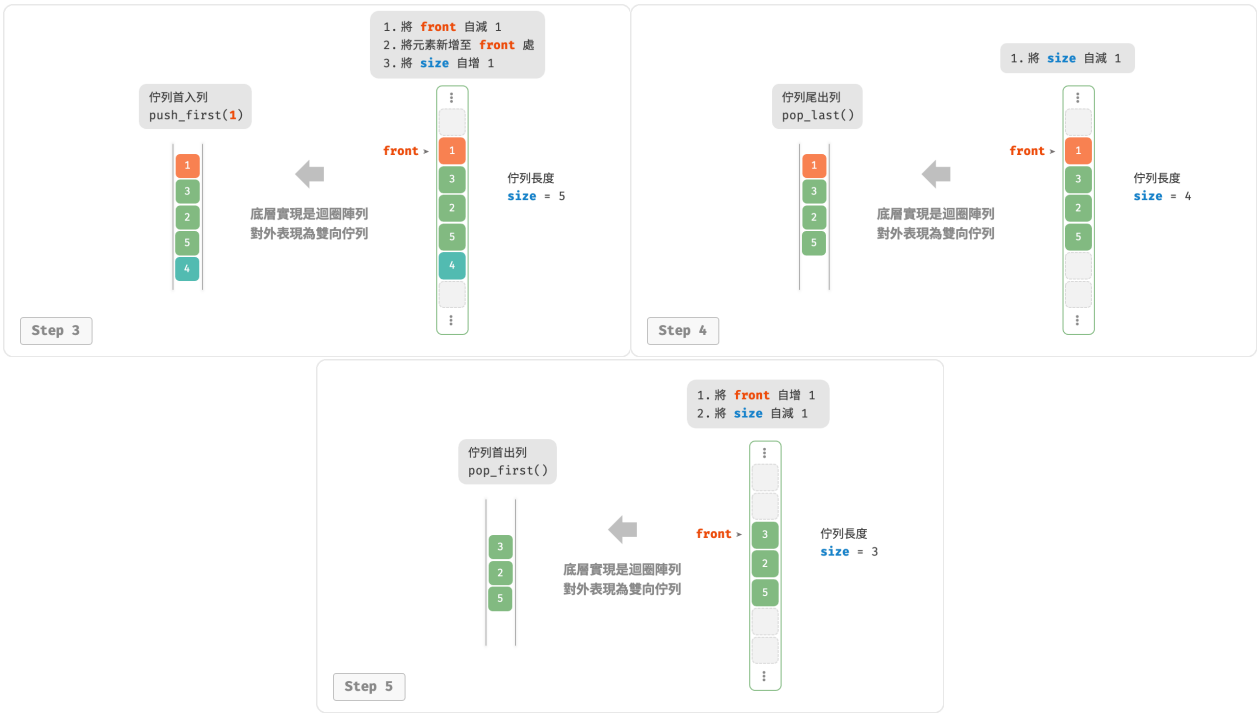


圖 5-9　基於陣列實現雙向佇列的入列出列操作

在佇列的實現基礎上，僅需增加“佇列首入列”和“佇列尾出列”的方法：

```
// === File: array_deque.rs ===

/* 基於環形陣列實現的雙向佇列 */
struct ArrayDeque {
    nums: Vec<i32>,  // 用於儲存雙向佇列元素的陣列
    front: usize,    // 佇列首指標，指向佇列首元素
    que_size: usize, // 雙向佇列長度
}

impl ArrayDeque {
    /* 建構子 */
    pub fn new(capacity: usize) -> Self {
        Self {
            nums: vec![0; capacity],
            front: 0,
            que_size: 0,
        }
    }

    /* 獲取雙向佇列的容量 */
    pub fn capacity(&self) -> usize {
        self.nums.len()
    }
```

```rust
/* 獲取雙向佇列的長度 */
pub fn size(&self) -> usize {
    self.que_size
}

/* 判斷雙向佇列是否為空 */
pub fn is_empty(&self) -> bool {
    self.que_size == 0
}

/* 計算環形陣列索引 */
fn index(&self, i: i32) -> usize {
    // 透過取餘操作實現陣列首尾相連
    // 當 i 越過陣列尾部後，回到頭部
    // 當 i 越過陣列頭部後，回到尾部
    return ((i + self.capacity() as i32) % self.capacity() as i32) as usize;
}

/* 佇列首入列 */
pub fn push_first(&mut self, num: i32) {
    if self.que_size == self.capacity() {
        println!(" 雙向佇列已滿");
        return;
    }
    // 佇列首指標向左移動一位
    // 透過取餘操作實現 front 越過陣列頭部後回到尾部
    self.front = self.index(self.front as i32 - 1);
    // 將 num 新增至佇列首
    self.nums[self.front] = num;
    self.que_size += 1;
}

/* 佇列尾入列 */
pub fn push_last(&mut self, num: i32) {
    if self.que_size == self.capacity() {
        println!(" 雙向佇列已滿");
        return;
    }
    // 計算佇列尾指標，指向佇列尾索引 + 1
    let rear = self.index(self.front as i32 + self.que_size as i32);
    // 將 num 新增至佇列尾
    self.nums[rear] = num;
    self.que_size += 1;
}

/* 佇列首出列 */
```

```
    fn pop_first(&mut self) -> i32 {
        let num = self.peek_first();
        // 佇列首指標向後移動一位
        self.front = self.index(self.front as i32 + 1);
        self.que_size -= 1;
        num
    }

    /* 佇列尾出列 */
    fn pop_last(&mut self) -> i32 {
        let num = self.peek_last();
        self.que_size -= 1;
        num
    }

    /* 訪問佇列首元素 */
    fn peek_first(&self) -> i32 {
        if self.is_empty() {
            panic!(" 雙向佇列為空")
        };
        self.nums[self.front]
    }

    /* 訪問佇列尾元素 */
    fn peek_last(&self) -> i32 {
        if self.is_empty() {
            panic!(" 雙向佇列為空")
        };
        // 計算尾元素索引
        let last = self.index(self.front as i32 + self.que_size as i32 - 1);
        self.nums[last]
    }

    /* 返回陣列用於列印 */
    fn to_array(&self) -> Vec<i32> {
        // 僅轉換有效長度範圍內的串列元素
        let mut res = vec![0; self.que_size];
        let mut j = self.front;
        for i in 0..self.que_size {
            res[i] = self.nums[self.index(j as i32)];
            j += 1;
        }
        res
    }
}
```

5.3.3 雙向佇列應用

雙向佇列兼具堆疊與佇列的邏輯，**因此它可以實現這兩者的所有應用場景，同時提供更高的自由度**。

我們知道，軟體的“撤銷”功能通常使用堆疊來實現：系統將每次更改操作 `push` 到堆疊中，然後透過 `pop` 實現撤銷。然而，考慮到系統資源的限制，軟體通常會限制撤銷的步數（例如僅允許儲存 50 步）。當堆疊的長度超過 50 時，軟體需要在堆疊底（佇列首）執行刪除操作。**但堆疊無法實現該功能，此時就需要使用雙向佇列來替代堆疊**。請注意，“撤銷”的核心邏輯仍然遵循堆疊的先入後出原則，只是雙向佇列能夠更加靈活地實現一些額外邏輯。

5.4 小結

1. 重點回顧

- 堆疊是一種遵循先入後出原則的資料結構，可透過陣列或鏈結串列來實現。
- 在時間效率方面，堆疊的陣列實現具有較高的平均效率，但在擴容過程中，單次入堆疊操作的時間複雜度會劣化至 $O(n)$ 。相比之下，堆疊的鏈結串列實現具有更為穩定的效率表現。
- 在空間效率方面，堆疊的陣列實現可能導致一定程度的空間浪費。但需要注意的是，鏈結串列節點所佔用的記憶體空間比陣列元素更大。
- 佇列是一種遵循先入先出原則的資料結構，同樣可以透過陣列或鏈結串列來實現。在時間效率和空間效率的對比上，佇列的結論與前述堆疊的結論相似。
- 雙向佇列是一種具有更高自由度的佇列，它允許在兩端進行元素的新增和刪除操作。

2. Q & A

Q：瀏覽器的前進後退是否是雙向鏈結串列實現？

瀏覽器的前進後退功能本質上是“堆疊”的體現。當用戶訪問一個新頁面時，該頁面會被新增到堆疊頂；當用戶點選後退按鈕時，該頁面會從堆疊頂彈出。使用雙向佇列可以方便地實現一些額外操作，這個在“雙向佇列”章節有提到。

Q：在出堆疊後，是否需要釋放出堆疊節點的記憶體？

如果後續仍需要使用彈出節點，則不需要釋放記憶體。若之後不需要用到，`Java` 和 `Python` 等語言擁有自動垃圾回收機制，因此不需要手動釋放記憶體；在 `C` 和 `C++` 中需要手動釋放記憶體。

Q：雙向佇列像是兩個堆疊拼接在了一起，它的用途是什麼？

雙向佇列就像是堆疊和佇列的組合或兩個堆疊拼在了一起。它表現的是堆疊 + 佇列的邏輯，因此可以實現堆疊與佇列的所有應用，並且更加靈活。

Q：撤銷（undo）和反撤銷（redo）具體是如何實現的？

使用兩個堆疊，堆疊 `A` 用於撤銷，堆疊 `B` 用於反撤銷。

1. 每當使用者執行一個操作，將這個操作壓入堆疊 `A` ，並清空堆疊 `B` 。
2. 當用戶執行“撤銷”時，從堆疊 `A` 中彈出最近的操作，並將其壓入堆疊 `B` 。
3. 當用戶執行“反撤銷”時，從堆疊 `B` 中彈出最近的操作，並將其壓入堆疊 `A` 。

第 6 章　雜湊表

Abstract

在計算機世界中，雜湊表如同一位聰慧的圖書管理員。

他知道如何計算索書號，從而可以快速找到目標圖書。

6.1　雜湊表

雜湊表（hash table），又稱散列表，它透過建立鍵 `key` 與值 `value` 之間的對映，實現高效的元素查詢。具體而言，我們向雜湊表中輸入一個鍵 `key` ，則可以在 $O(1)$ 時間內獲取對應的值 `value` 。

如圖 6-1 所示，給定 n 個學生，每個學生都有“姓名”和“學號”兩項資料。假如我們希望實現“輸入一個學號，返回對應的姓名”的查詢功能，則可以採用圖 6-1 所示的雜湊表來實現。

圖 6-1　雜湊表的抽象表示

除雜湊表外，陣列和鏈結串列也可以實現查詢功能，它們的效率對比如表 6-1 所示。

- **新增元素**：僅需將元素新增至陣列（鏈結串列）的尾部即可，使用 $O(1)$ 時間。
- **查詢元素**：由於陣列（鏈結串列）是亂序的，因此需要走訪其中的所有元素，使用 $O(n)$ 時間。
- **刪除元素**：需要先查詢到元素，再從陣列（鏈結串列）中刪除，使用 $O(n)$ 時間。

表 6-1　元素查詢效率對比

	陣列	鏈結串列	雜湊表
查詢元素	$O(n)$	$O(n)$	$O(1)$
新增元素	$O(1)$	$O(1)$	$O(1)$
刪除元素	$O(n)$	$O(n)$	$O(1)$

觀察發現，**在雜湊表中進行增刪查改的時間複雜度都是 $O(1)$** ，非常高效。

6.1.1　雜湊表常用操作

雜湊表的常見操作包括：初始化、查詢操作、新增鍵值對和刪除鍵值對等，示例程式碼如下：

```rust
// === File: hash_map.rs ===

use std::collections::HashMap;

/* 初始化雜湊表 */
let mut map: HashMap<i32, String> = HashMap::new();

/* 新增操作 */
// 在雜湊表中新增鍵值對 (key, value)
map.insert(12836, " 小哈".to_string());
map.insert(15937, " 小囉".to_string());
map.insert(16750, " 小算".to_string());
map.insert(13279, " 小法".to_string());
map.insert(10583, " 小鴨".to_string());

/* 查詢操作 */
// 向雜湊表中輸入鍵 key ，得到值 value
let _name: Option<&String> = map.get(&15937);

/* 刪除操作 */
// 在雜湊表中刪除鍵值對 (key, value)
let _removed_value: Option<String> = map.remove(&10583);
```

雜湊表有三種常用的走訪方式：走訪鍵值對、走訪鍵和走訪值。示例程式碼如下：

```rust
// === File: hash_map.rs ===

/* 走訪雜湊表 */
// 走訪鍵值對 Key->Value
for (key, value) in &map {
    println!("{key} -> {value}");
}

// 單獨走訪鍵 Key
for key in map.keys() {
    println!("{key}");
}

// 單獨走訪值 Value
for value in map.values() {
    println!("{value}");
}
```

6.1.2　雜湊表簡單實現

我們先考慮最簡單的情況，**僅用一個陣列來實現雜湊表**。在雜湊表中，我們將陣列中的每個空位稱為桶（bucket），每個桶可儲存一個鍵值對。因此，查詢操作就是找到 `key` 對應的桶，並在桶中獲取 `value` 。

那麼，如何基於 `key` 定位對應的桶呢？這是透過雜湊函式（hash function）實現的。雜湊函式的作用是將一個較大的輸入空間對映到一個較小的輸出空間。在雜湊表中，輸入空間是所有 `key` ，輸出空間是所有桶（陣列索引）。換句話說，輸入一個 `key` ，**我們可以透過雜湊函式得到該 `key` 對應的鍵值對在陣列中的儲存位置**。

輸入一個 `key` ，雜湊函式的計算過程分為以下兩步。

1. 透過某種雜湊演算法 `hash()` 計算得到雜湊值。
2. 將雜湊值對桶數量（陣列長度）`capacity` 取模，從而獲取該 `key` 對應的陣列索引 `index` 。

```
index = hash(key) % capacity
```

隨後，我們就可以利用 `index` 在雜湊表中訪問對應的桶，從而獲取 `value` 。

設陣列長度 `capacity = 100`、雜湊演算法 `hash(key) = key` ，易得雜湊函式為 `key % 100` 。圖 6-2 以 `key` 學號和 `value` 姓名為例，展示了雜湊函式的工作原理。

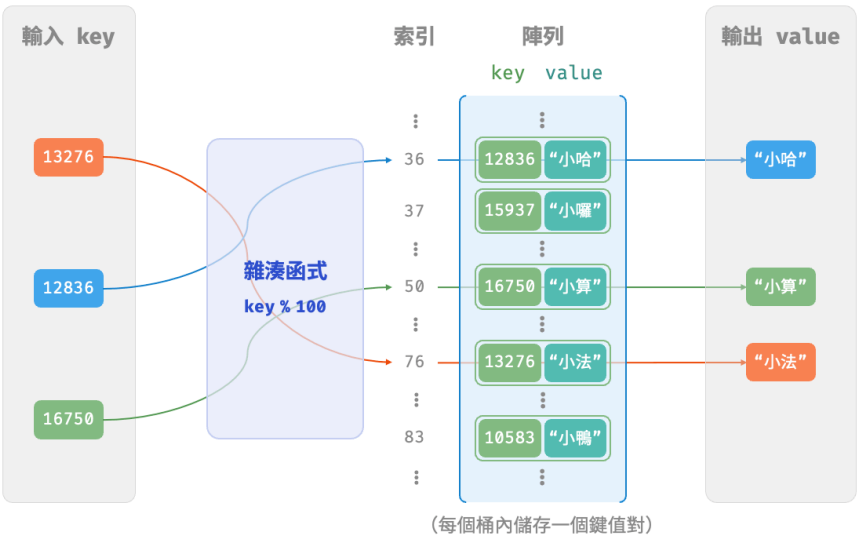


圖 6-2　雜湊函式工作原理

以下程式碼實現了一個簡單雜湊表。其中，我們將 `key` 和 `value` 封裝成一個類別 `Pair` ，以表示鍵值對。

```
// === File: array_hash_map.rs ===

/* 鍵值對 */
#[derive(Debug, Clone, PartialEq)]
pub struct Pair {
    pub key: i32,
```

```
    pub val: String,
}

/* 基於陣列實現的雜湊表 */
pub struct ArrayHashMap {
    buckets: Vec<Option<Pair>>,
}

impl ArrayHashMap {
    pub fn new() -> ArrayHashMap {
        // 初始化陣列，包含 100 個桶
        Self {
            buckets: vec![None; 100],
        }
    }

    /* 雜湊函式 */
    fn hash_func(&self, key: i32) -> usize {
        key as usize % 100
    }

    /* 查詢操作 */
    pub fn get(&self, key: i32) -> Option<&String> {
        let index = self.hash_func(key);
        self.buckets[index].as_ref().map(|pair| &pair.val)
    }

    /* 新增操作 */
    pub fn put(&mut self, key: i32, val: &str) {
        let index = self.hash_func(key);
        self.buckets[index] = Some(Pair {
            key,
            val: val.to_string(),
        });
    }

    /* 刪除操作 */
    pub fn remove(&mut self, key: i32) {
        let index = self.hash_func(key);
        // 置為 None ，代表刪除
        self.buckets[index] = None;
    }

    /* 獲取所有鍵值對 */
    pub fn entry_set(&self) -> Vec<&Pair> {
        self.buckets
            .iter()
```

```
            .filter_map(|pair| pair.as_ref())
            .collect()
    }

    /* 獲取所有鍵 */
    pub fn key_set(&self) -> Vec<&i32> {
        self.buckets
            .iter()
            .filter_map(|pair| pair.as_ref().map(|pair| &pair.key))
            .collect()
    }

    /* 獲取所有值 */
    pub fn value_set(&self) -> Vec<&String> {
        self.buckets
            .iter()
            .filter_map(|pair| pair.as_ref().map(|pair| &pair.val))
            .collect()
    }

    /* 列印雜湊表 */
    pub fn print(&self) {
        for pair in self.entry_set() {
            println!("{} -> {}", pair.key, pair.val);
        }
    }
}
```

6.1.3　雜湊衝突與擴容

從本質上看，雜湊函式的作用是將所有 `key` 構成的輸入空間對映到陣列所有索引構成的輸出空間，而輸入空間往往遠大於輸出空間。因此，**理論上一定存在“多個輸入對應相同輸出”的情況**。

對於上述示例中的雜湊函式，當輸入的 `key` 後兩位相同時，雜湊函式的輸出結果也相同。例如，查詢學號為 12836 和 20336 的兩個學生時，我們得到：

```
12836 % 100 = 36
20336 % 100 = 36
```

如圖 6-3 所示，兩個學號指向了同一個姓名，這顯然是不對的。我們將這種多個輸入對應同一輸出的情況稱為雜湊衝突（hash collision）。

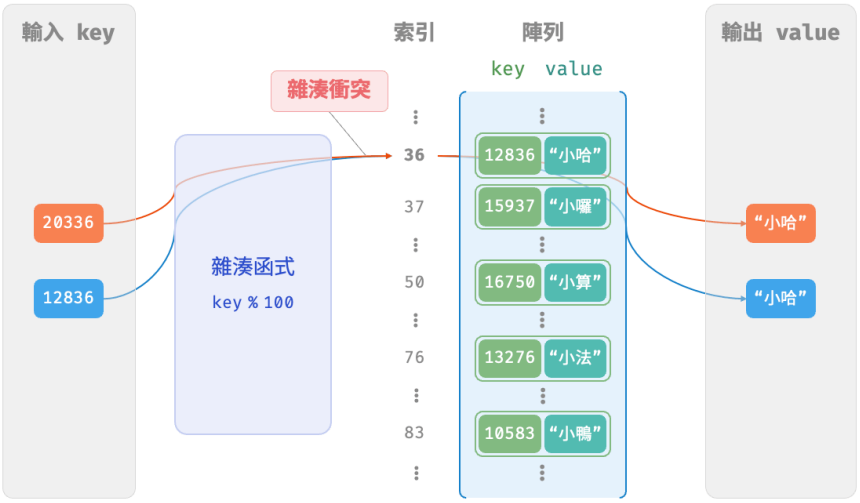


圖 6-3　雜湊衝突示例

容易想到，雜湊表容量 n 越大，多個 `key` 被分配到同一個桶中的機率就越低，衝突就越少。因此，**我們可以透過擴容雜湊表來減少雜湊衝突**。

如圖 6-4 所示，擴容前鍵值對 `(136, A)` 和 `(236, D)` 發生衝突，擴容後衝突消失。

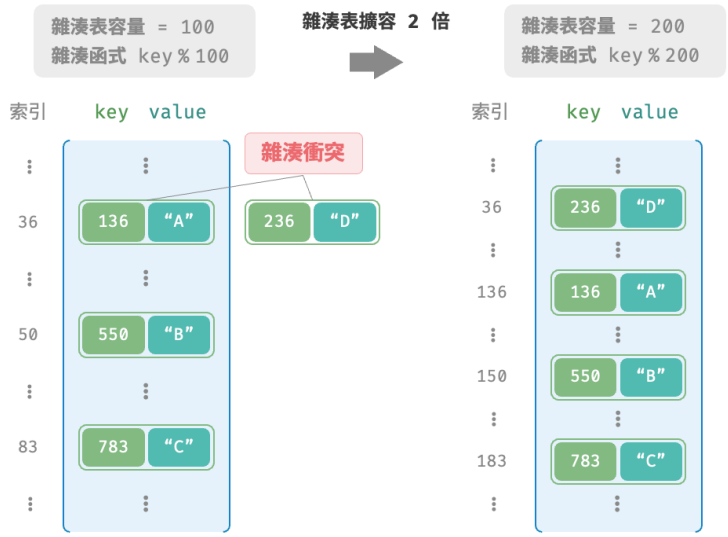


圖 6-4　雜湊表擴容

類似於陣列擴容，雜湊表擴容需將所有鍵值對從原雜湊表遷移至新雜湊表，非常耗時；並且由於雜湊表容量 `capacity` 改變，我們需要透過雜湊函式來重新計算所有鍵值對的儲存位置，這進一步增加了擴容過程的計算開銷。為此，程式語言通常會預留足夠大的雜湊表容量，防止頻繁擴容。

負載因子（load factor）是雜湊表的一個重要概念，其定義為雜湊表的元素數量除以桶數量，用於衡量雜湊衝突的嚴重程度，**也常作為雜湊表擴容的觸發條件**。例如在 Java 中，當負載因子超過 0.75 時，系統會將雜湊表擴容至原先的 2 倍。

6.2　雜湊衝突

上一節提到，**通常情況下雜湊函式的輸入空間遠大於輸出空間**，因此理論上雜湊衝突是不可避免的。比如，輸入空間為全體整數，輸出空間為陣列容量大小，則必然有多個整數對映至同一桶索引。

雜湊衝突會導致查詢結果錯誤，嚴重影響雜湊表的可用性。為了解決該問題，每當遇到雜湊衝突時，我們就進行雜湊表擴容，直至衝突消失為止。此方法簡單粗暴且有效，但效率太低，因為雜湊表擴容需要進行大量的資料搬運與雜湊值計算。為了提升效率，我們可以採用以下策略。

1. 改良雜湊表資料結構，**使得雜湊表可以在出現雜湊衝突時正常工作**。
2. 僅在必要時，即當雜湊衝突比較嚴重時，才執行擴容操作。

雜湊表的結構改良方法主要包括“鏈式位址”和“開放定址”。

6.2.1　鏈式位址

在原始雜湊表中，每個桶僅能儲存一個鍵值對。**鏈式位址**（separate chaining）將單個元素轉換為鏈結串列，將鍵值對作為鏈結串列節點，將所有發生衝突的鍵值對都儲存在同一鏈結串列中。圖 6-5 展示了一個鏈式位址雜湊表的例子。

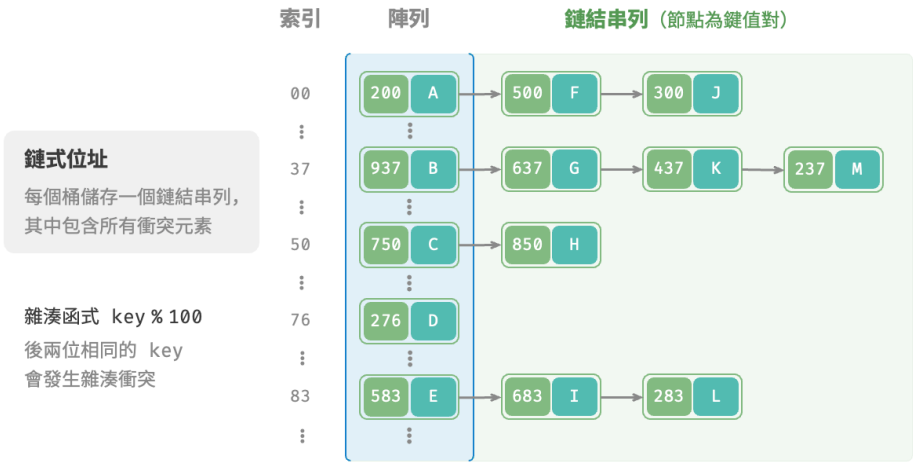


圖 6-5　鏈式位址雜湊表

基於鏈式位址實現的雜湊表的操作方法發生了以下變化。

- **查詢元素**：輸入 `key`，經過雜湊函式得到桶索引，即可訪問鏈結串列頭節點，然後走訪鏈結串列並對比 `key` 以查詢目標鍵值對。
- **新增元素**：首先透過雜湊函式訪問鏈結串列頭節點，然後將節點（鍵值對）新增到鏈結串列中。
- **刪除元素**：根據雜湊函式的結果訪問鏈結串列頭部，接著走訪鏈結串列以查詢目標節點並將其刪除。

鏈式位址存在以下侷限性。

- **佔用空間增大**：鏈結串列包含節點指標，它相比陣列更加耗費記憶體空間。

- **查詢效率降低**：因為需要線性走訪鏈結串列來查詢對應元素。

以下程式碼給出了鏈式位址雜湊表的簡單實現，需要注意兩點。

- 使用串列（動態陣列）代替鏈結串列，從而簡化程式碼。在這種設定下，雜湊表（陣列）包含多個桶，每個桶都是一個串列。
- 以下實現包含雜湊表擴容方法。當負載因子超過 $\frac{2}{3}$ 時，我們將雜湊表擴容至原先的 2 倍。

```rust
// === File: hash_map_chaining.rs ===

/* 鏈式位址雜湊表 */
struct HashMapChaining {
    size: i32,
    capacity: i32,
    load_thres: f32,
    extend_ratio: i32,
    buckets: Vec<Vec<Pair>>,
}

impl HashMapChaining {
    /* 建構子 */
    fn new() -> Self {
        Self {
            size: 0,
            capacity: 4,
            load_thres: 2.0 / 3.0,
            extend_ratio: 2,
            buckets: vec![vec![]; 4],
        }
    }

    /* 雜湊函式 */
    fn hash_func(&self, key: i32) -> usize {
        key as usize % self.capacity as usize
    }

    /* 負載因子 */
    fn load_factor(&self) -> f32 {
        self.size as f32 / self.capacity as f32
    }

    /* 刪除操作 */
    fn remove(&mut self, key: i32) -> Option<String> {
        let index = self.hash_func(key);
        let bucket = &mut self.buckets[index];

        // 走訪桶，從中刪除鍵值對
        for i in 0..bucket.len() {
```

```
            if bucket[i].key == key {
                let pair = bucket.remove(i);
                self.size -= 1;
                return Some(pair.val);
            }
        }

        // 若未找到 key ，则返回 None
        None
    }

    /* 擴容雜湊表 */
    fn extend(&mut self) {
        // 暫存原雜湊表
        let buckets_tmp = std::mem::replace(&mut self.buckets, vec![]);

        // 初始化擴容後的新雜湊表
        self.capacity *= self.extend_ratio;
        self.buckets = vec![Vec::new(); self.capacity as usize];
        self.size = 0;

        // 將鍵值對從原雜湊表搬運至新雜湊表
        for bucket in buckets_tmp {
            for pair in bucket {
                self.put(pair.key, pair.val);
            }
        }
    }

    /* 列印雜湊表 */
    fn print(&self) {
        for bucket in &self.buckets {
            let mut res = Vec::new();
            for pair in bucket {
                res.push(format!("{} -> {}", pair.key, pair.val));
            }
            println!("{:?}", res);
        }
    }

    /* 新增操作 */
    fn put(&mut self, key: i32, val: String) {
        // 當負載因子超過閾值時，執行擴容
        if self.load_factor() > self.load_thres {
            self.extend();
        }
```

```
        let index = self.hash_func(key);
        let bucket = &mut self.buckets[index];

        // 走訪桶，若遇到指定 key ，則更新對應 val 並返回
        for pair in bucket {
            if pair.key == key {
                pair.val = val;
                return;
            }
        }
        let bucket = &mut self.buckets[index];

        // 若無該 key ，則將鍵值對新增至尾部
        let pair = Pair { key, val };
        bucket.push(pair);
        self.size += 1;
    }

    /* 查詢操作 */
    fn get(&self, key: i32) -> Option<&str> {
        let index = self.hash_func(key);
        let bucket = &self.buckets[index];

        // 走訪桶，若找到 key ，則返回對應 val
        for pair in bucket {
            if pair.key == key {
                return Some(&pair.val);
            }
        }

        // 若未找到 key ，則返回 None
        None
    }
}
```

值得注意的是，當鏈結串列很長時，查詢效率 $O(n)$ 很差。**此時可以將鏈結串列轉換為“AVL 樹”或“紅黑樹”**，從而將查詢操作的時間複雜度最佳化至 $O(\log n)$ 。

6.2.2　開放定址

開放定址（open addressing）不引入額外的資料結構，而是透過“多次探測”來處理雜湊衝突，探測方式主要包括線性探查、平方探測和多次雜湊等。

下面以線性探查為例，介紹開放定址雜湊表的工作機制。

1. 線性探查

線性探查採用固定步長的線性搜尋來進行探測，其操作方法與普通雜湊表有所不同。

- **插入元素**：透過雜湊函式計算桶索引，若發現桶內已有元素，則從衝突位置向後線性走訪（步長通常為 1 ），直至找到空桶，將元素插入其中。
- **查詢元素**：若發現雜湊衝突，則使用相同步長向後進行線性走訪，直到找到對應元素，返回 `value` 即可；如果遇到空桶，說明目標元素不在雜湊表中，返回 `None` 。

圖 6-6 展示了開放定址（線性探查）雜湊表的鍵值對分佈。根據此雜湊函式，最後兩位相同的 `key` 都會被對映到相同的桶。而透過線性探查，它們被依次儲存在該桶以及之下的桶中。

圖 6-6　開放定址（線性探查）雜湊表的鍵值對分佈

然而，**線性探查容易產生“聚集現象”**。具體來說，陣列中連續被佔用的位置越長，這些連續位置發生雜湊衝突的可能性越大，從而進一步促使該位置的聚堆積生長，形成惡性迴圈，最終導致增刪查改操作效率劣化。

值得注意的是，**我們不能在開放定址雜湊表中直接刪除元素**。這是因為刪除元素會在陣列內產生一個空桶 `None` ，而當查詢元素時，線性探查到該空桶就會返回，因此在該空桶之下的元素都無法再被訪問到，程式可能誤判這些元素不存在，如圖 6-7 所示。

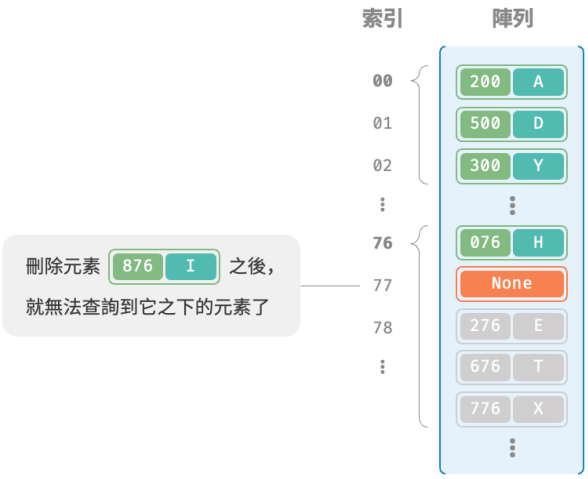


圖 6-7　在開放定址中刪除元素導致的查詢問題

為了解決該問題，我們可以採用懶刪除（lazy deletion）機制：它不直接從雜湊表中移除元素，**而是利用一個常數 `TOMBSTONE` 來標記這個桶**。在該機制下，`None` 和 `TOMBSTONE` 都代表空桶，都可以放置鍵值對。但不同的是，線性探查到 `TOMBSTONE` 時應該繼續走訪，因為其之下可能還存在鍵值對。

然而，**懶刪除可能會加速雜湊表的效能退化**。這是因為每次刪除操作都會產生一個刪除標記，隨著 `TOMBSTONE` 的增加，搜尋時間也會增加，因為線性探查可能需要跳過多個 `TOMBSTONE` 才能找到目標元素。

為此，考慮線上性探查中記錄遇到的首個 `TOMBSTONE` 的索引，並將搜尋到的目標元素與該 `TOMBSTONE` 交換位置。這樣做的好處是當每次查詢或新增元素時，元素會被移動至距離理想位置（探測起始點）更近的桶，從而最佳化查詢效率。

以下程式碼實現了一個包含懶刪除的開放定址（線性探查）雜湊表。為了更加充分地使用雜湊表的空間，我們將雜湊表看作一個“環形陣列”，當越過陣列尾部時，回到頭部繼續走訪。

```
// === File: hash_map_open_addressing.rs ===

/* 開放定址雜湊表 */
struct HashMapOpenAddressing {
    size: usize,                // 鍵值對數量
    capacity: usize,            // 雜湊表容量
    load_thres: f64,            // 觸發擴容的負載因子閾值
    extend_ratio: usize,        // 擴容倍數
    buckets: Vec<Option<Pair>>, // 桶陣列
    TOMBSTONE: Option<Pair>,    // 刪除標記
}

impl HashMapOpenAddressing {
    /* 建構子 */
    fn new() -> Self {
        Self {
```

```
            size: 0,
            capacity: 4,
            load_thres: 2.0 / 3.0,
            extend_ratio: 2,
            buckets: vec![None; 4],
            TOMBSTONE: Some(Pair {
                key: -1,
                val: "-1".to_string(),
            }),
        }
    }

    /* 雜湊函式 */
    fn hash_func(&self, key: i32) -> usize {
        (key % self.capacity as i32) as usize
    }

    /* 負載因子 */
    fn load_factor(&self) -> f64 {
        self.size as f64 / self.capacity as f64
    }

    /* 搜尋 key 對應的桶索引 */
    fn find_bucket(&mut self, key: i32) -> usize {
        let mut index = self.hash_func(key);
        let mut first_tombstone = -1;
        // 線性探查，當遇到空桶時跳出
        while self.buckets[index].is_some() {
            // 若遇到 key，返回對應的桶索引
            if self.buckets[index].as_ref().unwrap().key == key {
                // 若之前遇到了刪除標記，則將建值對移動至該索引
                if first_tombstone != -1 {
                    self.buckets[first_tombstone as usize] = self.buckets[index].take();
                    self.buckets[index] = self.TOMBSTONE.clone();
                    return first_tombstone as usize; // 返回移動後的桶索引
                }
                return index; // 返回桶索引
            }
            // 記錄遇到的首個刪除標記
            if first_tombstone == -1 && self.buckets[index] == self.TOMBSTONE {
                first_tombstone = index as i32;
            }
            // 計算桶索引，越過尾部則返回頭部
            index = (index + 1) % self.capacity;
        }
        // 若 key 不存在，則返回新增點的索引
        if first_tombstone == -1 {
```

```
            index
        } else {
            first_tombstone as usize
        }
    }

    /* 查詢操作 */
    fn get(&mut self, key: i32) -> Option<&str> {
        // 搜尋 key 對應的桶索引
        let index = self.find_bucket(key);
        // 若找到鍵值對，則返回對應 val
        if self.buckets[index].is_some() && self.buckets[index] != self.TOMBSTONE {
            return self.buckets[index].as_ref().map(|pair| &pair.val as &str);
        }
        // 若鍵值對不存在，則返回 null
        None
    }

    /* 新增操作 */
    fn put(&mut self, key: i32, val: String) {
        // 當負載因子超過閾值時，執行擴容
        if self.load_factor() > self.load_thres {
            self.extend();
        }
        // 搜尋 key 對應的桶索引
        let index = self.find_bucket(key);
        // 若找到鍵值對，則覆蓋 val 並返回
        if self.buckets[index].is_some() && self.buckets[index] != self.TOMBSTONE {
            self.buckets[index].as_mut().unwrap().val = val;
            return;
        }
        // 若鍵值對不存在，則新增該鍵值對
        self.buckets[index] = Some(Pair { key, val });
        self.size += 1;
    }

    /* 刪除操作 */
    fn remove(&mut self, key: i32) {
        // 搜尋 key 對應的桶索引
        let index = self.find_bucket(key);
        // 若找到鍵值對，則用刪除標記覆蓋它
        if self.buckets[index].is_some() && self.buckets[index] != self.TOMBSTONE {
            self.buckets[index] = self.TOMBSTONE.clone();
            self.size -= 1;
        }
    }
```

```
    /* 擴容雜湊表 */
    fn extend(&mut self) {
        // 暫存原雜湊表
        let buckets_tmp = self.buckets.clone();
        // 初始化擴容後的新雜湊表
        self.capacity *= self.extend_ratio;
        self.buckets = vec![None; self.capacity];
        self.size = 0;

        // 將鍵值對從原雜湊表搬運至新雜湊表
        for pair in buckets_tmp {
            if pair.is_none() || pair == self.TOMBSTONE {
                continue;
            }
            let pair = pair.unwrap();

            self.put(pair.key, pair.val);
        }
    }
    /* 列印雜湊表 */
    fn print(&self) {
        for pair in &self.buckets {
            if pair.is_none() {
                println!("null");
            } else if pair == &self.TOMBSTONE {
                println!("TOMBSTONE");
            } else {
                let pair = pair.as_ref().unwrap();
                println!("{} -> {}", pair.key, pair.val);
            }
        }
    }
}
```

2. 平方探測

平方探測與線性探查類似，都是開放定址的常見策略之一。當發生衝突時，平方探測不是簡單地跳過一個固定的步數，而是跳過“探測次數的平方”的步數，即 $1, 4, 9, \dots$ 步。

平方探測主要具有以下優勢。

- 平方探測透過跳過探測次數平方的距離，試圖緩解線性探查的聚集效應。
- 平方探測會跳過更大的距離來尋找空位置，有助於資料分佈得更加均勻。

然而，平方探測並不是完美的。

- 仍然存在聚集現象，即某些位置比其他位置更容易被佔用。

- 由於平方的增長，平方探測可能不會探測整個雜湊表，這意味著即使雜湊表中有空桶，平方探測也可能無法訪問到它。

3. 多次雜湊

顧名思義，多次雜湊方法使用多個雜湊函式 $f_1(x)$、$f_2(x)$、$f_3(x)$、... 進行探測。

- **插入元素**：若雜湊函式 $f_1(x)$ 出現衝突，則嘗試 $f_2(x)$ ，以此類推，直到找到空位後插入元素。
- **查詢元素**：在相同的雜湊函式順序下進行查詢，直到找到目標元素時返回；若遇到空位或已嘗試所有雜湊函式，說明雜湊表中不存在該元素，則返回 `None` 。

與線性探查相比，多次雜湊方法不易產生聚集，但多個雜湊函式會帶來額外的計算量。

> **Tip**
> 請注意，開放定址（線性探查、平方探測和多次雜湊）雜湊表都存在“不能直接刪除元素”的問題。

6.2.3 程式語言的選擇

各種程式語言採取了不同的雜湊表實現策略，下面舉幾個例子。

- Python 採用開放定址。字典 `dict` 使用偽隨機數進行探測。
- Java 採用鏈式位址。自 JDK 1.8 以來，當 `HashMap` 內陣列長度達到 64 且鏈結串列長度達到 8 時，鏈結串列會轉換為紅黑樹以提升查詢效能。
- Go 採用鏈式位址。Go 規定每個桶最多儲存 8 個鍵值對，超出容量則連線一個溢位桶；當溢位桶過多時，會執行一次特殊的等量擴容操作，以確保效能。

6.3 雜湊演算法

前兩節介紹了雜湊表的工作原理和雜湊衝突的處理方法。然而無論是開放定址還是鏈式位址，**它們只能保證雜湊表可以在發生衝突時正常工作，而無法減少雜湊衝突的發生**。

如果雜湊衝突過於頻繁，雜湊表的效能則會急劇劣化。如圖 6-8 所示，對於鏈式位址雜湊表，理想情況下鍵值對均勻分佈在各個桶中，達到最佳查詢效率；最差情況下所有鍵值對都儲存到同一個桶中，時間複雜度退化至 $O(n)$ 。

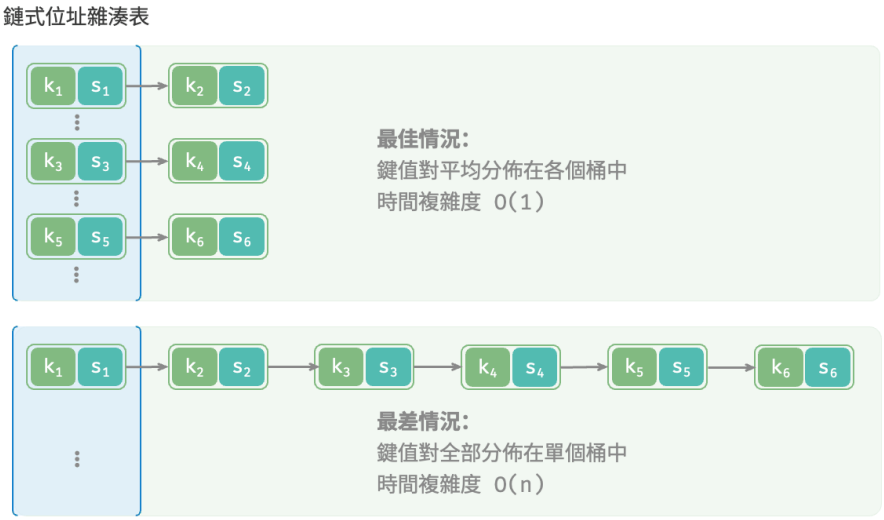


圖 6-8　雜湊衝突的最佳情況與最差情況

鍵值對的分佈情況由雜湊函式決定。回憶雜湊函式的計算步驟，先計算雜湊值，再對陣列長度取模：

```
index = hash(key) % capacity
```

觀察以上公式，當雜湊表容量 `capacity` 固定時，**雜湊演算法 `hash()` 決定了輸出值**，進而決定了鍵值對在雜湊表中的分佈情況。

這意味著，為了降低雜湊衝突的發生機率，我們應當將注意力集中在雜湊演算法 `hash()` 的設計上。

6.3.1　雜湊演算法的目標

為了實現“既快又穩”的雜湊表資料結構，雜湊演算法應具備以下特點。

- **確定性**：對於相同的輸入，雜湊演算法應始終產生相同的輸出。這樣才能確保雜湊表是可靠的。
- **效率高**：計算雜湊值的過程應該足夠快。計算開銷越小，雜湊表的實用性越高。
- **均勻分佈**：雜湊演算法應使得鍵值對均勻分佈在雜湊表中。分佈越均勻，雜湊衝突的機率就越低。

實際上，雜湊演算法除了可以用於實現雜湊表，還廣泛應用於其他領域中。

- **密碼儲存**：為了保護使用者密碼的安全，系統通常不會直接儲存使用者的明文密碼，而是儲存密碼的雜湊值。當用戶輸入密碼時，系統會對輸入的密碼計算雜湊值，然後與儲存的雜湊值進行比較。如果兩者匹配，那麼密碼就被視為正確。
- **資料完整性檢查**：資料傳送方可以計算資料的雜湊值並將其一同傳送；接收方可以重新計算接收到的資料的雜湊值，並與接收到的雜湊值進行比較。如果兩者匹配，那麼資料就被視為完整。

對於密碼學的相關應用，為了防止從雜湊值推導出原始密碼等逆向工程，雜湊演算法需要具備更高等級的安全特性。

- **單向性**：無法透過雜湊值反推出關於輸入資料的任何資訊。

- **抗碰撞性**：應當極難找到兩個不同的輸入，使得它們的雜湊值相同。
- **雪崩效應**：輸入的微小變化應當導致輸出的顯著且不可預測的變化。

請注意，**“均勻分佈”與“抗碰撞性”是兩個獨立的概念**，滿足均勻分佈不一定滿足抗碰撞性。例如，在隨機輸入 `key` 下，雜湊函式 `key % 100` 可以產生均勻分佈的輸出。然而該雜湊演算法過於簡單，所有後兩位相等的 `key` 的輸出都相同，因此我們可以很容易地從雜湊值反推出可用的 `key` ，從而破解密碼。

6.3.2　雜湊演算法的設計

雜湊演算法的設計是一個需要考慮許多因素的複雜問題。然而對於某些要求不高的場景，我們也能設計一些簡單的雜湊演算法。

- **加法雜湊**：對輸入的每個字元的 ASCII 碼進行相加，將得到的總和作為雜湊值。
- **乘法雜湊**：利用乘法的不相關性，每輪乘以一個常數，將各個字元的 ASCII 碼累積到雜湊值中。
- **互斥或雜湊**：將輸入資料的每個元素透過互斥或操作累積到一個雜湊值中。
- **旋轉雜湊**：將每個字元的 ASCII 碼累積到一個雜湊值中，每次累積之前都會對雜湊值進行旋轉操作。

```rust
// === File: simple_hash.rs ===

/* 加法雜湊 */
fn add_hash(key: &str) -> i32 {
    let mut hash = 0_i64;
    const MODULUS: i64 = 1000000007;

    for c in key.chars() {
        hash = (hash + c as i64) % MODULUS;
    }

    hash as i32
}

/* 乘法雜湊 */
fn mul_hash(key: &str) -> i32 {
    let mut hash = 0_i64;
    const MODULUS: i64 = 1000000007;

    for c in key.chars() {
        hash = (31 * hash + c as i64) % MODULUS;
    }

    hash as i32
}

/* 互斥或雜湊 */
fn xor_hash(key: &str) -> i32 {
    let mut hash = 0_i64;
```

```
    const MODULUS: i64 = 1000000007;

    for c in key.chars() {
        hash ^= c as i64;
    }

    (hash & MODULUS) as i32
}

/* 旋轉雜湊 */
fn rot_hash(key: &str) -> i32 {
    let mut hash = 0_i64;
    const MODULUS: i64 = 1000000007;

    for c in key.chars() {
        hash = ((hash << 4) ^ (hash >> 28) ^ c as i64) % MODULUS;
    }

    hash as i32
}
```

觀察發現，每種雜湊演算法的最後一步都是對大質數 1000000007 取模，以確保雜湊值在合適的範圍內。值得思考的是，為什麼要強調對質數取模，或者說對合數取模的弊端是什麼？這是一個有趣的問題。

先丟擲結論：**使用大質數作為模數，可以最大化地保證雜湊值的均勻分佈**。因為質數不與其他數字存在公約數，可以減少因取模操作而產生的週期性模式，從而避免雜湊衝突。

舉個例子，假設我們選擇合數 9 作為模數，它可以被 3 整除，那麼所有可以被 3 整除的 `key` 都會被對映到 0、3、6 這三個雜湊值。

$$
\begin{aligned}
\text{modulus} &= 9 \\
\text{key} &= \{0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, \dots\} \\
\text{hash} &= \{0, 3, 6, 0, 3, 6, 0, 3, 6, 0, 3, 6, \dots\}
\end{aligned}
$$

如果輸入 `key` 恰好滿足這種等差數列的資料分佈，那麼雜湊值就會出現聚堆積，從而加重雜湊衝突。現在，假設將 `modulus` 替換為質數 13 ，由於 `key` 和 `modulus` 之間不存在公約數，因此輸出的雜湊值的均勻性會明顯提升。

$$
\begin{aligned}
\text{modulus} &= 13 \\
\text{key} &= \{0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, \dots\} \\
\text{hash} &= \{0, 3, 6, 9, 12, 2, 5, 8, 11, 1, 4, 7, \dots\}
\end{aligned}
$$

值得說明的是，如果能夠保證 `key` 是隨機均勻分佈的，那麼選擇質數或者合數作為模數都可以，它們都能輸出均勻分佈的雜湊值。而當 `key` 的分佈存在某種週期性時，對合數取模更容易出現聚集現象。

總而言之，我們通常選取質數作為模數，並且這個質數最好足夠大，以儘可能消除週期性模式，提升雜湊演算法的穩健性。

6.3.3 常見雜湊演算法

不難發現，以上介紹的簡單雜湊演算法都比較“脆弱”，遠遠沒有達到雜湊演算法的設計目標。例如，由於加法和互斥或滿足交換律，因此加法雜湊和互斥或雜湊無法區分內容相同但順序不同的字串，這可能會加劇雜湊衝突，並引起一些安全問題。

在實際中，我們通常會用一些標準雜湊演算法，例如 MD5、SHA-1、SHA-2 和 SHA-3 等。它們可以將任意長度的輸入資料對映到恆定長度的雜湊值。

近一個世紀以來，雜湊演算法處在不斷升級與最佳化的過程中。一部分研究人員努力提升雜湊演算法的效能，另一部分研究人員和駭客則致力於尋找雜湊演算法的安全性問題。表 6-2 展示了在實際應用中常見的雜湊演算法。

- MD5 和 SHA-1 已多次被成功攻擊，因此它們被各類安全應用棄用。
- SHA-2 系列中的 SHA-256 是最安全的雜湊演算法之一，仍未出現成功的攻擊案例，因此常用在各類安全應用與協議中。
- SHA-3 相較 SHA-2 的實現開銷更低、計算效率更高，但目前使用覆蓋度不如 SHA-2 系列。

表 6-2　常見的雜湊演算法

	MD5	SHA-1	SHA-2	SHA-3
推出時間	1992	1995	2002	2008
輸出長度	128 bit	160 bit	256/512 bit	224/256/384/512 bit
雜湊衝突	較多	較多	很少	很少
安全等級	低，已被成功攻擊	低，已被成功攻擊	高	高
應用	已被棄用，仍用於資料完整性檢查	已被棄用	加密貨幣交易驗證、數字簽名等	可用於替代 SHA-2

6.3.4 資料結構的雜湊值

我們知道，雜湊表的 `key` 可以是整數、小數或字串等資料型別。程式語言通常會為這些資料型別提供內建的雜湊演算法，用於計算雜湊表中的桶索引。以 Python 為例，我們可以呼叫 `hash()` 函式來計算各種資料型別的雜湊值。

- 整數和布林量的雜湊值就是其本身。
- 浮點數和字串的雜湊值計算較為複雜，有興趣的讀者請自行學習。

- 元組的雜湊值是對其中每一個元素進行雜湊，然後將這些雜湊值組合起來，得到單一的雜湊值。
- 物件的雜湊值基於其記憶體位址生成。透過重寫物件的雜湊方法，可實現基於內容生成雜湊值。

> **Tip**
> 請注意，不同程式語言的內建雜湊值計算函式的定義和方法不同。

```rust
// === File: built_in_hash.rs ===

use std::collections::hash_map::DefaultHasher;
use std::hash::{Hash, Hasher};

let num = 3;
let mut num_hasher = DefaultHasher::new();
num.hash(&mut num_hasher);
let hash_num = num_hasher.finish();
// 整數 3 的雜湊值為 568126464209439262

let bol = true;
let mut bol_hasher = DefaultHasher::new();
bol.hash(&mut bol_hasher);
let hash_bol = bol_hasher.finish();
// 布林量 true 的雜湊值為 4952851536318644461

let dec: f32 = 3.14159;
let mut dec_hasher = DefaultHasher::new();
dec.to_bits().hash(&mut dec_hasher);
let hash_dec = dec_hasher.finish();
// 小數 3.14159 的雜湊值為 2566941990314602357

let str = "Hello 演算法";
let mut str_hasher = DefaultHasher::new();
str.hash(&mut str_hasher);
let hash_str = str_hasher.finish();
// 字串“Hello 演算法”的雜湊值為 16092673739211250988

let arr = (&12836, &" 小哈");
let mut tup_hasher = DefaultHasher::new();
arr.hash(&mut tup_hasher);
let hash_tup = tup_hasher.finish();
// 元組 (12836, " 小哈") 的雜湊值為 1885128010422702749

let node = ListNode::new(42);
let mut hasher = DefaultHasher::new();
node.borrow().val.hash(&mut hasher);
let hash = hasher.finish();
// 節點物件 RefCell { value: ListNode { val: 42, next: None } } 的雜湊值為 15387811073369036852
```

在許多程式語言中，**只有不可變物件才可作為雜湊表的 `key`**。假如我們將串列（動態陣列）作為 `key`，當串列的內容發生變化時，它的雜湊值也隨之改變，我們就無法在雜湊表中查詢到原先的 `value` 了。

雖然自定義物件（比如鏈結串列節點）的成員變數是可變的，但它是可雜湊的。**這是因為物件的雜湊值通常是基於記憶體位址生成的**，即使物件的內容發生了變化，但它的記憶體位址不變，雜湊值仍然是不變的。

細心的你可能發現在不同控制檯中執行程式時，輸出的雜湊值是不同的。**這是因為 Python 直譯器在每次啟動時，都會為字串雜湊函式加入一個隨機的鹽（salt）值**。這種做法可以有效防止 HashDoS 攻擊，提升雜湊演算法的安全性。

6.4　小結

1.　重點回顧

- 輸入 `key`，雜湊表能夠在 $O(1)$ 時間內查詢到 `value`，效率非常高。
- 常見的雜湊表操作包括查詢、新增鍵值對、刪除鍵值對和走訪雜湊表等。
- 雜湊函式將 `key` 對映為陣列索引，從而訪問對應桶並獲取 `value`。
- 兩個不同的 `key` 可能在經過雜湊函式後得到相同的陣列索引，導致查詢結果出錯，這種現象被稱為雜湊衝突。
- 雜湊表容量越大，雜湊衝突的機率就越低。因此可以透過擴容雜湊表來緩解雜湊衝突。與陣列擴容類似，雜湊表擴容操作的開銷很大。
- 負載因子定義為雜湊表中元素數量除以桶數量，反映了雜湊衝突的嚴重程度，常用作觸發雜湊表擴容的條件。
- 鏈式位址透過將單個元素轉化為鏈結串列，將所有衝突元素儲存在同一個鏈結串列中。然而，鏈結串列過長會降低查詢效率，可以透過進一步將鏈結串列轉換為紅黑樹來提高效率。
- 開放定址透過多次探測來處理雜湊衝突。線性探查使用固定步長，缺點是不能刪除元素，且容易產生聚集。多次雜湊使用多個雜湊函式進行探測，相較線性探查更不易產生聚集，但多個雜湊函式增加了計算量。
- 不同程式語言採取了不同的雜湊表實現。例如，Java 的 `HashMap` 使用鏈式位址，而 Python 的 `Dict` 採用開放定址。
- 在雜湊表中，我們希望雜湊演算法具有確定性、高效率和均勻分佈的特點。在密碼學中，雜湊演算法還應該具備抗碰撞性和雪崩效應。
- 雜湊演算法通常採用大質數作為模數，以最大化地保證雜湊值均勻分佈，減少雜湊衝突。
- 常見的雜湊演算法包括 MD5、SHA-1、SHA-2 和 SHA-3 等。MD5 常用於校驗檔案完整性，SHA-2 常用於安全應用與協議。
- 程式語言通常會為資料型別提供內建雜湊演算法，用於計算雜湊表中的桶索引。通常情況下，只有不可變物件是可雜湊的。

2.　Q & A

Q：雜湊表的時間複雜度在什麼情況下是 $O(n)$？

當雜湊衝突比較嚴重時，雜湊表的時間複雜度會退化至 $O(n)$。當雜湊函式設計得比較好、容量設定比較合理、衝突比較平均時，時間複雜度是 $O(1)$。我們使用程式語言內建的雜湊表時，通常認為時間複雜度是 $O(1)$。

Q：為什麼不使用雜湊函式 $f(x) = x$ 呢？這樣就不會有衝突了。

在 $f(x) = x$ 雜湊函式下，每個元素對應唯一的桶索引，這與陣列等價。然而，輸入空間通常遠大於輸出空間（陣列長度），因此雜湊函式的最後一步往往是對陣列長度取模。換句話說，雜湊表的目標是將一個較大的狀態空間對映到一個較小的空間，並提供 $O(1)$ 的查詢效率。

Q：雜湊表底層實現是陣列、鏈結串列、二元樹，但為什麼效率可以比它們更高呢？

首先，雜湊表的時間效率變高，但空間效率變低了。雜湊表有相當一部分記憶體未使用。

其次，只是在特定使用場景下時間效率變高了。如果一個功能能夠在相同的時間複雜度下使用陣列或鏈結串列實現，那麼通常比雜湊表更快。這是因為雜湊函式計算需要開銷，時間複雜度的常數項更大。

最後，雜湊表的時間複雜度可能發生劣化。例如在鏈式位址中，我們採取在鏈結串列或紅黑樹中執行查詢操作，仍然有退化至 $O(n)$ 時間的風險。

Q：多次雜湊有不能直接刪除元素的缺陷嗎？標記為已刪除的空間還能再次使用嗎？

多次雜湊是開放定址的一種，開放定址法都有不能直接刪除元素的缺陷，需要透過標記刪除。標記為已刪除的空間可以再次使用。當將新元素插入雜湊表，並且透過雜湊函式找到標記為已刪除的位置時，該位置可以被新元素使用。這樣做既能保持雜湊表的探測序列不變，又能保證雜湊表的空間使用率。

Q：為什麼線上性探查中，查詢元素的時候會出現雜湊衝突呢？

查詢的時候透過雜湊函式找到對應的桶和鍵值對，發現 `key` 不匹配，這就代表有雜湊衝突。因此，線性探查法會根據預先設定的步長依次向下查詢，直至找到正確的鍵值對或無法找到跳出為止。

Q：為什麼雜湊表擴容能夠緩解雜湊衝突？

雜湊函式的最後一步往往是對陣列長度 n 取模（取餘），讓輸出值落在陣列索引範圍內；在擴容後，陣列長度 n 發生變化，而 `key` 對應的索引也可能發生變化。原先落在同一個桶的多個 `key`，在擴容後可能會被分配到多個桶中，從而實現雜湊衝突的緩解。

第 7 章　樹

Abstract

參天大樹充滿生命力，根深葉茂，分枝扶疏。

它為我們展現了資料分治的生動形態。

7.1　二元樹

二元樹（binary tree）是一種非線性資料結構，代表“祖先”與“後代”之間的派生關係，體現了“一分為二”的分治邏輯。與鏈結串列類似，二元樹的基本單元是節點，每個節點包含值、左子節點引用和右子節點引用。

```rust
use std::rc::Rc;
use std::cell::RefCell;

/* 二元樹節點結構體 */
struct TreeNode {
    val: i32,                                // 節點值
    left: Option<Rc<RefCell<TreeNode>>>,     // 左子節點引用
    right: Option<Rc<RefCell<TreeNode>>>,    // 右子節點引用
}

impl TreeNode {
    /* 建構子 */
    fn new(val: i32) -> Rc<RefCell<Self>> {
        Rc::new(RefCell::new(Self {
            val,
            left: None,
            right: None
        }))
    }
}
```

每個節點都有兩個引用（指標），分別指向**左子節點**（left-child node）和**右子節點**（right-child node），該節點被稱為這兩個子節點的**父節點**（parent node）。當給定一個二元樹的節點時，我們將該節點的左子節點及其以下節點形成的樹稱為該節點的**左子樹**（left subtree），同理可得**右子樹**（right subtree）。

在二元樹中，除葉節點外，其他所有節點都包含子節點和非空子樹。如圖 7-1 所示，如果將“節點 2”視為父節點，則其左子節點和右子節點分別是“節點 4”和“節點 5”，左子樹是“節點 4 及其以下節點形成的樹”，右子樹是“節點 5 及其以下節點形成的樹”。

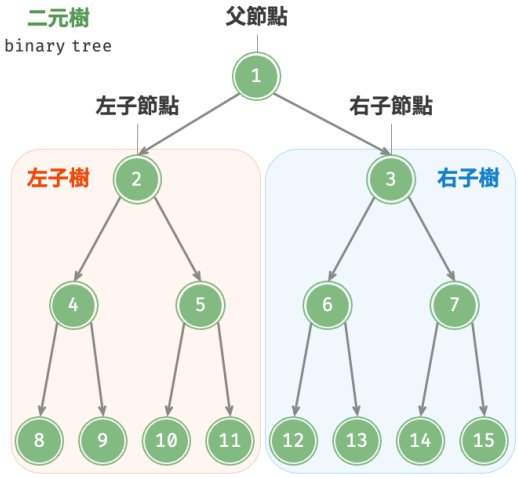


圖 7-1　父節點、子節點、子樹

7.1.1　二元樹常見術語

二元樹的常用術語如圖 7-2 所示。

- 根節點（root node）：位於二元樹頂層的節點，沒有父節點。
- 葉節點（leaf node）：沒有子節點的節點，其兩個指標均指向 `None` 。
- 邊（edge）：連線兩個節點的線段，即節點引用（指標）。
- 節點所在的層（level）：從頂至底遞增，根節點所在層為 1 。
- 節點的度（degree）：節點的子節點的數量。在二元樹中，度的取值範圍是 0、1、2 。
- 二元樹的高度（height）：從根節點到最遠葉節點所經過的邊的數量。
- 節點的深度（depth）：從根節點到該節點所經過的邊的數量。
- 節點的高度（height）：從距離該節點最遠的葉節點到該節點所經過的邊的數量。

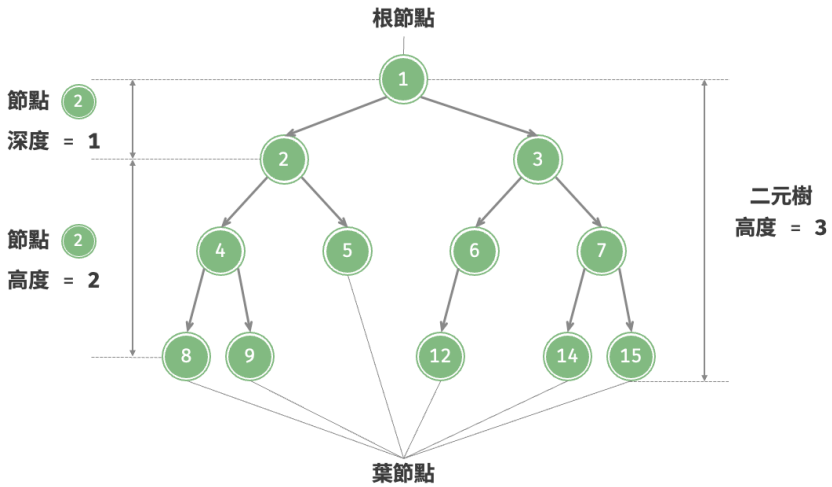


圖 7-2　二元樹的常用術語

> **Tip**
> 請注意，我們通常將“高度”和“深度”定義為“經過的邊的數量”，但有些題目或教材可能會將其定義為“經過的節點的數量”。在這種情況下，高度和深度都需要加 1 。

7.1.2　二元樹基本操作

1.　初始化二元樹

與鏈結串列類似，首先初始化節點，然後構建引用（指標）。

```
// === File: binary_tree.rs ===

// 初始化節點
let n1 = TreeNode::new(1);
let n2 = TreeNode::new(2);
let n3 = TreeNode::new(3);
let n4 = TreeNode::new(4);
let n5 = TreeNode::new(5);
// 構建節點之間的引用（指標）
n1.borrow_mut().left = Some(n2.clone());
n1.borrow_mut().right = Some(n3);
n2.borrow_mut().left = Some(n4);
n2.borrow_mut().right = Some(n5);
```

2.　插入與刪除節點

與鏈結串列類似，在二元樹中插入與刪除節點可以透過修改指標來實現。圖 7-3 給出了一個示例。

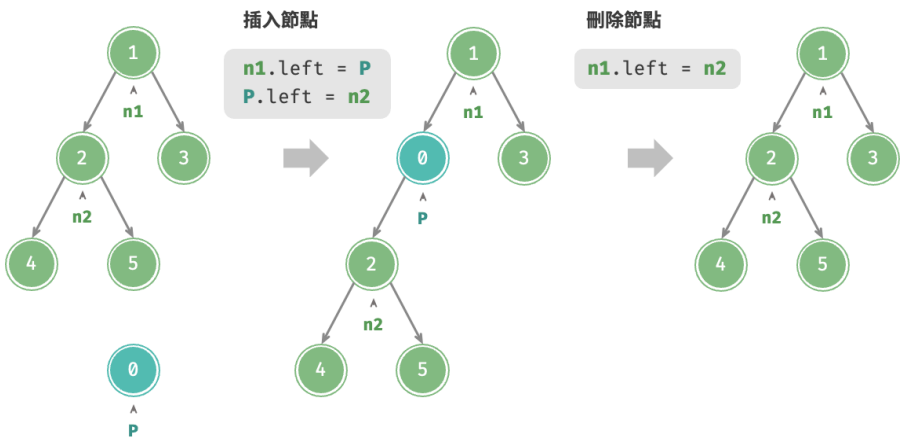


圖 7-3　在二元樹中插入與刪除節點

```
// === File: binary_tree.rs ===

let p = TreeNode::new(0);
// 在 n1 -> n2 中間插入節點 P
n1.borrow_mut().left = Some(p.clone());
p.borrow_mut().left = Some(n2.clone());
// 刪除節點 p
n1.borrow_mut().left = Some(n2);
```

> **Tip**
> 需要注意的是，插入節點可能會改變二元樹的原有邏輯結構，而刪除節點通常意味著刪除該節點及其所有子樹。因此，在二元樹中，插入與刪除通常是由一套操作配合完成的，以實現有實際意義的操作。

7.1.3　常見二元樹型別

1.　完美二元樹

如圖 7-4 所示，完美二元樹（perfect binary tree）所有層的節點都被完全填滿。在完美二元樹中，葉節點的度為 0，其餘所有節點的度都為 2；若樹的高度為 h，則節點總數為 $2^{h+1} - 1$，呈現標準的指數級關係，反映了自然界中常見的細胞分裂現象。

> **Tip**
> 請注意，在中文社群中，完美二元樹常被稱為滿二元樹。

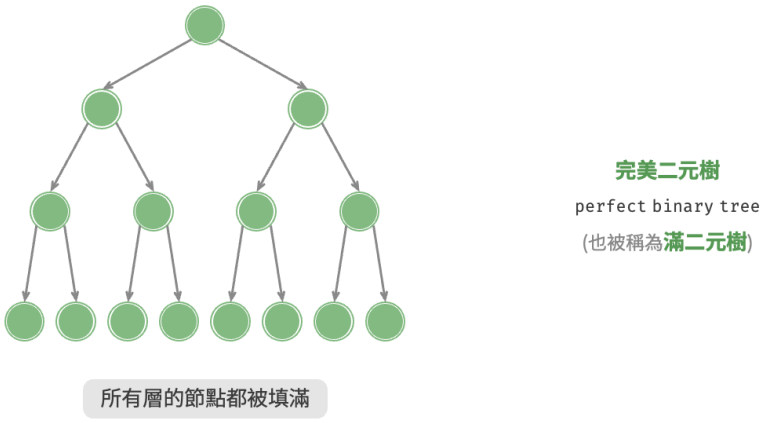


圖 7-4　完美二元樹

2. 完全二元樹

如圖 7-5 所示，**完全二元樹**（complete binary tree）只有最底層的節點未被填滿，且最底層節點儘量靠左填充。請注意，完美二元樹也是一棵完全二元樹。

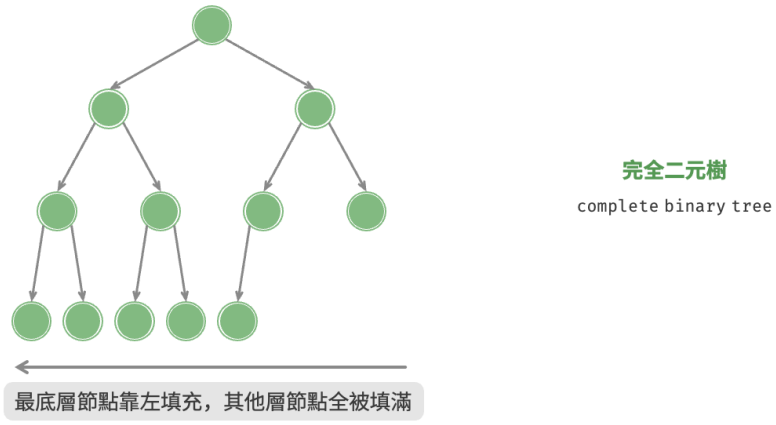


圖 7-5　完全二元樹

3. 完滿二元樹

如圖 7-6 所示，**完滿二元樹**（full binary tree）除了葉節點之外，其餘所有節點都有兩個子節點。

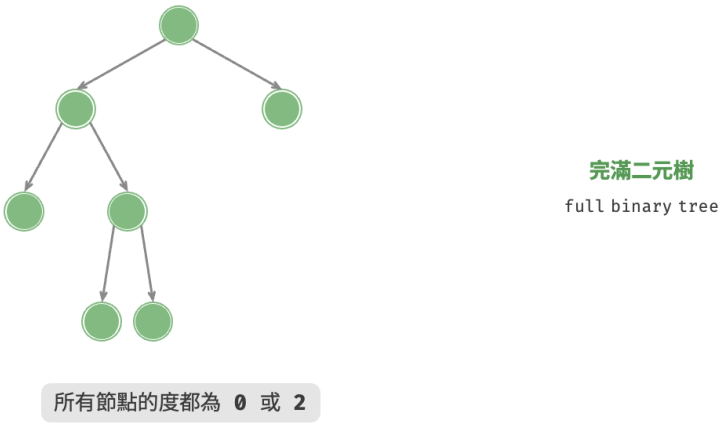


圖 7-6　完滿二元樹

4. 平衡二元樹

如圖 7-7 所示，**平衡二元樹**（balanced binary tree）中任意節點的左子樹和右子樹的高度之差的絕對值不超過 1 。

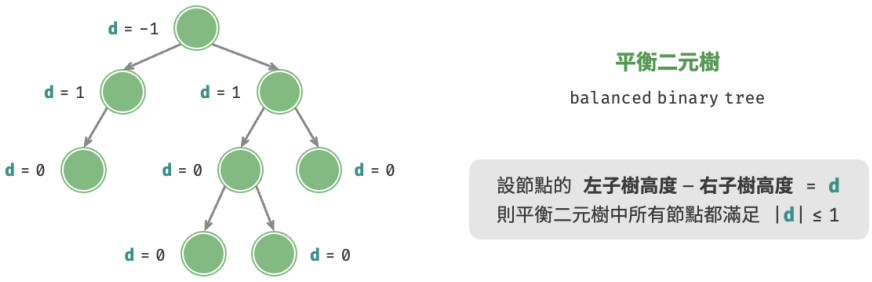


圖 7-7　平衡二元樹

7.1.4　二元樹的退化

圖 7-8 展示了二元樹的理想結構與退化結構。當二元樹的每層節點都被填滿時，達到“完美二元樹”；而當所有節點都偏向一側時，二元樹退化為“鏈結串列”。

- 完美二元樹是理想情況，可以充分發揮二元樹“分治”的優勢。
- 鏈結串列則是另一個極端，各項操作都變為線性操作，時間複雜度退化至 $O(n)$ 。

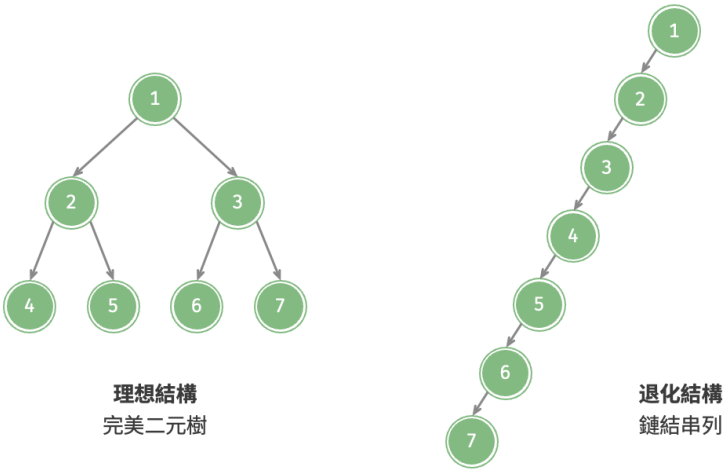


圖 7-8　二元樹的最佳結構與最差結構

如表 7-1 所示，在最佳結構和最差結構下，二元樹的葉節點數量、節點總數、高度等達到極大值或極小值。

表 7-1　二元樹的最佳結構與最差結構

	完美二元樹	鏈結串列
第 i 層的節點數量	2^{i-1}	1
高度為 h 的樹的葉節點數量	2^h	1
高度為 h 的樹的節點總數	$2^{h+1}-1$	$h+1$
節點總數為 n 的樹的高度	$\log_2(n+1)-1$	$n-1$

7.2　二元樹走訪

從物理結構的角度來看，樹是一種基於鏈結串列的資料結構，因此其走訪方式是透過指標逐個訪問節點。然而，樹是一種非線性資料結構，這使得走訪樹比走訪鏈結串列更加複雜，需要藉助搜尋演算法來實現。

二元樹常見的走訪方式包括層序走訪、前序走訪、中序走訪和後序走訪等。

7.2.1　層序走訪

如圖 7-9 所示，**層序走訪**（level-order traversal）從頂部到底部逐層走訪二元樹，並在每一層按照從左到右的順序訪問節點。

層序走訪本質上屬於**廣度優先走訪**（breadth-first traversal），也稱**廣度優先搜尋**（breadth-first search, BFS），它體現了一種“一圈一圈向外擴展”的逐層走訪方式。

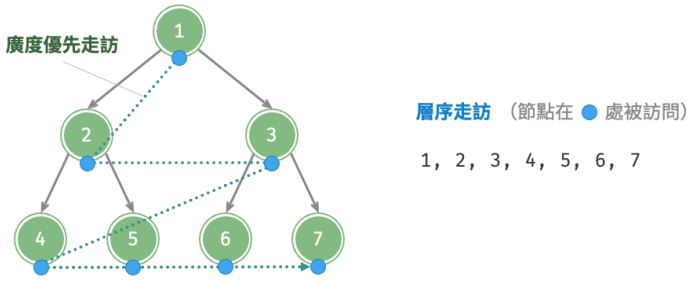


圖 7-9　二元樹的層序走訪

1.　程式碼實現

廣度優先走訪通常藉助“佇列”來實現。佇列遵循“先進先出”的規則，而廣度優先走訪則遵循“逐層推進”的規則，兩者背後的思想是一致的。實現程式碼如下：

```
// === File: binary_tree_bfs.rs ===

/* 層序走訪 */
```

```rust
fn level_order(root: &Rc<RefCell<TreeNode>>) -> Vec<i32> {
    // 初始化佇列，加入根節點
    let mut que = VecDeque::new();
    que.push_back(root.clone());
    // 初始化一個串列，用於儲存走訪序列
    let mut vec = Vec::new();

    while let Some(node) = que.pop_front() {
        // 隊列出隊
        vec.push(node.borrow().val); // 儲存節點值
        if let Some(left) = node.borrow().left.as_ref() {
            que.push_back(left.clone()); // 左子節點入列
        }
        if let Some(right) = node.borrow().right.as_ref() {
            que.push_back(right.clone()); // 右子節點入列
        };
    }
    vec
}
```

2. 複雜度分析

- **時間複雜度為 $O(n)$**：所有節點被訪問一次，使用 $O(n)$ 時間，其中 n 為節點數量。
- **空間複雜度為 $O(n)$**：在最差情況下，即滿二元樹時，走訪到最底層之前，佇列中最多同時存在 $(n+1)/2$ 個節點，佔用 $O(n)$ 空間。

7.2.2　前序、中序、後序走訪

相應地，前序、中序和後序走訪都屬於深度優先走訪（depth-first traversal），也稱深度優先搜尋（depth-first search, DFS），它體現了一種“先走到盡頭，再回溯繼續”的走訪方式。

圖 7-10 展示了對二元樹進行深度優先走訪的工作原理。**深度優先走訪就像是繞著整棵二元樹的外圍“走”一圈**，在每個節點都會遇到三個位置，分別對應前序走訪、中序走訪和後序走訪。

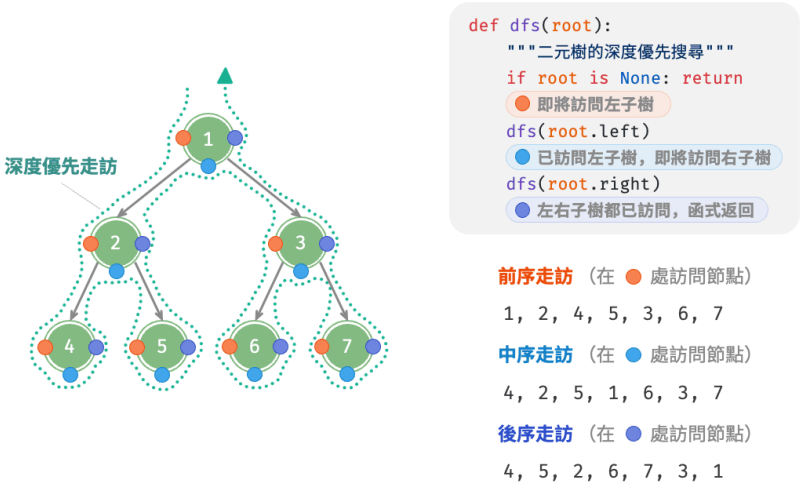


圖 7-10　二元搜尋樹的前序、中序、後序走訪

1. 程式碼實現

深度優先搜尋通常基於遞迴實現：

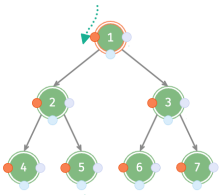
```
// === File: binary_tree_dfs.rs ===

/* 前序走訪 */
fn pre_order(root: Option<&Rc<RefCell<TreeNode>>>) -> Vec<i32> {
    let mut result = vec![];

    fn dfs(root: Option<&Rc<RefCell<TreeNode>>>, res: &mut Vec<i32>) {
        if let Some(node) = root {
            // 訪問優先順序：根節點 -> 左子樹 -> 右子樹
            let node = node.borrow();
            res.push(node.val);
            dfs(node.left.as_ref(), res);
            dfs(node.right.as_ref(), res);
        }
    }
    dfs(root, &mut result);

    result
}

/* 中序走訪 */
fn in_order(root: Option<&Rc<RefCell<TreeNode>>>) -> Vec<i32> {
    let mut result = vec![];
```

```
    fn dfs(root: Option<&Rc<RefCell<TreeNode>>>, res: &mut Vec<i32>) {
        if let Some(node) = root {
            // 訪問優先順序：左子樹 -> 根節點 -> 右子樹
            let node = node.borrow();
            dfs(node.left.as_ref(), res);
            res.push(node.val);
            dfs(node.right.as_ref(), res);
        }
    }
    dfs(root, &mut result);

    result
}

/* 後序走訪 */
fn post_order(root: Option<&Rc<RefCell<TreeNode>>>) -> Vec<i32> {
    let mut result = vec![];

    fn dfs(root: Option<&Rc<RefCell<TreeNode>>>, res: &mut Vec<i32>) {
        if let Some(node) = root {
            // 訪問優先順序：左子樹 -> 右子樹 -> 根節點
            let node = node.borrow();
            dfs(node.left.as_ref(), res);
            dfs(node.right.as_ref(), res);
            res.push(node.val);
        }
    }

    dfs(root, &mut result);

    result
}
```

> **Tip**
> 深度優先搜尋也可以基於迭代實現，有興趣的讀者可以自行研究。

圖 7-11 展示了前序走訪二元樹的遞迴過程，其可分為“遞”和“迴”兩個逆向的部分。

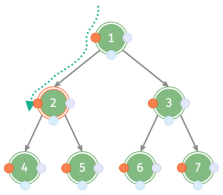
1. “遞”表示開啟新方法，程式在此過程中訪問下一個節點。
2. “迴”表示函式返回，代表當前節點已經訪問完畢。



遞：向下遞推
迴：向上回溯

前序走訪（在 ● 處訪問節點）
1

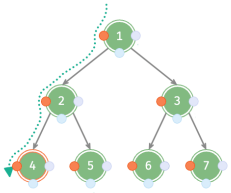
Step 1



遞：向下遞推
迴：向上回溯

前序走訪（在 ● 處訪問節點）
1, 2

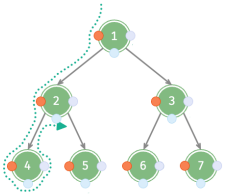
Step 2



遞：向下遞推
迴：向上回溯

前序走訪（在 ● 處訪問節點）
1, 2, 4

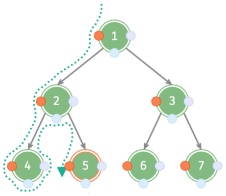
Step 3



遞：向下遞推
迴：向上回溯

前序走訪（在 ● 處訪問節點）
1, 2, 4

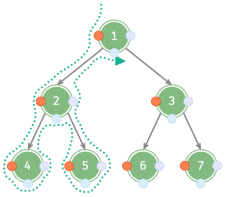
Step 4



遞：向下遞推
迴：向上回溯

前序走訪（在 ● 處訪問節點）
1, 2, 4, 5

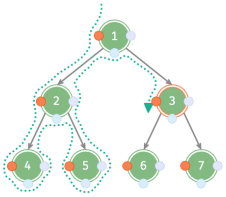
Step 5



遞：向下遞推
迴：向上回溯

前序走訪（在 ● 處訪問節點）
1, 2, 4, 5

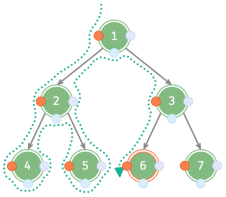
Step 6



遞：向下遞推
迴：向上回溯

前序走訪（在 ● 處訪問節點）
1, 2, 4, 5, 3

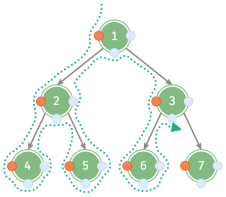
Step 7



遞：向下遞推
迴：向上回溯

前序走訪（在 ● 處訪問節點）
1, 2, 4, 5, 3, 6

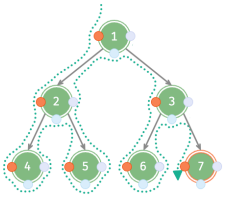
Step 8



遞：向下遞推
迴：向上回溯

前序走訪（在 ● 處訪問節點）
1, 2, 4, 5, 3, 6

Step 9



遞：向下遞推
迴：向上回溯

前序走訪（在 ● 處訪問節點）
1, 2, 4, 5, 3, 6, 7

Step 10

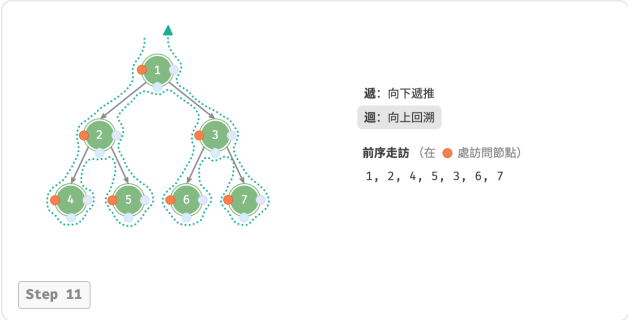


圖 7-11　前序走訪的遞迴過程

2. 複雜度分析

- **時間複雜度為 $O(n)$**：所有節點被訪問一次，使用 $O(n)$ 時間。
- **空間複雜度為 $O(n)$**：在最差情況下，即樹退化為鏈結串列時，遞迴深度達到 n，系統佔用 $O(n)$ 堆疊幀空間。

7.3　二元樹陣列表示

在鏈結串列表示下，二元樹的儲存單元為節點 `TreeNode`，節點之間透過指標相連線。上一節介紹了鏈結串列表示下的二元樹的各項基本操作。

那麼，我們能否用陣列來表示二元樹呢？答案是肯定的。

7.3.1　表示完美二元樹

先分析一個簡單案例。給定一棵完美二元樹，我們將所有節點按照層序走訪的順序儲存在一個陣列中，則每個節點都對應唯一的陣列索引。

根據層序走訪的特性，我們可以推導出父節點索引與子節點索引之間的“對映公式”：**若某節點的索引為 i，則該節點的左子節點索引為 $2i + 1$，右子節點索引為 $2i + 2$**。圖 7-12 展示了各個節點索引之間的對映關係。

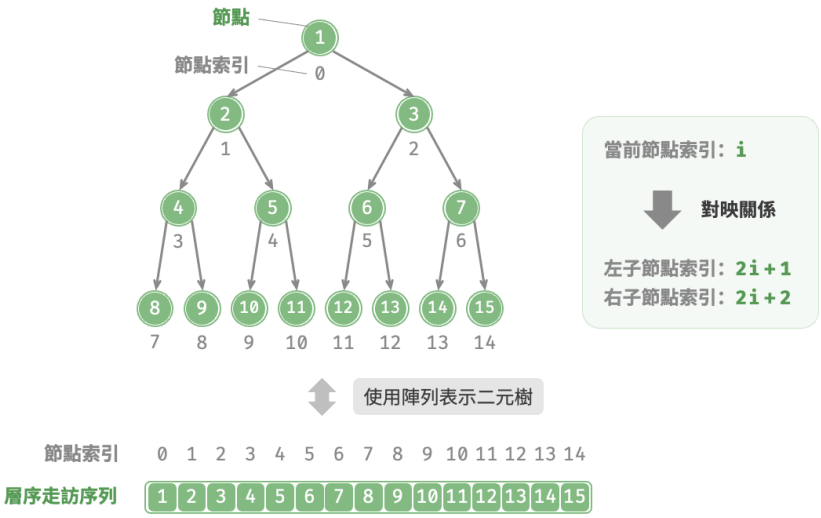


圖 7-12　完美二元樹的陣列表示

對映公式的角色相當於鏈結串列中的節點引用（指標）。給定陣列中的任意一個節點，我們都可以透過對映公式來訪問它的左（右）子節點。

7.3.2　表示任意二元樹

完美二元樹是一個特例，在二元樹的中間層通常存在許多 None 。由於層序走訪序列並不包含這些 None ，因此我們無法僅憑該序列來推測 None 的數量和分佈位置。**這意味著存在多種二元樹結構都符合該層序走訪序列**。

如圖 7-13 所示，給定一棵非完美二元樹，上述陣列表示方法已經失效。

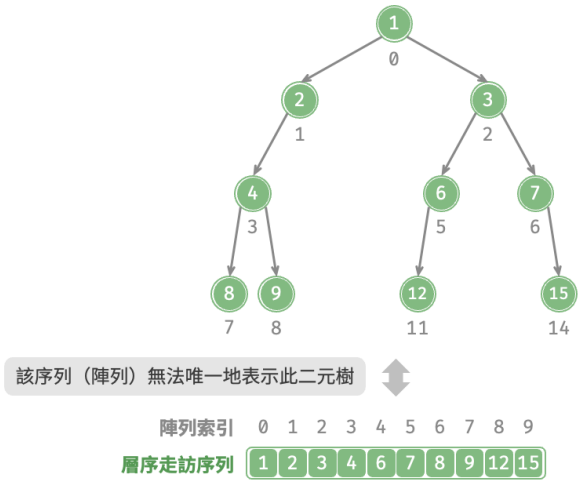


圖 7-13　層序走訪序列對應多種二元樹可能性

為了解決此問題，**我們可以考慮在層序走訪序列中顯式地寫出所有 `None`**。如圖 7-14 所示，這樣處理後，層序走訪序列就可以唯一表示二元樹了。示例程式碼如下：

```
/* 二元樹的陣列表示 */
// 使用 None 來標記空位
let tree = [Some(1), Some(2), Some(3), Some(4), None, Some(6), Some(7), Some(8), Some(9), None, None,
↪  Some(12), None, None, Some(15)];
```

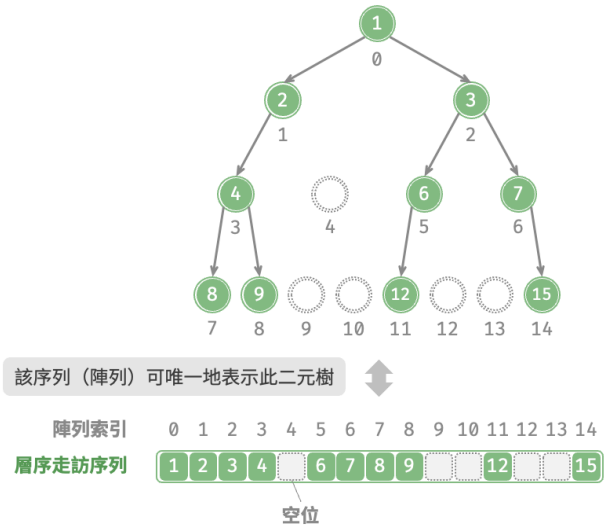


圖 7-14　任意型別二元樹的陣列表示

值得說明的是，**完全二元樹非常適合使用陣列來表示**。回顧完全二元樹的定義，`None` 只出現在最底層且靠右的位置，**因此所有 `None` 一定出現在層序走訪序列的末尾**。

這意味著使用陣列表示完全二元樹時，可以省略儲存所有 `None`，非常方便。圖 7-15 給出了一個例子。

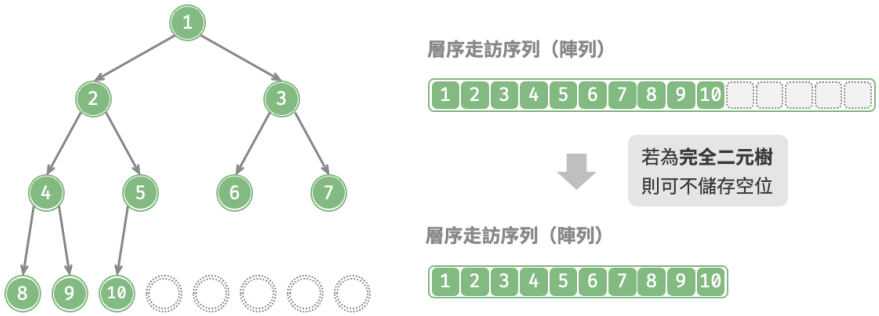


圖 7-15　完全二元樹的陣列表示

以下程式碼實現了一棵基於陣列表示的二元樹，包括以下幾種操作。

- 給定某節點，獲取它的值、左（右）子節點、父節點。
- 獲取前序走訪、中序走訪、後序走訪、層序走訪序列。

```rust
// === File: array_binary_tree.rs ===

/* 陣列表示下的二元樹類別 */
struct ArrayBinaryTree {
    tree: Vec<Option<i32>>,
}

impl ArrayBinaryTree {
    /* 建構子 */
    fn new(arr: Vec<Option<i32>>) -> Self {
        Self { tree: arr }
    }

    /* 串列容量 */
    fn size(&self) -> i32 {
        self.tree.len() as i32
    }

    /* 獲取索引為 i 節點的值 */
    fn val(&self, i: i32) -> Option<i32> {
        // 若索引越界，則返回 None ，代表空位
        if i < 0 || i >= self.size() {
            None
        } else {
            self.tree[i as usize]
        }
    }

    /* 獲取索引為 i 節點的左子節點的索引 */
    fn left(&self, i: i32) -> i32 {
        2 * i + 1
    }

    /* 獲取索引為 i 節點的右子節點的索引 */
    fn right(&self, i: i32) -> i32 {
        2 * i + 2
    }

    /* 獲取索引為 i 節點的父節點的索引 */
    fn parent(&self, i: i32) -> i32 {
        (i - 1) / 2
    }

    /* 層序走訪 */
```

```
    fn level_order(&self) -> Vec<i32> {
        self.tree.iter().filter_map(|&x| x).collect()
    }

    /* 深度優先走訪 */
    fn dfs(&self, i: i32, order: &'static str, res: &mut Vec<i32>) {
        if self.val(i).is_none() {
            return;
        }
        let val = self.val(i).unwrap();
        // 前序走訪
        if order == "pre" {
            res.push(val);
        }
        self.dfs(self.left(i), order, res);
        // 中序走訪
        if order == "in" {
            res.push(val);
        }
        self.dfs(self.right(i), order, res);
        // 後序走訪
        if order == "post" {
            res.push(val);
        }
    }

    /* 前序走訪 */
    fn pre_order(&self) -> Vec<i32> {
        let mut res = vec![];
        self.dfs(0, "pre", &mut res);
        res
    }

    /* 中序走訪 */
    fn in_order(&self) -> Vec<i32> {
        let mut res = vec![];
        self.dfs(0, "in", &mut res);
        res
    }

    /* 後序走訪 */
    fn post_order(&self) -> Vec<i32> {
        let mut res = vec![];
        self.dfs(0, "post", &mut res);
        res
    }
}
```

7.3.3　優點與侷限性

二元樹的陣列表示主要有以下優點。

- 陣列儲存在連續的記憶體空間中，對快取友好，訪問與走訪速度較快。
- 不需要儲存指標，比較節省空間。
- 允許隨機訪問節點。

然而，陣列表示也存在一些侷限性。

- 陣列儲存需要連續記憶體空間，因此不適合儲存資料量過大的樹。
- 增刪節點需要透過陣列插入與刪除操作實現，效率較低。
- 當二元樹中存在大量 `None` 時，陣列中包含的節點資料比重較低，空間利用率較低。

7.4　二元搜尋樹

如圖 7-16 所示，**二元搜尋樹**（binary search tree）滿足以下條件。

1. 對於根節點，左子樹中所有節點的值 $<$ 根節點的值 $<$ 右子樹中所有節點的值。
2. 任意節點的左、右子樹也是二元搜尋樹，即同樣滿足條件 `1.` 。

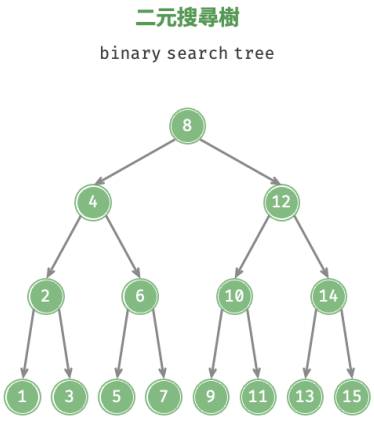


圖 7-16　二元搜尋樹

7.4.1　二元搜尋樹的操作

我們將二元搜尋樹封裝為一個類別 `BinarySearchTree` ，並宣告一個成員變數 `root` ，指向樹的根節點。

1.　查詢節點

給定目標節點值 `num` ，可以根據二元搜尋樹的性質來查詢。如圖 7-17 所示，我們宣告一個節點 `cur` ，從二元樹的根節點 `root` 出發，迴圈比較節點值 `cur.val` 和 `num` 之間的大小關係。

- 若 `cur.val < num`，說明目標節點在 `cur` 的右子樹中，因此執行 `cur = cur.right`。
- 若 `cur.val > num`，說明目標節點在 `cur` 的左子樹中，因此執行 `cur = cur.left`。
- 若 `cur.val = num`，說明找到目標節點，跳出迴圈並返回該節點。

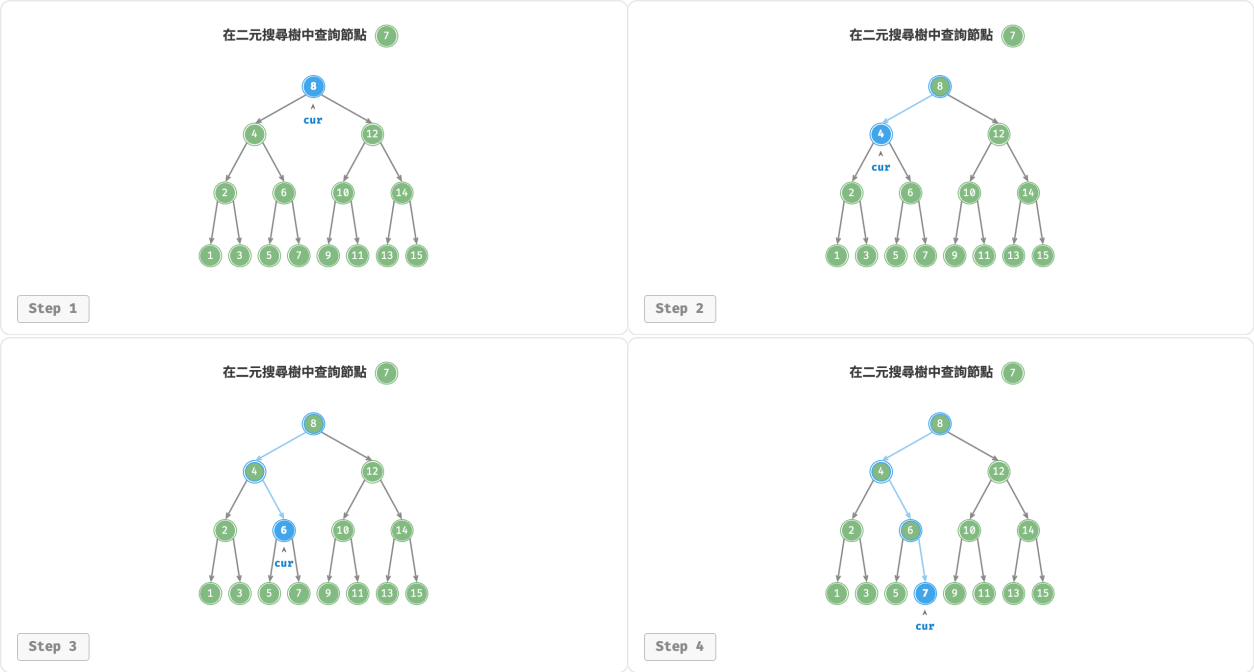


圖 7-17　二元搜尋樹查詢節點示例

二元搜尋樹的查詢操作與二分搜尋演算法的工作原理一致，都是每輪排除一半情況。迴圈次數最多為二元樹的高度，當二元樹平衡時，使用 $O(\log n)$ 時間。示例程式碼如下：

```
// === File: binary_search_tree.rs ===

/* 查詢節點 */
pub fn search(&self, num: i32) -> OptionTreeNodeRc {
    let mut cur = self.root.clone();
    // 迴圈查詢，越過葉節點後跳出
    while let Some(node) = cur.clone() {
        match num.cmp(&node.borrow().val) {
            // 目標節點在 cur 的右子樹中
            Ordering::Greater => cur = node.borrow().right.clone(),
            // 目標節點在 cur 的左子樹中
            Ordering::Less => cur = node.borrow().left.clone(),
            // 找到目標節點，跳出迴圈
            Ordering::Equal => break,
        }
    }

    // 返回目標節點
```

```
    cur
}
```

2. 插入節點

給定一個待插入元素 `num`，為了保持二元搜尋樹“左子樹 < 根節點 < 右子樹”的性質，插入操作流程如圖 7-18 所示。

1. **查詢插入位置**：與查詢操作相似，從根節點出發，根據當前節點值和 `num` 的大小關係迴圈向下搜尋，直到越過葉節點（走訪至 `None`）時跳出迴圈。
2. **在該位置插入節點**：初始化節點 `num`，將該節點置於 `None` 的位置。

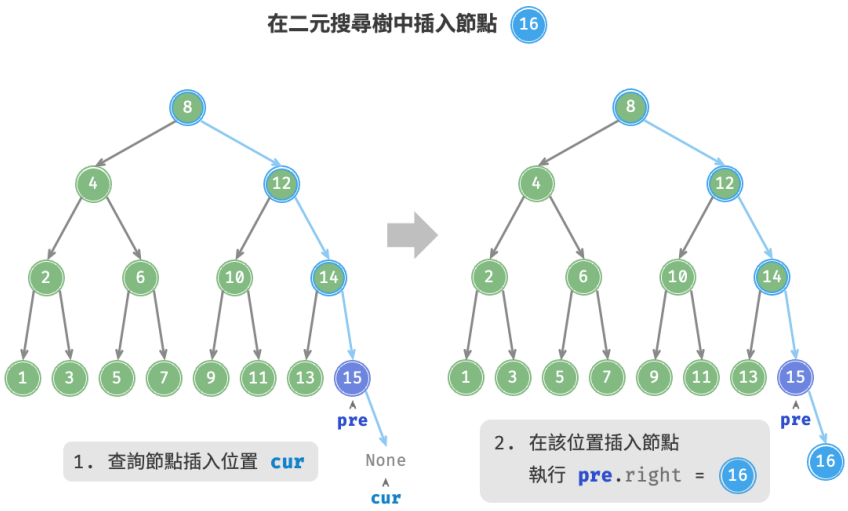


圖 7-18　在二元搜尋樹中插入節點

在程式碼實現中，需要注意以下兩點。

- 二元搜尋樹不允許存在重複節點，否則將違反其定義。因此，若待插入節點在樹中已存在，則不執行插入，直接返回。
- 為了實現插入節點，我們需要藉助節點 `pre` 儲存上一輪迴圈的節點。這樣在走訪至 `None` 時，我們可以獲取到其父節點，從而完成節點插入操作。

```
// === File: binary_search_tree.rs ===

/* 插入節點 */
pub fn insert(&mut self, num: i32) {
    // 若樹為空，則初始化根節點
    if self.root.is_none() {
        self.root = Some(TreeNode::new(num));
```

```
        return;
    }
    let mut cur = self.root.clone();
    let mut pre = None;
    // 迴圈查詢，越過葉節點後跳出
    while let Some(node) = cur.clone() {
        match num.cmp(&node.borrow().val) {
            // 找到重複節點，直接返回
            Ordering::Equal => return,
            // 插入位置在 cur 的右子樹中
            Ordering::Greater => {
                pre = cur.clone();
                cur = node.borrow().right.clone();
            }
            // 插入位置在 cur 的左子樹中
            Ordering::Less => {
                pre = cur.clone();
                cur = node.borrow().left.clone();
            }
        }
    }
    // 插入節點
    let pre = pre.unwrap();
    let node = Some(TreeNode::new(num));
    if num > pre.borrow().val {
        pre.borrow_mut().right = node;
    } else {
        pre.borrow_mut().left = node;
    }
}
```

與查詢節點相同，插入節點使用 $O(\log n)$ 時間。

3. 刪除節點

先在二元樹中查詢到目標節點，再將其刪除。與插入節點類似，我們需要保證在刪除操作完成後，二元搜尋樹的“左子樹 < 根節點 < 右子樹”的性質仍然滿足。因此，我們根據目標節點的子節點數量，分 0、1 和 2 三種情況，執行對應的刪除節點操作。

如圖 7-19 所示，當待刪除節點的度為 0 時，表示該節點是葉節點，可以直接刪除。

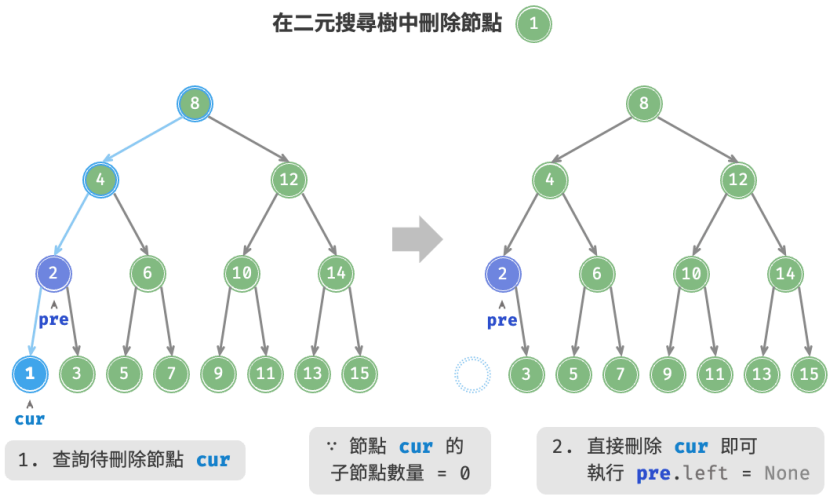


圖 7-19　在二元搜尋樹中刪除節點（度為 0 ）

如圖 7-20 所示，當待刪除節點的度為 1 時，將待刪除節點替換為其子節點即可。

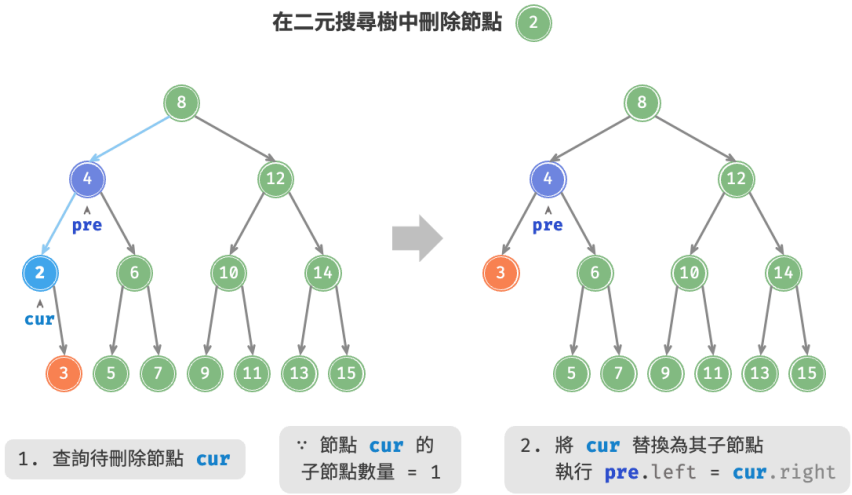


圖 7-20　在二元搜尋樹中刪除節點（度為 1 ）

當待刪除節點的度為 2 時，我們無法直接刪除它，而需要使用一個節點替換該節點。由於要保持二元搜尋樹“左子樹 < 根節點 < 右子樹”的性質，**因此這個節點可以是右子樹的最小節點或左子樹的最大節點**。

假設我們選擇右子樹的最小節點（中序走訪的下一個節點），則刪除操作流程如圖 7-21 所示。

1. 找到待刪除節點在“中序走訪序列”中的下一個節點，記為 `tmp` 。
2. 用 `tmp` 的值覆蓋待刪除節點的值，並在樹中遞迴刪除節點 `tmp` 。

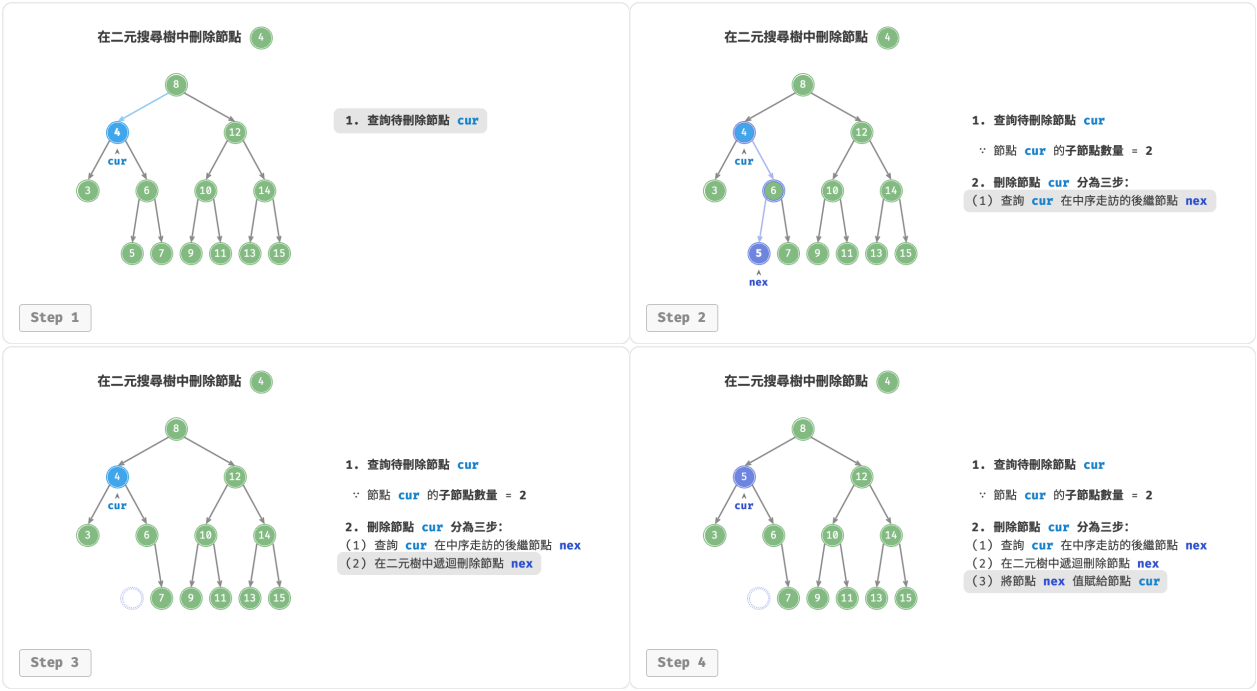


圖 7-21　在二元搜尋樹中刪除節點（度為 2 ）

刪除節點操作同樣使用 $O(\log n)$ 時間，其中查詢待刪除節點需要 $O(\log n)$ 時間，獲取中序走訪後繼節點需要 $O(\log n)$ 時間。示例程式碼如下：

```
// === File: binary_search_tree.rs ===

/* 刪除節點 */
pub fn remove(&mut self, num: i32) {
    // 若樹為空，直接提前返回
    if self.root.is_none() {
        return;
    }
    let mut cur = self.root.clone();
    let mut pre = None;
    // 迴圈查詢，越過葉節點後跳出
    while let Some(node) = cur.clone() {
        match num.cmp(&node.borrow().val) {
            // 找到待刪除節點，跳出迴圈
            Ordering::Equal => break,
            // 待刪除節點在 cur 的右子樹中
            Ordering::Greater => {
                pre = cur.clone();
                cur = node.borrow().right.clone();
            }
            // 待刪除節點在 cur 的左子樹中
            Ordering::Less => {
```

```
                pre = cur.clone();
                cur = node.borrow().left.clone();
            }
        }
    }
    // 若無待刪除節點，則直接返回
    if cur.is_none() {
        return;
    }
    let cur = cur.unwrap();
    let (left_child, right_child) = (cur.borrow().left.clone(), cur.borrow().right.clone());
    match (left_child.clone(), right_child.clone()) {
        // 子節點數量 = 0 or 1
        (None, None) | (Some(_), None) | (None, Some(_)) => {
            // 當子節點數量 = 0 / 1 時， child = nullptr / 該子節點
            let child = left_child.or(right_child);
            let pre = pre.unwrap();
            // 刪除節點 cur
            if !Rc::ptr_eq(&cur, self.root.as_ref().unwrap()) {
                let left = pre.borrow().left.clone();
                if left.is_some() && Rc::ptr_eq(left.as_ref().unwrap(), &cur) {
                    pre.borrow_mut().left = child;
                } else {
                    pre.borrow_mut().right = child;
                }
            } else {
                // 若刪除節點為根節點，則重新指定根節點
                self.root = child;
            }
        }
        // 子節點數量 = 2
        (Some(_), Some(_)) => {
            // 獲取中序走訪中 cur 的下一個節點
            let mut tmp = cur.borrow().right.clone();
            while let Some(node) = tmp.clone() {
                if node.borrow().left.is_some() {
                    tmp = node.borrow().left.clone();
                } else {
                    break;
                }
            }
            let tmp_val = tmp.unwrap().borrow().val;
            // 遞迴刪除節點 tmp
            self.remove(tmp_val);
            // 用 tmp 覆蓋 cur
            cur.borrow_mut().val = tmp_val;
        }
```

```
    }
}
```

4. 中序走訪有序

如圖 7-22 所示，二元樹的中序走訪遵循“左 → 根 → 右”的走訪順序，而二元搜尋樹滿足“左子節點 < 根節點 < 右子節點”的大小關係。

這意味著在二元搜尋樹中進行中序走訪時，總是會優先走訪下一個最小節點，從而得出一個重要性質：**二元搜尋樹的中序走訪序列是升序的**。

利用中序走訪升序的性質，我們在二元搜尋樹中獲取有序資料僅需 $O(n)$ 時間，無須進行額外的排序操作，非常高效。

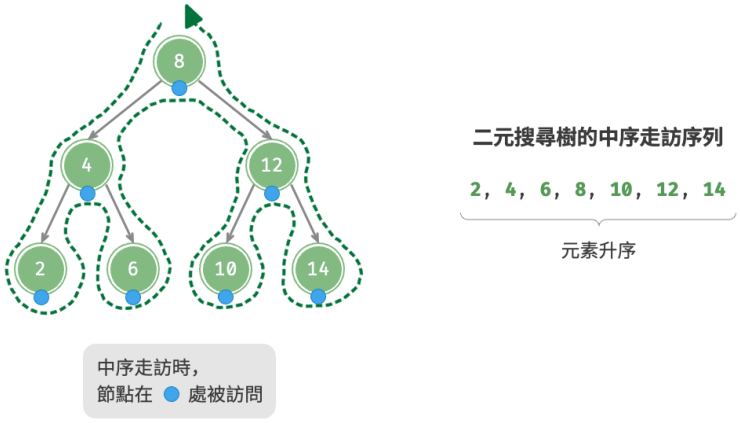


圖 7-22　二元搜尋樹的中序走訪序列

7.4.2　二元搜尋樹的效率

給定一組資料，我們考慮使用陣列或二元搜尋樹儲存。觀察表 7-2 ，二元搜尋樹的各項操作的時間複雜度都是對數階，具有穩定且高效的效能。只有在高頻新增、低頻查詢刪除資料的場景下，陣列比二元搜尋樹的效率更高。

表 7-2　陣列與搜尋樹的效率對比

	無序陣列	二元搜尋樹
查詢元素	$O(n)$	$O(\log n)$
插入元素	$O(1)$	$O(\log n)$
刪除元素	$O(n)$	$O(\log n)$

無序陣列	二元搜尋樹

在理想情況下，二元搜尋樹是“平衡”的，這樣就可以在 $\log n$ 輪迴圈內查詢任意節點。

然而，如果我們在二元搜尋樹中不斷地插入和刪除節點，可能導致二元樹退化為圖 7-23 所示的鏈結串列，這時各種操作的時間複雜度也會退化為 $O(n)$ 。

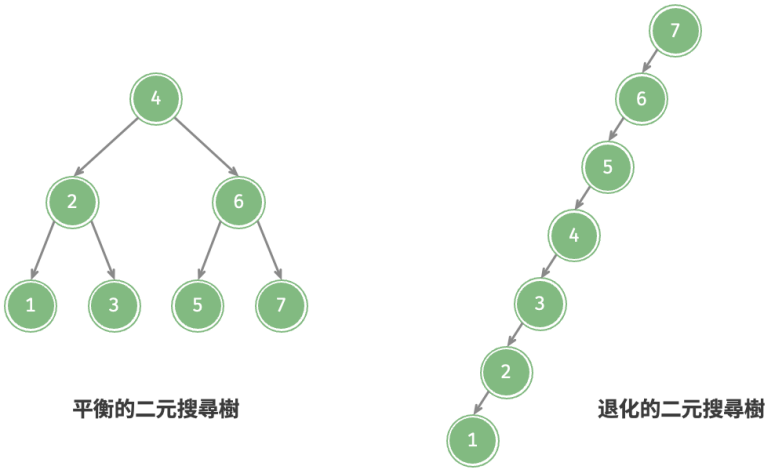


圖 7-23　二元搜尋樹退化

7.4.3　二元搜尋樹常見應用

- 用作系統中的多級索引，實現高效的查詢、插入、刪除操作。
- 作為某些搜尋演算法的底層資料結構。
- 用於儲存資料流，以保持其有序狀態。

7.5　AVL 樹 *

在“二元搜尋樹”章節中我們提到，在多次插入和刪除操作後，二元搜尋樹可能退化為鏈結串列。在這種情況下，所有操作的時間複雜度將從 $O(\log n)$ 劣化為 $O(n)$ 。

如圖 7-24 所示，經過兩次刪除節點操作，這棵二元搜尋樹便會退化為鏈結串列。

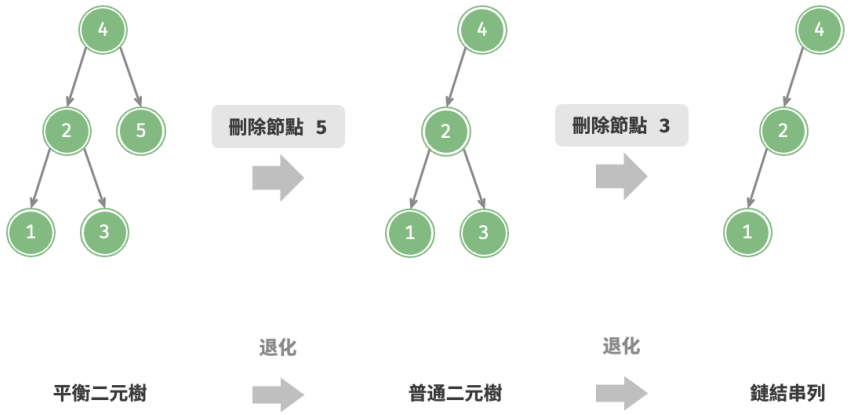


圖 7-24　AVL 樹在刪除節點後發生退化

再例如，在圖 7-25 所示的完美二元樹中插入兩個節點後，樹將嚴重向左傾斜，查詢操作的時間複雜度也隨之劣化。

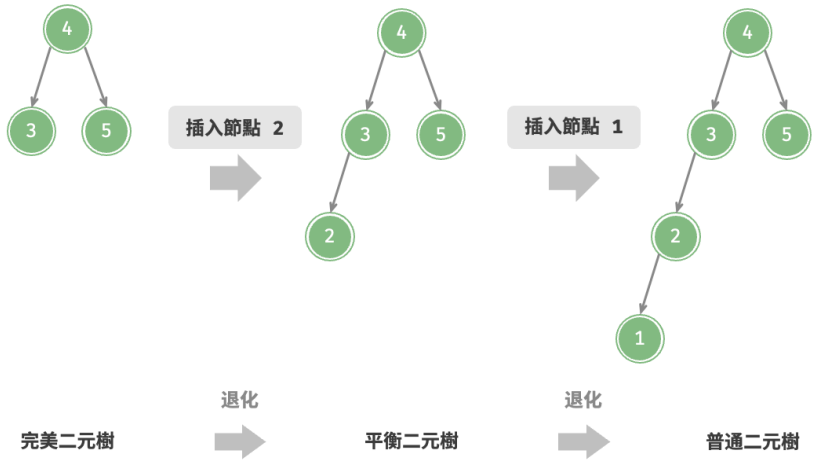


圖 7-25　AVL 樹在插入節點後發生退化

1962 年 G. M. Adelson-Velsky 和 E. M. Landis 在論文“An algorithm for the organization of information”中提出了 AVL 樹。論文中詳細描述了一系列操作，確保在持續新增和刪除節點後，AVL 樹不會退化，從而使得各種操作的時間複雜度保持在 $O(\log n)$ 級別。換句話說，在需要頻繁進行增刪查改操作的場景中，AVL 樹能始終保持高效的資料操作效能，具有很好的應用價值。

7.5.1　AVL 樹常見術語

AVL 樹既是二元搜尋樹，也是平衡二元樹，同時滿足這兩類二元樹的所有性質，因此是一種**平衡二元搜尋樹**（balanced binary search tree）。

1. 節點高度

由於 AVL 樹的相關操作需要獲取節點高度，因此我們需要為節點類別新增 `height` 變數：

```
use std::rc::Rc;
use std::cell::RefCell;

/* AVL 樹節點結構體 */
struct TreeNode {
    val: i32,                                // 節點值
    height: i32,                             // 節點高度
    left: Option<Rc<RefCell<TreeNode>>>,     // 左子節點
    right: Option<Rc<RefCell<TreeNode>>>,    // 右子節點
}

impl TreeNode {
    /* 建構子 */
    fn new(val: i32) -> Rc<RefCell<Self>> {
        Rc::new(RefCell::new(Self {
            val,
            height: 0,
            left: None,
            right: None
        }))
    }
}
```

“節點高度”是指從該節點到它的最遠葉節點的距離，即所經過的“邊”的數量。需要特別注意的是，葉節點的高度為 0 ，而空節點的高度為 -1 。我們將建立兩個工具函式，分別用於獲取和更新節點的高度：

```
// === File: avl_tree.rs ===

/* 獲取節點高度 */
fn height(node: OptionTreeNodeRc) -> i32 {
    // 空節點高度為 -1 ，葉節點高度為 0
    match node {
        Some(node) => node.borrow().height,
        None => -1,
    }
}

/* 更新節點高度 */
fn update_height(node: OptionTreeNodeRc) {
    if let Some(node) = node {
        let left = node.borrow().left.clone();
        let right = node.borrow().right.clone();
        // 節點高度等於最高子樹高度 + 1
```

```
        node.borrow_mut().height = std::cmp::max(Self::height(left), Self::height(right)) + 1;
    }
}
```

2. 節點平衡因子

節點的平衡因子（balance factor）定義為節點左子樹的高度減去右子樹的高度，同時規定空節點的平衡因子為 0 。我們同樣將獲取節點平衡因子的功能封裝成函式，方便後續使用：

```
// === File: avl_tree.rs ===

/* 獲取平衡因子 */
fn balance_factor(node: OptionTreeNodeRc) -> i32 {
    match node {
        // 空節點平衡因子為 0
        None => 0,
        // 節點平衡因子 = 左子樹高度 - 右子樹高度
        Some(node) => {
            Self::height(node.borrow().left.clone()) - Self::height(node.borrow().right.clone())
        }
    }
}
```

> **Tip**
> 設平衡因子為 f ，則一棵 AVL 樹的任意節點的平衡因子皆滿足 $-1 \leq f \leq 1$ 。

7.5.2 AVL 樹旋轉

AVL 樹的特點在於“旋轉”操作，它能夠在不影響二元樹的中序走訪序列的前提下，使失衡節點重新恢復平衡。換句話說，**旋轉操作既能保持“二元搜尋樹”的性質，也能使樹重新變為“平衡二元樹”**。

我們將平衡因子絕對值 > 1 的節點稱為“失衡節點”。根據節點失衡情況的不同，旋轉操作分為四種：右旋、左旋、先右旋後左旋、先左旋後右旋。下面詳細介紹這些旋轉操作。

1. 右旋

如圖 7-26 所示，節點下方為平衡因子。從底至頂看，二元樹中首個失衡節點是“節點 3”。我們關注以該失衡節點為根節點的子樹，將該節點記為 `node` ，其左子節點記為 `child` ，執行“右旋”操作。完成右旋後，子樹恢復平衡，並且仍然保持二元搜尋樹的性質。

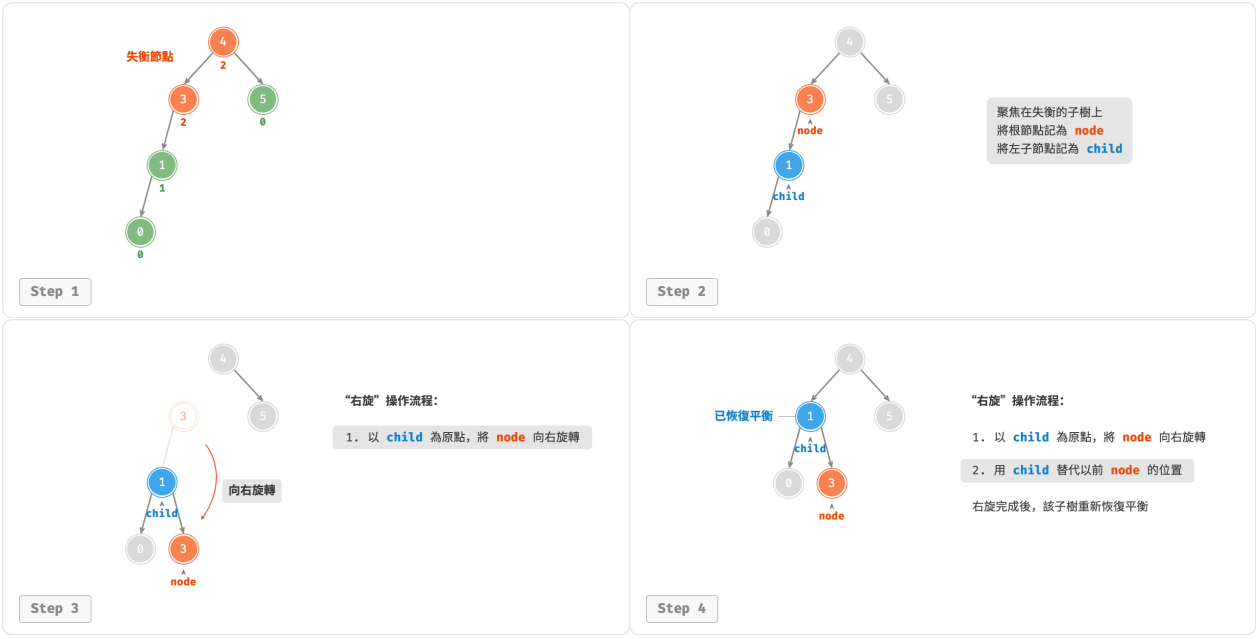


圖 7-26　右旋操作步驟

如圖 7-27 所示，當節點 `child` 有右子節點（記為 `grand_child`）時，需要在右旋中新增一步：將 `grand_child` 作為 `node` 的左子節點。

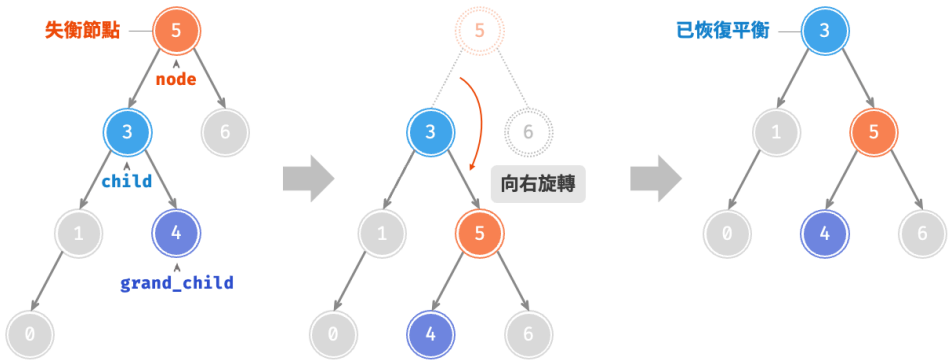


圖 7-27　有 grand_child 的右旋操作

"向右旋轉"是一種形象化的說法，實際上需要透過修改節點指標來實現，程式碼如下所示：

```
// === File: avl_tree.rs ===

/* 右旋操作 */
fn right_rotate(node: OptionTreeNodeRc) -> OptionTreeNodeRc {
    match node {
        Some(node) => {
```

```
            let child = node.borrow().left.clone().unwrap();
            let grand_child = child.borrow().right.clone();
            // 以 child 為原點，將 node 向右旋轉
            child.borrow_mut().right = Some(node.clone());
            node.borrow_mut().left = grand_child;
            // 更新節點高度
            Self::update_height(Some(node));
            Self::update_height(Some(child.clone()));
            // 返回旋轉後子樹的根節點
            Some(child)
        }
        None => None,
    }
}
```

2. 左旋

相應地，如果考慮上述失衡二元樹的“映象”，則需要執行圖 7-28 所示的“左旋”操作。

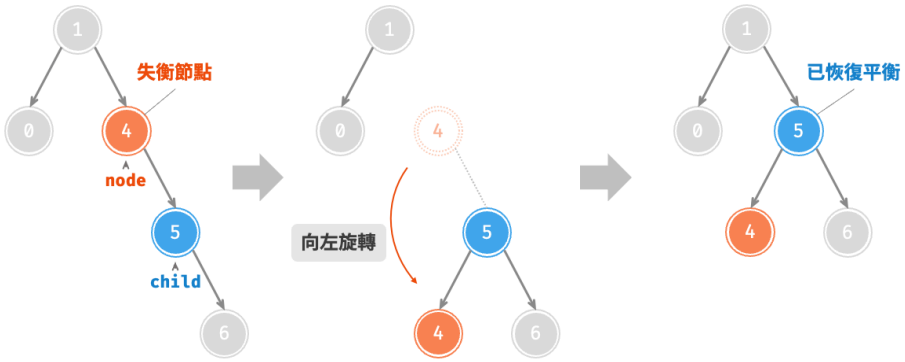


圖 7-28　左旋操作

同理，如圖 7-29 所示，當節點 `child` 有左子節點（記為 `grand_child`）時，需要在左旋中新增一步：將 `grand_child` 作為 `node` 的右子節點。

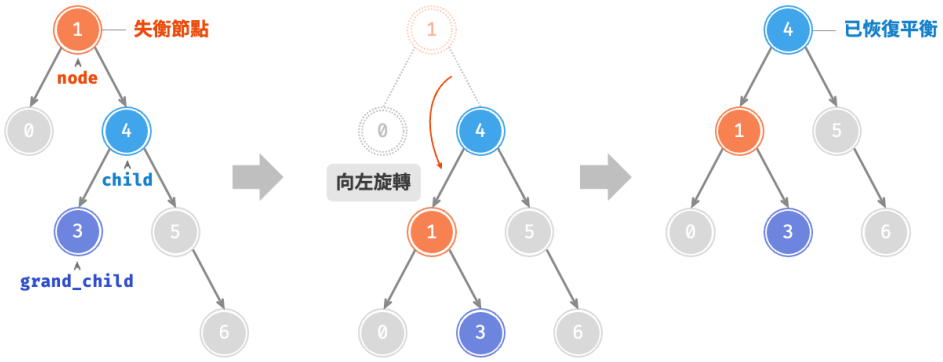


圖 7-29　有 grand_child 的左旋操作

可以觀察到，**右旋和左旋操作在邏輯上是映象對稱的，它們分別解決的兩種失衡情況也是對稱的**。基於對稱性，我們只需將右旋的實現程式碼中的所有的 `left` 替換為 `right`，將所有的 `right` 替換為 `left`，即可得到左旋的實現程式碼：

```
// === File: avl_tree.rs ===

/* 左旋操作 */
fn left_rotate(node: OptionTreeNodeRc) -> OptionTreeNodeRc {
    match node {
        Some(node) => {
            let child = node.borrow().right.clone().unwrap();
            let grand_child = child.borrow().left.clone();
            // 以 child 為原點，將 node 向左旋轉
            child.borrow_mut().left = Some(node.clone());
            node.borrow_mut().right = grand_child;
            // 更新節點高度
            Self::update_height(Some(node));
            Self::update_height(Some(child.clone()));
            // 返回旋轉後子樹的根節點
            Some(child)
        }
        None => None,
    }
}
```

3. 先左旋後右旋

對於圖 7-30 中的失衡節點 3，僅使用左旋或右旋都無法使子樹恢復平衡。此時需要先對 `child` 執行“左旋”，再對 `node` 執行“右旋”。

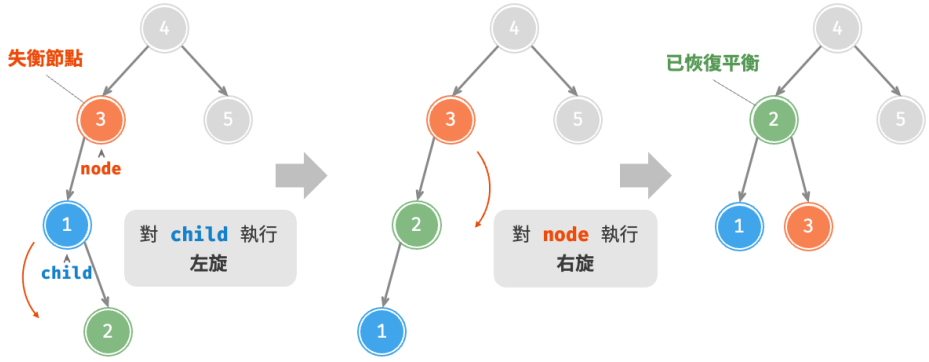


圖 7-30　先左旋後右旋

4. 先右旋後左旋

如圖 7-31 所示，對於上述失衡二元樹的映象情況，需要先對 `child` 執行“右旋”，再對 `node` 執行“左旋”。

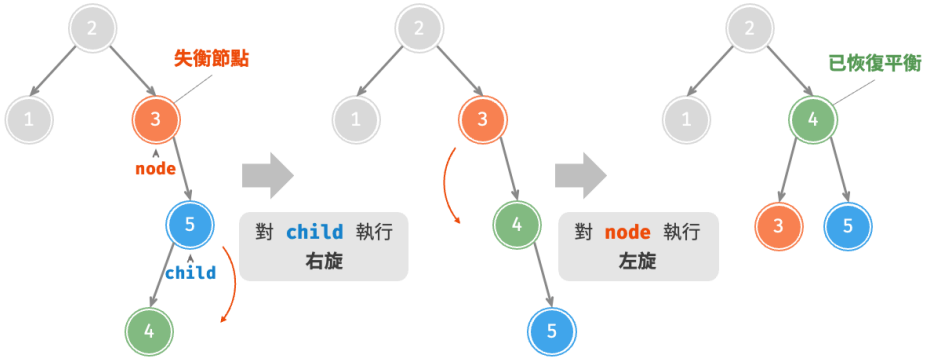


圖 7-31　先右旋後左旋

5. 旋轉的選擇

圖 7-32 展示的四種失衡情況與上述案例逐個對應，分別需要採用右旋、先左旋後右旋、先右旋後左旋、左旋的操作。

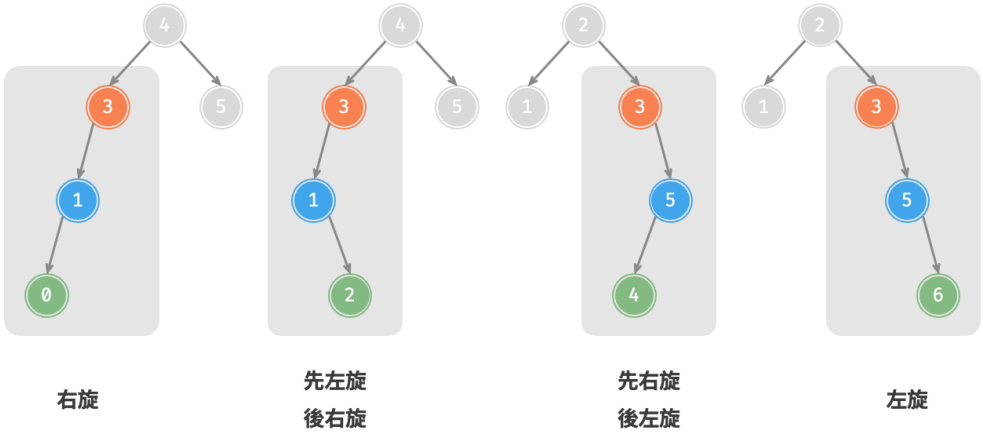


圖 7-32　AVL 樹的四種旋轉情況

如下表所示，我們透過判斷失衡節點的平衡因子以及較高一側子節點的平衡因子的正負號，來確定失衡節點屬於圖 7-32 中的哪種情況。

表 7-3　四種旋轉情況的選擇條件

失衡節點的平衡因子	子節點的平衡因子	應採用的旋轉方法
> 1 （左偏樹）	≥ 0	右旋
> 1 （左偏樹）	< 0	先左旋後右旋
< -1 （右偏樹）	≤ 0	左旋
< -1 （右偏樹）	> 0	先右旋後左旋

為了便於使用，我們將旋轉操作封裝成一個函式。**有了這個函式，我們就能對各種失衡情況進行旋轉，使失衡節點重新恢復平衡**。程式碼如下所示：

```
// === File: avl_tree.rs ===

/* 執行旋轉操作，使該子樹重新恢復平衡 */
fn rotate(node: OptionTreeNodeRc) -> OptionTreeNodeRc {
    // 獲取節點 node 的平衡因子
    let balance_factor = Self::balance_factor(node.clone());
    // 左偏樹
    if balance_factor > 1 {
        let node = node.unwrap();
        if Self::balance_factor(node.borrow().left.clone()) >= 0 {
            // 右旋
            Self::right_rotate(Some(node))
```

```
        } else {
            // 先左旋後右旋
            let left = node.borrow().left.clone();
            node.borrow_mut().left = Self::left_rotate(left);
            Self::right_rotate(Some(node))
        }
    }
    // 右偏樹
    else if balance_factor < -1 {
        let node = node.unwrap();
        if Self::balance_factor(node.borrow().right.clone()) <= 0 {
            // 左旋
            Self::left_rotate(Some(node))
        } else {
            // 先右旋後左旋
            let right = node.borrow().right.clone();
            node.borrow_mut().right = Self::right_rotate(right);
            Self::left_rotate(Some(node))
        }
    } else {
        // 平衡樹，無須旋轉，直接返回
        node
    }
}
```

7.5.3 AVL 樹常用操作

1. 插入節點

AVL 樹的節點插入操作與二元搜尋樹在主體上類似。唯一的區別在於，在 AVL 樹中插入節點後，從該節點到根節點的路徑上可能會出現一系列失衡節點。因此，**我們需要從這個節點開始，自底向上執行旋轉操作，使所有失衡節點恢復平衡**。程式碼如下所示：

```
// === File: avl_tree.rs ===

/* 插入節點 */
fn insert(&mut self, val: i32) {
    self.root = Self::insert_helper(self.root.clone(), val);
}

/* 遞迴插入節點（輔助方法） */
fn insert_helper(node: OptionTreeNodeRc, val: i32) -> OptionTreeNodeRc {
    match node {
        Some(mut node) => {
            /* 1. 查詢插入位置並插入節點 */
```

```
            match {
                let node_val = node.borrow().val;
                node_val
            }
            .cmp(&val)
            {
                Ordering::Greater => {
                    let left = node.borrow().left.clone();
                    node.borrow_mut().left = Self::insert_helper(left, val);
                }
                Ordering::Less => {
                    let right = node.borrow().right.clone();
                    node.borrow_mut().right = Self::insert_helper(right, val);
                }
                Ordering::Equal => {
                    return Some(node); // 重複節點不插入，直接返回
                }
            }
            Self::update_height(Some(node.clone())); // 更新節點高度

            /* 2. 執行旋轉操作，使該子樹重新恢復平衡 */
            node = Self::rotate(Some(node)).unwrap();
            // 返回子樹的根節點
            Some(node)
        }
        None => Some(TreeNode::new(val)),
    }
}
```

2. 刪除節點

類似地，在二元搜尋樹的刪除節點方法的基礎上，需要從底至頂執行旋轉操作，使所有失衡節點恢復平衡。程式碼如下所示：

```
// === File: avl_tree.rs ===

/* 刪除節點 */
fn remove(&self, val: i32) {
    Self::remove_helper(self.root.clone(), val);
}

/* 遞迴刪除節點（輔助方法） */
fn remove_helper(node: OptionTreeNodeRc, val: i32) -> OptionTreeNodeRc {
    match node {
        Some(mut node) => {
            /* 1. 查詢節點並刪除 */
```

```
            if val < node.borrow().val {
                let left = node.borrow().left.clone();
                node.borrow_mut().left = Self::remove_helper(left, val);
            } else if val > node.borrow().val {
                let right = node.borrow().right.clone();
                node.borrow_mut().right = Self::remove_helper(right, val);
            } else if node.borrow().left.is_none() || node.borrow().right.is_none() {
                let child = if node.borrow().left.is_some() {
                    node.borrow().left.clone()
                } else {
                    node.borrow().right.clone()
                };
                match child {
                    // 子節點數量 = 0 ，直接刪除 node 並返回
                    None => {
                        return None;
                    }
                    // 子節點數量 = 1 ，直接刪除 node
                    Some(child) => node = child,
                }
            } else {
                // 子節點數量 = 2 ，則將中序走訪的下個節點刪除，並用該節點替換當前節點
                let mut temp = node.borrow().right.clone().unwrap();
                loop {
                    let temp_left = temp.borrow().left.clone();
                    if temp_left.is_none() {
                        break;
                    }
                    temp = temp_left.unwrap();
                }
                let right = node.borrow().right.clone();
                node.borrow_mut().right = Self::remove_helper(right, temp.borrow().val);
                node.borrow_mut().val = temp.borrow().val;
            }
            Self::update_height(Some(node.clone())); // 更新節點高度

            /* 2. 執行旋轉操作，使該子樹重新恢復平衡 */
            node = Self::rotate(Some(node)).unwrap();
            // 返回子樹的根節點
            Some(node)
        }
        None => None,
    }
}
```

3. 查詢節點

AVL 樹的節點查詢操作與二元搜尋樹一致，在此不再贅述。

7.5.4 AVL 樹典型應用

- 組織和儲存大型資料，適用於高頻查詢、低頻增刪的場景。
- 用於構建資料庫中的索引系統。
- 紅黑樹也是一種常見的平衡二元搜尋樹。相較於 AVL 樹，紅黑樹的平衡條件更寬鬆，插入與刪除節點所需的旋轉操作更少，節點增刪操作的平均效率更高。

7.6 小結

1. 重點回顧

- 二元樹是一種非線性資料結構，體現“一分為二”的分治邏輯。每個二元樹節點包含一個值以及兩個指標，分別指向其左子節點和右子節點。
- 對於二元樹中的某個節點，其左（右）子節點及其以下形成的樹被稱為該節點的左（右）子樹。
- 二元樹的相關術語包括根節點、葉節點、層、度、邊、高度和深度等。
- 二元樹的初始化、節點插入和節點刪除操作與鏈結串列操作方法類似。
- 常見的二元樹型別有完美二元樹、完全二元樹、完滿二元樹和平衡二元樹。完美二元樹是最理想的狀態，而鏈結串列是退化後的最差狀態。
- 二元樹可以用陣列表示，方法是將節點值和空位按層序走訪順序排列，並根據父節點與子節點之間的索引對映關係來實現指標。
- 二元樹的層序走訪是一種廣度優先搜尋方法，它體現了“一圈一圈向外擴展”的逐層走訪方式，通常透過佇列來實現。
- 前序、中序、後序走訪皆屬於深度優先搜尋，它們體現了“先走到盡頭，再回溯繼續”的走訪方式，通常使用遞迴來實現。
- 二元搜尋樹是一種高效的元素查詢資料結構，其查詢、插入和刪除操作的時間複雜度均為 $O(\log n)$ 。當二元搜尋樹退化為鏈結串列時，各項時間複雜度會劣化至 $O(n)$ 。
- AVL 樹，也稱平衡二元搜尋樹，它透過旋轉操作確保在不斷插入和刪除節點後樹仍然保持平衡。
- AVL 樹的旋轉操作包括右旋、左旋、先右旋再左旋、先左旋再右旋。在插入或刪除節點後，AVL 樹會從底向頂執行旋轉操作，使樹重新恢復平衡。

2. Q & A

Q：對於只有一個節點的二元樹，樹的高度和根節點的深度都是 0 嗎？

是的，因為高度和深度通常定義為“經過的邊的數量”。

Q：二元樹中的插入與刪除一般由一套操作配合完成，這裡的“一套操作”指什麼呢？可以理解為資源的子節點的資源釋放嗎？

拿二元搜尋樹來舉例，刪除節點操作要分三種情況處理，其中每種情況都需要進行多個步驟的節點操作。

Q：為什麼 DFS 走訪二元樹有前、中、後三種順序，分別有什麼用呢？

與順序和逆序走訪陣列類似，前序、中序、後序走訪是三種二元樹走訪方法，我們可以使用它們得到一個特定順序的走訪結果。例如在二元搜尋樹中，由於節點大小滿足 `左子節點值 < 根節點值 < 右子節點值`，因此我們只要按照“左 → 根 → 右”的優先順序走訪樹，就可以獲得有序的節點序列。

Q：右旋操作是處理失衡節點 `node`、`child`、`grand_child` 之間的關係，那 `node` 的父節點和 `node` 原來的連線不需要維護嗎？右旋操作後豈不是斷掉了？

我們需要從遞迴的視角來看這個問題。右旋操作 `right_rotate(root)` 傳入的是子樹的根節點，最終 `return child` 返回旋轉之後的子樹的根節點。子樹的根節點和其父節點的連線是在該函式返回後完成的，不屬於右旋操作的維護範圍。

Q：在 C++ 中，函式被劃分到 `private` 和 `public` 中，這方面有什麼考量嗎？為什麼要將 `height()` 函式和 `updateHeight()` 函式分別放在 `public` 和 `private` 中呢？

主要看方法的使用範圍，如果方法只在類別內部使用，那麼就設計為 `private`。例如，使用者單獨呼叫 `updateHeight()` 是沒有意義的，它只是插入、刪除操作中的一步。而 `height()` 是訪問節點高度，類似於 `vector.size()`，因此設定成 `public` 以便使用。

Q：如何從一組輸入資料構建一棵二元搜尋樹？根節點的選擇是不是很重要？

是的，構建樹的方法已在二元搜尋樹程式碼中的 `build_tree()` 方法中給出。至於根節點的選擇，我們通常會將輸入資料排序，然後將中點元素作為根節點，再遞迴地構建左右子樹。這樣做可以最大程度保證樹的平衡性。

Q：在 Java 中，字串對比是否一定要用 `equals()` 方法？

在 Java 中，對於基本資料型別，`==` 用於對比兩個變數的值是否相等。對於引用型別，兩種符號的工作原理是不同的。

- `==`：用來比較兩個變數是否指向同一個物件，即它們在記憶體中的位置是否相同。
- `equals()`：用來對比兩個物件的值是否相等。

因此，如果要對比值，我們應該使用 `equals()`。然而，透過 `String a = "hi"; String b = "hi";` 初始化的字串都儲存在字串常數池中，它們指向同一個物件，因此也可以用 `a == b` 來比較兩個字串的內容。

Q：廣度優先走訪到最底層之前，佇列中的節點數量是 2^h 嗎？

是的，例如高度 $h = 2$ 的滿二元樹，其節點總數 $n = 7$，則底層節點數量 $4 = 2^h = (n + 1)/2$。

第 8 章　堆積

Abstract

堆積就像是山嶽峰巒，層疊起伏、形態各異。

座座山峰高低錯落，而最高的山峰總是最先映入眼簾。

8.1　堆積

堆積（heap）是一種滿足特定條件的完全二元樹，主要可分為兩種型別，如圖 8-1 所示。

- **小頂堆積**（min heap）：任意節點的值 ≤ 其子節點的值。
- **大頂堆積**（max heap）：任意節點的值 ≥ 其子節點的值。

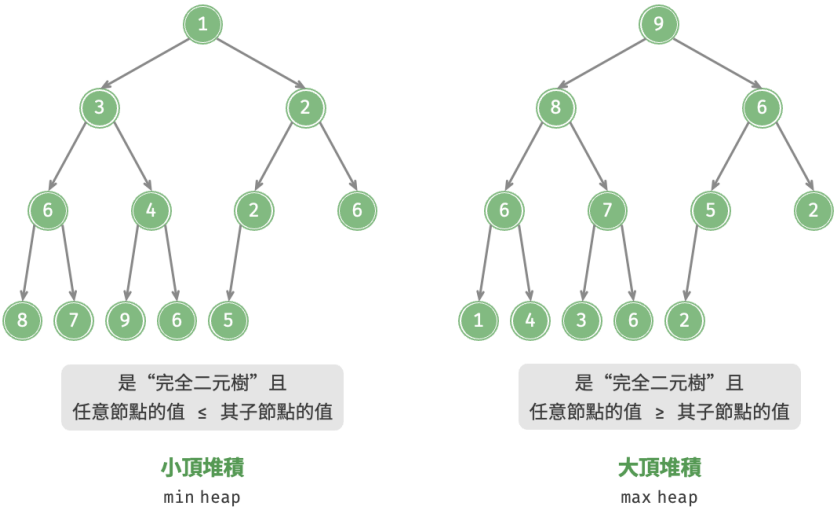


圖 8-1　小頂堆積與大頂堆積

堆積作為完全二元樹的一個特例，具有以下特性。

- 最底層節點靠左填充，其他層的節點都被填滿。
- 我們將二元樹的根節點稱為“堆積頂”，將底層最靠右的節點稱為“堆積底”。
- 對於大頂堆積（小頂堆積），堆積頂元素（根節點）的值是最大（最小）的。

8.1.1　堆積的常用操作

需要指出的是，許多程式語言提供的是**優先佇列**（priority queue），這是一種抽象的資料結構，定義為具有優先順序排序的佇列。

實際上，**堆積通常用於實現優先佇列，大頂堆積相當於元素按從大到小的順序出列的優先佇列**。從使用角度來看，我們可以將“優先佇列”和“堆積”看作等價的資料結構。因此，本書對兩者不做特別區分，統一稱作“堆積”。

堆積的常用操作見表 8-1 ，方法名需要根據程式語言來確定。

表 8-1　堆積的操作效率

方法名	描述	時間複雜度
`push()`	元素入堆積	$O(\log n)$
`pop()`	堆積頂元素出堆積	$O(\log n)$
`peek()`	訪問堆積頂元素（對於大 / 小頂堆積分別為最大 / 小值）	$O(1)$
`size()`	獲取堆積的元素數量	$O(1)$
`isEmpty()`	判斷堆積是否為空	$O(1)$

在實際應用中，我們可以直接使用程式語言提供的堆積類別（或優先佇列類別）。

類似於排序演算法中的“從小到大排列”和“從大到小排列”，我們可以透過設定一個 `flag` 或修改 `Comparator` 實現“小頂堆積”與“大頂堆積”之間的轉換。程式碼如下所示：

```rust
// === File: heap.rs ===

use std::collections::BinaryHeap;
use std::cmp::Reverse;

/* 初始化堆積 */
// 初始化小頂堆積
let mut min_heap = BinaryHeap::<Reverse<i32>>::new();
// 初始化大頂堆積
let mut max_heap = BinaryHeap::new();

/* 元素入堆積 */
max_heap.push(1);
max_heap.push(3);
max_heap.push(2);
max_heap.push(5);
max_heap.push(4);

/* 獲取堆積頂元素 */
let peek = max_heap.peek().unwrap();  // 5

/* 堆積頂元素出堆積 */
// 出堆積元素會形成一個從大到小的序列
let peek = max_heap.pop().unwrap();   // 5
let peek = max_heap.pop().unwrap();   // 4
let peek = max_heap.pop().unwrap();   // 3
let peek = max_heap.pop().unwrap();   // 2
let peek = max_heap.pop().unwrap();   // 1

/* 獲取堆積大小 */
let size = max_heap.len();

/* 判斷堆積是否為空 */
```

```
let is_empty = max_heap.is_empty();

/* 輸入串列並建堆積 */
let min_heap = BinaryHeap::from(vec![Reverse(1), Reverse(3), Reverse(2), Reverse(5), Reverse(4)]);
```

8.1.2　堆積的實現

下文實現的是大頂堆積。若要將其轉換為小頂堆積，只需將所有大小邏輯判斷進行逆轉（例如，將 $\geq$ 替換為 $\leq$ ）。感興趣的讀者可以自行實現。

1.　堆積的儲存與表示

“二元樹”章節講過，完全二元樹非常適合用陣列來表示。由於堆積正是一種完全二元樹，**因此我們將採用陣列來儲存堆積**。

當使用陣列表示二元樹時，元素代表節點值，索引代表節點在二元樹中的位置。**節點指標透過索引對映公式來實現**。

如圖 8-2 所示，給定索引 i ，其左子節點的索引為 $2i + 1$ ，右子節點的索引為 $2i + 2$ ，父節點的索引為 $(i - 1)/2$（向下整除）。當索引越界時，表示空節點或節點不存在。

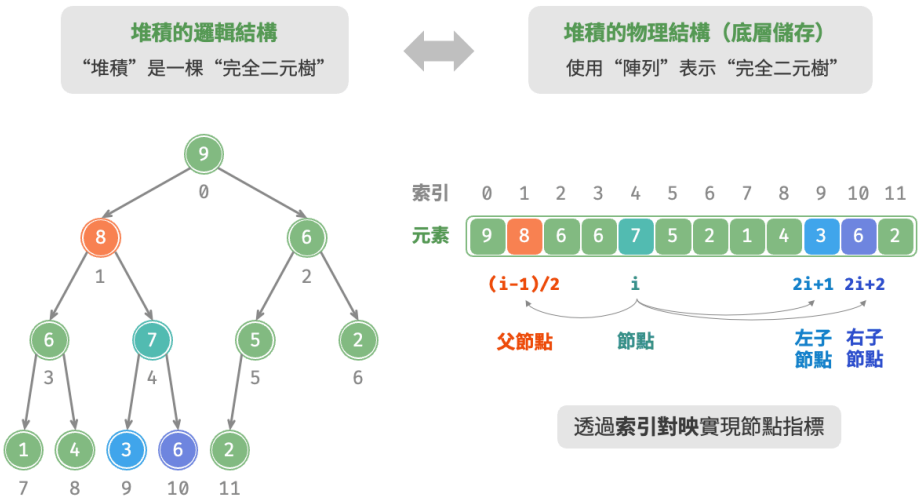


圖 8-2　堆積的表示與儲存

我們可以將索引對映公式封裝成函式，方便後續使用：

```
// === File: my_heap.rs ===

/* 獲取左子節點的索引 */
```

```rust
fn left(i: usize) -> usize {
    2 * i + 1
}

/* 獲取右子節點的索引 */
fn right(i: usize) -> usize {
    2 * i + 2
}

/* 獲取父節點的索引 */
fn parent(i: usize) -> usize {
    (i - 1) / 2 // 向下整除
}
```

2. 訪問堆積頂元素

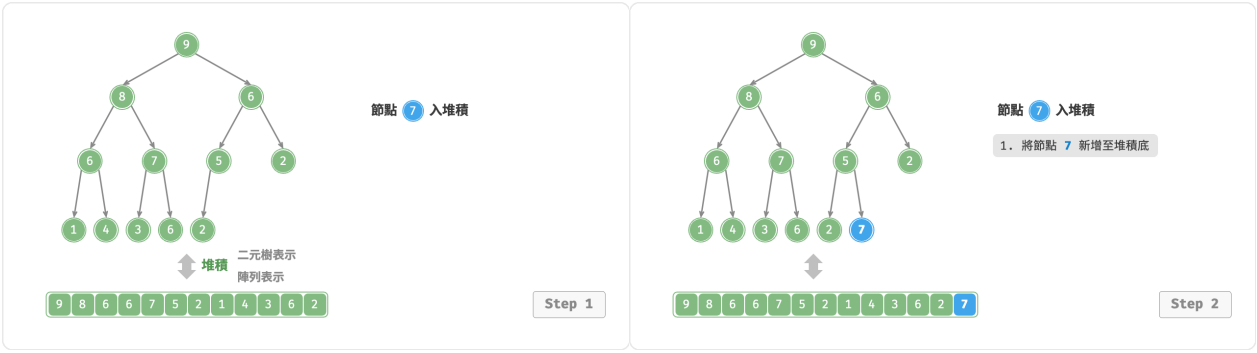
堆積頂元素即為二元樹的根節點，也就是串列的首個元素：

```rust
// === File: my_heap.rs ===

/* 訪問堆積頂元素 */
fn peek(&self) -> Option<i32> {
    self.max_heap.first().copied()
}
```

3. 元素入堆積

給定元素 `val`，我們首先將其新增到堆積底。新增之後，由於 `val` 可能大於堆積中其他元素，堆積的成立條件可能已被破壞，**因此需要修復從插入節點到根節點的路徑上的各個節點**，這個操作被稱為**堆積化**（heapify）。

考慮從入堆積節點開始，**從底至頂執行堆積化**。如圖 8-3 所示，我們比較插入節點與其父節點的值，如果插入節點更大，則將它們交換。然後繼續執行此操作，從底至頂修復堆積中的各個節點，直至越過根節點或遇到無須交換的節點時結束。



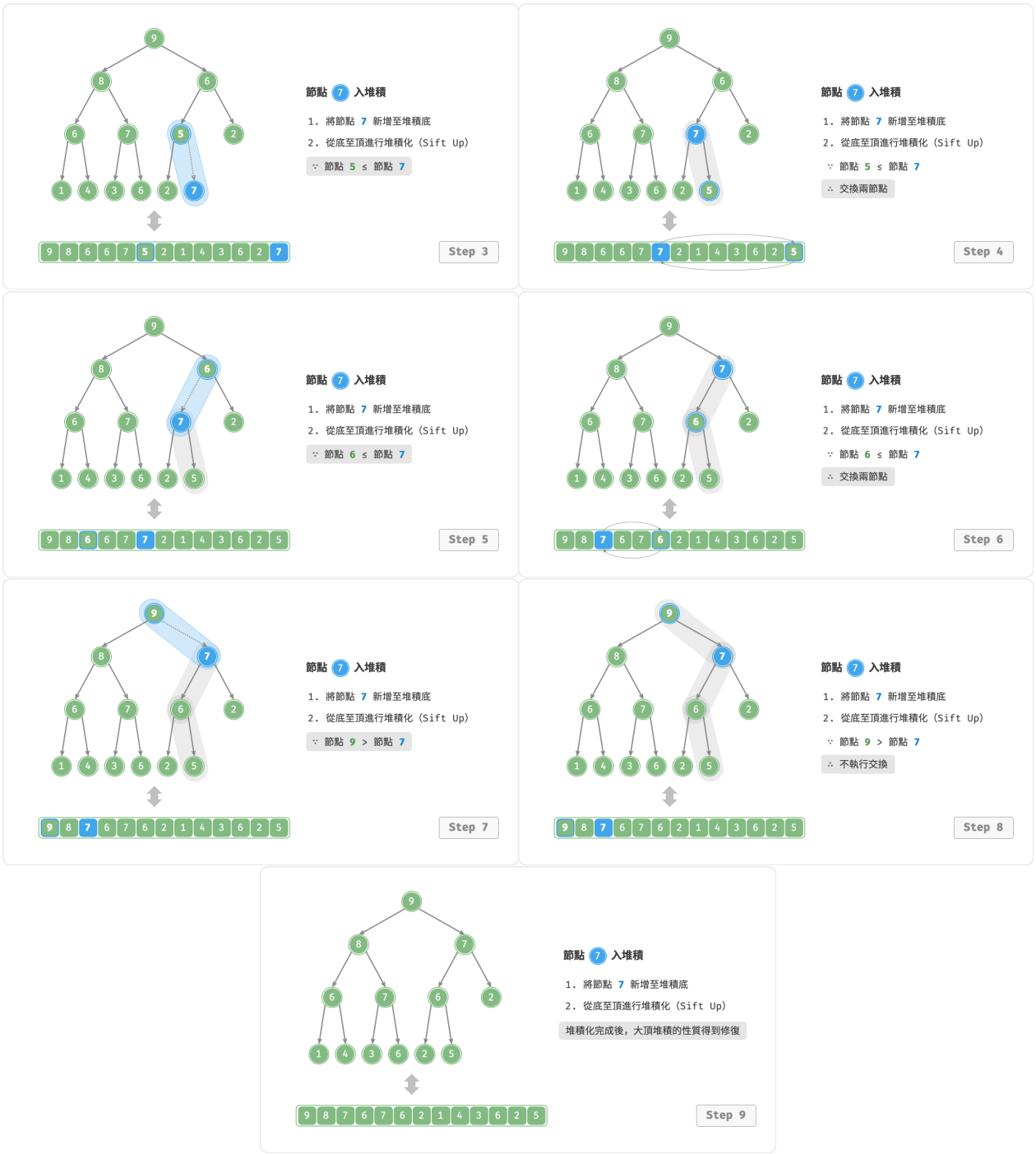


圖 8-3　元素入堆積步驟

設節點總數為 n，則樹的高度為 $O(\log n)$。由此可知，堆積化操作的迴圈輪數最多為 $O(\log n)$，**元素入堆積操作的時間複雜度為 $O(\log n)$**。程式碼如下所示：

```
// === File: my_heap.rs ===

/* 元素入堆積 */
```

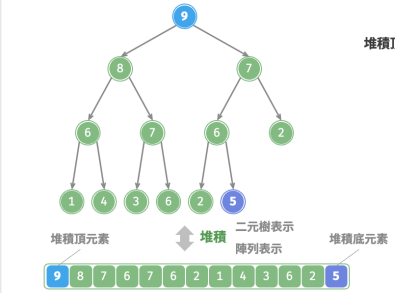
```rust
fn push(&mut self, val: i32) {
    // 新增節點
    self.max_heap.push(val);
    // 從底至頂堆積化
    self.sift_up(self.size() - 1);
}

/* 從節點 i 開始，從底至頂堆積化 */
fn sift_up(&mut self, mut i: usize) {
    loop {
        // 節點 i 已經是堆積頂節點了，結束堆積化
        if i == 0 {
            break;
        }
        // 獲取節點 i 的父節點
        let p = Self::parent(i);
        // 當“節點無須修復”時，結束堆積化
        if self.max_heap[i] <= self.max_heap[p] {
            break;
        }
        // 交換兩節點
        self.swap(i, p);
        // 迴圈向上堆積化
        i = p;
    }
}
```

4. 堆積頂元素出堆積

堆積頂元素是二元樹的根節點，即串列首元素。如果我們直接從串列中刪除首元素，那麼二元樹中所有節點的索引都會發生變化，這將使得後續使用堆積化進行修復變得困難。為了儘量減少元素索引的變動，我們採用以下操作步驟。

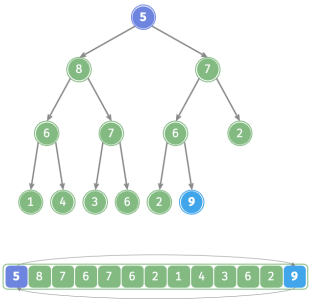
1. 交換堆積頂元素與堆積底元素（交換根節點與最右葉節點）。
2. 交換完成後，將堆積底從串列中刪除（注意，由於已經交換，因此實際上刪除的是原來的堆積頂元素）。
3. 從根節點開始，**從頂至底執行堆積化**。

如圖 8-4 所示，**“從頂至底堆積化”的操作方向與“從底至頂堆積化”相反**，我們將根節點的值與其兩個子節點的值進行比較，將最大的子節點與根節點交換。然後迴圈執行此操作，直到越過葉節點或遇到無須交換的節點時結束。



堆積頂元素出堆積

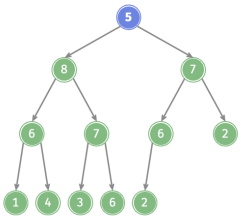
Step 1



堆積頂元素出堆積

1. 將堆積頂元素與堆積底元素交換

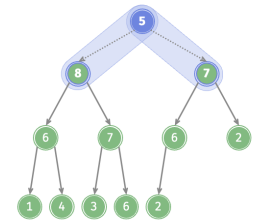
Step 2



堆積頂元素出堆積

1. 將堆積頂元素與堆積底元素交換
2. 彈出當前的堆積底元素

Step 3

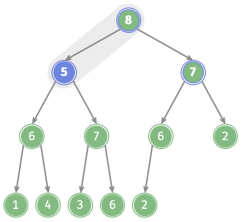


堆積頂元素出堆積

1. 將堆積頂元素與堆積底元素交換
2. 彈出當前的堆積底元素
3. 從頂至底進行堆積化（Sift Down）

∵ 在節點 `5`, `8`, `7` 中，節點 `8` 最大

Step 4



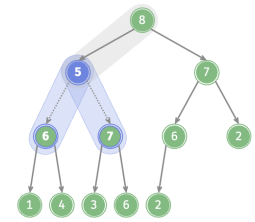
堆積頂元素出堆積

1. 將堆積頂元素與堆積底元素交換
2. 彈出當前的堆積底元素
3. 從頂至底進行堆積化（Sift Down）

∵ 在節點 `5`, `8`, `7` 中，節點 `8` 最大

∴ 交換節點 `5` 與節點 `8`

Step 5

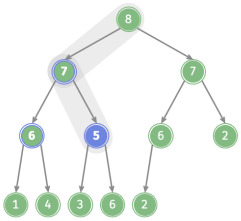


堆積頂元素出堆積

1. 將堆積頂元素與堆積底元素交換
2. 彈出當前的堆積底元素
3. 從頂至底進行堆積化（Sift Down）

∵ 在節點 `5`, `6`, `7` 中，節點 `7` 最大

Step 6



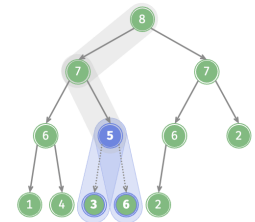
堆積頂元素出堆積

1. 將堆積頂元素與堆積底元素交換
2. 彈出當前的堆積底元素
3. 從頂至底進行堆積化（Sift Down）

∵ 在節點 `5`, `6`, `7` 中，節點 `7` 最大

∴ 交換節點 `5` 與節點 `7`

Step 7

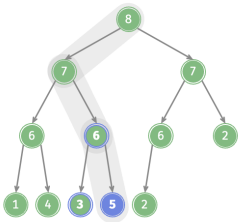


堆積頂元素出堆積

1. 將堆積頂元素與堆積底元素交換
2. 彈出當前的堆積底元素
3. 從頂至底進行堆積化（Sift Down）

∵ 在節點 `5`, `3`, `6` 中，節點 `6` 最大

Step 8

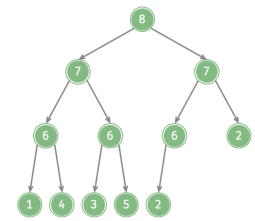


堆積頂元素出堆積

1. 將堆積頂元素與堆積底元素交換
2. 彈出當前的堆積底元素
3. 從頂至底進行堆積化（Sift Down）

∴ 交換節點 `5` 與節點 `6`

Step 9



堆積頂元素出堆積

1. 將堆積頂元素與堆積底元素交換
2. 彈出當前的堆積底元素
3. 從頂至底進行堆積化（Sift Down）

堆積化完成後，大頂堆積的性質得到修復

Step 10

圖 8-4　堆積頂元素出堆積步驟

與元素入堆積操作相似，堆積頂元素出堆積操作的時間複雜度也為 $O(\log n)$ 。程式碼如下所示：

```rust
// === File: my_heap.rs ===

/* 元素出堆積 */
fn pop(&mut self) -> i32 {
    // 判空處理
    if self.is_empty() {
        panic!("index out of bounds");
    }
    // 交換根節點與最右葉節點（交換首元素與尾元素）
    self.swap(0, self.size() - 1);
    // 刪除節點
    let val = self.max_heap.pop().unwrap();
    // 從頂至底堆積化
    self.sift_down(0);
    // 返回堆積頂元素
    val
}

/* 從節點 i 開始，從頂至底堆積化 */
fn sift_down(&mut self, mut i: usize) {
    loop {
        // 判斷節點 i, l, r 中值最大的節點，記為 ma
        let (l, r, mut ma) = (Self::left(i), Self::right(i), i);
        if l < self.size() && self.max_heap[l] > self.max_heap[ma] {
            ma = l;
        }
        if r < self.size() && self.max_heap[r] > self.max_heap[ma] {
            ma = r;
        }
        // 若節點 i 最大或索引 l, r 越界，則無須繼續堆積化，跳出
        if ma == i {
            break;
        }
        // 交換兩節點
        self.swap(i, ma);
        // 迴圈向下堆積化
        i = ma;
    }
}
```

8.1.3　堆積的常見應用

- **優先佇列**：堆積通常作為實現優先佇列的首選資料結構，其入列和出列操作的時間複雜度均為 $O(\log n)$ ，而建堆積操作為 $O(n)$ ，這些操作都非常高效。
- **堆積排序**：給定一組資料，我們可以用它們建立一個堆積，然後不斷地執行元素出堆積操作，從而得到有序資料。然而，我們通常會使用一種更優雅的方式實現堆積排序，詳見“堆積排序”章節。
- **獲取最大的 k 個元素**：這是一個經典的演算法問題，同時也是一種典型應用，例如選擇熱度前 10 的新聞作為微博熱搜，選取銷量前 10 的商品等。

8.2　建堆積操作

在某些情況下，我們希望使用一個串列的所有元素來構建一個堆積，這個過程被稱為“建堆積操作”。

8.2.1　藉助入堆積操作實現

我們首先建立一個空堆積，然後走訪串列，依次對每個元素執行“入堆積操作”，即先將元素新增至堆積的尾部，再對該元素執行“從底至頂”堆積化。

每當一個元素入堆積，堆積的長度就加一。由於節點是從頂到底依次被新增進二元樹的，因此堆積是“自上而下”構建的。

設元素數量為 n , 每個元素的入堆積操作使用 $O(\log n)$ 時間, 因此該建堆積方法的時間複雜度為 $O(n \log n)$ 。

8.2.2　透過走訪堆積化實現

實際上，我們可以實現一種更為高效的建堆積方法，共分為兩步。

1. 將串列所有元素原封不動地新增到堆積中，此時堆積的性質尚未得到滿足。
2. 倒序走訪堆積（層序走訪的倒序），依次對每個非葉節點執行“從頂至底堆積化”。

每當堆積化一個節點後，以該節點為根節點的子樹就形成一個合法的子堆積。而由於是倒序走訪，因此堆積是“自下而上”構建的。

之所以選擇倒序走訪，是因為這樣能夠保證當前節點之下的子樹已經是合法的子堆積，這樣堆積化當前節點才是有效的。

值得說明的是，**由於葉節點沒有子節點，因此它們天然就是合法的子堆積，無須堆積化**。如以下程式碼所示，最後一個非葉節點是最後一個節點的父節點，我們從它開始倒序走訪並執行堆積化：

```
// === File: my_heap.rs ===

/* 建構子，根據輸入串列建堆積 */
fn new(nums: Vec<i32>) -> Self {
    // 將串列元素原封不動新增進堆積
```

```
    let mut heap = MaxHeap { max_heap: nums };
    // 堆積化除葉節點以外的其他所有節點
    for i in (0..=Self::parent(heap.size() - 1)).rev() {
        heap.sift_down(i);
    }
    heap
}
```

8.2.3　複雜度分析

下面，我們來嘗試推算第二種建堆積方法的時間複雜度。

- 假設完全二元樹的節點數量為 n ，則葉節點數量為 $(n + 1)/2$ ，其中 $/$ 為向下整除。因此需要堆積化的節點數量為 $(n - 1)/2$ 。
- 在從頂至底堆積化的過程中，每個節點最多堆積化到葉節點，因此最大迭代次數為二元樹高度 $\log n$ 。

將上述兩者相乘，可得到建堆積過程的時間複雜度為 $O(n \log n)$ 。**但這個估算結果並不準確，因為我們沒有考慮到二元樹底層節點數量遠多於頂層節點的性質**。

接下來我們來進行更為準確的計算。為了降低計算難度，假設給定一個節點數量為 n 、高度為 h 的“完美二元樹”，該假設不會影響計算結果的正確性。

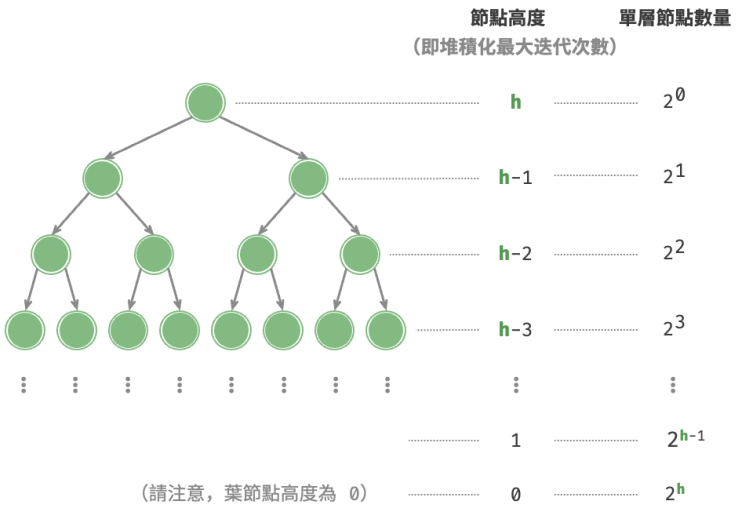


圖 8-5　完美二元樹的各層節點數量

如圖 8-5 所示，節點“從頂至底堆積化”的最大迭代次數等於該節點到葉節點的距離，而該距離正是“節點高度”。因此，我們可以對各層的“節點數量 × 節點高度”求和，**得到所有節點的堆積化迭代次數的總和**。

$$T(h) = 2^0h + 2^1(h-1) + 2^2(h-2) + \cdots + 2^{(h-1)} \times 1$$

化簡上式需要藉助中學的數列知識，先將 $T(h)$ 乘以 2 ，得到：

$$
\begin{aligned}
T(h) &= 2^0 h + 2^1 (h-1) + 2^2 (h-2) + \cdots + 2^{h-1} \times 1 \\
2T(h) &= 2^1 h + 2^2 (h-1) + 2^3 (h-2) + \cdots + 2^h \times 1
\end{aligned}
$$

使用錯位相減法，用下式 $2T(h)$ 減去上式 $T(h)$ ，可得：

$$
2T(h) - T(h) = T(h) = -2^0 h + 2^1 + 2^2 + \cdots + 2^{h-1} + 2^h
$$

觀察上式，發現 $T(h)$ 是一個等比數列，可直接使用求和公式，得到時間複雜度為：

$$
\begin{aligned}
T(h) &= 2\frac{1-2^h}{1-2} - h \\
&= 2^{h+1} - h - 2 \\
&= O(2^h)
\end{aligned}
$$

進一步，高度為 h 的完美二元樹的節點數量為 $n = 2^{h+1} - 1$ ，易得複雜度為 $O(2^h) = O(n)$ 。以上推算表明，**輸入串列並建堆積的時間複雜度為 $O(n)$ ，非常高效**。

8.3　Top-k 問題

Question
給定一個長度為 n 的無序陣列 `nums` ，請返回陣列中最大的 k 個元素。

對於該問題，我們先介紹兩種思路比較直接的解法，再介紹效率更高的堆積解法。

8.3.1　方法一：走訪選擇

我們可以進行圖 8-6 所示的 k 輪走訪，分別在每輪中提取第 1、2、...、k 大的元素，時間複雜度為 $O(nk)$ 。

此方法只適用於 $k \ll n$ 的情況，因為當 k 與 n 比較接近時，其時間複雜度趨向於 $O(n^2)$ ，非常耗時。

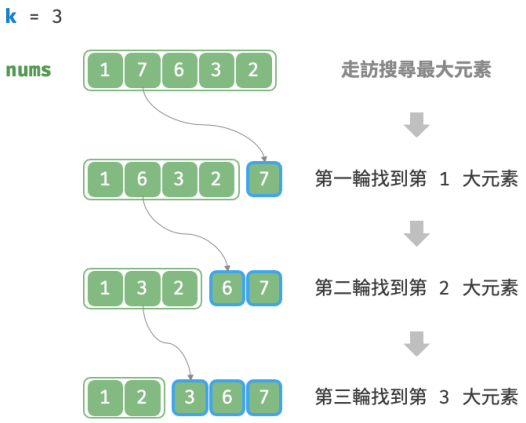


圖 8-6　走訪尋找最大的 k 個元素

Tip
當 $k = n$ 時，我們可以得到完整的有序序列，此時等價於“選擇排序”演算法。

8.3.2　方法二：排序

如圖 8-7 所示，我們可以先對陣列 `nums` 進行排序，再返回最右邊的 k 個元素，時間複雜度為 $O(n \log n)$ 。顯然，該方法“超額”完成任務了，因為我們只需找出最大的 k 個元素即可，而不需要排序其他元素。

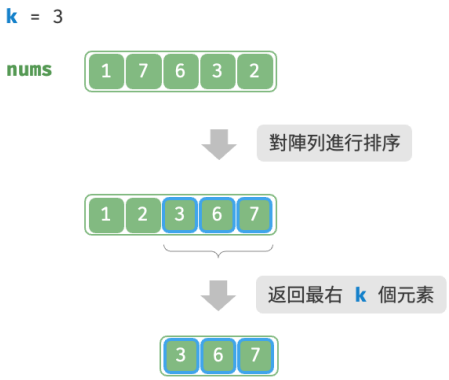


圖 8-7　排序尋找最大的 k 個元素

8.3.3　方法三：堆積

我們可以基於堆積更加高效地解決 Top-k 問題，流程如圖 8-8 所示。

1. 初始化一個小頂堆積，其堆積頂元素最小。
2. 先將陣列的前 k 個元素依次入堆積。
3. 從第 $k + 1$ 個元素開始，若當前元素大於堆積頂元素，則將堆積頂元素出堆積，並將當前元素入堆積。
4. 走訪完成後，堆積中儲存的就是最大的 k 個元素。

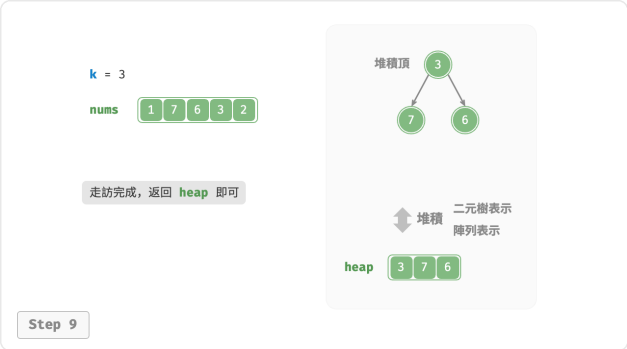


圖 8-8　基於堆積尋找最大的 k 個元素

示例程式碼如下：

```
// === File: top_k.rs ===

/* 基於堆積查詢陣列中最大的 k 個元素 */
fn top_k_heap(nums: Vec<i32>, k: usize) -> BinaryHeap<Reverse<i32>> {
    // BinaryHeap 是大頂堆積，使用 Reverse 將元素取反，從而實現小頂堆積
    let mut heap = BinaryHeap::<Reverse<i32>>::new();
    // 將陣列的前 k 個元素入堆積
    for &num in nums.iter().take(k) {
        heap.push(Reverse(num));
    }
    // 從第 k+1 個元素開始，保持堆積的長度為 k
    for &num in nums.iter().skip(k) {
        // 若當前元素大於堆積頂元素，則將堆積頂元素出堆積、當前元素入堆積
        if num > heap.peek().unwrap().0 {
            heap.pop();
            heap.push(Reverse(num));
        }
    }
    heap
}
```

總共執行了 n 輪入堆積和出堆積，堆積的最大長度為 k，因此時間複雜度為 $O(n \log k)$。該方法的效率很高，當 k 較小時，時間複雜度趨向 $O(n)$；當 k 較大時，時間複雜度不會超過 $O(n \log n)$。

另外，該方法適用於動態資料流的使用場景。在不斷加入資料時，我們可以持續維護堆積內的元素，從而實現最大的 k 個元素的動態更新。

8.4 小結

1. 重點回顧

- 堆積是一棵完全二元樹，根據成立條件可分為大頂堆積和小頂堆積。大（小）頂堆積的堆積頂元素是最大（小）的。
- 優先佇列的定義是具有出列優先順序的佇列，通常使用堆積來實現。
- 堆積的常用操作及其對應的時間複雜度包括：元素入堆積 $O(\log n)$、堆積頂元素出堆積 $O(\log n)$ 和訪問堆積頂元素 $O(1)$ 等。
- 完全二元樹非常適合用陣列表示，因此我們通常使用陣列來儲存堆積。
- 堆積化操作用於維護堆積的性質，在入堆積和出堆積操作中都會用到。
- 輸入 n 個元素並建堆積的時間複雜度可以最佳化至 $O(n)$ ，非常高效。
- Top-k 是一個經典演算法問題，可以使用堆積資料結構高效解決，時間複雜度為 $O(n \log k)$ 。

2. Q & A

Q：資料結構的“堆積”與記憶體管理的“堆積”是同一個概念嗎？

兩者不是同一個概念，只是碰巧都叫“堆積”。計算機系統記憶體中的堆積是動態記憶體分配的一部分，程式在執行時可以使用它來儲存資料。程式可以請求一定量的堆積記憶體，用於儲存如物件和陣列等複雜結構。當這些資料不再需要時，程式需要釋放這些記憶體，以防止記憶體流失。相較於堆疊記憶體，堆積記憶體的管理和使用需要更謹慎，使用不當可能會導致記憶體流失和野指標等問題。

第 9 章　圖

Abstract

在生命旅途中，我們就像是一個個節點，被無數看不見的邊相連。

每一次的相識與相離，都在這張巨大的網路圖中留下獨特的印記。

9.1 圖

圖（graph）是一種非線性資料結構，由**頂點**（vertex）和**邊**（edge）組成。我們可以將圖 G 抽象地表示為一組頂點 V 和一組邊 E 的集合。以下示例展示了一個包含 5 個頂點和 7 條邊的圖。

$$
\begin{aligned}
V &= \{1, 2, 3, 4, 5\} \\
E &= \{(1, 2), (1, 3), (1, 5), (2, 3), (2, 4), (2, 5), (4, 5)\} \\
G &= \{V, E\}
\end{aligned}
$$

如果將頂點看作節點，將邊看作連線各個節點的引用（指標），我們就可以將圖看作一種從鏈結串列拓展而來的資料結構。如圖 9-1 所示，**相較於線性關係（鏈結串列）和分治關係（樹），網路關係（圖）的自由度更高**，因而更為複雜。

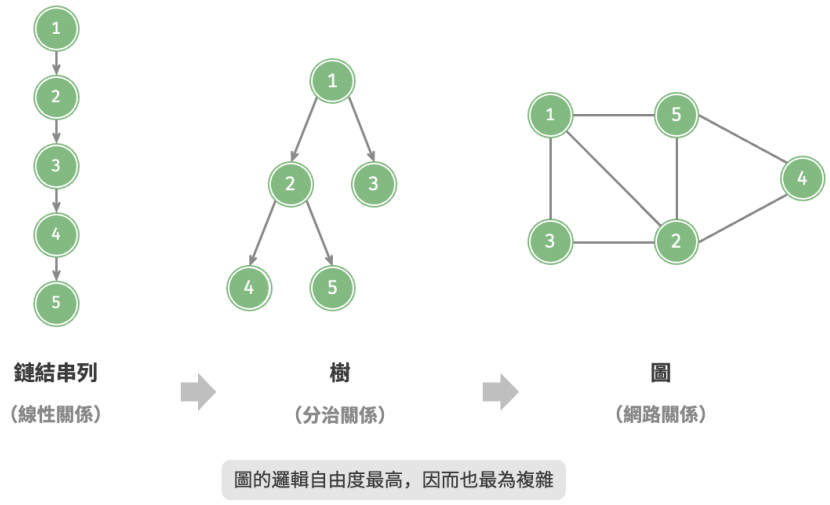


圖 9-1　鏈結串列、樹、圖之間的關係

9.1.1 圖的常見型別與術語

根據邊是否具有方向，可分為**無向圖**（undirected graph）和**有向圖**（directed graph），如圖 9-2 所示。

- 在無向圖中，邊表示兩頂點之間的“雙向”連線關係，例如微信或 QQ 中的“好友關係”。
- 在有向圖中，邊具有方向性，即 $A \to B$ 和 $A \leftarrow B$ 兩個方向的邊是相互獨立的，例如微博或抖音上的“關注”與“被關注”關係。

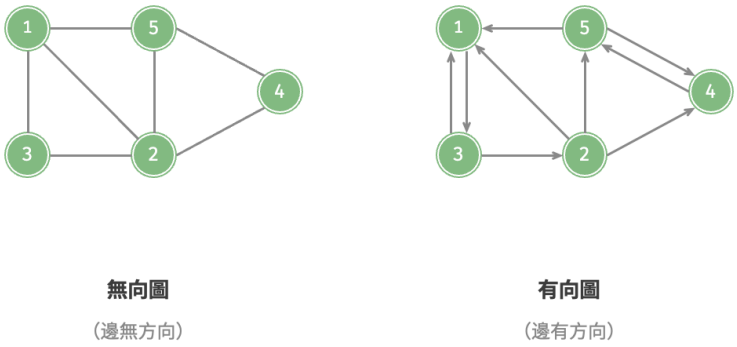


圖 9-2　有向圖與無向圖

根據所有頂點是否連通，可分為**連通圖**（connected graph）和**非連通圖**（disconnected graph），如圖 9-3 所示。

- 對於連通圖，從某個頂點出發，可以到達其餘任意頂點。
- 對於非連通圖，從某個頂點出發，至少有一個頂點無法到達。

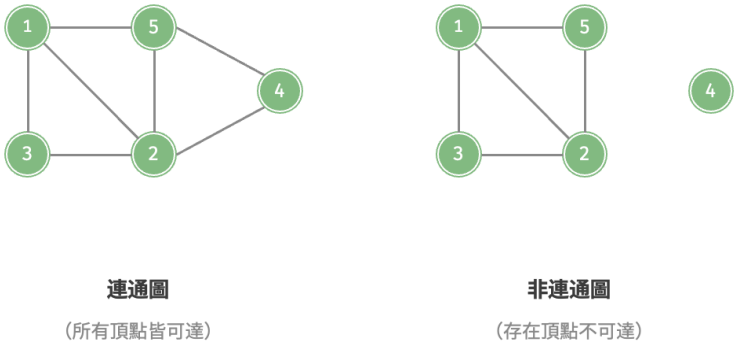


圖 9-3　連通圖與非連通圖

我們還可以為邊新增“權重”變數，從而得到如圖 9-4 所示的**有權圖**（weighted graph）。例如在《王者榮耀》等手遊中，系統會根據共同遊戲時間來計算玩家之間的“親密度”，這種親密度網路就可以用有權圖來表示。

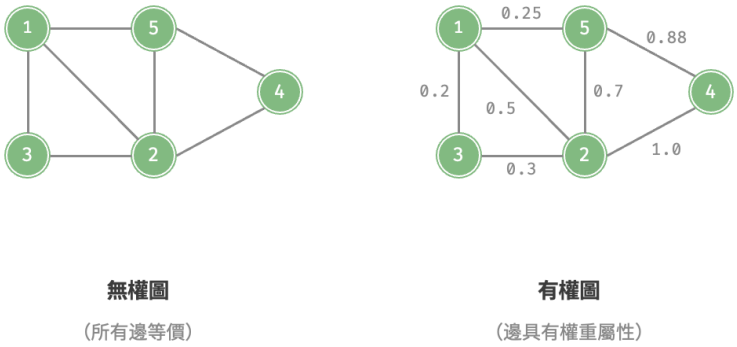


圖 9-4　有權圖與無權圖

圖資料結構包含以下常用術語。

- **鄰接**（adjacency）：當兩頂點之間存在邊相連時，稱這兩頂點“鄰接”。在圖 9-4 中，頂點 1 的鄰接頂點為頂點 2、3、5。
- **路徑**（path）：從頂點 A 到頂點 B 經過的邊構成的序列被稱為從 A 到 B 的“路徑”。在圖 9-4 中，邊序列 1-5-2-4 是頂點 1 到頂點 4 的一條路徑。
- **度**（degree）：一個頂點擁有的邊數。對於有向圖，**入度**（in-degree）表示有多少條邊指向該頂點，**出度**（out-degree）表示有多少條邊從該頂點指出。

9.1.2　圖的表示

圖的常用表示方式包括“鄰接矩陣”和“鄰接表”。以下使用無向圖進行舉例。

1.　鄰接矩陣

設圖的頂點數量為 n，**鄰接矩陣**（adjacency matrix）使用一個 $n \times n$ 大小的矩陣來表示圖，每一行（列）代表一個頂點，矩陣元素代表邊，用 1 或 0 表示兩個頂點之間是否存在邊。

如圖 9-5 所示，設鄰接矩陣為 M、頂點串列為 V，那麼矩陣元素 $M[i, j] = 1$ 表示頂點 $V[i]$ 到頂點 $V[j]$ 之間存在邊，反之 $M[i, j] = 0$ 表示兩頂點之間無邊。

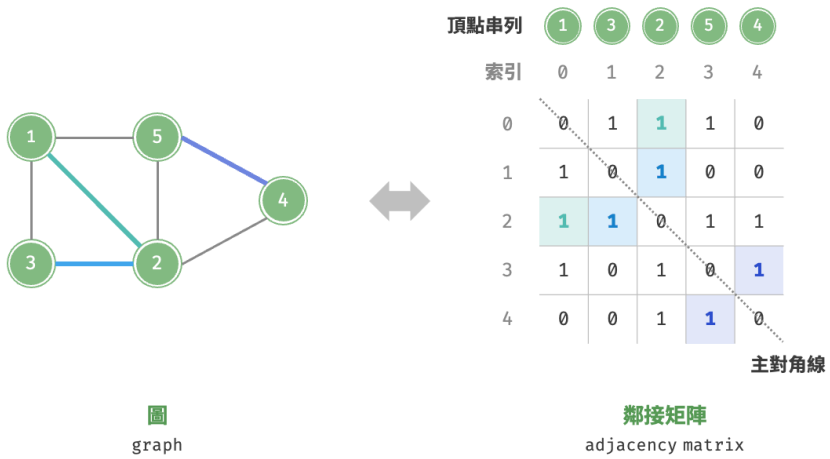


圖 9-5　圖的鄰接矩陣表示

鄰接矩陣具有以下特性。

- 在簡單圖中，頂點不能與自身相連，此時鄰接矩陣主對角線元素沒有意義。
- 對於無向圖，兩個方向的邊等價，此時鄰接矩陣關於主對角線對稱。
- 將鄰接矩陣的元素從 1 和 0 替換為權重，則可表示有權圖。

使用鄰接矩陣表示圖時，我們可以直接訪問矩陣元素以獲取邊，因此增刪查改操作的效率很高，時間複雜度均為 $O(1)$ 。然而，矩陣的空間複雜度為 $O(n^2)$ ，記憶體佔用較多。

2. 鄰接表

鄰接表（adjacency list）使用 n 個鏈結串列來表示圖，鏈結串列節點表示頂點。第 i 個鏈結串列對應頂點 i ，其中儲存了該頂點的所有鄰接頂點（與該頂點相連的頂點）。圖 9-6 展示了一個使用鄰接表儲存的圖的示例。

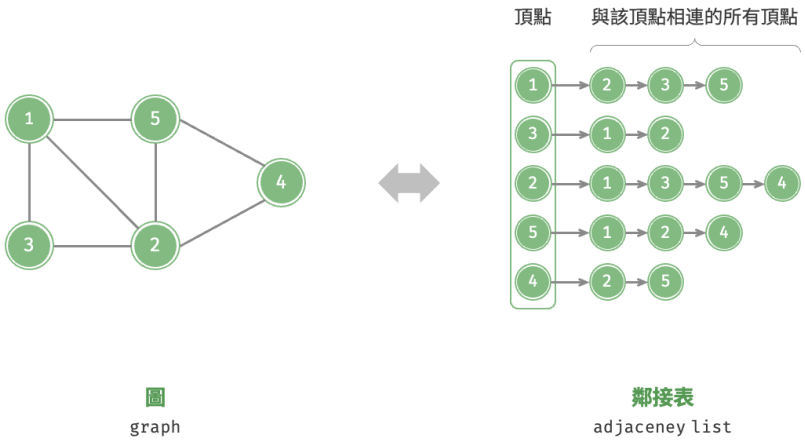


圖 9-6　圖的鄰接表表示

鄰接表僅儲存實際存在的邊，而邊的總數通常遠小於 n^2 ，因此它更加節省空間。然而，在鄰接表中需要透過走訪鏈結串列來查詢邊，因此其時間效率不如鄰接矩陣。

觀察圖 9-6 ，**鄰接表結構與雜湊表中的“鏈式位址”非常相似，因此我們也可以採用類似的方法來最佳化效率**。比如當鏈結串列較長時，可以將鏈結串列轉化為 AVL 樹或紅黑樹，從而將時間效率從 $O(n)$ 最佳化至 $O(\log n)$ ；還可以把鏈結串列轉換為雜湊表，從而將時間複雜度降至 $O(1)$ 。

9.1.3　圖的常見應用

如表 9-1 所示，許多現實系統可以用圖來建模，相應的問題也可以約化為圖計算問題。

表 9-1　現實生活中常見的圖

	頂點	邊	圖計算問題
社交網路	使用者	好友關係	潛在好友推薦
地鐵線路	站點	站點間的連通性	最短路線推薦

	頂點	邊	圖計算問題
太陽系	星體	星體間的萬有引力作用	行星軌道計算

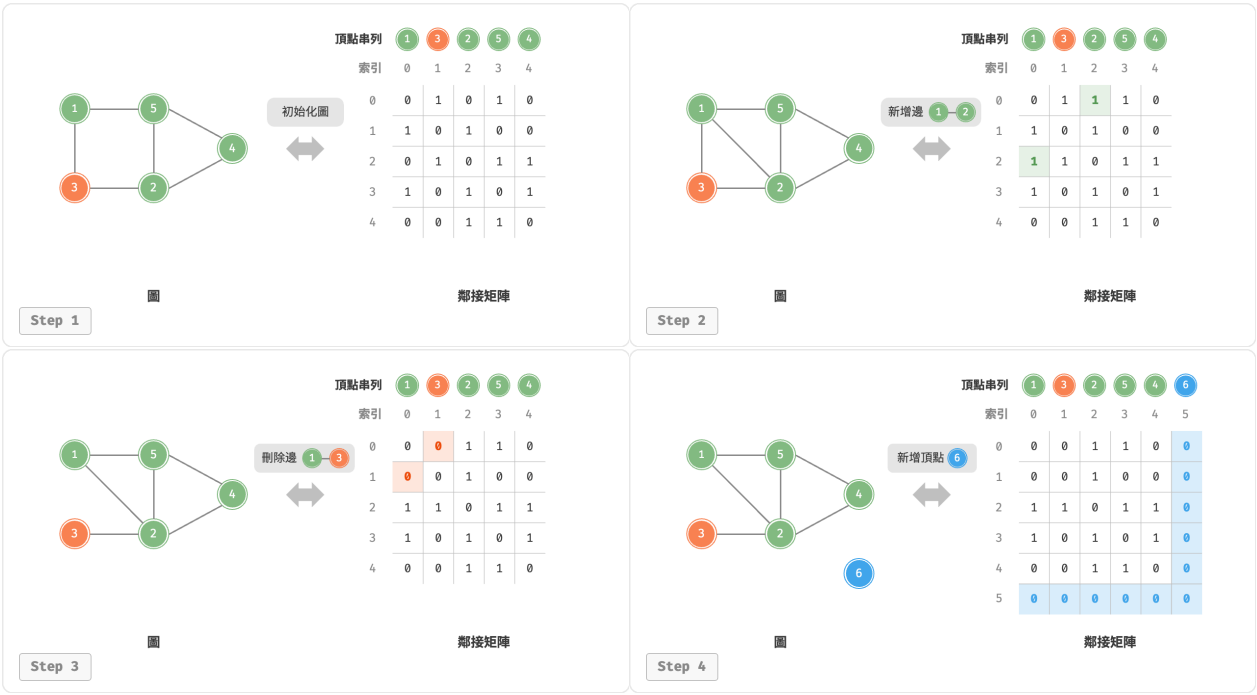
9.2　圖的基礎操作

圖的基礎操作可分為對“邊”的操作和對“頂點”的操作。在“鄰接矩陣”和“鄰接表”兩種表示方法下，實現方式有所不同。

9.2.1　基於鄰接矩陣的實現

給定一個頂點數量為 n 的無向圖，則各種操作的實現方式如圖 9-7 所示。

- **新增或刪除邊**：直接在鄰接矩陣中修改指定的邊即可，使用 $O(1)$ 時間。而由於是無向圖，因此需要同時更新兩個方向的邊。
- **新增頂點**：在鄰接矩陣的尾部新增一行一列，並全部填 0 即可，使用 $O(n)$ 時間。
- **刪除頂點**：在鄰接矩陣中刪除一行一列。當刪除首行首列時達到最差情況，需要將 $(n-1)^2$ 個元素“向左上移動”，從而使用 $O(n^2)$ 時間。
- **初始化**：傳入 n 個頂點，初始化長度為 n 的頂點串列 `vertices`，使用 $O(n)$ 時間；初始化 $n \times n$ 大小的鄰接矩陣 `adjMat`，使用 $O(n^2)$ 時間。



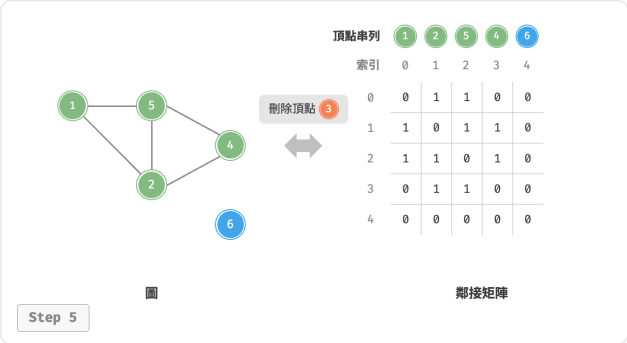


圖 9-7　鄰接矩陣的初始化、增刪邊、增刪頂點

以下是基於鄰接矩陣表示圖的實現程式碼：

```
// === File: graph_adjacency_matrix.rs ===

/* 基於鄰接矩陣實現的無向圖型別 */
pub struct GraphAdjMat {
    // 頂點串列，元素代表“頂點值”，索引代表“頂點索引”
    pub vertices: Vec<i32>,
    // 鄰接矩陣，行列索引對應“頂點索引”
    pub adj_mat: Vec<Vec<i32>>,
}

impl GraphAdjMat {
    /* 建構子 */
    pub fn new(vertices: Vec<i32>, edges: Vec<[usize; 2]>) -> Self {
        let mut graph = GraphAdjMat {
            vertices: vec![],
            adj_mat: vec![],
        };
        // 新增頂點
        for val in vertices {
            graph.add_vertex(val);
        }
        // 新增邊
        // 請注意，edges 元素代表頂點索引，即對應 vertices 元素索引
        for edge in edges {
            graph.add_edge(edge[0], edge[1])
        }

        graph
    }

    /* 獲取頂點數量 */
    pub fn size(&self) -> usize {
        self.vertices.len()
```

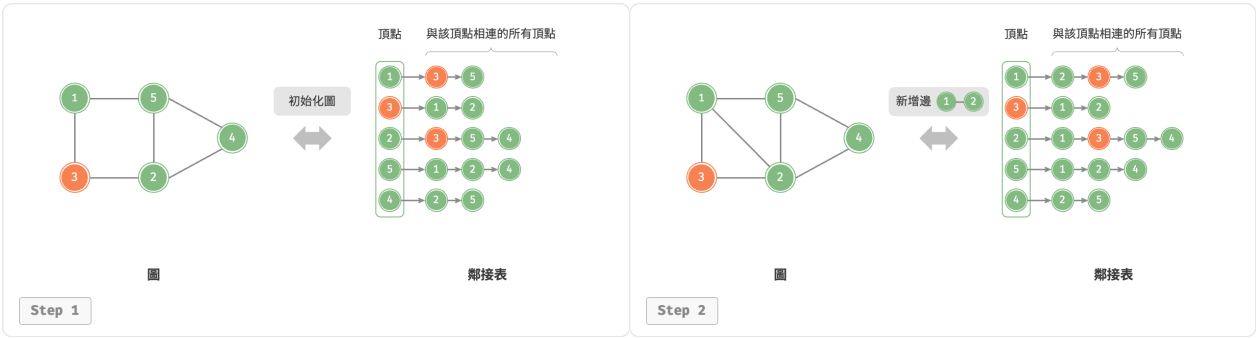
```
}

/* 新增頂點 */
pub fn add_vertex(&mut self, val: i32) {
    let n = self.size();
    // 向頂點串列中新增新頂點的值
    self.vertices.push(val);
    // 在鄰接矩陣中新增一行
    self.adj_mat.push(vec![0; n]);
    // 在鄰接矩陣中新增一列
    for row in &mut self.adj_mat {
        row.push(0);
    }
}

/* 刪除頂點 */
pub fn remove_vertex(&mut self, index: usize) {
    if index >= self.size() {
        panic!("index error")
    }
    // 在頂點串列中移除索引 index 的頂點
    self.vertices.remove(index);
    // 在鄰接矩陣中刪除索引 index 的行
    self.adj_mat.remove(index);
    // 在鄰接矩陣中刪除索引 index 的列
    for row in &mut self.adj_mat {
        row.remove(index);
    }
}

/* 新增邊 */
pub fn add_edge(&mut self, i: usize, j: usize) {
    // 參數 i, j 對應 vertices 元素索引
    // 索引越界與相等處理
    if i >= self.size() || j >= self.size() || i == j {
        panic!("index error")
    }
    // 在無向圖中，鄰接矩陣關於主對角線對稱，即滿足 (i, j) == (j, i)
    self.adj_mat[i][j] = 1;
    self.adj_mat[j][i] = 1;
}

/* 刪除邊 */
// 參數 i, j 對應 vertices 元素索引
pub fn remove_edge(&mut self, i: usize, j: usize) {
    // 參數 i, j 對應 vertices 元素索引
    // 索引越界與相等處理
```

```rust
        if i >= self.size() || j >= self.size() || i == j {
            panic!("index error")
        }
        self.adj_mat[i][j] = 0;
        self.adj_mat[j][i] = 0;
    }

    /* 列印鄰接矩陣 */
    pub fn print(&self) {
        println!(" 頂點串列 = {:?}", self.vertices);
        println!(" 鄰接矩陣 =");
        println!("[");
        for row in &self.adj_mat {
            println!("  {:?},", row);
        }
        println!("]")
    }
}
```

9.2.2　基於鄰接表的實現

設無向圖的頂點總數為 n、邊總數為 m ，則可根據圖 9-8 所示的方法實現各種操作。

- **新增邊**：在頂點對應鏈結串列的末尾新增邊即可，使用 $O(1)$ 時間。因為是無向圖，所以需要同時新增兩個方向的邊。
- **刪除邊**：在頂點對應鏈結串列中查詢並刪除指定邊，使用 $O(m)$ 時間。在無向圖中，需要同時刪除兩個方向的邊。
- **新增頂點**：在鄰接表中新增一個鏈結串列，並將新增頂點作為鏈結串列頭節點，使用 $O(1)$ 時間。
- **刪除頂點**：需走訪整個鄰接表，刪除包含指定頂點的所有邊，使用 $O(n + m)$ 時間。
- **初始化**：在鄰接表中建立 n 個頂點和 $2m$ 條邊，使用 $O(n + m)$ 時間。



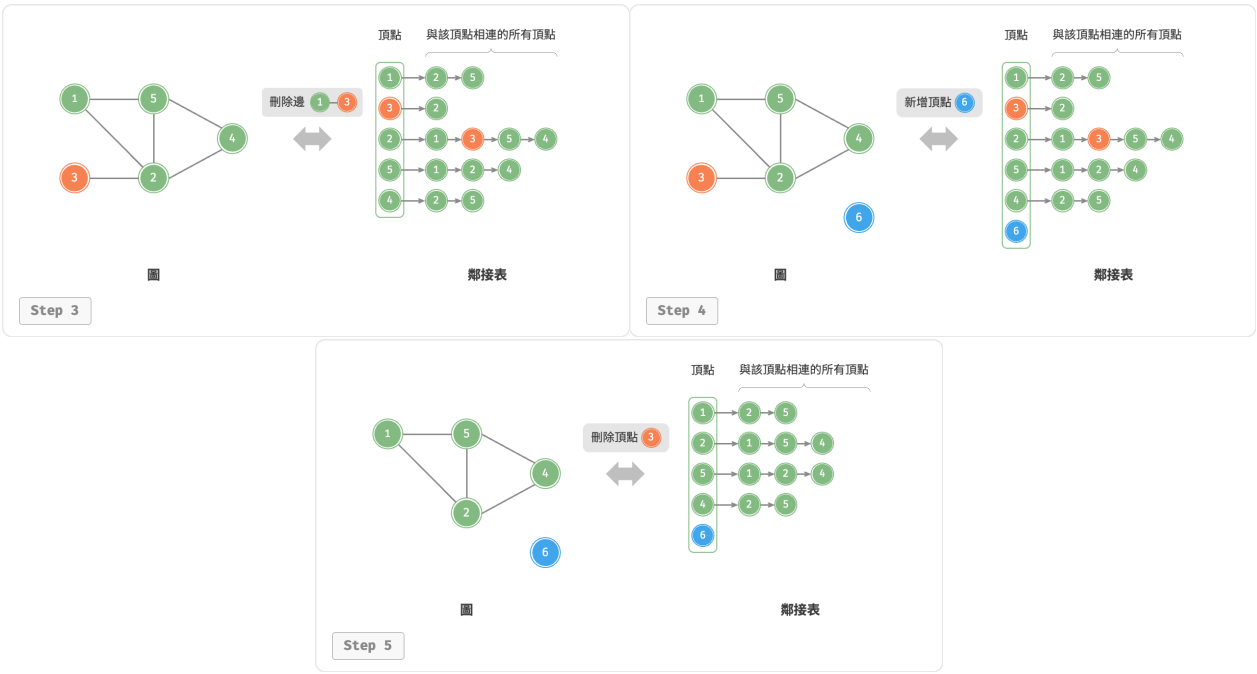


圖 9-8　鄰接表的初始化、增刪邊、增刪頂點

以下是鄰接表的程式碼實現。對比圖 9-8 ，實際程式碼有以下不同。

- 為了方便新增與刪除頂點，以及簡化程式碼，我們使用串列（動態陣列）來代替鏈結串列。
- 使用雜湊表來儲存鄰接表，`key` 為頂點例項，`value` 為該頂點的鄰接頂點串列（鏈結串列）。

另外，我們在鄰接表中使用 `Vertex` 類別來表示頂點，這樣做的原因是：如果與鄰接矩陣一樣，用串列索引來區分不同頂點，那麼假設要刪除索引為 i 的頂點，則需走訪整個鄰接表，將所有大於 i 的索引全部減 1 ，效率很低。而如果每個頂點都是唯一的 `Vertex` 例項，刪除某一頂點之後就無須改動其他頂點了。

```
// === File: graph_adjacency_list.rs ===

/* 基於鄰接表實現的無向圖型別 */
pub struct GraphAdjList {
    // 鄰接表，key：頂點，value：該頂點的所有鄰接頂點
    pub adj_list: HashMap<Vertex, Vec<Vertex>>,
}

impl GraphAdjList {
    /* 建構子 */
    pub fn new(edges: Vec<[Vertex; 2]>) -> Self {
        let mut graph = GraphAdjList {
            adj_list: HashMap::new(),
        };
        // 新增所有頂點和邊
        for edge in edges {
            graph.add_vertex(edge[0]);
```

```
            graph.add_vertex(edge[1]);
            graph.add_edge(edge[0], edge[1]);
        }

        graph
    }

    /* 獲取頂點數量 */
    #[allow(unused)]
    pub fn size(&self) -> usize {
        self.adj_list.len()
    }

    /* 新增邊 */
    pub fn add_edge(&mut self, vet1: Vertex, vet2: Vertex) {
        if !self.adj_list.contains_key(&vet1) || !self.adj_list.contains_key(&vet2) || vet1 == vet2
        {
            panic!("value error");
        }
        // 新增邊 vet1 - vet2
        self.adj_list.get_mut(&vet1).unwrap().push(vet2);
        self.adj_list.get_mut(&vet2).unwrap().push(vet1);
    }

    /* 刪除邊 */
    #[allow(unused)]
    pub fn remove_edge(&mut self, vet1: Vertex, vet2: Vertex) {
        if !self.adj_list.contains_key(&vet1) || !self.adj_list.contains_key(&vet2) || vet1 == vet2
        {
            panic!("value error");
        }
        // 刪除邊 vet1 - vet2
        self.adj_list
            .get_mut(&vet1)
            .unwrap()
            .retain(|&vet| vet != vet2);
        self.adj_list
            .get_mut(&vet2)
            .unwrap()
            .retain(|&vet| vet != vet1);
    }

    /* 新增頂點 */
    pub fn add_vertex(&mut self, vet: Vertex) {
        if self.adj_list.contains_key(&vet) {
            return;
        }
```

```
        // 在鄰接表中新增一個新鏈結串列
        self.adj_list.insert(vet, vec![]);
    }

    /* 刪除頂點 */
    #[allow(unused)]
    pub fn remove_vertex(&mut self, vet: Vertex) {
        if !self.adj_list.contains_key(&vet) {
            panic!("value error");
        }
        // 在鄰接表中刪除頂點 vet 對應的鏈結串列
        self.adj_list.remove(&vet);
        // 走訪其他頂點的鏈結串列，刪除所有包含 vet 的邊
        for list in self.adj_list.values_mut() {
            list.retain(|&v| v != vet);
        }
    }

    /* 列印鄰接表 */
    pub fn print(&self) {
        println!(" 鄰接表 =");
        for (vertex, list) in &self.adj_list {
            let list = list.iter().map(|vertex| vertex.val).collect::<Vec<i32>>();
            println!("{}: {:?},", vertex.val, list);
        }
    }
}
```

9.2.3　效率對比

設圖中共有 n 個頂點和 m 條邊，表 9-2 對比了鄰接矩陣和鄰接表的時間效率和空間效率。

表 9-2　鄰接矩陣與鄰接表對比

	鄰接矩陣	鄰接表（鏈結串列）	鄰接表（雜湊表）
判斷是否鄰接	$O(1)$	$O(m)$	$O(1)$
新增邊	$O(1)$	$O(1)$	$O(1)$
刪除邊	$O(1)$	$O(m)$	$O(1)$
新增頂點	$O(n)$	$O(1)$	$O(1)$
刪除頂點	$O(n^2)$	$O(n+m)$	$O(n)$
記憶體空間佔用	$O(n^2)$	$O(n+m)$	$O(n+m)$

觀察表 9-2 ，似乎鄰接表（雜湊表）的時間效率與空間效率最優。但實際上，在鄰接矩陣中操作邊的效率更高，只需一次陣列訪問或賦值操作即可。綜合來看，鄰接矩陣體現了“以空間換時間”的原則，而鄰接表體現了“以時間換空間”的原則。

9.3　圖的走訪

樹代表的是“一對多”的關係，而圖則具有更高的自由度，可以表示任意的“多對多”關係。因此，我們可以把樹看作圖的一種特例。顯然，**樹的走訪操作也是圖的走訪操作的一種特例**。

圖和樹都需要應用搜索演算法來實現走訪操作。圖的走訪方式也可分為兩種：**廣度優先走訪**和**深度優先走訪**。

9.3.1　廣度優先走訪

廣度優先走訪是一種由近及遠的走訪方式，從某個節點出發，始終優先訪問距離最近的頂點，並一層層向外擴張。如圖 9-9 所示，從左上角頂點出發，首先走訪該頂點的所有鄰接頂點，然後走訪下一個頂點的所有鄰接頂點，以此類推，直至所有頂點訪問完畢。

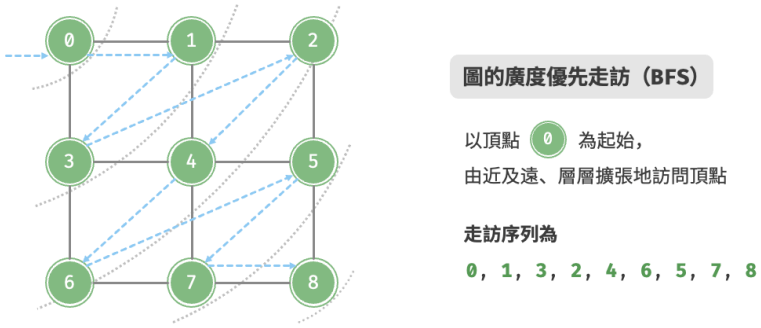


圖 9-9　圖的廣度優先走訪

1. 演算法實現

BFS 通常藉助佇列來實現，程式碼如下所示。佇列具有“先入先出”的性質，這與 BFS 的“由近及遠”的思想異曲同工。

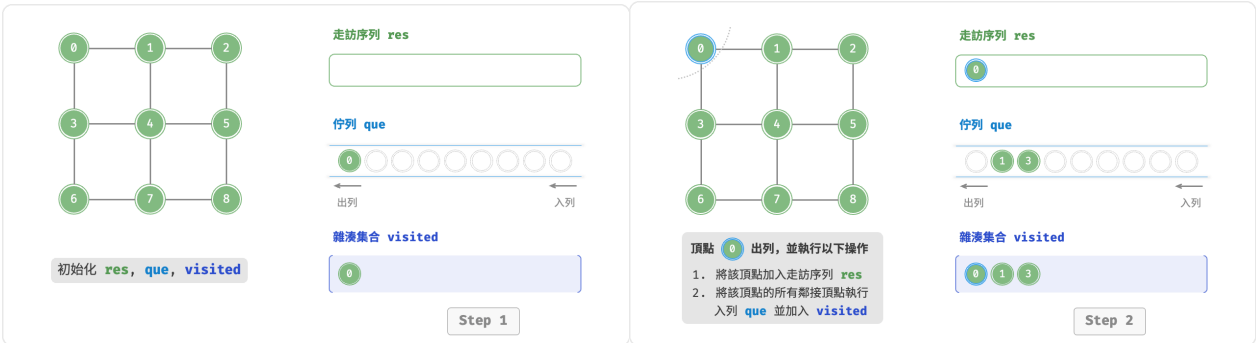
1. 將走訪起始頂點 `startVet` 加入列列，並開啟迴圈。
2. 在迴圈的每輪迭代中，彈出佇列首頂點並記錄訪問，然後將該頂點的所有鄰接頂點加入到佇列尾部。
3. 迴圈步驟 `2.` ，直到所有頂點被訪問完畢後結束。

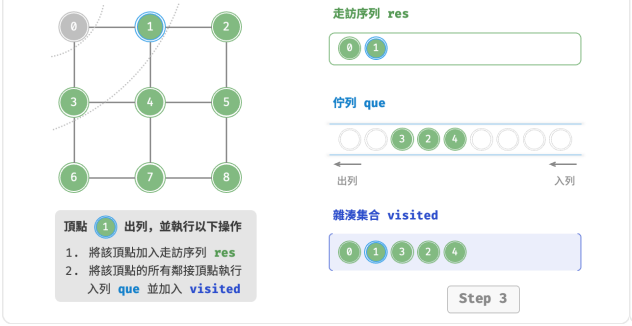
為了防止重複走訪頂點，我們需要藉助一個雜湊集合 `visited` 來記錄哪些節點已被訪問。

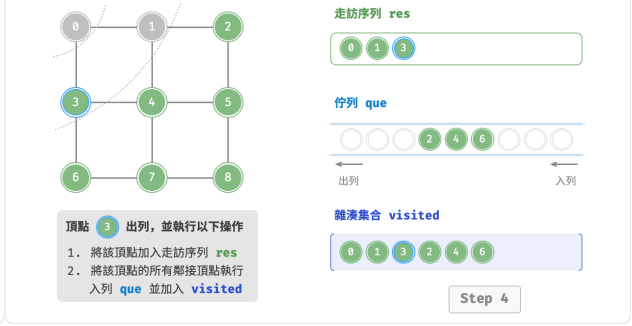
> **Tip**
> 雜湊集合可以看作一個只儲存 `key` 而不儲存 `value` 的雜湊表，它可以在 $O(1)$ 時間複雜度下進行 `key` 的增刪查改操作。根據 `key` 的唯一性，雜湊集合通常用於資料去重等場景。

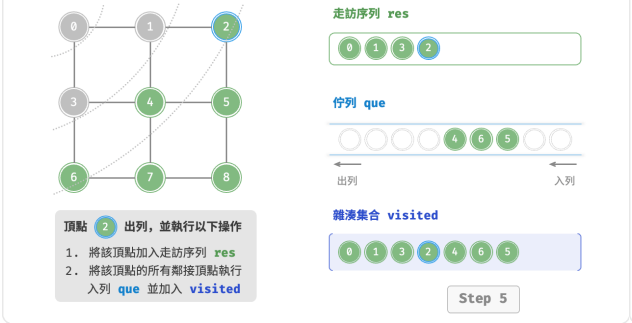
```rust
// === File: graph_bfs.rs ===

/* 廣度優先走訪 */
// 使用鄰接表來表示圖，以便獲取指定頂點的所有鄰接頂點
fn graph_bfs(graph: GraphAdjList, start_vet: Vertex) -> Vec<Vertex> {
    // 頂點走訪序列
    let mut res = vec![];
    // 雜湊集合，用於記錄已被訪問過的頂點
    let mut visited = HashSet::new();
    visited.insert(start_vet);
    // 佇列用於實現 BFS
    let mut que = VecDeque::new();
    que.push_back(start_vet);
    // 以頂點 vet 為起點，迴圈直至訪問完所有頂點
    while !que.is_empty() {
        let vet = que.pop_front().unwrap(); // 佇列首頂點出隊
        res.push(vet); // 記錄訪問頂點

        // 走訪該頂點的所有鄰接頂點
        if let Some(adj_vets) = graph.adj_list.get(&vet) {
            for &adj_vet in adj_vets {
                if visited.contains(&adj_vet) {
                    continue; // 跳過已被訪問的頂點
                }
                que.push_back(adj_vet); // 只入列未訪問的頂點
                visited.insert(adj_vet); // 標記該頂點已被訪問
            }
        }
    }
    // 返回頂點走訪序列
    res
}
```

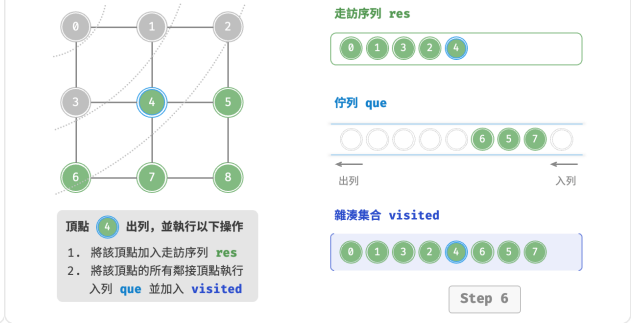
程式碼相對抽象，建議對照圖 9-10 來加深理解。

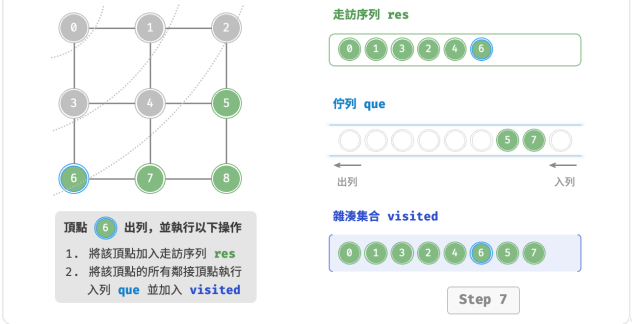


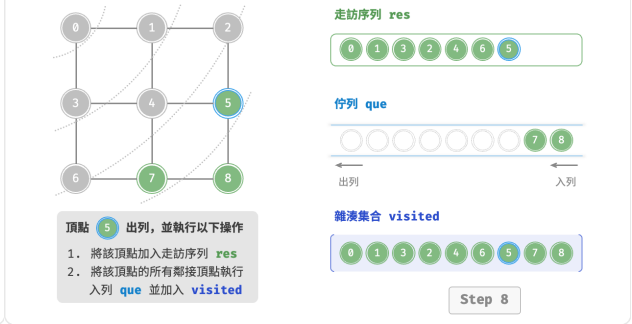

走訪序列 res
佇列 que
出列
入列
雜湊集合 visited
頂點 1 出列，並執行以下操作
1. 將該頂點加入走訪序列 res
2. 將該頂點的所有鄰接頂點執行入列 que 並加入 visited
Step 3

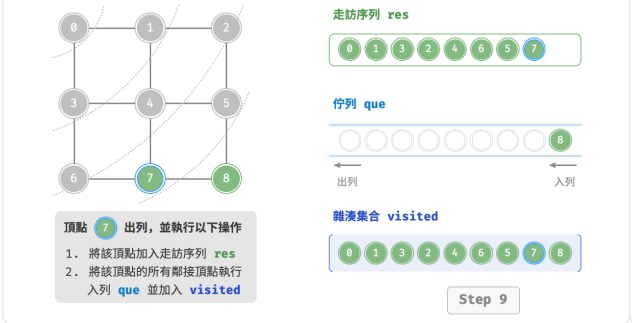

走訪序列 res
佇列 que
出列
入列
雜湊集合 visited
頂點 3 出列，並執行以下操作
1. 將該頂點加入走訪序列 res
2. 將該頂點的所有鄰接頂點執行入列 que 並加入 visited
Step 4

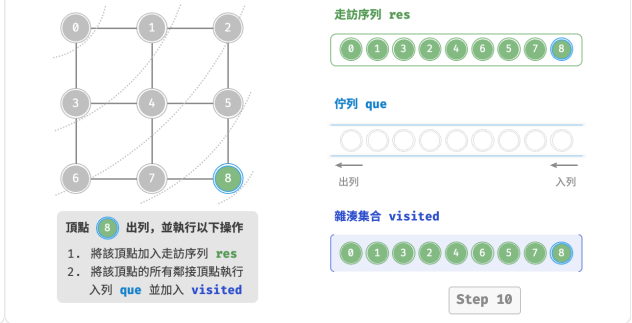

走訪序列 res
佇列 que
出列
入列
雜湊集合 visited
頂點 2 出列，並執行以下操作
1. 將該頂點加入走訪序列 res
2. 將該頂點的所有鄰接頂點執行入列 que 並加入 visited
Step 5


走訪序列 res
佇列 que
出列
入列
雜湊集合 visited
頂點 4 出列，並執行以下操作
1. 將該頂點加入走訪序列 res
2. 將該頂點的所有鄰接頂點執行入列 que 並加入 visited
Step 6


走訪序列 res
佇列 que
出列
入列
雜湊集合 visited
頂點 6 出列，並執行以下操作
1. 將該頂點加入走訪序列 res
2. 將該頂點的所有鄰接頂點執行入列 que 並加入 visited
Step 7


走訪序列 res
佇列 que
出列
入列
雜湊集合 visited
頂點 5 出列，並執行以下操作
1. 將該頂點加入走訪序列 res
2. 將該頂點的所有鄰接頂點執行入列 que 並加入 visited
Step 8


走訪序列 res
佇列 que
出列
入列
雜湊集合 visited
頂點 7 出列，並執行以下操作
1. 將該頂點加入走訪序列 res
2. 將該頂點的所有鄰接頂點執行入列 que 並加入 visited
Step 9


走訪序列 res
佇列 que
出列
入列
雜湊集合 visited
頂點 8 出列，並執行以下操作
1. 將該頂點加入走訪序列 res
2. 將該頂點的所有鄰接頂點執行入列 que 並加入 visited
Step 10

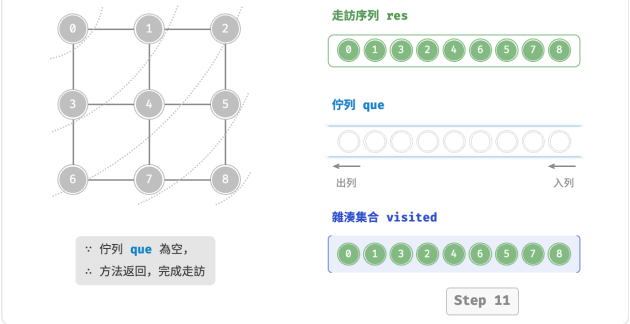

走訪序列 res
佇列 que
出列
入列
雜湊集合 visited
∵ 佇列 que 為空，
∴ 方法返回，完成走訪
Step 11

圖 9-10　圖的廣度優先走訪步驟

廣度優先走訪的序列是否唯一？

不唯一。廣度優先走訪只要求按“由近及遠”的順序走訪，**而多個相同距離的頂點的走訪順序允許被任意打亂**。以圖 9-10 為例，頂點 1、3 的訪問順序可以交換，頂點 2、4、6 的訪問順序也可以任意交換。

2. 複雜度分析

時間複雜度：所有頂點都會入列並出隊一次，使用 $O(|V|)$ 時間；在走訪鄰接頂點的過程中，由於是無向圖，因此所有邊都會被訪問 2 次，使用 $O(2|E|)$ 時間；總體使用 $O(|V| + |E|)$ 時間。

空間複雜度：串列 `res`，雜湊集合 `visited`，佇列 `que` 中的頂點數量最多為 $|V|$，使用 $O(|V|)$ 空間。

9.3.2　深度優先走訪

深度優先走訪是一種優先走到底、無路可走再回頭的走訪方式。如圖 9-11 所示，從左上角頂點出發，訪問當前頂點的某個鄰接頂點，直到走到盡頭時返回，再繼續走到盡頭並返回，以此類推，直至所有頂點走訪完成。

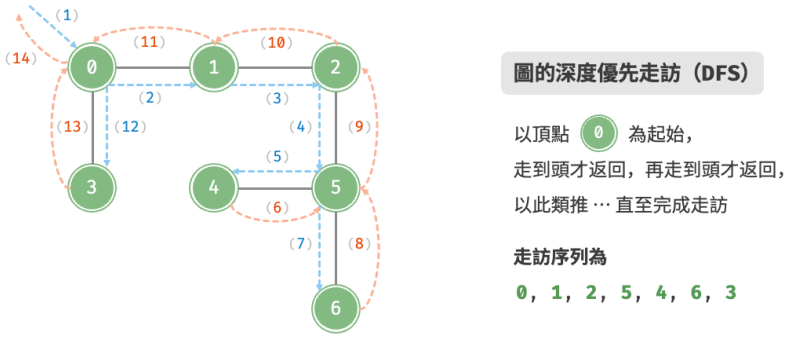


圖 9-11　圖的深度優先走訪

1. 演算法實現

這種“走到盡頭再返回”的演算法範式通常基於遞迴來實現。與廣度優先走訪類似，在深度優先走訪中，我們也需要藉助一個雜湊集合 `visited` 來記錄已被訪問的頂點，以避免重複訪問頂點。

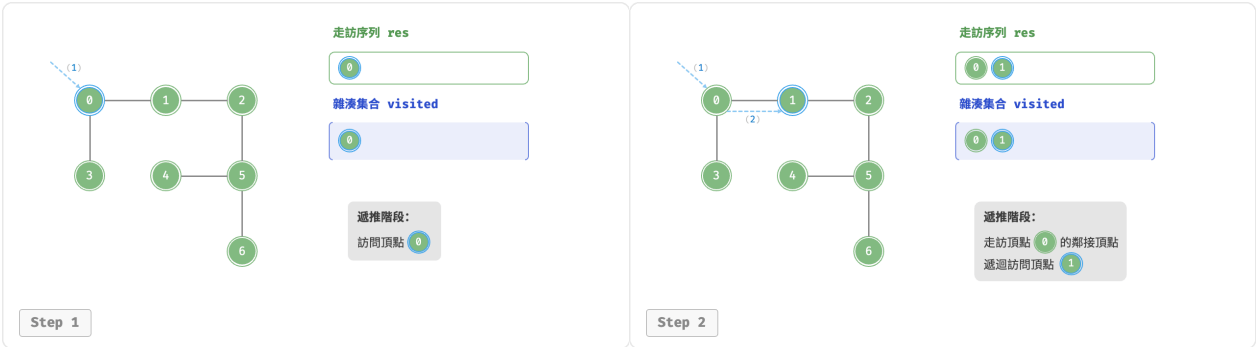
```
// === File: graph_dfs.rs ===

/* 深度優先走訪輔助函式 */
fn dfs(graph: &GraphAdjList, visited: &mut HashSet<Vertex>, res: &mut Vec<Vertex>, vet: Vertex) {
```

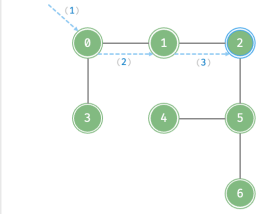
```
    res.push(vet); // 記錄訪問頂點
    visited.insert(vet); // 標記該頂點已被訪問
                         // 走訪該頂點的所有鄰接頂點
    if let Some(adj_vets) = graph.adj_list.get(&vet) {
        for &adj_vet in adj_vets {
            if visited.contains(&adj_vet) {
                continue; // 跳過已被訪問的頂點
            }
            // 遞迴訪問鄰接頂點
            dfs(graph, visited, res, adj_vet);
        }
    }
}

/* 深度優先走訪 */
// 使用鄰接表來表示圖，以便獲取指定頂點的所有鄰接頂點
fn graph_dfs(graph: GraphAdjList, start_vet: Vertex) -> Vec<Vertex> {
    // 頂點走訪序列
    let mut res = vec![];
    // 雜湊集合，用於記錄已被訪問過的頂點
    let mut visited = HashSet::new();
    dfs(&graph, &mut visited, &mut res, start_vet);

    res
}
```

深度優先走訪的演算法流程如圖 9-12 所示。

- **直虛線代表向下遞推**，表示開啟了一個新的遞迴方法來訪問新頂點。
- **曲虛線代表向上回溯**，表示此遞迴方法已經返回，回溯到了開啟此方法的位置。

為了加深理解，建議將圖 9-12 與程式碼結合起來，在腦中模擬（或者用筆畫下來）整個 DFS 過程，包括每個遞迴方法何時開啟、何時返回。





走訪序列 res

0 1 2

雜湊集合 visited

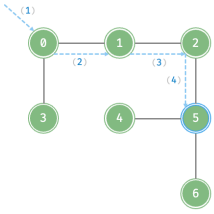
0 1 2

遞推階段：

走訪頂點 1 的鄰接頂點

遞迴訪問頂點 2

Step 3



走訪序列 res

0 1 2 5

雜湊集合 visited

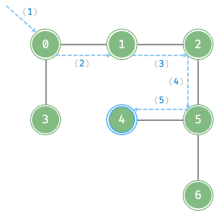
0 1 2 5

遞推階段：

走訪頂點 2 的鄰接頂點

遞迴訪問頂點 5

Step 4



走訪序列 res

0 1 2 5 4

雜湊集合 visited

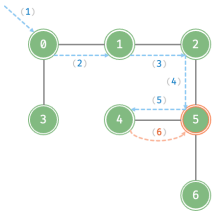
0 1 2 5 4

遞推階段：

走訪頂點 5 的鄰接頂點

遞迴訪問頂點 4

Step 5



走訪序列 res

0 1 2 5 4

雜湊集合 visited

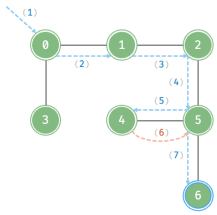
0 1 2 5 4

回溯階段：

頂點 4 的所有鄰接頂點都已訪問

因此該方法返回，回溯至頂點 5

Step 6



走訪序列 res

0 1 2 5 4 6

雜湊集合 visited

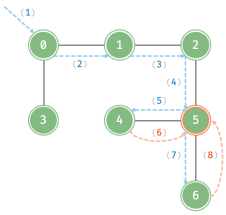
0 1 2 5 4 6

遞推階段：

走訪頂點 5 的鄰接頂點

遞迴訪問頂點 6

Step 7



走訪序列 res

0 1 2 5 4 6

雜湊集合 visited

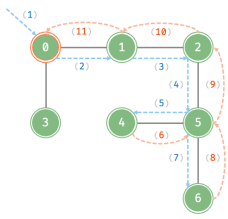
0 1 2 5 4 6

回溯階段：

頂點 6 的所有鄰接頂點都已訪問

因此該方法返回，回溯至頂點 5

Step 8



走訪序列 res

0 1 2 5 4 6

雜湊集合 visited

0 1 2 5 4 6

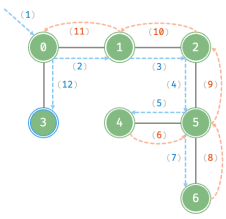
回溯階段：

頂點 6 的所有鄰接頂點都已訪問

因此該方法返回，回溯至頂點 5

以此類推 … 最終回溯至頂點 0

Step 9



走訪序列 res

0 1 2 5 4 6 3

雜湊集合 visited

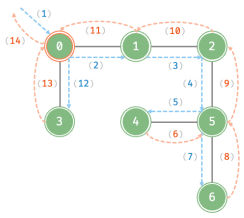
0 1 2 5 4 6 3

遞推階段：

走訪頂點 0 的鄰接頂點

遞迴訪問頂點 3

Step 10



走訪序列 res

0 1 2 5 4 6 3

雜湊集合 visited

0 1 2 5 4 6 3

回溯階段：

頂點 3 的所有鄰接頂點都已訪問

因此該方法返回，回溯至頂點 0

頂點 0 方法也返回，即完成走訪

Step 11

圖 9-12　圖的深度優先走訪步驟

深度優先走訪的序列是否唯一？
與廣度優先走訪類似，深度優先走訪序列的順序也不是唯一的。給定某頂點，先往哪個方向探索都可以，即鄰接頂點的順序可以任意打亂，都是深度優先走訪。
以樹的走訪為例，“根 $\rightarrow$ 左 $\rightarrow$ 右”“左 $\rightarrow$ 根 $\rightarrow$ 右”“左 $\rightarrow$ 右 $\rightarrow$ 根”分別對應前序、中序、後序走訪，它們展示了三種走訪優先順序，然而這三者都屬於深度優先走訪。

2. 複雜度分析

時間複雜度：所有頂點都會被訪問 1 次，使用 $O(|V|)$ 時間；所有邊都會被訪問 2 次，使用 $O(2|E|)$ 時間；總體使用 $O(|V| + |E|)$ 時間。

空間複雜度：串列 `res` ，雜湊集合 `visited` 頂點數量最多為 $|V|$ ，遞迴深度最大為 $|V|$ ，因此使用 $O(|V|)$ 空間。

9.4　小結

1. 重點回顧

- 圖由頂點和邊組成，可以表示為一組頂點和一組邊構成的集合。
- 相較於線性關係（鏈結串列）和分治關係（樹），網路關係（圖）具有更高的自由度，因而更為複雜。
- 有向圖的邊具有方向性，連通圖中的任意頂點均可達，有權圖的每條邊都包含權重變數。
- 鄰接矩陣利用矩陣來表示圖，每一行（列）代表一個頂點，矩陣元素代表邊，用 1 或 0 表示兩個頂點之間有邊或無邊。鄰接矩陣在增刪查改操作上效率很高，但空間佔用較多。
- 鄰接表使用多個鏈結串列來表示圖，第 i 個鏈結串列對應頂點 i ，其中儲存了該頂點的所有鄰接頂點。鄰接表相對於鄰接矩陣更加節省空間，但由於需要走訪鏈結串列來查詢邊，因此時間效率較低。
- 當鄰接表中的鏈結串列過長時，可以將其轉換為紅黑樹或雜湊表，從而提升查詢效率。
- 從演算法思想的角度分析，鄰接矩陣體現了“以空間換時間”，鄰接表體現了“以時間換空間”。
- 圖可用於建模各類現實系統，如社交網路、地鐵線路等。
- 樹是圖的一種特例，樹的走訪也是圖的走訪的一種特例。
- 圖的廣度優先走訪是一種由近及遠、層層擴張的搜尋方式，通常藉助佇列實現。
- 圖的深度優先走訪是一種優先走到底、無路可走時再回溯的搜尋方式，常基於遞迴來實現。

2. Q & A

Q：路徑的定義是頂點序列還是邊序列？

維基百科上不同語言版本的定義不一致：英文版是“路徑是一個邊序列”，而中文版是“路徑是一個頂點序列”。以下是英文版原文：In graph theory, a path in a graph is a finite or infinite sequence of edges which joins a sequence of vertices.

在本文中，路徑被視為一個邊序列，而不是一個頂點序列。這是因為兩個頂點之間可能存在多條邊連線，此時每條邊都對應一條路徑。

Q：非連通圖中是否會有無法走訪到的點？

在非連通圖中，從某個頂點出發，至少有一個頂點無法到達。走訪非連通圖需要設定多個起點，以走訪到圖的所有連通分量。

Q：在鄰接表中，“與該頂點相連的所有頂點”的頂點順序是否有要求？

可以是任意順序。但在實際應用中，可能需要按照指定規則來排序，比如按照頂點新增的次序，或者按照頂點值大小的順序等，這樣有助於快速查詢“帶有某種極值”的頂點。

第 10 章　搜尋

Abstract

搜尋是一場未知的冒險，我們或許需要走遍神秘空間的每個角落，又或許可以快速鎖定目標。

在這場尋覓之旅中，每一次探索都可能得到一個未曾料想的答案。

10.1　二分搜尋

二分搜尋（binary search）是一種基於分治策略的高效搜尋演算法。它利用資料的有序性，每輪縮小一半搜尋範圍，直至找到目標元素或搜尋區間為空為止。

Question

給定一個長度為 n 的陣列 `nums`，元素按從小到大的順序排列且不重複。請查詢並返回元素 `target` 在該陣列中的索引。若陣列不包含該元素，則返回 -1。示例如圖 10-1 所示。

圖 10-1　二分搜尋示例資料

如圖 10-2 所示，我們先初始化指標 $i = 0$ 和 $j = n - 1$，分別指向陣列首元素和尾元素，代表搜尋區間 $[0, n - 1]$。請注意，中括號表示閉區間，其包含邊界值本身。

接下來，迴圈執行以下兩步。

1. 計算中點索引 $m = \lfloor (i + j)/2 \rfloor$，其中 $\lfloor \rfloor$ 表示向下取整操作。
2. 判斷 `nums[m]` 和 `target` 的大小關係，分為以下三種情況。
    1. 當 `nums[m] < target` 時，說明 `target` 在區間 $[m + 1, j]$ 中，因此執行 $i = m + 1$。
    2. 當 `nums[m] > target` 時，說明 `target` 在區間 $[i, m - 1]$ 中，因此執行 $j = m - 1$。
    3. 當 `nums[m] = target` 時，說明找到 `target`，因此返回索引 m。

若陣列不包含目標元素，搜尋區間最終會縮小為空。此時返回 -1。

圖 10-2　二分搜尋流程

值得注意的是，由於 i 和 j 都是 `int` 型別，**因此 $i + j$ 可能會超出 `int` 型別的取值範圍**。為了避免大數越界，我們通常採用公式 $m = \lfloor i + (j - i)/2 \rfloor$ 來計算中點。

程式碼如下所示：

```
// === File: binary_search.rs ===

/* 二分搜尋（雙閉區間） */
fn binary_search(nums: &[i32], target: i32) -> i32 {
    // 初始化雙閉區間 [0, n-1] ，即 i, j 分別指向陣列首元素、尾元素
    let mut i = 0;
    let mut j = nums.len() as i32 - 1;
    // 迴圈，當搜尋區間為空時跳出（當 i > j 時為空）
    while i <= j {
        let m = i + (j - i) / 2; // 計算中點索引 m
        if nums[m as usize] < target {
            // 此情況說明 target 在區間 [m+1, j] 中
```

```
            i = m + 1;
        } else if nums[m as usize] > target {
            // 此情況說明 target 在區間 [i, m-1] 中
            j = m - 1;
        } else {
            // 找到目標元素，返回其索引
            return m;
        }
    }
    // 未找到目標元素，返回 -1
    return -1;
}
```

時間複雜度為 $O(\log n)$：在二分迴圈中，區間每輪縮小一半，因此迴圈次數為 $\log_2 n$ 。

空間複雜度為 $O(1)$：指標 i 和 j 使用常數大小空間。

10.1.1　區間表示方法

除了上述雙閉區間外，常見的區間表示還有“左閉右開”區間，定義為 $[0, n)$ ，即左邊界包含自身，右邊界不包含自身。在該表示下，區間 $[i, j)$ 在 $i = j$ 時為空。

我們可以基於該表示實現具有相同功能的二分搜尋演算法：

```
// === File: binary_search.rs ===

/* 二分搜尋（左閉右開區間） */
fn binary_search_lcro(nums: &[i32], target: i32) -> i32 {
    // 初始化左閉右開區間 [0, n) ，即 i, j 分別指向陣列首元素、尾元素 +1
    let mut i = 0;
    let mut j = nums.len() as i32;
    // 迴圈，當搜尋區間為空時跳出（當 i = j 時為空）
    while i < j {
        let m = i + (j - i) / 2; // 計算中點索引 m
        if nums[m as usize] < target {
            // 此情況說明 target 在區間 [m+1, j) 中
            i = m + 1;
        } else if nums[m as usize] > target {
            // 此情況說明 target 在區間 [i, m) 中
            j = m;
        } else {
            // 找到目標元素，返回其索引
            return m;
        }
    }
    // 未找到目標元素，返回 -1
```

```
    return -1;
}
```

如圖 10-3 所示，在兩種區間表示下，二分搜尋演算法的初始化、迴圈條件和縮小區間操作皆有所不同。

由於“雙閉區間”表示中的左右邊界都被定義為閉區間，因此透過指標 i 和指標 j 縮小區間的操作也是對稱的。這樣更不容易出錯，**因此一般建議採用“雙閉區間”的寫法**。

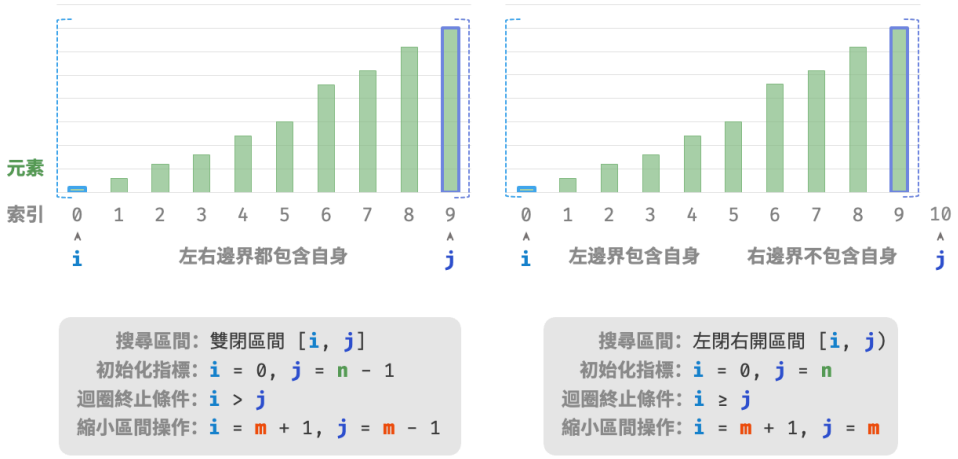


圖 10-3　兩種區間定義

10.1.2　優點與侷限性

二分搜尋在時間和空間方面都有較好的效能。

- 二分搜尋的時間效率高。在大資料量下，對數階的時間複雜度具有顯著優勢。例如，當資料大小 $n = 2^{20}$ 時，線性查詢需要 $2^{20} = 1048576$ 輪迴圈，而二分搜尋僅需 $\log_2 2^{20} = 20$ 輪迴圈。
- 二分搜尋無須額外空間。相較於需要藉助額外空間的搜尋演算法（例如雜湊查詢），二分搜尋更加節省空間。

然而，二分搜尋並非適用於所有情況，主要有以下原因。

- 二分搜尋僅適用於有序資料。若輸入資料無序，為了使用二分搜尋而專門進行排序，得不償失。因為排序演算法的時間複雜度通常為 $O(n \log n)$，比線性查詢和二分搜尋都更高。對於頻繁插入元素的場景，為保持陣列有序性，需要將元素插入到特定位置，時間複雜度為 $O(n)$，也是非常昂貴的。
- 二分搜尋僅適用於陣列。二分搜尋需要跳躍式（非連續地）訪問元素，而在鏈結串列中執行跳躍式訪問的效率較低，因此不適合應用在鏈結串列或基於鏈結串列實現的資料結構。
- 小資料量下，線性查詢效能更佳。線上性查詢中，每輪只需 1 次判斷操作；而在二分搜尋中，需要 1 次加法、1 次除法、1 ~ 3 次判斷操作、1 次加法（減法），共 4 ~ 6 個單元操作；因此，當資料量 n 較小時，線性查詢反而比二分搜尋更快。

10.2　二分搜尋插入點

二分搜尋不僅可用於搜尋目標元素，還可用於解決許多變種問題，比如搜尋目標元素的插入位置。

10.2.1　無重複元素的情況

Question

給定一個長度為 n 的有序陣列 `nums` 和一個元素 `target`，陣列不存在重複元素。現將 `target` 插入陣列 `nums` 中，並保持其有序性。若陣列中已存在元素 `target`，則插入到其左方。請返回插入後 `target` 在陣列中的索引。示例如圖 10-4 所示。

圖 10-4　二分搜尋插入點示例資料

如果想複用上一節的二分搜尋程式碼，則需要回答以下兩個問題。

問題一：當陣列中包含 `target` 時，插入點的索引是否是該元素的索引？

題目要求將 `target` 插入到相等元素的左邊，這意味著新插入的 `target` 替換了原來 `target` 的位置。也就是說，**當陣列包含 `target` 時，插入點的索引就是該 `target` 的索引**。

問題二：當陣列中不存在 `target` 時，插入點是哪個元素的索引？

進一步思考二分搜尋過程：當 `nums[m] < target` 時 i 移動，這意味著指標 i 在向大於等於 `target` 的元素靠近。同理，指標 j 始終在向小於等於 `target` 的元素靠近。

因此二分結束時一定有：i 指向首個大於 `target` 的元素，j 指向首個小於 `target` 的元素。**易得當陣列不包含 `target` 時，插入索引為 i**。程式碼如下所示：

```rust
// === File: binary_search_insertion.rs ===

/* 二分搜尋插入點（無重複元素） */
fn binary_search_insertion_simple(nums: &[i32], target: i32) -> i32 {
    let (mut i, mut j) = (0, nums.len() as i32 - 1); // 初始化雙閉區間 [0, n-1]
    while i <= j {
```

```
        let m = i + (j - i) / 2; // 計算中點索引 m
        if nums[m as usize] < target {
            i = m + 1; // target 在區間 [m+1, j] 中
        } else if nums[m as usize] > target {
            j = m - 1; // target 在區間 [i, m-1] 中
        } else {
            return m;
        }
    }
    // 未找到 target ，返回插入點 i
    i
}
```

10.2.2　存在重複元素的情況

> **Question**
> 在上一題的基礎上，規定陣列可能包含重複元素，其餘不變。

假設陣列中存在多個 `target` ，則普通二分搜尋只能返回其中一個 `target` 的索引，**而無法確定該元素的左邊和右邊還有多少 `target`**。

題目要求將目標元素插入到最左邊，**所以我們需要查詢陣列中最左一個 `target` 的索引**。初步考慮透過圖 10-5 所示的步驟實現。

1. 執行二分搜尋，得到任意一個 `target` 的索引，記為 k 。
2. 從索引 k 開始，向左進行線性走訪，當找到最左邊的 `target` 時返回。

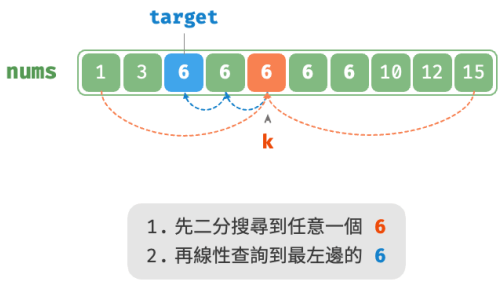


圖 10-5　線性查詢重複元素的插入點

此方法雖然可用，但其包含線性查詢，因此時間複雜度為 $O(n)$ 。當陣列中存在很多重複的 `target` 時，該方法效率很低。

現考慮拓展二分搜尋程式碼。如圖 10-6 所示，整體流程保持不變，每輪先計算中點索引 m ，再判斷 `target` 和 `nums[m]` 的大小關係，分為以下幾種情況。

- 當 `nums[m] < target` 或 `nums[m] > target` 時，說明還沒有找到 `target`，因此採用普通二分搜尋的縮小區間操作，**從而使指標 i 和 j 向 `target` 靠近**。
- 當 `nums[m] == target` 時，說明小於 `target` 的元素在區間 $[i, m-1]$ 中，因此採用 $j = m - 1$ 來縮小區間，**從而使指標 j 向小於 `target` 的元素靠近**。

迴圈完成後，i 指向最左邊的 `target`，j 指向首個小於 `target` 的元素，**因此索引 i 就是插入點**。

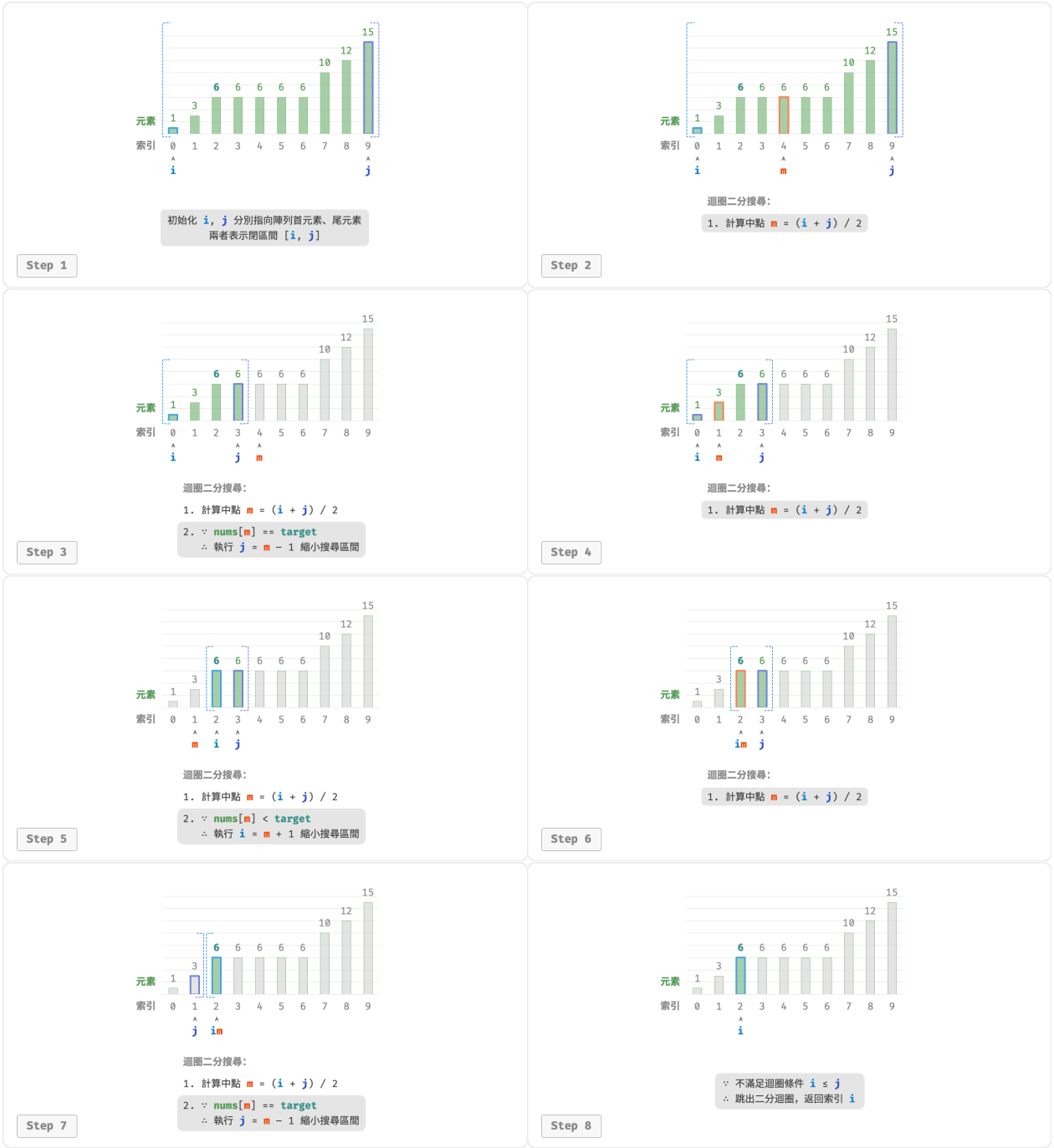


圖 10-6　二分搜尋重複元素的插入點的步驟

觀察以下程式碼，判斷分支 `nums[m] > target` 和 `nums[m] == target` 的操作相同，因此兩者可以合併。

即便如此，我們仍然可以將判斷條件保持展開，因為其邏輯更加清晰、可讀性更好。

```rust
// === File: binary_search_insertion.rs ===

/* 二分搜尋插入點（存在重複元素） */
pub fn binary_search_insertion(nums: &[i32], target: i32) -> i32 {
    let (mut i, mut j) = (0, nums.len() as i32 - 1); // 初始化雙閉區間 [0, n-1]
    while i <= j {
        let m = i + (j - i) / 2; // 計算中點索引 m
        if nums[m as usize] < target {
            i = m + 1; // target 在區間 [m+1, j] 中
        } else if nums[m as usize] > target {
            j = m - 1; // target 在區間 [i, m-1] 中
        } else {
            j = m - 1; // 首個小於 target 的元素在區間 [i, m-1] 中
        }
    }
    // 返回插入點 i
    i
}
```

> **Tip**
> 本節的程式碼都是“雙閉區間”寫法。有興趣的讀者可以自行實現“左閉右開”寫法。

總的來看，二分搜尋無非就是給指標 i 和 j 分別設定搜尋目標，目標可能是一個具體的元素（例如 `target`），也可能是一個元素範圍（例如小於 `target` 的元素）。

在不斷的迴圈二分中，指標 i 和 j 都逐漸逼近預先設定的目標。最終，它們或是成功找到答案，或是越過邊界後停止。

10.3　二分搜尋邊界

10.3.1　查詢左邊界

> **Question**
> 給定一個長度為 n 的有序陣列 `nums`，其中可能包含重複元素。請返回陣列中最左一個元素 `target` 的索引。若陣列中不包含該元素，則返回 -1 。

回憶二分搜尋插入點的方法，搜尋完成後 i 指向最左一個 `target`，**因此查詢插入點本質上是在查詢最左一個 `target` 的索引**。

考慮透過查詢插入點的函式實現查詢左邊界。請注意，陣列中可能不包含 `target`，這種情況可能導致以下兩種結果。

- 插入點的索引 i 越界。
- 元素 `nums[i]` 與 `target` 不相等。

當遇到以上兩種情況時，直接返回 -1 即可。程式碼如下所示：

```
// === File: binary_search_edge.rs ===

/* 二分搜尋最左一個 target */
fn binary_search_left_edge(nums: &[i32], target: i32) -> i32 {
    // 等價於查詢 target 的插入點
    let i = binary_search_insertion(nums, target);
    // 未找到 target ，返回 -1
    if i == nums.len() as i32 || nums[i as usize] != target {
        return -1;
    }
    // 找到 target ，返回索引 i
    i
}
```

10.3.2　查詢右邊界

那麼如何查詢最右一個 `target` 呢？最直接的方式是修改程式碼，替換在 `nums[m] == target` 情況下的指標收縮操作。程式碼在此省略，有興趣的讀者可以自行實現。

下面我們介紹兩種更加取巧的方法。

1.　複用查詢左邊界

實際上，我們可以利用查詢最左元素的函式來查詢最右元素，具體方法為：**將查詢最右一個 `target` 轉化為查詢最左一個 `target + 1`**。

如圖 10-7 所示，查詢完成後，指標 i 指向最左一個 `target + 1`（如果存在），而 j 指向最右一個 `target`，**因此返回 j 即可**。

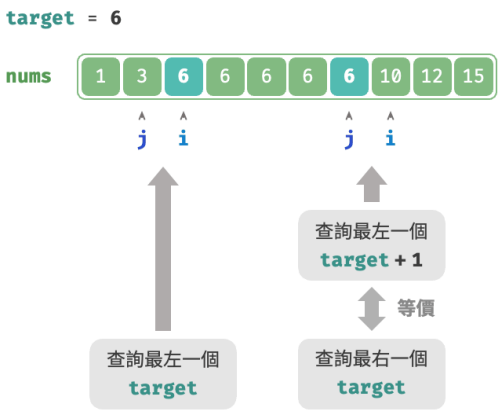


圖 10-7　將查詢右邊界轉化為查詢左邊界

請注意，返回的插入點是 i ，因此需要將其減 1 ，從而獲得 j ：

```rust
// === File: binary_search_edge.rs ===

/* 二分搜尋最右一個 target */
fn binary_search_right_edge(nums: &[i32], target: i32) -> i32 {
    // 轉化為查詢最左一個 target + 1
    let i = binary_search_insertion(nums, target + 1);
    // j 指向最右一個 target ，i 指向首個大於 target 的元素
    let j = i - 1;
    // 未找到 target ，返回 -1
    if j == -1 || nums[j as usize] != target {
        return -1;
    }
    // 找到 target ，返回索引 j
    j
}
```

2. 轉化為查詢元素

我們知道，當陣列不包含 `target` 時，最終 i 和 j 會分別指向首個大於、小於 `target` 的元素。

因此，如圖 10-8 所示，我們可以構造一個陣列中不存在的元素，用於查詢左右邊界。

- 查詢最左一個 `target` ：可以轉化為查詢 `target - 0.5` ，並返回指標 i 。
- 查詢最右一個 `target` ：可以轉化為查詢 `target + 0.5` ，並返回指標 j 。

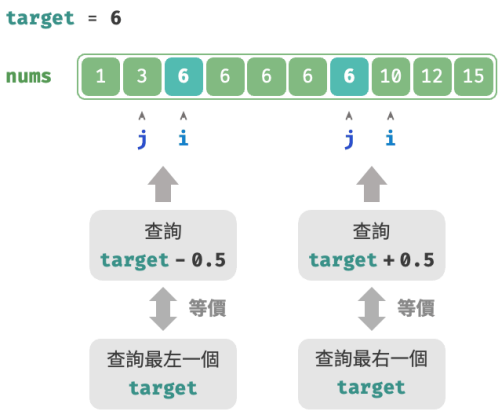


圖 10-8　將查詢邊界轉化為查詢元素

程式碼在此省略，以下兩點值得注意。

- 給定陣列不包含小數，這意味著我們無須關心如何處理相等的情況。
- 因為該方法引入了小數，所以需要將函式中的變數 `target` 改為浮點數型別（Python 無須改動）。

10.4　雜湊最佳化策略

在演算法題中，**我們常透過將線性查詢替換為雜湊查詢來降低演算法的時間複雜度**。我們藉助一個演算法題來加深理解。

> **Question**
> 給定一個整數陣列 `nums` 和一個目標元素 `target`，請在陣列中搜索“和”為 `target` 的兩個元素，並返回它們的陣列索引。返回任意一個解即可。

10.4.1　線性查詢：以時間換空間

考慮直接走訪所有可能的組合。如圖 10-9 所示，我們開啟一個兩層迴圈，在每輪中判斷兩個整數的和是否為 `target`，若是，則返回它們的索引。

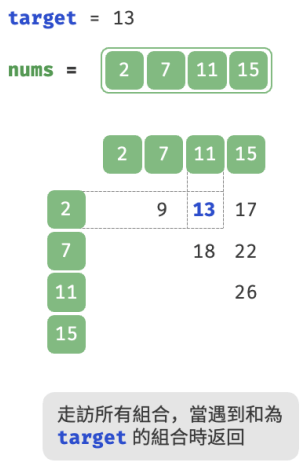


圖 10-9　線性查詢求解兩數之和

程式碼如下所示：

```rust
// === File: two_sum.rs ===

/* 方法一：暴力列舉 */
pub fn two_sum_brute_force(nums: &Vec<i32>, target: i32) -> Option<Vec<i32>> {
    let size = nums.len();
    // 兩層迴圈，時間複雜度為 O(n^2)
```

```
    for i in 0..size - 1 {
        for j in i + 1..size {
            if nums[i] + nums[j] == target {
                return Some(vec![i as i32, j as i32]);
            }
        }
    }
    None
}
```

此方法的時間複雜度為 $O(n^2)$，空間複雜度為 $O(1)$，在大資料量下非常耗時。

10.4.2　雜湊查詢：以空間換時間

考慮藉助一個雜湊表，鍵值對分別為陣列元素和元素索引。迴圈走訪陣列，每輪執行圖 10-10 所示的步驟。

1. 判斷數字 `target - nums[i]` 是否在雜湊表中，若是，則直接返回這兩個元素的索引。
2. 將鍵值對 `nums[i]` 和索引 `i` 新增進雜湊表。

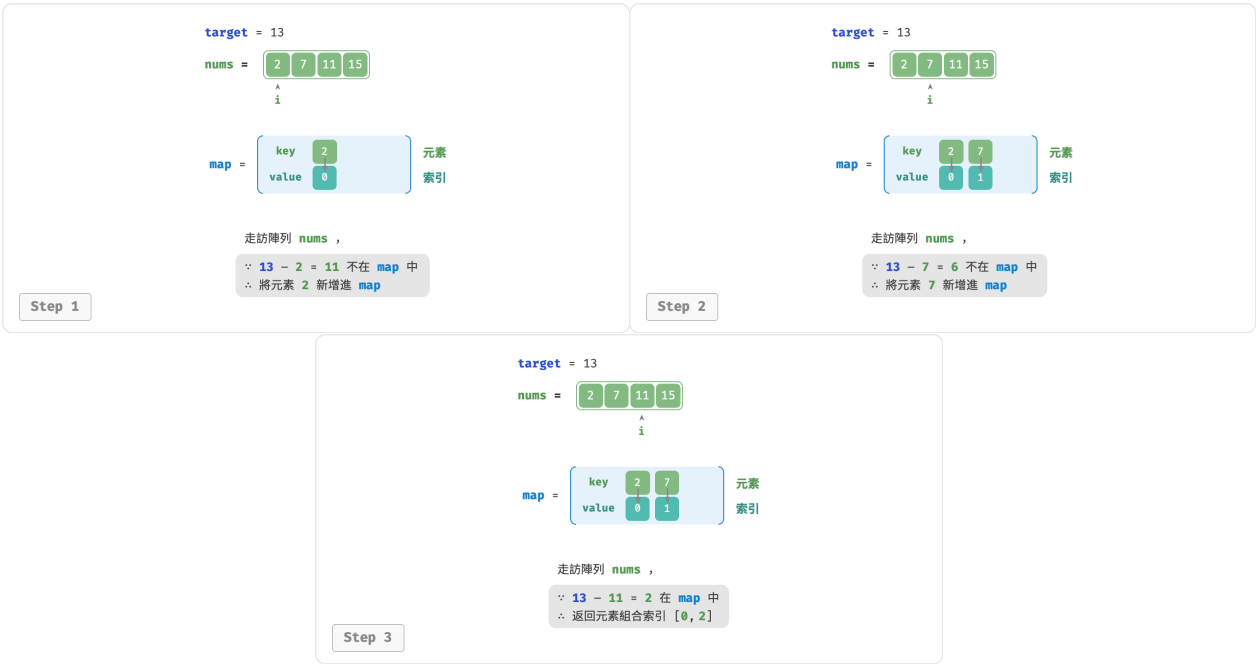


圖 10-10　輔助雜湊表求解兩數之和

實現程式碼如下所示，僅需單層迴圈即可：

```
// === File: two_sum.rs ===

/* 方法二：輔助雜湊表 */
```

```rust
pub fn two_sum_hash_table(nums: &Vec<i32>, target: i32) -> Option<Vec<i32>> {
    // 輔助雜湊表，空間複雜度為 O(n)
    let mut dic = HashMap::new();
    // 單層迴圈，時間複雜度為 O(n)
    for (i, num) in nums.iter().enumerate() {
        match dic.get(&(target - num)) {
            Some(v) => return Some(vec![*v as i32, i as i32]),
            None => dic.insert(num, i as i32),
        };
    }
    None
}
```

此方法透過雜湊查詢將時間複雜度從 $O(n^2)$ 降至 $O(n)$，大幅提升執行效率。

由於需要維護一個額外的雜湊表，因此空間複雜度為 $O(n)$。**儘管如此，該方法的整體時空效率更為均衡，因此它是本題的最優解法**。

10.5　重識搜尋演算法

搜尋演算法（searching algorithm）用於在資料結構（例如陣列、鏈結串列、樹或圖）中搜索一個或一組滿足特定條件的元素。

搜尋演算法可根據實現思路分為以下兩類。

- **透過走訪資料結構來定位目標元素**，例如陣列、鏈結串列、樹和圖的走訪等。
- **利用資料組織結構或資料包含的先驗資訊，實現高效元素查詢**，例如二分搜尋、雜湊查詢和二元搜尋樹查詢等。

不難發現，這些知識點都已在前面的章節中介紹過，因此搜尋演算法對於我們來說並不陌生。在本節中，我們將從更加系統的視角切入，重新審視搜尋演算法。

10.5.1　暴力搜尋

暴力搜尋透過走訪資料結構的每個元素來定位目標元素。

- “線性搜尋”適用於陣列和鏈結串列等線性資料結構。它從資料結構的一端開始，逐個訪問元素，直到找到目標元素或到達另一端仍沒有找到目標元素為止。
- “廣度優先搜尋”和“深度優先搜尋”是圖和樹的兩種走訪策略。廣度優先搜尋從初始節點開始逐層搜尋，由近及遠地訪問各個節點。深度優先搜尋從初始節點開始，沿著一條路徑走到頭，再回溯並嘗試其他路徑，直到走訪完整個資料結構。

暴力搜尋的優點是簡單且通用性好，**無須對資料做預處理和藉助額外的資料結構**。

然而，**此類演算法的時間複雜度為 $O(n)$**，其中 n 為元素數量，因此在資料量較大的情況下效能較差。

10.5.2　自適應搜尋

自適應搜尋利用資料的特有屬性（例如有序性）來最佳化搜尋過程，從而更高效地定位目標元素。

- “二分搜尋”利用資料的有序性實現高效查詢，僅適用於陣列。
- “雜湊查詢”利用雜湊表將搜尋資料和目標資料建立為鍵值對對映，從而實現查詢操作。
- “樹查詢”在特定的樹結構（例如二元搜尋樹）中，基於比較節點值來快速排除節點，從而定位目標元素。

此類演算法的優點是效率高，**時間複雜度可達到 $O(\log n)$ 甚至 $O(1)$** 。

然而，**使用這些演算法往往需要對資料進行預處理**。例如，二分搜尋需要預先對陣列進行排序，雜湊查詢和樹查詢都需要藉助額外的資料結構，維護這些資料結構也需要額外的時間和空間開銷。

> **Tip**
> 自適應搜尋演算法常被稱為查詢演算法，**主要用於在特定資料結構中快速檢索目標元素**。

10.5.3　搜尋方法選取

給定大小為 n 的一組資料，我們可以使用線性搜尋、二分搜尋、樹查詢、雜湊查詢等多種方法從中搜索目標元素。各個方法的工作原理如圖 10-11 所示。

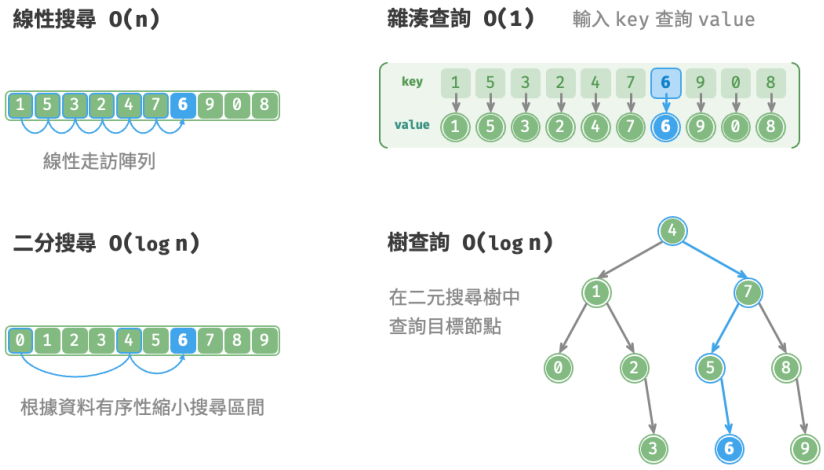


圖 10-11　多種搜尋策略

上述幾種方法的操作效率與特性如表 10-1 所示。

表 10-1　查詢演算法效率對比

	線性搜尋	二分搜尋	樹查詢	雜湊查詢
查詢元素	$O(n)$	$O(\log n)$	$O(\log n)$	$O(1)$
插入元素	$O(1)$	$O(n)$	$O(\log n)$	$O(1)$
刪除元素	$O(n)$	$O(n)$	$O(\log n)$	$O(1)$
額外空間	$O(1)$	$O(1)$	$O(n)$	$O(n)$
資料預處理	/	排序 $O(n \log n)$	建樹 $O(n \log n)$	建雜湊表 $O(n)$
資料是否有序	無序	有序	有序	無序

搜尋演算法的選擇還取決於資料體量、搜尋效能要求、資料查詢與更新頻率等。

線性搜尋

- 通用性較好，無須任何資料預處理操作。假如我們僅需查詢一次資料，那麼其他三種方法的資料預處理的時間比線性搜尋的時間還要更長。
- 適用於體量較小的資料，此情況下時間複雜度對效率影響較小。
- 適用於資料更新頻率較高的場景，因為該方法不需要對資料進行任何額外維護。

二分搜尋

- 適用於大資料量的情況，效率表現穩定，最差時間複雜度為 $O(\log n)$ 。
- 資料量不能過大，因為儲存陣列需要連續的記憶體空間。
- 不適用於高頻增刪資料的場景，因為維護有序陣列的開銷較大。

雜湊查詢

- 適合對查詢效能要求很高的場景，平均時間複雜度為 $O(1)$ 。
- 不適合需要有序資料或範圍查詢的場景，因為雜湊表無法維護資料的有序性。
- 對雜湊函式和雜湊衝突處理策略的依賴性較高，具有較大的效能劣化風險。
- 不適合資料量過大的情況，因為雜湊表需要額外空間來最大程度地減少衝突，從而提供良好的查詢效能。

樹查詢

- 適用於海量資料，因為樹節點在記憶體中是分散儲存的。
- 適合需要維護有序資料或範圍查詢的場景。
- 在持續增刪節點的過程中，二元搜尋樹可能產生傾斜，時間複雜度劣化至 $O(n)$ 。
- 若使用 AVL 樹或紅黑樹，則各項操作可在 $O(\log n)$ 效率下穩定執行，但維護樹平衡的操作會增加額外的開銷。

10.6　小結

- 二分搜尋依賴資料的有序性，透過迴圈逐步縮減一半搜尋區間來進行查詢。它要求輸入資料有序，且僅適用於陣列或基於陣列實現的資料結構。

- 暴力搜尋透過走訪資料結構來定位資料。線性搜尋適用於陣列和鏈結串列，廣度優先搜尋和深度優先搜尋適用於圖和樹。此類演算法通用性好，無須對資料進行預處理，但時間複雜度 $O(n)$ 較高。
- 雜湊查詢、樹查詢和二分搜尋屬於高效搜尋方法，可在特定資料結構中快速定位目標元素。此類演算法效率高，時間複雜度可達 $O(\log n)$ 甚至 $O(1)$ ，但通常需要藉助額外資料結構。
- 實際中，我們需要對資料體量、搜尋效能要求、資料查詢和更新頻率等因素進行具體分析，從而選擇合適的搜尋方法。
- 線性搜尋適用於小型或頻繁更新的資料；二分搜尋適用於大型、排序的資料；雜湊查詢適用於對查詢效率要求較高且無須範圍查詢的資料；樹查詢適用於需要維護順序和支持範圍查詢的大型動態資料。
- 用雜湊查詢替換線性查詢是一種常用的最佳化執行時間的策略，可將時間複雜度從 $O(n)$ 降至 $O(1)$ 。

第 11 章　排序

Abstract

排序猶如一把將混亂變為秩序的魔法鑰匙，使我們能以更高效的方式理解與處理資料。

無論是簡單的升序，還是複雜的分類排列，排序都向我們展示了資料的和諧美感。

11.1　排序演算法

排序演算法（sorting algorithm）用於對一組資料按照特定順序進行排列。排序演算法有著廣泛的應用，因為有序資料通常能夠被更高效地查詢、分析和處理。

如圖 11-1 所示，排序演算法中的資料型別可以是整數、浮點數、字元或字串等。排序的判斷規則可根據需求設定，如數字大小、字元 ASCII 碼順序或自定義規則。

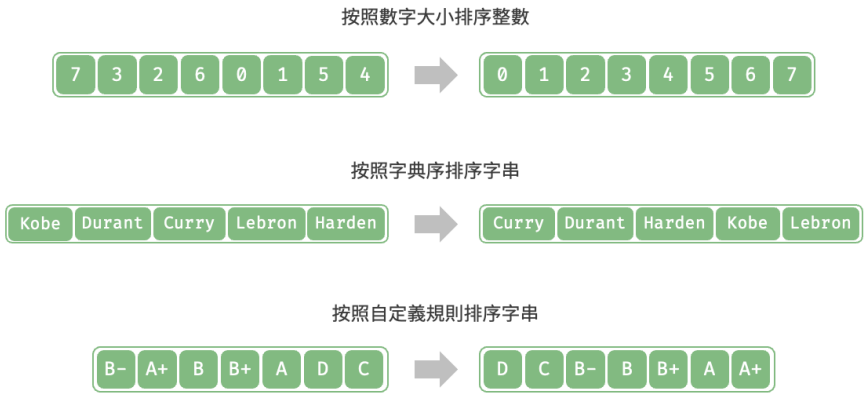


圖 11-1　資料型別和判斷規則示例

11.1.1　評價維度

執行效率：我們期望排序演算法的時間複雜度儘量低，且總體操作數量較少（時間複雜度中的常數項變小）。對於大資料量的情況，執行效率顯得尤為重要。

就地性：顧名思義，**原地排序**透過在原陣列上直接操作實現排序，無須藉助額外的輔助陣列，從而節省記憶體。通常情況下，原地排序的資料搬運操作較少，執行速度也更快。

穩定性：**穩定排序**在完成排序後，相等元素在陣列中的相對順序不發生改變。

穩定排序是多級排序場景的必要條件。假設我們有一個儲存學生資訊的表格，第 1 列和第 2 列分別是姓名和年齡。在這種情況下，**非穩定排序**可能導致輸入資料的有序性喪失：

```
# 輸入資料是按照姓名排序好的
# (name, age)
  ('A', 19)
  ('B', 18)
  ('C', 21)
  ('D', 19)
  ('E', 23)

# 假設使用非穩定排序演算法按年齡排序串列，
# 結果中 ('D', 19) 和 ('A', 19) 的相對位置改變，
```

```
# 輸入資料按姓名排序的性質丟失
('B', 18)
('D', 19)
('A', 19)
('C', 21)
('E', 23)
```

自適應性：自適應排序能夠利用輸入資料已有的順序資訊來減少計算量，達到更優的時間效率。自適應排序演算法的最佳時間複雜度通常優於平均時間複雜度。

是否基於比較：基於比較的排序依賴比較運算子（$<$、$=$、$>$）來判斷元素的相對順序，從而排序整個陣列，理論最優時間複雜度為 $O(n \log n)$ 。而非比較排序不使用比較運算子，時間複雜度可達 $O(n)$ ，但其通用性相對較差。

11.1.2　理想排序演算法

執行快、原地、穩定、自適應、通用性好。顯然，迄今為止尚未發現兼具以上所有特性的排序演算法。因此，在選擇排序演算法時，需要根據具體的資料特點和問題需求來決定。

接下來，我們將共同學習各種排序演算法，並基於上述評價維度對各個排序演算法的優缺點進行分析。

11.2　選擇排序

選擇排序（selection sort）的工作原理非常簡單：開啟一個迴圈，每輪從未排序區間選擇最小的元素，將其放到已排序區間的末尾。

設陣列的長度為 n ，選擇排序的演算法流程如圖 11-2 所示。

1. 初始狀態下，所有元素未排序，即未排序（索引）區間為 $[0, n-1]$ 。
2. 選取區間 $[0, n-1]$ 中的最小元素，將其與索引 0 處的元素交換。完成後，陣列前 1 個元素已排序。
3. 選取區間 $[1, n-1]$ 中的最小元素，將其與索引 1 處的元素交換。完成後，陣列前 2 個元素已排序。
4. 以此類推。經過 $n-1$ 輪選擇與交換後，陣列前 $n-1$ 個元素已排序。
5. 僅剩的一個元素必定是最大元素，無須排序，因此陣列排序完成。

圖 11-2　選擇排序步驟

在程式碼中，我們用 k 來記錄未排序區間內的最小元素：

```rust
// === File: selection_sort.rs ===

/* 選擇排序 */
fn selection_sort(nums: &mut [i32]) {
    if nums.is_empty() {
        return;
    }
    let n = nums.len();
    // 外迴圈：未排序區間為 [i, n-1]
    for i in 0..n - 1 {
        // 內迴圈：找到未排序區間內的最小元素
        let mut k = i;
        for j in i + 1..n {
            if nums[j] < nums[k] {
                k = j; // 記錄最小元素的索引
            }
        }
        // 將該最小元素與未排序區間的首個元素交換
        nums.swap(i, k);
    }
}
```

11.2.1　演算法特性

- **時間複雜度為 $O(n^2)$、非自適應排序**：外迴圈共 $n - 1$ 輪，第一輪的未排序區間長度為 n ，最後一輪的未排序區間長度為 2 ，即各輪外迴圈分別包含 n、$n - 1$、…、3、2 輪內迴圈，求和為 $\frac{(n-1)(n+2)}{2}$ 。
- **空間複雜度為 $O(1)$、原地排序**：指標 i 和 j 使用常數大小的額外空間。
- **非穩定排序**：如圖 11-3 所示，元素 `nums[i]` 有可能被交換至與其相等的元素的右邊，導致兩者的相對順序發生改變。

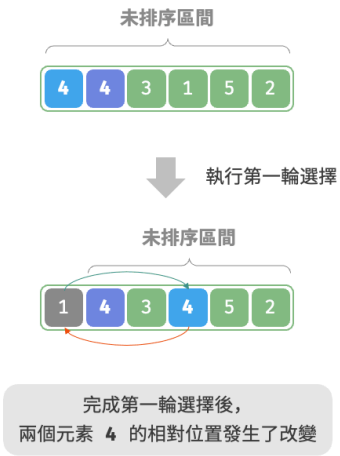


圖 11-3　選擇排序非穩定示例

11.3　泡沫排序

泡沫排序（bubble sort）透過連續地比較與交換相鄰元素實現排序。這個過程就像氣泡從底部升到頂部一樣，因此得名泡沫排序。

如圖 11-4 所示，冒泡過程可以利用元素交換操作來模擬：從陣列最左端開始向右走訪，依次比較相鄰元素大小，如果“左元素 > 右元素”就交換二者。走訪完成後，最大的元素會被移動到陣列的最右端。

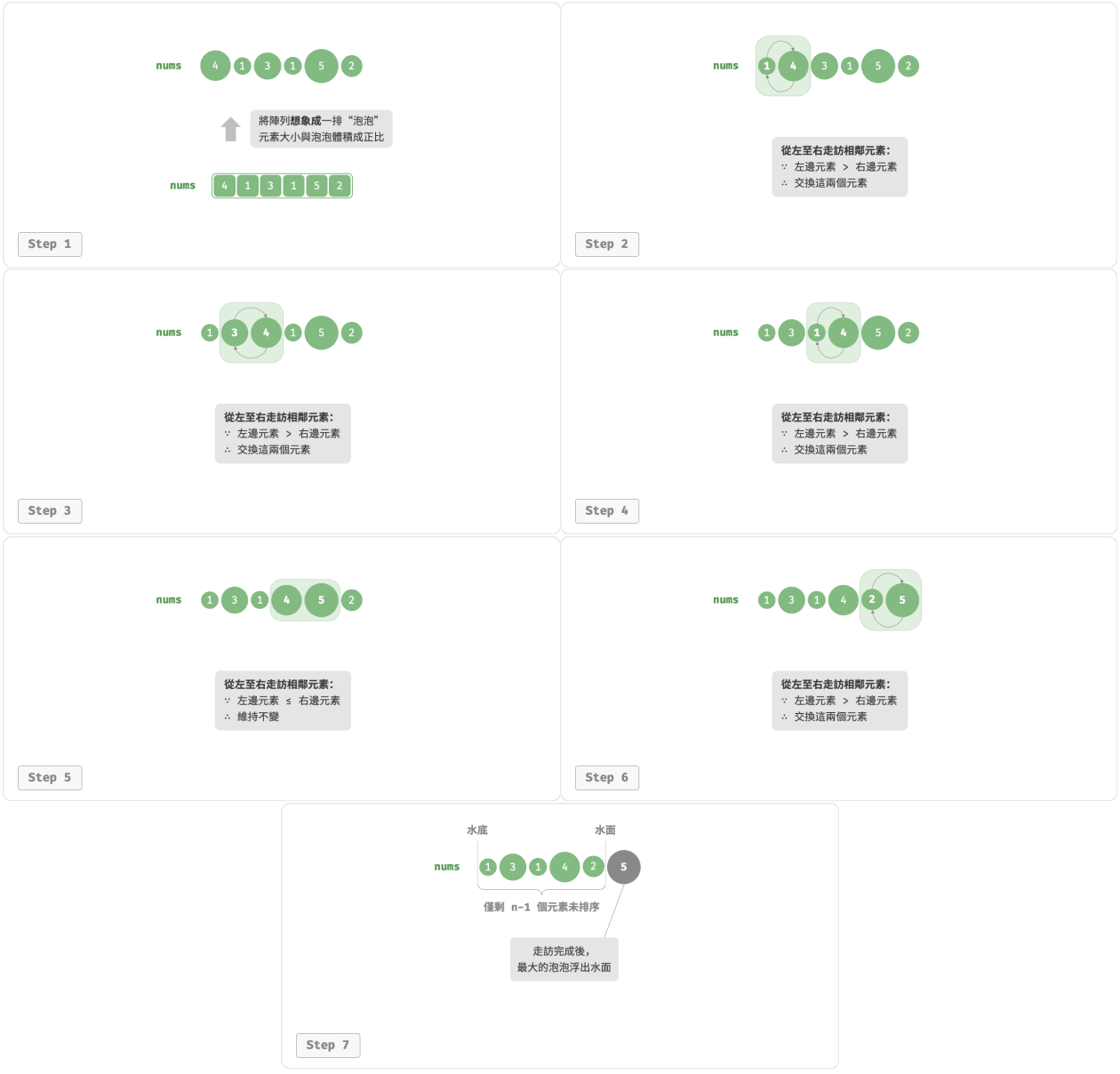


圖 11-4　利用元素交換操作模擬冒泡

11.3.1　演算法流程

設陣列的長度為 n，泡沫排序的步驟如圖 11-5 所示。

1. 首先，對 n 個元素執行“冒泡”，**將陣列的最大元素交換至正確位置**。
2. 接下來，對剩餘 $n-1$ 個元素執行“冒泡”，**將第二大元素交換至正確位置**。
3. 以此類推，經過 $n-1$ 輪“冒泡”後，**前 $n-1$ 大的元素都被交換至正確位置**。
4. 僅剩的一個元素必定是最小元素，無須排序，因此陣列排序完成。

圖 11-5　泡沫排序流程

示例程式碼如下：

```
// === File: bubble_sort.rs ===

/* 泡沫排序 */
fn bubble_sort(nums: &mut [i32]) {
    // 外迴圈：未排序區間為 [0, i]
    for i in (1..nums.len()).rev() {
        // 內迴圈：將未排序區間 [0, i] 中的最大元素交換至該區間的最右端
        for j in 0..i {
            if nums[j] > nums[j + 1] {
                // 交換 nums[j] 與 nums[j + 1]
                nums.swap(j, j + 1);
            }
        }
    }
}
```

11.3.2　效率最佳化

我們發現，如果某輪“冒泡”中沒有執行任何交換操作，說明陣列已經完成排序，可直接返回結果。因此，可以增加一個標誌位 `flag` 來監測這種情況，一旦出現就立即返回。

經過最佳化，泡沫排序的最差時間複雜度和平均時間複雜度仍為 $O(n^2)$；但當輸入陣列完全有序時，可達到最佳時間複雜度 $O(n)$。

```rust
// === File: bubble_sort.rs ===

/* 泡沫排序（標誌最佳化） */
fn bubble_sort_with_flag(nums: &mut [i32]) {
    // 外迴圈：未排序區間為 [0, i]
    for i in (1..nums.len()).rev() {
        let mut flag = false; // 初始化標誌位
        // 內迴圈：將未排序區間 [0, i] 中的最大元素交換至該區間的最右端
        for j in 0..i {
            if nums[j] > nums[j + 1] {
                // 交換 nums[j] 與 nums[j + 1]
                nums.swap(j, j + 1);
                flag = true; // 記錄交換元素
            }
        }
        if !flag {
            break; // 此輪“冒泡”未交換任何元素，直接跳出
        };
    }
}
```

11.3.3　演算法特性

- **時間複雜度為 $O(n^2)$、自適應排序**：各輪“冒泡”走訪的陣列長度依次為 $n-1$、$n-2$、…、2、1，總和為 $(n-1)n/2$。在引入 `flag` 最佳化後，最佳時間複雜度可達到 $O(n)$。
- **空間複雜度為 $O(1)$、原地排序**：指標 i 和 j 使用常數大小的額外空間。
- **穩定排序**：由於在“冒泡”中遇到相等元素不交換。

11.4　插入排序

插入排序（insertion sort）是一種簡單的排序演算法，它的工作原理與手動整理一副牌的過程非常相似。

具體來說，我們在未排序區間選擇一個基準元素，將該元素與其左側已排序區間的元素逐一比較大小，並將該元素插入到正確的位置。

圖 11-6 展示了陣列插入元素的操作流程。設基準元素為 `base`，我們需要將從目標索引到 `base` 之間的所有元素向右移動一位，然後將 `base` 賦值給目標索引。

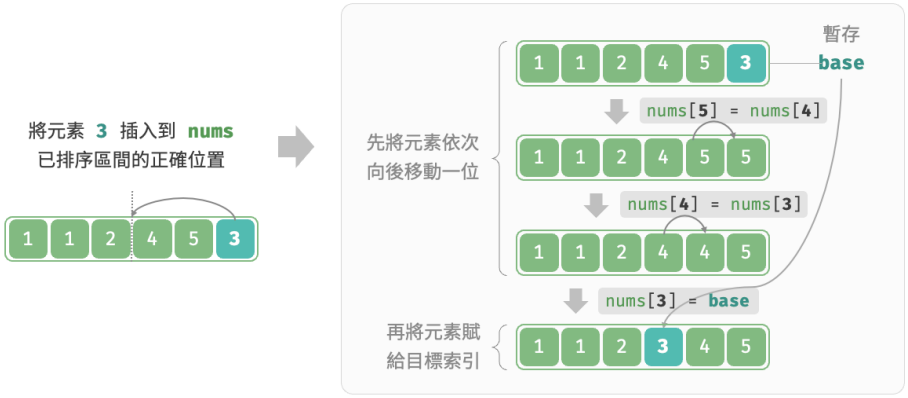


圖 11-6　單次插入操作

11.4.1　演算法流程

插入排序的整體流程如圖 11-7 所示。

1. 初始狀態下，陣列的第 1 個元素已完成排序。
2. 選取陣列的第 2 個元素作為 `base`，將其插入到正確位置後，**陣列的前 2 個元素已排序**。
3. 選取第 3 個元素作為 `base`，將其插入到正確位置後，**陣列的前 3 個元素已排序**。
4. 以此類推，在最後一輪中，選取最後一個元素作為 `base`，將其插入到正確位置後，**所有元素均已排序**。

圖 11-7　插入排序流程

示例程式碼如下：

```rust
// === File: insertion_sort.rs ===

/* 插入排序 */
fn insertion_sort(nums: &mut [i32]) {
    // 外迴圈：已排序區間為 [0, i-1]
    for i in 1..nums.len() {
        let (base, mut j) = (nums[i], (i - 1) as i32);
        // 內迴圈：將 base 插入到已排序區間 [0, i-1] 中的正確位置
        while j >= 0 && nums[j as usize] > base {
            nums[(j + 1) as usize] = nums[j as usize]; // 將 nums[j] 向右移動一位
            j -= 1;
        }
        nums[(j + 1) as usize] = base; // 將 base 賦值到正確位置
    }
}
```

11.4.2　演算法特性

- **時間複雜度為 $O(n^2)$、自適應排序**：在最差情況下，每次插入操作分別需要迴圈 $n-1$、$n-2$、…、2、1 次，求和得到 $(n-1)n/2$ ，因此時間複雜度為 $O(n^2)$ 。在遇到有序資料時，插入操作會提前終止。當輸入陣列完全有序時，插入排序達到最佳時間複雜度 $O(n)$ 。
- **空間複雜度為 $O(1)$、原地排序**：指標 i 和 j 使用常數大小的額外空間。
- **穩定排序**：在插入操作過程中，我們會將元素插入到相等元素的右側，不會改變它們的順序。

11.4.3　插入排序的優勢

插入排序的時間複雜度為 $O(n^2)$ ，而我們即將學習的快速排序的時間複雜度為 $O(n \log n)$ 。儘管插入排序的時間複雜度更高，**但在資料量較小的情況下，插入排序通常更快**。

這個結論與線性查詢和二分搜尋的適用情況的結論類似。快速排序這類 $O(n \log n)$ 的演算法屬於基於分治策略的排序演算法，往往包含更多單元計算操作。而在資料量較小時，n^2 和 $n \log n$ 的數值比較接近，複雜度不佔主導地位，每輪中的單元操作數量起到決定性作用。

實際上，許多程式語言（例如 Java）的內建排序函式採用了插入排序，大致思路為：對於長陣列，採用基於分治策略的排序演算法，例如快速排序；對於短陣列，直接使用插入排序。

雖然泡沫排序、選擇排序和插入排序的時間複雜度都為 $O(n^2)$ ，但在實際情況中，**插入排序的使用頻率顯著高於泡沫排序和選擇排序**，主要有以下原因。

- 泡沫排序基於元素交換實現，需要藉助一個臨時變數，共涉及 3 個單元操作；插入排序基於元素賦值實現，僅需 1 個單元操作。因此，**泡沫排序的計算開銷通常比插入排序更高**。
- 選擇排序在任何情況下的時間複雜度都為 $O(n^2)$ 。**如果給定一組部分有序的資料，插入排序通常比選擇排序效率更高**。
- 選擇排序不穩定，無法應用於多級排序。

11.5　快速排序

快速排序（quick sort）是一種基於分治策略的排序演算法，執行高效，應用廣泛。

快速排序的核心操作是“哨兵劃分”，其目標是：選擇陣列中的某個元素作為“基準數”，將所有小於基準數的元素移到其左側，而大於基準數的元素移到其右側。具體來說，哨兵劃分的流程如圖 11-8 所示。

1. 選取陣列最左端元素作為基準數，初始化兩個指標 `i` 和 `j` 分別指向陣列的兩端。
2. 設定一個迴圈，在每輪中使用 `i`（`j`）分別尋找第一個比基準數大（小）的元素，然後交換這兩個元素。
3. 迴圈執行步驟 `2.`，直到 `i` 和 `j` 相遇時停止，最後將基準數交換至兩個子陣列的分界線。

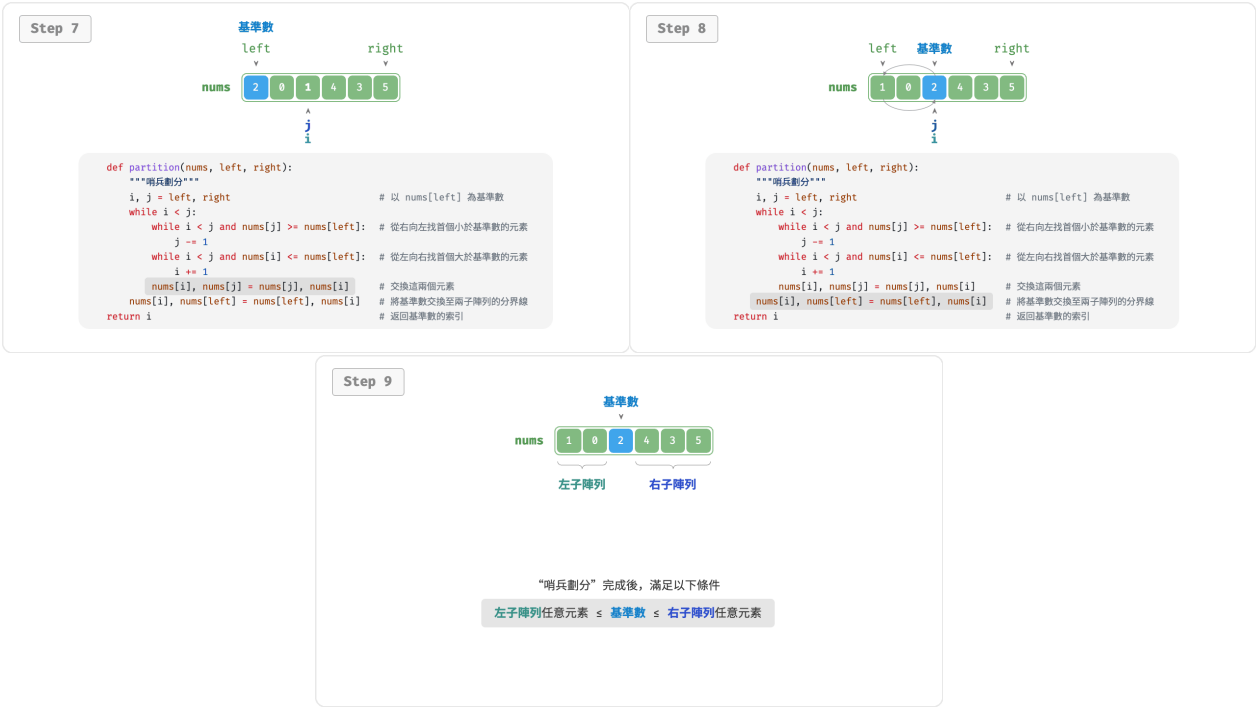


圖 11-8　哨兵劃分步驟

哨兵劃分完成後，原陣列被劃分成三部分：左子陣列、基準數、右子陣列，且滿足“左子陣列任意元素 $\leq$ 基準數 $\leq$ 右子陣列任意元素”。因此，我們接下來只需對這兩個子陣列進行排序。

快速排序的分治策略

哨兵劃分的實質是將一個較長陣列的排序問題簡化為兩個較短陣列的排序問題。

```rust
// === File: quick_sort.rs ===

/* 哨兵劃分 */
fn partition(nums: &mut [i32], left: usize, right: usize) -> usize {
    // 以 nums[left] 為基準數
    let (mut i, mut j) = (left, right);
    while i < j {
        while i < j && nums[j] >= nums[left] {
            j -= 1; // 從右向左找首個小於基準數的元素
        }
        while i < j && nums[i] <= nums[left] {
            i += 1; // 從左向右找首個大於基準數的元素
        }
        nums.swap(i, j); // 交換這兩個元素
    }
    nums.swap(i, left); // 將基準數交換至兩子陣列的分界線
    i // 返回基準數的索引
}
```

11.5.1　演算法流程

快速排序的整體流程如圖 11-9 所示。

1. 首先，對原陣列執行一次“哨兵劃分”，得到未排序的左子陣列和右子陣列。
2. 然後，對左子陣列和右子陣列分別遞迴執行“哨兵劃分”。
3. 持續遞迴，直至子陣列長度為 1 時終止，從而完成整個陣列的排序。

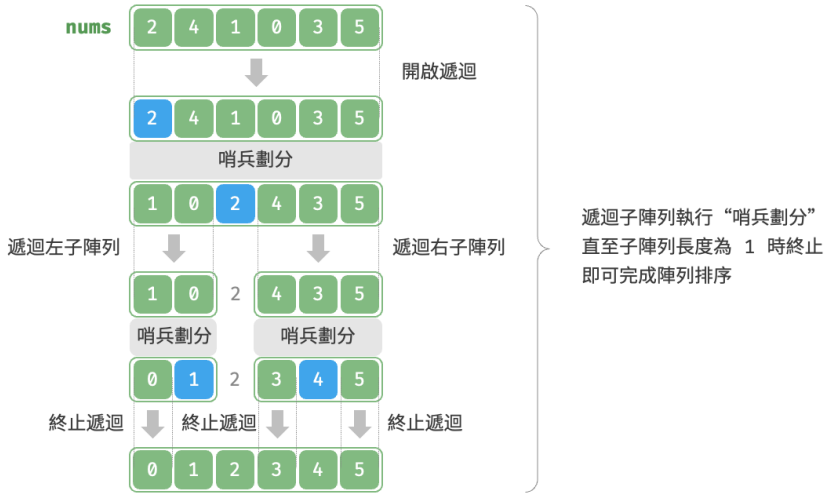


圖 11-9　快速排序流程

```rust
// === File: quick_sort.rs ===

/* 快速排序 */
pub fn quick_sort(left: i32, right: i32, nums: &mut [i32]) {
    // 子陣列長度為 1 時終止遞迴
    if left >= right {
        return;
    }
    // 哨兵劃分
    let pivot = Self::partition(nums, left as usize, right as usize) as i32;
    // 遞迴左子陣列、右子陣列
    Self::quick_sort(left, pivot - 1, nums);
    Self::quick_sort(pivot + 1, right, nums);
}
```

11.5.2　演算法特性

- **時間複雜度為 $O(n \log n)$、非自適應排序**：在平均情況下，哨兵劃分的遞迴層數為 $\log n$，每層中的總迴圈數為 n，總體使用 $O(n \log n)$ 時間。在最差情況下，每輪哨兵劃分操作都將長度為 n 的陣列劃

分為長度為 0 和 $n-1$ 的兩個子陣列，此時遞迴層數達到 n，每層中的迴圈數為 n，總體使用 $O(n^2)$ 時間。

- **空間複雜度為 $O(n)$、原地排序**：在輸入陣列完全倒序的情況下，達到最差遞迴深度 n，使用 $O(n)$ 堆疊幀空間。排序操作是在原陣列上進行的，未藉助額外陣列。
- **非穩定排序**：在哨兵劃分的最後一步，基準數可能會被交換至相等元素的右側。

11.5.3　快速排序為什麼快

從名稱上就能看出，快速排序在效率方面應該具有一定的優勢。儘管快速排序的平均時間複雜度與“合併排序”和“堆積排序”相同，但通常快速排序的效率更高，主要有以下原因。

- **出現最差情況的機率很低**：雖然快速排序的最差時間複雜度為 $O(n^2)$，沒有合併排序穩定，但在絕大多數情況下，快速排序能在 $O(n \log n)$ 的時間複雜度下執行。
- **快取使用效率高**：在執行哨兵劃分操作時，系統可將整個子陣列載入到快取，因此訪問元素的效率較高。而像“堆積排序”這類演算法需要跳躍式訪問元素，從而缺乏這一特性。
- **複雜度的常數係數小**：在上述三種演算法中，快速排序的比較、賦值、交換等操作的總數量最少。這與“插入排序”比“泡沫排序”更快的原因類似。

11.5.4　基準數最佳化

快速排序在某些輸入下的時間效率可能降低。舉一個極端例子，假設輸入陣列是完全倒序的，由於我們選擇最左端元素作為基準數，那麼在哨兵劃分完成後，基準數被交換至陣列最右端，導致左子陣列長度為 $n-1$、右子陣列長度為 0。如此遞迴下去，每輪哨兵劃分後都有一個子陣列的長度為 0，分治策略失效，快速排序退化為“泡沫排序”的近似形式。

為了儘量避免這種情況發生，**我們可以最佳化哨兵劃分中的基準數的選取策略**。例如，我們可以隨機選取一個元素作為基準數。然而，如果運氣不佳，每次都選到不理想的基準數，效率仍然不盡如人意。

需要注意的是，程式語言通常生成的是“偽隨機數”。如果我們針對偽隨機數序列構建一個特定的測試樣例，那麼快速排序的效率仍然可能劣化。

為了進一步改進，我們可以在陣列中選取三個候選元素（通常為陣列的首、尾、中點元素），**並將這三個候選元素的中位數作為基準數**。這樣一來，基準數“既不太小也不太大”的機率將大幅提升。當然，我們還可以選取更多候選元素，以進一步提高演算法的穩健性。採用這種方法後，時間複雜度劣化至 $O(n^2)$ 的機率大大降低。

示例程式碼如下：

```rust
// === File: quick_sort.rs ===

/* 選取三個候選元素的中位數 */
fn median_three(nums: &mut [i32], left: usize, mid: usize, right: usize) -> usize {
    let (l, m, r) = (nums[left], nums[mid], nums[right]);
    if (l <= m && m <= r) || (r <= m && m <= l) {
        return mid; // m 在 l 和 r 之間
    }
```

```
    if (m <= l && l <= r) || (r <= l && l <= m) {
        return left; // l 在 m 和 r 之間
    }
    right
}

/* 哨兵劃分（三數取中值） */
fn partition(nums: &mut [i32], left: usize, right: usize) -> usize {
    // 選取三個候選元素的中位數
    let med = Self::median_three(nums, left, (left + right) / 2, right);
    // 將中位數交換至陣列最左端
    nums.swap(left, med);
    // 以 nums[left] 為基準數
    let (mut i, mut j) = (left, right);
    while i < j {
        while i < j && nums[j] >= nums[left] {
            j -= 1; // 從右向左找首個小於基準數的元素
        }
        while i < j && nums[i] <= nums[left] {
            i += 1; // 從左向右找首個大於基準數的元素
        }
        nums.swap(i, j); // 交換這兩個元素
    }
    nums.swap(i, left); // 將基準數交換至兩子陣列的分界線
    i // 返回基準數的索引
}
```

11.5.5　尾遞迴最佳化

在某些輸入下，快速排序可能佔用空間較多。以完全有序的輸入陣列為例，設遞迴中的子陣列長度為 m ，每輪哨兵劃分操作都將產生長度為 0 的左子陣列和長度為 $m - 1$ 的右子陣列，這意味著每一層遞迴呼叫減少的問題規模非常小（只減少一個元素），遞迴樹的高度會達到 $n - 1$ ，此時需要佔用 $O(n)$ 大小的堆疊幀空間。

為了防止堆疊幀空間的累積，我們可以在每輪哨兵排序完成後，比較兩個子陣列的長度，**僅對較短的子陣列進行遞迴**。由於較短子陣列的長度不會超過 $n/2$ ，因此這種方法能確保遞迴深度不超過 $\log n$ ，從而將最差空間複雜度最佳化至 $O(\log n)$ 。程式碼如下所示：

```
// === File: quick_sort.rs ===

/* 快速排序（尾遞迴最佳化） */
pub fn quick_sort(mut left: i32, mut right: i32, nums: &mut [i32]) {
    // 子陣列長度為 1 時終止
    while left < right {
        // 哨兵劃分操作
        let pivot = Self::partition(nums, left as usize, right as usize) as i32;
```

```
        // 對兩個子陣列中較短的那個執行快速排序
        if pivot - left < right - pivot {
            Self::quick_sort(left, pivot - 1, nums); // 遞迴排序左子陣列
            left = pivot + 1; // 剩餘未排序區間為 [pivot + 1, right]
        } else {
            Self::quick_sort(pivot + 1, right, nums); // 遞迴排序右子陣列
            right = pivot - 1; // 剩餘未排序區間為 [left, pivot - 1]
        }
    }
}
```

11.6　合併排序

合併排序（merge sort）是一種基於分治策略的排序演算法，包含圖 11-10 所示的“劃分”和“合併”階段。

1. **劃分階段**：透過遞迴不斷地將陣列從中點處分開，將長陣列的排序問題轉換為短陣列的排序問題。
2. **合併階段**：當子陣列長度為 1 時終止劃分，開始合併，持續地將左右兩個較短的有序陣列合併為一個較長的有序陣列，直至結束。

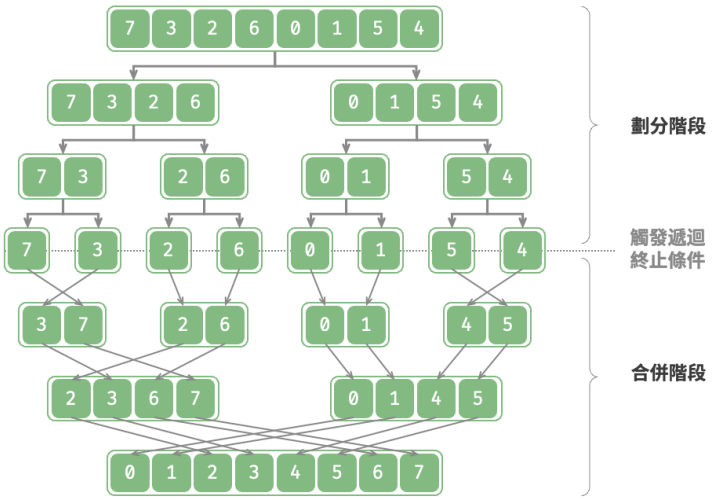


圖 11-10　合併排序的劃分與合併階段

11.6.1　演算法流程

如圖 11-11 所示，“劃分階段”從頂至底遞迴地將陣列從中點切分為兩個子陣列。

1. 計算陣列中點 `mid`，遞迴劃分左子陣列（區間 `[left, mid]`）和右子陣列（區間 `[mid + 1, right]`）。
2. 遞迴執行步驟 `1.`，直至子陣列區間長度為 1 時終止。

“合併階段”從底至頂地將左子陣列和右子陣列合併為一個有序陣列。需要注意的是，從長度為 1 的子陣列開始合併，合併階段中的每個子陣列都是有序的。

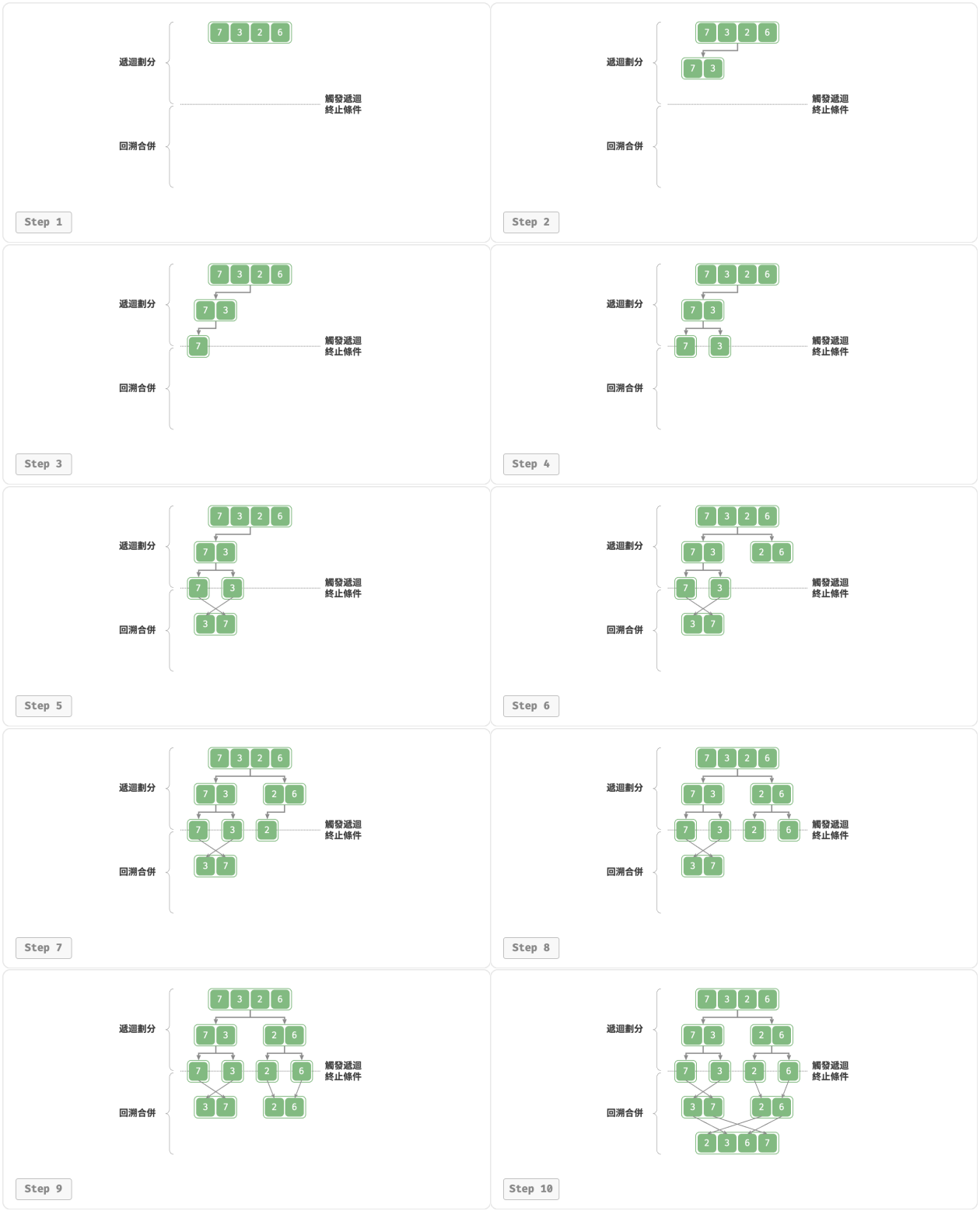


圖 11-11　合併排序步驟

觀察發現，合併排序與二元樹後序走訪的遞迴順序是一致的。

- **後序走訪**：先遞迴左子樹，再遞迴右子樹，最後處理根節點。
- **合併排序**：先遞迴左子陣列，再遞迴右子陣列，最後處理合併。

合併排序的實現如以下程式碼所示。請注意，`nums` 的待合併區間為 `[left, right]` ，而 `tmp` 的對應區間為 `[0, right - left]` 。

```
// === File: merge_sort.rs ===

/* 合併左子陣列和右子陣列 */
fn merge(nums: &mut [i32], left: usize, mid: usize, right: usize) {
    // 左子陣列區間為 [left, mid]，右子陣列區間為 [mid+1, right]
    // 建立一個臨時陣列 tmp ，用於存放合併後的結果
    let tmp_size = right - left + 1;
    let mut tmp = vec![0; tmp_size];
    // 初始化左子陣列和右子陣列的起始索引
    let (mut i, mut j, mut k) = (left, mid + 1, 0);
    // 當左右子陣列都還有元素時，進行比較並將較小的元素複製到臨時陣列中
    while i <= mid && j <= right {
        if nums[i] <= nums[j] {
            tmp[k] = nums[i];
            i += 1;
        } else {
            tmp[k] = nums[j];
            j += 1;
        }
        k += 1;
    }
    // 將左子陣列和右子陣列的剩餘元素複製到臨時陣列中
    while i <= mid {
        tmp[k] = nums[i];
        k += 1;
        i += 1;
    }
    while j <= right {
        tmp[k] = nums[j];
        k += 1;
        j += 1;
    }
    // 將臨時陣列 tmp 中的元素複製回原陣列 nums 的對應區間
    for k in 0..tmp_size {
        nums[left + k] = tmp[k];
    }
}

/* 合併排序 */
fn merge_sort(nums: &mut [i32], left: usize, right: usize) {
```

```
    // 終止條件
    if left >= right {
        return; // 當子陣列長度為 1 時終止遞迴
    }

    // 劃分階段
    let mid = left + (right - left) / 2; // 計算中點
    merge_sort(nums, left, mid); // 遞迴左子陣列
    merge_sort(nums, mid + 1, right); // 遞迴右子陣列

    // 合併階段
    merge(nums, left, mid, right);
}
```

11.6.2　演算法特性

- **時間複雜度為 $O(n \log n)$、非自適應排序**：劃分產生高度為 $\log n$ 的遞迴樹，每層合併的總操作數量為 n，因此總體時間複雜度為 $O(n \log n)$。
- **空間複雜度為 $O(n)$、非原地排序**：遞迴深度為 $\log n$，使用 $O(\log n)$ 大小的堆疊幀空間。合併操作需要藉助輔助陣列實現，使用 $O(n)$ 大小的額外空間。
- **穩定排序**：在合併過程中，相等元素的次序保持不變。

11.6.3　鏈結串列排序

對於鏈結串列，合併排序相較於其他排序演算法具有顯著優勢，**可以將鏈結串列排序任務的空間複雜度最佳化至 $O(1)$**。

- **劃分階段**：可以使用“迭代”替代“遞迴”來實現鏈結串列劃分工作，從而省去遞迴使用的堆疊幀空間。
- **合併階段**：在鏈結串列中，節點增刪操作僅需改變引用（指標）即可實現，因此合併階段（將兩個短有序鏈結串列合併為一個長有序鏈結串列）無須建立額外鏈結串列。

具體實現細節比較複雜，有興趣的讀者可以查閱相關資料進行學習。

11.7　堆積排序

> **Tip**
> 閱讀本節前，請確保已學完“堆積“章節。

堆積排序（heap sort）是一種基於堆積資料結構實現的高效排序演算法。我們可以利用已經學過的“建堆積操作”和“元素出堆積操作”實現堆積排序。

1. 輸入陣列並建立小頂堆積，此時最小元素位於堆積頂。
2. 不斷執行出堆積操作，依次記錄出堆積元素，即可得到從小到大排序的序列。

以上方法雖然可行，但需要藉助一個額外陣列來儲存彈出的元素，比較浪費空間。在實際中，我們通常使用一種更加優雅的實現方式。

11.7.1　演算法流程

設陣列的長度為 n ，堆積排序的流程如圖 11-12 所示。

1. 輸入陣列並建立大頂堆積。完成後，最大元素位於堆積頂。
2. 將堆積頂元素（第一個元素）與堆積底元素（最後一個元素）交換。完成交換後，堆積的長度減 1 ，已排序元素數量加 1 。
3. 從堆積頂元素開始，從頂到底執行堆積化操作（sift down）。完成堆積化後，堆積的性質得到修復。
4. 迴圈執行第 `2.` 步和第 `3.` 步。迴圈 $n - 1$ 輪後，即可完成陣列排序。

> **Tip**
> 實際上，元素出堆積操作中也包含第 `2.` 步和第 `3.` 步，只是多了一個彈出元素的步驟。

圖 11-12　堆積排序步驟

在程式碼實現中，我們使用了與“堆積”章節相同的從頂至底堆積化 `sift_down()` 函式。值得注意的是，由於堆積的長度會隨著提取最大元素而減小，因此我們需要給 `sift_down()` 函式新增一個長度參數 n ，用於指定堆積的當前有效長度。程式碼如下所示：

```
// === File: heap_sort.rs ===

/* 堆積的長度為 n ，從節點 i 開始，從頂至底堆積化 */
fn sift_down(nums: &mut [i32], n: usize, mut i: usize) {
    loop {
        // 判斷節點 i, l, r 中值最大的節點，記為 ma
        let l = 2 * i + 1;
        let r = 2 * i + 2;
        let mut ma = i;
        if l < n && nums[l] > nums[ma] {
            ma = l;
```

```
        }
        if r < n && nums[r] > nums[ma] {
            ma = r;
        }
        // 若節點 i 最大或索引 l, r 越界，則無須繼續堆積化，跳出
        if ma == i {
            break;
        }
        // 交換兩節點
        nums.swap(i, ma);
        // 迴圈向下堆積化
        i = ma;
    }
}

/* 堆積排序 */
fn heap_sort(nums: &mut [i32]) {
    // 建堆積操作：堆積化除葉節點以外的其他所有節點
    for i in (0..nums.len() / 2).rev() {
        sift_down(nums, nums.len(), i);
    }
    // 從堆積中提取最大元素，迴圈 n-1 輪
    for i in (1..nums.len()).rev() {
        // 交換根節點與最右葉節點（交換首元素與尾元素）
        nums.swap(0, i);
        // 以根節點為起點，從頂至底進行堆積化
        sift_down(nums, i, 0);
    }
}
```

11.7.2　演算法特性

- **時間複雜度為 $O(n \log n)$、非自適應排序**：建堆積操作使用 $O(n)$ 時間。從堆積中提取最大元素的時間複雜度為 $O(\log n)$ ，共迴圈 $n - 1$ 輪。
- **空間複雜度為 $O(1)$、原地排序**：幾個指標變數使用 $O(1)$ 空間。元素交換和堆積化操作都是在原陣列上進行的。
- **非穩定排序**：在交換堆積頂元素和堆積底元素時，相等元素的相對位置可能發生變化。

11.8　桶排序

前述幾種排序演算法都屬於“基於比較的排序演算法”，它們透過比較元素間的大小來實現排序。此類排序演算法的時間複雜度無法超越 $O(n \log n)$ 。接下來，我們將探討幾種“非比較排序演算法”，它們的時間複雜度可以達到線性階。

桶排序（bucket sort）是分治策略的一個典型應用。它透過設定一些具有大小順序的桶，每個桶對應一個數據範圍，將資料平均分配到各個桶中；然後，在每個桶內部分別執行排序；最終按照桶的順序將所有資料合併。

11.8.1　演算法流程

考慮一個長度為 n 的陣列，其元素是範圍 $[0, 1)$ 內的浮點數。桶排序的流程如圖 11-13 所示。

1. 初始化 k 個桶，將 n 個元素分配到 k 個桶中。
2. 對每個桶分別執行排序（這裡採用程式語言的內建排序函式）。
3. 按照桶從小到大的順序合併結果。

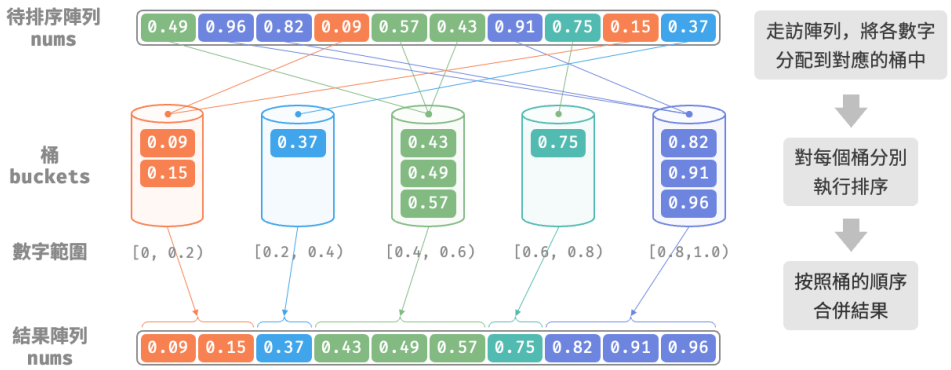


圖 11-13　桶排序演算法流程

程式碼如下所示：

```
// === File: bucket_sort.rs ===

/* 桶排序 */
fn bucket_sort(nums: &mut [f64]) {
    // 初始化 k = n/2 個桶，預期向每個桶分配 2 個元素
    let k = nums.len() / 2;
    let mut buckets = vec![vec![]; k];
    // 1. 將陣列元素分配到各個桶中
    for &num in nums.iter() {
        // 輸入資料範圍為 [0, 1)，使用 num * k 對映到索引範圍 [0, k-1]
        let i = (num * k as f64) as usize;
        // 將 num 新增進桶 i
        buckets[i].push(num);
    }
    // 2. 對各個桶執行排序
    for bucket in &mut buckets {
        // 使用內建排序函式，也可以替換成其他排序演算法
```

```
        bucket.sort_by(|a, b| a.partial_cmp(b).unwrap());
    }
    // 3. 走訪桶合併結果
    let mut i = 0;
    for bucket in buckets.iter() {
        for &num in bucket.iter() {
            nums[i] = num;
            i += 1;
        }
    }
}
```

11.8.2　演算法特性

桶排序適用於處理體量很大的資料。例如，輸入資料包含 100 萬個元素，由於空間限制，系統記憶體無法一次性載入所有資料。此時，可以將資料分成 1000 個桶，然後分別對每個桶進行排序，最後將結果合併。

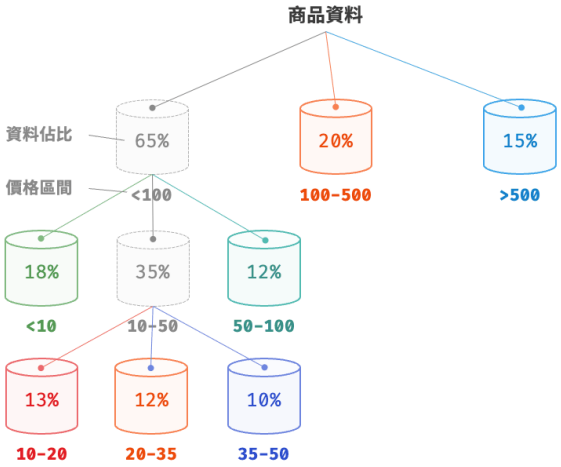
- **時間複雜度為 $O(n + k)$**：假設元素在各個桶內平均分佈，那麼每個桶內的元素數量為 $\frac{n}{k}$ 。假設排序單個桶使用 $O(\frac{n}{k} \log \frac{n}{k})$ 時間，則排序所有桶使用 $O(n \log \frac{n}{k})$ 時間。**當桶數量 k 比較大時，時間複雜度則趨向於 $O(n)$** 。合併結果時需要走訪所有桶和元素，花費 $O(n + k)$ 時間。在最差情況下，所有資料被分配到一個桶中，且排序該桶使用 $O(n^2)$ 時間。
- **空間複雜度為 $O(n + k)$、非原地排序**：需要藉助 k 個桶和總共 n 個元素的額外空間。
- 桶排序是否穩定取決於排序桶內元素的演算法是否穩定。

11.8.3　如何實現平均分配

桶排序的時間複雜度理論上可以達到 $O(n)$ ，**關鍵在於將元素均勻分配到各個桶中**，因為實際資料往往不是均勻分佈的。例如，我們想要將淘寶上的所有商品按價格範圍平均分配到 10 個桶中，但商品價格分佈不均，低於 100 元的非常多，高於 1000 元的非常少。若將價格區間平均劃分為 10 個，各個桶中的商品數量差距會非常大。

為實現平均分配，我們可以先設定一條大致的分界線，將資料粗略地分到 3 個桶中。**分配完畢後，再將商品較多的桶繼續劃分為 3 個桶，直至所有桶中的元素數量大致相等**。

如圖 11-14 所示，這種方法本質上是建立一棵遞迴樹，目標是讓葉節點的值儘可能平均。當然，不一定要每輪將資料劃分為 3 個桶，具體劃分方式可根據資料特點靈活選擇。



資料劃分結果

價格區間	資料佔比
<10	18%
10-20	13%
20-35	12%
35-50	10%
50-100	12%
100-500	20%
>500	15%

圖 11-14　遞迴劃分桶

如果我們提前知道商品價格的機率分佈，**則可以根據資料機率分佈設定每個桶的價格分界線**。值得注意的是，資料分佈並不一定需要特意統計，也可以根據資料特點採用某種機率模型進行近似。

如圖 11-15 所示，我們假設商品價格服從正態分佈，這樣就可以合理地設定價格區間，從而將商品平均分配到各個桶中。

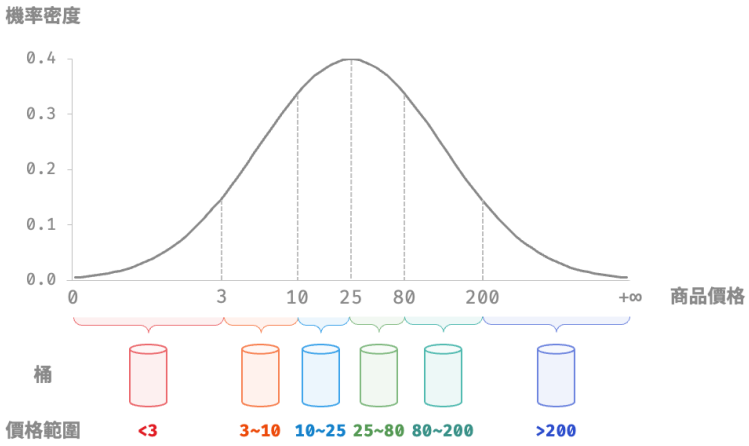


圖 11-15　根據機率分佈劃分桶

11.9　計數排序

計數排序（counting sort）透過統計元素數量來實現排序，通常應用於整數陣列。

11.9.1　簡單實現

先來看一個簡單的例子。給定一個長度為 n 的陣列 `nums`，其中的元素都是“非負整數”，計數排序的整體流程如圖 11-16 所示。

1. 走訪陣列，找出其中的最大數字，記為 m，然後建立一個長度為 $m + 1$ 的輔助陣列 `counter`。
2. **藉助 `counter` 統計 `nums` 中各數字的出現次數**，其中 `counter[num]` 對應數字 `num` 的出現次數。統計方法很簡單，只需走訪 `nums`（設當前數字為 `num`），每輪將 `counter[num]` 增加 1 即可。
3. **由於 `counter` 的各個索引天然有序，因此相當於所有數字已經排序好了**。接下來，我們走訪 `counter`，根據各數字出現次數從小到大的順序填入 `nums` 即可。

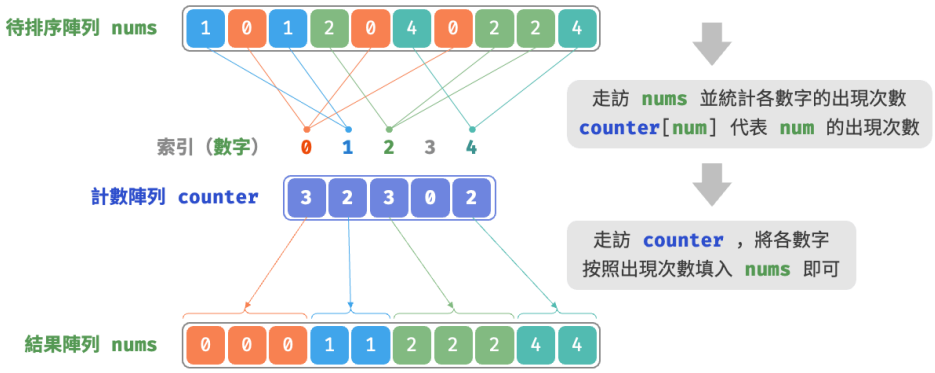


圖 11-16　計數排序流程

程式碼如下所示：

```rust
// === File: counting_sort.rs ===

/* 計數排序 */
// 簡單實現，無法用於排序物件
fn counting_sort_naive(nums: &mut [i32]) {
    // 1. 統計陣列最大元素 m
    let m = *nums.iter().max().unwrap();
    // 2. 統計各數字的出現次數
    // counter[num] 代表 num 的出現次數
    let mut counter = vec![0; m as usize + 1];
    for &num in nums.iter() {
        counter[num as usize] += 1;
    }
    // 3. 走訪 counter ，將各元素填入原陣列 nums
    let mut i = 0;
    for num in 0..m + 1 {
        for _ in 0..counter[num as usize] {
            nums[i] = num;
```

```
            i += 1;
        }
    }
}
```

計數排序與桶排序的關聯

從桶排序的角度看，我們可以將計數排序中的計數陣列 `counter` 的每個索引視為一個桶，將統計數量的過程看作將各個元素分配到對應的桶中。本質上，計數排序是桶排序在整型資料下的一個特例。

11.9.2　完整實現

細心的讀者可能發現了，**如果輸入資料是物件，上述步驟 `3.` 就失效了**。假設輸入資料是商品物件，我們想按照商品價格（類別的成員變數）對商品進行排序，而上述演算法只能給出價格的排序結果。

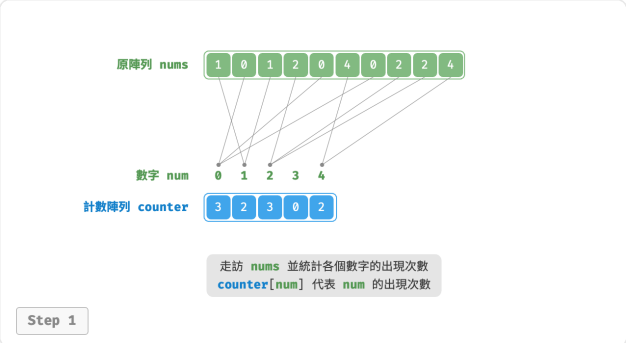
那麼如何才能得到原資料的排序結果呢？我們首先計算 `counter` 的“前綴和”。顧名思義，索引 `i` 處的前綴和 `prefix[i]` 等於陣列前 `i` 個元素之和：

$$\text{prefix}[i] = \sum_{j=0}^{i} \text{counter}[\text{j}]$$

前綴和具有明確的意義，`prefix[num] - 1` 代表元素 `num` 在結果陣列 `res` 中最後一次出現的索引。這個資訊非常關鍵，因為它告訴我們各個元素應該出現在結果陣列的哪個位置。接下來，我們倒序走訪原陣列 `nums` 的每個元素 `num`，在每輪迭代中執行以下兩步。

1. 將 `num` 填入陣列 `res` 的索引 `prefix[num] - 1` 處。
2. 令前綴和 `prefix[num]` 減小 1，從而得到下次放置 `num` 的索引。

走訪完成後，陣列 `res` 中就是排序好的結果，最後使用 `res` 覆蓋原陣列 `nums` 即可。圖 11-17 展示了完整的計數排序流程。



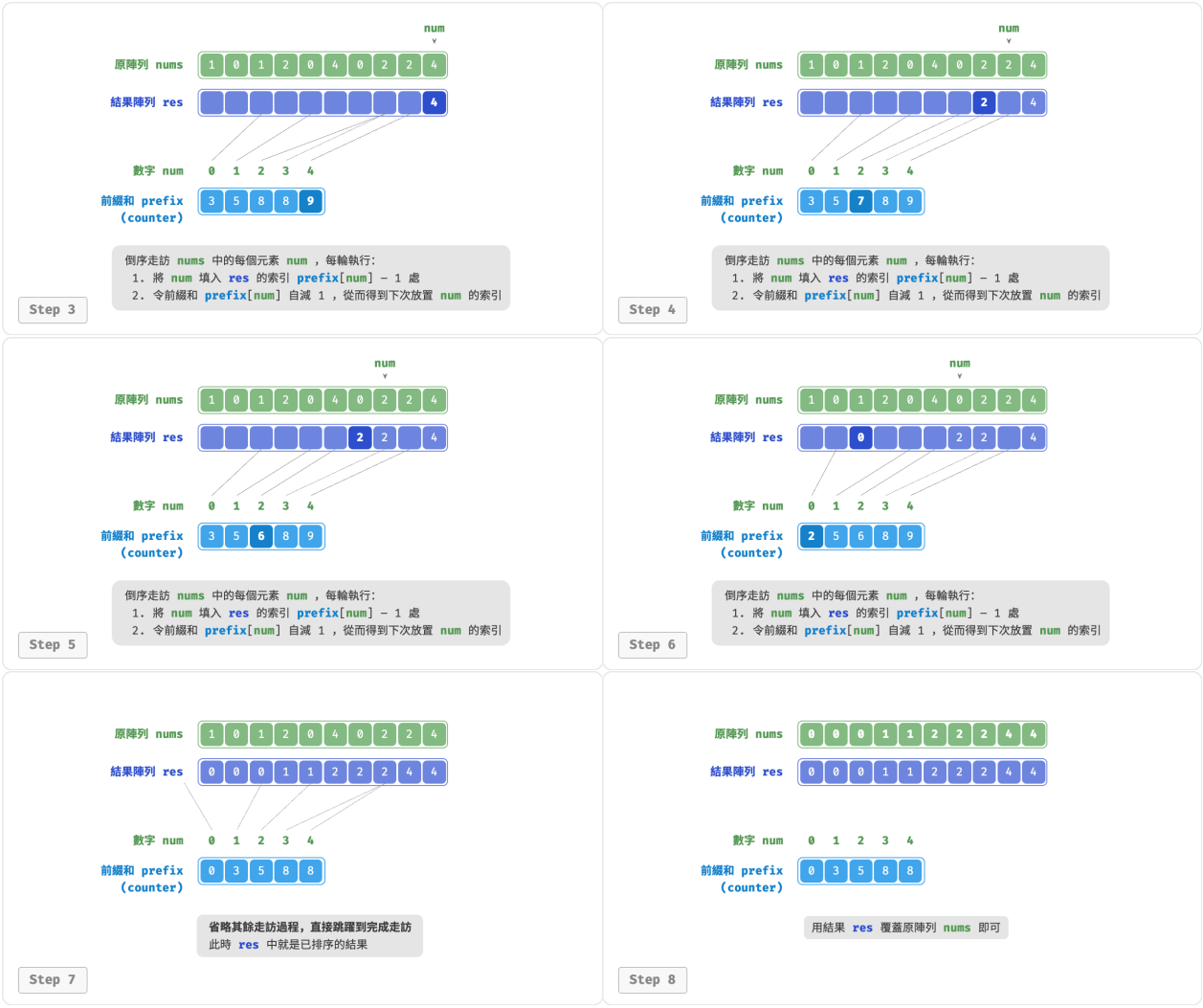


圖 11-17　計數排序步驟

計數排序的實現程式碼如下所示：

```rust
// === File: counting_sort.rs ===

/* 計數排序 */
// 完整實現，可排序物件，並且是穩定排序
fn counting_sort(nums: &mut [i32]) {
    // 1. 統計陣列最大元素 m
    let m = *nums.iter().max().unwrap() as usize;
    // 2. 統計各數字的出現次數
    // counter[num] 代表 num 的出現次數
    let mut counter = vec![0; m + 1];
    for &num in nums.iter() {
        counter[num as usize] += 1;
    }
```

```
    // 3. 求 counter 的前綴和，將“出現次數”轉換為“尾索引”
    // 即 counter[num]-1 是 num 在 res 中最後一次出現的索引
    for i in 0..m {
        counter[i + 1] += counter[i];
    }
    // 4. 倒序走訪 nums ，將各元素填入結果陣列 res
    // 初始化陣列 res 用於記錄結果
    let n = nums.len();
    let mut res = vec![0; n];
    for i in (0..n).rev() {
        let num = nums[i];
        res[counter[num as usize] - 1] = num; // 將 num 放置到對應索引處
        counter[num as usize] -= 1; // 令前綴和自減 1 ，得到下次放置 num 的索引
    }
    // 使用結果陣列 res 覆蓋原陣列 nums
    nums.copy_from_slice(&res)
}
```

11.9.3　演算法特性

- **時間複雜度為 $O(n + m)$、非自適應排序**：涉及走訪 `nums` 和走訪 `counter` ，都使用線性時間。一般情況下 $n \gg m$ ，時間複雜度趨於 $O(n)$ 。
- **空間複雜度為 $O(n + m)$、非原地排序**：藉助了長度分別為 n 和 m 的陣列 `res` 和 `counter` 。
- **穩定排序**：由於向 `res` 中填充元素的順序是“從右向左”的，因此倒序走訪 `nums` 可以避免改變相等元素之間的相對位置，從而實現穩定排序。實際上，正序走訪 `nums` 也可以得到正確的排序結果，但結果是非穩定的。

11.9.4　侷限性

看到這裡，你也許會覺得計數排序非常巧妙，僅透過統計數量就可以實現高效的排序。然而，使用計數排序的前置條件相對較為嚴格。

計數排序只適用於非負整數。若想將其用於其他型別的資料，需要確保這些資料可以轉換為非負整數，並且在轉換過程中不能改變各個元素之間的相對大小關係。例如，對於包含負數的整數陣列，可以先給所有數字加上一個常數，將全部數字轉化為正數，排序完成後再轉換回去。

計數排序適用於資料量大但資料範圍較小的情況。比如，在上述示例中 m 不能太大，否則會佔用過多空間。而當 $n \ll m$ 時，計數排序使用 $O(m)$ 時間，可能比 $O(n \log n)$ 的排序演算法還要慢。

11.10　基數排序

上一節介紹了計數排序，它適用於資料量 n 較大但資料範圍 m 較小的情況。假設我們需要對 $n = 10^6$ 個學號進行排序，而學號是一個 8 位數字，這意味著資料範圍 $m = 10^8$ 非常大，使用計數排序需要分配大量記憶體空間，而基數排序可以避免這種情況。

基數排序（radix sort）的核心思想與計數排序一致，也透過統計個數來實現排序。在此基礎上，基數排序利用數字各位之間的遞進關係，依次對每一位進行排序，從而得到最終的排序結果。

11.10.1　演算法流程

以學號資料為例，假設數字的最低位是第 1 位，最高位是第 8 位，基數排序的流程如圖 11-18 所示。

1. 初始化位數 $k = 1$ 。
2. 對學號的第 k 位執行“計數排序”。完成後，資料會根據第 k 位從小到大排序。
3. 將 k 增加 1 ，然後返回步驟 `2.` 繼續迭代，直到所有位都排序完成後結束。

圖 11-18　基數排序演算法流程

下面剖析程式碼實現。對於一個 d 進位制的數字 x ，要獲取其第 k 位 x_k ，可以使用以下計算公式：

$$x_k = \lfloor \frac{x}{d^{k-1}} \rfloor \bmod d$$

其中 $\lfloor a \rfloor$ 表示對浮點數 a 向下取整，而 $\bmod d$ 表示對 d 取模（取餘）。對於學號資料，$d = 10$ 且 $k \in [1, 8]$ 。

此外，我們需要小幅改動計數排序程式碼，使之可以根據數字的第 k 位進行排序：

```rust
// === File: radix_sort.rs ===

/* 獲取元素 num 的第 k 位，其中 exp = 10^(k-1) */
fn digit(num: i32, exp: i32) -> usize {
    // 傳入 exp 而非 k 可以避免在此重複執行昂貴的次方計算
    return ((num / exp) % 10) as usize;
}
```

```rust
/* 計數排序（根據 nums 第 k 位排序） */
fn counting_sort_digit(nums: &mut [i32], exp: i32) {
    // 十進位制的位範圍為 0~9 ，因此需要長度為 10 的桶陣列
    let mut counter = [0; 10];
    let n = nums.len();
    // 統計 0~9 各數字的出現次數
    for i in 0..n {
        let d = digit(nums[i], exp); // 獲取 nums[i] 第 k 位，記為 d
        counter[d] += 1; // 統計數字 d 的出現次數
    }
    // 求前綴和，將“出現個數”轉換為“陣列索引”
    for i in 1..10 {
        counter[i] += counter[i - 1];
    }
    // 倒序走訪，根據桶內統計結果，將各元素填入 res
    let mut res = vec![0; n];
    for i in (0..n).rev() {
        let d = digit(nums[i], exp);
        let j = counter[d] - 1; // 獲取 d 在陣列中的索引 j
        res[j] = nums[i]; // 將當前元素填入索引 j
        counter[d] -= 1; // 將 d 的數量減 1
    }
    // 使用結果覆蓋原陣列 nums
    nums.copy_from_slice(&res);
}

/* 基數排序 */
fn radix_sort(nums: &mut [i32]) {
    // 獲取陣列的最大元素，用於判斷最大位數
    let m = *nums.into_iter().max().unwrap();
    // 按照從低位到高位的順序走訪
    let mut exp = 1;
    while exp <= m {
        counting_sort_digit(nums, exp);
        exp *= 10;
    }
}
```

為什麼從最低位開始排序？

在連續的排序輪次中，後一輪排序會覆蓋前一輪排序的結果。舉例來說，如果第一輪排序結果 $a < b$ ，而第二輪排序結果 $a > b$ ，那麼第二輪的結果將取代第一輪的結果。由於數字的高位優先順序高於低位，因此應該先排序低位再排序高位。

11.10.2　演算法特性

相較於計數排序，基數排序適用於數值範圍較大的情況，**但前提是資料必須可以表示為固定位數的格式，且位數不能過大**。例如，浮點數不適合使用基數排序，因為其位數 k 過大，可能導致時間複雜度 $O(nk) \gg O(n^2)$ 。

- **時間複雜度為 $O(nk)$、非自適應排序**：設資料量為 n、資料為 d 進位制、最大位數為 k ，則對某一位執行計數排序使用 $O(n + d)$ 時間，排序所有 k 位使用 $O((n + d)k)$ 時間。通常情況下，d 和 k 都相對較小，時間複雜度趨向 $O(n)$ 。
- **空間複雜度為 $O(n + d)$、非原地排序**：與計數排序相同，基數排序需要藉助長度為 n 和 d 的陣列 `res` 和 `counter` 。
- **穩定排序**：當計數排序穩定時，基數排序也穩定；當計數排序不穩定時，基數排序無法保證得到正確的排序結果。

11.11　小結

1.　重點回顧

- 泡沫排序透過交換相鄰元素來實現排序。透過新增一個標誌位來實現提前返回，我們可以將泡沫排序的最佳時間複雜度最佳化到 $O(n)$ 。
- 插入排序每輪將未排序區間內的元素插入到已排序區間的正確位置，從而完成排序。雖然插入排序的時間複雜度為 $O(n^2)$ ，但由於單元操作相對較少，因此在小資料量的排序任務中非常受歡迎。
- 快速排序基於哨兵劃分操作實現排序。在哨兵劃分中，有可能每次都選取到最差的基準數，導致時間複雜度劣化至 $O(n^2)$ 。引入中位數基準數或隨機基準數可以降低這種劣化的機率。尾遞迴方法可以有效地減少遞迴深度，將空間複雜度最佳化到 $O(\log n)$ 。
- 合併排序包括劃分和合並兩個階段，典型地體現了分治策略。在合併排序中，排序陣列需要建立輔助陣列，空間複雜度為 $O(n)$ ；然而排序鏈結串列的空間複雜度可以最佳化至 $O(1)$ 。
- 桶排序包含三個步驟：資料分桶、桶內排序和合並結果。它同樣體現了分治策略，適用於資料體量很大的情況。桶排序的關鍵在於對資料進行平均分配。
- 計數排序是桶排序的一個特例，它透過統計資料出現的次數來實現排序。計數排序適用於資料量大但資料範圍有限的情況，並且要求資料能夠轉換為正整數。
- 基數排序透過逐位排序來實現資料排序，要求資料能夠表示為固定位數的數字。
- 總的來說，我們希望找到一種排序演算法，具有高效率、穩定、原地以及自適應性等優點。然而，正如其他資料結構和演算法一樣，沒有一種排序演算法能夠同時滿足所有這些條件。在實際應用中，我們需要根據資料的特性來選擇合適的排序演算法。
- 圖 11-19 對比了主流排序演算法的效率、穩定性、就地性和自適應性等。

		時間複雜度			空間複雜度	穩定性	就地性	自適應性	基於比較
		最佳	平均	最差	最差				
走訪排序 $O(n^2)$	選擇排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	非穩定	原地	非自適應	比較
	泡沫排序	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	穩定	原地	自適應	比較
	插入排序	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	穩定	原地	自適應	比較
分治排序 $O(n \log n)$	快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	非穩定	原地	非自適應	比較
	合併排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	穩定	非原地	非自適應	比較
	堆積排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	非穩定	原地	非自適應	比較
線性排序 $O(n)$	桶排序	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n + k)$	穩定	非原地	非自適應	非比較
	計數排序	$O(n + m)$	$O(n + m)$	$O(n + m)$	$O(n + m)$	穩定	非原地	非自適應	非比較
	基數排序	$O(n k)$	$O(n k)$	$O(n k)$	$O(n + b)$	穩定	非原地	非自適應	非比較

差　中　優

n 為資料量大小
桶排序中，k 為桶數量
計數排序中，m 為資料範圍
基數排序中，k 為最大位數，資料為 b 進位制

圖 11-19　排序演算法對比

2. Q & A

Q：排序演算法穩定性在什麼情況下是必需的？

在現實中，我們有可能基於物件的某個屬性進行排序。例如，學生有姓名和身高兩個屬性，我們希望實現一個多級排序：先按照姓名進行排序，得到 `(A, 180) (B, 185) (C, 170) (D, 170)`；再對身高進行排序。由於排序演算法不穩定，因此可能得到 `(D, 170) (C, 170) (A, 180) (B, 185)`。

可以發現，學生 D 和 C 的位置發生了交換，姓名的有序性被破壞了，而這是我們不希望看到的。

Q：哨兵劃分中“從右往左查詢”與“從左往右查詢”的順序可以交換嗎？

不行，當我們以最左端元素為基準數時，必須先“從右往左查詢”再“從左往右查詢”。這個結論有些反直覺，我們來剖析一下原因。

哨兵劃分 `partition()` 的最後一步是交換 `nums[left]` 和 `nums[i]`。完成交換後，基準數左邊的元素都 `<=` 基準數，**這就要求最後一步交換前 `nums[left] >= nums[i]` 必須成立**。假設我們先“從左往右查詢”，那麼如果找不到比基準數更大的元素，**則會在 `i == j` 時跳出迴圈，此時可能 `nums[j] == nums[i] > nums[left]`**。也就是說，此時最後一步交換操作會把一個比基準數更大的元素交換至陣列最左端，導致哨兵劃分失敗。

舉個例子，給定陣列 `[0, 0, 0, 0, 1]`，如果先“從左向右查詢”，哨兵劃分後陣列為 `[1, 0, 0, 0, 0]`，這個結果是不正確的。

再深入思考一下，如果我們選擇 `nums[right]` 為基準數，那麼正好反過來，必須先“從左往右查詢”。

Q：關於尾遞迴最佳化，為什麼選短的陣列能保證遞迴深度不超過 $\log n$？

遞迴深度就是當前未返回的遞迴方法的數量。每輪哨兵劃分我們將原陣列劃分為兩個子陣列。在尾遞迴最佳化後，向下遞迴的子陣列長度最大為原陣列長度的一半。假設最差情況，一直為一半長度，那麼最終的遞迴深度就是 $\log n$。

回顧原始的快速排序，我們有可能會連續地遞迴長度較大的陣列，最差情況下為 n、$n - 1$、...、2、1，遞迴深度為 n。尾遞迴最佳化可以避免這種情況出現。

Q：當陣列中所有元素都相等時，快速排序的時間複雜度是 $O(n^2)$ 嗎？該如何處理這種退化情況？

是的。對於這種情況，可以考慮透過哨兵劃分將陣列劃分為三個部分：小於、等於、大於基準數。僅向下遞迴小於和大於的兩部分。在該方法下，輸入元素全部相等的陣列，僅一輪哨兵劃分即可完成排序。

Q：桶排序的最差時間複雜度為什麼是 $O(n^2)$ ？

最差情況下，所有元素被分至同一個桶中。如果我們採用一個 $O(n^2)$ 演算法來排序這些元素，則時間複雜度為 $O(n^2)$ 。

第 12 章　分治

Abstract

難題被逐層拆解，每一次的拆解都使它變得更為簡單。

分而治之揭示了一個重要的事實：從簡單做起，一切都不再複雜。

12.1　分治演算法

分治（divide and conquer），全稱分而治之，是一種非常重要且常見的演算法策略。分治通常基於遞迴實現，包括“分”和“治”兩個步驟。

1. **分（劃分階段）**：遞迴地將原問題分解為兩個或多個子問題，直至到達最小子問題時終止。
2. **治（合併階段）**：從已知解的最小子問題開始，從底至頂地將子問題的解進行合併，從而構建出原問題的解。

如圖 12-1 所示，“合併排序”是分治策略的典型應用之一。

1. **分**：遞迴地將原陣列（原問題）劃分為兩個子陣列（子問題），直到子陣列只剩一個元素（最小子問題）。
2. **治**：從底至頂地將有序的子陣列（子問題的解）進行合併，從而得到有序的原陣列（原問題的解）。

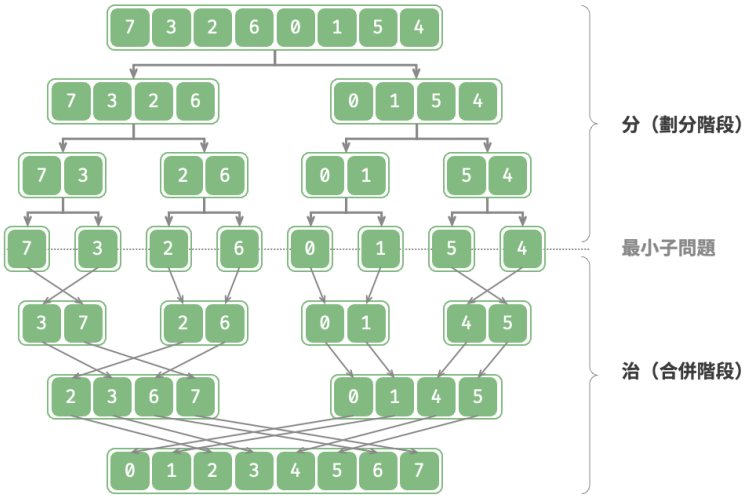


圖 12-1　合併排序的分治策略

12.1.1　如何判斷分治問題

一個問題是否適合使用分治解決，通常可以參考以下幾個判斷依據。

1. **問題可以分解**：原問題可以分解成規模更小、類似的子問題，以及能夠以相同方式遞迴地進行劃分。
2. **子問題是獨立的**：子問題之間沒有重疊，互不依賴，可以獨立解決。
3. **子問題的解可以合併**：原問題的解透過合併子問題的解得來。

顯然，合併排序滿足以上三個判斷依據。

1. **問題可以分解**：遞迴地將陣列（原問題）劃分為兩個子陣列（子問題）。
2. **子問題是獨立的**：每個子陣列都可以獨立地進行排序（子問題可以獨立進行求解）。
3. **子問題的解可以合併**：兩個有序子陣列（子問題的解）可以合併為一個有序陣列（原問題的解）。

12.1.2　透過分治提升效率

分治不僅可以有效地解決演算法問題，往往還可以提升演算法效率。在排序演算法中，快速排序、合併排序、堆積排序相較於選擇、冒泡、插入排序更快，就是因為它們應用了分治策略。

那麼，我們不禁發問：**為什麼分治可以提升演算法效率，其底層邏輯是什麼**？換句話說，將大問題分解為多個子問題、解決子問題、將子問題的解合併為原問題的解，這幾步的效率為什麼比直接解決原問題的效率更高？這個問題可以從操作數量和平行計算兩方面來討論。

1.　操作數量最佳化

以“泡沫排序”為例，其處理一個長度為 n 的陣列需要 $O(n^2)$ 時間。假設我們按照圖 12-2 所示的方式，將陣列從中點處分為兩個子陣列，則劃分需要 $O(n)$ 時間，排序每個子陣列需要 $O((n/2)^2)$ 時間，合併兩個子陣列需要 $O(n)$ 時間，總體時間複雜度為：

$$
O(n + (\frac{n}{2})^2 \times 2 + n) = O(\frac{n^2}{2} + 2n)
$$

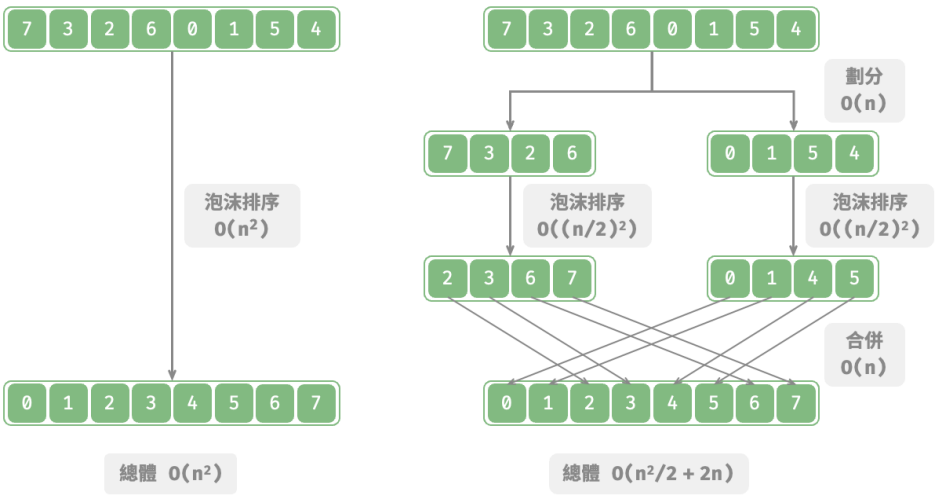


圖 12-2　劃分陣列前後的泡沫排序

接下來，我們計算以下不等式，其左邊和右邊分別為劃分前和劃分後的操作總數：

$$
\begin{aligned}
n^2 &> \frac{n^2}{2} + 2n \\
n^2 - \frac{n^2}{2} - 2n &> 0 \\
n(n-4) &> 0
\end{aligned}
$$

這意味著當 $n > 4$ 時，劃分後的操作數量更少，排序效率應該更高。請注意，劃分後的時間複雜度仍然是平方階 $O(n^2)$ ，只是複雜度中的常數項變小了。

進一步想，**如果我們把子陣列不斷地再從中點處劃分為兩個子陣列**，直至子陣列只剩一個元素時停止劃分呢？這種思路實際上就是“合併排序”，時間複雜度為 $O(n \log n)$ 。

再思考，**如果我們多設定幾個劃分點**，將原陣列平均劃分為 k 個子陣列呢？這種情況與“桶排序”非常類似，它非常適合排序海量資料，理論上時間複雜度可以達到 $O(n + k)$ 。

2. 平行計算最佳化

我們知道，分治生成的子問題是相互獨立的，**因此通常可以並行解決**。也就是說，分治不僅可以降低演算法的時間複雜度，**還有利於作業系統的並行最佳化**。

並行最佳化在多核或多處理器的環境中尤其有效，因為系統可以同時處理多個子問題，更加充分地利用計算資源，從而顯著減少總體的執行時間。

比如在圖 12-3 所示的“桶排序”中，我們將海量的資料平均分配到各個桶中，則可將所有桶的排序任務分散到各個計算單元，完成後再合併結果。

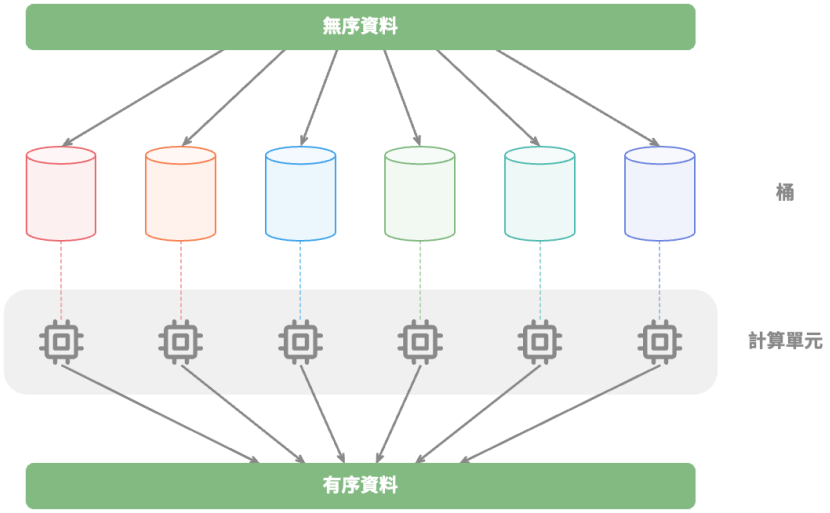


圖 12-3　桶排序的平行計算

12.1.3　分治常見應用

一方面，分治可以用來解決許多經典演算法問題。

- **尋找最近點對**：該演算法首先將點集分成兩部分，然後分別找出兩部分中的最近點對，最後找出跨越兩部分的最近點對。
- **大整數乘法**：例如 Karatsuba 演算法，它將大整數乘法分解為幾個較小的整數的乘法和加法。
- **矩陣乘法**：例如 Strassen 演算法，它將大矩陣乘法分解為多個小矩陣的乘法和加法。
- **河內塔問題**：河內塔問題可以透過遞迴解決，這是典型的分治策略應用。
- **求解逆序對**：在一個序列中，如果前面的數字大於後面的數字，那麼這兩個數字構成一個逆序對。求解逆序對問題可以利用分治的思想，藉助合併排序進行求解。

另一方面，分治在演算法和資料結構的設計中應用得非常廣泛。

- **二分搜尋**：二分搜尋是將有序陣列從中點索引處分為兩部分，然後根據目標值與中間元素值比較結果，決定排除哪一半區間，並在剩餘區間執行相同的二分操作。
- **合併排序**：本節開頭已介紹，不再贅述。
- **快速排序**：快速排序是選取一個基準值，然後把陣列分為兩個子陣列，一個子陣列的元素比基準值小，另一子陣列的元素比基準值大，再對這兩部分進行相同的劃分操作，直至子陣列只剩下一個元素。
- **桶排序**：桶排序的基本思想是將資料分散到多個桶，然後對每個桶內的元素進行排序，最後將各個桶的元素依次取出，從而得到一個有序陣列。
- **樹**：例如二元搜尋樹、AVL 樹、紅黑樹、B 樹、B+ 樹等，它們的查詢、插入和刪除等操作都可以視為分治策略的應用。
- **堆積**：堆積是一種特殊的完全二元樹，其各種操作，如插入、刪除和堆積化，實際上都隱含了分治的思想。
- **雜湊表**：雖然雜湊表並不直接應用分治，但某些雜湊衝突解決方案間接應用了分治策略，例如，鏈式位址中的長鏈結串列會被轉化為紅黑樹，以提升查詢效率。

可以看出，**分治是一種“潤物細無聲”的演算法思想**，隱含在各種演算法與資料結構之中。

12.2　分治搜尋策略

我們已經學過，搜尋演算法分為兩大類。

- **暴力搜尋**：它透過走訪資料結構實現，時間複雜度為 $O(n)$ 。
- **自適應搜尋**：它利用特有的資料組織形式或先驗資訊，時間複雜度可達到 $O(\log n)$ 甚至 $O(1)$ 。

實際上，**時間複雜度為 $O(\log n)$ 的搜尋演算法通常是基於分治策略實現的**，例如二分搜尋和樹。

- 二分搜尋的每一步都將問題（在陣列中搜索目標元素）分解為一個小問題（在陣列的一半中搜索目標元素），這個過程一直持續到陣列為空或找到目標元素為止。
- 樹是分治思想的代表，在二元搜尋樹、AVL 樹、堆積等資料結構中，各種操作的時間複雜度皆為 $O(\log n)$ 。

二分搜尋的分治策略如下所示。

- **問題可以分解**：二分搜尋遞迴地將原問題（在陣列中進行查詢）分解為子問題（在陣列的一半中進行查詢），這是透過比較中間元素和目標元素來實現的。
- **子問題是獨立的**：在二分搜尋中，每輪只處理一個子問題，它不受其他子問題的影響。
- **子問題的解無須合併**：二分搜尋旨在查詢一個特定元素，因此不需要將子問題的解進行合併。當子問題得到解決時，原問題也會同時得到解決。

分治能夠提升搜尋效率，本質上是因為暴力搜尋每輪只能排除一個選項，**而分治搜尋每輪可以排除一半選項**。

1. 基於分治實現二分搜尋

在之前的章節中，二分搜尋是基於遞推（迭代）實現的。現在我們基於分治（遞迴）來實現它。

Question

給定一個長度為 n 的有序陣列 `nums`，其中所有元素都是唯一的，請查詢元素 `target`。

從分治角度，我們將搜尋區間 $[i, j]$ 對應的子問題記為 $f(i, j)$。

以原問題 $f(0, n-1)$ 為起始點，透過以下步驟進行二分搜尋。

1. 計算搜尋區間 $[i, j]$ 的中點 m，根據它排除一半搜尋區間。
2. 遞迴求解規模減小一半的子問題，可能為 $f(i, m-1)$ 或 $f(m+1, j)$。
3. 迴圈第 `1.` 步和第 `2.` 步，直至找到 `target` 或區間為空時返回。

圖 12-4 展示了在陣列中二分搜尋元素 6 的分治過程。

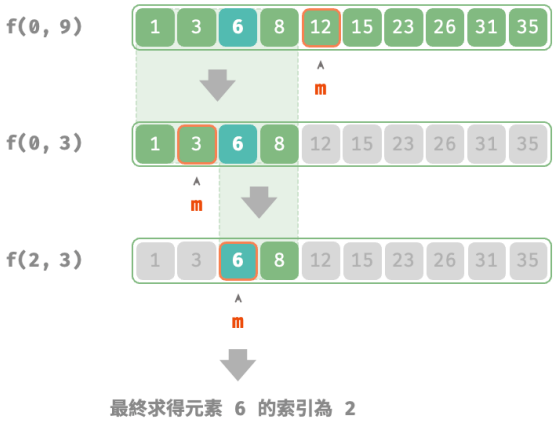


圖 12-4　二分搜尋的分治過程

在實現程式碼中，我們宣告一個遞迴函式 `dfs()` 來求解問題 $f(i, j)$：

```
// === File: binary_search_recur.rs ===

/* 二分搜尋：問題 f(i, j) */
fn dfs(nums: &[i32], target: i32, i: i32, j: i32) -> i32 {
    // 若區間為空，代表無目標元素，則返回 -1
    if i > j {
        return -1;
    }
    let m: i32 = i + (j - i) / 2;
    if nums[m as usize] < target {
        // 遞迴子問題 f(m+1, j)
        return dfs(nums, target, m + 1, j);
    } else if nums[m as usize] > target {
        // 遞迴子問題 f(i, m-1)
        return dfs(nums, target, i, m - 1);
```

```
    } else {
        // 找到目標元素，返回其索引
        return m;
    }
}

/* 二分搜尋 */
fn binary_search(nums: &[i32], target: i32) -> i32 {
    let n = nums.len() as i32;
    // 求解問題 f(0, n-1)
    dfs(nums, target, 0, n - 1)
}
```

12.3　構建二元樹問題

Question

給定一棵二元樹的前序走訪 `preorder` 和中序走訪 `inorder`，請從中構建二元樹，返回二元樹的根節點。假設二元樹中沒有值重複的節點（如圖 12-5 所示）。

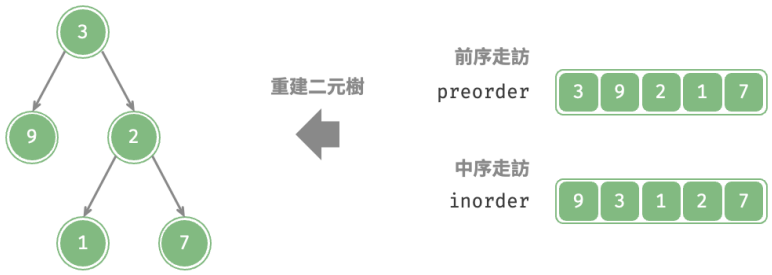


圖 12-5　構建二元樹的示例資料

1.　判斷是否為分治問題

原問題定義為從 `preorder` 和 `inorder` 構建二元樹，是一個典型的分治問題。

- **問題可以分解**：從分治的角度切入，我們可以將原問題劃分為兩個子問題：構建左子樹、構建右子樹，加上一步操作：初始化根節點。而對於每棵子樹（子問題），我們仍然可以複用以上劃分方法，將其劃分為更小的子樹（子問題），直至達到最小子問題（空子樹）時終止。
- **子問題是獨立的**：左子樹和右子樹是相互獨立的，它們之間沒有交集。在構建左子樹時，我們只需關注中序走訪和前序走訪中與左子樹對應的部分。右子樹同理。
- **子問題的解可以合併**：一旦得到了左子樹和右子樹（子問題的解），我們就可以將它們連結到根節點上，得到原問題的解。

2. 如何劃分子樹

根據以上分析，這道題可以使用分治來求解，**但如何透過前序走訪 `preorder` 和中序走訪 `inorder` 來劃分左子樹和右子樹呢**？

根據定義，`preorder` 和 `inorder` 都可以劃分為三個部分。

- 前序走訪：`[ 根節點 | 左子樹 | 右子樹 ]`，例如圖 12-5 的樹對應 `[ 3 | 9 | 2 1 7 ]`。
- 中序走訪：`[ 左子樹 | 根節點 | 右子樹 ]`，例如圖 12-5 的樹對應 `[ 9 | 3 | 1 2 7 ]`。

以上圖資料為例，我們可以透過圖 12-6 所示的步驟得到劃分結果。

1. 前序走訪的首元素 3 是根節點的值。
2. 查詢根節點 3 在 `inorder` 中的索引，利用該索引可將 `inorder` 劃分為 `[ 9 | 3 | 1 2 7 ]`。
3. 根據 `inorder` 的劃分結果，易得左子樹和右子樹的節點數量分別為 1 和 3，從而可將 `preorder` 劃分為 `[ 3 | 9 | 2 1 7 ]`。

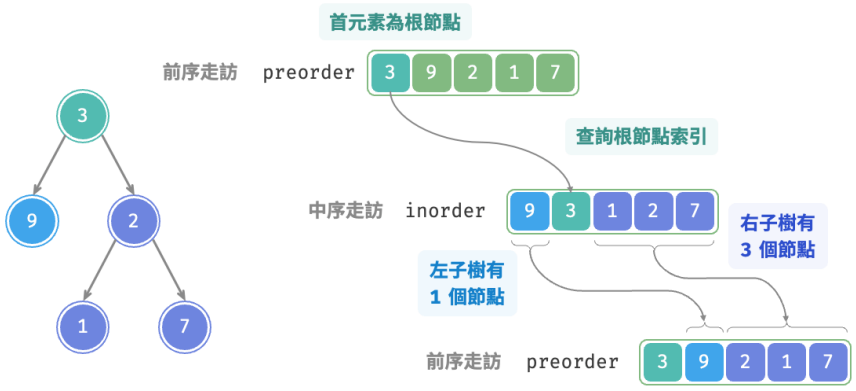


圖 12-6　在前序走訪和中序走訪中劃分子樹

3. 基於變數描述子樹區間

根據以上劃分方法，**我們已經得到根節點、左子樹、右子樹在 `preorder` 和 `inorder` 中的索引區間**。而為了描述這些索引區間，我們需要藉助幾個指標變數。

- 將當前樹的根節點在 `preorder` 中的索引記為 i。
- 將當前樹的根節點在 `inorder` 中的索引記為 m。
- 將當前樹在 `inorder` 中的索引區間記為 $[l, r]$。

如表 12-1 所示，透過以上變數即可表示根節點在 `preorder` 中的索引，以及子樹在 `inorder` 中的索引區間。

表 12-1　根節點和子樹在前序走訪和中序走訪中的索引

	根節點在 `preorder` 中的索引	子樹在 `inorder` 中的索引區間
當前樹	i	$[l, r]$
左子樹	$i + 1$	$[l, m - 1]$
右子樹	$i + 1 + (m - l)$	$[m + 1, r]$

請注意，右子樹根節點索引中的 $(m - l)$ 的含義是“左子樹的節點數量”，建議結合圖 12-7 理解。

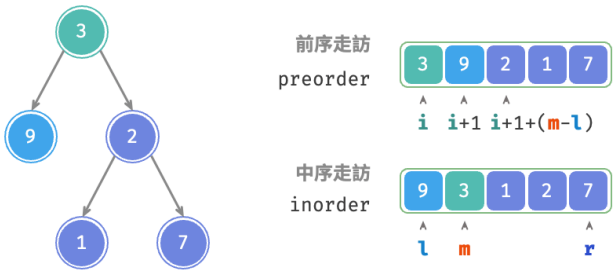


圖 12-7　根節點和左右子樹的索引區間表示

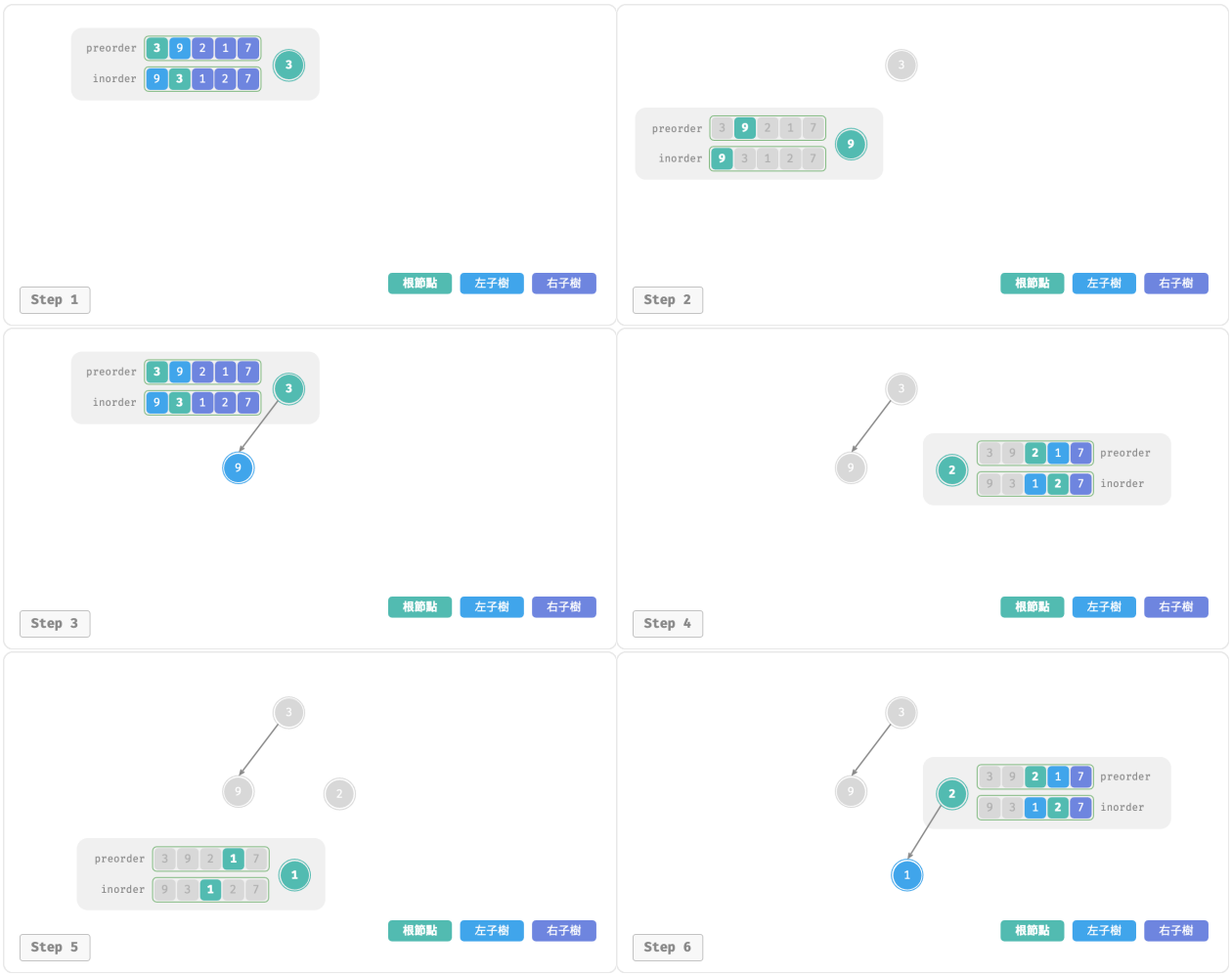
4. 程式碼實現

為了提升查詢 m 的效率，我們藉助一個雜湊表 `hmap` 來儲存陣列 `inorder` 中元素到索引的對映：

```
// === File: build_tree.rs ===

/* 構建二元樹：分治 */
fn dfs(
    preorder: &[i32],
    inorder_map: &HashMap<i32, i32>,
    i: i32,
    l: i32,
    r: i32,
) -> Option<Rc<RefCell<TreeNode>>> {
    // 子樹區間為空時終止
    if r - l < 0 {
        return None;
    }
    // 初始化根節點
    let root = TreeNode::new(preorder[i as usize]);
    // 查詢 m ，從而劃分左右子樹
    let m = inorder_map.get(&preorder[i as usize]).unwrap();
    // 子問題：構建左子樹
    root.borrow_mut().left = dfs(preorder, inorder_map, i + 1, l, m - 1);
```

```
    // 子問題：構建右子樹
    root.borrow_mut().right = dfs(preorder, inorder_map, i + 1 + m - l, m + 1, r);
    // 返回根節點
    Some(root)
}

/* 構建二元樹 */
fn build_tree(preorder: &[i32], inorder: &[i32]) -> Option<Rc<RefCell<TreeNode>>> {
    // 初始化雜湊表，儲存 inorder 元素到索引的對映
    let mut inorder_map: HashMap<i32, i32> = HashMap::new();
    for i in 0..inorder.len() {
        inorder_map.insert(inorder[i], i as i32);
    }
    let root = dfs(preorder, &inorder_map, 0, 0, inorder.len() as i32 - 1);
    root
}
```

圖 12-8 展示了構建二元樹的遞迴過程，各個節點是在向下“遞”的過程中建立的，而各條邊（引用）是在向上“迴”的過程中建立的。



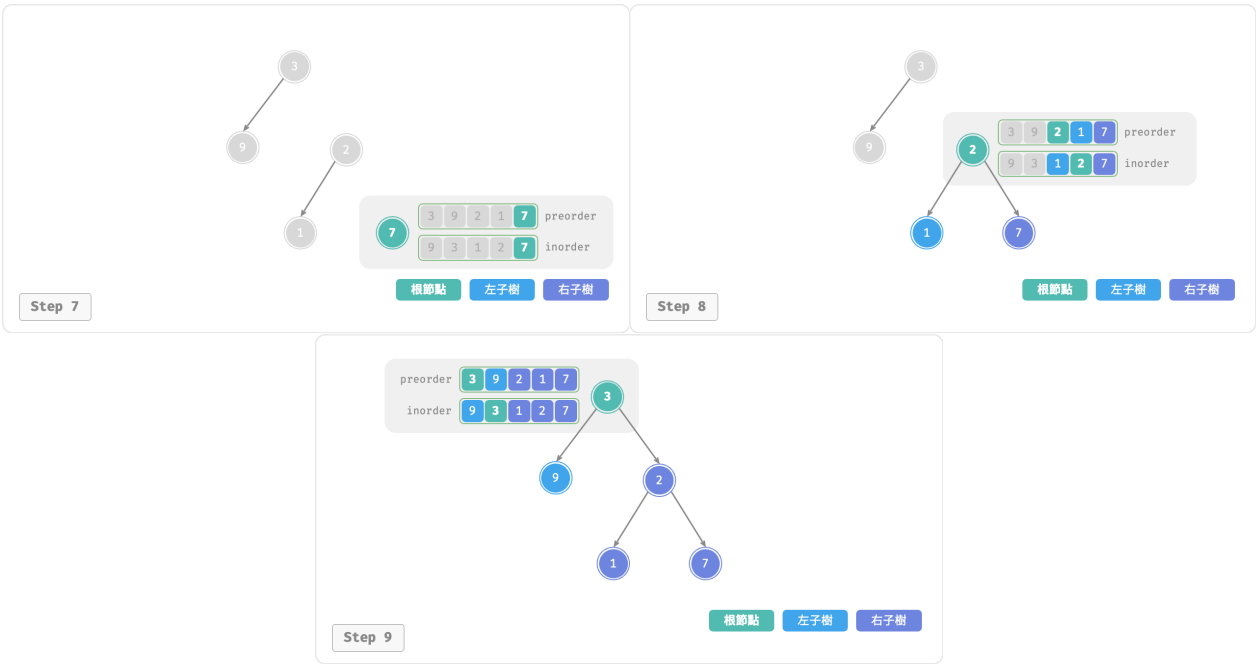


圖 12-8　構建二元樹的遞迴過程

每個遞迴函式內的前序走訪 `preorder` 和中序走訪 `inorder` 的劃分結果如圖 12-9 所示。

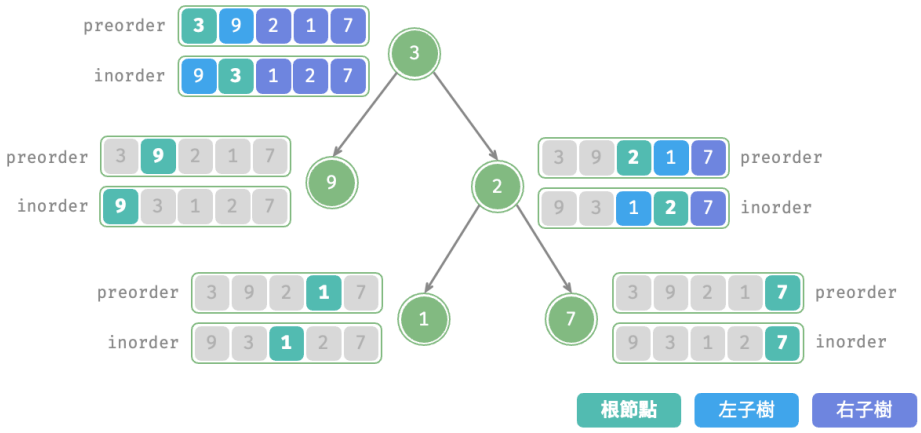


圖 12-9　每個遞迴函式中的劃分結果

設樹的節點數量為 n，初始化每一個節點（執行一個遞迴函式 `dfs()`）使用 $O(1)$ 時間。**因此總體時間複雜度為 $O(n)$**。

雜湊表儲存 `inorder` 元素到索引的對映，空間複雜度為 $O(n)$。在最差情況下，即二元樹退化為鏈結串列時，遞迴深度達到 n，使用 $O(n)$ 的堆疊幀空間。**因此總體空間複雜度為 $O(n)$**。

12.4　河內塔問題

在合併排序和構建二元樹中，我們都是將原問題分解為兩個規模為原問題一半的子問題。然而對於河內塔問題，我們採用不同的分解策略。

Question

給定三根柱子，記為 A、B 和 C。起始狀態下，柱子 A 上套著 n 個圓盤，它們從上到下按照從小到大的順序排列。我們的任務是要把這 n 個圓盤移到柱子 C 上，並保持它們的原有順序不變（如圖 12-10 所示）。在移動圓盤的過程中，需要遵守以下規則。

1. 圓盤只能從一根柱子頂部拿出，從另一根柱子頂部放入。
2. 每次只能移動一個圓盤。
3. 小圓盤必須時刻位於大圓盤之上。

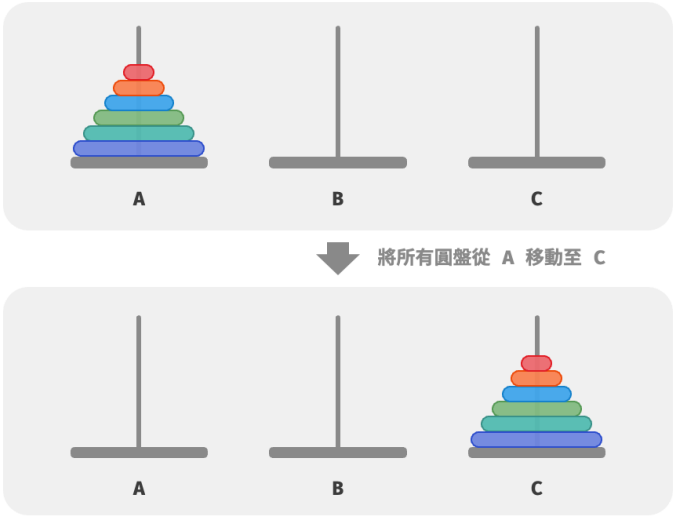


圖 12-10　河內塔問題示例

我們將規模為 i 的河內塔問題記作 $f(i)$。例如 $f(3)$ 代表將 3 個圓盤從 A 移動至 C 的河內塔問題。

1. 考慮基本情況

如圖 12-11 所示，對於問題 $f(1)$，即當只有一個圓盤時，我們將它直接從 A 移動至 C 即可。

圖 12-11　規模為 1 的問題的解

如圖 12-12 所示，對於問題 $f(2)$，即當有兩個圓盤時，**由於要時刻滿足小圓盤在大圓盤之上，因此需要藉助 B 來完成移動**。

1. 先將上面的小圓盤從 A 移至 B 。
2. 再將大圓盤從 A 移至 C 。
3. 最後將小圓盤從 B 移至 C 。

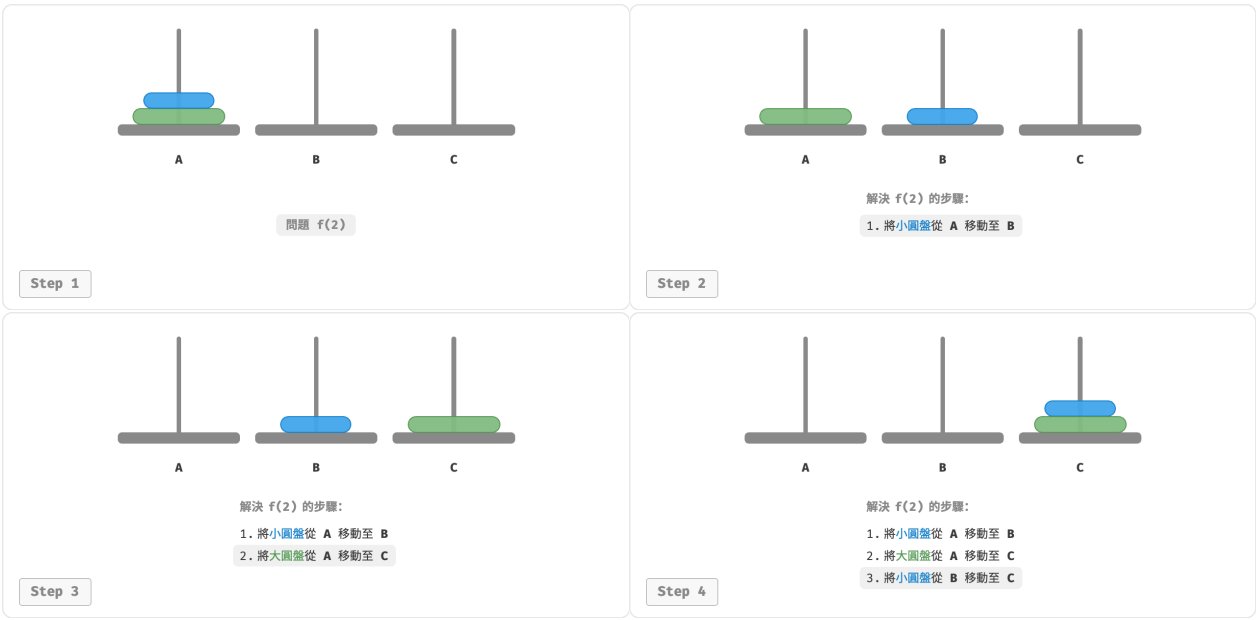


圖 12-12　規模為 2 的問題的解

解決問題 $f(2)$ 的過程可總結為：**將兩個圓盤藉助 B 從 A 移至 C** 。其中，C 稱為目標柱、B 稱為緩衝柱。

2. 子問題分解

對於問題 $f(3)$，即當有三個圓盤時，情況變得稍微複雜了一些。

因為已知 $f(1)$ 和 $f(2)$ 的解，所以我們可從分治角度思考，**將 A 頂部的兩個圓盤看作一個整體**，執行圖 12-13 所示的步驟。這樣三個圓盤就被順利地從 A 移至 C 了。

1. 令 B 為目標柱、C 為緩衝柱，將兩個圓盤從 A 移至 B 。
2. 將 A 中剩餘的一個圓盤從 A 直接移動至 C 。
3. 令 C 為目標柱、A 為緩衝柱，將兩個圓盤從 B 移至 C 。

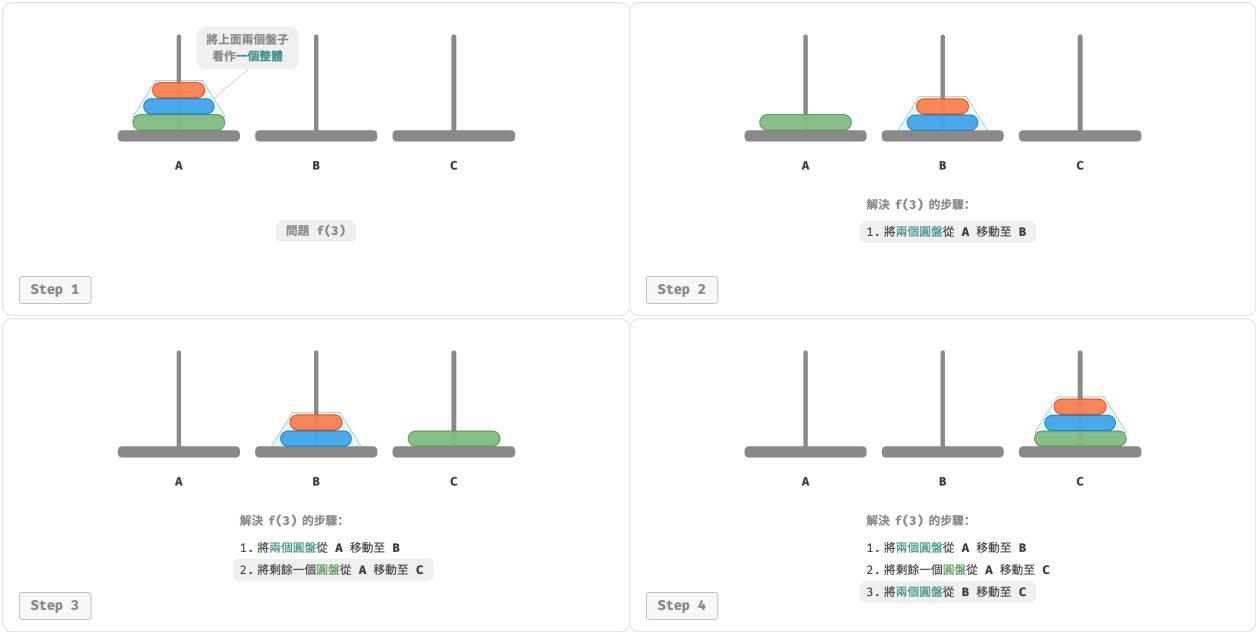


圖 12-13　規模為 3 的問題的解

從本質上看，**我們將問題 $f(3)$ 劃分為兩個子問題 $f(2)$ 和一個子問題 $f(1)$** 。按順序解決這三個子問題之後，原問題隨之得到解決。這說明子問題是獨立的，而且解可以合併。

至此，我們可總結出圖 12-14 所示的解決河內塔問題的分治策略：將原問題 $f(n)$ 劃分為兩個子問題 $f(n-1)$ 和一個子問題 $f(1)$ ，並按照以下順序解決這三個子問題。

1. 將 $n-1$ 個圓盤藉助 C 從 A 移至 B 。
2. 將剩餘 1 個圓盤從 A 直接移至 C 。
3. 將 $n-1$ 個圓盤藉助 A 從 B 移至 C 。

對於這兩個子問題 $f(n-1)$ ，**可以透過相同的方式進行遞迴劃分**，直至達到最小子問題 $f(1)$ 。而 $f(1)$ 的解是已知的，只需一次移動操作即可。

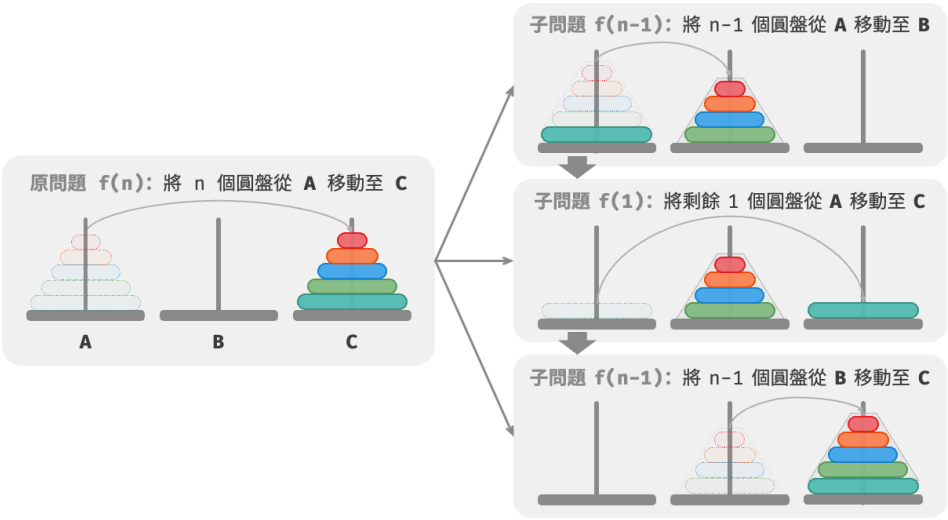


圖 12-14　解決河內塔問題的分治策略

3. 程式碼實現

在程式碼中，我們宣告一個遞迴函式 `dfs(i, src, buf, tar)`，它的作用是將柱 `src` 頂部的 i 個圓盤藉助緩衝柱 `buf` 移動至目標柱 `tar`：

```rust
// === File: hanota.rs ===

/* 移動一個圓盤 */
fn move_pan(src: &mut Vec<i32>, tar: &mut Vec<i32>) {
    // 從 src 頂部拿出一個圓盤
    let pan = src.pop().unwrap();
    // 將圓盤放入 tar 頂部
    tar.push(pan);
}

/* 求解河內塔問題 f(i) */
fn dfs(i: i32, src: &mut Vec<i32>, buf: &mut Vec<i32>, tar: &mut Vec<i32>) {
    // 若 src 只剩下一個圓盤，則直接將其移到 tar
    if i == 1 {
        move_pan(src, tar);
        return;
    }
    // 子問題 f(i-1) ：將 src 頂部 i-1 個圓盤藉助 tar 移到 buf
    dfs(i - 1, src, tar, buf);
    // 子問題 f(1) ：將 src 剩餘一個圓盤移到 tar
    move_pan(src, tar);
    // 子問題 f(i-1) ：將 buf 頂部 i-1 個圓盤藉助 src 移到 tar
    dfs(i - 1, buf, src, tar);
```

```rust
}

/* 求解河內塔問題 */
fn solve_hanota(A: &mut Vec<i32>, B: &mut Vec<i32>, C: &mut Vec<i32>) {
    let n = A.len() as i32;
    // 將 A 頂部 n 個圓盤藉助 B 移到 C
    dfs(n, A, B, C);
}
```

如圖 12-15 所示，河內塔問題形成一棵高度為 n 的遞迴樹，每個節點代表一個子問題，對應一個開啟的 `dfs()` 函式，**因此時間複雜度為 $O(2^n)$，空間複雜度為 $O(n)$**。

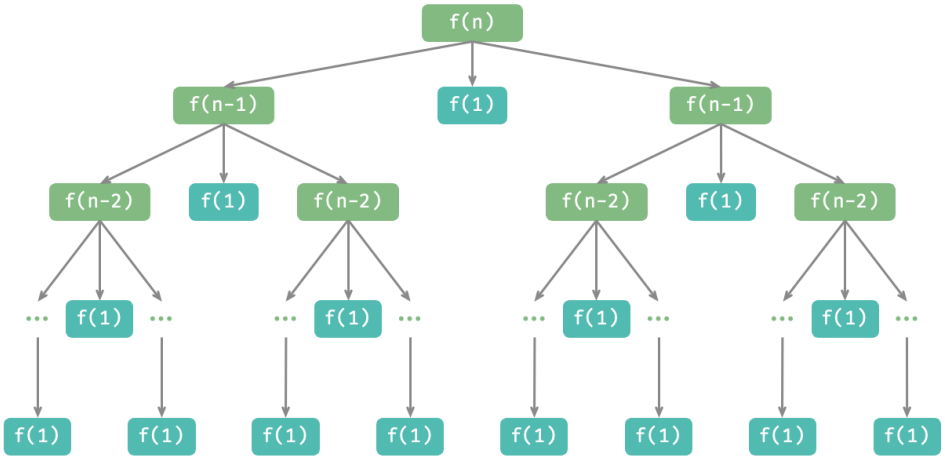


圖 12-15　河內塔問題的遞迴樹

Quote

河內塔問題源自一個古老的傳說。在古印度的一個寺廟裡，僧侶們有三根高大的鑽石柱子，以及 64 個大小不一的金圓盤。僧侶們不斷地移動圓盤，他們相信在最後一個圓盤被正確放置的那一刻，這個世界就會結束。

然而，即使僧侶們每秒鐘移動一次，總共需要大約 $2^{64} \approx 1.84 \times 10^{19}$ 秒，合約 5850 億年，遠遠超過了現在對宇宙年齡的估計。所以，倘若這個傳說是真的，我們應該不需要擔心世界末日的到來。

12.5　小結

- 分治是一種常見的演算法設計策略，包括分（劃分）和治（合併）兩個階段，通常基於遞迴實現。
- 判斷是否是分治演算法問題的依據包括：問題能否分解、子問題是否獨立、子問題能否合併。
- 合併排序是分治策略的典型應用，其遞迴地將陣列劃分為等長的兩個子陣列，直到只剩一個元素時開始逐層合併，從而完成排序。

- 引入分治策略往往可以提升演算法效率。一方面，分治策略減少了操作數量；另一方面，分治後有利於系統的並行最佳化。
- 分治既可以解決許多演算法問題，也廣泛應用於資料結構與演算法設計中，處處可見其身影。
- 相較於暴力搜尋，自適應搜尋效率更高。時間複雜度為 $O(\log n)$ 的搜尋演算法通常是基於分治策略實現的。
- 二分搜尋是分治策略的另一個典型應用，它不包含將子問題的解進行合併的步驟。我們可以透過遞迴分治實現二分搜尋。
- 在構建二元樹的問題中，構建樹（原問題）可以劃分為構建左子樹和右子樹（子問題），這可以透過劃分前序走訪和中序走訪的索引區間來實現。
- 在河內塔問題中，一個規模為 n 的問題可以劃分為兩個規模為 $n - 1$ 的子問題和一個規模為 1 的子問題。按順序解決這三個子問題後，原問題隨之得到解決。

第 13 章　回溯

Abstract

我們如同迷宮中的探索者，在前進的道路上可能會遇到困難。

回溯的力量讓我們能夠重新開始，不斷嘗試，最終找到通往光明的出口。

13.1　回溯演算法

回溯演算法（backtracking algorithm）是一種透過窮舉來解決問題的方法，它的核心思想是從一個初始狀態出發，暴力搜尋所有可能的解決方案，當遇到正確的解則將其記錄，直到找到解或者嘗試了所有可能的選擇都無法找到解為止。

回溯演算法通常採用“深度優先搜尋”來走訪解空間。在“二元樹”章節中，我們提到前序、中序和後序走訪都屬於深度優先搜尋。接下來，我們利用前序走訪構造一個回溯問題，逐步瞭解回溯演算法的工作原理。

> **例題一**
> 給定一棵二元樹，搜尋並記錄所有值為 7 的節點，請返回節點串列。

對於此題，我們前序走訪這棵樹，並判斷當前節點的值是否為 7 ，若是，則將該節點的值加入結果串列 `res` 之中。相關過程實現如圖 13-1 和以下程式碼所示：

```rust
// === File: preorder_traversal_i_compact.rs ===

/* 前序走訪：例題一 */
fn pre_order(res: &mut Vec<Rc<RefCell<TreeNode>>>, root: Option<&Rc<RefCell<TreeNode>>>) {
    if root.is_none() {
        return;
    }
    if let Some(node) = root {
        if node.borrow().val == 7 {
            // 記錄解
            res.push(node.clone());
        }
        pre_order(res, node.borrow().left.as_ref());
        pre_order(res, node.borrow().right.as_ref());
    }
}
```

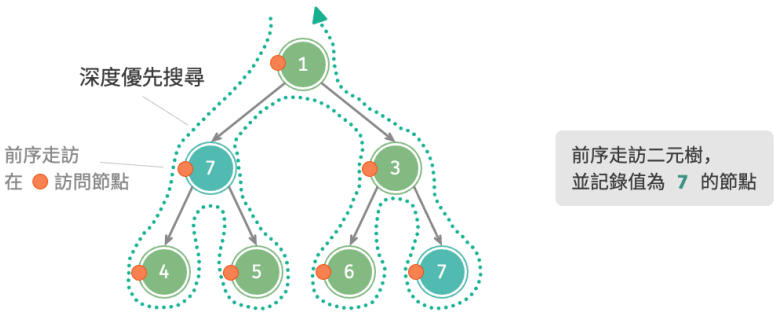


圖 13-1　在前序走訪中搜索節點

13.1.1　嘗試與回退

之所以稱之為回溯演算法，是因為該演算法在搜尋解空間時會採用“嘗試”與“回退”的策略。當演算法在搜尋過程中遇到某個狀態無法繼續前進或無法得到滿足條件的解時，它會撤銷上一步的選擇，退回到之前的狀態，並嘗試其他可能的選擇。

對於例題一，訪問每個節點都代表一次“嘗試”，而越過葉節點或返回父節點的 `return` 則表示“回退”。

值得說明的是，**回退並不僅僅包括函式返回**。為解釋這一點，我們對例題一稍作拓展。

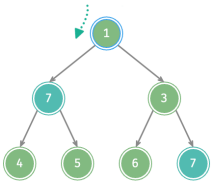
> **例題二**
> 在二元樹中搜索所有值為 7 的節點，**請返回根節點到這些節點的路徑**。

在例題一程式碼的基礎上，我們需要藉助一個串列 `path` 記錄訪問過的節點路徑。當訪問到值為 7 的節點時，則複製 `path` 並新增進結果串列 `res` 。走訪完成後，`res` 中儲存的就是所有的解。程式碼如下所示：

```
// === File: preorder_traversal_ii_compact.rs ===

/* 前序走訪：例題二 */
fn pre_order(
    res: &mut Vec<Vec<Rc<RefCell<TreeNode>>>>,
    path: &mut Vec<Rc<RefCell<TreeNode>>>,
    root: Option<&Rc<RefCell<TreeNode>>>,
) {
    if root.is_none() {
        return;
    }
    if let Some(node) = root {
        // 嘗試
        path.push(node.clone());
        if node.borrow().val == 7 {
            // 記錄解
            res.push(path.clone());
        }
        pre_order(res, path, node.borrow().left.as_ref());
        pre_order(res, path, node.borrow().right.as_ref());
        // 回退
        path.pop();
    }
}
```

在每次“嘗試”中，我們透過將當前節點新增進 `path` 來記錄路徑；而在“回退”前，我們需要將該節點從 `path` 中彈出，**以恢復本次嘗試之前的狀態**。

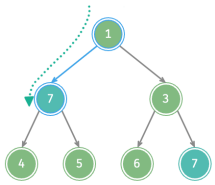
觀察圖 13-2 所示的過程，**我們可以將嘗試和回退理解為“前進”與“撤銷”**，兩個操作互為逆向。



嘗試：遞迴走訪，更新狀態
回退：恢復狀態，函式返回
path = [1]

res = []

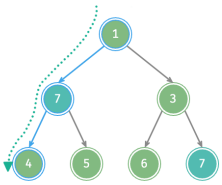
Step 1



嘗試：遞迴走訪，更新狀態
回退：恢復狀態，函式返回
path = [1, 7]

遇到節點 7 ，則將路徑新增至結果
res = [[1, 7]]

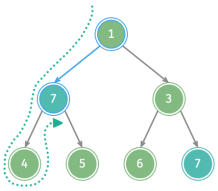
Step 2



嘗試：遞迴走訪，更新狀態
回退：恢復狀態，函式返回
path = [1, 7, 4]

res = [[1, 7]]

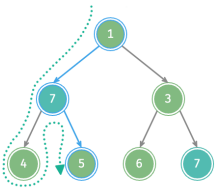
Step 3



嘗試：遞迴走訪，更新狀態
回退：恢復狀態，函式返回
path = [1, 7]

res = [[1, 7]]

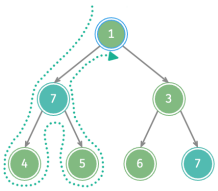
Step 4



嘗試：遞迴走訪，更新狀態
回退：恢復狀態，函式返回
path = [1, 7, 5]

res = [[1, 7]]

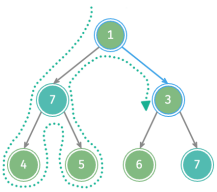
Step 5



嘗試：遞迴走訪，更新狀態
回退：恢復狀態，函式返回
path = [1]

res = [[1, 7]]

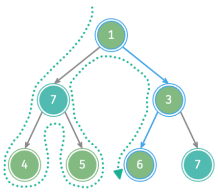
Step 6



嘗試：遞迴走訪，更新狀態
回退：恢復狀態，函式返回
path = [1, 3]

res = [[1, 7]]

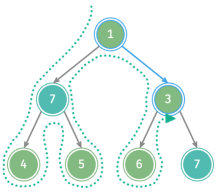
Step 7



嘗試：遞迴走訪，更新狀態
回退：恢復狀態，函式返回
path = [1, 3, 6]

res = [[1, 7]]

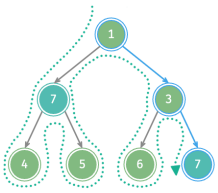
Step 8



嘗試：遞迴走訪，更新狀態
回退：恢復狀態，函式返回
path = [1, 3]

res = [[1, 7]]

Step 9



嘗試：遞迴走訪，更新狀態
回退：恢復狀態，函式返回
path = [1, 3, 7]

遇到節點 7 ，則將路徑新增至結果
res = [[1, 7], [1, 3, 7]]

Step 10

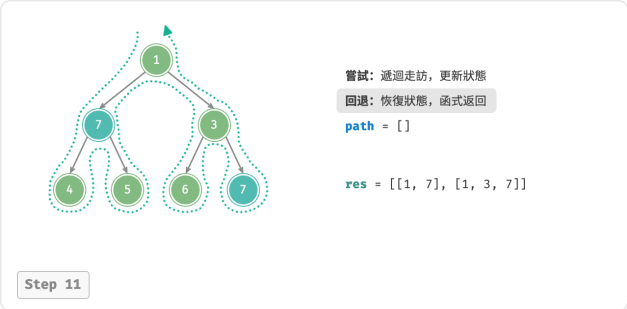


圖 13-2　嘗試與回退

13.1.2　剪枝

複雜的回溯問題通常包含一個或多個約束條件，**約束條件通常可用於“剪枝”**。

> **例題三**
> 在二元樹中搜索所有值為 7 的節點，請返回根節點到這些節點的路徑，**並要求路徑中不包含值為 3 的節點**。

為了滿足以上約束條件，**我們需要新增剪枝操作**：在搜尋過程中，若遇到值為 3 的節點，則提前返回，不再繼續搜尋。程式碼如下所示：

```
// === File: preorder_traversal_iii_compact.rs ===

/* 前序走訪：例題三 */
fn pre_order(
    res: &mut Vec<Vec<Rc<RefCell<TreeNode>>>>,
    path: &mut Vec<Rc<RefCell<TreeNode>>>,
    root: Option<&Rc<RefCell<TreeNode>>>,
) {
    // 剪枝
    if root.is_none() || root.as_ref().unwrap().borrow().val == 3 {
        return;
    }
    if let Some(node) = root {
        // 嘗試
        path.push(node.clone());
        if node.borrow().val == 7 {
            // 記錄解
            res.push(path.clone());
        }
        pre_order(res, path, node.borrow().left.as_ref());
        pre_order(res, path, node.borrow().right.as_ref());
        // 回退
        path.pop();
```

```
    }
}
```

“剪枝”是一個非常形象的名詞。如圖 13-3 所示，在搜尋過程中，**我們“剪掉”了不滿足約束條件的搜尋分支**，避免許多無意義的嘗試，從而提高了搜尋效率。

圖 13-3　根據約束條件剪枝

13.1.3　框架程式碼

接下來，我們嘗試將回溯的“嘗試、回退、剪枝”的主體框架提煉出來，提升程式碼的通用性。

在以下框架程式碼中，`state` 表示問題的當前狀態，`choices` 表示當前狀態下可以做出的選擇：

```
/* 回溯演算法框架 */
fn backtrack(state: &mut State, choices: &Vec<Choice>, res: &mut Vec<State>) {
    // 判斷是否為解
    if is_solution(state) {
        // 記錄解
        record_solution(state, res);
        // 不再繼續搜尋
        return;
    }
    // 走訪所有選擇
    for choice in choices {
        // 剪枝：判斷選擇是否合法
        if is_valid(state, choice) {
            // 嘗試：做出選擇，更新狀態
            make_choice(state, choice);
            backtrack(state, choices, res);
            // 回退：撤銷選擇，恢復到之前的狀態
            undo_choice(state, choice);
        }
    }
}
```

接下來，我們基於框架程式碼來解決例題三。狀態 `state` 為節點走訪路徑，選擇 `choices` 為當前節點的左子節點和右子節點，結果 `res` 是路徑串列：

```
// === File: preorder_traversal_iii_template.rs ===

/* 判斷當前狀態是否為解 */
fn is_solution(state: &mut Vec<Rc<RefCell<TreeNode>>>) -> bool {
    return !state.is_empty() && state.last().unwrap().borrow().val == 7;
}

/* 記錄解 */
fn record_solution(
    state: &mut Vec<Rc<RefCell<TreeNode>>>,
    res: &mut Vec<Vec<Rc<RefCell<TreeNode>>>>,
) {
    res.push(state.clone());
}

/* 判斷在當前狀態下，該選擇是否合法 */
fn is_valid(_: &mut Vec<Rc<RefCell<TreeNode>>>, choice: Option<&Rc<RefCell<TreeNode>>>) -> bool {
    return choice.is_some() && choice.unwrap().borrow().val != 3;
}

/* 更新狀態 */
fn make_choice(state: &mut Vec<Rc<RefCell<TreeNode>>>, choice: Rc<RefCell<TreeNode>>) {
    state.push(choice);
}

/* 恢復狀態 */
fn undo_choice(state: &mut Vec<Rc<RefCell<TreeNode>>>, _: Rc<RefCell<TreeNode>>) {
    state.pop();
}

/* 回溯演算法：例題三 */
fn backtrack(
    state: &mut Vec<Rc<RefCell<TreeNode>>>,
    choices: &Vec<Option<&Rc<RefCell<TreeNode>>>>,
    res: &mut Vec<Vec<Rc<RefCell<TreeNode>>>>,
) {
    // 檢查是否為解
    if is_solution(state) {
        // 記錄解
        record_solution(state, res);
    }
    // 走訪所有選擇
    for &choice in choices.iter() {
        // 剪枝：檢查選擇是否合法
```

```
        if is_valid(state, choice) {
            // 嘗試：做出選擇，更新狀態
            make_choice(state, choice.unwrap().clone());
            // 進行下一輪選擇
            backtrack(
                state,
                &vec![
                    choice.unwrap().borrow().left.as_ref(),
                    choice.unwrap().borrow().right.as_ref(),
                ],
                res,
            );
            // 回退：撤銷選擇，恢復到之前的狀態
            undo_choice(state, choice.unwrap().clone());
        }
    }
}
```

根據題意，我們在找到值為 7 的節點後應該繼續搜尋，**因此需要將記錄解之後的 `return` 語句刪除**。圖 13-4 對比了保留或刪除 `return` 語句的搜尋過程。

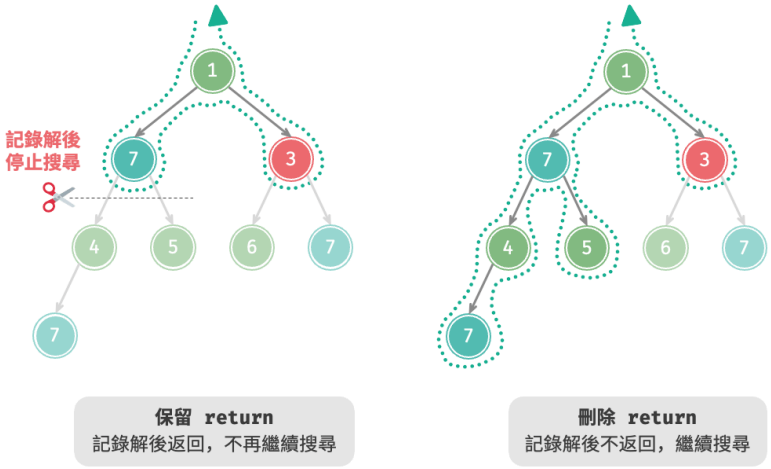


圖 13-4　保留與刪除 return 的搜尋過程對比

相比基於前序走訪的程式碼實現，基於回溯演算法框架的程式碼實現雖然顯得囉唆，但通用性更好。實際上，**許多回溯問題可以在該框架下解決**。我們只需根據具體問題來定義 `state` 和 `choices`，並實現框架中的各個方法即可。

13.1.4　常用術語

為了更清晰地分析演算法問題，我們總結一下回溯演算法中常用術語的含義，並對照例題三給出對應示例，如表 13-1 所示。

表 13-1　常見的回溯演算法術語

名詞	定義	例題三
解（solution）	解是滿足問題特定條件的答案，可能有一個或多個	根節點到節點 7 的滿足約束條件的所有路徑
約束條件（constraint）	約束條件是問題中限制解的可行性的條件，通常用於剪枝	路徑中不包含節點 3
狀態（state）	狀態表示問題在某一時刻的情況，包括已經做出的選擇	當前已訪問的節點路徑，即 `path` 節點串列
嘗試（attempt）	嘗試是根據可用選擇來探索解空間的過程，包括做出選擇，更新狀態，檢查是否為解	遞迴訪問左（右）子節點，將節點新增進 `path`，判斷節點的值是否為 7
回退（backtracking）	回退指遇到不滿足約束條件的狀態時，撤銷前面做出的選擇，回到上一個狀態	當越過葉節點、結束節點訪問、遇到值為 3 的節點時終止搜尋，函式返回
剪枝（pruning）	剪枝是根據問題特性和約束條件避免無意義的搜尋路徑的方法，可提高搜尋效率	當遇到值為 3 的節點時，則不再繼續搜尋

> **Tip**
> 問題、解、狀態等概念是通用的，在分治、回溯、動態規劃、貪婪等演算法中都有涉及。

13.1.5　優點與侷限性

回溯演算法本質上是一種深度優先搜尋演算法，它嘗試所有可能的解決方案直到找到滿足條件的解。這種方法的優點在於能夠找到所有可能的解決方案，而且在合理的剪枝操作下，具有很高的效率。

然而，在處理大規模或者複雜問題時，**回溯演算法的執行效率可能難以接受**。

- **時間**：回溯演算法通常需要走訪狀態空間的所有可能，時間複雜度可以達到指數階或階乘階。
- **空間**：在遞迴呼叫中需要儲存當前的狀態（例如路徑、用於剪枝的輔助變數等），當深度很大時，空間需求可能會變得很大。

即便如此，**回溯演算法仍然是某些搜尋問題和約束滿足問題的最佳解決方案**。對於這些問題，由於無法預測哪些選擇可生成有效的解，因此我們必須對所有可能的選擇進行走訪。在這種情況下，**關鍵是如何最佳化效率**，常見的效率最佳化方法有兩種。

- **剪枝**：避免搜尋那些肯定不會產生解的路徑，從而節省時間和空間。
- **啟發式搜尋**：在搜尋過程中引入一些策略或者估計值，從而優先搜尋最有可能產生有效解的路徑。

13.1.6 回溯典型例題

回溯演算法可用於解決許多搜尋問題、約束滿足問題和組合最佳化問題。

搜尋問題：這類問題的目標是找到滿足特定條件的解決方案。

- 全排列問題：給定一個集合，求出其所有可能的排列組合。
- 子集和問題：給定一個集合和一個目標和，找到集合中所有和為目標和的子集。
- 河內塔問題：給定三根柱子和一系列大小不同的圓盤，要求將所有圓盤從一根柱子移動到另一根柱子，每次只能移動一個圓盤，且不能將大圓盤放在小圓盤上。

約束滿足問題：這類問題的目標是找到滿足所有約束條件的解。

- n 皇后：在 $n \times n$ 的棋盤上放置 n 個皇后，使得它們互不攻擊。
- 數獨：在 9×9 的網格中填入數字 $1 \sim 9$ ，使得每行、每列和每個 3×3 子網格中的數字不重複。
- 圖著色問題：給定一個無向圖，用最少的顏色給圖的每個頂點著色，使得相鄰頂點顏色不同。

組合最佳化問題：這類問題的目標是在一個組合空間中找到滿足某些條件的最優解。

- 0-1 背包問題：給定一組物品和一個背包，每個物品有一定的價值和重量，要求在背包容量限制內，選擇物品使得總價值最大。
- 旅行商問題：在一個圖中，從一個點出發，訪問所有其他點恰好一次後返回起點，求最短路徑。
- 最大團問題：給定一個無向圖，找到最大的完全子圖，即子圖中的任意兩個頂點之間都有邊相連。

請注意，對於許多組合最佳化問題，回溯不是最優解決方案。

- 0-1 背包問題通常使用動態規劃解決，以達到更高的時間效率。
- 旅行商是一個著名的 NP-Hard 問題，常用解法有遺傳演算法和蟻群演算法等。
- 最大團問題是圖論中的一個經典問題，可用貪婪演算法等啟發式演算法來解決。

13.2 全排列問題

全排列問題是回溯演算法的一個典型應用。它的定義是在給定一個集合（如一個陣列或字串）的情況下，找出其中元素的所有可能的排列。

表 13-2 列舉了幾個示例資料，包括輸入陣列和對應的所有排列。

表 13-2　全排列示例

輸入陣列	所有排列
$[1]$	$[1]$
$[1, 2]$	$[1, 2], [2, 1]$
$[1, 2, 3]$	$[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]$

13.2.1　無相等元素的情況

> **Question**
> 輸入一個整數陣列，其中不包含重複元素，返回所有可能的排列。

從回溯演算法的角度看，**我們可以把生成排列的過程想象成一系列選擇的結果**。假設輸入陣列為 $[1, 2, 3]$，如果我們先選擇 1，再選擇 3，最後選擇 2，則獲得排列 $[1, 3, 2]$。回退表示撤銷一個選擇，之後繼續嘗試其他選擇。

從回溯程式碼的角度看，候選集合 `choices` 是輸入陣列中的所有元素，狀態 `state` 是直至目前已被選擇的元素。請注意，每個元素只允許被選擇一次，**因此 `state` 中的所有元素都應該是唯一的**。

如圖 13-5 所示，我們可以將搜尋過程展開成一棵遞迴樹，樹中的每個節點代表當前狀態 `state`。從根節點開始，經過三輪選擇後到達葉節點，每個葉節點都對應一個排列。

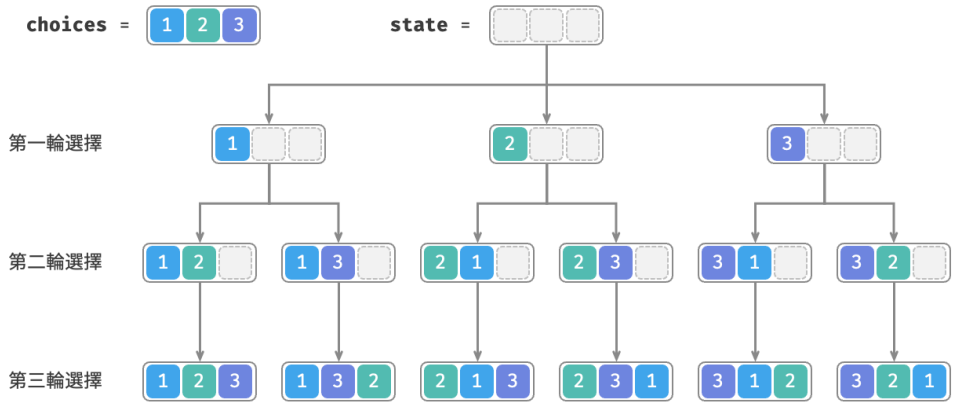


圖 13-5　全排列的遞迴樹

1.　重複選擇剪枝

為了實現每個元素只被選擇一次，我們考慮引入一個布林型陣列 `selected`，其中 `selected[i]` 表示 `choices[i]` 是否已被選擇，並基於它實現以下剪枝操作。

- 在做出選擇 `choice[i]` 後，我們就將 `selected[i]` 賦值為 True，代表它已被選擇。
- 走訪選擇串列 `choices` 時，跳過所有已被選擇的節點，即剪枝。

如圖 13-6 所示，假設我們第一輪選擇 1，第二輪選擇 3，第三輪選擇 2，則需要在第二輪剪掉元素 1 的分支，在第三輪剪掉元素 1 和元素 3 的分支。

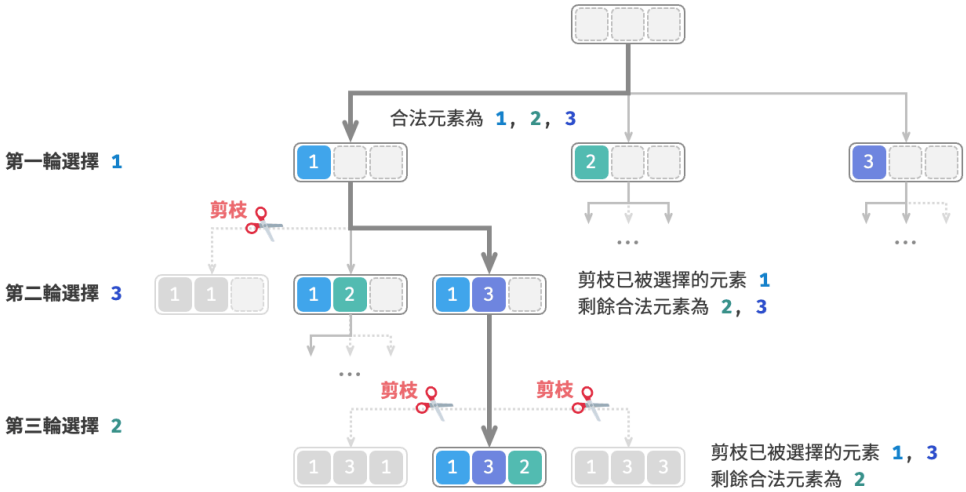


圖 13-6　全排列剪枝示例

觀察圖 13-6 發現，該剪枝操作將搜尋空間大小從 $O(n^n)$ 減小至 $O(n!)$ 。

2. 程式碼實現

想清楚以上資訊之後，我們就可以在框架程式碼中做“完形填空”了。為了縮短整體程式碼，我們不單獨實現框架程式碼中的各個函式，而是將它們展開在 `backtrack()` 函式中：

```
// === File: permutations_i.rs ===

/* 回溯演算法：全排列 I */
fn backtrack(mut state: Vec<i32>, choices: &[i32], selected: &mut [bool], res: &mut Vec<Vec<i32>>) {
    // 當狀態長度等於元素數量時，記錄解
    if state.len() == choices.len() {
        res.push(state);
        return;
    }
    // 走訪所有選擇
    for i in 0..choices.len() {
        let choice = choices[i];
        // 剪枝：不允許重複選擇元素
        if !selected[i] {
            // 嘗試：做出選擇，更新狀態
            selected[i] = true;
            state.push(choice);
            // 進行下一輪選擇
            backtrack(state.clone(), choices, selected, res);
            // 回退：撤銷選擇，恢復到之前的狀態
            selected[i] = false;
```

```
            state.pop();
        }
    }
}

/* 全排列 I */
fn permutations_i(nums: &mut [i32]) -> Vec<Vec<i32>> {
    let mut res = Vec::new(); // 狀態（子集）
    backtrack(Vec::new(), nums, &mut vec![false; nums.len()], &mut res);
    res
}
```

13.2.2　考慮相等元素的情況

Question

輸入一個整數陣列，**陣列中可能包含重複元素**，返回所有不重複的排列。

假設輸入陣列為 $[1, 1, 2]$ 。為了方便區分兩個重複元素 1 ，我們將第二個 1 記為 $\hat{1}$ 。

如圖 13-7 所示，上述方法生成的排列有一半是重複的。

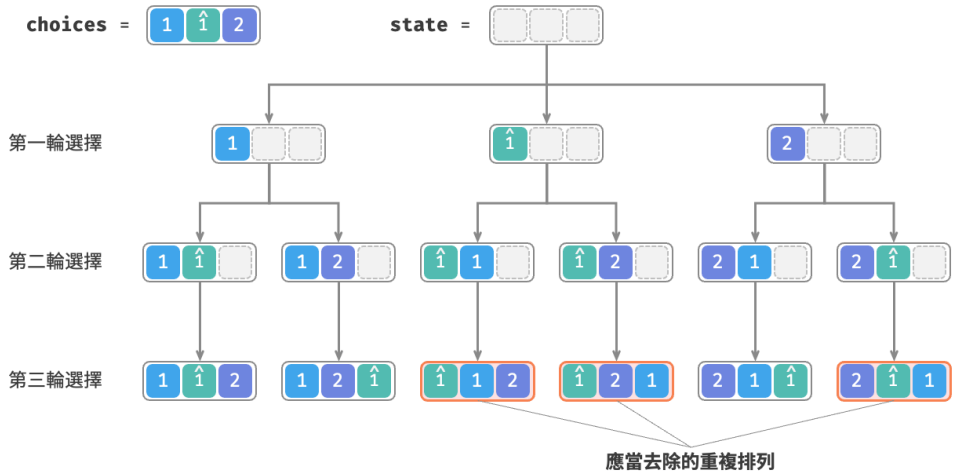


圖 13-7　重複排列

那麼如何去除重複的排列呢？最直接地，考慮藉助一個雜湊集合，直接對排列結果進行去重。然而這樣做不夠優雅，**因為生成重複排列的搜尋分支沒有必要，應當提前識別並剪枝**，這樣可以進一步提升演算法效率。

1.　相等元素剪枝

觀察圖 13-8 ，在第一輪中，選擇 1 或選擇 $\hat{1}$ 是等價的，在這兩個選擇之下生成的所有排列都是重複的。因此應該把 $\hat{1}$ 剪枝。

同理，在第一輪選擇 2 之後，第二輪選擇中的 1 和 $\hat{1}$ 也會產生重複分支，因此也應將第二輪的 $\hat{1}$ 剪枝。

從本質上看，**我們的目標是在某一輪選擇中，保證多個相等的元素僅被選擇一次**。

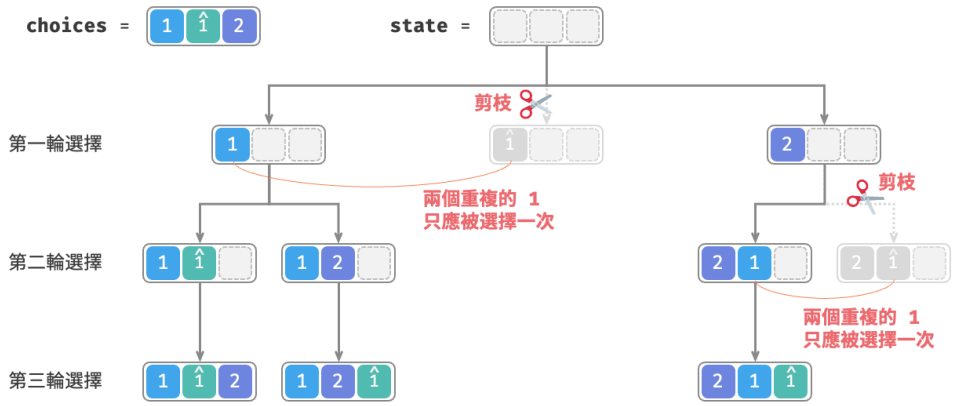


圖 13-8　重複排列剪枝

2. 程式碼實現

在上一題的程式碼的基礎上，我們考慮在每一輪選擇中開啟一個雜湊集合 `duplicated`，用於記錄該輪中已經嘗試過的元素，並將重複元素剪枝：

```
// === File: permutations_ii.rs ===

/* 回溯演算法：全排列 II */
fn backtrack(mut state: Vec<i32>, choices: &[i32], selected: &mut [bool], res: &mut Vec<Vec<i32>>) {
    // 當狀態長度等於元素數量時，記錄解
    if state.len() == choices.len() {
        res.push(state);
        return;
    }
    // 走訪所有選擇
    let mut duplicated = HashSet::<i32>::new();
    for i in 0..choices.len() {
        let choice = choices[i];
        // 剪枝：不允許重複選擇元素 且 不允許重複選擇相等元素
        if !selected[i] && !duplicated.contains(&choice) {
            // 嘗試：做出選擇，更新狀態
            duplicated.insert(choice); // 記錄選擇過的元素值
            selected[i] = true;
            state.push(choice);
            // 進行下一輪選擇
            backtrack(state.clone(), choices, selected, res);
            // 回退：撤銷選擇，恢復到之前的狀態
```

```
            selected[i] = false;
            state.pop();
        }
    }
}

/* 全排列 II */
fn permutations_ii(nums: &mut [i32]) -> Vec<Vec<i32>> {
    let mut res = Vec::new();
    backtrack(Vec::new(), nums, &mut vec![false; nums.len()], &mut res);
    res
}
```

假設元素兩兩之間互不相同，則 n 個元素共有 $n!$ 種排列（階乘）；在記錄結果時，需要複製長度為 n 的串列，使用 $O(n)$ 時間。**因此時間複雜度為 $O(n!n)$** 。

最大遞迴深度為 n ，使用 $O(n)$ 堆疊幀空間。`selected` 使用 $O(n)$ 空間。同一時刻最多共有 n 個 `duplicated` ，使用 $O(n^2)$ 空間。**因此空間複雜度為 $O(n^2)$** 。

3. 兩種剪枝對比

請注意，雖然 `selected` 和 `duplicated` 都用於剪枝，但兩者的目標不同。

- **重複選擇剪枝**：整個搜尋過程中只有一個 `selected` 。它記錄的是當前狀態中包含哪些元素，其作用是避免某個元素在 `state` 中重複出現。
- **相等元素剪枝**：每輪選擇（每個呼叫的 `backtrack` 函式）都包含一個 `duplicated` 。它記錄的是在本輪走訪（`for` 迴圈）中哪些元素已被選擇過，其作用是保證相等元素只被選擇一次。

圖 13-9 展示了兩個剪枝條件的生效範圍。注意，樹中的每個節點代表一個選擇，從根節點到葉節點的路徑上的各個節點構成一個排列。

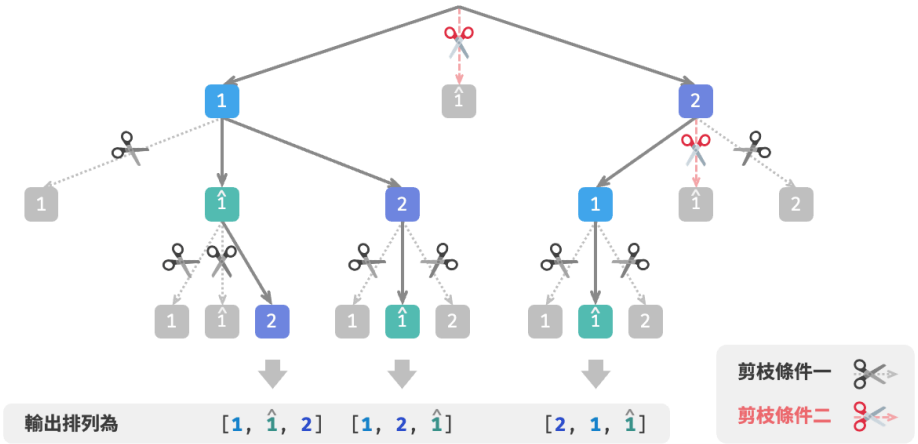


圖 13-9　兩種剪枝條件的作用範圍

13.3　子集和問題

13.3.1　無重複元素的情況

> **Question**
> 給定一個正整數陣列 `nums` 和一個目標正整數 `target`，請找出所有可能的組合，使得組合中的元素和等於 `target`。給定陣列無重複元素，每個元素可以被選取多次。請以串列形式返回這些組合，串列中不應包含重複組合。

例如，輸入集合 $\{3, 4, 5\}$ 和目標整數 9，解為 $\{3, 3, 3\}, \{4, 5\}$。需要注意以下兩點。

- 輸入集合中的元素可以被無限次重複選取。
- 子集不區分元素順序，比如 $\{4, 5\}$ 和 $\{5, 4\}$ 是同一個子集。

1.　參考全排列解法

類似於全排列問題，我們可以把子集的生成過程想象成一系列選擇的結果，並在選擇過程中實時更新“元素和”，當元素和等於 `target` 時，就將子集記錄至結果串列。

而與全排列問題不同的是，**本題集合中的元素可以被無限次選取**，因此無須藉助 `selected` 布林串列來記錄元素是否已被選擇。我們可以對全排列程式碼進行小幅修改，初步得到解題程式碼：

```rust
// === File: subset_sum_i_naive.rs ===

/* 回溯演算法：子集和 I */
fn backtrack(
    state: &mut Vec<i32>,
    target: i32,
    total: i32,
    choices: &[i32],
    res: &mut Vec<Vec<i32>>,
) {
    // 子集和等於 target 時，記錄解
    if total == target {
        res.push(state.clone());
        return;
    }
    // 走訪所有選擇
    for i in 0..choices.len() {
        // 剪枝：若子集和超過 target ，則跳過該選擇
        if total + choices[i] > target {
            continue;
        }
        // 嘗試：做出選擇，更新元素和 total
        state.push(choices[i]);
        // 進行下一輪選擇
```

```
        backtrack(state, target, total + choices[i], choices, res);
        // 回退：撤銷選擇，恢復到之前的狀態
        state.pop();
    }
}

/* 求解子集和 I（包含重複子集） */
fn subset_sum_i_naive(nums: &[i32], target: i32) -> Vec<Vec<i32>> {
    let mut state = Vec::new(); // 狀態（子集）
    let total = 0; // 子集和
    let mut res = Vec::new(); // 結果串列（子集串列）
    backtrack(&mut state, target, total, nums, &mut res);
    res
}
```

向以上程式碼輸入陣列 $[3, 4, 5]$ 和目標元素 9，輸出結果為 $[3, 3, 3], [4, 5], [5, 4]$。**雖然成功找出了所有和為 9 的子集，但其中存在重複的子集 $[4, 5]$ 和 $[5, 4]$**。

這是因為搜尋過程是區分選擇順序的，然而子集不區分選擇順序。如圖 13-10 所示，先選 4 後選 5 與先選 5 後選 4 是不同的分支，但對應同一個子集。

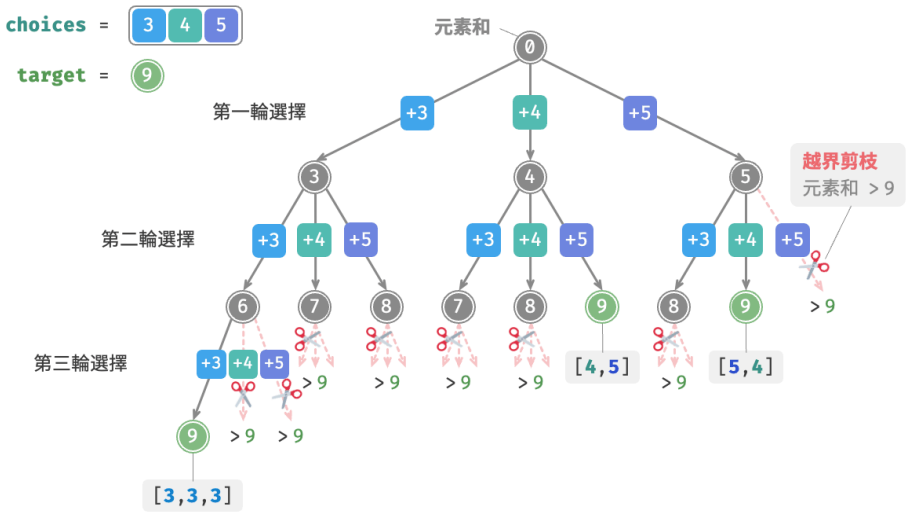


圖 13-10　子集搜尋與越界剪枝

為了去除重複子集，**一種直接的思路是對結果串列進行去重**。但這個方法效率很低，有兩方面原因。

- 當陣列元素較多，尤其是當 `target` 較大時，搜尋過程會產生大量的重複子集。
- 比較子集（陣列）的異同非常耗時，需要先排序陣列，再比較陣列中每個元素的異同。

2. 重複子集剪枝

我們考慮在搜尋過程中透過剪枝進行去重。觀察圖 13-11 ，重複子集是在以不同順序選擇陣列元素時產生的，例如以下情況。

1. 當第一輪和第二輪分別選擇 3 和 4 時，會生成包含這兩個元素的所有子集，記為 $[3, 4, \dots]$ 。
2. 之後，當第一輪選擇 4 時，**則第二輪應該跳過** 3 ，因為該選擇產生的子集 $[4, 3, \dots]$ 和第 `1.` 步中生成的子集完全重複。

在搜尋過程中，每一層的選擇都是從左到右被逐個嘗試的，因此越靠右的分支被剪掉的越多。

1. 前兩輪選擇 3 和 5 ，生成子集 $[3, 5, \dots]$ 。
2. 前兩輪選擇 4 和 5 ，生成子集 $[4, 5, \dots]$ 。
3. 若第一輪選擇 5 ，**則第二輪應該跳過** 3 **和** 4 ，因為子集 $[5, 3, \dots]$ 和 $[5, 4, \dots]$ 與第 `1.` 步和第 `2.` 步中描述的子集完全重複。

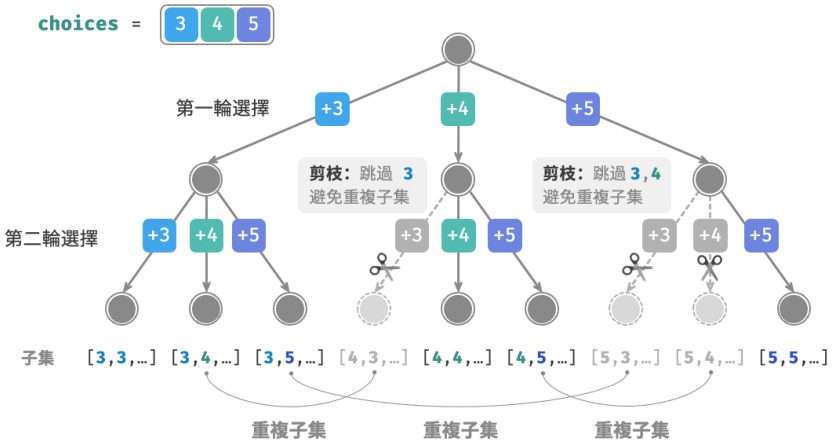


圖 13-11　不同選擇順序導致的重複子集

總結來看，給定輸入陣列 $[x_1, x_2, \dots, x_n]$ ，設搜尋過程中的選擇序列為 $[x_{i_1}, x_{i_2}, \dots, x_{i_m}]$ ，則該選擇序列需要滿足 $i_1 \leq i_2 \leq \cdots \leq i_m$ ，**不滿足該條件的選擇序列都會造成重複，應當剪枝**。

3. 程式碼實現

為實現該剪枝，我們初始化變數 `start` ，用於指示走訪起始點。**當做出選擇 x_i 後，設定下一輪從索引 i 開始走訪**。這樣做就可以讓選擇序列滿足 $i_1 \leq i_2 \leq \cdots \leq i_m$ ，從而保證子集唯一。

除此之外，我們還對程式碼進行了以下兩項最佳化。

- 在開啟搜尋前，先將陣列 `nums` 排序。在走訪所有選擇時，**當子集和超過 `target` 時直接結束迴圈**，因為後邊的元素更大，其子集和一定超過 `target` 。
- 省去元素和變數 `total` ，**透過在 `target` 上執行減法來統計元素和**，當 `target` 等於 0 時記錄解。

```rust
// === File: subset_sum_i.rs ===

/* 回溯演算法：子集和 I */
fn backtrack(
    state: &mut Vec<i32>,
    target: i32,
    choices: &[i32],
    start: usize,
    res: &mut Vec<Vec<i32>>,
) {
    // 子集和等於 target 時，記錄解
    if target == 0 {
        res.push(state.clone());
        return;
    }
    // 走訪所有選擇
    // 剪枝二：從 start 開始走訪，避免生成重複子集
    for i in start..choices.len() {
        // 剪枝一：若子集和超過 target ，則直接結束迴圈
        // 這是因為陣列已排序，後邊元素更大，子集和一定超過 target
        if target - choices[i] < 0 {
            break;
        }
        // 嘗試：做出選擇，更新 target, start
        state.push(choices[i]);
        // 進行下一輪選擇
        backtrack(state, target - choices[i], choices, i, res);
        // 回退：撤銷選擇，恢復到之前的狀態
        state.pop();
    }
}

/* 求解子集和 I */
fn subset_sum_i(nums: &mut [i32], target: i32) -> Vec<Vec<i32>> {
    let mut state = Vec::new(); // 狀態（子集）
    nums.sort(); // 對 nums 進行排序
    let start = 0; // 走訪起始點
    let mut res = Vec::new(); // 結果串列（子集串列）
    backtrack(&mut state, target, nums, start, &mut res);
    res
}
```

圖 13-12 所示為將陣列 $[3, 4, 5]$ 和目標元素 9 輸入以上程式碼後的整體回溯過程。

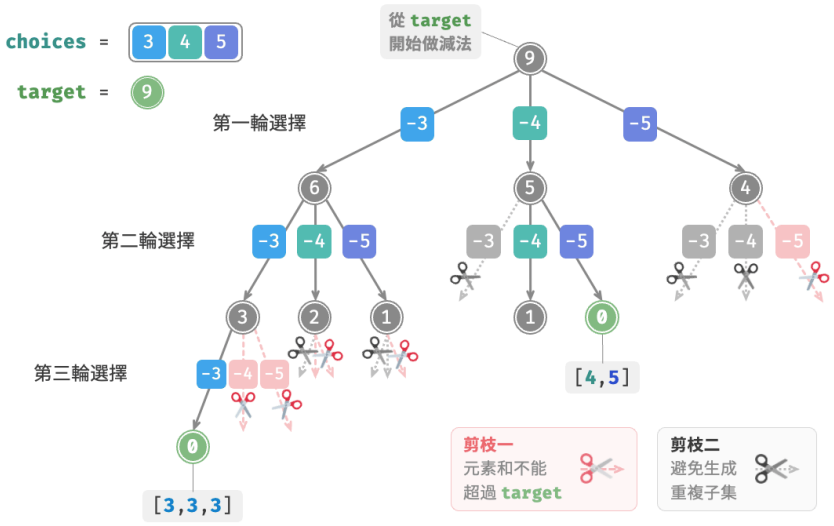


圖 13-12　子集和 I 回溯過程

13.3.2　考慮重複元素的情況

Question

給定一個正整數陣列 `nums` 和一個目標正整數 `target`，請找出所有可能的組合，使得組合中的元素和等於 `target`。**給定陣列可能包含重複元素，每個元素只可被選擇一次**。請以串列形式返回這些組合，串列中不應包含重複組合。

相比於上題，**本題的輸入陣列可能包含重複元素**，這引入了新的問題。例如，給定陣列 $[4, \hat{4}, 5]$ 和目標元素 9，則現有程式碼的輸出結果為 $[4, 5], [\hat{4}, 5]$，出現了重複子集。

造成這種重複的原因是相等元素在某輪中被多次選擇。在圖 13-13 中，第一輪共有三個選擇，其中兩個都為 4，會產生兩個重複的搜尋分支，從而輸出重複子集；同理，第二輪的兩個 4 也會產生重複子集。

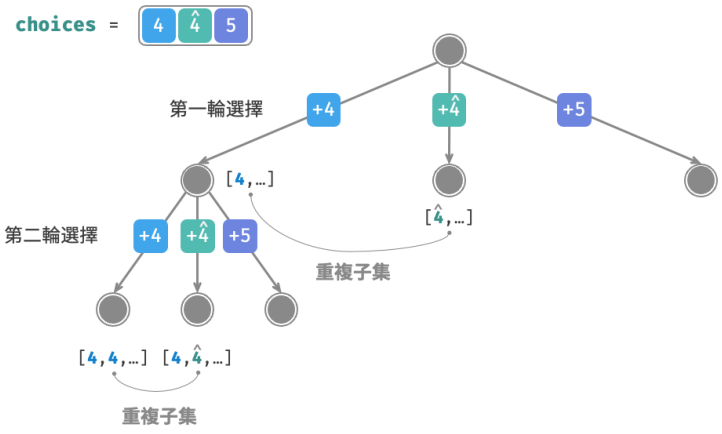


圖 13-13　相等元素導致的重複子集

1. 相等元素剪枝

為解決此問題，**我們需要限制相等元素在每一輪中只能被選擇一次**。實現方式比較巧妙：由於陣列是已排序的，因此相等元素都是相鄰的。這意味著在某輪選擇中，若當前元素與其左邊元素相等，則說明它已經被選擇過，因此直接跳過當前元素。

與此同時，**本題規定每個陣列元素只能被選擇一次**。幸運的是，我們也可以利用變數 `start` 來滿足該約束：當做出選擇 x_i 後，設定下一輪從索引 $i + 1$ 開始向後走訪。這樣既能去除重複子集，也能避免重複選擇元素。

2. 程式碼實現

```rust
// === File: subset_sum_ii.rs ===

/* 回溯演算法：子集和 II */
fn backtrack(
    state: &mut Vec<i32>,
    target: i32,
    choices: &[i32],
    start: usize,
    res: &mut Vec<Vec<i32>>,
) {
    // 子集和等於 target 時，記錄解
    if target == 0 {
        res.push(state.clone());
        return;
    }
    // 走訪所有選擇
    // 剪枝二：從 start 開始走訪，避免生成重複子集
    // 剪枝三：從 start 開始走訪，避免重複選擇同一元素
    for i in start..choices.len() {
        // 剪枝一：若子集和超過 target ，則直接結束迴圈
        // 這是因為陣列已排序，後邊元素更大，子集和一定超過 target
        if target - choices[i] < 0 {
            break;
        }
        // 剪枝四：如果該元素與左邊元素相等，說明該搜尋分支重複，直接跳過
        if i > start && choices[i] == choices[i - 1] {
            continue;
        }
        // 嘗試：做出選擇，更新 target, start
        state.push(choices[i]);
        // 進行下一輪選擇
        backtrack(state, target - choices[i], choices, i + 1, res);
```

```
        // 回退：撤銷選擇，恢復到之前的狀態
        state.pop();
    }
}

/* 求解子集和 II */
fn subset_sum_ii(nums: &mut [i32], target: i32) -> Vec<Vec<i32>> {
    let mut state = Vec::new(); // 狀態（子集）
    nums.sort(); // 對 nums 進行排序
    let start = 0; // 走訪起始點
    let mut res = Vec::new(); // 結果串列（子集串列）
    backtrack(&mut state, target, nums, start, &mut res);
    res
}
```

圖 13-14 展示了陣列 $[4, 4, 5]$ 和目標元素 9 的回溯過程，共包含四種剪枝操作。請你將圖示與程式碼註釋相結合，理解整個搜尋過程，以及每種剪枝操作是如何工作的。

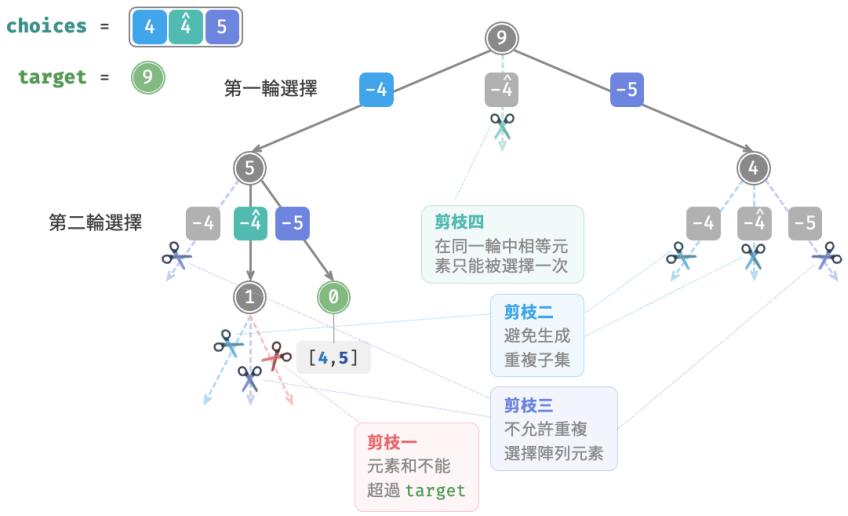


圖 13-14　子集和 II 回溯過程

13.4　n 皇后問題

Question

根據國際象棋的規則，皇后可以攻擊與同處一行、一列或一條斜線上的棋子。給定 n 個皇后和一個 $n \times n$ 大小的棋盤，尋找使得所有皇后之間無法相互攻擊的擺放方案。

如圖 13-15 所示，當 $n = 4$ 時，共可以找到兩個解。從回溯演算法的角度看，$n \times n$ 大小的棋盤共有 n^2 個格子，給出了所有的選擇 `choices`。在逐個放置皇后的過程中，棋盤狀態在不斷地變化，每個時刻的棋盤就是狀態 `state`。

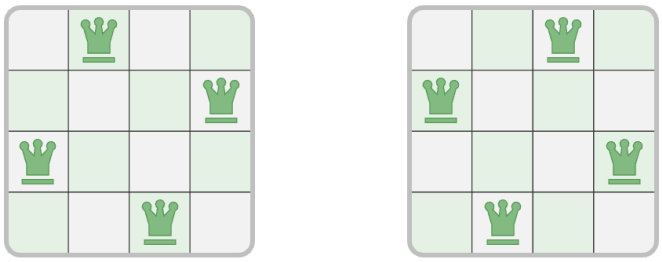


圖 13-15　4 皇后問題的解

圖 13-16 展示了本題的三個約束條件：**多個皇后不能在同一行、同一列、同一條對角線上**。值得注意的是，對角線分為主對角線 `\` 和次對角線 `/` 兩種。

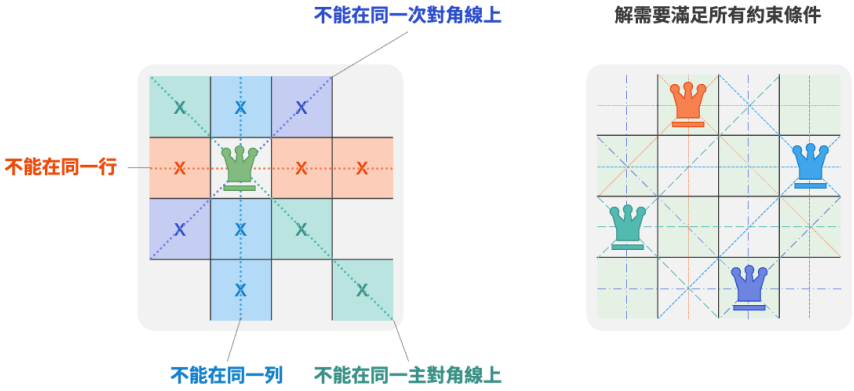


圖 13-16　n 皇后問題的約束條件

1. 逐行放置策略

皇后的數量和棋盤的行數都為 n，因此我們容易得到一個推論：**棋盤每行都允許且只允許放置一個皇后**。

也就是說，我們可以採取逐行放置策略：從第一行開始，在每行放置一個皇后，直至最後一行結束。

圖 13-17 所示為 4 皇后問題的逐行放置過程。受畫幅限制，圖 13-17 僅展開了第一行的其中一個搜尋分支，並且將不滿足列約束和對角線約束的方案都進行了剪枝。

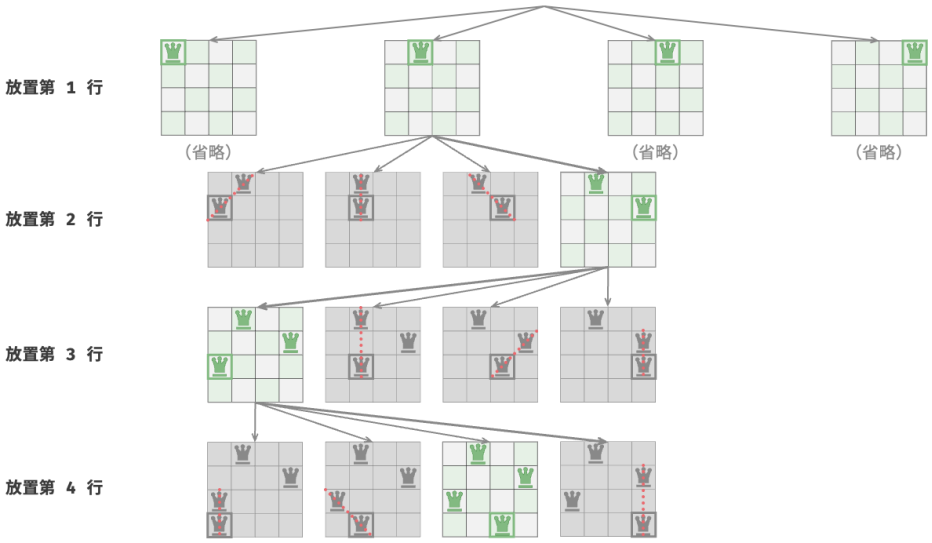


圖 13-17　逐行放置策略

從本質上看，**逐行放置策略起到了剪枝的作用**，它避免了同一行出現多個皇后的所有搜尋分支。

2. 列與對角線剪枝

為了滿足列約束，我們可以利用一個長度為 n 的布林型陣列 `cols` 記錄每一列是否有皇后。在每次決定放置前，我們透過 `cols` 將已有皇后的列進行剪枝，並在回溯中動態更新 `cols` 的狀態。

> **Tip**
> 請注意，矩陣的起點位於左上角，其中行索引從上到下增加，列索引從左到右增加。

那麼，如何處理對角線約束呢？設棋盤中某個格子的行列索引為 (row, col) ，選定矩陣中的某條主對角線，我們發現該對角線上所有格子的行索引減列索引都相等，**即主對角線上所有格子的 $row - col$ 為恆定值**。

也就是說，如果兩個格子滿足 $row_1 - col_1 = row_2 - col_2$ ，則它們一定處在同一條主對角線上。利用該規律，我們可以藉助圖 13-18 所示的陣列 `diags1` 記錄每條主對角線上是否有皇后。

同理，**次對角線上的所有格子的 $row + col$ 是恆定值**。我們同樣也可以藉助陣列 `diags2` 來處理次對角線約束。

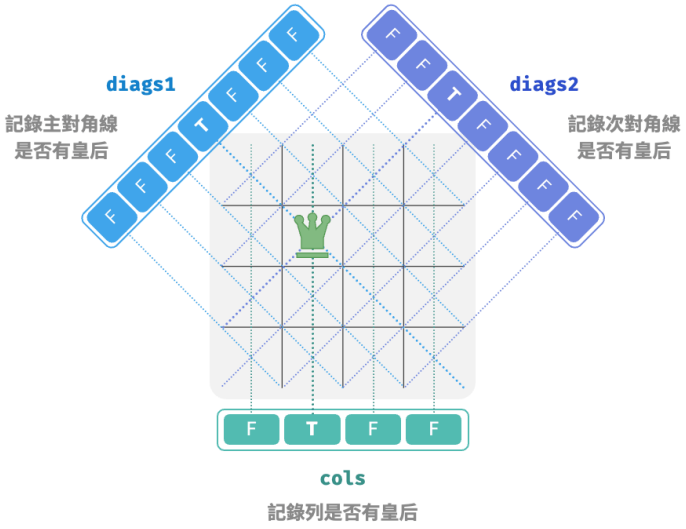


圖 13-18　處理列約束和對角線約束

3. 程式碼實現

請注意，n 維方陣中 $row - col$ 的範圍是 $[-n + 1, n - 1]$，$row + col$ 的範圍是 $[0, 2n - 2]$，所以主對角線和次對角線的數量都為 $2n - 1$，即陣列 `diags1` 和 `diags2` 的長度都為 $2n - 1$。

```rust
// === File: n_queens.rs ===

/* 回溯演算法：n 皇后 */
fn backtrack(
    row: usize,
    n: usize,
    state: &mut Vec<Vec<String>>,
    res: &mut Vec<Vec<Vec<String>>>,
    cols: &mut [bool],
    diags1: &mut [bool],
    diags2: &mut [bool],
) {
    // 當放置完所有行時，記錄解
    if row == n {
        res.push(state.clone());
        return;
    }
    // 走訪所有列
    for col in 0..n {
        // 計算該格子對應的主對角線和次對角線
        let diag1 = row + n - 1 - col;
        let diag2 = row + col;
        // 剪枝：不允許該格子所在列、主對角線、次對角線上存在皇后
```

```
        if !cols[col] && !diags1[diag1] && !diags2[diag2] {
            // 嘗試：將皇后放置在該格子
            state[row][col] = "Q".into();
            (cols[col], diags1[diag1], diags2[diag2]) = (true, true, true);
            // 放置下一行
            backtrack(row + 1, n, state, res, cols, diags1, diags2);
            // 回退：將該格子恢復為空位
            state[row][col] = "#".into();
            (cols[col], diags1[diag1], diags2[diag2]) = (false, false, false);
        }
    }
}

/* 求解 n 皇后 */
fn n_queens(n: usize) -> Vec<Vec<Vec<String>>> {
    // 初始化 n*n 大小的棋盤，其中 'Q' 代表皇后，'#' 代表空位
    let mut state: Vec<Vec<String>> = vec![vec!["#".to_string(); n]; n];
    let mut cols = vec![false; n]; // 記錄列是否有皇后
    let mut diags1 = vec![false; 2 * n - 1]; // 記錄主對角線上是否有皇后
    let mut diags2 = vec![false; 2 * n - 1]; // 記錄次對角線上是否有皇后
    let mut res: Vec<Vec<Vec<String>>> = Vec::new();

    backtrack(
        0,
        n,
        &mut state,
        &mut res,
        &mut cols,
        &mut diags1,
        &mut diags2,
    );

    res
}
```

逐行放置 n 次，考慮列約束，則從第一行到最後一行分別有 n、$n-1$、…、2、1 個選擇，使用 $O(n!)$ 時間。當記錄解時，需要複製矩陣 `state` 並新增進 `res`，複製操作使用 $O(n^2)$ 時間。因此，**總體時間複雜度為** $O(n! \cdot n^2)$。實際上，根據對角線約束的剪枝也能夠大幅縮小搜尋空間，因而搜尋效率往往優於以上時間複雜度。

陣列 `state` 使用 $O(n^2)$ 空間，陣列 `cols`、`diags1` 和 `diags2` 皆使用 $O(n)$ 空間。最大遞迴深度為 n，使用 $O(n)$ 堆疊幀空間。因此，**空間複雜度為** $O(n^2)$。

13.5 小結

1. 重點回顧

- 回溯演算法本質是窮舉法，透過對解空間進行深度優先走訪來尋找符合條件的解。在搜尋過程中，遇到滿足條件的解則記錄，直至找到所有解或走訪完成後結束。
- 回溯演算法的搜尋過程包括嘗試與回退兩個部分。它透過深度優先搜尋來嘗試各種選擇，當遇到不滿足約束條件的情況時，則撤銷上一步的選擇，退回到之前的狀態，並繼續嘗試其他選擇。嘗試與回退是兩個方向相反的操作。
- 回溯問題通常包含多個約束條件，它們可用於實現剪枝操作。剪枝可以提前結束不必要的搜尋分支，大幅提升搜尋效率。
- 回溯演算法主要可用於解決搜尋問題和約束滿足問題。組合最佳化問題雖然可以用回溯演算法解決，但往往存在效率更高或效果更好的解法。
- 全排列問題旨在搜尋給定集合元素的所有可能的排列。我們藉助一個陣列來記錄每個元素是否被選擇，剪掉重複選擇同一元素的搜尋分支，確保每個元素只被選擇一次。
- 在全排列問題中，如果集合中存在重複元素，則最終結果會出現重複排列。我們需要約束相等元素在每輪中只能被選擇一次，這通常藉助一個雜湊集合來實現。
- 子集和問題的目標是在給定集合中找到和為目標值的所有子集。集合不區分元素順序，而搜尋過程會輸出所有順序的結果，產生重複子集。我們在回溯前將資料進行排序，並設定一個變數來指示每一輪的走訪起始點，從而將生成重複子集的搜尋分支進行剪枝。
- 對於子集和問題，陣列中的相等元素會產生重複集合。我們利用陣列已排序的前置條件，透過判斷相鄰元素是否相等實現剪枝，從而確保相等元素在每輪中只能被選中一次。
- n 皇后問題旨在尋找將 n 個皇后放置到 $n \times n$ 尺寸棋盤上的方案，要求所有皇后兩兩之間無法攻擊對方。該問題的約束條件有行約束、列約束、主對角線和次對角線約束。為滿足行約束，我們採用按行放置的策略，保證每一行放置一個皇后。
- 列約束和對角線約束的處理方式類似。對於列約束，我們利用一個陣列來記錄每一列是否有皇后，從而指示選中的格子是否合法。對於對角線約束，我們藉助兩個陣列來分別記錄該主、次對角線上是否存在皇后；難點在於找處在到同一主（副）對角線上格子滿足的行列索引規律。

2. Q & A

Q：怎麼理解回溯和遞迴的關係？

總的來看，回溯是一種“演算法策略”，而遞迴更像是一個“工具”。

- 回溯演算法通常基於遞迴實現。然而，回溯是遞迴的應用場景之一，是遞迴在搜尋問題中的應用。
- 遞迴的結構體現了“子問題分解”的解題範式，常用於解決分治、回溯、動態規劃（記憶化遞迴）等問題。

第 14 章　動態規劃

Abstract

小溪匯入河流，江河匯入大海。

動態規劃將小問題的解彙集成大問題的答案，一步步引領我們走向解決問題的彼岸。

14.1　初探動態規劃

動態規劃（dynamic programming）是一個重要的演算法範式，它將一個問題分解為一系列更小的子問題，並透過儲存子問題的解來避免重複計算，從而大幅提升時間效率。

在本節中，我們從一個經典例題入手，先給出它的暴力回溯解法，觀察其中包含的重疊子問題，再逐步導出更高效的動態規劃解法。

> **爬樓梯**
> 給定一個共有 n 階的樓梯，你每步可以上 1 階或者 2 階，請問有多少種方案可以爬到樓頂？

如圖 14-1 所示，對於一個 3 階樓梯，共有 3 種方案可以爬到樓頂。

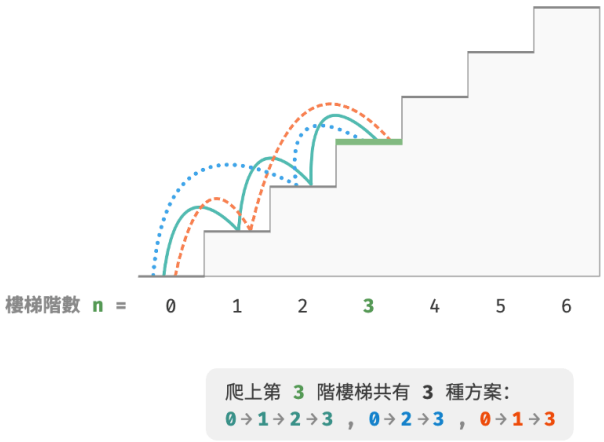


圖 14-1　爬到第 3 階的方案數量

本題的目標是求解方案數量，**我們可以考慮透過回溯來窮舉所有可能性**。具體來說，將爬樓梯想象為一個多輪選擇的過程：從地面出發，每輪選擇上 1 階或 2 階，每當到達樓梯頂部時就將方案數量加 1 ，當越過樓梯頂部時就將其剪枝。程式碼如下所示：

```
// === File: climbing_stairs_backtrack.rs ===

/* 回溯 */
fn backtrack(choices: &[i32], state: i32, n: i32, res: &mut [i32]) {
    // 當爬到第 n 階時，方案數量加 1
    if state == n {
        res[0] = res[0] + 1;
    }
    // 走訪所有選擇
    for &choice in choices {
        // 剪枝：不允許越過第 n 階
        if state + choice > n {
```

```
            continue;
        }
        // 嘗試：做出選擇，更新狀態
        backtrack(choices, state + choice, n, res);
        // 回退
    }
}

/* 爬樓梯：回溯 */
fn climbing_stairs_backtrack(n: usize) -> i32 {
    let choices = vec![1, 2]; // 可選擇向上爬 1 階或 2 階
    let state = 0; // 從第 0 階開始爬
    let mut res = Vec::new();
    res.push(0); // 使用 res[0] 記錄方案數量
    backtrack(&choices, state, n as i32, &mut res);
    res[0]
}
```

14.1.1　方法一：暴力搜尋

回溯演算法通常並不顯式地對問題進行拆解，而是將求解問題看作一系列決策步驟，透過試探和剪枝，搜尋所有可能的解。

我們可以嘗試從問題分解的角度分析這道題。設爬到第 i 階共有 $dp[i]$ 種方案，那麼 $dp[i]$ 就是原問題，其子問題包括：

$$dp[i-1], dp[i-2], \dots, dp[2], dp[1]$$

由於每輪只能上 1 階或 2 階，因此當我們站在第 i 階樓梯上時，上一輪只可能站在第 $i-1$ 階或第 $i-2$ 階上。換句話說，我們只能從第 $i-1$ 階或第 $i-2$ 階邁向第 i 階。

由此便可得出一個重要推論：**爬到第 $i-1$ 階的方案數加上爬到第 $i-2$ 階的方案數就等於爬到第 i 階的方案數**。公式如下：

$$dp[i] = dp[i-1] + dp[i-2]$$

這意味著在爬樓梯問題中，各個子問題之間存在遞推關係，**原問題的解可以由子問題的解構建得來**。圖 14-2 展示了該遞推關係。

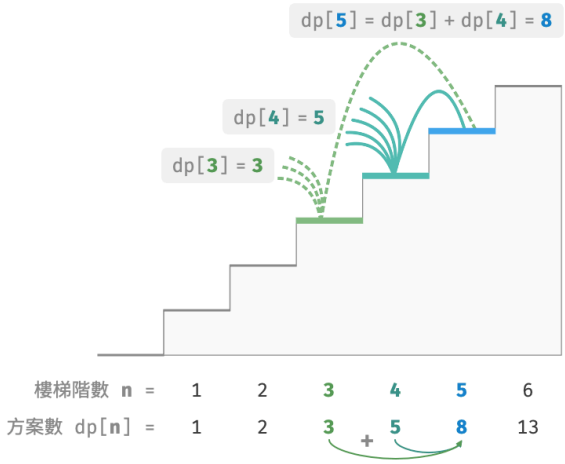


圖 14-2　方案數量遞推關係

我們可以根據遞推公式得到暴力搜尋解法。以 $dp[n]$ 為起始點，**遞迴地將一個較大問題拆解為兩個較小問題的和**，直至到達最小子問題 $dp[1]$ 和 $dp[2]$ 時返回。其中，最小子問題的解是已知的，即 $dp[1] = 1$、$dp[2] = 2$，表示爬到第 1、2 階分別有 1、2 種方案。

觀察以下程式碼，它和標準回溯程式碼都屬於深度優先搜尋，但更加簡潔：

```
// === File: climbing_stairs_dfs.rs ===

/* 搜尋 */
fn dfs(i: usize) -> i32 {
    // 已知 dp[1] 和 dp[2] ，返回之
    if i == 1 || i == 2 {
        return i as i32;
    }
    // dp[i] = dp[i-1] + dp[i-2]
    let count = dfs(i - 1) + dfs(i - 2);
    count
}

/* 爬樓梯：搜尋 */
fn climbing_stairs_dfs(n: usize) -> i32 {
    dfs(n)
}
```

圖 14-3 展示了暴力搜尋形成的遞迴樹。對於問題 $dp[n]$，其遞迴樹的深度為 n，時間複雜度為 $O(2^n)$。指數階屬於爆炸式增長，如果我們輸入一個比較大的 n，則會陷入漫長的等待之中。

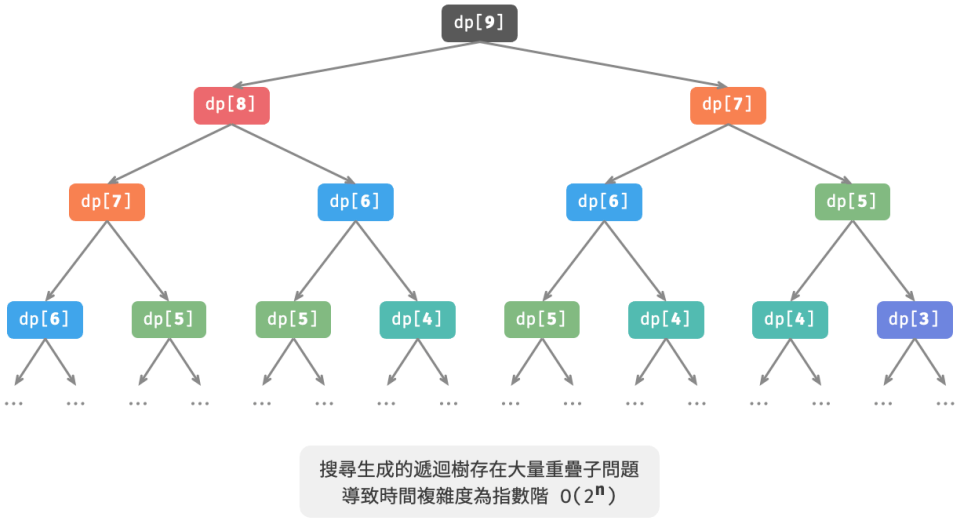


圖 14-3　爬樓梯對應遞迴樹

觀察圖 14-3，**指數階的時間複雜度是“重疊子問題”導致的**。例如 $dp[9]$ 被分解為 $dp[8]$ 和 $dp[7]$，$dp[8]$ 被分解為 $dp[7]$ 和 $dp[6]$，兩者都包含子問題 $dp[7]$。

以此類推，子問題中包含更小的重疊子問題，子子孫孫無窮盡也。絕大部分計算資源都浪費在這些重疊的子問題上。

14.1.2　方法二：記憶化搜尋

為了提升演算法效率，**我們希望所有的重疊子問題都只被計算一次**。為此，我們宣告一個陣列 `mem` 來記錄每個子問題的解，並在搜尋過程中將重疊子問題剪枝。

1. 當首次計算 $dp[i]$ 時，我們將其記錄至 `mem[i]`，以便之後使用。
2. 當再次需要計算 $dp[i]$ 時，我們便可直接從 `mem[i]` 中獲取結果，從而避免重複計算該子問題。

程式碼如下所示：

```rust
// === File: climbing_stairs_dfs_mem.rs ===

/* 記憶化搜尋 */
fn dfs(i: usize, mem: &mut [i32]) -> i32 {
    // 已知 dp[1] 和 dp[2] ，返回之
    if i == 1 || i == 2 {
        return i as i32;
    }
    // 若存在記錄 dp[i] ，則直接返回之
    if mem[i] != -1 {
        return mem[i];
    }
```

```
    // dp[i] = dp[i-1] + dp[i-2]
    let count = dfs(i - 1, mem) + dfs(i - 2, mem);
    // 記錄 dp[i]
    mem[i] = count;
    count
}

/* 爬樓梯：記憶化搜尋 */
fn climbing_stairs_dfs_mem(n: usize) -> i32 {
    // mem[i] 記錄爬到第 i 階的方案總數，-1 代表無記錄
    let mut mem = vec![-1; n + 1];
    dfs(n, &mut mem)
}
```

觀察圖 14-4，**經過記憶化處理後，所有重疊子問題都只需計算一次，時間複雜度最佳化至 $O(n)$**，這是一個巨大的飛躍。

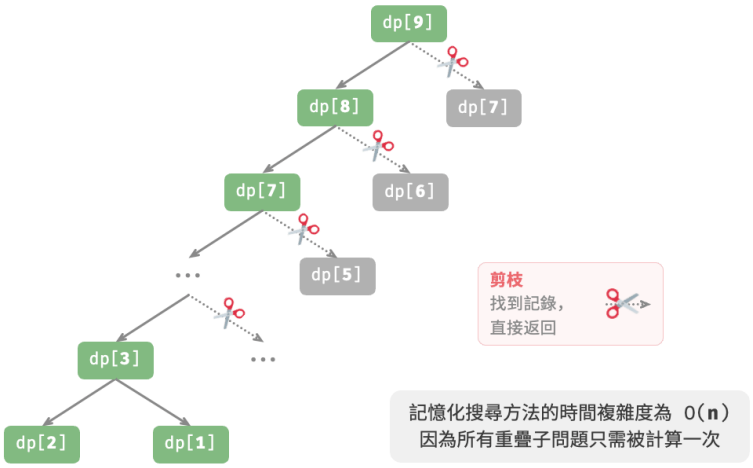


圖 14-4　記憶化搜尋對應遞迴樹

14.1.3　方法三：動態規劃

記憶化搜尋是一種“從頂至底”的方法：我們從原問題（根節點）開始，遞迴地將較大子問題分解為較小子問題，直至解已知的最小子問題（葉節點）。之後，透過回溯逐層收集子問題的解，構建出原問題的解。

與之相反，**動態規劃是一種“從底至頂”的方法**：從最小子問題的解開始，迭代地構建更大子問題的解，直至得到原問題的解。

由於動態規劃不包含回溯過程，因此只需使用迴圈迭代實現，無須使用遞迴。在以下程式碼中，我們初始化一個陣列 `dp` 來儲存子問題的解，它起到了與記憶化搜尋中陣列 `mem` 相同的記錄作用：

```rust
// === File: climbing_stairs_dp.rs ===

/* 爬樓梯：動態規劃 */
fn climbing_stairs_dp(n: usize) -> i32 {
    // 已知 dp[1] 和 dp[2] ，返回之
    if n == 1 || n == 2 {
        return n as i32;
    }
    // 初始化 dp 表，用於儲存子問題的解
    let mut dp = vec![-1; n + 1];
    // 初始狀態：預設最小子問題的解
    dp[1] = 1;
    dp[2] = 2;
    // 狀態轉移：從較小子問題逐步求解較大子問題
    for i in 3..=n {
        dp[i] = dp[i - 1] + dp[i - 2];
    }
    dp[n]
}
```

圖 14-5 模擬了以上程式碼的執行過程。

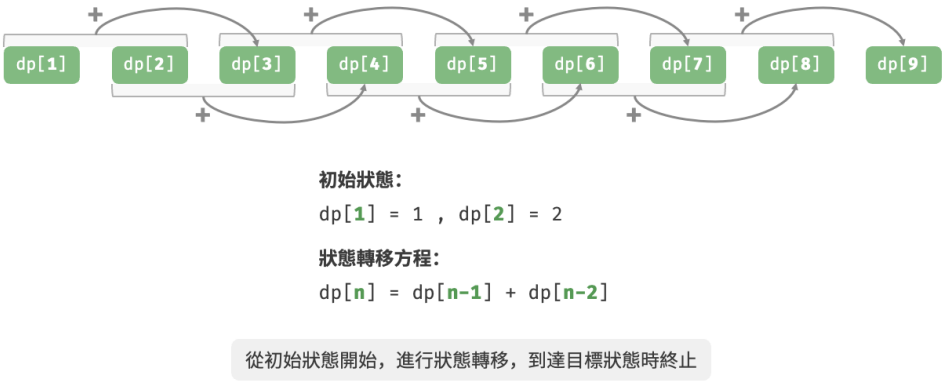


圖 14-5　爬樓梯的動態規劃過程

與回溯演算法一樣，動態規劃也使用“狀態”概念來表示問題求解的特定階段，每個狀態都對應一個子問題以及相應的區域性最優解。例如，爬樓梯問題的狀態定義為當前所在樓梯階數 i 。

根據以上內容，我們可以總結出動態規劃的常用術語。

- 將陣列 dp 稱為 dp 表，$dp[i]$ 表示狀態 i 對應子問題的解。
- 將最小子問題對應的狀態（第 1 階和第 2 階樓梯）稱為**初始狀態**。
- 將遞推公式 $dp[i] = dp[i-1] + dp[i-2]$ 稱為**狀態轉移方程**。

14.1.4　空間最佳化

細心的讀者可能發現了，**由於 $dp[i]$ 只與 $dp[i-1]$ 和 $dp[i-2]$ 有關，因此我們無須使用一個陣列 `dp` 來儲存所有子問題的解**，而只需兩個變數滾動前進即可。程式碼如下所示：

```rust
// === File: climbing_stairs_dp.rs ===

/* 爬樓梯：空間最佳化後的動態規劃 */
fn climbing_stairs_dp_comp(n: usize) -> i32 {
    if n == 1 || n == 2 {
        return n as i32;
    }
    let (mut a, mut b) = (1, 2);
    for _ in 3..=n {
        let tmp = b;
        b = a + b;
        a = tmp;
    }
    b
}
```

觀察以上程式碼，由於省去了陣列 `dp` 佔用的空間，因此空間複雜度從 $O(n)$ 降至 $O(1)$ 。

在動態規劃問題中，當前狀態往往僅與前面有限個狀態有關，這時我們可以只保留必要的狀態，透過“降維”來節省記憶體空間。**這種空間最佳化技巧被稱為“滾動變數”或“滾動陣列”**。

14.2　動態規劃問題特性

在上一節中，我們學習了動態規劃是如何透過子問題分解來求解原問題的。實際上，子問題分解是一種通用的演算法思路，在分治、動態規劃、回溯中的側重點不同。

- 分治演算法遞迴地將原問題劃分為多個相互獨立的子問題，直至最小子問題，並在回溯中合併子問題的解，最終得到原問題的解。
- 動態規劃也對問題進行遞迴分解，但與分治演算法的主要區別是，動態規劃中的子問題是相互依賴的，在分解過程中會出現許多重疊子問題。
- 回溯演算法在嘗試和回退中窮舉所有可能的解，並透過剪枝避免不必要的搜尋分支。原問題的解由一系列決策步驟構成，我們可以將每個決策步驟之前的子序列看作一個子問題。

實際上，動態規劃常用來求解最最佳化問題，它們不僅包含重疊子問題，還具有另外兩大特性：最優子結構、無後效性。

14.2.1　最優子結構

我們對爬樓梯問題稍作改動，使之更加適合展示最優子結構概念。

爬樓梯最小代價

給定一個樓梯，你每步可以上 1 階或者 2 階，每一階樓梯上都貼有一個非負整數，表示你在該臺階所需要付出的代價。給定一個非負整數陣列 $cost$ ，其中 $cost[i]$ 表示在第 i 個臺階需要付出的代價，$cost[0]$ 為地面（起始點）。請計算最少需要付出多少代價才能到達頂部？

如圖 14-6 所示，若第 1、2、3 階的代價分別為 1、10、1 ，則從地面爬到第 3 階的最小代價為 2 。

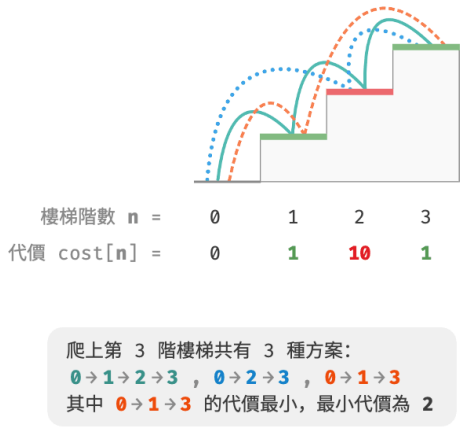


圖 14-6　爬到第 3 階的最小代價

設 $dp[i]$ 為爬到第 i 階累計付出的代價，由於第 i 階只可能從 $i-1$ 階或 $i-2$ 階走來，因此 $dp[i]$ 只可能等於 $dp[i-1]+cost[i]$ 或 $dp[i-2]+cost[i]$ 。為了儘可能減少代價，我們應該選擇兩者中較小的那一個：

$$dp[i] = \min(dp[i-1], dp[i-2]) + cost[i]$$

這便可以引出最優子結構的含義：**原問題的最優解是從子問題的最優解構建得來的**。

本題顯然具有最優子結構：我們從兩個子問題最優解 $dp[i-1]$ 和 $dp[i-2]$ 中挑選出較優的那一個，並用它構建出原問題 $dp[i]$ 的最優解。

那麼，上一節的爬樓梯題目有沒有最優子結構呢？它的目標是求解方案數量，看似是一個計數問題，但如果換一種問法：“求解最大方案數量”。我們意外地發現，**雖然題目修改前後是等價的，但最優子結構浮現出來了**：第 n 階最大方案數量等於第 $n-1$ 階和第 $n-2$ 階最大方案數量之和。所以說，最優子結構的解釋方式比較靈活，在不同問題中會有不同的含義。

根據狀態轉移方程，以及初始狀態 $dp[1]=cost[1]$ 和 $dp[2]=cost[2]$ ，我們就可以得到動態規劃程式碼：

```
// === File: min_cost_climbing_stairs_dp.rs ===

/* 爬樓梯最小代價：動態規劃 */
fn min_cost_climbing_stairs_dp(cost: &[i32]) -> i32 {
```

```rust
    let n = cost.len() - 1;
    if n == 1 || n == 2 {
        return cost[n];
    }
    // 初始化 dp 表，用於儲存子問題的解
    let mut dp = vec![-1; n + 1];
    // 初始狀態：預設最小子問題的解
    dp[1] = cost[1];
    dp[2] = cost[2];
    // 狀態轉移：從較小子問題逐步求解較大子問題
    for i in 3..=n {
        dp[i] = cmp::min(dp[i - 1], dp[i - 2]) + cost[i];
    }
    dp[n]
}
```

圖 14-7 展示了以上程式碼的動態規劃過程。

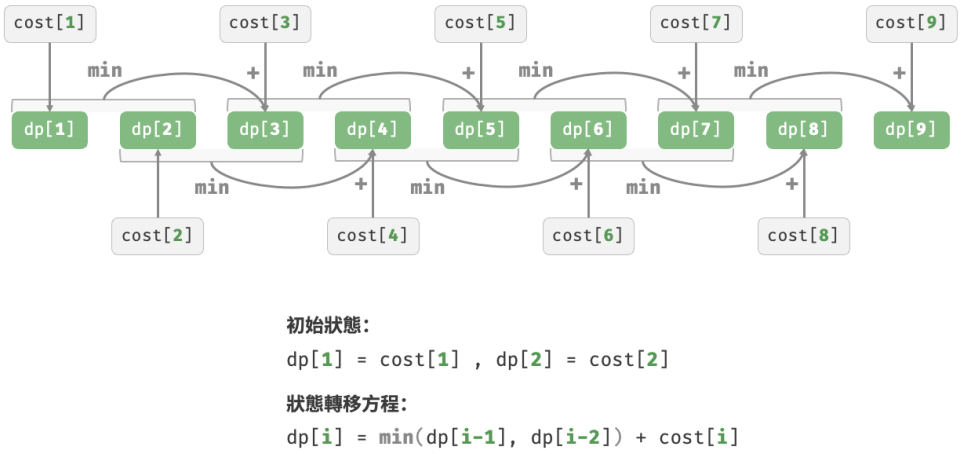


圖 14-7　爬樓梯最小代價的動態規劃過程

本題也可以進行空間最佳化，將一維壓縮至零維，使得空間複雜度從 $O(n)$ 降至 $O(1)$ ：

```rust
// === File: min_cost_climbing_stairs_dp.rs ===

/* 爬樓梯最小代價：空間最佳化後的動態規劃 */
fn min_cost_climbing_stairs_dp_comp(cost: &[i32]) -> i32 {
    let n = cost.len() - 1;
    if n == 1 || n == 2 {
        return cost[n];
    };
    let (mut a, mut b) = (cost[1], cost[2]);
```

```
    for i in 3..=n {
        let tmp = b;
        b = cmp::min(a, tmp) + cost[i];
        a = tmp;
    }
    b
}
```

14.2.2　無後效性

無後效性是動態規劃能夠有效解決問題的重要特性之一，其定義為：**給定一個確定的狀態，它的未來發展只與當前狀態有關，而與過去經歷的所有狀態無關**。

以爬樓梯問題為例，給定狀態 i，它會發展出狀態 $i+1$ 和狀態 $i+2$，分別對應跳 1 步和跳 2 步。在做出這兩種選擇時，我們無須考慮狀態 i 之前的狀態，它們對狀態 i 的未來沒有影響。

然而，如果我們給爬樓梯問題新增一個約束，情況就不一樣了。

> **帶約束爬樓梯**
> 給定一個共有 n 階的樓梯，你每步可以上 1 階或者 2 階，**但不能連續兩輪跳 1 階**，請問有多少種方案可以爬到樓頂？

如圖 14-8 所示，爬上第 3 階僅剩 2 種可行方案，其中連續三次跳 1 階的方案不滿足約束條件，因此被捨棄。

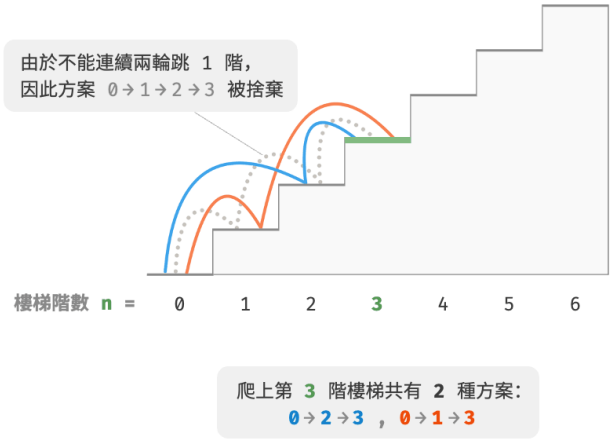


圖 14-8　帶約束爬到第 3 階的方案數量

在該問題中，如果上一輪是跳 1 階上來的，那麼下一輪就必須跳 2 階。這意味著，**下一步選擇不能由當前狀態（當前所在樓梯階數）獨立決定，還和前一個狀態（上一輪所在樓梯階數）有關**。

不難發現，此問題已不滿足無後效性，狀態轉移方程 $dp[i] = dp[i-1] + dp[i-2]$ 也失效了，因為 $dp[i-1]$ 代表本輪跳 1 階，但其中包含了許多“上一輪是跳 1 階上來的”方案，而為了滿足約束，我們就不能將 $dp[i-1]$ 直接計入 $dp[i]$ 中。

為此，我們需要擴展狀態定義：**狀態 $[i, j]$ 表示處在第 i 階並且上一輪跳了 j 階**，其中 $j \in \{1, 2\}$ 。此狀態定義有效地區分了上一輪跳了 1 階還是 2 階，我們可以據此判斷當前狀態是從何而來的。

- 當上一輪跳了 1 階時，上上一輪只能選擇跳 2 階，即 $dp[i, 1]$ 只能從 $dp[i-1, 2]$ 轉移過來。
- 當上一輪跳了 2 階時，上上一輪可選擇跳 1 階或跳 2 階，即 $dp[i, 2]$ 可以從 $dp[i-2, 1]$ 或 $dp[i-2, 2]$ 轉移過來。

如圖 14-9 所示，在該定義下，$dp[i, j]$ 表示狀態 $[i, j]$ 對應的方案數。此時狀態轉移方程為：

$$
\begin{cases}
dp[i, 1] = dp[i-1, 2] \\
dp[i, 2] = dp[i-2, 1] + dp[i-2, 2]
\end{cases}
$$

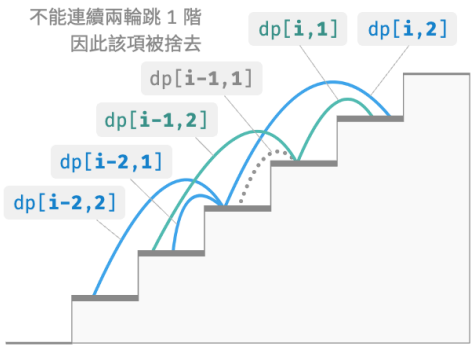


圖 14-9　考慮約束下的遞推關係

最終，返回 $dp[n, 1] + dp[n, 2]$ 即可，兩者之和代表爬到第 n 階的方案總數：

```rust
// === File: climbing_stairs_constraint_dp.rs ===

/* 帶約束爬樓梯：動態規劃 */
fn climbing_stairs_constraint_dp(n: usize) -> i32 {
    if n == 1 || n == 2 {
        return 1;
    };
    // 初始化 dp 表，用於儲存子問題的解
    let mut dp = vec![vec![-1; 3]; n + 1];
    // 初始狀態：預設最小子問題的解
    dp[1][1] = 1;
    dp[1][2] = 0;
    dp[2][1] = 0;
    dp[2][2] = 1;
    // 狀態轉移：從較小子問題逐步求解較大子問題
    for i in 3..=n {
        dp[i][1] = dp[i - 1][2];
        dp[i][2] = dp[i - 2][1] + dp[i - 2][2];
```

```
    }
    dp[n][1] + dp[n][2]
}
```

在上面的案例中，由於僅需多考慮前面一個狀態，因此我們仍然可以透過擴展狀態定義，使得問題重新滿足無後效性。然而，某些問題具有非常嚴重的“有後效性”。

爬樓梯與障礙生成
給定一個共有 n 階的樓梯，你每步可以上 1 階或者 2 階。**規定當爬到第 i 階時，系統自動會在第 $2i$ 階上放上障礙物，之後所有輪都不允許跳到第 $2i$ 階上**。例如，前兩輪分別跳到了第 2、3 階上，則之後就不能跳到第 4、6 階上。請問有多少種方案可以爬到樓頂?

在這個問題中，下次跳躍依賴過去所有的狀態，因為每一次跳躍都會在更高的階梯上設定障礙，並影響未來的跳躍。對於這類問題，動態規劃往往難以解決。

實際上，許多複雜的組合最佳化問題（例如旅行商問題）不滿足無後效性。對於這類問題，我們通常會選擇使用其他方法，例如啟發式搜尋、遺傳演算法、強化學習等，從而在有限時間內得到可用的區域性最優解。

14.3　動態規劃解題思路

上兩節介紹了動態規劃問題的主要特徵，接下來我們一起探究兩個更加實用的問題。

1. 如何判斷一個問題是不是動態規劃問題?
2. 求解動態規劃問題該從何處入手，完整步驟是什麼?

14.3.1　問題判斷

總的來說，如果一個問題包含重疊子問題、最優子結構，並滿足無後效性，那麼它通常適合用動態規劃求解。然而，我們很難從問題描述中直接提取出這些特性。因此我們通常會放寬條件，**先觀察問題是否適合使用回溯（窮舉）解決**。

適合用回溯解決的問題通常滿足“決策樹模型”，這種問題可以使用樹形結構來描述，其中每一個節點代表一個決策，每一條路徑代表一個決策序列。

換句話說，如果問題包含明確的決策概念，並且解是透過一系列決策產生的，那麼它就滿足決策樹模型，通常可以使用回溯來解決。

在此基礎上，動態規劃問題還有一些判斷的“加分項”。

- 問題包含最大（小）或最多（少）等最最佳化描述。
- 問題的狀態能夠使用一個串列、多維矩陣或樹來表示，並且一個狀態與其周圍的狀態存在遞推關係。

相應地，也存在一些“減分項”。

- 問題的目標是找出所有可能的解決方案，而不是找出最優解。
- 問題描述中有明顯的排列組合的特徵，需要返回具體的多個方案。

如果一個問題滿足決策樹模型，並具有較為明顯的“加分項”，我們就可以假設它是一個動態規劃問題，並在求解過程中驗證它。

14.3.2　問題求解步驟

動態規劃的解題流程會因問題的性質和難度而有所不同，但通常遵循以下步驟：描述決策，定義狀態，建立 dp 表，推導狀態轉移方程，確定邊界條件等。

為了更形象地展示解題步驟，我們使用一個經典問題“最小路徑和”來舉例。

> **Question**
> 給定一個 $n \times m$ 的二維網格 `grid`，網格中的每個單元格包含一個非負整數，表示該單元格的代價。機器人以左上角單元格為起始點，每次只能向下或者向右移動一步，直至到達右下角單元格。請返回從左上角到右下角的最小路徑和。

圖 14-10 展示了一個例子，給定網格的最小路徑和為 13 。

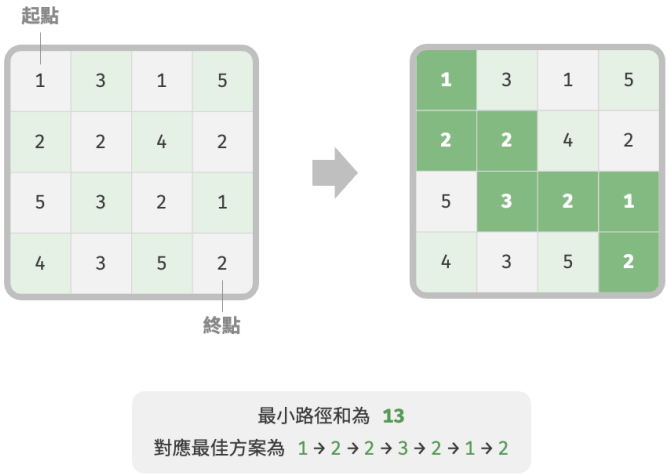


圖 14-10　最小路徑和示例資料

第一步：思考每輪的決策，定義狀態，從而得到 dp 表

本題的每一輪的決策就是從當前格子向下或向右走一步。設當前格子的行列索引為 $[i, j]$ ，則向下或向右走一步後，索引變為 $[i+1, j]$ 或 $[i, j+1]$ 。因此，狀態應包含行索引和列索引兩個變數，記為 $[i, j]$ 。

狀態 $[i, j]$ 對應的子問題為：從起始點 $[0, 0]$ 走到 $[i, j]$ 的最小路徑和，解記為 $dp[i, j]$ 。

至此，我們就得到了圖 14-11 所示的二維 dp 矩陣，其尺寸與輸入網格 $grid$ 相同。

圖 14-11　狀態定義與 dp 表

Note

動態規劃和回溯過程可以描述為一個決策序列，而狀態由所有決策變數構成。它應當包含描述解題進度的所有變數，其包含了足夠的資訊，能夠用來推導出下一個狀態。

每個狀態都對應一個子問題，我們會定義一個 dp 表來儲存所有子問題的解，狀態的每個獨立變數都是 dp 表的一個維度。從本質上看，dp 表是狀態和子問題的解之間的對映。

第二步：找出最優子結構，進而推導出狀態轉移方程

對於狀態 $[i, j]$ ，它只能從上邊格子 $[i-1, j]$ 和左邊格子 $[i, j-1]$ 轉移而來。因此最優子結構為：到達 $[i, j]$ 的最小路徑和由 $[i, j-1]$ 的最小路徑和與 $[i-1, j]$ 的最小路徑和中較小的那一個決定。

根據以上分析，可推出圖 14-12 所示的狀態轉移方程：

$$dp[i, j] = \min(dp[i-1, j], dp[i, j-1]) + grid[i, j]$$

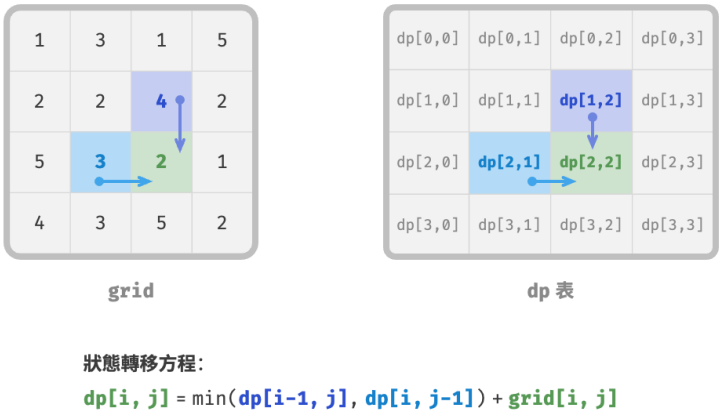


圖 14-12　最優子結構與狀態轉移方程

> **Note**
> 根據定義好的 dp 表，思考原問題和子問題的關係，找出透過子問題的最優解來構造原問題的最優解的方法，即最優子結構。
> 一旦我們找到了最優子結構，就可以使用它來構建出狀態轉移方程。

第三步：確定邊界條件和狀態轉移順序

在本題中，處在首行的狀態只能從其左邊的狀態得來，處在首列的狀態只能從其上邊的狀態得來，因此首行 $i = 0$ 和首列 $j = 0$ 是邊界條件。

如圖 14-13 所示，由於每個格子是由其左方格子和上方格子轉移而來，因此我們使用迴圈來走訪矩陣，外迴圈走訪各行，內迴圈走訪各列。

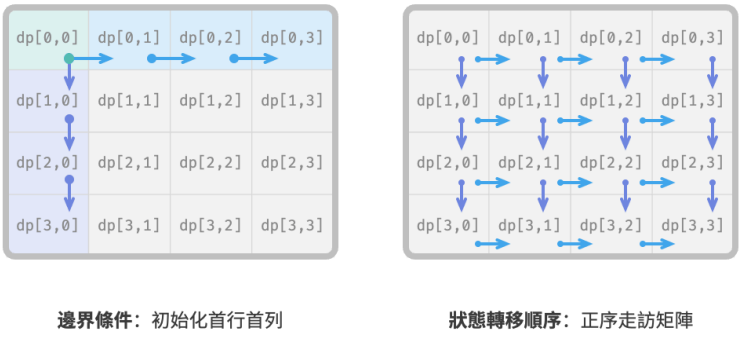


圖 14-13　邊界條件與狀態轉移順序

> **Note**
> 邊界條件在動態規劃中用於初始化 dp 表，在搜尋中用於剪枝。
> 狀態轉移順序的核心是要保證在計算當前問題的解時，所有它依賴的更小子問題的解都已經被正確地計算出來。

根據以上分析，我們已經可以直接寫出動態規劃程式碼。然而子問題分解是一種從頂至底的思想，因此按照“暴力搜尋 $\rightarrow$ 記憶化搜尋 $\rightarrow$ 動態規劃”的順序實現更加符合思維習慣。

1. 方法一：暴力搜尋

從狀態 $[i, j]$ 開始搜尋，不斷分解為更小的狀態 $[i-1, j]$ 和 $[i, j-1]$，遞迴函式包括以下要素。

- **遞迴參數**：狀態 $[i, j]$。
- **返回值**：從 $[0, 0]$ 到 $[i, j]$ 的最小路徑和 $dp[i, j]$。
- **終止條件**：當 $i = 0$ 且 $j = 0$ 時，返回代價 $grid[0, 0]$。
- **剪枝**：當 $i < 0$ 時或 $j < 0$ 時索引越界，此時返回代價 $+\infty$，代表不可行。

實現程式碼如下：

```rust
// === File: min_path_sum.rs ===

/* 最小路徑和：暴力搜尋 */
fn min_path_sum_dfs(grid: &Vec<Vec<i32>>, i: i32, j: i32) -> i32 {
    // 若為左上角單元格，則終止搜尋
    if i == 0 && j == 0 {
        return grid[0][0];
    }
    // 若行列索引越界，則返回 +∞ 代價
    if i < 0 || j < 0 {
        return i32::MAX;
    }
    // 計算從左上角到 (i-1, j) 和 (i, j-1) 的最小路徑代價
    let up = min_path_sum_dfs(grid, i - 1, j);
    let left = min_path_sum_dfs(grid, i, j - 1);
    // 返回從左上角到 (i, j) 的最小路徑代價
    std::cmp::min(left, up) + grid[i as usize][j as usize]
}
```

圖 14-14 給出了以 $dp[2, 1]$ 為根節點的遞迴樹，其中包含一些重疊子問題，其數量會隨著網格 `grid` 的尺寸變大而急劇增多。

從本質上看，造成重疊子問題的原因為：**存在多條路徑可以從左上角到達某一單元格**。

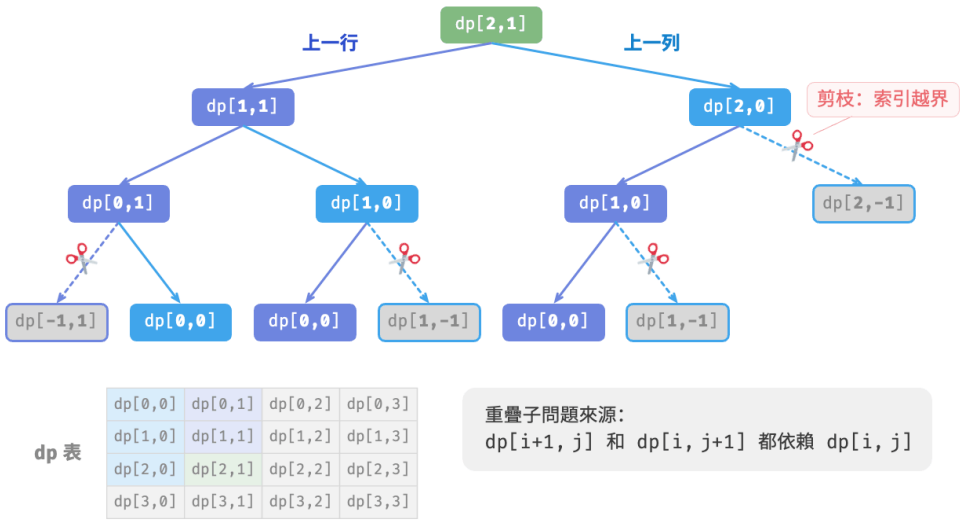


圖 14-14　暴力搜尋遞迴樹

每個狀態都有向下和向右兩種選擇，從左上角走到右下角總共需要 $m + n - 2$ 步，所以最差時間複雜度為 $O(2^{m+n})$。請注意，這種計算方式未考慮臨近網格邊界的情況，當到達網路邊界時只剩下一種選擇，因此實際的路徑數量會少一些。

2. 方法二：記憶化搜尋

我們引入一個和網格 `grid` 相同尺寸的記憶串列 `mem`，用於記錄各個子問題的解，並將重疊子問題進行剪枝：

```rust
// === File: min_path_sum.rs ===

/* 最小路徑和：記憶化搜尋 */
fn min_path_sum_dfs_mem(grid: &Vec<Vec<i32>>, mem: &mut Vec<Vec<i32>>, i: i32, j: i32) -> i32 {
    // 若為左上角單元格，則終止搜尋
    if i == 0 && j == 0 {
        return grid[0][0];
    }
    // 若行列索引越界，則返回 +∞ 代價
    if i < 0 || j < 0 {
        return i32::MAX;
    }
    // 若已有記錄，則直接返回
    if mem[i as usize][j as usize] != -1 {
        return mem[i as usize][j as usize];
    }
    // 左邊和上邊單元格的最小路徑代價
    let up = min_path_sum_dfs_mem(grid, mem, i - 1, j);
    let left = min_path_sum_dfs_mem(grid, mem, i, j - 1);
    // 記錄並返回左上角到 (i, j) 的最小路徑代價
    mem[i as usize][j as usize] = std::cmp::min(left, up) + grid[i as usize][j as usize];
    mem[i as usize][j as usize]
}
```

如圖 14-15 所示，在引入記憶化後，所有子問題的解只需計算一次，因此時間複雜度取決於狀態總數，即網格尺寸 $O(nm)$ 。

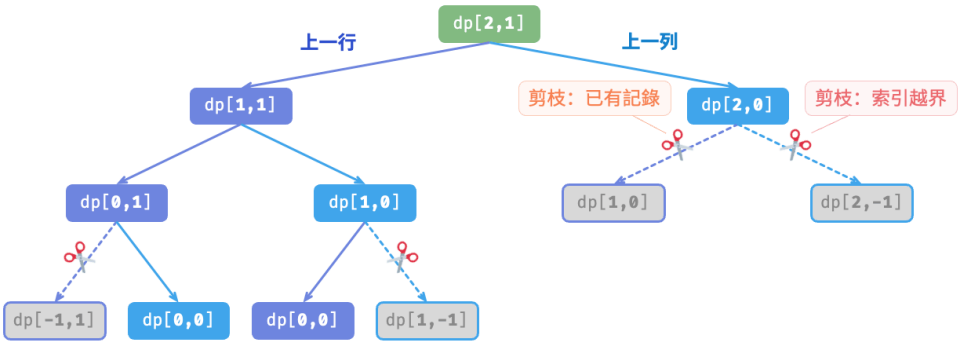


圖 14-15　記憶化搜尋遞迴樹

3. 方法三：動態規劃

基於迭代實現動態規劃解法，程式碼如下所示：

```rust
// === File: min_path_sum.rs ===

/* 最小路徑和：動態規劃 */
fn min_path_sum_dp(grid: &Vec<Vec<i32>>) -> i32 {
    let (n, m) = (grid.len(), grid[0].len());
    // 初始化 dp 表
    let mut dp = vec![vec![0; m]; n];
    dp[0][0] = grid[0][0];
    // 狀態轉移：首行
    for j in 1..m {
        dp[0][j] = dp[0][j - 1] + grid[0][j];
    }
    // 狀態轉移：首列
    for i in 1..n {
        dp[i][0] = dp[i - 1][0] + grid[i][0];
    }
    // 狀態轉移：其餘行和列
    for i in 1..n {
        for j in 1..m {
            dp[i][j] = std::cmp::min(dp[i][j - 1], dp[i - 1][j]) + grid[i][j];
        }
    }
    dp[n - 1][m - 1]
}
```

圖 14-16 展示了最小路徑和的狀態轉移過程，其走訪了整個網格，**因此時間複雜度為 $O(nm)$**。

陣列 `dp` 大小為 $n \times m$，**因此空間複雜度為 $O(nm)$**。

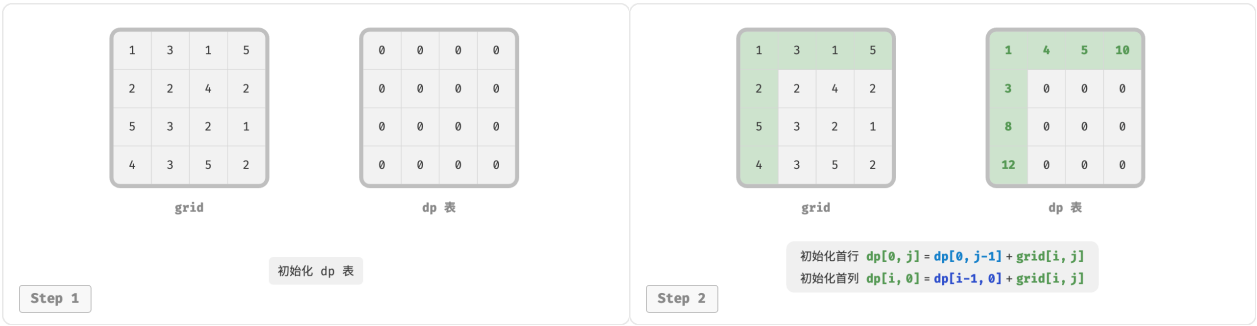


圖 14-16　最小路徑和的動態規劃過程

4. 空間最佳化

由於每個格子只與其左邊和上邊的格子有關，因此我們可以只用一個單行陣列來實現 dp 表。

請注意，因為陣列 `dp` 只能表示一行的狀態，所以我們無法提前初始化首列狀態，而是在走訪每行時更新它：

```rust
// === File: min_path_sum.rs ===

/* 最小路徑和：空間最佳化後的動態規劃 */
fn min_path_sum_dp_comp(grid: &Vec<Vec<i32>>) -> i32 {
    let (n, m) = (grid.len(), grid[0].len());
    // 初始化 dp 表
    let mut dp = vec![0; m];
    // 狀態轉移：首行
    dp[0] = grid[0][0];
    for j in 1..m {
        dp[j] = dp[j - 1] + grid[0][j];
    }
    // 狀態轉移：其餘行
    for i in 1..n {
        // 狀態轉移：首列
        dp[0] = dp[0] + grid[i][0];
        // 狀態轉移：其餘列
        for j in 1..m {
            dp[j] = std::cmp::min(dp[j - 1], dp[j]) + grid[i][j];
        }
    }
    dp[m - 1]
}
```

14.4　0-1 背包問題

背包問題是一個非常好的動態規劃入門題目，是動態規劃中最常見的問題形式。其具有很多變種，例如 0-1 背包問題、完全背包問題、多重背包問題等。

在本節中，我們先來求解最常見的 0-1 背包問題。

> **Question**
> 給定 n 個物品，第 i 個物品的重量為 $wgt[i-1]$、價值為 $val[i-1]$，和一個容量為 cap 的背包。每個物品只能選擇一次，問在限定背包容量下能放入物品的最大價值。

觀察圖 14-17，由於物品編號 i 從 1 開始計數，陣列索引從 0 開始計數，因此物品 i 對應重量 $wgt[i-1]$ 和價值 $val[i-1]$。

圖 14-17　0-1 背包的示例資料

我們可以將 0-1 背包問題看作一個由 n 輪決策組成的過程，對於每個物體都有不放入和放入兩種決策，因此該問題滿足決策樹模型。

該問題的目標是求解“在限定背包容量下能放入物品的最大價值”，因此較大機率是一個動態規劃問題。

第一步：思考每輪的決策，定義狀態，從而得到 dp 表

對於每個物品來說，不放入背包，背包容量不變；放入背包，背包容量減小。由此可得狀態定義：當前物品編號 i 和背包容量 c，記為 $[i, c]$。

狀態 $[i, c]$ 對應的子問題為：**前 i 個物品在容量為 c 的背包中的最大價值**，記為 $dp[i, c]$。

待求解的是 $dp[n, cap]$，因此需要一個尺寸為 $(n + 1) \times (cap + 1)$ 的二維 dp 表。

第二步：找出最優子結構，進而推導出狀態轉移方程

當我們做出物品 i 的決策後，剩餘的是前 $i - 1$ 個物品決策的子問題，可分為以下兩種情況。

- **不放入物品 i**：背包容量不變，狀態變化為 $[i - 1, c]$。
- **放入物品 i**：背包容量減少 $wgt[i - 1]$，價值增加 $val[i - 1]$，狀態變化為 $[i - 1, c - wgt[i - 1]]$。

上述分析向我們揭示了本題的最優子結構：**最大價值 $dp[i, c]$ 等於不放入物品 i 和放入物品 i 兩種方案中價值更大的那一個**。由此可推導出狀態轉移方程：

$$dp[i, c] = \max(dp[i - 1, c], dp[i - 1, c - wgt[i - 1]] + val[i - 1])$$

需要注意的是，若當前物品重量 $wgt[i - 1]$ 超出剩餘背包容量 c，則只能選擇不放入背包。

第三步：確定邊界條件和狀態轉移順序

當無物品或背包容量為 0 時最大價值為 0，即首列 $dp[i, 0]$ 和首行 $dp[0, c]$ 都等於 0。

當前狀態 $[i, c]$ 從上方的狀態 $[i - 1, c]$ 和左上方的狀態 $[i - 1, c - wgt[i - 1]]$ 轉移而來，因此透過兩層迴圈正序走訪整個 dp 表即可。

根據以上分析，我們接下來按順序實現暴力搜尋、記憶化搜尋、動態規劃解法。

1. 方法一：暴力搜尋

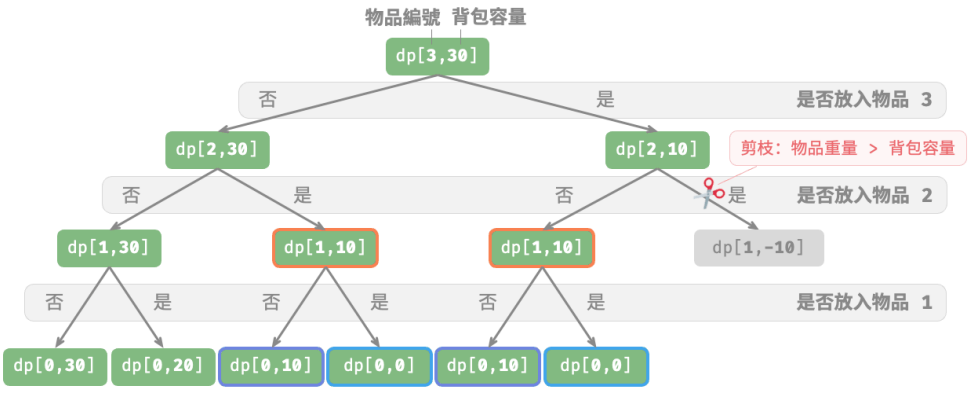
搜尋程式碼包含以下要素。

- **遞迴參數**：狀態 $[i, c]$ 。
- **返回值**：子問題的解 $dp[i, c]$ 。
- **終止條件**：當物品編號越界 $i = 0$ 或背包剩餘容量為 0 時，終止遞迴並返回價值 0 。
- **剪枝**：若當前物品重量超出背包剩餘容量，則只能選擇不放入背包。

```rust
// === File: knapsack.rs ===

/* 0-1 背包：暴力搜尋 */
fn knapsack_dfs(wgt: &[i32], val: &[i32], i: usize, c: usize) -> i32 {
    // 若已選完所有物品或背包無剩餘容量，則返回價值 0
    if i == 0 || c == 0 {
        return 0;
    }
    // 若超過背包容量，則只能選擇不放入背包
    if wgt[i - 1] > c as i32 {
        return knapsack_dfs(wgt, val, i - 1, c);
    }
    // 計算不放入和放入物品 i 的最大價值
    let no = knapsack_dfs(wgt, val, i - 1, c);
    let yes = knapsack_dfs(wgt, val, i - 1, c - wgt[i - 1] as usize) + val[i - 1];
    // 返回兩種方案中價值更大的那一個
    std::cmp::max(no, yes)
}
```

如圖 14-18 所示，由於每個物品都會產生不選和選兩條搜尋分支，因此時間複雜度為 $O(2^n)$ 。

觀察遞迴樹，容易發現其中存在重疊子問題，例如 $dp[1, 10]$ 等。而當物品較多、背包容量較大，尤其是相同重量的物品較多時，重疊子問題的數量將會大幅增多。



物品編號	i	1	2	3
物品重量	wgt[i]	10	20	20
物品價值	val[i]	60	110	120

背包容量　cap = 30

圖 14-18　0-1 背包問題的暴力搜尋遞迴樹

2. 方法二：記憶化搜尋

為了保證重疊子問題只被計算一次，我們藉助記憶串列 `mem` 來記錄子問題的解，其中 `mem[i][c]` 對應 $dp[i, c]$ 。

引入記憶化之後，**時間複雜度取決於子問題數量**，也就是 $O(n \times cap)$ 。實現程式碼如下：

```rust
// === File: knapsack.rs ===

/* 0-1 背包：記憶化搜尋 */
fn knapsack_dfs_mem(wgt: &[i32], val: &[i32], mem: &mut Vec<Vec<i32>>, i: usize, c: usize) -> i32 {
    // 若已選完所有物品或背包無剩餘容量，則返回價值 0
    if i == 0 || c == 0 {
        return 0;
    }
    // 若已有記錄，則直接返回
    if mem[i][c] != -1 {
        return mem[i][c];
    }
    // 若超過背包容量，則只能選擇不放入背包
    if wgt[i - 1] > c as i32 {
        return knapsack_dfs_mem(wgt, val, mem, i - 1, c);
    }
    // 計算不放入和放入物品 i 的最大價值
    let no = knapsack_dfs_mem(wgt, val, mem, i - 1, c);
    let yes = knapsack_dfs_mem(wgt, val, mem, i - 1, c - wgt[i - 1] as usize) + val[i - 1];
    // 記錄並返回兩種方案中價值更大的那一個
    mem[i][c] = std::cmp::max(no, yes);
    mem[i][c]
}
```

圖 14-19 展示了在記憶化搜尋中被剪掉的搜尋分支。

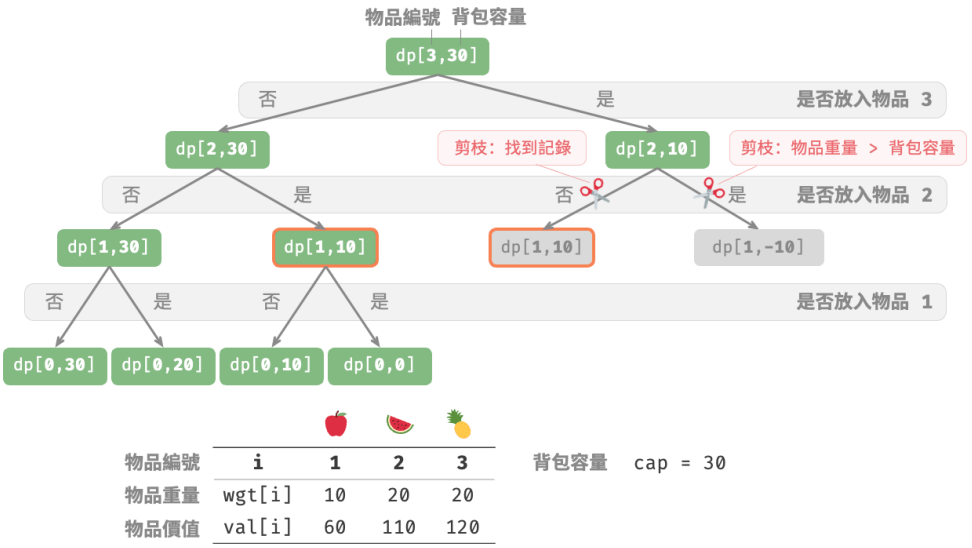


圖 14-19　0-1 背包問題的記憶化搜尋遞迴樹

3. 方法三：動態規劃

動態規劃實質上就是在狀態轉移中填充 dp 表的過程，程式碼如下所示：

```rust
// === File: knapsack.rs ===

/* 0-1 背包：動態規劃 */
fn knapsack_dp(wgt: &[i32], val: &[i32], cap: usize) -> i32 {
    let n = wgt.len();
    // 初始化 dp 表
    let mut dp = vec![vec![0; cap + 1]; n + 1];
    // 狀態轉移
    for i in 1..=n {
        for c in 1..=cap {
            if wgt[i - 1] > c as i32 {
                // 若超過背包容量，則不選物品 i
                dp[i][c] = dp[i - 1][c];
            } else {
                // 不選和選物品 i 這兩種方案的較大值
                dp[i][c] = std::cmp::max(
                    dp[i - 1][c],
                    dp[i - 1][c - wgt[i - 1] as usize] + val[i - 1],
                );
            }
        }
    }
    dp[n][cap]
}
```

如圖 14-20 所示，時間複雜度和空間複雜度都由陣列 `dp` 大小決定，即 $O(n \times cap)$ 。

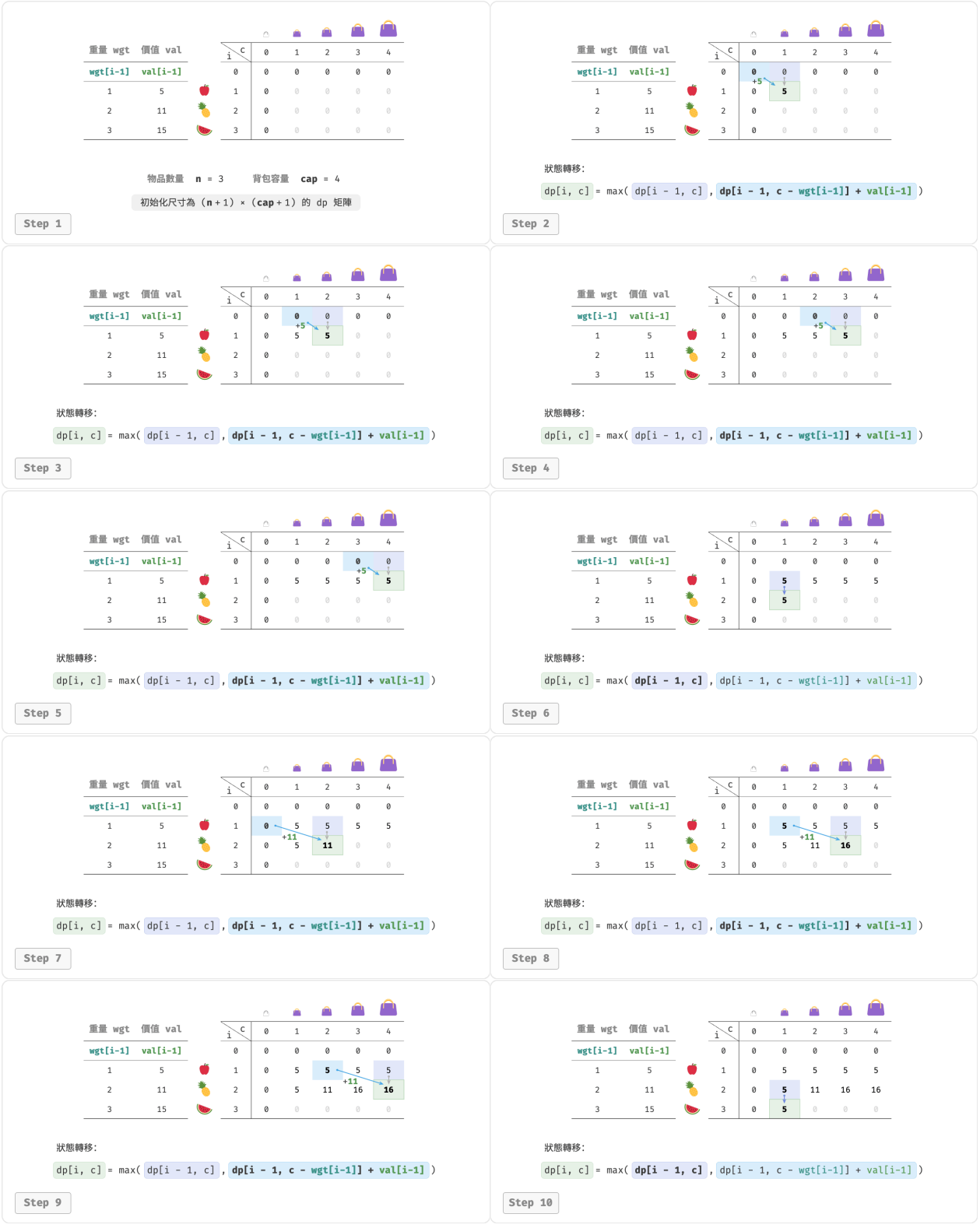


圖 14-20　0-1 背包問題的動態規劃過程

4. 空間最佳化

由於每個狀態都只與其上一行的狀態有關，因此我們可以使用兩個陣列滾動前進，將空間複雜度從 $O(n^2)$ 降至 $O(n)$ 。

進一步思考，我們能否僅用一個陣列實現空間最佳化呢？觀察可知，每個狀態都是由正上方或左上方的格子轉移過來的。假設只有一個陣列，當開始走訪第 i 行時，該陣列儲存的仍然是第 $i-1$ 行的狀態。

- 如果採取正序走訪，那麼走訪到 $dp[i,j]$ 時，左上方 $dp[i-1,1]$ ~ $dp[i-1,j-1]$ 值可能已經被覆蓋，此時就無法得到正確的狀態轉移結果。
- 如果採取倒序走訪，則不會發生覆蓋問題，狀態轉移可以正確進行。

圖 14-21 展示了在單個陣列下從第 $i=1$ 行轉換至第 $i=2$ 行的過程。請思考正序走訪和倒序走訪的區別。

圖 14-21　0-1 背包的空間最佳化後的動態規劃過程

在程式碼實現中，我們僅需將陣列 `dp` 的第一維 i 直接刪除，並且把內迴圈更改為倒序走訪即可：

```
// === File: knapsack.rs ===

/* 0-1 背包：空間最佳化後的動態規劃 */
fn knapsack_dp_comp(wgt: &[i32], val: &[i32], cap: usize) -> i32 {
    let n = wgt.len();
    // 初始化 dp 表
    let mut dp = vec![0; cap + 1];
    // 狀態轉移
    for i in 1..=n {
        // 倒序走訪
        for c in (1..=cap).rev() {
            if wgt[i - 1] <= c as i32 {
                // 不選和選物品 i 這兩種方案的較大值
                dp[c] = std::cmp::max(dp[c], dp[c - wgt[i - 1] as usize] + val[i - 1]);
            }
        }
    }
    dp[cap]
}
```

14.5　完全背包問題

在本節中，我們先求解另一個常見的背包問題：完全背包，再瞭解它的一種特例：零錢兌換。

14.5.1　完全背包問題

Question

給定 n 個物品，第 i 個物品的重量為 $wgt[i-1]$、價值為 $val[i-1]$，和一個容量為 cap 的背包。**每個物品可以重複選取**，問在限定背包容量下能放入物品的最大價值。示例如圖 14-22 所示。

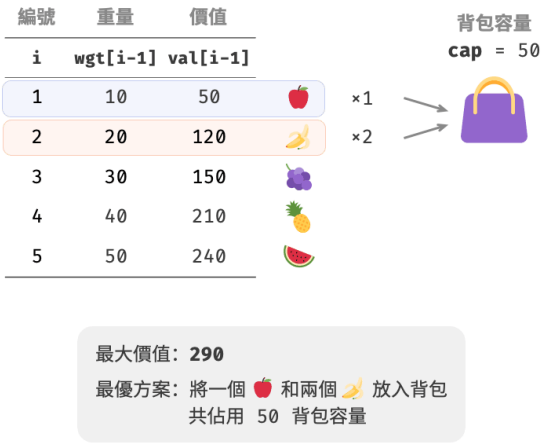


圖 14-22　完全背包問題的示例資料

1. 動態規劃思路

完全背包問題和 0-1 背包問題非常相似，**區別僅在於不限制物品的選擇次數**。

- 在 0-1 背包問題中，每種物品只有一個，因此將物品 i 放入背包後，只能從前 $i-1$ 個物品中選擇。
- 在完全背包問題中，每種物品的數量是無限的，因此將物品 i 放入背包後，**仍可以從前 i 個物品中選擇**。

在完全背包問題的規定下，狀態 $[i, c]$ 的變化分為兩種情況。

- **不放入物品 i**：與 0-1 背包問題相同，轉移至 $[i-1, c]$。
- **放入物品 i**：與 0-1 背包問題不同，轉移至 $[i, c-wgt[i-1]]$。

從而狀態轉移方程變為：

$$dp[i,c] = \max(dp[i-1,c], dp[i,c-wgt[i-1]] + val[i-1])$$

2. 程式碼實現

對比兩道題目的程式碼，狀態轉移中有一處從 $i-1$ 變為 i，其餘完全一致：

```rust
// === File: unbounded_knapsack.rs ===

/* 完全背包：動態規劃 */
fn unbounded_knapsack_dp(wgt: &[i32], val: &[i32], cap: usize) -> i32 {
    let n = wgt.len();
    // 初始化 dp 表
    let mut dp = vec![vec![0; cap + 1]; n + 1];
    // 狀態轉移
    for i in 1..=n {
        for c in 1..=cap {
            if wgt[i - 1] > c as i32 {
                // 若超過背包容量，則不選物品 i
                dp[i][c] = dp[i - 1][c];
            } else {
                // 不選和選物品 i 這兩種方案的較大值
                dp[i][c] = std::cmp::max(dp[i - 1][c], dp[i][c - wgt[i - 1] as usize] + val[i - 1]);
            }
        }
    }
    return dp[n][cap];
}
```

3. 空間最佳化

由於當前狀態是從左邊和上邊的狀態轉移而來的，**因此空間最佳化後應該對 dp 表中的每一行進行正序走訪**。

這個走訪順序與 0-1 背包正好相反。請藉助圖 14-23 來理解兩者的區別。

圖 14-23　完全背包問題在空間最佳化後的動態規劃過程

程式碼實現比較簡單，僅需將陣列 `dp` 的第一維刪除：

```rust
// === File: unbounded_knapsack.rs ===

/* 完全背包：空間最佳化後的動態規劃 */
fn unbounded_knapsack_dp_comp(wgt: &[i32], val: &[i32], cap: usize) -> i32 {
    let n = wgt.len();
    // 初始化 dp 表
    let mut dp = vec![0; cap + 1];
    // 狀態轉移
    for i in 1..=n {
        for c in 1..=cap {
            if wgt[i - 1] > c as i32 {
                // 若超過背包容量，則不選物品 i
                dp[c] = dp[c];
            } else {
                // 不選和選物品 i 這兩種方案的較大值
                dp[c] = std::cmp::max(dp[c], dp[c - wgt[i - 1] as usize] + val[i - 1]);
            }
        }
    }
    dp[cap]
}
```

14.5.2　零錢兌換問題

背包問題是一大類動態規劃問題的代表，其擁有很多變種，例如零錢兌換問題。

> **Question**
> 給定 n 種硬幣，第 i 種硬幣的面值為 $coins[i-1]$，目標金額為 amt，**每種硬幣可以重複選取**，問能夠湊出目標金額的最少硬幣數量。如果無法湊出目標金額，則返回 -1。示例如圖 14-24 所示。

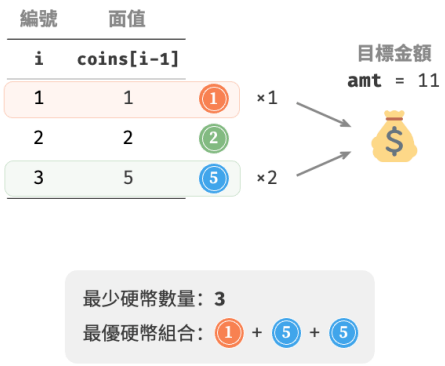


圖 14-24　零錢兌換問題的示例資料

1. 動態規劃思路

零錢兌換可以看作完全背包問題的一種特殊情況，兩者具有以下關聯與不同點。

- 兩道題可以相互轉換，“物品”對應“硬幣”、“物品重量”對應“硬幣面值”、“背包容量”對應“目標金額”。
- 最佳化目標相反，完全背包問題是要最大化物品價值，零錢兌換問題是要最小化硬幣數量。
- 完全背包問題是求“不超過”背包容量下的解，零錢兌換是求“恰好”湊到目標金額的解。

第一步：思考每輪的決策，定義狀態，從而得到 dp 表

狀態 $[i, a]$ 對應的子問題為：**前 i 種硬幣能夠湊出金額 a 的最少硬幣數量**，記為 $dp[i, a]$ 。

二維 dp 表的尺寸為 $(n + 1) \times (amt + 1)$ 。

第二步：找出最優子結構，進而推導出狀態轉移方程

本題與完全背包問題的狀態轉移方程存在以下兩點差異。

- 本題要求最小值，因此需將運算子 $\max()$ 更改為 $\min()$ 。
- 最佳化主體是硬幣數量而非商品價值，因此在選中硬幣時執行 $+1$ 即可。

$$dp[i, a] = \min(dp[i-1, a], dp[i, a - coins[i-1]] + 1)$$

第三步：確定邊界條件和狀態轉移順序

當目標金額為 0 時，湊出它的最少硬幣數量為 0 ，即首列所有 $dp[i, 0]$ 都等於 0 。

當無硬幣時，**無法湊出任意 > 0 的目標金額**，即是無效解。為使狀態轉移方程中的 $\min()$ 函式能夠識別並過濾無效解，我們考慮使用 $+\infty$ 來表示它們，即令首行所有 $dp[0, a]$ 都等於 $+\infty$ 。

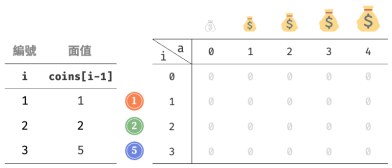
2. 程式碼實現

大多數程式語言並未提供 $+\infty$ 變數，只能使用整型 `int` 的最大值來代替。而這又會導致大數越界：狀態轉移方程中的 $+1$ 操作可能發生溢位。

為此，我們採用數字 $amt + 1$ 來表示無效解，因為湊出 amt 的硬幣數量最多為 amt 。最後返回前，判斷 $dp[n, amt]$ 是否等於 $amt + 1$ ，若是則返回 -1 ，代表無法湊出目標金額。程式碼如下所示：

```rust
// === File: coin_change.rs ===

/* 零錢兌換：動態規劃 */
fn coin_change_dp(coins: &[i32], amt: usize) -> i32 {
    let n = coins.len();
    let max = amt + 1;
    // 初始化 dp 表
    let mut dp = vec![vec![0; amt + 1]; n + 1];
    // 狀態轉移：首行首列
    for a in 1..=amt {
        dp[0][a] = max;
    }
    // 狀態轉移：其餘行和列
    for i in 1..=n {
        for a in 1..=amt {
            if coins[i - 1] > a as i32 {
                // 若超過目標金額，則不選硬幣 i
                dp[i][a] = dp[i - 1][a];
            } else {
                // 不選和選硬幣 i 這兩種方案的較小值
                dp[i][a] = std::cmp::min(dp[i - 1][a], dp[i][a - coins[i - 1] as usize] + 1);
            }
        }
    }
    if dp[n][amt] != max {
        return dp[n][amt] as i32;
    } else {
        -1
    }
}
```

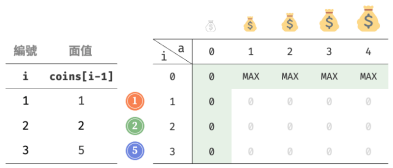
圖 14-25 展示了零錢兌換的動態規劃過程，和完全背包問題非常相似。



硬幣種類 **n** = 3　　目標金額 **amt** = 4

初始化尺寸為 (**n** + 1) × (**amt** + 1) 的 dp 矩陣

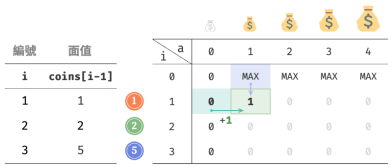
Step 1



硬幣種類 **n** = 3　　目標金額 **amt** = 4

初始化首行為 MAX = **amt** + 1 、首列為 0

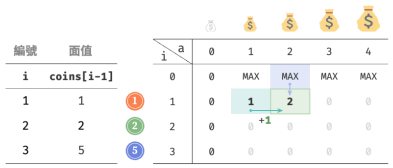
Step 2



狀態轉移：

dp[i, a] = min(dp[i - 1, a] , **dp[i, a - coins[i-1]] + 1**)

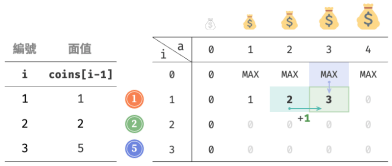
Step 3



狀態轉移：

dp[i, a] = min(dp[i - 1, a] , **dp[i, a - coins[i-1]] + 1**)

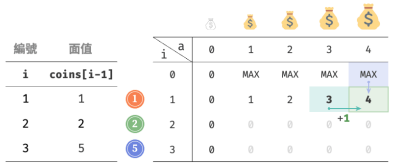
Step 4



狀態轉移：

dp[i, a] = min(dp[i - 1, a] , **dp[i, a - coins[i-1]] + 1**)

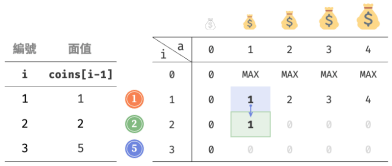
Step 5



狀態轉移：

dp[i, a] = min(dp[i - 1, a] , **dp[i, a - coins[i-1]] + 1**)

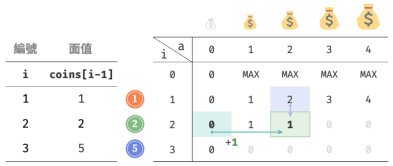
Step 6



狀態轉移：

dp[i, a] = min(**dp[i - 1, a]** , dp[i, a - coins[i-1]] + 1)

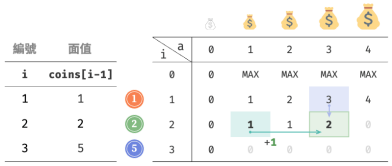
Step 7



狀態轉移：

dp[i, a] = min(dp[i - 1, a] , **dp[i, a - coins[i-1]] + 1**)

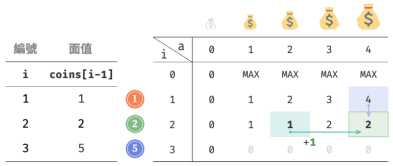
Step 8



狀態轉移：

dp[i, a] = min(dp[i - 1, a] , **dp[i, a - coins[i-1]] + 1**)

Step 9



狀態轉移：

dp[i, a] = min(dp[i - 1, a] , **dp[i, a - coins[i-1]] + 1**)

Step 10

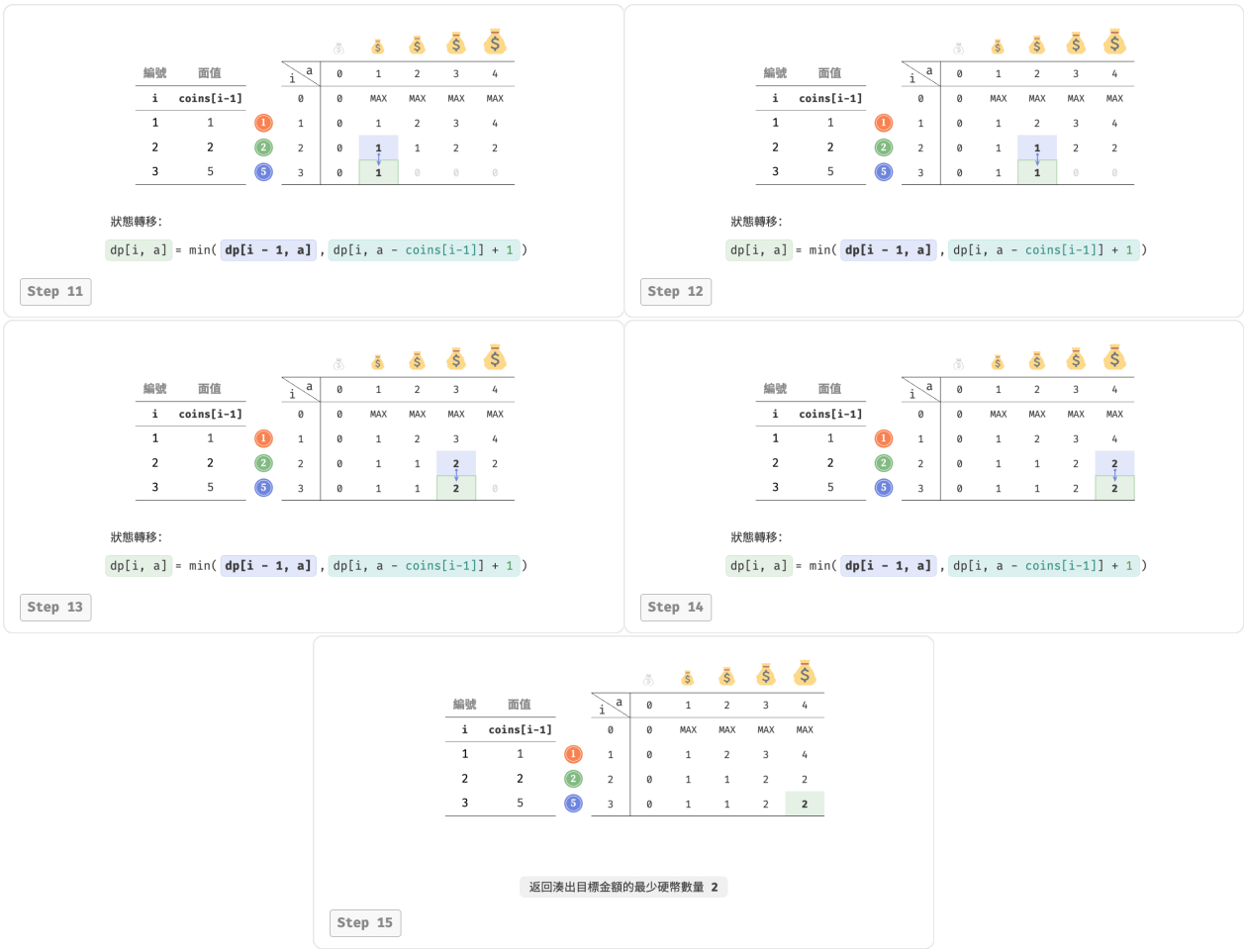


圖 14-25　零錢兌換問題的動態規劃過程

3. 空間最佳化

零錢兌換的空間最佳化的處理方式和完全背包問題一致：

```rust
// === File: coin_change.rs ===

/* 零錢兌換：空間最佳化後的動態規劃 */
fn coin_change_dp_comp(coins: &[i32], amt: usize) -> i32 {
    let n = coins.len();
    let max = amt + 1;
    // 初始化 dp 表
    let mut dp = vec![0; amt + 1];
    dp.fill(max);
    dp[0] = 0;
    // 狀態轉移
    for i in 1..=n {
        for a in 1..=amt {
```

```
            if coins[i - 1] > a as i32 {
                // 若超過目標金額，則不選硬幣 i
                dp[a] = dp[a];
            } else {
                // 不選和選硬幣 i 這兩種方案的較小值
                dp[a] = std::cmp::min(dp[a], dp[a - coins[i - 1] as usize] + 1);
            }
        }
    }
    if dp[amt] != max {
        return dp[amt] as i32;
    } else {
        -1
    }
}
```

14.5.3　零錢兌換問題 II

Question

給定 n 種硬幣，第 i 種硬幣的面值為 $coins[i-1]$，目標金額為 amt，每種硬幣可以重複選取，**問湊出目標金額的硬幣組合數量**。示例如圖 14-26 所示。

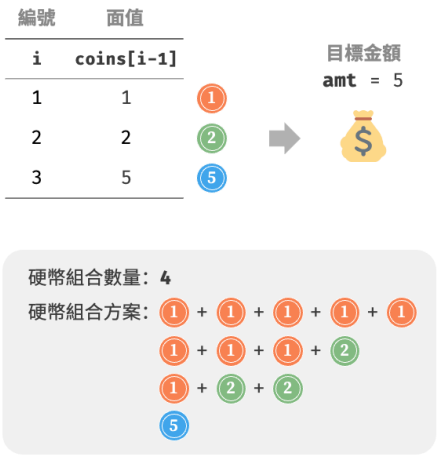


圖 14-26　零錢兌換問題 II 的示例資料

1. 動態規劃思路

相比於上一題，本題目標是求組合數量，因此子問題變為：**前 i 種硬幣能夠湊出金額 a 的組合數量**。而 dp 表仍然是尺寸為 $(n+1) \times (amt+1)$ 的二維矩陣。

當前狀態的組合數量等於不選當前硬幣與選當前硬幣這兩種決策的組合數量之和。狀態轉移方程為：

$$dp[i, a] = dp[i-1, a] + dp[i, a - coins[i-1]]$$

當目標金額為 0 時，無須選擇任何硬幣即可湊出目標金額，因此應將首列所有 $dp[i, 0]$ 都初始化為 1 。當無硬幣時，無法湊出任何 > 0 的目標金額，因此首行所有 $dp[0, a]$ 都等於 0 。

2. 程式碼實現

```rust
// === File: coin_change_ii.rs ===

/* 零錢兌換 II：動態規劃 */
fn coin_change_ii_dp(coins: &[i32], amt: usize) -> i32 {
    let n = coins.len();
    // 初始化 dp 表
    let mut dp = vec![vec![0; amt + 1]; n + 1];
    // 初始化首列
    for i in 0..=n {
        dp[i][0] = 1;
    }
    // 狀態轉移
    for i in 1..=n {
        for a in 1..=amt {
            if coins[i - 1] > a as i32 {
                // 若超過目標金額，則不選硬幣 i
                dp[i][a] = dp[i - 1][a];
            } else {
                // 不選和選硬幣 i 這兩種方案之和
                dp[i][a] = dp[i - 1][a] + dp[i][a - coins[i - 1] as usize];
            }
        }
    }
    dp[n][amt]
}
```

3. 空間最佳化

空間最佳化處理方式相同，刪除硬幣維度即可：

```rust
// === File: coin_change_ii.rs ===

/* 零錢兌換 II：空間最佳化後的動態規劃 */
fn coin_change_ii_dp_comp(coins: &[i32], amt: usize) -> i32 {
    let n = coins.len();
```

```rust
    // 初始化 dp 表
    let mut dp = vec![0; amt + 1];
    dp[0] = 1;
    // 狀態轉移
    for i in 1..=n {
        for a in 1..=amt {
            if coins[i - 1] > a as i32 {
                // 若超過目標金額，則不選硬幣 i
                dp[a] = dp[a];
            } else {
                // 不選和選硬幣 i 這兩種方案之和
                dp[a] = dp[a] + dp[a - coins[i - 1] as usize];
            }
        }
    }
    dp[amt]
}
```

14.6　編輯距離問題

編輯距離，也稱 Levenshtein 距離，指兩個字串之間互相轉換的最少修改次數，通常用於在資訊檢索和自然語言處理中度量兩個序列的相似度。

> **Question**
> 輸入兩個字串 s 和 t，返回將 s 轉換為 t 所需的最少編輯步數。
> 你可以在一個字串中進行三種編輯操作：插入一個字元、刪除一個字元、將字元替換為任意一個字元。

如圖 14-27 所示，將 `kitten` 轉換為 `sitting` 需要編輯 3 步，包括 2 次替換操作與 1 次新增操作；將 `hello` 轉換為 `algo` 需要 3 步，包括 2 次替換操作和 1 次刪除操作。

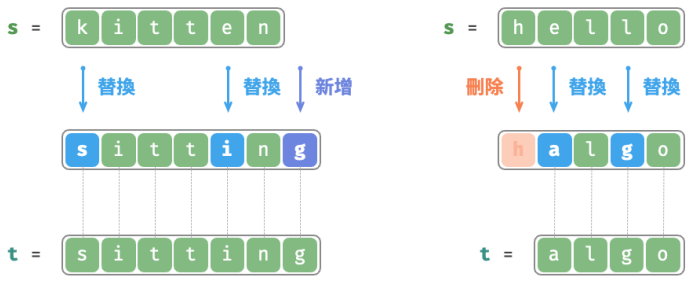


圖 14-27　編輯距離的示例資料

編輯距離問題可以很自然地用決策樹模型來解釋。字串對應樹節點，一輪決策（一次編輯操作）對應樹的一條邊。

如圖 14-28 所示，在不限制操作的情況下，每個節點都可以派生出許多條邊，每條邊對應一種操作，這意味著從 `hello` 轉換到 `algo` 有許多種可能的路徑。

從決策樹的角度看，本題的目標是求解節點 `hello` 和節點 `algo` 之間的最短路徑。

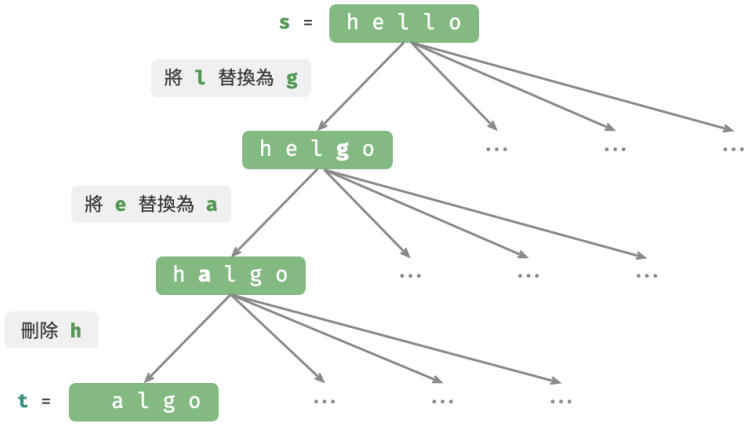


圖 14-28　基於決策樹模型表示編輯距離問題

1. 動態規劃思路

第一步：思考每輪的決策，定義狀態，從而得到 dp 表

每一輪的決策是對字串 s 進行一次編輯操作。

我們希望在編輯操作的過程中，問題的規模逐漸縮小，這樣才能構建子問題。設字串 s 和 t 的長度分別為 n 和 m，我們先考慮兩字串尾部的字元 $s[n-1]$ 和 $t[m-1]$。

- 若 $s[n-1]$ 和 $t[m-1]$ 相同，我們可以跳過它們，直接考慮 $s[n-2]$ 和 $t[m-2]$。
- 若 $s[n-1]$ 和 $t[m-1]$ 不同，我們需要對 s 進行一次編輯（插入、刪除、替換），使得兩字串尾部的字元相同，從而可以跳過它們，考慮規模更小的問題。

也就是說，我們在字串 s 中進行的每一輪決策（編輯操作），都會使得 s 和 t 中剩餘的待匹配字元發生變化。因此，狀態為當前在 s 和 t 中考慮的第 i 和第 j 個字元，記為 $[i, j]$。

狀態 $[i, j]$ 對應的子問題：**將 s 的前 i 個字元更改為 t 的前 j 個字元所需的最少編輯步數**。

至此，得到一個尺寸為 $(i+1) \times (j+1)$ 的二維 dp 表。

第二步：找出最優子結構，進而推導出狀態轉移方程

考慮子問題 $dp[i, j]$，其對應的兩個字串的尾部字元為 $s[i-1]$ 和 $t[j-1]$，可根據不同編輯操作分為圖 14-29 所示的三種情況。

1. 在 $s[i-1]$ 之後新增 $t[j-1]$，則剩餘子問題 $dp[i, j-1]$。
2. 刪除 $s[i-1]$，則剩餘子問題 $dp[i-1, j]$。
3. 將 $s[i-1]$ 替換為 $t[j-1]$，則剩餘子問題 $dp[i-1, j-1]$。

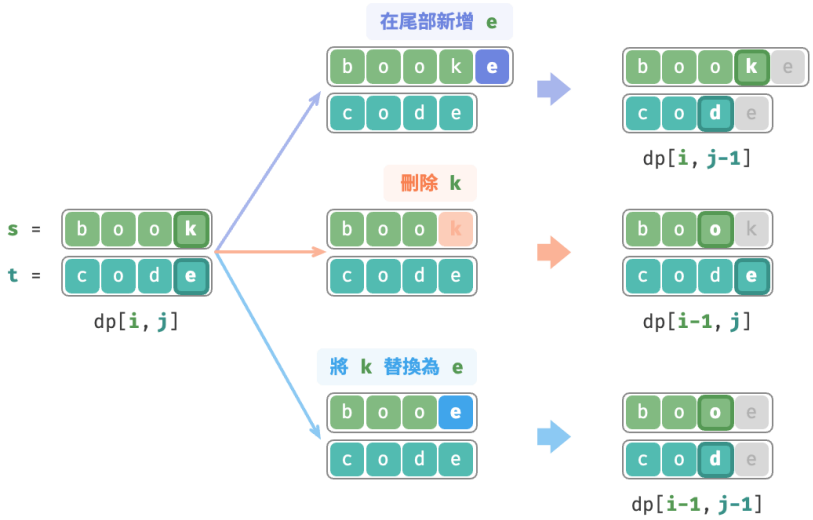


圖 14-29　編輯距離的狀態轉移

根據以上分析，可得最優子結構：$dp[i, j]$ 的最少編輯步數等於 $dp[i, j-1]$、$dp[i-1, j]$、$dp[i-1, j-1]$ 三者中的最少編輯步數，再加上本次的編輯步數 1。對應的狀態轉移方程為：

$$dp[i, j] = \min(dp[i, j-1], dp[i-1, j], dp[i-1, j-1]) + 1$$

請注意，**當 $s[i-1]$ 和 $t[j-1]$ 相同時，無須編輯當前字元**，這種情況下的狀態轉移方程為：

$$dp[i, j] = dp[i-1, j-1]$$

第三步：確定邊界條件和狀態轉移順序

當兩字串都為空時，編輯步數為 0，即 $dp[0, 0] = 0$。當 s 為空但 t 不為空時，最少編輯步數等於 t 的長度，即首行 $dp[0, j] = j$。當 s 不為空但 t 為空時，最少編輯步數等於 s 的長度，即首列 $dp[i, 0] = i$。

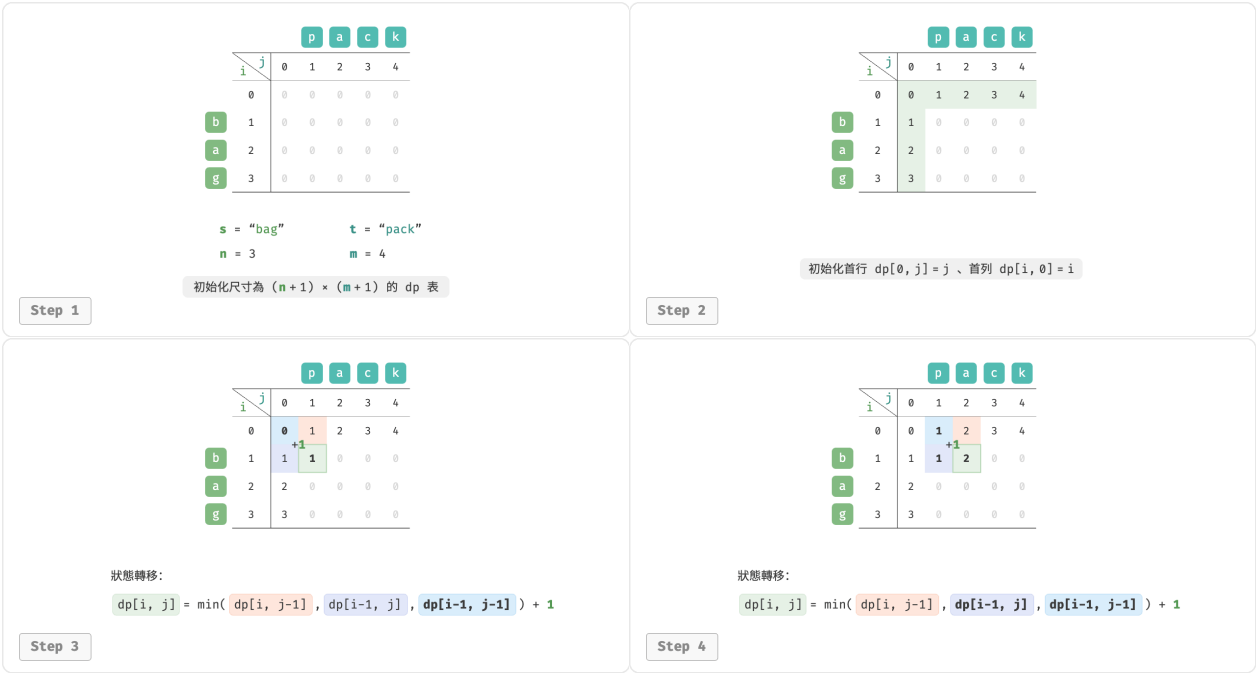
觀察狀態轉移方程，解 $dp[i, j]$ 依賴左方、上方、左上方的解，因此透過兩層迴圈正序走訪整個 dp 表即可。

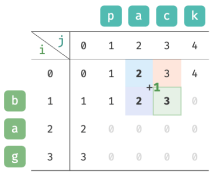
2. 程式碼實現

```rust
// === File: edit_distance.rs ===

/* 編輯距離：動態規劃 */
fn edit_distance_dp(s: &str, t: &str) -> i32 {
    let (n, m) = (s.len(), t.len());
    let mut dp = vec![vec![0; m + 1]; n + 1];
    // 狀態轉移：首行首列
    for i in 1..=n {
```

```
        dp[i][0] = i as i32;
    }
    for j in 1..m {
        dp[0][j] = j as i32;
    }
    // 狀態轉移：其餘行和列
    for i in 1..=n {
        for j in 1..=m {
            if s.chars().nth(i - 1) == t.chars().nth(j - 1) {
                // 若兩字元相等，則直接跳過此兩字元
                dp[i][j] = dp[i - 1][j - 1];
            } else {
                // 最少編輯步數 = 插入、刪除、替換這三種操作的最少編輯步數 + 1
                dp[i][j] =
                    std::cmp::min(std::cmp::min(dp[i][j - 1], dp[i - 1][j]), dp[i - 1][j - 1]) + 1;
            }
        }
    }
    dp[n][m]
}
```

如圖 14-30 所示，編輯距離問題的狀態轉移過程與背包問題非常類似，都可以看作填寫一個二維網格的過程。

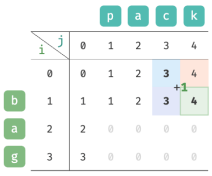




狀態轉移：

$dp[i, j] = \min(dp[i, j-1], \mathbf{dp[i-1, j]}, \mathbf{dp[i-1, j-1]}) + 1$

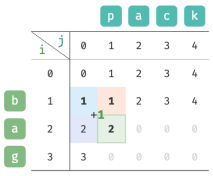
Step 5



狀態轉移：

$dp[i, j] = \min(dp[i, j-1], \mathbf{dp[i-1, j]}, \mathbf{dp[i-1, j-1]}) + 1$

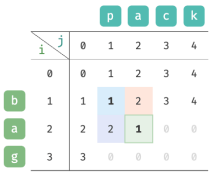
Step 6



狀態轉移：

$dp[i, j] = \min(\mathbf{dp[i, j-1]}, dp[i-1, j], \mathbf{dp[i-1, j-1]}) + 1$

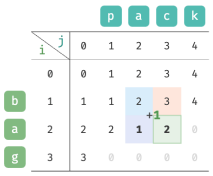
Step 7



狀態轉移：

$\because s[i-1] = t[i-1] = \mathbf{'a'} \quad \therefore dp[i, j] = \mathbf{dp[i-1, j-1]}$

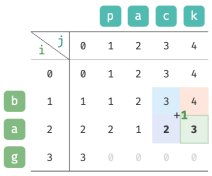
Step 8



狀態轉移：

$dp[i, j] = \min(dp[i, j-1], \mathbf{dp[i-1, j]}, dp[i-1, j-1]) + 1$

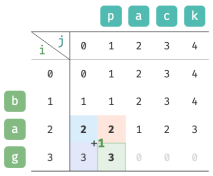
Step 9



狀態轉移：

$dp[i, j] = \min(dp[i, j-1], \mathbf{dp[i-1, j]}, dp[i-1, j-1]) + 1$

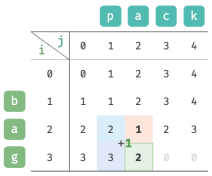
Step 10



狀態轉移：

$dp[i, j] = \min(\mathbf{dp[i, j-1]}, dp[i-1, j], dp[i-1, j-1]) + 1$

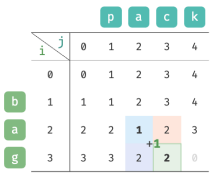
Step 11



狀態轉移：

$dp[i, j] = \min(\mathbf{dp[i, j-1]}, dp[i-1, j], dp[i-1, j-1]) + 1$

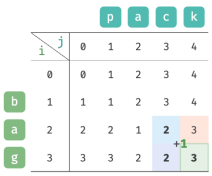
Step 12



狀態轉移：

$dp[i, j] = \min(dp[i, j-1], dp[i-1, j], \mathbf{dp[i-1, j-1]}) + 1$

Step 13



狀態轉移：

$dp[i, j] = \min(dp[i, j-1], dp[i-1, j], \mathbf{dp[i-1, j-1]}) + 1$

Step 14

圖 14-30　編輯距離的動態規劃過程

3. 空間最佳化

由於 $dp[i, j]$ 是由上方 $dp[i-1, j]$、左方 $dp[i, j-1]$、左上方 $dp[i-1, j-1]$ 轉移而來的，而正序走訪會丟失左上方 $dp[i-1, j-1]$，倒序走訪無法提前構建 $dp[i, j-1]$，因此兩種走訪順序都不可取。

為此，我們可以使用一個變數 `leftup` 來暫存左上方的解 $dp[i-1, j-1]$，從而只需考慮左方和上方的解。此時的情況與完全背包問題相同，可使用正序走訪。程式碼如下所示：

```
// === File: edit_distance.rs ===

/* 編輯距離：空間最佳化後的動態規劃 */
fn edit_distance_dp_comp(s: &str, t: &str) -> i32 {
    let (n, m) = (s.len(), t.len());
    let mut dp = vec![0; m + 1];
    // 狀態轉移：首行
    for j in 1..m {
        dp[j] = j as i32;
    }
    // 狀態轉移：其餘行
    for i in 1..=n {
        // 狀態轉移：首列
        let mut leftup = dp[0]; // 暫存 dp[i-1, j-1]
        dp[0] = i as i32;
        // 狀態轉移：其餘列
        for j in 1..=m {
            let temp = dp[j];
            if s.chars().nth(i - 1) == t.chars().nth(j - 1) {
                // 若兩字元相等，則直接跳過此兩字元
                dp[j] = leftup;
            } else {
                // 最少編輯步數 = 插入、刪除、替換這三種操作的最少編輯步數 + 1
                dp[j] = std::cmp::min(std::cmp::min(dp[j - 1], dp[j]), leftup) + 1;
            }
            leftup = temp; // 更新為下一輪的 dp[i-1, j-1]
```

```
        }
    }
    dp[m]
}
```

14.7　小結

- 動態規劃對問題進行分解，並透過儲存子問題的解來規避重複計算，提高計算效率。
- 不考慮時間的前提下，所有動態規劃問題都可以用回溯（暴力搜尋）進行求解，但遞迴樹中存在大量的重疊子問題，效率極低。透過引入記憶化串列，可以儲存所有計算過的子問題的解，從而保證重疊子問題只被計算一次。
- 記憶化搜尋是一種從頂至底的遞迴式解法，而與之對應的動態規劃是一種從底至頂的遞推式解法，其如同“填寫表格”一樣。由於當前狀態僅依賴某些區域性狀態，因此我們可以消除 dp 表的一個維度，從而降低空間複雜度。
- 子問題分解是一種通用的演算法思路，在分治、動態規劃、回溯中具有不同的性質。
- 動態規劃問題有三大特性：重疊子問題、最優子結構、無後效性。
- 如果原問題的最優解可以從子問題的最優解構建得來，則它就具有最優子結構。
- 無後效性指對於一個狀態，其未來發展只與該狀態有關，而與過去經歷的所有狀態無關。許多組合最佳化問題不具有無後效性，無法使用動態規劃快速求解。

背包問題

- 背包問題是最典型的動態規劃問題之一，具有 0-1 背包、完全背包、多重背包等變種。
- 0-1 背包的狀態定義為前 i 個物品在容量為 c 的背包中的最大價值。根據不放入背包和放入背包兩種決策，可得到最優子結構，並構建出狀態轉移方程。在空間最佳化中，由於每個狀態依賴正上方和左上方的狀態，因此需要倒序走訪串列，避免左上方狀態被覆蓋。
- 完全背包問題的每種物品的選取數量無限制，因此選擇放入物品的狀態轉移與 0-1 背包問題不同。由於狀態依賴正上方和正左方的狀態，因此在空間最佳化中應當正序走訪。
- 零錢兌換問題是完全背包問題的一個變種。它從求“最大”價值變為求“最小”硬幣數量，因此狀態轉移方程中的 $\max()$ 應改為 $\min()$ 。從追求“不超過”背包容量到追求“恰好”湊出目標金額，因此使用 $amt + 1$ 來表示“無法湊出目標金額”的無效解。
- 零錢兌換問題 II 從求“最少硬幣數量”改為求“硬幣組合數量”，狀態轉移方程相應地從 $\min()$ 改為求和運算子。

編輯距離問題

- 編輯距離（Levenshtein 距離）用於衡量兩個字串之間的相似度，其定義為從一個字串到另一個字串的最少編輯步數，編輯操作包括新增、刪除、替換。
- 編輯距離問題的狀態定義為將 s 的前 i 個字元更改為 t 的前 j 個字元所需的最少編輯步數。當 $s[i] \neq t[j]$ 時，具有三種決策：新增、刪除、替換，它們都有相應的剩餘子問題。據此便可以找出最優子結構與構建狀態轉移方程。而當 $s[i] = t[j]$ 時，無須編輯當前字元。
- 在編輯距離中，狀態依賴其正上方、正左方、左上方的狀態，因此空間最佳化後正序或倒序走訪都無法正確地進行狀態轉移。為此，我們利用一個變數暫存左上方狀態，從而轉化到與完全背包問題等價的情況，可以在空間最佳化後進行正序走訪。

第 15 章　貪婪

Abstract

向日葵朝著太陽轉動，時刻追求自身成長的最大可能。

貪婪策略在一輪輪的簡單選擇中，逐步導向最佳答案。

15.1　貪婪演算法

貪婪演算法（greedy algorithm）是一種常見的解決最佳化問題的演算法，其基本思想是在問題的每個決策階段，都選擇當前看起來最優的選擇，即貪婪地做出區域性最優的決策，以期獲得全域性最優解。貪婪演算法簡潔且高效，在許多實際問題中有著廣泛的應用。

貪婪演算法和動態規劃都常用於解決最佳化問題。它們之間存在一些相似之處，比如都依賴最優子結構性質，但工作原理不同。

- 動態規劃會根據之前階段的所有決策來考慮當前決策，並使用過去子問題的解來構建當前子問題的解。
- 貪婪演算法不會考慮過去的決策，而是一路向前地進行貪婪選擇，不斷縮小問題範圍，直至問題被解決。

我們先透過例題“零錢兌換”瞭解貪婪演算法的工作原理。這道題已經在“完全背包問題”章節中介紹過，相信你對它並不陌生。

Question

給定 n 種硬幣，第 i 種硬幣的面值為 $coins[i-1]$，目標金額為 amt，每種硬幣可以重複選取，問能夠湊出目標金額的最少硬幣數量。如果無法湊出目標金額，則返回 -1。

本題採取的貪婪策略如圖 15-1 所示。給定目標金額，**我們貪婪地選擇不大於且最接近它的硬幣**，不斷迴圈該步驟，直至湊出目標金額為止。

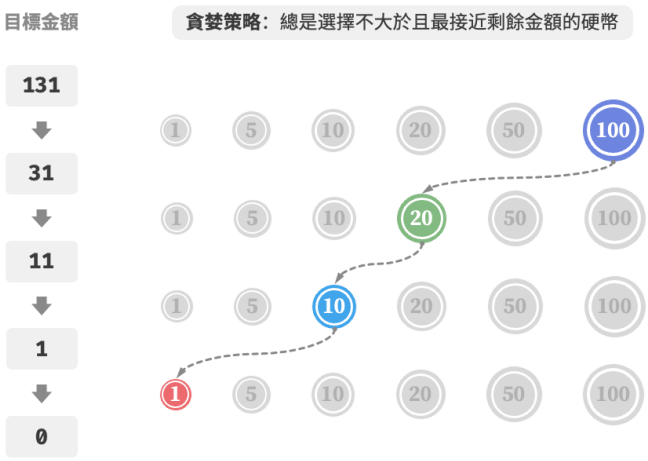


圖 15-1　零錢兌換的貪婪策略

實現程式碼如下所示：

```
// === File: coin_change_greedy.rs ===

/* 零錢兌換：貪婪 */
fn coin_change_greedy(coins: &[i32], mut amt: i32) -> i32 {
    // 假設 coins 串列有序
```

```
    let mut i = coins.len() - 1;
    let mut count = 0;
    // 迴圈進行貪婪選擇，直到無剩餘金額
    while amt > 0 {
        // 找到小於且最接近剩餘金額的硬幣
        while i > 0 && coins[i] > amt {
            i -= 1;
        }
        // 選擇 coins[i]
        amt -= coins[i];
        count += 1;
    }
    // 若未找到可行方案，則返回 -1
    if amt == 0 {
        count
    } else {
        -1
    }
}
```

你可能會不由地發出感嘆：So clean！貪婪演算法僅用約十行程式碼就解決了零錢兌換問題。

15.1.1　貪婪演算法的優點與侷限性

貪婪演算法不僅操作直接、實現簡單，而且通常效率也很高。在以上程式碼中，記硬幣最小面值為 $\min(coins)$ ，則貪婪選擇最多迴圈 $amt / \min(coins)$ 次，時間複雜度為 $O(amt / \min(coins))$ 。這比動態規劃解法的時間複雜度 $O(n \times amt)$ 小了一個數量級。

然而，**對於某些硬幣面值組合，貪婪演算法並不能找到最優解**。圖 15-2 給出了兩個示例。

- **正例** $coins = [1, 5, 10, 20, 50, 100]$：在該硬幣組合下，給定任意 amt ，貪婪演算法都可以找到最優解。
- **反例** $coins = [1, 20, 50]$：假設 $amt = 60$ ，貪婪演算法只能找到 $50 + 1 \times 10$ 的兌換組合，共計 11 枚硬幣，但動態規劃可以找到最優解 $20 + 20 + 20$ ，僅需 3 枚硬幣。
- **反例** $coins = [1, 49, 50]$：假設 $amt = 98$ ，貪婪演算法只能找到 $50 + 1 \times 48$ 的兌換組合，共計 49 枚硬幣，但動態規劃可以找到最優解 $49 + 49$ ，僅需 2 枚硬幣。

圖 15-2　貪婪演算法無法找出最優解的示例

也就是說，對於零錢兌換問題，貪婪演算法無法保證找到全域性最優解，並且有可能找到非常差的解。它更適合用動態規劃解決。

一般情況下，貪婪演算法的適用情況分以下兩種。

1. **可以保證找到最優解**：貪婪演算法在這種情況下往往是最優選擇，因為它往往比回溯、動態規劃更高效。
2. **可以找到近似最優解**：貪婪演算法在這種情況下也是可用的。對於很多複雜問題來說，尋找全域性最優解非常困難，能以較高效率找到次優解也是非常不錯的。

15.1.2　貪婪演算法特性

那麼問題來了，什麼樣的問題適合用貪婪演算法求解呢？或者說，貪婪演算法在什麼情況下可以保證找到最優解？

相較於動態規劃，貪婪演算法的使用條件更加苛刻，其主要關注問題的兩個性質。

- **貪婪選擇性質**：只有當局部最優選擇始終可以導致全域性最優解時，貪婪演算法才能保證得到最優解。
- **最優子結構**：原問題的最優解包含子問題的最優解。

最優子結構已經在“動態規劃”章節中介紹過，這裡不再贅述。值得注意的是，一些問題的最優子結構並不明顯，但仍然可使用貪婪演算法解決。

我們主要探究貪婪選擇性質的判斷方法。雖然它的描述看上去比較簡單，**但實際上對於許多問題，證明貪婪選擇性質並非易事**。

例如零錢兌換問題，我們雖然能夠容易地舉出反例，對貪婪選擇性質進行證偽，但證實的難度較大。如果問：**滿足什麼條件的硬幣組合可以使用貪婪演算法求解**？我們往往只能憑藉直覺或舉例子來給出一個模稜兩可的答案，而難以給出嚴謹的數學證明。

> **Quote**
> 有一篇論文給出了一個 $O(n^3)$ 時間複雜度的演算法，用於判斷一個硬幣組合能否使用貪婪演算法找出任意金額的最優解。
> Pearson, D. A polynomial-time algorithm for the change-making problem[J]. Operations Research Letters, 2005, 33(3): 231-234.

15.1.3　貪婪演算法解題步驟

貪婪問題的解決流程大體可分為以下三步。

1. **問題分析**：梳理與理解問題特性，包括狀態定義、最佳化目標和約束條件等。這一步在回溯和動態規劃中都有涉及。
2. **確定貪婪策略**：確定如何在每一步中做出貪婪選擇。這個策略能夠在每一步減小問題的規模，並最終解決整個問題。
3. **正確性證明**：通常需要證明問題具有貪婪選擇性質和最優子結構。這個步驟可能需要用到數學證明，例如歸納法或反證法等。

確定貪婪策略是求解問題的核心步驟，但實施起來可能並不容易，主要有以下原因。

- **不同問題的貪婪策略的差異較大**。對於許多問題來說，貪婪策略比較淺顯，我們透過一些大概的思考與嘗試就能得出。而對於一些複雜問題，貪婪策略可能非常隱蔽，這種情況就非常考驗個人的解題經驗與演算法能力了。
- **某些貪婪策略具有較強的迷惑性**。當我們滿懷信心設計好貪婪策略，寫出解題程式碼並提交執行，很可能發現部分測試樣例無法透過。這是因為設計的貪婪策略只是“部分正確”的，上文介紹的零錢兌換就是一個典型案例。

為了保證正確性，我們應該對貪婪策略進行嚴謹的數學證明，**通常需要用到反證法或數學歸納法**。

然而，正確性證明也很可能不是一件易事。如若沒有頭緒，我們通常會選擇面向測試用例進行程式碼除錯，一步步修改與驗證貪婪策略。

15.1.4　貪婪演算法典型例題

貪婪演算法常常應用在滿足貪婪選擇性質和最優子結構的最佳化問題中，以下列舉了一些典型的貪婪演算法問題。

- **硬幣找零問題**：在某些硬幣組合下，貪婪演算法總是可以得到最優解。
- **區間排程問題**：假設你有一些任務，每個任務在一段時間內進行，你的目標是完成儘可能多的任務。如果每次都選擇結束時間最早的任務，那麼貪婪演算法就可以得到最優解。
- **分數背包問題**：給定一組物品和一個載重量，你的目標是選擇一組物品，使得總重量不超過載重量，且總價值最大。如果每次都選擇價效比最高（價值 / 重量）的物品，那麼貪婪演算法在一些情況下可以得到最優解。
- **股票買賣問題**：給定一組股票的歷史價格，你可以進行多次買賣，但如果你已經持有股票，那麼在賣出之前不能再買，目標是獲取最大利潤。
- **霍夫曼編碼**：霍夫曼編碼是一種用於無損資料壓縮的貪婪演算法。透過構建霍夫曼樹，每次選擇出現頻率最低的兩個節點合併，最後得到的霍夫曼樹的帶權路徑長度（編碼長度）最小。
- **Dijkstra 演算法**：它是一種解決給定源頂點到其餘各頂點的最短路徑問題的貪婪演算法。

15.2　分數背包問題

Question

給定 n 個物品，第 i 個物品的重量為 $wgt[i-1]$、價值為 $val[i-1]$ ，和一個容量為 cap 的背包。每個物品只能選擇一次，**但可以選擇物品的一部分，價值根據選擇的重量比例計算**，問在限定背包容量下背包中物品的最大價值。示例如圖 15-3 所示。

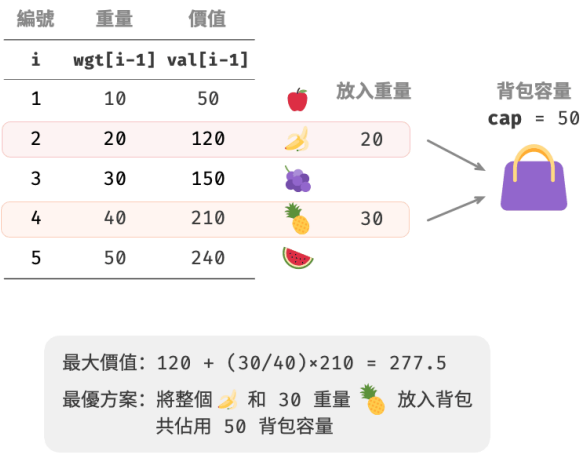


圖 15-3　分數背包問題的示例資料

分數背包問題和 0-1 背包問題整體上非常相似，狀態包含當前物品 i 和容量 c，目標是求限定背包容量下的最大價值。

不同點在於，本題允許只選擇物品的一部分。如圖 15-4 所示，**我們可以對物品任意地進行切分，並按照重量比例來計算相應價值**。

1. 對於物品 i，它在單位重量下的價值為 $val[i-1]/wgt[i-1]$，簡稱單位價值。
2. 假設放入一部分物品 i，重量為 w，則背包增加的價值為 $w \times val[i-1]/wgt[i-1]$。

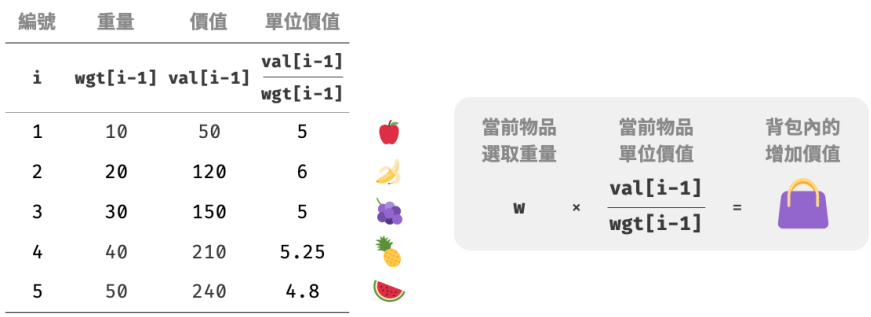


圖 15-4　物品在單位重量下的價值

1. 貪婪策略確定

最大化背包內物品總價值，**本質上是最大化單位重量下的物品價值**。由此便可推理出圖 15-5 所示的貪婪策略。

1. 將物品按照單位價值從高到低進行排序。
2. 走訪所有物品，**每輪貪婪地選擇單位價值最高的物品**。

3. 若剩餘背包容量不足，則使用當前物品的一部分填滿背包。

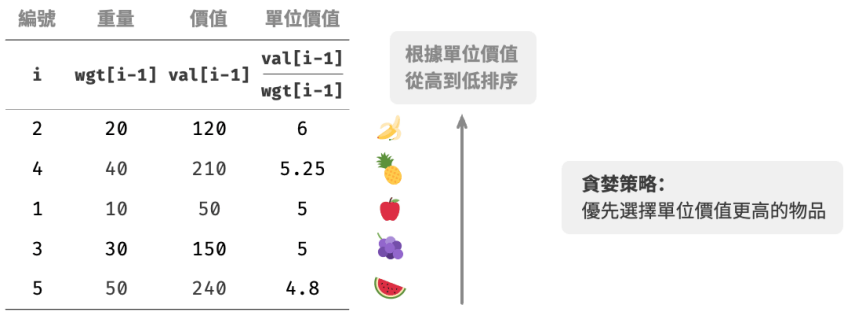


圖 15-5　分數背包問題的貪婪策略

2. 程式碼實現

我們建立了一個物品類別 `Item`，以便將物品按照單位價值進行排序。迴圈進行貪婪選擇，當背包已滿時跳出並返回解：

```rust
// === File: fractional_knapsack.rs ===

/* 物品 */
struct Item {
    w: i32, // 物品重量
    v: i32, // 物品價值
}

impl Item {
    fn new(w: i32, v: i32) -> Self {
        Self { w, v }
    }
}

/* 分數背包：貪婪 */
fn fractional_knapsack(wgt: &[i32], val: &[i32], mut cap: i32) -> f64 {
    // 建立物品串列，包含兩個屬性：重量、價值
    let mut items = wgt
        .iter()
        .zip(val.iter())
        .map(|(&w, &v)| Item::new(w, v))
        .collect::<Vec<Item>>();
    // 按照單位價值 item.v / item.w 從高到低進行排序
    items.sort_by(|a, b| {
        (b.v as f64 / b.w as f64)
            .partial_cmp(&(a.v as f64 / a.w as f64))
```

```
            .unwrap()
    });
    // 迴圈貪婪選擇
    let mut res = 0.0;
    for item in &items {
        if item.w <= cap {
            // 若剩餘容量充足，則將當前物品整個裝進背包
            res += item.v as f64;
            cap -= item.w;
        } else {
            // 若剩餘容量不足，則將當前物品的一部分裝進背包
            res += item.v as f64 / item.w as f64 * cap as f64;
            // 已無剩餘容量，因此跳出迴圈
            break;
        }
    }
    res
}
```

除排序之外，在最差情況下，需要走訪整個物品串列，**因此時間複雜度為 $O(n)$** ，其中 n 為物品數量。

由於初始化了一個 `Item` 物件串列，**因此空間複雜度為 $O(n)$** 。

3. 正確性證明

採用反證法。假設物品 x 是單位價值最高的物品，使用某演算法求得最大價值為 `res` ，但該解中不包含物品 x 。

現在從背包中拿出單位重量的任意物品，並替換為單位重量的物品 x 。由於物品 x 的單位價值最高，因此替換後的總價值一定大於 `res` 。**這與 `res` 是最優解矛盾，說明最優解中必須包含物品 x** 。

對於該解中的其他物品，我們也可以構建出上述矛盾。總而言之，**單位價值更大的物品總是更優選擇**，這說明貪婪策略是有效的。

如圖 15-6 所示，如果將物品重量和物品單位價值分別看作一張二維圖表的橫軸和縱軸，則分數背包問題可轉化為“求在有限橫軸區間下圍成的最大面積”。這個類比可以幫助我們從幾何角度理解貪婪策略的有效性。

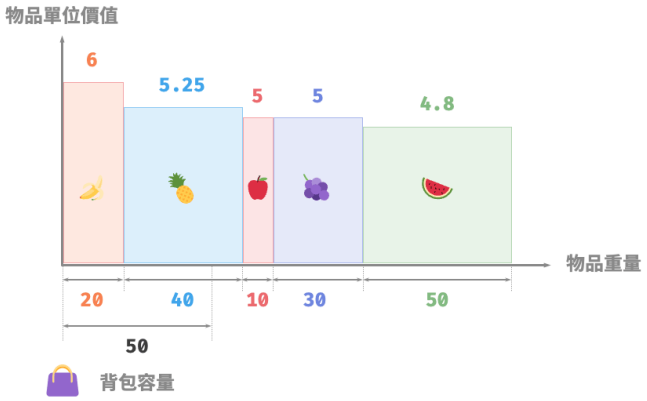


圖 15-6　分數背包問題的幾何表示

15.3　最大容量問題

Question

輸入一個陣列 ht，其中的每個元素代表一個垂直隔板的高度。陣列中的任意兩個隔板，以及它們之間的空間可以組成一個容器。

容器的容量等於高度和寬度的乘積（面積），其中高度由較短的隔板決定，寬度是兩個隔板的陣列索引之差。

請在陣列中選擇兩個隔板，使得組成的容器的容量最大，返回最大容量。示例如圖 15-7 所示。

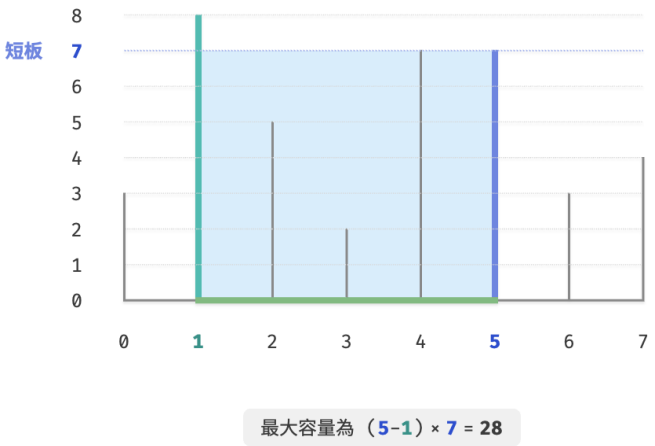


圖 15-7　最大容量問題的示例資料

容器由任意兩個隔板圍成，**因此本題的狀態為兩個隔板的索引，記為** $[i, j]$。

根據題意，容量等於高度乘以寬度，其中高度由短板決定，寬度是兩隔板的陣列索引之差。設容量為 $cap[i, j]$ ，則可得計算公式：

$$cap[i, j] = \min(ht[i], ht[j]) \times (j - i)$$

設陣列長度為 n ，兩個隔板的組合數量（狀態總數）為 $C_n^2 = \frac{n(n-1)}{2}$ 個。最直接地，**我們可以窮舉所有狀態**，從而求得最大容量，時間複雜度為 $O(n^2)$ 。

1. 貪婪策略確定

這道題還有更高效率的解法。如圖 15-8 所示，現選取一個狀態 $[i, j]$ ，其滿足索引 $i < j$ 且高度 $ht[i] < ht[j]$ ，即 i 為短板、j 為長板。

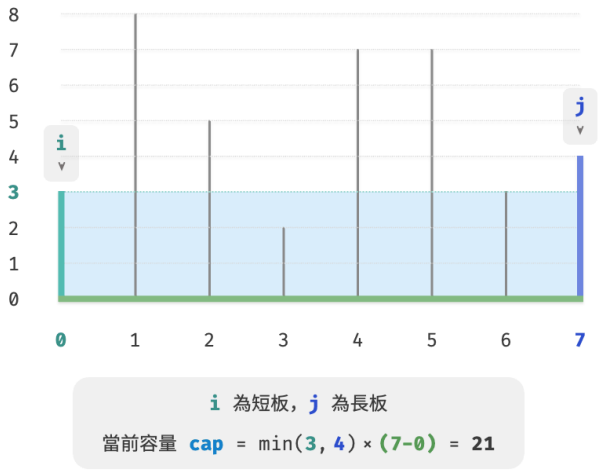


圖 15-8　初始狀態

如圖 15-9 所示，**若此時將長板 j 向短板 i 靠近，則容量一定變小**。

這是因為在移動長板 j 後，寬度 $j - i$ 肯定變小；而高度由短板決定，因此高度只可能不變（i 仍為短板）或變小（移動後的 j 成為短板）。

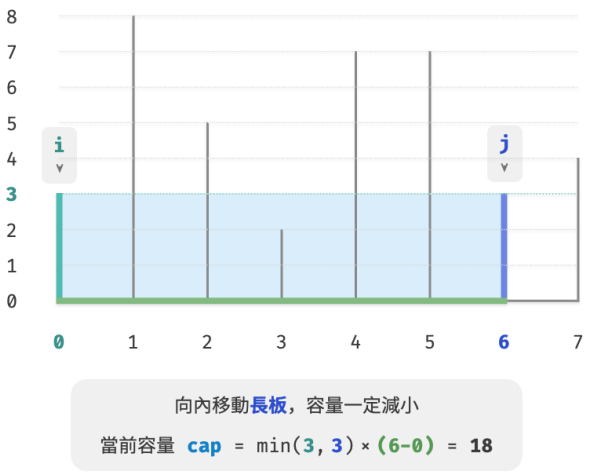


圖 15-9　向內移動長板後的狀態

反向思考，**我們只有向內收縮短板 i，才有可能使容量變大**。因為雖然寬度一定變小，**但高度可能會變大**（移動後的短板 i 可能會變長）。例如在圖 15-10 中，移動短板後面積變大。

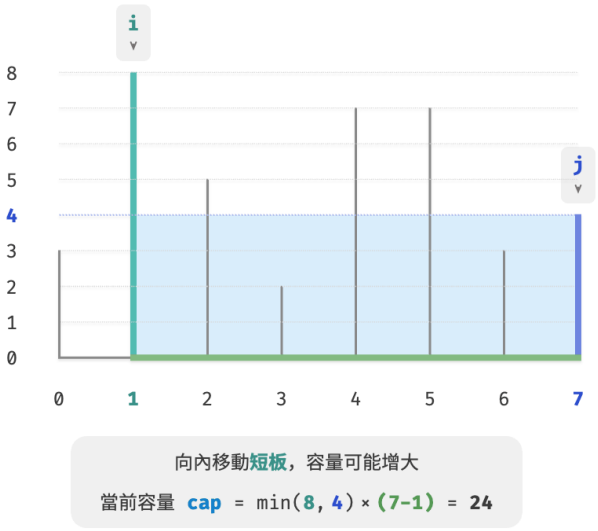


圖 15-10　向內移動短板後的狀態

由此便可推出本題的貪婪策略：初始化兩指標，使其分列容器兩端，每輪向內收縮短板對應的指標，直至兩指標相遇。

圖 15-11 展示了貪婪策略的執行過程。

1. 初始狀態下，指標 i 和 j 分列陣列兩端。
2. 計算當前狀態的容量 $cap[i, j]$，並更新最大容量。
3. 比較板 i 和板 j 的高度，並將短板向內移動一格。

4. 迴圈執行第 `2.` 步和第 `3.` 步，直至 i 和 j 相遇時結束。

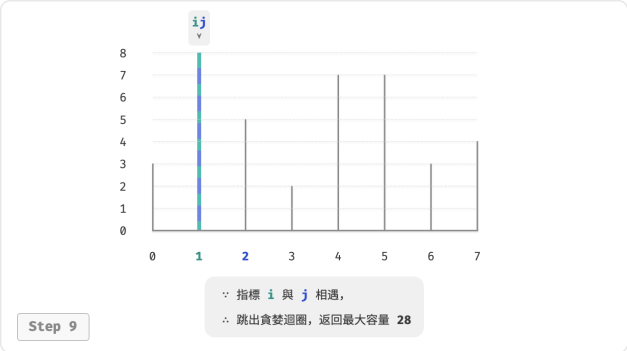


圖 15-11　最大容量問題的貪婪過程

2. 程式碼實現

程式碼迴圈最多 n 輪，**因此時間複雜度為 $O(n)$**。

變數 i、j、res 使用常數大小的額外空間，**因此空間複雜度為 $O(1)$**。

```
// === File: max_capacity.rs ===

/* 最大容量：貪婪 */
fn max_capacity(ht: &[i32]) -> i32 {
    // 初始化 i，j，使其分列陣列兩端
    let mut i = 0;
    let mut j = ht.len() - 1;
    // 初始最大容量為 0
    let mut res = 0;
    // 迴圈貪婪選擇，直至兩板相遇
    while i < j {
        // 更新最大容量
        let cap = std::cmp::min(ht[i], ht[j]) * (j - i) as i32;
        res = std::cmp::max(res, cap);
        // 向內移動短板
        if ht[i] < ht[j] {
            i += 1;
        } else {
            j -= 1;
        }
    }
    res
}
```

3. 正確性證明

之所以貪婪比窮舉更快，是因為每輪的貪婪選擇都會“跳過”一些狀態。

比如在狀態 $cap[i, j]$ 下，i 為短板、j 為長板。若貪婪地將短板 i 向內移動一格，會導致圖 15-12 所示的狀態被“跳過”。**這意味著之後無法驗證這些狀態的容量大小**。

$$cap[i, i+1], cap[i, i+2], \dots, cap[i, j-2], cap[i, j-1]$$

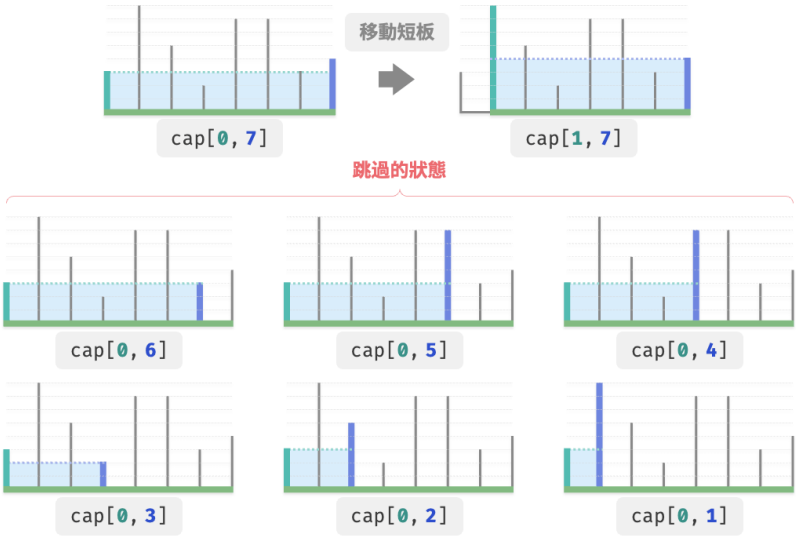


圖 15-12　移動短板導致被跳過的狀態

觀察發現，**這些被跳過的狀態實際上就是將長板 j 向內移動的所有狀態**。前面我們已經證明內移長板一定會導致容量變小。也就是說，被跳過的狀態都不可能是最優解，**跳過它們不會導致錯過最優解**。

以上分析說明，移動短板的操作是“安全”的，貪婪策略是有效的。

15.4　最大切分乘積問題

Question

給定一個正整數 n ，將其切分為至少兩個正整數的和，求切分後所有整數的乘積最大是多少，如圖 15-13 所示。

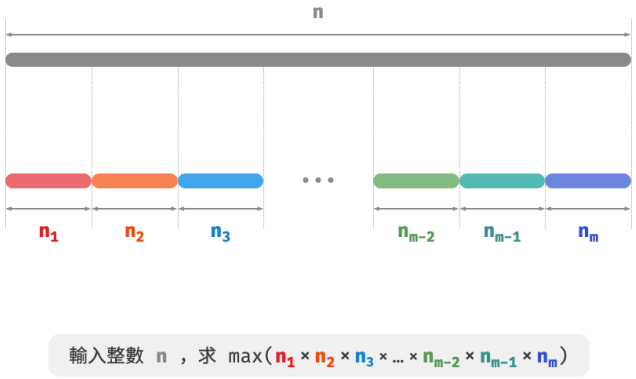


圖 15-13　最大切分乘積的問題定義

假設我們將 n 切分為 m 個整數因子，其中第 i 個因子記為 n_i ，即

$$n = \sum_{i=1}^{m} n_i$$

本題的目標是求得所有整數因子的最大乘積，即

$$\max(\prod_{i=1}^{m} n_i)$$

我們需要思考的是：切分數量 m 應該多大，每個 n_i 應該是多少？

1. 貪婪策略確定

根據經驗，兩個整數的乘積往往比它們的加和更大。假設從 n 中分出一個因子 2 ，則它們的乘積為 $2(n-2)$ 。我們將該乘積與 n 作比較：

$$\begin{aligned} 2(n-2) &\geq n \\ 2n - n - 4 &\geq 0 \\ n &\geq 4 \end{aligned}$$

如圖 15-14 所示，當 $n \geq 4$ 時，切分出一個 2 後乘積會變大，**這說明大於等於 4 的整數都應該被切分**。

貪婪策略一：如果切分方案中包含 ≥ 4 的因子，那麼它就應該被繼續切分。最終的切分方案只應出現 1、2、3 這三種因子。

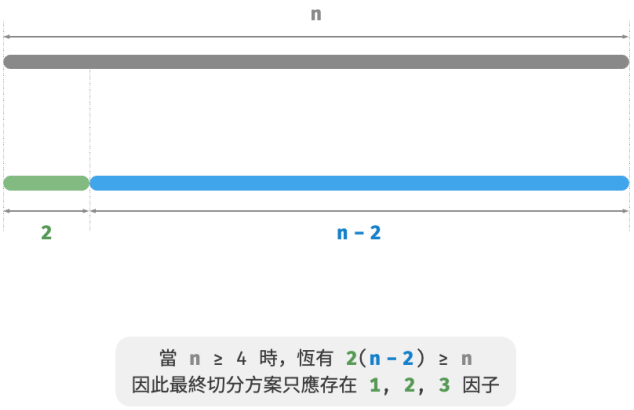


圖 15-14　切分導致乘積變大

接下來思考哪個因子是最優的。在 1、2、3 這三個因子中，顯然 1 是最差的，因為 $1 \times (n-1) < n$ 恆成立，即切分出 1 反而會導致乘積減小。

如圖 15-15 所示，當 $n = 6$ 時，有 $3 \times 3 > 2 \times 2 \times 2$。**這意味著切分出 3 比切分出 2 更優**。

貪婪策略二：在切分方案中，最多隻應存在兩個 2。因為三個 2 總是可以替換為兩個 3，從而獲得更大的乘積。

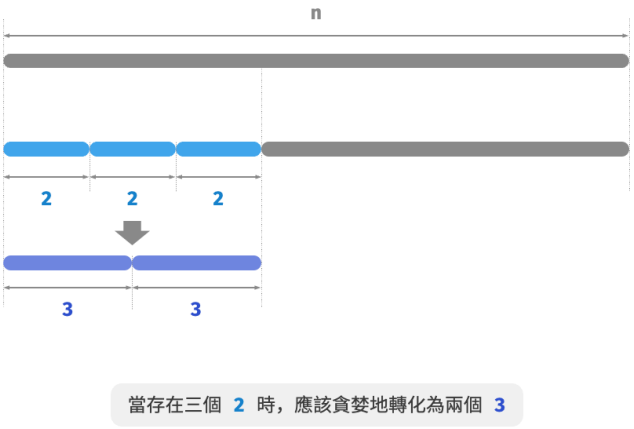


圖 15-15　最優切分因子

綜上所述，可推理出以下貪婪策略。

1. 輸入整數 n，從其不斷地切分出因子 3，直至餘數為 0、1、2。
2. 當餘數為 0 時，代表 n 是 3 的倍數，因此不做任何處理。
3. 當餘數為 2 時，不繼續劃分，保留。
4. 當餘數為 1 時，由於 $2 \times 2 > 1 \times 3$，因此應將最後一個 3 替換為 2。

2. 程式碼實現

如圖 15-16 所示，我們無須透過迴圈來切分整數，而可以利用向下整除運算得到 3 的個數 a ，用取模運算得到餘數 b ，此時有：

$$n = 3a + b$$

請注意，對於 $n \leq 3$ 的邊界情況，必須拆分出一個 1 ，乘積為 $1 \times (n - 1)$ 。

```rust
// === File: max_product_cutting.rs ===

/* 最大切分乘積：貪婪 */
fn max_product_cutting(n: i32) -> i32 {
    // 當 n <= 3 時，必須切分出一個 1
    if n <= 3 {
        return 1 * (n - 1);
    }
    // 貪婪地切分出 3 ，a 為 3 的個數，b 為餘數
    let a = n / 3;
    let b = n % 3;
    if b == 1 {
        // 當餘數為 1 時，將一對 1 * 3 轉化為 2 * 2
        3_i32.pow(a as u32 - 1) * 2 * 2
    } else if b == 2 {
        // 當餘數為 2 時，不做處理
        3_i32.pow(a as u32) * 2
    } else {
        // 當餘數為 0 時，不做處理
        3_i32.pow(a as u32)
    }
}
```

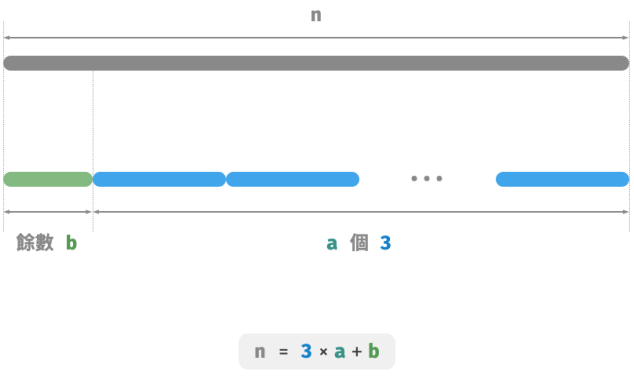


圖 15-16　最大切分乘積的計算方法

時間複雜度取決於程式語言的冪運算的實現方法。以 Python 為例，常用的冪計算函式有三種。

- 運算子 `**` 和函式 `pow()` 的時間複雜度均為 $O(\log\ a)$ 。
- 函式 `math.pow()` 內部呼叫 C 語言庫的 `pow()` 函式，其執行浮點取冪，時間複雜度為 $O(1)$ 。

變數 a 和 b 使用常數大小的額外空間，**因此空間複雜度為** $O(1)$ 。

3. 正確性證明

使用反證法，只分析 $n \geq 3$ 的情況。

1. **所有因子** ≤ 3 ：假設最優切分方案中存在 ≥ 4 的因子 x ，那麼一定可以將其繼續劃分為 $2(x-2)$ ，從而獲得更大的乘積。這與假設矛盾。
2. **切分方案不包含** 1 ：假設最優切分方案中存在一個因子 1 ，那麼它一定可以合併入另外一個因子中，以獲得更大的乘積。這與假設矛盾。
3. **切分方案最多包含兩個** 2 ：假設最優切分方案中包含三個 2 ，那麼一定可以替換為兩個 3 ，乘積更大。這與假設矛盾。

15.5 小結

- 貪婪演算法通常用於解決最最佳化問題，其原理是在每個決策階段都做出區域性最優的決策，以期獲得全域性最優解。
- 貪婪演算法會迭代地做出一個又一個的貪婪選擇，每輪都將問題轉化成一個規模更小的子問題，直到問題被解決。
- 貪婪演算法不僅實現簡單，還具有很高的解題效率。相比於動態規劃，貪婪演算法的時間複雜度通常更低。
- 在零錢兌換問題中，對於某些硬幣組合，貪婪演算法可以保證找到最優解；對於另外一些硬幣組合則不然，貪婪演算法可能找到很差的解。
- 適合用貪婪演算法求解的問題具有兩大性質：貪婪選擇性質和最優子結構。貪婪選擇性質代表貪婪策略的有效性。
- 對於某些複雜問題，貪婪選擇性質的證明並不簡單。相對來說，證偽更加容易，例如零錢兌換問題。
- 求解貪婪問題主要分為三步：問題分析、確定貪婪策略、正確性證明。其中，確定貪婪策略是核心步驟，正確性證明往往是難點。
- 分數背包問題在 0-1 背包的基礎上，允許選擇物品的一部分，因此可使用貪婪演算法求解。貪婪策略的正確性可以使用反證法來證明。
- 最大容量問題可使用窮舉法求解，時間複雜度為 $O(n^2)$ 。透過設計貪婪策略，每輪向內移動短板，可將時間複雜度最佳化至 $O(n)$ 。
- 在最大切分乘積問題中，我們先後推理出兩個貪婪策略：≥ 4 的整數都應該繼續切分，最優切分因子為 3 。程式碼中包含冪運算，時間複雜度取決於冪運算實現方法，通常為 $O(1)$ 或 $O(\log n)$ 。

第 16 章　附錄

本章內容

16.1　程式設計環境安裝

16.1.1　安裝 IDE

推薦使用開源、輕量的 VS Code 作為本地整合開發環境（IDE）。訪問 VS Code 官網，根據作業系統選擇相應版本的 VS Code 進行下載和安裝。

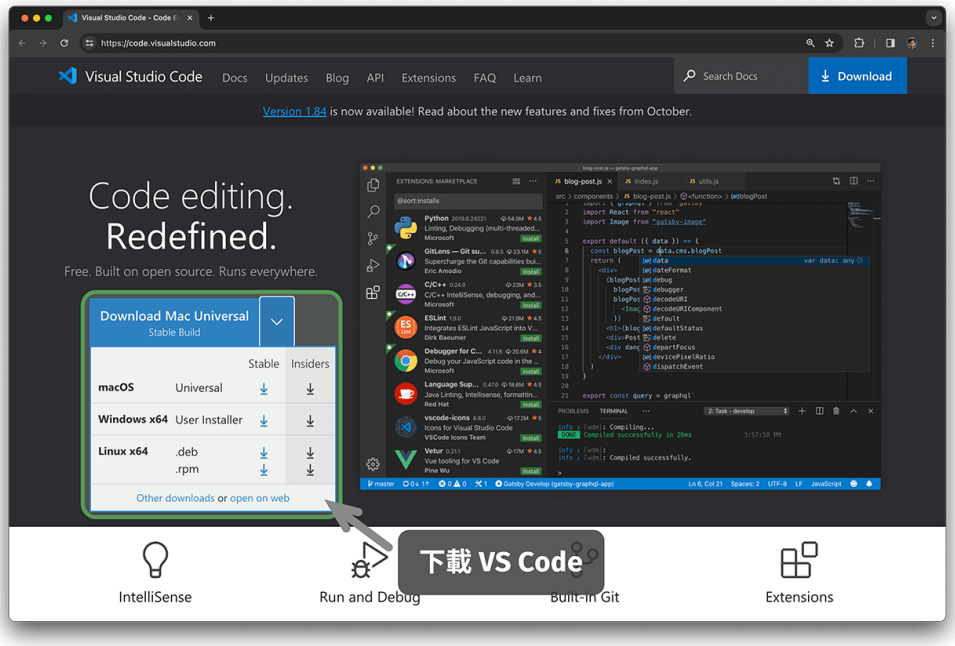


圖 16-1　從官網下載 VS Code

VS Code 擁有強大的擴展包生態系統，支持大多數程式語言的執行和除錯。以 Python 為例，安裝“Python Extension Pack”擴展包之後，即可進行 Python 程式碼除錯。安裝步驟如圖 16-2 所示。

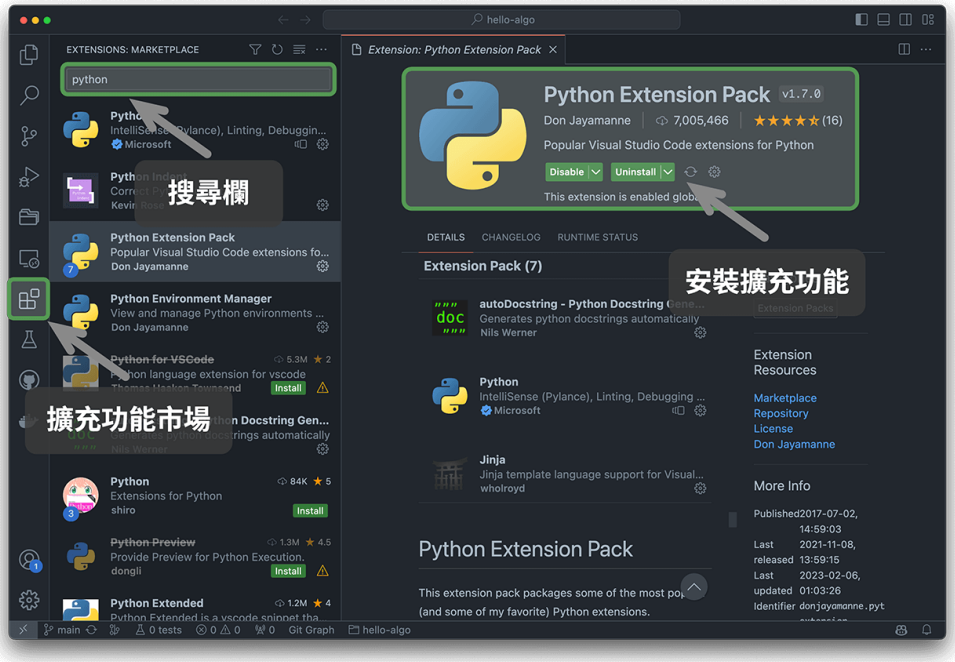


圖 16-2　安裝 VS Code 擴展包

16.1.2　安裝語言環境

1.　Python 環境

1. 下載並安裝 Miniconda3 ，需要 Python 3.10 或更新版本。
2. 在 VS Code 的擴充功能市場中搜索 `python` ，安裝 Python Extension Pack 。
3. （可選）在命令列輸入 `pip install black` ，安裝程式碼格式化工具。

2.　C/C++ 環境

1. Windows 系統需要安裝 MinGW （配置教程）；MacOS 自帶 Clang ，無須安裝。
2. 在 VS Code 的擴充功能市場中搜索 `c++` ，安裝 C/C++ Extension Pack 。
3. （可選）開啟 Settings 頁面，搜尋 `Clang_format_fallback Style` 程式碼格式化選項，設定為 `{ BasedOnStyle: Microsoft, BreakBeforeBraces: Attach }` 。

3.　Java 環境

1. 下載並安裝 OpenJDK （版本需滿足 > JDK 9）。
2. 在 VS Code 的擴充功能市場中搜索 `java` ，安裝 Extension Pack for Java 。

4. C# 環境

1. 下載並安裝 .Net 8.0 。
2. 在 VS Code 的擴充功能市場中搜索 `C# Dev Kit` ，安裝 C# Dev Kit （配置教程）。
3. 也可使用 Visual Studio（安裝教程）。

5. Go 環境

1. 下載並安裝 go 。
2. 在 VS Code 的擴充功能市場中搜索 `go` ，安裝 Go 。
3. 按快捷鍵 `Ctrl + Shift + P` 撥出命令欄，輸入 go ，選擇 `Go: Install/Update Tools` ，全部勾選並安裝即可。

6. Swift 環境

1. 下載並安裝 Swift 。
2. 在 VS Code 的擴充功能市場中搜索 `swift` ，安裝 Swift for Visual Studio Code 。

7. JavaScript 環境

1. 下載並安裝 Node.js 。
2. （可選）在 VS Code 的擴充功能市場中搜索 `Prettier` ，安裝程式碼格式化工具。

8. TypeScript 環境

1. 同 JavaScript 環境安裝步驟。
2. 安裝 TypeScript Execute (tsx) 。
3. 在 VS Code 的擴充功能市場中搜索 `typescript` ，安裝 Pretty TypeScript Errors 。

9. Dart 環境

1. 下載並安裝 Dart 。
2. 在 VS Code 的擴充功能市場中搜索 `dart` ，安裝 Dart 。

10. Rust 環境

1. 下載並安裝 Rust 。
2. 在 VS Code 的擴充功能市場中搜索 `rust` ，安裝 rust-analyzer 。

16.2　一起參與創作

由於筆者能力有限，書中難免存在一些遺漏和錯誤，請您諒解。如果您發現了筆誤、連結失效、內容缺失、文字歧義、解釋不清晰或行文結構不合理等問題，請協助我們進行修正，以給讀者提供更優質的學習資源。

所有撰稿人的 GitHub ID 將在本書倉庫、網頁版和 PDF 版的主頁上進行展示，以感謝他們對開源社群的無私奉獻。

> **開源的魅力**
> 紙質圖書的兩次印刷的間隔時間往往較久，內容更新非常不方便。
> 而在本開源書中，內容更迭的時間被縮短至數日甚至幾個小時。

1.　內容微調

如圖 16-3 所示，每個頁面的右上角都有“編輯圖示”。您可以按照以下步驟修改文字或程式碼。

1. 點選“編輯圖示”，如果遇到“需要 Fork 此倉庫”的提示，請同意該操作。
2. 修改 Markdown 源檔案內容，檢查內容的正確性，並儘量保持排版格式的統一。
3. 在頁面底部填寫修改說明，然後點選“Propose file change”按鈕。頁面跳轉後，點選“Create pull request”按鈕即可發起拉取請求。

圖 16-3　頁面編輯按鍵

圖片無法直接修改，需要透過新建 Issue 或評論留言來描述問題，我們會盡快重新繪製並替換圖片。

2. 內容創作

如果您有興趣參與此開源專案，包括將程式碼翻譯成其他程式語言、擴展文章內容等，那麼需要實施以下 Pull Request 工作流程。

1. 登入 GitHub ，將本書的程式碼倉庫 Fork 到個人帳號下。
2. 進入您的 Fork 倉庫網頁，使用 `git clone` 命令將倉庫克隆至本地。
3. 在本地進行內容創作，並進行完整測試，驗證程式碼的正確性。
4. 將本地所做更改 Commit ，然後 Push 至遠端倉庫。
5. 重新整理倉庫網頁，點選“Create pull request”按鈕即可發起拉取請求。

3. Docker 部署

在 `hello-algo` 根目錄下，執行以下 Docker 指令碼，即可在 `http://localhost:8000` 訪問本專案：

```
docker-compose up -d
```

使用以下命令即可刪除部署：

```
docker-compose down
```

16.3 術語表

表 16-1 列出了書中出現的重要術語，值得注意以下幾點。

- 建議記住名詞的英文叫法，以便閱讀英文文獻。
- 部分名詞在簡體中文和繁體中文下的叫法不同。

表 16-1　資料結構與演算法的重要名詞

English	簡體中文	繁體中文
algorithm	算法	演算法
data structure	数据结构	資料結構
code	代码	程式碼
file	文件	檔案
function	函数	函式
method	方法	方法
variable	变量	變數
asymptotic complexity analysis	渐近复杂度分析	漸近複雜度分析

English	簡體中文	繁體中文
time complexity	时间复杂度	時間複雜度
space complexity	空间复杂度	空間複雜度
loop	循环	迴圈
iteration	迭代	迭代
recursion	递归	遞迴
tail recursion	尾递归	尾遞迴
recursion tree	递归树	遞迴樹
big-O notation	大 O 记号	大 O 記號
asymptotic upper bound	渐近上界	漸近上界
sign-magnitude	原码	原碼
1’s complement	反码	一補數
2’s complement	补码	二補數
array	数组	陣列
index	索引	索引
linked list	链表	鏈結串列
linked list node, list node	链表节点	鏈結串列節點
head node	头节点	頭節點
tail node	尾节点	尾節點
list	列表	串列
dynamic array	动态数组	動態陣列
hard disk	硬盘	硬碟
random-access memory (RAM)	内存	記憶體
cache memory	缓存	快取
cache miss	缓存未命中	快取未命中
cache hit rate	缓存命中率	快取命中率
stack	栈	堆疊
top of the stack	栈顶	堆疊頂
bottom of the stack	栈底	堆疊底
queue	队列	佇列
double-ended queue	双向队列	雙向佇列
front of the queue	队首	佇列首
rear of the queue	队尾	佇列尾
hash table	哈希表	雜湊表
hash set	哈希集合	雜湊集合
bucket	桶	桶

English	簡體中文	繁體中文
hash function	哈希函数	雜湊函式
hash collision	哈希冲突	雜湊衝突
load factor	负载因子	負載因子
separate chaining	链式地址	鏈結位址
open addressing	开放寻址	開放定址
linear probing	线性探测	線性探查
lazy deletion	懒删除	懶刪除
binary tree	二叉树	二元樹
tree node	树节点	樹節點
left-child node	左子节点	左子節點
right-child node	右子节点	右子節點
parent node	父节点	父節點
left subtree	左子树	左子樹
right subtree	右子树	右子樹
root node	根节点	根節點
leaf node	叶节点	葉節點
edge	边	邊
level	层	層
degree	度	度
height	高度	高度
depth	深度	深度
perfect binary tree	完美二叉树	完美二元樹
complete binary tree	完全二叉树	完全二元樹
full binary tree	完满二叉树	完滿二元樹
balanced binary tree	平衡二叉树	平衡二元樹
binary search tree	二叉搜索树	二元搜尋樹
AVL tree	AVL 树	AVL 樹
red-black tree	红黑树	紅黑樹
level-order traversal	层序遍历	層序走訪
breadth-first traversal	广度优先遍历	廣度優先走訪
depth-first traversal	深度优先遍历	深度優先走訪
binary search tree	二叉搜索树	二元搜尋樹
balanced binary search tree	平衡二叉搜索树	平衡二元搜尋樹
balance factor	平衡因子	平衡因子
heap	堆	堆積

English	簡體中文	繁體中文
max heap	大顶堆	大頂堆積
min heap	小顶堆	小頂堆積
priority queue	优先队列	優先佇列
heapify	堆化	堆積化
top-k problem	Top-k 问题	Top-k 問題
graph	图	圖
vertex	顶点	頂點
undirected graph	无向图	無向圖
directed graph	有向图	有向圖
connected graph	连通图	連通圖
disconnected graph	非连通图	非連通圖
weighted graph	有权图	有權圖
adjacency	邻接	鄰接
path	路径	路徑
in-degree	入度	入度
out-degree	出度	出度
adjacency matrix	邻接矩阵	鄰接矩陣
adjacency list	邻接表	鄰接表
breadth-first search	广度优先搜索	廣度優先搜尋
depth-first search	深度优先搜索	深度優先搜尋
binary search	二分查找	二分搜尋
searching algorithm	搜索算法	搜尋演算法
sorting algorithm	排序算法	排序演算法
selection sort	选择排序	選擇排序
bubble sort	冒泡排序	泡沫排序
insertion sort	插入排序	插入排序
quick sort	快速排序	快速排序
merge sort	归并排序	合併排序
heap sort	堆排序	堆積排序
bucket sort	桶排序	桶排序
counting sort	计数排序	計數排序
radix sort	基数排序	基數排序
divide and conquer	分治	分治
hanota problem	汉诺塔问题	河內塔問題
backtracking algorithm	回溯算法	回溯演算法

English	簡體中文	繁體中文
constraint	约束	約束
solution	解	解
state	状态	狀態
pruning	剪枝	剪枝
permutations problem	全排列问题	全排列問題
subset-sum problem	子集和问题	子集合問題
n-queens problem	n 皇后问题	n 皇后問題
dynamic programming	动态规划	動態規劃
initial state	初始状态	初始狀態
state-transition equation	状态转移方程	狀態轉移方程
knapsack problem	背包问题	背包問題
edit distance problem	编辑距离问题	編輯距離問題
greedy algorithm	贪心算法	貪婪演算法

“Among the universe's 200 billion galaxies, encountering you, a shining star, is this book's great fortune.”

“在宇宙的 2000 億個星系中，遇到你這顆閃耀的星星，是這本書的巨大幸運”