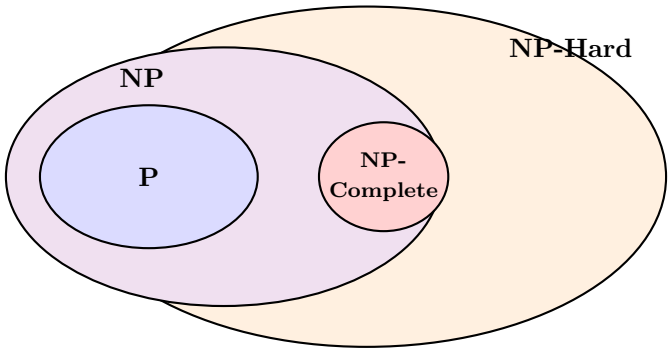
NP、NP-Complete 與 NP-Hard

完整白話教材

Computational Complexity —A Plain-Language Guide

從「為什麼有些問題算不動」出發，
一路講到歸約、經典難題圖鑑、精確解、近似演算法、
啟發式方法，以及真實世界如何與 NP 共處



演算法教學系列

內附完整、已編譯驗證的 C++17 實作（13 個經典演算法）

假設 P ≠ NP 的世界地圖（多數學者相信的版本）

Contents

前言：這份教材要回答的三個問題

寫程式的人遲早會撞到一種牆：問題明明「聽起來很簡單」，程式卻怎麼寫都跑不完。例如「把 40 個包裹分給兩台車，讓兩台車的重量儘量平均」——聽起來像小學數學，但暴力枚舉有 $2^{40} \approx 10^{12}$ 種分法；再快的電腦，指數成長都能讓它跪下。

這面牆有個名字，叫 **NP-Hard**。本教材用白話回答三個問題：

1. **這面牆是什麼？**——P、NP、NP-Complete、NP-Hard 到底在說什麼？（第 1–4 章，全部用比喻與例子講，數學符號降到最低）
2. **撞牆之後怎麼辦？**——四條實戰出路：聰明的暴力（精確解）、有品質保證的妥協（近似演算法）、賭運氣但通常很準（啟發式）、以及「先檢查你的問題是不是特例」（回到 P）。（第 5–9 章）
3. **真實世界怎麼與 NP 共處？**——晶片驗證、物流路線、排班、密碼學，每天都在跟 NP-Hard 問題打交道，而且活得很好。（第 10 章）

本教材的特色

每個概念先給**白話比喻**，再給嚴謹定義；每個演算法都有對應的 **已實際編譯執行通過**的 C++17 程式碼（第 11 章完整列出，共 13 個演算法），並附上執行結果。建議搭配同系列的《圖論演算法完整教材》閱讀——許多 NP-Complete 問題都定義在圖上。

Chapter 1

為什麼有些問題「算不動」？

1.1 兩種成長速度：多項式 vs. 指數

演算法的核心貨幣是**時間隨輸入大小的成長速度**。同樣是「變慢」，n^2 跟 2^n 是兩個完全不同的世界：

n	n^2	n^3	2^n	$n!$
10	100	1,000	1,024	363 萬
20	400	8,000	104 萬	2.4×10^{18}
50	2,500	12.5 萬	10^{15}	3×10^{64}
100	1 萬	100 萬	10^{30}	9×10^{157}

假設電腦每秒做 10^9 次運算：$n = 100$ 時 n^3 只要 0.001 秒，而 2^n 要 10^{13} **年**——宇宙年齡的一千倍。這就是為什麼理論界把「**多項式時間**（polynomial time）」當作「有效率、可實用」的分界線：多項式再大也追得上，指數永遠追不上。

白話比喻

把多項式演算法想成「搭高鐵」，指數演算法想成「用走的」。短程（n 小）兩者差不多，但距離一拉長，走路的人永遠到不了。「演算法快不快」問的不是今天跑多久，而是**資料變大十倍時，你會慢多少倍**。

1.2 枚舉的詛咒：組合爆炸

很多實際問題的「候選解」數量是指數級的：

- 40 個物品「選或不選」：$2^{40} \approx 10^{12}$ 種組合。
- 20 個城市的巡迴順序：$19! \approx 10^{17}$ 條路線。
- 100 個布林變數的真假指派：$2^{100} \approx 10^{30}$ 種。

這些問題有個共同特徵：**檢查一個候選解對不對很快**（把路線長度加一加、把公式代入驗證），**但候選解多到不可能全部看完**。「驗證容易、求解疑似困難」——這正是 NP 理論要形式化的現象。

1.3 一個具體例子：拼圖 vs. 檢查拼圖

千片拼圖

給你一盒 1000 片的拼圖，拼好它可能要花整個週末；但如果朋友拿一幅「已拼好」的拼圖給你看，你掃一眼（每片跟鄰居接得起來嗎？）幾分鐘就能確認他沒亂拼。**解很難找，但答案很好驗**——NP 問題全長這樣。

於是自然冒出一個看似天真、實則是電腦科學最深的問題：

「既然驗證這麼快，有沒有可能其實『求解』也能一樣快，只是我們還沒想到方法 ?」

這就是價值一百萬美元的 **P vs. NP** 問題（第 2 章）。

Chapter 2

P、NP、NP-Complete、NP-Hard 白話定義

2.1 先說清楚：我們談的是「決策問題」

複雜度理論的標準舞台是**決策問題**（decision problem）——答案只有「是／否」的問題。

- 優化版：「最短的巡迴路線是多長？」
- 決策版：「存在長度 $\leq K$ 的巡迴路線嗎？」

兩者實務上等價（用二分搜尋 K 就能從決策版逼出優化版），但理論上用決策版比較乾淨。下文的 P、NP 都是「決策問題的集合」。

2.2 P：解得快

定義 2.1 (P)**. P** 是所有「存在多項式時間演算法可以解」的決策問題集合。即：有一個演算法，對大小為 n 的輸入，最多花 $c \cdot n^k$ 步就給出正確的是／否。

排序、最短路徑、最小生成樹、二分圖匹配、質數判定、2-SAT——都在 P。白話：**P = 我們已經知道怎麼有效率解決的問題**。

2.3 NP：驗得快

定義 2.2 (NP)**. NP** 是所有「若答案為是，存在一個**證據**（certificate），可以在多項式時間內**驗證**」的決策問題集合。NP 的全名是 Nondeterministic Polynomial time。

出題老師與改考卷

NP 問題就像改考卷：你自己解這題可能要很久，但學生把完整過程寫出來（= 證據），你照著檢查一遍很快。「存在 $\leq K$ 的巡迴」的證據就是那條路線本身——把長度加總一比對就好。

常見誤解

NP 不是「Non-Polynomial（非多項式）」！這是最常見的誤解。NP 說的是「**驗證**只要多項式時間」，完全沒說求解一定很慢。事實上所有 P 問題都屬於 NP（自己解一遍就是驗證），所以 $P \subseteq NP$。

2.4 NP-Complete：NP 裡最難的一批

定義 2.3 (NP-Complete)**.** 問題 X 是 **NP-Complete**（NP 完全），若：(1) $X \in NP$；且 (2) NP 裡的**每一個**問題都能在多項式時間內**歸約**成 X（見第 3 章）。

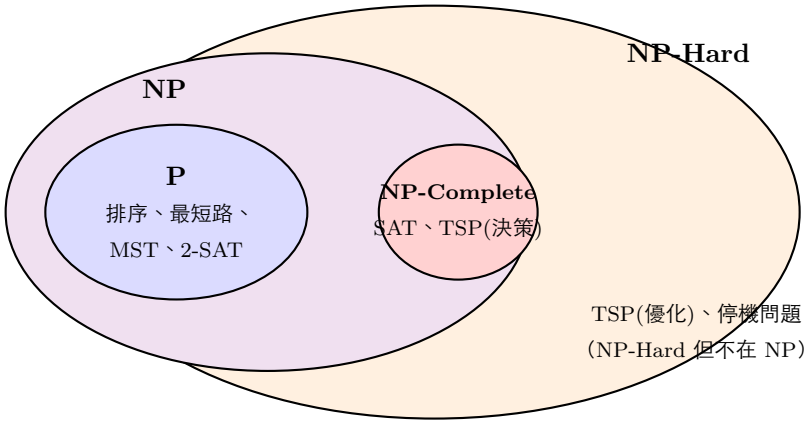
白話：NP-Complete 問題是 NP 世界的「萬能鑰匙孔」——任何 NP 問題都能翻譯成它。因此只要**任何一個** NP-Complete 問題找到多項式解法，**整個 NP 全部**瞬間變成 P，世界改寫（$P = NP$）。反過來，只要你相信 $P \neq NP$，那所有 NP-Complete 問題都沒有多項式解法。

2.5 NP-Hard：至少跟 NP-Complete 一樣難

定義 2.4 (NP-Hard)**.** 問題 Y 是 **NP-Hard**，若 NP 裡每個問題都能多項式歸約成 Y。注意：**不要求 Y 本身屬於 NP**——Y 甚至可以不是決策問題。

例如「TSP 優化版（求最短巡迴）」是 NP-Hard：它不是是／否問題，嚴格說不在 NP 裡，但它顯然不比決策版容易。極端例子如停機問題（Halting Problem）也是 NP-Hard——它根本不可計算。

2.6 一張圖看懂四個類別



類別	白話	代表
P	解得快	排序、最短路、MST
NP	驗得快（解不一定快）	所有 P + 一大堆難題
NP-Complete	NP 裡最難、彼此等價	SAT、著色、Hamilton
NP-Hard	至少跟 NPC 一樣難，可能更難	TSP 優化、停機問題

2.7 P vs. NP：百萬美元懸案

「$P = NP$ 嗎？」是克雷數學研究所七大**千禧年大獎難題**之一，懸賞一百萬美元，至今（2026 年）仍未解決。學界壓倒性地相信 $P \neq NP$ ——「找解」本質上就是比「驗解」難——但沒有人能證明。已知的證明技巧（相對化、自然證明）都被證明「不足以」解決這題，這也是它難的原因之一。

如果有一天 P

現代密碼學（RSA、橢圓曲線）都建立在「某些問題難解」的假設上。若 $P = NP$ 且演算法實用，加密會崩潰、但同時蛋白質摺疊、最佳化設計、自動定理證明都將迎刃而解——一半是災難片、一半是科幻片。所幸多數專家認為這一天不會來。

Chapter 3

歸約：難度的「翻譯機」

3.1 歸約的白話意思

定義 3.1 (多項式時間歸約)**.** 問題 A **歸約**到問題 B（記作 $A \leq_p B$），意思是：存在一個多項式時間的「翻譯程式」，把 A 的任何輸入轉成 B 的輸入，且兩者答案一致。於是「會解 B」就自動「會解 A」。

翻譯機比喻

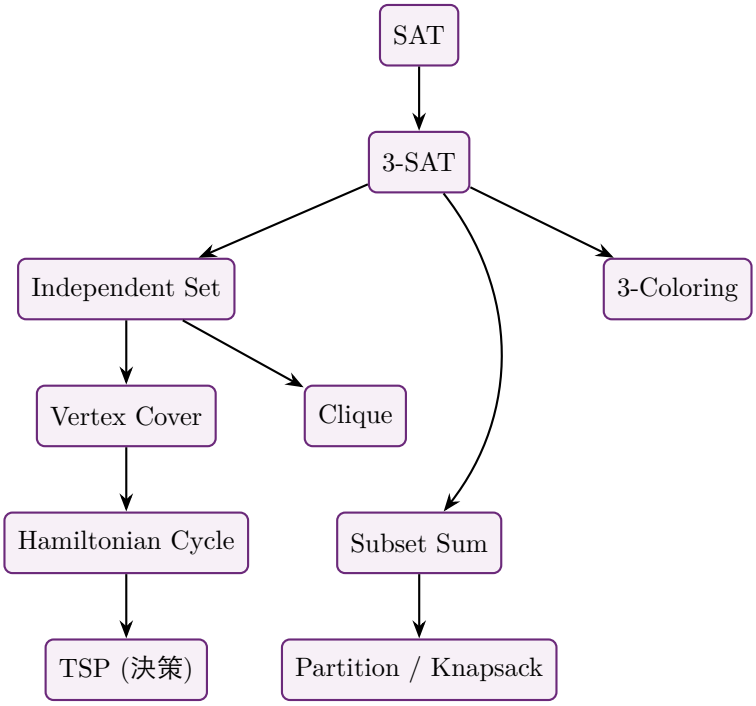
你不會講法文，但你有「中翻法」翻譯機，而且你朋友會解法文謎題。那麼任何中文謎題你都能解：先翻譯、再請朋友解。所以**法文謎題至少跟中文謎題一樣難**——這就是 $A \leq_p B$ 時「B 不比 A 簡單」的意思。歸約的方向常讓初學者暈：**把 A 歸約到 B，證明的是 B 難（若 A 已知難）**。

3.2 Cook–Levin 定理：第一把萬能鑰匙

定理 3.1 (Cook–Levin, 1971)**. SAT**（布林可滿足性問題）是 NP-Complete。

白話：任何 NP 問題的「驗證過程」都能被編碼成一條巨大的布林公式，使得「公式可滿足 $\iff$ 原問題答案為是」。這是史上第一個被證明的 NP-Complete 問題——有了第一個，之後只要「已知 NPC 問題 $\leq_p$ 新問題」，新問題就也是 NPC。Karp 在 1972 年一口氣用歸約證明了 21 個經典問題都是 NPC，開啟了整個領域。

3.3 經典歸約鏈



箭頭 $A \to B$ 表示「A 歸約到 B，因此 B 也是 NP-Complete」。所有 NPC 問題彼此都可互相歸約——它們是**同一種難**的不同化身。

3.4 體驗一個歸約：Independent Set $\leq_p$ Vertex Cover

這是最漂亮的入門歸約，一行就講完：

定理 3.2. 在 n 個頂點的圖中，S 是獨立集 $\iff$ $V \setminus S$ 是頂點覆蓋。因此「存在大小 $\geq k$ 的獨立集」$\iff$「存在大小 $\leq n-k$ 的頂點覆蓋」。

直覺：獨立集內部沒有任何邊，所以每條邊至少有一端在集合外——集合外的點恰好「覆蓋」了所有邊。翻譯程式只需要把 k 換成 $n-k$，顯然是多項式時間。兩題難度綁定，一起難、一起 easy。

實戰價值

歸約不只是理論遊戲。實務上它反過來用：把你的怪問題**歸約成 SAT**，再丟給工業級 SAT solver（第 8 章）——這是現代硬體驗證、排程系統的標準做法。「會建模」比「會寫搜尋」更值錢。

Chapter 4

經典 NP-Complete 問題圖鑑

以下每個問題都給：白話定義、一句話記憶點、真實應用。它們是面試、競賽與論文中的常客。

4.1 邏輯類

SAT（布林可滿足性）

給一條布林公式，如 $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3)$，問：有沒有一組真假指派讓整條公式為真？**記憶點**：NPC 的始祖（Cook–Levin）。**應用**：晶片等價性驗證、軟體模型檢查、排程。

3-SAT

限制每個子句恰好 3 個文字的 SAT。依然 NPC，且因為結構整齊，是**最常拿來當歸約起點**的問題。（對照：2-SAT 每子句 2 個文字，卻掉回 P——見第 9 章。）

4.2 圖論類

Vertex Cover（頂點覆蓋）

選最少的頂點，使每條邊至少有一端被選中。**比喻**：在每條走廊都看得到的位置裝最少的監視器。**應用**：感測器佈點、測試案例挑選。

Independent Set（獨立集）

選最多的頂點，任兩個都不相鄰。**比喻**：宴會排座位，互相有過節的人不能同桌。與 Vertex Cover 互補（S 獨立 $\iff$ $V \setminus S$ 覆蓋）。

Clique（最大團）

找最大的「彼此全認識」子圖。**應用**：社群偵測、蛋白質交互作用分析。與 Independent Set 在補圖上等價。

Graph Coloring（圖著色，$k \geq 3$）

用 k 種顏色塗頂點，相鄰不同色。**應用**：排課／排考（衝突的課不同時段）、編譯器暫存器配置、無線頻譜分配。注意 $k = 2$（二分圖判定）在 P。

Hamiltonian Path / Cycle

一條「恰好經過每個頂點一次」的路徑／迴圈。**對照組**：歐拉路徑（每條邊恰一次）在 P！一字之差，難度天壤之別。

4.3 數字與組合類

Subset Sum（子集和）

一堆整數裡，能否挑出一個子集恰好加到目標值 T？**應用**：對帳（哪幾筆交易加起來是這個總額？）。

Partition（分割）

能否把一堆數分成總和相等的兩半？是 Subset Sum 取 $T =$ 總和/2 的特例，仍 NPC。**比喻**：兩台貨車載重平均。

0/1 Knapsack（背包，決策版）

容量 W 的背包、每件物品有重量與價值，能否裝出價值 $\geq V$？**重要註腳**：有 $O(nW)$ 的 DP——但這是**偽多項式**（見第 6 章），不與 NPC 矛盾。

Bin Packing（裝箱）

把不同大小的物品裝進容量固定的箱子，最少用幾箱？**應用**：貨櫃裝載、雲端虛擬機配置、廣告版位。

Set Cover（集合覆蓋）

給一堆集合，選最少個把全集蓋滿。**應用**：選最少的測試涵蓋所有功能、消防站選址。

4.4 路徑類

TSP（旅行推銷員）

訪問每個城市恰一次並回到起點，總距離最短。決策版 NPC；優化版 NP-Hard。**應用**：物流配送、電路板鑽孔路徑、基因定序排程。

4.5 速查表

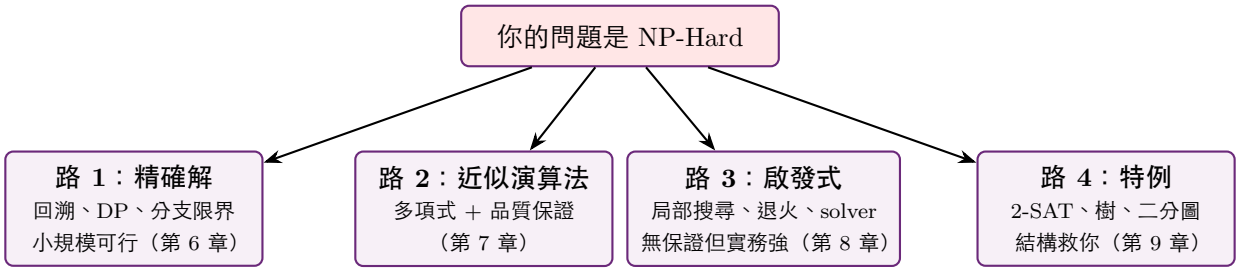
問題	一句話	在 P 的近親
SAT / 3-SAT	公式能為真嗎	2-SAT（SCC 解）
Vertex Cover	最少點守住所有邊	二分圖上 = 最大匹配（König）
Independent Set	最多互不相鄰	樹上 DP 可解
Clique	最大全連通子圖	—
Coloring ($k \geq 3$)	相鄰不同色	$k = 2$ 二分圖判定

Hamiltonian	每點恰一次	歐拉路徑（每邊恰一次）
Subset Sum	挑數加到 T	值域小 -> 偽多項式 DP
Knapsack	有限容量最大價值	$O(nW)$ DP
Bin Packing	最少箱子	—
Set Cover	最少集合蓋全	—
TSP	最短巡迴	度量空間有近似保證

Chapter 5

撞牆之後的四條路

證明（或強烈懷疑）你的問題是 NP-Hard 之後，**不代表舉手投降**。實務上有四條路，後面四章各講一條：



實戰決策順序

先走**路 4**（我的輸入有特殊結構嗎？規模多大？參數小嗎？），再評估**路 1**（$n \leq 20$ 可位元壓縮 DP、$n \leq 40$ 可 meet-in-the-middle），需要品質保證走**路 2**，追求實戰效果走**路 3**（現代 solver 強得驚人）。

Chapter 6

路 1：精確解——聰明的暴力

精確解仍是指數時間，但透過**剪枝**與**記憶化**，實務可解的規模遠比天真枚舉大。所有程式碼見第 11 章。

6.1 回溯 + 剪枝：N-Queens

回溯法（backtracking）= 深度優先枚舉 + 「一發現不可行就立刻退回」。N-Queens（在 $n \times n$ 棋盤放 n 個互不攻擊的皇后）是教科書範例：

- 逐列放置：每列恰一個皇后，衝突只需檢查「直行、兩條對角線」。
- 用三個布林陣列，$O(1)$ 判斷衝突；一衝突整棵子樹直接跳過。

實測：$n = 8$ 得 92 解、$n = 10$ 得 724 解，瞬間完成——剪枝把 $8^8 \approx 1600$ 萬的枚舉空間砍到只剩幾千個節點。

6.2 DPLL：SAT 求解器的雛形

DPLL（Davis–Putnam–Logemann–Loveland, 1962）= 回溯 + 兩條關鍵推理：

- **單元傳播**（unit propagation）：某子句只剩一個未定文字時，它**被迫**為真——連鎖反應常能瞬間定下大量變數。
- **提早偵測矛盾**：出現空子句立刻回溯。

現代工業 SAT solver 的核心 **CDCL**（Conflict-Driven Clause Learning）就是 DPLL 加上「從衝突中學出新子句、非時序回跳」等強化，實務能處理**數百萬變數**的實例（第 10 章）。

6.3 偽多項式 DP：Knapsack 與 Subset Sum

Knapsack 有經典 DP：$dp[c]$ = 容量 c 下的最大價值，時間 $O(nW)$。「咦？這不是多項式嗎？」——不是。

偽多項式（pseudo-polynomial）

$O(nW)$ 中的 W 是**數值**，不是輸入長度。輸入用二進位寫，W 的位數只有 $\log W$；$W = 2^{60}$ 時輸入才 60 個位元，$O(nW)$ 卻是天文數字。對「輸入長度」而言它仍是指數。這種演算法叫**偽多項式**：值域小時實用，值域大時破功。Knapsack 這類「值域小就能解」的問題稱為**弱 NP-hard**。

6.4 Meet-in-the-Middle：$2^n \to 2^{n/2}$

Subset Sum 在值域巨大時 DP 失效，但可以**對半分**：枚舉左半所有子集和（$2^{n/2}$ 個）、右半也枚舉並排序，再對每個左半的和二分搜尋「互補值」。$n = 40$ 時從 10^{12} 降到約 2×10^6——瞬間可解。空間換時間的經典。

6.5 位元壓縮 DP：Hamiltonian 與 TSP（Held–Karp）

用一個整數的位元表示「已拜訪的城市集合」：

$$dp[\text{mask}][v] = \text{走過集合 mask、目前在城市} v \text{ 的最短距離}$$

轉移即嘗試下一個未拜訪城市。時間 $O(n^2 2^n)$、空間 $O(n2^n)$——$n = 20$ 約 4 億次運算，秒級可解。比 $n!$ 枚舉快了 10^{11} 倍。這是「指數但聰明」的代表作：實測 4 城市經典例最佳解 80，正確。

6.6 分支限界與 FPT：Vertex Cover

「覆蓋數 $\le k$?」有個優雅的二分支：任取一條邊 (u, v)，**u 或 v 必有一個要選**——分兩支遞迴，深度最多 k。時間 $O(2^k \cdot \text{poly}(n))$：對**輸入大小**指數在 k 上，n 再大、只要 k 小就跑得動。這叫**參數化複雜度／ FPT**（Fixed-Parameter Tractable），是理論界對付 NP-Hard 的第五條暗路。（目前最好的 Vertex Cover FPT 約 $O(1.25^k + kn)$。）

Chapter 7

路 2：近似演算法——妥協，但有收據

7.1 近似比：品質保證書

定義 7.1 (α-近似演算法)**.** 一個多項式時間演算法若保證輸出 $\leq \alpha \cdot OPT$（最小化問題），就叫 **α-近似演算法**。α 稱為近似比。

白話：我不給你最好的，但**白紙黑字保證**離最好差不到 α 倍。這跟啟發式的差別就在這張「收據」。

7.2 Vertex Cover 的 2-近似：漂亮到不像話

1. 隨便找一條還沒被覆蓋的邊 (u, v)，**兩個端點都選**。
2. 刪掉所有被覆蓋的邊，重複直到沒有邊。

為什麼是 2-近似？我們選的邊彼此不共點（是一組匹配），最優解對每條這種邊**至少要選 1 個**端點，我們選了 2 個——所以最多是最優的 2 倍。三行演算法、一行證明。（實測：5 邊小圖輸出 4 個點，OPT 為 2，正好卡在保證線上。）

理論冷知識

在「唯一賽局猜想」（Unique Games Conjecture）成立下，2 就是 Vertex Cover 能做到的最佳近似比——這個樸素演算法竟是理論最優。

7.3 Set Cover 的貪婪 $\ln n$-近似

每一輪選「能新覆蓋最多元素」的集合，直到蓋滿。可證明近似比為 $H_n \approx \ln n$，而且（除非 $P = NP$）**沒有演算法能做得更好**——貪婪就是極限。實務中它常遠比理論保證好。

7.4 Bin Packing 的 First-Fit Decreasing

物品由大到小排序，逐一放進「第一個放得下的箱子」。保證用箱數 $\leq \frac{11}{9}OPT + 1$。直覺：大物品先卡位，小物品填縫，浪費不會太多。實測 10 件物品用 5 箱，正好等於下界 $\lceil$總體積$\rceil$——碰到最優。

7.5 Metric TSP：MST 2-近似與 Christofides 1.5-近似

當距離滿足三角不等式（繞路不會更近）時：

- **MST 2-近似**：造最小生成樹 → 沿樹走一圈（每邊走兩次）→ 跳過重複城市（shortcut）。因為 $MST \leq OPT$，繞兩圈 $\leq 2 \cdot OPT$。
- **Christofides 1.5-近似**：MST + 奇度數頂點間的最小完美匹配 → 歐拉迴路 → shortcut。近似比 1.5，懸了 45 年後才在 2020 年被改進了 10^{-36} 那麼一點——理論突破，實務不變。

常見誤解

一般（非度量）TSP **不存在**任何常數近似比演算法（除非 $P = NP$）。「你的距離有沒有三角不等式」決定了你有沒有安全網。

Chapter 8

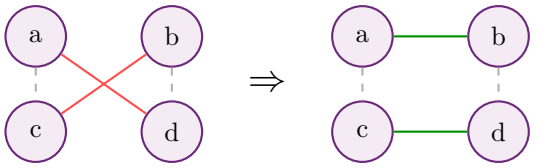
路 3：啟發式——沒有保證，但實務常勝

8.1 建構式啟發：最近鄰

TSP 的最近鄰法：每次走向「還沒去過的最近城市」。快（$O(n^2)$）、直覺，但品質平平（典型離最優 10–25%），常拿來當**起始解**。

8.2 局部搜尋：2-opt

拿到一條路線後，反覆嘗試「把路線中某一段**反轉**」——等價於把兩條交叉的邊解開——只要變短就採用，直到無法改善。



交叉：$ad + cb$　　解開後更短：$ab + cd$

我們的實測（12 城市隨機點）：最近鄰起手後 2-opt 收斂，離 Held–Karp 精確解仍有 4.5% 差距——這示範了局部搜尋會卡在 **局部最優**：附近沒有更好的解，但全域還有。

8.3 跳出局部最優：模擬退火與其他

- **模擬退火**（Simulated Annealing）：偶爾**接受變差的解**，接受機率隨「溫度」下降——早期亂跳探索、後期收斂精修。靈感來自金屬退火。
- **禁忌搜尋**（Tabu Search）：記住最近走過的解，禁止走回頭路。
- **遺傳演算法**（Genetic Algorithm）：一群解互相「交配 + 突變」，優勝劣汰。
- **蟻群演算法**（Ant Colony）：用「費洛蒙」累積好路徑的記憶。

這些統稱 **metaheuristics**：不保證品質、不保證收斂時間，但工程實務上經常是「效果最好的那個」。

8.4 工業級現實：solver 比你想的強大

- **TSP**：精確 solver **Concorde**（分支切割法）已解出 85,900 城市的最優解；啟發式 **LKH**（Lin–Kernighan–Helsgaun）對數萬城市常在**最優 0.02% 之內**，秒級完成——物流業的日常主力。

- **SAT**：CDCL solver（MiniSat、Kissat 等）處理百萬變數等級的硬體驗證實例；晶片業每天靠它檢查電路等價性。
- **通用建模**：ILP solver（Gurobi、CPLEX）、Google OR-Tools（CP-SAT）讓你「描述問題、不寫演算法」。

給工程師的忠告

遇到 NP-Hard 問題，2026 年的正確反應往往**不是自己寫搜尋**，而是把問題**建模成 SAT / ILP / CP**，交給身經百戰的 solver。你的競爭力在於建模的品質（變數怎麼取、對稱性怎麼消）。

Chapter 9

路 4：特例回到 P——先檢查你的問題長什麼樣

NP-Hard 是對**最壞情況、一般形式**的判決。你手上的實例常常有特殊結構，讓問題整個掉回 P。

9.1 2-SAT：一字之差，天壤之別

3-SAT 是 NPC，但每子句只有**兩個**文字的 2-SAT 可以線性時間解：

- 子句 $(a \vee b)$ 改寫成兩條**蘊含**：$\neg a \Rightarrow b$、$\neg b \Rightarrow a$。
- 把每個變數拆成「真、假」兩個節點，蘊含變成有向邊，建出**蘊含圖**。
- 跑 **SCC**（強連通分量，見圖論教材 Tarjan）：若某變數 x 與 $\neg x$ 落在同一分量 $\rightarrow$ 矛盾，UNSAT；否則依分量的拓撲序即可讀出一組解。

為什麼 3 個文字就不行？因為 $(a \vee b \vee c)$ 無法寫成單純的「一個假就逼另一個真」——蘊含結構崩掉了。**難與易的分界線常常細得驚人。**

9.2 其他常見的「掉回 P」情境

一般情況（難）	特例（easy）
Vertex Cover（NPC）	**二分圖**上 = 最大匹配（König 定理），多項式
Independent Set（NPC）	**樹**上用 DP $O(n)$；區間圖上貪婪即可
Graph Coloring（NPC, $k \geq 3$）	**區間圖**上貪婪著色最優（排課常是區間 !）
Hamiltonian（NPC）	改問**歐拉路徑**（每邊一次）就在 P
Subset Sum / Knapsack	**值域小**時偽多項式 DP 實用
TSP（NP-Hard）	**n 小**（≤ 20）Held–Karp；度量空間有近似保證
一般 SAT（NPC）	**2-SAT**、Horn-SAT（每子句至多一正文字）皆 P
覆蓋/排程若干 NPC	**參數 k 小**時 FPT：$O(2^k \cdot poly(n))$

實戰檢查清單

拿到疑似 NP-Hard 的問題先問四句：(1) 我的圖是樹／二分圖／區間圖嗎？(2) 數值範圍小嗎（偽多項式可行）？(3) 要找的東西很小嗎（k 小 $\rightarrow$ FPT）？(4) 規模就這麼點嗎（$n \leq 20 \rightarrow$ 位元壓縮 DP 直接上）？四個都「否」，才進近似／啟發式。

Chapter 10

真實世界如何與 NP 共處

10.1 密碼學：把「難」當成資產

NP-Hard 不全是壞消息——現代密碼學**依賴**「某些問題難解」：RSA 靠大整數分解難、離散對數系統靠指數難。如果 $P = NP$ 且演算法實用，這些防線一夜蒸發。「難」在這裡是保護你銀行帳戶的城牆。（註：分解質因數目前**未被證明** NP-Complete，它是「疑似不在 P、也疑似不是 NPC」的中間地帶，也是量子電腦 Shor 演算法能攻破的原因。）

10.2 晶片與軟體驗證：SAT solver 的主場

驗證兩個電路等價、驗證程式不會進入危險狀態，都被編碼成 SAT。CDCL solver 例行處理**數百萬變數**的實例；從 CPU 設計、作業系統驗證到數學定理的機器證明，SAT 被稱為電腦科學「安靜的革命」。

10.3 物流與路線：TSP/VRP 的日常

外送平台、貨運公司每天解的就是 TSP 與它的加強版 VRP（多車、時間窗、容量）。實務配方：**LKH 這類 heuristic 起手 + 局部搜尋精修**，數萬個點在幾秒內得到離最優 1% 內的路線；需要保證時用 Concorde 這類精確 solver 處理較小的實例。

10.4 排程與資源分配

- **排課／排班**：本質是圖著色 + 各種限制，實務用 CP-SAT / ILP solver 建模求解。
- **編譯器暫存器配置**：變數衝突圖著色，LLVM 等用啟發式（圖著色 + spill）。
- **雲端資源打包**：把 VM 塞進最少的實體機 = Bin Packing，資料中心的省電核心邏輯。
- **頻譜分配**：基地台頻率互擾 = 著色問題。

10.5 生物資訊與其他

蛋白質結構比對、基因組重組距離、系統發生樹重建，多數是 NP-Hard，靠啟發式與近似撐起整個領域。連遊戲設計都有份——數獨（一般化）、踩地雷（一般化）、俄羅斯方塊（離線版）都被證明 NP-Complete。

一句話總結

NP-Hard 的正確解讀不是「無解」，而是「**別追求萬用的完美解法**」。真實世界靠：特例結構、小參數、近似保證、強大 solver、以及「夠好就好」的工程智慧，跟 NP 和平共處了半個世紀。

Chapter 11

完整 C++ 實作

本章提供一份**完整、可直接編譯執行**的 C++17 程式碼，涵蓋全書 13 個演算法。此程式碼已實際以 `g++ -std=c++17 -O2 -Wall -Wextra` 編譯並執行驗證通過。

包含的演算法

精確解：N-Queens 回溯、DPLL SAT、Subset Sum(DP 與 meet-in-the-middle)、0/1 Knapsack DP、圖著色回溯、Hamiltonian Path 位元壓縮 DP、TSP Held–Karp、Vertex Cover 分支限界。
近似：Vertex Cover 2-近似、Set Cover 貪婪、Bin Packing FFD。
啟發式：TSP 最近鄰 + 2-opt。
特例回到 P：2-SAT（SCC）。

11.1 編譯與執行

```
g++ -std=c++17 -O2 -Wall -Wextra -o np_algorithms np_algorithms.cpp
./np_algorithms
```

程式碼 11.1: 編譯指令

11.2 原始碼

```
// np_algorithms.cpp
// ============================================================================
// NP-Complete / NP-Hard 問題的實戰解法示範（C++17）
//
// 涵蓋三大類技巧：
//   A. 精確解（指數時間，但小規模實用）
//      1. N-Queens            —— 回溯 + 剪枝
//      2. DPLL SAT Solver     —— 單元傳播 + 回溯（現代 CDCL 的雛形）
//      3. Subset Sum          —— 偽多項式 DP + Meet-in-the-Middle
//      4. 0/1 Knapsack        —— 偽多項式 DP
//      5. Graph Coloring      —— 回溯 + 剪枝
//      6. Hamiltonian Path    —— 位元壓縮 DP  O(n^2 * 2^n)
//      7. TSP (Held-Karp)     —— 位元壓縮 DP  O(n^2 * 2^n)
//      8. Vertex Cover        —— 分支限界（參數化 FPT 思想）
//   B. 近似演算法（多項式時間 + 品質保證）
//      9. Vertex Cover 2-近似（取極大匹配）
//     10. Set Cover 貪婪 ln(n)-近似
//     11. Bin Packing First-Fit Decreasing (<= 11/9 OPT + 1)
//     12. Metric TSP：最近鄰啟發 + 2-opt 局部搜尋
//   C. 特例回到 P（結構讓問題變簡單）
//     13. 2-SAT —— 用 SCC（Tarjan）在線性時間求解
//
```

```cpp
// Compile: g++ -std=c++17 -O2 -Wall -Wextra -o np_algorithms np_algorithms.cpp
// Run:     ./np_algorithms
// ============================================================================

#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>
#include <cmath>
#include <climits>
#include <random>
#include <functional>
#include <iomanip>

using namespace std;

// ============================================================================
// 1. N-Queens ── 回溯 + 剪枝（用三個布林陣列 O(1) 檢查衝突）
// ============================================================================
class NQueens {
    int n;
    long long solutions = 0;
    vector<bool> col, diag1, diag2;
    void solve(int row) {
        if (row == n) { ++solutions; return; }
        for (int c = 0; c < n; ++c) {
            if (col[c] || diag1[row + c] || diag2[row - c + n - 1])
                continue;
            col[c] = diag1[row + c] = diag2[row - c + n - 1] = true;
            solve(row + 1);                          // 只往合法分支走（剪枝）
            col[c] = diag1[row + c] = diag2[row - c + n - 1] = false;
        }
    }
public:
    long long count(int n_) {
        n = n_; solutions = 0;
        col.assign(n, false);
        diag1.assign(2 * n - 1, false);
        diag2.assign(2 * n - 1, false);
        solve(0);
        return solutions;
    }
};

// ============================================================================
// 2. DPLL SAT Solver ── CNF 可滿足性判定
//    文字編碼：變數 v 的正文字 = +v，負文字 = -v（v 由 1 起算）
// ============================================================================
class DPLL {
    int numVars;
    vector<vector<int>> clauses;

    // 對部分賦值 assign（0=未定, 1=真, -1=假）做單元傳播
    // 回傳：1 = 全部子句已滿足, -1 = 出現空子句（矛盾）, 0 = 未定
    int propagate(vector<int>& assign) {
        bool changed = true;
        while (changed) {
```

```
            changed = false;
            bool allSat = true;
            for (const auto& cl : clauses) {
                int unassigned = 0, lastLit = 0;
                bool sat = false;
                for (int lit : cl) {
                    int v = abs(lit), want = lit > 0 ? 1 : -1;
                    if (assign[v] == want) { sat = true; break; }
                    if (assign[v] == 0) { ++unassigned; lastLit = lit; }
                }
                if (sat) continue;
                allSat = false;
                if (unassigned == 0) return -1;          // 空子句 -> 矛盾
                if (unassigned == 1) {                   // 單元子句 -> 強制賦值
                    assign[abs(lastLit)] = lastLit > 0 ? 1 : -1;
                    changed = true;
                }
            }
            if (allSat) return 1;
        }
        return 0;
    }

    bool dfs(vector<int> assign) {
        int st = propagate(assign);
        if (st == 1) { model = assign; return true; }
        if (st == -1) return false;
        int v = 1;
        while (assign[v] != 0) ++v;                      // 挑第一個未賦值變數
        for (int val : {1, -1}) {                        // 先試真、再試假
            assign[v] = val;
            if (dfs(assign)) return true;
        }
        return false;
    }
public:
    vector<int> model;                                   // 求出的賦值
    bool solve(int nv, const vector<vector<int>>& cls) {
        numVars = nv; clauses = cls;
        return dfs(vector<int>(nv + 1, 0));
    }
};

// ============================================================================
// 3. Subset Sum
// ============================================================================
// 3a. 偽多項式 DP：O(n * target)
bool subsetSumDP(const vector<int>& a, int target) {
    vector<bool> dp(target + 1, false);
    dp[0] = true;
    for (int x : a)
        for (int s = target; s >= x; --s)
            if (dp[s - x]) dp[s] = true;
    return dp[target];
}

// 3b. Meet-in-the-Middle：O(2^(n/2) * n)，值域大時的利器
```

```cpp
bool subsetSumMITM(const vector<int>& a, long long target) {
    int n = (int)a.size(), h = n / 2;
    auto enumerate = [](const vector<int>& part) {
        vector<long long> sums;
        int m = (int)part.size();
        for (int mask = 0; mask < (1 << m); ++mask) {
            long long s = 0;
            for (int i = 0; i < m; ++i)
                if (mask & (1 << i)) s += part[i];
            sums.push_back(s);
        }
        return sums;
    };
    vector<int> L(a.begin(), a.begin() + h), R(a.begin() + h, a.end());
    vector<long long> sl = enumerate(L), sr = enumerate(R);
    sort(sr.begin(), sr.end());
    for (long long s : sl)
        if (binary_search(sr.begin(), sr.end(), target - s)) return true;
    return false;
}

// ============================================================================
// 4. 0/1 Knapsack —— 偽多項式 DP：O(n * W)
// ============================================================================
int knapsack01(const vector<int>& w, const vector<int>& v, int W) {
    vector<int> dp(W + 1, 0);
    for (size_t i = 0; i < w.size(); ++i)
        for (int cap = W; cap >= w[i]; --cap)
            dp[cap] = max(dp[cap], dp[cap - w[i]] + v[i]);
    return dp[W];
}

// ============================================================================
// 5. Graph Coloring —— 回溯：能否用 k 色著色
// ============================================================================
class GraphColoring {
    const vector<vector<int>>& adj;
    int n, k;
    vector<int> color;
    bool ok(int u, int c) {
        for (int v : adj[u]) if (color[v] == c) return false;
        return true;
    }
    bool dfs(int u) {
        if (u == n) return true;
        for (int c = 0; c < k; ++c) {
            if (!ok(u, c)) continue;                    // 剪枝：衝突不往下走
            color[u] = c;
            if (dfs(u + 1)) return true;
            color[u] = -1;
        }
        return false;
    }
public:
    GraphColoring(const vector<vector<int>>& g) : adj(g), n((int)g.size()) {}
    bool colorable(int k_) {
        k = k_; color.assign(n, -1);
```

```cpp
        return dfs(0);
    }
};

// ============================================================================
// 6. Hamiltonian Path —— 位元壓縮 DP：O(n^2 * 2^n)
//    dp[mask][v] = 走過集合 mask、目前停在 v，是否可行
// ============================================================================
bool hamiltonianPath(const vector<vector<int>>& adj) {
    int n = (int)adj.size();
    if (n == 0) return false;
    if (n == 1) return true;
    vector<vector<char>> dp(1 << n, vector<char>(n, 0));
    for (int v = 0; v < n; ++v) dp[1 << v][v] = 1;       // 任意起點
    for (int mask = 1; mask < (1 << n); ++mask)
        for (int v = 0; v < n; ++v) {
            if (!dp[mask][v]) continue;
            for (int u : adj[v])
                if (!(mask & (1 << u)))
                    dp[mask | (1 << u)][u] = 1;
        }
    for (int v = 0; v < n; ++v)
        if (dp[(1 << n) - 1][v]) return true;
    return false;
}

// ============================================================================
// 7. TSP —— Held-Karp 位元壓縮 DP：O(n^2 * 2^n)，n<=20 實用
// ============================================================================
long long tspHeldKarp(const vector<vector<int>>& dist) {
    int n = (int)dist.size();
    const long long INF = LLONG_MAX / 4;
    vector<vector<long long>> dp(1 << n, vector<long long>(n, INF));
    dp[1][0] = 0;                                        // 從城市 0 出發
    for (int mask = 1; mask < (1 << n); ++mask) {
        if (!(mask & 1)) continue;                       // 必含起點
        for (int v = 0; v < n; ++v) {
            if (dp[mask][v] == INF) continue;
            for (int u = 0; u < n; ++u)
                if (!(mask & (1 << u)))
                    dp[mask | (1 << u)][u] =
                        min(dp[mask | (1 << u)][u], dp[mask][v] +
                            dist[v][u]);
        }
    }
    long long best = INF;
    for (int v = 1; v < n; ++v)
        if (dp[(1 << n) - 1][v] != INF)
            best = min(best, dp[(1 << n) - 1][v] + dist[v][0]);  //
                回到起點
    return best;
}

// ============================================================================
// 8. Vertex Cover —— 分支限界（FPT 思想）：cover 大小 <= k 是否可行？
//    關鍵觀察：任一條邊 (u,v)，u 或 v 至少要選一個 -> 二分支
// ============================================================================
bool vertexCoverK(vector<vector<int>> adj, int k) {
    // 找任一條還存在的邊
```

```
    int u = -1, v = -1;
    for (int i = 0; i < (int)adj.size() && u == -1; ++i)
        if (!adj[i].empty()) { u = i; v = adj[i][0]; }
    if (u == -1) return true;                                  // 沒邊了 -> 已覆蓋
    if (k == 0) return false;                                  // 還有邊但配額用完

    auto removeVertex = [](vector<vector<int>> g, int x) {
        for (int y : g[x]) {
            auto& lst = g[y];
            lst.erase(remove(lst.begin(), lst.end(), x), lst.end());
        }
        g[x].clear();
        return g;
    };
    return vertexCoverK(removeVertex(adj, u), k - 1)           // 選 u
        || vertexCoverK(removeVertex(adj, v), k - 1);          // 選 v
}

// ============================================================================
// 9. Vertex Cover 2-近似 —— 取「極大匹配」的兩端點
//    保證：|解| <= 2 * OPT（每條匹配邊至少要 1 個點，我們取了 2 個）
// ============================================================================
vector<int> vertexCover2Approx(int n, vector<pair<int,int>> edges) {
    vector<bool> inCover(n, false);
    vector<int> cover;
    for (auto [u, v] : edges) {
        if (inCover[u] || inCover[v]) continue;                // 邊已被覆蓋
        inCover[u] = inCover[v] = true;                        // 兩端點都收
        cover.push_back(u); cover.push_back(v);
    }
    return cover;
}

// ============================================================================
// 10. Set Cover 貪婪近似 —— 每輪挑「新覆蓋元素最多」的集合
//     保證：|解| <= ln(n) * OPT + 1（幾乎是能做到的最佳近似比）
// ============================================================================
vector<int> setCoverGreedy(int universe, const vector<vector<int>>& sets) {
    vector<bool> covered(universe, false);
    int remaining = universe;
    vector<int> chosen;
    while (remaining > 0) {
        int best = -1, bestGain = 0;
        for (int i = 0; i < (int)sets.size(); ++i) {
            int gain = 0;
            for (int x : sets[i]) if (!covered[x]) ++gain;
            if (gain > bestGain) { bestGain = gain; best = i; }
        }
        if (best == -1) break;                                 // 無法覆蓋
        chosen.push_back(best);
        for (int x : sets[best])
            if (!covered[x]) { covered[x] = true; --remaining; }
    }
    return chosen;
}

```

```cpp
// ============================================================================
// 11. Bin Packing —— First-Fit Decreasing
//     保證：使用箱數 <= 11/9 * OPT + 1
// ============================================================================
int binPackingFFD(vector<double> items, double capacity) {
    sort(items.rbegin(), items.rend());                  // 由大到小
    vector<double> bins;                                 // 每箱剩餘空間
    for (double it : items) {
        bool placed = false;
        for (double& room : bins)
            if (room >= it - 1e-12) { room -= it; placed = true; break; }
        if (!placed) bins.push_back(capacity - it);      // 開新箱
    }
    return (int)bins.size();
}

// ============================================================================
// 12. Metric TSP 啟發式 —— 最近鄰建構 + 2-opt 改善
// ============================================================================
static double tourLength(const vector<int>& tour,
                         const vector<vector<double>>& d) {
    double len = 0;
    int n = (int)tour.size();
    for (int i = 0; i < n; ++i) len += d[tour[i]][tour[(i + 1) % n]];
    return len;
}

vector<int> tspNearestNeighbor(const vector<vector<double>>& d) {
    int n = (int)d.size();
    vector<bool> used(n, false);
    vector<int> tour = {0};
    used[0] = true;
    for (int step = 1; step < n; ++step) {
        int cur = tour.back(), best = -1;
        for (int v = 0; v < n; ++v)
            if (!used[v] && (best == -1 || d[cur][v] < d[cur][best]))
                best = v;
        used[best] = true;
        tour.push_back(best);
    }
    return tour;
}

// 2-opt：反覆嘗試「反轉一段路徑」，只要變短就採用
void tsp2opt(vector<int>& tour, const vector<vector<double>>& d) {
    int n = (int)tour.size();
    bool improved = true;
    while (improved) {
        improved = false;
        for (int i = 0; i < n - 1; ++i)
            for (int j = i + 2; j < n; ++j) {
                if (i == 0 && j == n - 1) continue;      // 同一條邊
                int a = tour[i], b = tour[i + 1];
                int c = tour[j], e = tour[(j + 1) % n];
                double delta = d[a][c] + d[b][e] - d[a][b] - d[c][e];
                if (delta < -1e-10) {
                    reverse(tour.begin() + i + 1, tour.begin() + j + 1);
                    improved = true;
                }
```

```
            }
        }
    }

// ============================================================================
// 13. 2-SAT ——— 特例回到 P：蘊含圖 + Tarjan SCC，O(n + m)
//     子句 (a  b) 等價於 ¬a→b 與 ¬b→a
// ============================================================================
class TwoSAT {
    int n;                                  // 變數個數
    vector<vector<int>> g;                  // 蘊含圖，節點 2v=正、2v+1=負
    vector<int> comp, order;
    vector<bool> visited;

    void dfs1(int u) {
        visited[u] = true;
        for (int v : g[u]) if (!visited[v]) dfs1(v);
        order.push_back(u);
    }
    void dfs2(int u, int c, const vector<vector<int>>& gr) {
        comp[u] = c;
        for (int v : gr[u]) if (comp[v] == -1) dfs2(v, c, gr);
    }
public:
    vector<bool> value;                     // 求出的賦值
    TwoSAT(int vars) : n(vars), g(2 * vars) {}

    // 加入子句 (x_i = vi)  (x_j = vj)，vi/vj 為 true 表示正文字
    void addClause(int i, bool vi, int j, bool vj) {
        int a = 2 * i + (vi ? 0 : 1);       // 文字 a
        int b = 2 * j + (vj ? 0 : 1);       // 文字 b
        g[a ^ 1].push_back(b);              // ¬a -> b
        g[b ^ 1].push_back(a);              // ¬b -> a
    }

    bool solve() {                          // Kosaraju 兩遍 DFS 求 SCC
        int N = 2 * n;
        visited.assign(N, false);
        order.clear();
        for (int i = 0; i < N; ++i) if (!visited[i]) dfs1(i);
        vector<vector<int>> gr(N);
        for (int u = 0; u < N; ++u) for (int v : g[u]) gr[v].push_back(u);
        comp.assign(N, -1);
        int c = 0;
        for (int i = N - 1; i >= 0; --i)
            if (comp[order[i]] == -1) dfs2(order[i], c++, gr);
        value.assign(n, false);
        for (int v = 0; v < n; ++v) {
            if (comp[2 * v] == comp[2 * v + 1]) return false; // x 與 ¬x 同分量
            value[v] = comp[2 * v] > comp[2 * v + 1];
        }
        return true;
    }
};

// ============================================================================
// Demonstrations
// ============================================================================
```

```cpp
static void hr(const string& title) {
    cout << "\n" << string(62, '-') << "\n" << title << "\n";
}

int main() {
    cout << fixed << setprecision(3);

    // ---- 1. N-Queens ----
    hr("1. N-Queens (backtracking + pruning)");
    NQueens nq;
    for (int n : {4, 8, 10})
        cout << "  n=" << n << " -> " << nq.count(n) << " solutions\n";

    // ---- 2. DPLL SAT ----
    hr("2. DPLL SAT solver");
    {
        // (x1 v x2) & (~x1 v x3) & (~x2 v ~x3) & (x1 v x3)
        DPLL s;
        bool sat = s.solve(3, {{1,2},{-1,3},{-2,-3},{1,3}});
        cout << "  formula A: " << (sat ? "SAT" : "UNSAT");
        if (sat) {
            cout << "  model:";
            for (int v = 1; v <= 3; ++v)
                cout << " x" << v << "=" << (s.model[v] == 1);
        }
        cout << "\n";
        // 不可滿足：(x1) & (~x1)
        DPLL s2;
        cout << "  formula B (x & ~x): "
             << (s2.solve(1, {{1},{-1}}) ? "SAT" : "UNSAT") << "\n";
    }

    // ---- 3. Subset Sum ----
    hr("3. Subset Sum (DP + meet-in-the-middle)");
    {
        vector<int> a = {3, 34, 4, 12, 5, 2};
        cout << "  {3,34,4,12,5,2} sum=9  : DP=" << subsetSumDP(a, 9)
             << " MITM=" << subsetSumMITM(a, 9) << "\n";
        cout << "  {3,34,4,12,5,2} sum=30 : DP=" << subsetSumDP(a, 30)
             << " MITM=" << subsetSumMITM(a, 30) << "\n";
    }

    // ---- 4. Knapsack ----
    hr("4. 0/1 Knapsack (pseudo-polynomial DP)");
    {
        vector<int> w = {10, 20, 30}, v = {60, 100, 120};
        cout << "  W=50, weights{10,20,30}, values{60,100,120} -> best
            value = "
             << knapsack01(w, v, 50) << " (expect 220)\n";
    }

    // ---- 5. Graph Coloring ----
    hr("5. Graph Coloring (backtracking)");
    {
        // K4（完全圖）需要 4 色；C5（奇環）需要 3 色
        vector<vector<int>> k4 = {{1,2,3},{0,2,3},{0,1,3},{0,1,2}};
        vector<vector<int>> c5 = {{1,4},{0,2},{1,3},{2,4},{3,0}};
        GraphColoring g1(k4), g2(c5);
        cout << "  K4 with 3 colors: " << (g1.colorable(3) ? "yes" : "no")
```

```
480             << " | with 4: " << (g1.colorable(4) ? "yes" : "no") << "\n";
481         cout << "  C5 with 2 colors: " << (g2.colorable(2) ? "yes" : "no")
482             << " | with 3: " << (g2.colorable(3) ? "yes" : "no") << "\n";
483     }
484
485     // ---- 6. Hamiltonian Path ----
486     hr("6. Hamiltonian Path (bitmask DP)");
487     {
488         // path 圖 0-1-2-3 有；星狀圖（中心 0）沒有
489         vector<vector<int>> path = {{1},{0,2},{1,3},{2}};
490         vector<vector<int>> star = {{1,2,3},{0},{0},{0}};
491         cout << "  path graph: " << (hamiltonianPath(path) ? "yes" : "no")
492             << " | star graph: " << (hamiltonianPath(star) ? "yes" :
                "no") << "\n";
493     }
494
495     // ---- 7. TSP Held-Karp ----
496     hr("7. TSP exact (Held-Karp bitmask DP)");
497     {
498         vector<vector<int>> d = {
499             {0, 10, 15, 20},
500             {10, 0, 35, 25},
501             {15, 35, 0, 30},
502             {20, 25, 30, 0}};
503         cout << "  4-city classic -> optimal tour = " << tspHeldKarp(d)
504             << " (expect 80)\n";
505     }
506
507     // ---- 8. Vertex Cover exact ----
508     hr("8. Vertex Cover k-branching (FPT style)");
509     {
510         // 三角形 + 一條尾巴：0-1,1-2,2-0,2-3
511         vector<vector<int>> g = {{1,2},{0,2},{0,1,3},{2}};
512         cout << "  triangle+tail: k=1 " << (vertexCoverK(g,1) ? "yes" :
                "no")
513             << " | k=2 " << (vertexCoverK(g,2) ? "yes" : "no") << "\n";
514     }
515
516     // ---- 9. Vertex Cover 2-approx ----
517     hr("9. Vertex Cover 2-approximation (maximal matching)");
518     {
519         vector<pair<int,int>> edges = {{0,1},{0,2},{1,2},{2,3},{3,4}};
520         auto cover = vertexCover2Approx(5, edges);
521         cout << "  cover = {";
522         for (size_t i = 0; i < cover.size(); ++i)
523             cout << cover[i] << (i + 1 < cover.size() ? "," : "");
524         cout << "} size=" << cover.size() << " (OPT=2: {2,?} -> bound
                2*OPT=4)\n";
525     }
526
527     // ---- 10. Set Cover greedy ----
528     hr("10. Set Cover greedy approximation");
529     {
530         // 宇宙 {0..4}；S0={0,1,2}, S1={1,3}, S2={2,3}, S3={3,4}
531         vector<vector<int>> sets = {{0,1,2},{1,3},{2,3},{3,4}};
532         auto chosen = setCoverGreedy(5, sets);
533         cout << "  chosen sets:";
534         for (int i : chosen) cout << " S" << i;
535         cout << "  (count=" << chosen.size() << ")\n";
```

```
536     }
537
538     // ---- 11. Bin Packing FFD ----
539     hr("11. Bin Packing First-Fit Decreasing");
540     {
541         vector<double> items = {0.42, 0.25, 0.27, 0.07, 0.72, 0.86, 0.09,
                0.44, 0.50, 0.68};
542         cout << "  10 items, capacity 1.0 -> bins used = "
543              << binPackingFFD(items, 1.0)
544              << " (lower bound = ceil(sum) = "
545              << (int)ceil(accumulate(items.begin(), items.end(), 0.0)) <<
                    ")\n";
546     }
547
548     // ---- 12. TSP heuristic: NN + 2-opt vs exact ----
549     hr("12. Metric TSP: nearest-neighbor + 2-opt vs Held-Karp");
550     {
551         mt19937 rng(2026);
552         int n = 12;
553         vector<double> xs(n), ys(n);
554         for (int i = 0; i < n; ++i) {
555             xs[i] = (double)(rng() % 1000);
556             ys[i] = (double)(rng() % 1000);
557         }
558         vector<vector<double>> d(n, vector<double>(n, 0));
559         vector<vector<int>> di(n, vector<int>(n, 0));
560         for (int i = 0; i < n; ++i)
561             for (int j = 0; j < n; ++j) {
562                 d[i][j] = hypot(xs[i] - xs[j], ys[i] - ys[j]);
563                 di[i][j] = (int)llround(d[i][j] * 1000);   // 整數版給
                        Held-Karp
564             }
565         auto tour = tspNearestNeighbor(d);
566         double nnLen = tourLength(tour, d);
567         tsp2opt(tour, d);
568         double optLen2 = tourLength(tour, d);
569         double exact = (double)tspHeldKarp(di) / 1000.0;
570         cout << "  nearest neighbor : " << nnLen  << "\n";
571         cout << "  after 2-opt      : " << optLen2 << "\n";
572         cout << "  exact (Held-Karp): " << exact << "\n";
573         cout << "  2-opt gap        : "
574              << 100.0 * (optLen2 - exact) / exact << "%\n";
575     }
576
577     // ---- 13. 2-SAT ----
578     hr("13. 2-SAT via SCC (a P-time special case)");
579     {
580         // (x0 v x1) & (~x0 v x1) & (~x1 v x2)  ——  可滿足
581         TwoSAT ts(3);
582         ts.addClause(0, true, 1, true);
583         ts.addClause(0, false, 1, true);
584         ts.addClause(1, false, 2, true);
585         bool ok = ts.solve();
586         cout << "  formula C: " << (ok ? "SAT" : "UNSAT");
587         if (ok) {
588             cout << "  model:";
589             for (int v = 0; v < 3; ++v) cout << " x" << v << "=" <<
                    ts.value[v];
590         }
```

```
        cout << "\n";
        // (x0 ∨ x0) ∧ (¬x0 ∨ ¬x0) —— 矛盾
        TwoSAT ts2(1);
        ts2.addClause(0, true, 0, true);
        ts2.addClause(0, false, 0, false);
        cout << "  formula D (x & ~x): " << (ts2.solve() ? "SAT" :
            "UNSAT") << "\n";
    }

    cout << "\nAll demos finished.\n";
    return 0;
}
```

11.3 執行輸出（節錄）

```
1. N-Queens: n=4 -> 2 | n=8 -> 92 | n=10 -> 724 solutions
2. DPLL SAT: formula A: SAT  model: x1=1 x2=0 x3=1
             formula B (x & ~x): UNSAT
3. Subset Sum {3,34,4,12,5,2}: sum=9 -> yes | sum=30 -> no
4. Knapsack W=50 -> best value = 220 (expect 220)
5. Coloring: K4 3-color no / 4-color yes | C5 2-color no / 3-color yes
6. Hamiltonian: path graph yes | star graph no
7. TSP Held-Karp 4-city -> optimal = 80 (expect 80)
8. Vertex Cover branching: k=1 no | k=2 yes
9. VC 2-approx: size=4 (OPT=2, bound 2*OPT=4)
10. Set Cover greedy: S0 S3 (count=2)
11. Bin Packing FFD: bins used = 5 (= lower bound)
12. TSP 12 cities: NN 3574.4 -> 2-opt 3574.4 vs exact 3420.5 (gap 4.5%)
13. 2-SAT: formula C SAT (x0=1 x1=1 x2=1) | formula D UNSAT
```

程式碼 11.2: 程式實際輸出

Chapter 12

練習題

12.1 觀念題

1. 用自己的話解釋：為什麼「NP」不是「非多項式」的意思？
2. 你朋友宣稱：「我證明了 Sudoku（一般化 $n^2 \times n^2$）有多項式解法。」如果他是對的，對整個 NP 世界意味著什麼？
3. 為什麼「TSP 優化版」通常說是 NP-Hard 而不說 NP-Complete？
4. Knapsack 有 $O(nW)$ 的 DP，為什麼它仍是 NP-Complete？「偽多項式」的「偽」在哪裡？
5. 證明：S 是獨立集 $\iff$ $V \setminus S$ 是頂點覆蓋。並由此說明兩個問題的難度為何綁在一起。

12.2 實作題

6. 修改 N-Queens 程式，輸出**一組**具體解（而不只是計數）。
7. 把 Subset Sum 的 DP 改成同時輸出「選了哪些數」。
8. 為 DPLL 加上「純文字消去」（pure literal elimination）：若某變數在所有子句中只以同一極性出現，直接賦值。
9. 用 Held–Karp 的 dp 表回溯出實際的最短巡迴路線。
10. 實作模擬退火版 TSP，跟 2-opt 比較誰更接近 Held–Karp 精確解。
11. 實作「度量 TSP 的 MST 2-近似」（提示：用圖論教材的 Prim + DFS 走訪 + shortcut），實驗它實際離最優多遠。
12. 把「排課問題」（n 門課、衝突表、k 個時段）建模成圖著色，用回溯求解；再試著加上「先著色度數最大的點」的排序啟發，觀察加速。

12.3 挑戰題

13. 證明 Partition $\leq_p$ Bin Packing 決策版（提示：兩個容量恰好的箱子）。
14. 寫一個 3-SAT 隨機實例產生器，觀察「子句數/變數數」比值在 4.27 附近時 DPLL 突然變慢的**相變**現象。
15. 研究 Vertex Cover 的核心化（kernelization）：先套用「度數 $> k$ 的點必選」規則，再分支，能快多少？

Chapter 13

總結與延伸閱讀

13.1 全書速查表

技巧	複雜度	適用時機
回溯 + 剪枝	指數（剪枝後大減）	小規模、約束強
DPLL / CDCL	最壞指數	SAT 建模後丟 solver
偽多項式 DP	$O(nW)$	數值範圍小
Meet-in-the-middle	$O(2^{n/2})$	$n \le 40$ 的子集問題
位元壓縮 DP	$O(n^2 2^n)$	$n \le 20$（TSP、Hamiltonian）
分支限界 / FPT	$O(2^k poly(n))$	目標參數 k 小
VC 2-近似	$O(E)$	要保證、要快
Set Cover 貪婪	$O(\sum \lvert S_i \rvert)$	$\ln n$ 保證已是極限
Bin Packing FFD	$O(n \log n)$	11/9 OPT + 1
MST 2-近似 TSP	$O(E \log V)$	度量空間
2-opt / 退火	視收斂	實務品質好、無保證
2-SAT（SCC）	$O(n+m)$	每子句兩文字

13.2 學習路徑建議

1. **先會分類**：拿到問題能判斷「這是不是經典 NPC 的偽裝」。（多看第 4 章圖鑑，面試中「認出題型」值一半分數。）
2. **掌握三個精確技巧**：回溯剪枝、偽多項式 DP、位元壓縮 DP——競賽與面試的主力。
3. **記住三個近似經典**：VC 的 2、Set Cover 的 $\ln n$、Metric TSP 的 1.5——連證明一起記，很短。
4. **學會用 solver**：把問題編成 SAT / ILP 是現代工程師的核心能力。
5. **深入理論（選修）**：歸約證明、PCP 定理與不可近似性、參數化複雜度、平均情況複雜度。

13.3 延伸閱讀

- *Computers and Intractability*（Garey & Johnson）——NP-Complete 的聖經，附 300+ 問題圖鑑。
- *Introduction to Algorithms*（CLRS）第 34–35 章——NP-Completeness 與近似演算法的標準教材。
- *Algorithm Design*（Kleinberg & Tardos）第 8、10–12 章——歸約與近似講得最好懂的一本。

- *Approximation Algorithms*（Vazirani）——近似演算法專書。
- *Parameterized Algorithms*（Cygan et al.）——FPT 專書。
- *In Pursuit of the Traveling Salesman*（William Cook）——TSP 的科普史，Concorde 作者親筆。
- The Silent (R)evolution of SAT（CACM）——現代 SAT solver 綜述。
- Clay Mathematics Institute：P vs NP 千禧年大獎題官方頁面。

結語

NP 理論最迷人的地方，是它把「這題好難」從抱怨變成**定理**，又把「難」變成可以分類、可以交易、可以繞路的工程對象。懂得辨識難度、選對武器（精確、近似、啟發、特例），你就能在理論的懸崖邊，把真實世界的問題一個個解掉。