NP 難題演算法系列 · 第 02 篇

0/1 背包問題

0/1 Knapsack — Dynamic Programming & Branch-and-Bound

容量有限、每件物品拿或不拿：NP-Hard 卻有偽多項式解法的代表作

Contents

1 問題是什麼

白話比喻

你是登山客，背包限重 W 公斤。眼前 n 件裝備各有重量與價值：帳篷重但保命、巧克力輕又高熱量……每件**只能整件拿或整件不拿**（所以叫 0/1），怎麼裝總價值最高？

定義 1.1 (0/1 Knapsack)**.** 給定 n 件物品，第 i 件重 w_i、值 v_i，背包容量 W。求 $S \subseteq \{1,\dots,n\}$ 最大化 $\sum_{i\in S} v_i$，使 $\sum_{i\in S} w_i \le W$。

1.1 難度定位

- 判定版本（「價值能否 $\geq V$?」）是 **NP-Complete**（可由 Subset Sum 歸約）。
- 但它有 $O(nW)$ 的 DP——與輸入的**數值** W 成正比而非與輸入**長度**（$\log W$ 位元）成正比，所以叫**偽多項式（pseudo-polynomial）**。W 是一億時照樣跑不動。
- 屬於**弱 NP-Hard**：數值小就簡單，數值大才難。這與 TSP 那種**強 NP-Hard**（數值再小也難）形成對比。

常見誤解

貪心「先拿價值密度 v_i/w_i 最高的」對 0/1 背包**不對**！反例：$W = 50$，物品 $(w, v) = (10, 60), (20, 100), (30, 120)$。密度排序拿 $(10, 60), (20, 100)$ 得 160，但最優是 $(20, 100) + (30, 120) = 220$。密度貪心只對「可切割」的分數背包正確。

2 演算法一：動態規劃 $O(nW)$

2.1 狀態設計

$$dp[i][c] = \text{只考慮前} i \text{ 件、容量} c \text{ 時的最大價值}$$

第 i 件只有兩種命運：

$$dp[i][c] = \max\big(\underbrace{dp[i-1][c]}_{\text{不拿}},\ \underbrace{dp[i-1][c-w_i] + v_i}_{\text{拿（需} c \geq w_i\text{）}}\big)$$

答案是 $dp[n][W]$。時間 $O(nW)$、空間 $O(nW)$（保留全表才能還原方案）。

2.2 手算範例

物品 (w, v)：A$(2,3)$、B$(3,4)$、C$(4,5)$、D$(5,8)$，$W = 10$。表格橫軸容量、縱軸物品（只列關鍵欄）：

c	2	3	4	5	7	9	10
A$(2,3)$	3	3	3	3	3	3	3
+B$(3,4)$	3	4	4	7	7	7	7
+C$(4,5)$	3	4	5	7	8	12	12
+D$(5,8)$	3	4	5	8	11	12	**15**

$dp[4][10] = 15$：反推方案時逐列比對「值有沒有變」。$dp[4][10] = 15 \neq dp[3][10] = 12$，表示 D 有拿（剩容量 $10-5=5$）；$dp[3][5] = 7 = dp[2][5]$，值沒變表示 C 沒拿；$dp[2][5] = 7 \neq dp[1][5] = 3$，表示 B 有拿（剩容量 $5-3=2$）；$dp[1][2] = 3 \neq 0$，表示 A 有拿。方案 $=$ {A,B,D}，重 $2+3+5=10$、值 $3+4+8=15$。

2.3 一維空間優化

觀察轉移只用到上一列，可以壓成一維——但容量必須**由大到小**掃，否則同一件物品會被拿兩次（那是無限背包的寫法）：

```
for (const auto& it : items)
    for (int c = W; c >= it.w; --c)       // 倒序！
        dp[c] = max(dp[c], dp[c - it.w] + it.v);
```

3 演算法二：分支限界（Branch and Bound）

當 W 太大 DP 開不了表時，回到搜尋，但帶上「樂觀估計」剪枝。

3.1 上界函數

在節點（已決定前 k 件）估計「剩下的物品最多還能貢獻多少」：把剩餘物品依價值密度遞減排序，**允許切割**地裝滿剩餘容量（分數背包鬆弛）。分數背包的最優值 $\geq 0/1$ 的最優值，所以這是合法上界：

$$\text{if } \underbrace{\text{目前價值}}_{val} + \underbrace{\text{分數鬆弛上界}}_{bound} \leq \text{已知最佳} \Rightarrow \text{整枝剪掉}$$

3.2 效果

本篇程式 Example 3（$n = 30$、$W = 500$）：DP 檢查 $30 \times 500 = 15000$ 格；分支限界只拜訪 **59 個節點**。物品先按密度排序讓好解早出現、上界更快收緊，是剪枝威力的關鍵。

4 完整 C++ 程式

包含：二維 DP + 方案還原、一維優化 DP、分數鬆弛分支限界、偽多項式行為實測。編譯：

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

```cpp
// knapsack.cpp
// ------------------------------------------------------------------
// 0/1 Knapsack：
//   1. 二維 DP O(nW) + 回溯還原選了哪些物品
//   2. 一維空間優化 DP O(W)
//   3. 分支限界（回溯 + 分數背包上界剪枝）
//
// Compile: g++ -std=c++17 -O2 -Wall -Wextra -o knapsack knapsack.cpp
// ------------------------------------------------------------------
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>
#include <random>
#include <chrono>

using namespace std;

struct Item { int w, v; };

// ---------- 1. 二維 DP + 還原方案 ----------
pair<int, vector<int>> knapsackDP2D(const vector<Item>& items, int W) {
    int n = (int)items.size();
    // dp[i][c] = 只考慮前 i 件、容量 c 的最大價值
    vector<vector<int>> dp(n + 1, vector<int>(W + 1, 0));
    for (int i = 1; i <= n; ++i)
        for (int c = 0; c <= W; ++c) {
            dp[i][c] = dp[i - 1][c];                    // 不拿第 i 件
            if (c >= items[i - 1].w)                    // 拿第 i 件
                dp[i][c] = max(dp[i][c],
```

```
                                dp[i - 1][c - items[i - 1].w] + items[i - 1].v);
        }
    // 回溯：從 dp[n][W] 反推每件拿或不拿
    vector<int> chosen;
    int c = W;
    for (int i = n; i >= 1; --i)
        if (dp[i][c] != dp[i - 1][c]) {                    // 值變了 -> 有拿
            chosen.push_back(i - 1);
            c -= items[i - 1].w;
        }
    reverse(chosen.begin(), chosen.end());
    return {dp[n][W], chosen};
}

// ---------- 2. 一維空間優化（容量倒著掃） ----------
int knapsackDP1D(const vector<Item>& items, int W) {
    vector<int> dp(W + 1, 0);
    for (const auto& it : items)
        for (int c = W; c >= it.w; --c)                    // 倒序避免重複拿
            dp[c] = max(dp[c], dp[c - it.w] + it.v);
    return dp[W];
}

// ---------- 3. 分支限界 ----------
// 上界：剩餘容量用「分數背包」鬆弛（物品可切割）——一定 >= 真實最佳
class BranchAndBound {
    vector<Item> items;                 // 依價值密度排序後
    int W, best = 0;
public:
    long long nodes = 0;

    int solve(vector<Item> its, int W_) {
        items = std::move(its);
        W = W_;
        sort(items.begin(), items.end(), [](const Item& a, const Item& b) {
            return (long long)a.v * b.w > (long long)b.v * a.w;  // 密度遞減
        });
        best = 0; nodes = 0;
        dfs(0, 0, 0);
        return best;
    }
private:
    double fractionalBound(int idx, int cap) {
        double bound = 0;
        for (int i = idx; i < (int)items.size() && cap > 0; ++i) {
            if (items[i].w <= cap) { bound += items[i].v; cap -= items[i].w; }
            else { bound += (double)items[i].v * cap / items[i].w; cap = 0; }
        }
        return bound;
    }
    void dfs(int idx, int usedW, int val) {
        ++nodes;
```

```
        best = max(best, val);
        if (idx == (int)items.size()) return;
        // 剪枝：目前價值 + 樂觀上界 <= 已知最佳 -> 整枝砍掉
        if (val + fractionalBound(idx, W - usedW) <= best) return;
        if (usedW + items[idx].w <= W)                          // 拿
            dfs(idx + 1, usedW + items[idx].w, val + items[idx].v);
        dfs(idx + 1, usedW, val);                               // 不拿
    }
};

int main() {
    // ---- 範例 1：經典小例 ----
    cout << "=== Example 1: classic 3-item case ===\n";
    vector<Item> a = {{10, 60}, {20, 100}, {30, 120}};
    auto [bestA, pickA] = knapsackDP2D(a, 50);
    cout << "W=50, items (w,v) = (10,60)(20,100)(30,120)\n";
    cout << "best value = " << bestA << " (expect 220)\n";
    cout << "chosen items:";
    for (int i : pickA)
        cout << " #" << i << "(w=" << a[i].w << ",v=" << a[i].v << ")";
    cout << "\n1D DP check = " << knapsackDP1D(a, 50) << "\n\n";

    // ---- 範例 2：手算教學例（6 件） ----
    cout << "=== Example 2: 6-item worked example ===\n";
    vector<Item> b = {{2,3},{3,4},{4,5},{5,8},{9,10},{1,1}};
    auto [bestB, pickB] = knapsackDP2D(b, 10);
    cout << "W=10 -> best = " << bestB << ", chosen:";
    for (int i : pickB) cout << " #" << i;
    int wSum = 0, vSum = 0;
    for (int i : pickB) { wSum += b[i].w; vSum += b[i].v; }
    cout << "  (total w=" << wSum << ", v=" << vSum << ")\n\n";

    // ---- 範例 3：DP vs 分支限界，隨機大例 ----
    cout << "=== Example 3: DP vs branch-and-bound (n=30, W=500) ===\n";
    mt19937 rng(42);
    uniform_int_distribution<int> wd(5, 60), vd(10, 100);
    vector<Item> c;
    for (int i = 0; i < 30; ++i) c.push_back({wd(rng), vd(rng)});
    int W = 500;

    auto t0 = chrono::steady_clock::now();
    int dpBest = knapsackDP1D(c, W);
    auto t1 = chrono::steady_clock::now();
    BranchAndBound bb;
    int bbBest = bb.solve(c, W);
    auto t2 = chrono::steady_clock::now();
    cout << "DP  : best=" << dpBest << "  ("
         << chrono::duration<double, milli>(t1 - t0).count() << " ms)\n";
    cout << "B&B : best=" << bbBest << "  ("
         << chrono::duration<double, milli>(t2 - t1).count()
         << " ms, nodes=" << bb.nodes << ")\n";
    cout << "agree = " << (dpBest == bbBest ? "yes" : "NO!") << "\n\n";

    // ---- 範例 4：為什麼叫「偽多項式」——W 變大 DP 就爆 ----
    cout << "=== Example 4: pseudo-polynomial blow-up ===\n";
    for (int bigW : {1000, 100000, 10000000}) {
        auto t3 = chrono::steady_clock::now();
        int r = knapsackDP1D(c, bigW);
        auto t4 = chrono::steady_clock::now();
```

```
142         cout << "W=" << bigW << " -> best=" << r << "  ("
143              << chrono::duration<double, milli>(t4 - t3).count() << "
                    ms)\n";
144     }
145     cout << "(n fixed at 30; time grows linearly with W, the VALUE)\n";
146     return 0;
147 }
```

程式碼 1: knapsack.cpp — 0/1 背包三種解法完整實作

5 執行結果與解讀

```
=== Example 1: classic 3-item case ===
W=50, items (w,v) = (10,60)(20,100)(30,120)
best value = 220 (expect 220)
chosen items: #1(w=20,v=100) #2(w=30,v=120)
1D DP check = 220

=== Example 2: 6-item worked example ===
W=10 -> best = 15, chosen: #0 #1 #3  (total w=10, v=15)

=== Example 3: DP vs branch-and-bound (n=30, W=500) ===
DP  : best=1345  (0.014 ms)
B&B : best=1345  (0.004 ms, nodes=59)
agree = yes

=== Example 4: pseudo-polynomial blow-up ===
W=1000     -> best=1687  (0.031 ms)
W=100000   -> best=1687  (2.80 ms)
W=10000000 -> best=1687  (288.9 ms)
(n fixed at 30; time grows linearly with W, the VALUE)
```

筆記

Example 4 是「偽多項式」最直觀的展示：物品數固定 30 件，只是把容量從 10^3 放大到 10^7，時間就放大一萬倍——雖然答案從頭到尾都是 1687（物品全裝也裝不滿更大的背包）。演算法的複雜度依賴**數值大小**而非**輸入長度**。

6 實務應用

- **資源分配**：預算 W 元投資 n 個專案（各有成本與預期收益），選哪些？
- **雲端資源打包**：把工作負載塞進固定規格的 VM，最大化利用率。
- **廣告投放**：版位容量有限，挑選收益最高的廣告組合。
- **密碼學淵源**：早期的 Merkle–Hellman 背包公鑰系統基於 Subset Sum 難度（後來被破，但歷史意義重大）。
- **FPTAS**：背包有完全多項式時間近似方案——把價值除以 K 取整再跑 DP，可在 $O(n^3/\varepsilon)$ 內保證 $(1-\varepsilon)$ 最優。弱 NP-Hard 問題的專屬福利。

7 練習題

1. 用一維 DP 求 $(w,v) = (1,1),(3,4),(4,5),(5,7)$、$W = 7$，並手動還原方案（提示：一維表無法直接還原，需要另存決策或改用二維）。

2. 把程式改成**無限背包**（每件可拿多次）：只需把一維 DP 的容量迴圈改成正序。驗證 $(2,3)$ 單件物品、$W=10$ 時答案是 15。
3. 實作 FPTAS：價值縮放 $K=\varepsilon\cdot v_{\max}/n$，用 $\varepsilon=0.1$ 對照精確解的誤差。
4. 分支限界目前用「先拿再不拿」的順序分支，改成「先不拿」會怎樣？實測節點數。
5. 挑戰：二維背包（重量與體積雙限制）$O(nWV)$ DP。

8 小結

方法	時間	空間	適用時機
二維 DP	$O(nW)$	$O(nW)$	W 小、要還原方案
一維 DP	$O(nW)$	$O(W)$	W 小、只要最優值
分支限界	最壞 $O(2^n)$	$O(n)$	W 巨大、實例有結構
FPTAS	$O(n^3/\varepsilon)$	$O(n^2/\varepsilon)$	容忍 ε 誤差

延伸閱讀：Kellerer, Pferschy, Pisinger *Knapsack Problems*（背包問題的百科全書）；Martello & Toth *Knapsack Problems: Algorithms and Computer Implementations*。