线性 DP 与背包模型

从数字金字塔到 LIS 与 0-1 / 完全背包

HGNU-ACM 实验室算法组

承接数据结构篇，开启第五篇动态规划：四大核心模型深度推导 + 4 道配套真题

本章目录（上）

动态规划思想与数字三角形

最长上升 / 不下降子序列（LIS）

0-1 背包与一维倒序压缩

前三节建立动态规划思维：数字三角形（记忆化与递推）、LIS（以 $a[i]$ 结尾的状态定义）、0-1 背包（二维压缩与倒序枚举）。

本章目录（下）

完全背包与常见变形

四大配套真题精讲

常见错误、检查清单与练习安排

后三节深入完全背包正序复用，精讲 4 道配套真题（含路径还原与格式细节），并给出完整防错检查清单。

先想一个小例子：为什么不能“每步选最大的”？

生活场景：下楼梯捡金币

有一个 3 层的数字三角形，从顶端出发，每次只能往左下或右下走一步，想让一路捡到的总分最大：

第 1 层：5 → 第 2 层：8（左） 3（右） → 第 3 层：1（左） 2（中） 99（右）

- 只看眼前的贪心：站在 5 时看到 $8 > 3$，于是选左边的 8；结果第 3 层只能走到 1 或 2，总和最多 $5 + 8 + 2 = 15$。
- 真正的最优路线：第二步忍痛走较小的 3，第 3 层就能吃到大金币 99，总和高达 $5 + 3 + 99 = \mathbf{107}$！

本页要点

眼前利益最大不代表全局最优。前面吃一个小亏，可能换来后面更大的收益，所以不能盲目贪心。

为什么需要 DP？暴力搜索的算力危机

既然贪心会掉进坑里，那把所有路径全部走一遍取最大值行不行？

三角形行数 R	每次选择数	暴力搜索路径总数	耗时估算（按 10^8 次/秒）
$R = 5$	每步 2 选 1	$2^4 = 16$ 条	瞬间完成
$R = 20$	每步 2 选 1	$2^{19} \approx 5.2 \times 10^5$ 条	不到 0.01 秒
$R = 30$	每步 2 选 1	$2^{29} \approx 5.3 \times 10^8$ 条	超过 5 秒（**TLE**）
$R = 1000$	每步 2 选 1	$2^{999} \approx 10^{300}$ 条	宇宙毁灭也算不完

- 我们在搜索专题学过 DFS，但数字三角形第 R 行有 1000 层时，路径数是指数爆炸的。
- 仔细想想：走到同一个格子 (i, j) 后，从 (i, j) 往下能拿到的最高分其实是固定不变的！根本没必要每次从上面重新往下算一遍。

本页要点

把每个格子“往下走最多能拿多少分”记在表格里，只算一次、直接查表，就能把 2^R 降成 R^2！

三个核心术语的大白话解释

学动态规划（简称 **DP**）之前，先把课本上最吓人的三个词翻译成大白话：

1. 重叠子问题：不同路径走着走着会汇合到同一个格子 (i, j)。如果不记笔记，同一个格子的最好结果会被重复算成千上万次。
2. 最优子结构：从顶端出发的全局最高分，一定是由“左下角出发的最高分”和“右下角出发的最高分”中较大的那个拼出来的。
3. 无后效性：只要站在 (i, j)，接下来往下最多能拿多少分，只和当前位置有关，和前面是怎么走到这里的毫无关系。

写 DP 之前先回答四个问题

状态怎么定义（表格里存什么）； 转移方程怎么写（怎么用小格子算大格子）； 初值是什么； 按什么顺序填表。

本页要点

记住：动态规划不是什么神秘魔法，就是“定义状态 + 记笔记查表，不重复干同一件事”。

两种填表方向：记忆化递归 vs 自底向上递推

拿到数字三角形，通常有两种实现方式，它们计算量相同，但写法手感大不一样：

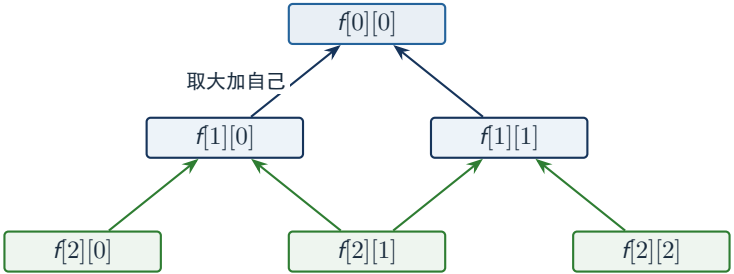
对比维度	记忆化递归（自顶向下）	递推填表（自底向上）
思考方向	从塔顶 $(0,0)$ 往下问，遇到算过的格子直接返回 `memo[i][j]`	从倒数第二层开始，一层层往上汇总到塔顶
边界处理	到底部最后一行直接返回本身的值	最后一行天然就是起点，完全不用判左右越界
最终答案	调用 `dfs(0, 0)` 的返回值	直接输出塔尖 `f[0][0]`
推荐程度	适合复杂树形/图上 DP	强烈推荐：代码极短、无递归栈开销

本页要点

对于数字三角形，自底向上递推把“多条终点取 max”变成“唯一汇入塔尖 f[0][0]”，最省心！

TikZ 状态网图：每个格子只看左下和右下

用 $f[i][j]$ 表示从第 i 行第 j 列走到最底层能拿到的最大和（下标从 0 开始）：



最底层直接等于数字本身（初值边界）

本页要点

状态转移方程：$f[i][j] = a[i][j] + \max\big(f[i+1][j],\ f[i+1][j+1]\big)$。每个格子都只依赖下一层的两个邻居！

手算跟踪表：4 层数字三角形自底向上填表

原三角形为：第 0 行 $[5]$，第 1 行 $[8,3]$，第 2 行 $[4,2,7]$，第 3 行 $[1,2,99,6]$。从第 3 行往第 0 行逐层填表：

填表行号	顺序	本行计算过程（$a[i][j]+\max(f[i+1][j], f[i+1][j+1])$）	本行结果 $f[i]$
$i=3$	初始边界	最底层无法再往下走，直接保留原值	$[1,2,99,6]$
$i=2$	自左向右	$4+\max(1,2)=6$，$2+\max(2,99)=101$，$7+\max(99,6)=106$	$[6,101,106]$
$i=1$	自左向右	$8+\max(6,101)=109$，$3+\max(101,106)=109$	$[109,109]$
$i=0$	汇总塔尖	$5+\max(109,109)=$ **114**	[**114**]

- 注意第 1 行的 8 虽然也能走到 99（$5+8+2+99=114$），第 1 行的 3 也能走到 99（$5+3+7+99=114$），但表格只关心当前最优值，最终准确汇入 $f[0][0]=114$。

本页要点

先写底层初值，再逐行向上合并；当 $i=0$ 填完时，塔尖 $f[0][0]$ 就是全局最优解。

数字三角形模板代码（自底向上原地更新）

```cpp
// 直接在原数组 f 上自底向上覆盖，省去额外数组
for (int i = R - 2; i >= 0; --i) {                  // 从倒数第二行往第 0 行推
    for (int j = 0; j <= i; ++j) {                  // 第 i 行共有 i + 1 个格子
        f[i][j] += max(f[i + 1][j], f[i + 1][j + 1]);
    }
}
cout << f[0][0] << '\n';                            // 答案唯一汇总在塔尖
```

为什么这样写

- 外层从 `R - 2` 开始，因为第 `R - 1` 行已经是底层初值。
- 当 $i \le R-2$ 且 $0 \le j \le i$ 时，下一行至少有 $i+2$ 个元素，所以 $f[i+1][j+1]$ 绝不越界。

本页要点：外层从 $R-2$ 倒序到 0，内层从 0 到 i，短短 5 行就是线性 DP 的标准雏形。

为什么这样写一定对？用不变量说清楚

我们用循环不变量（“每一轮开始前都必然成立的事实”）来验证自底向上填表的正确性：

阶段	循环不变量（恒成立的性质）
初始化	进入循环前，第 $R-1$ 行每个 $f[R-1][j]$ 就是从最后一行出发的最大路径和（只有一个数）。
保持	算第 i 行时，第 $i+1$ 行的所有状态已算好。任何从 (i,j) 出发的路径，第一步必走 $(i+1,j)$ 或 $(i+1,j+1)$，取两者的 max 加上 $a[i][j]$ 必为从 (i,j) 出发的最优解。
终止	循环在 $i=-1$ 时结束，此时第 0 行已算完，因此 $f[0][0]$ 必然是从塔顶 $(0,0)$ 到底部的全局最大路径和。

本页要点

每次“保持”都严格依赖下一行的最优子结构，没有任何漏算或重复。

上机验证：3 组测试数据与反事实追问

先用小数据验证程序边界，再思考“如果改动代码会怎样”：

组别	输入数据	预期输出	测试目的
常规测试	$R=3$：$\{5\},\{8,3\},\{4,2,7\}$ 略改（4 层见前）	114	验证一般情况的路径选择
最小边界	$R=1$：仅一行 $[42]$	42	外层循环 `i = -1` 直接跳过，输出 `f[0][0]`
特殊相等	$R=3$：全为 1 的三角形	3	验证等值转移不出错

反事实追问

- 追问 **1**：如果把外层循环写成正序 `for (int i = 0; i <= R - 2; ++i)` 会怎样？答：算第 i 行时第 $i+1$ 行还是原数据，结果全错（**WA**）！
- 追问 **2**：如果题目要求“从塔顶到底部的最小路径和”，只需把代码中的 `max` 改成什么？答：改成 `min` 即可！

数字三角形：常见错误对照

错误写法	后果与评测结果	正确做法与原因
外层写成 `i = R - 1` 开始	读取 `f[R][j]` 越界访问（**RE / WA**）	从 `R - 2` 开始，第 `R - 1` 行是初值
自顶向下推却忘记最后扫底行最大值	只输出 `f[R-1][0]` 导致 **WA**	改用自底向上汇入 `f[0][0]`，或最后扫描底行求 `max`
用贪心每次选左右较大者	局部最优陷阱，样例过不了（**WA**）	必须用 DP 保存每个状态的全局最优值

本页要点

写完填表代码先用 $R = 1$ 走一遍循环上下界，10 秒就能查出越界错误！

先分清两个词：子串 vs 子序列

看一个具体序列 $a = [1, 5, 3, 4, 4]$：

- 子串（连续子段）：必须在原序列里紧紧挨着。比如 $[5, 3, 4]$ 是子串，$[1, 3, 4]$ 不是子串。
- 子序列：可以跳着挑元素，只要保持原来从左到右的先后顺序即可。比如 $[1, 3, 4, 4]$ 是子序列！

上升 vs 不下降

- 严格上升（LIS）：后面必须严格大于前面（$<$），如 $[1, 3, 4]$ 长度 3。
- 不下降：后面大于等于前面即可（$\leq$），如 $[1, 3, 4, 4]$ 长度 4。

本页要点

做题第一眼先看清“严格上升”还是“不下降”，只差一个等号（< 还是 <=），结果完全不同！

为什么需要 DP？暴力枚举子序列为何 TLE

对于长度为 n 的序列，如果把所有子序列都挑出来检查是不是不下降：

序列长度 n	每个元素选或不选	子序列总数 2^n	运行时间估算
$n = 10$	2 种选择	1024	瞬间完成
$n = 25$	2 种选择	3.3×10^7	勉强卡过
$n = 200$	2 种选择	$2^{200} \approx 1.6 \times 10^{60}$	严重超时（**TLE**）

- 即使本题 $n \le 200$ 看起来很小，指数级 2^n 也是天文数字。
- 而如果用 DP 两重循环逐个找前驱，总运算次数大约是 $n^2/2 \approx 20000$ 次，连 1 毫秒都不到！

本页要点

当 n 达到 30 以上时，2^n 暴力枚举必死无疑；DP 只需把“以每个位置结尾的最好结果”记下来，复杂度降为 $O(n^2)$。

状态的灵魂：为什么必须定义成“以 a[i] 结尾”？

初学者写 LIS 最容易犯的错，就是把状态定义成：

错误定义：$dp[i]=$ “前 i 个数里的最长不下降子序列长度”

为什么接不下去了？假设前 3 个数 $[1,5,3]$ 的最长长度是 2（由 $[1,5]$ 或 $[1,3]$ 组成）。现在第 4 个数 $a[4]=4$ 来了，只知道长度是 2，却不知道这个长度 2 的序列尾巴是 5 还是 3！如果是 5 就接不上 4，如果是 3 就能接上！信息丢了，没法转移。

正确定义：$dp[i]=$ “必须以 $a[i]$ 结尾的最长不下降子序列长度”

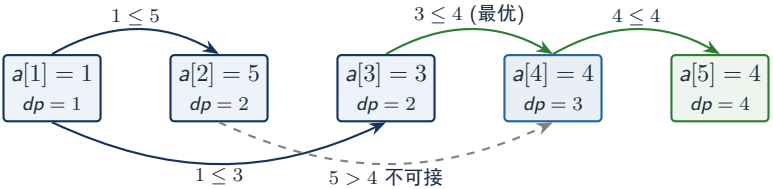
一旦规定死“尾巴必须是 $a[i]$”，前面只要某个 $j<i$ 满足 $a[j]\le a[i]$，就一定能把 $a[i]$ 接在以 $a[j]$ 结尾的序列后面，长度直接变 $dp[j]+1$！

本页要点

状态定义里“以第 i 个元素结尾”这 7 个字，把序列尾巴钉死了，才彻底满足无后效性！

TikZ 跳跃箭头图：向前寻找合法前驱

对于 $a = [1, 5, 3, 4, 4]$，每个位置 i 往左找所有满足 $a[j] \le a[i]$ 的位置 j：



本页要点

每个箭头只表示一次“$a[j] \le a[i]$ 且可以接在后面”；在所有绿色合法箭头里挑 $dp[j]$ 最大的那个加 1！

5×5 手算逐行跟踪表：手算 $a = [1, 5, 3, 4, 4]$

设初始每个位置自己单独成一个长度为 1 的子序列（$dp[i] = 1$）：

当前 i	$a[i]$	逐个检查 $j = 1 \ldots i-1$ 是否满足 $a[j] \le a[i]$	候选最大值	最终 $dp[i]$
1	1	没有前驱，直接保持初始值 1	—	1
2	5	$j = 1$：$1 \le 5$（可接，$dp[1] + 1 = 2$）	2	2
3	3	$j = 1$：$1 \le 3$（可接得 2）；$j = 2$：$5 > 3$（跳过）	2	2
4	4	$j = 1$（得 2）；$j = 2$（跳过）；$j = 3$：$3 \le 4$（得 3）	3	3
5	4	$j = 1$（得 2）；$j = 2$（跳过）；$j = 3$（得 3）；$j = 4$（$4 \le 4$ 得 4）	4	**4**

本页要点

第 5 行中，因为题目是不下降（允许相等），所以 $a[4] = 4$ 能接在 $a[5] = 4$ 前面，把长度推到 4。

转移方程拆解：为什么最终答案不是 dp[n]？

状态转移方程

对于每个 $i = 1, 2, \ldots, n$：

$$dp[i] = \max\left(1, \max_{1 \le j < i,\ a[j] \le a[i]} \left(dp[j] + 1\right)\right)$$

- 为什么答案不一定是 $dp[n]$？假设序列是 $a = [2, 3, 4, 1]$，前三个数一路上升到 $dp[3] = 3$，但最后一个数 $a[4] = 1$ 比前面所有数都小，只能自己重开一段，$dp[4] = 1$！
- 最长子序列不一定非得以最后一个数收尾，它可以停在序列里的任意一个位置。
- 所以算完所有的 $dp[i]$ 之后，答案必须是 $\max_{1 \le i \le n} dp[i]$。

本页要点

定义 $dp[i]$ 是为了方便转移，最后统计答案一定要对整个 $dp[1 \ldots n]$ 取最大值！

LIS 模板代码（求长度）

```
vector<int> dp(n + 1, 1);                    // 每个单元素自身至少构成长度 1 的序列
int ans = 1;                                 // 全局最长长度
for (int i = 1; i <= n; ++i) {               // 枚举当前结尾位置 i
    for (int j = 1; j < i; ++j) {            // 扫描前面所有可能的前驱位置 j
        if (a[j] <= a[i]) {                  // 若严格上升则改为 a[j] < a[i]
            dp[i] = max(dp[i], dp[j] + 1);   // 尝试接在 j 后面刷新长度
        }
    }
    ans = max(ans, dp[i]);                   // 随时更新全局最优答案
}
cout << ans << '\n';
```

初值为何全设为 1？

哪怕一个元素前面没有任何比它小的数（比如整个序列单调递减），它自己单独站在那里也算一个长度为 1 的合法子序列。如果初值设成 0，接上去就会少算自己这 1 个！

为什么这样写一定对？用不变量说清楚

阶段	循环不变量（恒成立的性质）
初始化	进入循环前，对所有 i，$dp[i] = 1$，表示只含 $a[i]$ 自己的长度为 1 的合法子序列。
保持	计算 $dp[i]$ 时，所有 $j < i$ 的 $dp[j]$ 已是“以 $a[j]$ 结尾的最长长度”。只要 $a[j] \le a[i]$，把 $a[i]$ 接上去就得到长度为 $dp[j] + 1$ 的合法序列，取遍所有合法 j 的最大值即得最优 $dp[i]$。
终止	外层循环结束时，$dp[1 \dots n]$ 全部确定；由于任何非空子序列必有最后一个元素，全局最长者必等于 $\max_{1 \le i \le n} dp[i]$。

本页要点

任何子序列都必有唯一的结尾元素，枚举结尾元素 i 就等于覆盖了所有可能的最长子序列。

上机验证：3 组测试数据与反事实追问

组别	输入数据	预期输出	测试目的
常规测试	$n = 5,\ a = [1, 5, 3, 4, 4]$	4	验证不下降（相等可接）且跳过局部大值 5
最小边界	$n = 1,\ a = [99]$	1	内层循环跳过，检验初值 1 是否生效
严格递减	$n = 4,\ a = [8, 6, 4, 2]$	1	没有任何 $a[j] \le a[i]$，检验全局答案仍为 1

反事实追问

- 追问 **1**：如果把 `a[j] <= a[i]` 改成 `a[j] < a[i]`，输入 $[1, 5, 3, 4, 4]$ 输出会变成几？答：末尾两个 4 不能互接，输出会从 4 变成 3！
- 追问 **2**：如果最后写成 `cout << dp[n]`，输入 $[2, 3, 4, 1]$ 会输出几？答：会错输出 1，而正确答案是 3（**WA**）！

LIS 常见错误对照

错误写法	后果与评测结果	正确做法与原因
`vector<int> dp(n+1, 0)`	长度少算自己这 1 个，全错（**WA**）	初值必须全为 1
输出 `dp[n]` 而不是 `ans`	结尾较小时漏掉前面的最长段（**WA**）	对所有 `dp[1..n]` 取 `max`
混淆 `<` 和 `<=`	不下降题写成严格上升（**WA**）	题目说“不下降”用 `<=`，“严格上升”用 `<`
状态定义成“前 i 个最长”	丢失结尾数值，无法判断能否接上	必须定义为“以 $a[i]$ 结尾”

本页要点

LIS 三句话检查：初值是不是 1？等号有没有看清？最后是不是求全局 max？

先想一个小例子：旅行者的背包怎么装？

生活场景：最多装 5 公斤的背包

面前有 3 件物品，每件要么不拿，要么只拿 1 次（所以叫“0-1”背包）：

物品 1：重量 2，价值 3　　物品 2：重量 3，价值 4　　物品 3：重量 4，价值 5

- 为什么不能按“单价（性价比）”贪心？假设背包容量 4，物品 A $(3, 6)$ 单价 2，物品 B $(2, 3.5)$、C $(2, 3.5)$ 单价 1.75。贪心先拿 A 占掉 3 公斤得 6 分，剩下 1 公斤装不下 B/C；不拿 A 而是拿 B+C 却能得 7 分！
- 为什么不能 2^n 暴力搜？当物品数 $n = 30$ 时，每件物品选或不选共 $2^{30} \approx 1.07 \times 10^9$ 种组合，直接超时（**TLE**）。

本页要点

背包容量 M 通常只有几百到几千，用容量作状态，就能把 2^n 降成 $n \times M$！

二维状态定义与两种决策

定义 $dp[i][j]$ 为：只考虑前 i 件物品，背包总重量不超过 j 时能拿到的最大总价值。考虑第 i 件物品（重量 w_i，价值 c_i）时，只有两条路：

1. 不装第 i 件（或者根本装不下，即 $j < w_i$）：那总价值就和“只看前 $i-1$ 件物品、容量 j”一模一样：
$$dp[i][j] = dp[i-1][j]$$
2. 装下第 i 件（前提是 $j \geq w_i$）：先腾出 w_i 的空间，用剩下的容量 $j - w_i$ 去装前 $i-1$ 件物品，再加上 c_i：
$$dp[i][j] = dp[i-1][j-w_i] + c_i$$

合并成二维转移方程

当 $j \geq w_i$ 时，$dp[i][j] = \max\big(dp[i-1][j],\ dp[i-1][j-w_i] + c_i\big)$；否则 $dp[i][j] = dp[i-1][j]$。

二维手算表：容量 $M=5$，3 件物品逐行填表

物品：第 1 件 $(w=2,c=3)$，第 2 件 $(w=3,c=4)$，第 3 件 $(w=4,c=5)$。初值第 0 行全为 0：

考虑前 i 件 \ 容量 j	0	1	2	3	4	5
$i=0$（无物品）	0	0	0	0	0	0
$i=1$：$(2,3)$	0	0	3	3	3	3
$i=2$：$(3,4)$	0	0	3	4	4	$\max(3, dp[1][2]+4)=\mathbf{7}$
$i=3$：$(4,5)$	0	0	3	4	$\max(4,0+5)=5$	$\max(7, dp[2][1]+5)=7$

- 观察第 2 行 $j=5$：腾出 $w_2=3$ 后看第 1 行 $j=5-3=2$ 的值 3，加上 $c_2=4$ 得到 7。
- 最终答案在右下角 $dp[3][5]=7$（即拿第 1 件和第 2 件，总重 $2+3=5$）。

本页要点

第 i 行的每个格子，只看上一行（第 $i-1$ 行）的正上方和左上方！

把二维压成一维：能省掉第一维 i 吗？

既然算第 i 行时，只用到上一行（第 $i-1$ 行）的数据，根本用不到第 $i-2$ 行，我们能不能只开一个一维数组 dp[0..M] 滚动更新？

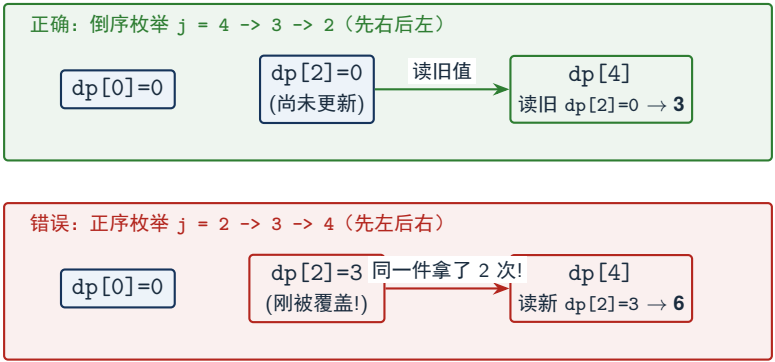
- 如果直接把 i 删掉，方程变成：$dp[j] = \max(dp[j],\ dp[j-w] + c)$。
- 但是有隐患：因为 j - w < j，dp[j - w] 在你的左边！
- 如果从小到大更新（先算左边的 dp[j - w]，再算右边的 dp[j]），等你算 dp[j] 时，左边的 dp[j - w] 已经被本轮新数据覆盖了，不再是上一轮的旧数据！

本页要点

一维数组里，想让 dp[j - w] 保持“上一轮的旧数据”不被覆盖，必须先算右边大的 j，再算左边小的 j！

TikZ 正反对比图：倒序用旧值 vs 正序覆盖成新值

设物品 $(w = 2, c = 3)$，容量 $M = 4$。算 dp[4] 时要读取左边的 dp[2]：



本页要点

倒序保证算 dp[j] 时，所有下标更小的 dp[j - w] 都还保留着上一轮的旧值！

正序写错覆盖跟踪表：为什么一件物品被拿了 2 次？

容量 $M = 4$，只有唯一一件物品 $(w = 2, c = 3)$。正确最大价值只能是 3。看正序循环如何翻车：

循环方向	当前 j	读到的 dp[j - 2] 是谁？	计算过程	更新后 dp[j]
错用正序	$j = 2$	读到旧 dp[0] = 0	$\max(0,\ 0 + 3) = 3$	dp[2] = 3
错用正序	$j = 3$	读到旧 dp[1] = 0	$\max(0,\ 0 + 3) = 3$	dp[3] = 3
错用正序	$j = 4$	读到刚被改写的 dp[2] = 3	$\max(0,\ 3 + 3) = \mathbf{6}$	dp[4] = 6（**WA!**）
正确倒序	$j = 4$	读到旧的 dp[2] = 0	$\max(0,\ 0 + 3) = 3$	dp[4] = 3
正确倒序	$j = 3$	读到旧 dp[1] = 0	$\max(0,\ 0 + 3) = 3$	dp[3] = 3
正确倒序	$j = 2$	读到旧 dp[0] = 0	$\max(0,\ 0 + 3) = 3$	dp[2] = 3

本页要点

只有一个物品时输出了 6（拿了 2 次），这就是 0-1 背包写成正序后的典型 WA 症状。

0-1 背包一维模板代码

```
vector<long long> dp(m + 1, 0);                  // 容量 0..m 的最大价值，初值全 0
for (int i = 0; i < n; ++i) {                    // 依次考虑每一件物品
    int w, c;
    cin >> w >> c;                               // 读入当前物品的重量 w 和价值 c
    for (int j = m; j >= w; --j) {               // 0-1 背包铁律：容量从 m 倒序到 w
        dp[j] = max(dp[j], dp[j - w] + c);       // 不拿 vs 拿这 1 件
    }
}
cout << dp[m] << '\n';                           // 输出容量不超过 m 的最大总价值
```

为什么内层从 m 到 w?

当 j < w 时根本装不下这件物品，dp[j] 保持不变，所以循环停在 j >= w 即可，同时天然防止 j - w < 0 数组越界！ 甚至如果输入的单个物品重量 w > m，条件 j >= w 一开始就为假，自动跳过该物品。

为什么倒序一定等价于二维？用不变量说清楚

阶段	一维滚动数组不变量
初始化	处理第 1 件物品前，`dp[0..m]` 全为 0，等于二维表第 0 行。
保持	第 i 轮从 $j=m$ 倒序到 w。当更新 `dp[j]` 时，所有 $<j$ 的下标（特别是 $j-w$）尚未在本轮修改，依然存着第 $i-1$ 轮的结果；而 $>j$ 的下标虽已更新成第 i 轮，但后续不再用到。
终止	第 n 轮结束时，`dp[m]` 严格等于二维数组中的 $dp[n][m]$。

本页要点

“从大到小更新”使得每次读取的 `dp[j - w]` 都是上一轮的快照，空间从 $O(nM)$ 降到 $O(M)$。

上机验证：3 组测试数据与反事实追问

组别	输入数据（M, n 及物品 w, c）	预期输出	测试目的
常规测试	$M=5, n=3$：$(2,3),(3,4),(4,5)$	7	验证选前两件的最优组合
超重物品	$M=4, n=2$：$(10,99),(3,5)$	5	验证 $w=10>M$ 时循环自动跳过不越界
零容量边界	$M=0, n=1$：$(2,5)$	0	验证 $j=0\geq 2$ 为假，输出 0

反事实追问

- 追问 **1**：如果把内层条件写成 `for (int j = m; j >= 0; --j)` 而不加 `if (j >= w)` 会怎样？答：当 $j<w$ 时访问 `dp[j - w]` 出现负数下标，直接数组越界（**RE**）！
- 追问 **2**：把物品输入顺序打乱，最终答案会变吗？答：不会，每件物品选或不选的结果和输入顺序无关。

扩展：背包三种常见问法与初始化对照

在算法竞赛中，同样是背包，问法稍微变一点，数组初值就要跟着变：

问题问法	数组初值设置	最终答案取法
不超过容量 M 的最大价值（最常见）	`dp[0..m]` 全部初始化为 0	直接输出 `dp[m]`
必须恰好装满 M 的最大价值	`dp[0] = 0`，其余 `dp[1..m]` 设为 `-INF`	若 `dp[m] < 0` 则无解，否则为 `dp[m]`
恰好装满 M 的方案总数	`dp[0] = 1`，其余 `dp[1..m]` 设为 0	转移改为 `dp[j] += dp[j - w]`，输出 `dp[m]`

本页要点

“最多装”全填 0，“恰好装满求最大”除 `dp[0]=0` 全填负无穷，“恰好装满求方案数”除 `dp[0]=1` 全填 0。

0-1 背包死穴对照

错误写法	后果与评测结果	正确做法与原因
内层写成 `j = w; j <= m; ++j`	同一件物品被无限拿取（**WA**）	0-1 背包必须倒序 `j = m; j >= w; --j`
内层终止条件写 `j >= 0`	访问 `dp[j - w]` 负下标（**RE**）	终止于 `j >= w` 即可
`dp` 数组用 `int` 存大价值	物品数和价值大时溢出（**WA**）	价值累加和方案数养成习惯开 `long long`
把 M 和 n 的读入顺序写反	读错容量和物品数（**WA / RE**）	看清题面先读容量还是先读物品数

本页要点

0-1 背包只要记住两件事：内层倒序防复用，终止边界 `j >= w` 防负下标。

引入：当每种物品有无限件时怎么办？

现在规则变了：每种物品不再只拿 1 件，而是可以拿 0 件、1 件、2 件、$\ldots$ 无限多件，只要总重量不超过 M！

- 笨办法：第 i 种物品最多能装 $k = \lfloor M/w_i \rfloor$ 件，把它硬拆成 k 件一模一样的物品跑 0-1 背包。
- 为什么太慢：如果 $M = 10^4, w_i = 1, n = 1000$，拆完总件数高达 10^7，直接超时（**TLE**）。
- 神奇的发现：刚才我们在 0-1 背包里，不小心把内层循环写成正序时，发生了什么？一件物品被重复拿了好多次！这恰好就是完全背包想要的效果！

本页要点

把 0-1 背包的“致命错误（正序循环）”反过来利用，就得到了 $O(nM)$ 的完全背包！

为什么正序就能实现“无限次拿取”？

看二维转移的本质区别：

- **0-1 背包**：装这件物品前，前面不能已经装过它，所以看上一行 `dp[i-1][j - w]`（旧数据）。
- **完全背包**：装了 1 件之后，还可以继续再装第 **2** 件、第 **3** 件！所以腾出 w 空间后，应该看同一行左边已经考虑过第 i 种物品的状态 `dp[i][j - w]`（本轮新数据）！

落到一维数组上

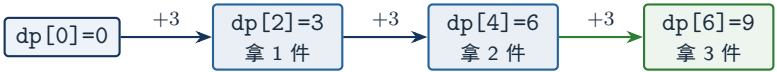
只要把内层循环从 `j = w` 到 `m` 正序枚举，算 `dp[j]` 时，左边的 `dp[j - w]` 刚刚才在本轮被更新过（已经允许拿当前物品了），再加一次 c，就是多拿一件！

本页要点

倒序保护旧值（最多 1 件），正序利用新值（无限件）。

TikZ 依赖箭头图：正序同一行向右累加

以物品 $(w=2, c=3)$、容量 $M=6$ 为例，看正序遍历如何自动实现拿 1 件、2 件、3 件：



正序 `j = 2 -> 4 -> 6`：每次都踩在前一次刚更新的新值上

本页要点

正序更新就像滚雪球：`dp[2]` 含 1 件 $\rightarrow$ `dp[4]` 在它基础上变 2 件 $\rightarrow$ `dp[6]` 变 3 件。

手算跟踪表：容量 $M = 6$，两种无限物品正序填表

物品 1：$(w = 2, c = 3)$，物品 2：$(w = 3, c = 5)$。初值 dp[0..6] 全为 0：

轮次 \ 容量 j	0	1	2	3	4	5	6
初始	0	0	0	0	0	0	0
第 1 轮 $(2,3)$ 正序	0	0	3	3	6	6	9
第 2 轮 $(3,5)$ 正序	0	0	3	$\max(3,5) = 5$	6	$\max(6, 3+5) = 8$	$\max(9, 5+5) = \mathbf{10}$

- 第 2 轮算 $j = 6$ 时，dp[3] 刚刚在本轮被更新为 5（拿了 1 个物品 2），所以 $dp[3] + 5 = 10$（相当于拿了 2 个物品 2），打败了 3 个物品 1 的 9！

本页要点

每轮正序扫完一遍，当前物品“拿任意多件”的所有可能就已经全部融合进 dp 数组里。

0-1 背包 vs 完全背包本质对比表

对比项目	0-1 背包	完全背包
每种物品数量	最多 1 件	无限多件
二维转移方程	$\max(dp[i-1][j],\ \mathbf{dp}[\mathbf{i}-\mathbf{1}][j-w]+c)$	$\max(dp[i-1][j],\ \mathbf{dp}[\mathbf{i}][j-w]+c)$
一维循环方向	倒序 `j = m; j >= w; --j`	正序 `j = w; j <= m; ++j`
读取数据新旧	必须读上一轮的旧值	故意读本轮刚刚更新的新值
核心防错	千万别写正序	千万别写倒序

本页要点

两段代码外观几乎一模一样，唯独内层 `for` 循环的方向完全相反！

完全背包模板代码

```
vector<long long> dp(m + 1, 0);                    // 容量 0..m 的最大价值，初值全 0
for (int i = 0; i < n; ++i) {                      // 依次考虑每一种物品
    int w, c;
    cin >> w >> c;                                 // 读入当前物品的重量 w 和价值 c
    for (int j = w; j <= m; ++j) {                 // 完全背包铁律：容量从 w 正序到 m
        dp[j] = max(dp[j], dp[j - w] + c);         // 利用本轮新值，允许重复拿取
    }
}
cout << dp[m] << '\n';                             // 若题目要求前缀如 "max=" 则按题面输出
```

循环边界为何从 j = w 开始？

因为当 j < w 时一件也装不下，状态不会改变；从 w 开始直接到 m，既省时间又保证 j - w >= 0 绝不越界！如果某个物品 w > m，条件 j <= m 一开始就为假，安全跳过。

为什么正序一定对？用不变量说清楚

阶段	完全背包一维循环不变量
初始化	处理第 1 种物品前，`dp[0..m]` 全为 0，表示没有任何物品可拿。
保持	第 i 轮从 $j = w$ 正序到 m。计算 `dp[j]` 时，`dp[j - w]` 已经完成了第 i 轮更新，存着“前 i 种物品装入容量 $j - w$（第 i 种可拿 $0, 1, 2 \ldots$ 次）”的最优解。加上 c 即表示再多拿 1 件第 i 种物品。
终止	第 n 轮结束后，`dp[m]` 必为前 n 种物品无限选取的最大总价值。

本页要点

归纳证明清楚表明：正序更新自动涵盖了选 $0, 1, 2, \ldots, \lfloor m/w \rfloor$ 件的所有情况。

常见变形速览：多重背包与求方案数

掌握了 0-1 和完全背包，另外两个高频变形一眼就能看穿：

1. 多重背包（第 i 种最多拿 s_i 件）：
   - 不能用正序（否则会超过 s_i 件），把 s_i 拆成 $1, 2, 4, \ldots$ 的二进制包裹（如 $s = 13$ 拆成 $1 + 2 + 4 + 6$），每个包裹看成一件新物品跑 **0-1** 背包倒序！
   - 任意 $0 \sim 13$ 的件数都能用 $\{1, 2, 4, 6\}$ 拼出来，复杂度从 $O(s_i)$ 降成 $O(\log s_i)$。
2. 求恰好凑出 M 的方案数（如硬币凑钱）：
   - 把 `max` 改成 `+`：`dp[j] += dp[j - w]`；初值设为 `dp[0] = 1`，其余全 0。
   - 硬币无限个就用正序（完全背包），每种硬币限 1 枚就用倒序（0-1 背包）。

本页要点

二进制拆分把“有限件”变成“0-1 背包”；求方案数把 `max` 换成 `+`、`dp[0]=1`。

完全背包：上机验证与常见易错

组别	输入数据（M, n 及物品 w, c）	预期输出	测试目的
常规对比	$M=6, n=2$：$(2,3),(3,5)$	10	检验选 2 个 $(3,5)$ 得到 10（0-1 只能得 8）
超重物品	$M=3, n=1$：$(5,100)$	0	验证 $w=5>M$ 时正序条件为假，不越界
单件恰好	$M=6, n=1$：$(2,3)$	9	验证同一件可拿 3 次

反事实追问与易错点

- 追问：如果“多重背包（限 2 件）”直接偷懒用完全背包的正序循环，会怎样？答：程序可能拿 3 件甚至更多，直接 **WA**！
- 格式易错：部分题（如一本通 1268）要求输出 `max=12`，漏写前缀 `max=` 会判 **WA / PE**。

例 1 | 数字金字塔 [入门]：先读懂要求

- 输入：第一行一个整数 R（$1 \le R \le 1000$）；接下来 R 行，第 i 行有 i 个非负整数（均 ≤ 100）。
- 处理：从最顶层走到最底层任意位置，每次只能走正下方（左下）或右下方，求经过数字的最大总和。
- 输出：单独一行一个整数，即最大路径和。

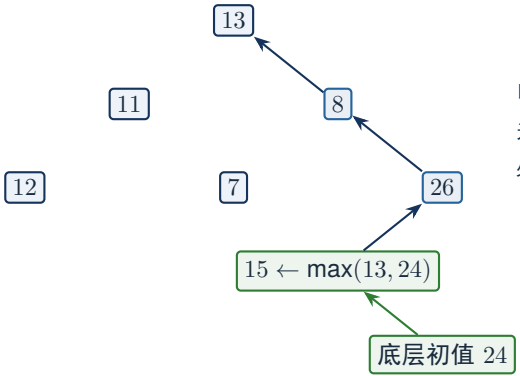
原题样例（$R = 5$）

输入：`13 / 11 8 / 12 7 26 / 6 14 15 8 / 12 7 13 24 11`
输出：`86`（路径 $13 \to 8 \to 26 \to 15 \to 24 = 86$）

本页要点

注意 $R \le 1000$，每个数 ≤ 100，最大总和不超过 $1000 \times 100 = 10^5$，用 `int` 足够。

例 1 | 数字金字塔：思路与建模图解



自底向上合并：

无需判 $j = 0$ 左越界

答案自动归集到塔尖

本页要点

用 `f[i][j]` 存第 i 行第 j 个数，从第 $R-2$ 行自底向上令 `f[i][j] += max(f[i+1][j], f[i+1][j+1])`！

例 1 | 数字金字塔：原题样例手推跟踪表

按自底向上顺序，把原题 5 层样例从第 4 行推到第 0 行（下标从 0 开始）：

行号 i	原行数据	自底向上合并后的 $f[i]$
$i=4$	$[12,\ 7,\ 13,\ 24,\ 11]$	$[12,\ 7,\ 13,\ 24,\ 11]$（底层初值不变）
$i=3$	$[6,\ 14,\ 15,\ 8]$	$[6+12,\ 14+13,\ 15+24,\ 8+24]=[18,\ 27,\ 39,\ 32]$
$i=2$	$[12,\ 7,\ 26]$	$[12+27,\ 7+39,\ 26+39]=[39,\ 46,\ 65]$
$i=1$	$[11,\ 8]$	$[11+46,\ 8+65]=[57,\ 73]$
$i=0$	$[13]$	$[13+\max(57,73)]=[\mathbf{86}]$

本页要点

手推核对：$i=0$ 时 $13+73=86$，与原题样例输出完全一致！

例 1 | 数字金字塔：核心步骤与相对模板改动

步骤	按顺序计算	纸上得到	核对问题
1	读入 R 并按行 `resize(i+1)`	锯齿形二维数组 f	第 i 行是否刚好读入 $i+1$ 个数？
2	外层 `i = R-2` 倒序到 0	从倒数第二行逐层向上	$R = 1$ 时循环是否直接跳过？
3	内层 `j = 0..i` 原地更新	`f[i][j] += max(...)`	是否误写成 `f[i+1][j-1]`？
4	输出 `f[0][0]`	唯一最大值 86	是否多输出了多余空格或文字？

相对基础模板改了哪里？

直接把输入数组 `f` 当作 DP 表进行原地覆盖，并用 `f[i].resize(i + 1)` 动态分配每行长度，不用开 1000×1000 的静态大数组。

例 1 | 数字金字塔：完整核心代码

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
    ios::sync_with_stdio(false); cin.tie(nullptr);
    int R; if (!(cin >> R)) return 0;
    vector<vector<int>> f(R);
    for (int i = 0; i < R; ++i) {
        f[i].resize(i + 1);
        for (int j = 0; j <= i; ++j) cin >> f[i][j];
    }
    for (int i = R - 2; i >= 0; --i)            // 从倒数第二行往塔顶推
        for (int j = 0; j <= i; ++j)
            f[i][j] += max(f[i + 1][j], f[i + 1][j + 1]);
    cout << f[0][0] << '\n';                    // 对应源码 code/pyramid_dp.cpp
    return 0;
}
```

本页要点：原地覆盖，零额外数组开销（对应 code/pyramid_dp.cpp）。

例 1 | 数字金字塔：易错点与 3 组自测数据

组别	自测输入	预期输出	检验目的
最小边界	`1 / 5`	5	外层循环不执行，直接输出 `f[0][0]`
全零边界	`3 / 0 / 0 0 / 0 0 0`	0	非负整数含 0 的情况
等值分支	`3 / 1 / 5 5 / 2 9 2`	15	两条路都能走到 9，结果唯一

常见易错

- 写错下标方向：若从 0 开始但把“左下方/右下方”误写成 `j-1` 和 `j`，在 $j = 0$ 时会直接越界（**RE**）。
- 外层从 `R-1` 开始：访问 `f[R][j]` 越界（**RE**）。

例 2 | 求最长不下降序列 [基础]：先读懂要求

- 输入：第一行 n（$1 \le n \le 200$）；第二行 n 个互不相同的整数 b_i（$1 \le b_i \le 10^5$）。
- 处理：求出长度最长的不下降子序列，不仅要输出长度，还要输出其中任意一条完整序列。
- 输出：第一行输出 `max=` 最大长度；第二行输出该序列，空格隔开。

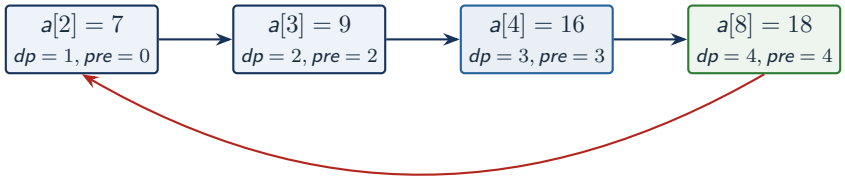
原题样例

输入：`14 / 13 7 9 16 38 24 37 18 44 19 21 22 63 15`
输出：`max=8`（换行）`7 9 16 18 19 21 22 63`

本页要点

和基础模板相比，本题多了一个难点：如何把选出来的那 8 个数还原并打印出来！

例 2 | 求最长不下降序列：用 pre[] 还原路径



- 每次发现 $a[j] \le a[i]$ 且接在 j 后面能让长度变长（或相等更新：`dp[j] + 1 >= dp[i]`），就顺手记录前驱：`pre[i] = j`。
- 同时用 `endp` 记录全局最长序列的最后一个位置。
- 结束后从 `endp` 顺着 `pre` 一路倒爬到 0，存入 `path` 后再用 `reverse` 翻正即可！

本页要点

“刷新状态时顺手记下前驱 `pre[i] = j`”，是所有 DP 还原具体方案的通用绝招！

例 2 | 求最长不下降序列：样例前 8 项手推跟踪表

取原题样例前 8 个数 $a[1..8] = [13, 7, 9, 16, 38, 24, 37, 18]$，逐步记录 dp[i] 与 pre[i]：

i	$a[i]$	满足 $a[j] \le a[i]$ 的最优前驱 j	dp[i]	pre[i]	当前最长终点
1	13	无	1	0	$endp = 1$ (1)
2	7	无	1	0	$endp = 2$ (1)
3	9	$j = 2$ ($7 \le 9$)	2	2	$endp = 3$ (2)
4	16	$j = 3$ ($9 \le 16$)	3	3	$endp = 4$ (3)
5	38	$j = 4$ ($16 \le 38$)	4	4	$endp = 5$ (4)
6	24	$j = 4$ ($16 \le 24$)	4	4	$endp = 6$ (4)
7	37	$j = 6$ ($24 \le 37$)	5	6	$endp = 7$ (5)
8	18	$j = 4$ ($16 \le 18$)	4	4	$endp = 7$ (5)

本页要点

到第 8 项结束时，再往后 $19, 21, 22, 63$ 都会一路接在 $a[8] = 18$ 后面，最终凑出长度 8！

例 2 | 求最长不下降序列：核心步骤与相对模板改动

步骤	按顺序计算	纸上得到	核对问题
1	初始化 dp 全 1，pre 全 0	n 个单点链，前驱为 0	下标是否从 1 开始（0 留作空标记）?
2	双重循环更新 dp[i]	刷新时记 pre[i] = j	是否同步更新全局 best 与 endp?
3	从 endp 沿 pre 倒推	逆序序列 path	是否记得 reverse(path)?
4	按题面格式输出	max=... 与空格序列	行末是否有多余空格?

相对基础模板改了哪里?

增加了前驱数组 pre[i] 和终点下标 endp； 当 dp[j] + 1 >= dp[i] 时更新 pre[i] = j； 循环后沿 pre 倒爬并反转。

例 2 | 求最长不下降序列：核心代码 (1/2)

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
    ios::sync_with_stdio(false); cin.tie(nullptr);
    int n; if (!(cin >> n)) return 0;
    vector<int> a(n + 1), dp(n + 1, 1), pre(n + 1, 0);
    for (int i = 1; i <= n; ++i) cin >> a[i];
    int best = 1, endp = 1;                          // 记录全局最长长度与终点下标
    for (int i = 1; i <= n; ++i) {
        for (int j = 1; j < i; ++j)
            if (a[j] <= a[i] && dp[j] + 1 >= dp[i]) {
                dp[i] = dp[j] + 1; pre[i] = j; // 刷新长度并记录前驱 j
            }
        if (dp[i] >= best) { best = dp[i]; endp = i; }
    }
```

本页要点：第 1 部分完成读入、DP 转移以及前驱记录，下页接路径回溯与输出。

例 2 | 求最长不下降序列：核心代码 (2/2)

```
    vector<int> path;
    for (int u = endp; u != 0; u = pre[u])      // 顺着 pre 一路倒推回起点
        path.push_back(a[u]);
    reverse(path.begin(), path.end());          // 倒推得到的是逆序，必须反转！
    cout << "max=" << best << '\n';             // 注意输出前缀 max=
    for (int i = 0; i < (int)path.size(); ++i)
        cout << path[i] << (i + 1 == (int)path.size() ? '\n' : ' ');
    return 0;
}
```

本页要点

两页合起来与源码 `code/lis_path.cpp` 完全一致；用 `u != 0` 判终点要求数组下标从 1 开始！

例 2 | 求最长不下降序列：易错点与 3 组自测数据

组别	自测输入	预期输出	检验目的
最小边界	`1 / 42`	`max=1` 与 `42`	检验 `best=1, endp=1` 能否正常输出
严格递减	`3 / 9 5 1`	`max=1` 与任意单数	无前驱时 `pre[u]=0` 正常终止
全上升	`4 / 2 4 6 8`	`max=4` 与 `2 4 6 8`	检验 `reverse` 后顺序是否正序

常见易错

- 忘记 `reverse`：顺着 `pre` 倒推出来的是从尾到头，直接输出会变成倒序（**WA**）。
- 下标从 **0** 开始却用 `pre = 0` 作空标记：会把下标 0 当成空导致死循环或漏掉第 0 个数！下标从 0 存时，`pre` 初值必须设为 `-1`。

例 3 | 01 背包问题 [基础]：先读懂要求

- 输入：第一行两个整数 M, N（$1 \le M \le 200, 1 \le N \le 30$），注意先容量 M，后物品数 N！接下来 N 行每行 W_i, C_i。
- 处理：每件物品最多选 1 次，总重量不超过 M，求最大总价值。
- 输出：一行一个整数，表示最大总价值。

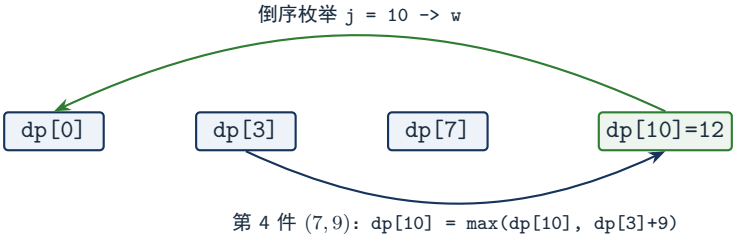
原题样例（$M = 10, N = 4$）

输入：`10 4 / 2 1 / 3 3 / 4 5 / 7 9`
输出：`12`（选第 2 件 $(3, 3)$ 和第 4 件 $(7, 9)$，重 $3 + 7 = 10$，价值 $3 + 9 = 12$）

本页要点

看清读入顺序：这题是 `cin >> m >> n`，千万别按习惯先读 `n` 再读 `m`！

例 3 | 01 背包问题：思路与建模图解



- 本题 $M \le 200, N \le 30$，使用一维滚动数组 `dp[0..M]` 只需 200 个位置。
- 对每件物品 (w, c)，从 `m` 倒序枚举到 `w`，保证每件物品至多被选中 1 次。

本页要点

直接套用标准 0-1 背包一维倒序模板即可秒杀。

例 3 | 01 背包问题：原题样例手推跟踪表

输入 $M = 10, N = 4$，物品依次为 $(2,1),(3,3),(4,5),(7,9)$。一维数组 `dp[0..10]` 逐轮状态如下：

轮次 \ 容量 j	0	1	2	3	4	5	6	7	8	9	10
初始	0	0	0	0	0	0	0	0	0	0	0
第 1 件 $(2,1)$ 倒序	0	0	1	1	1	1	1	1	1	1	1
第 2 件 $(3,3)$ 倒序	0	0	1	3	3	4	4	4	4	4	4
第 3 件 $(4,5)$ 倒序	0	0	1	3	5	5	6	8	8	9	9
第 4 件 $(7,9)$ 倒序	0	0	1	3	5	5	6	9	9	10	**12**

本页要点

第 4 轮算 $j = 10$ 时，读取的是第 3 轮的旧值 `dp[3] = 3`，加上 9 正好得到 12！

例 3 | 01 背包问题：核心步骤与相对模板改动

步骤	按顺序计算	纸上得到	核对问题
1	先读入 m 再读入 n	容量 10，物品数 4	有没有把 m 和 n 读反？
2	初始化 `dp(m+1, 0)`	全 0 数组	数据类型是否用了 `long long`？
3	每件倒序 `j = m..w`	逐轮刷新 `dp[j]`	循环方向是否为 `--j`？
4	输出 `dp[m]`	单个整数 12	本题有没有要求输出 `max=`？（没有！）

相对基础模板改了哪里？

几乎完全一致，唯一要注意的是读入顺序先 m 后 n，且输出是一个纯整数（不要画蛇添足加 `max=`）。

例 3 | 01 背包问题：完整核心代码

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
    ios::sync_with_stdio(false); cin.tie(nullptr);
    int m, n; if (!(cin >> m >> n)) return 0;   // 先读背包容量 m，再读物品数 n
    vector<long long> dp(m + 1, 0);
    for (int i = 0; i < n; ++i) {
        int w, c; cin >> w >> c;
        for (int j = m; j >= w; --j)            // 0-1 背包铁律：倒序循环 j
            dp[j] = max(dp[j], dp[j - w] + c);
    }
    cout << dp[m] << '\n';                      // 对应源码 code/knapsack_01.cpp
    return 0;
}
```

本页要点：14 行极简代码拿下 0-1 背包（对应 code/knapsack_01.cpp）。

例 3 | 01 背包问题：易错点与 3 组自测数据

组别	自测输入	预期输出	检验目的
最小边界	`1 1 / 2 10`	0	唯一物品重 $2 > 1$，装不下输出 0
倒正序区分	`4 1 / 2 3`	3	若错写成正序会输出 6，一秒抓错！
恰好装满	`5 2 / 2 3 / 3 4`	7	两件全拿刚好占满 5 公斤

常见易错

- 错把正序当倒序：在原题样例下，如果把内层写成正序，会重复拿第 3 件 $(4,5)$ 两次并加上 $(2,1)$ 凑出 11，或拿 5 次 $(3,3)$？$(3,3)$ 拿 3 次是 9，$(4,5)$ 拿 2 次加 $(2,1)$ 是 11，刚好都是 ≤ 12，容易在样例上漏网！必须用 `4 1 / 2 3` 自测！

例 4 | 完全背包问题 [基础]：先读懂要求

- 输入：第一行 M, N（$1 \le M \le 200, 1 \le N \le 30$）；接下来 N 行每行 W_i, C_i，每种物品数量无限。
- 处理：在总重量不超过 M 的前提下，最大化总价值。
- 输出：一行，格式必须为 `max=` 最大总价值！

原题样例（$M = 10, N = 4$）

输入：`10 4 / 2 1 / 3 3 / 4 5 / 7 9`
输出：`max=12`

本页要点

虽然输入数据和例 3 一模一样，但例 4 物品无限件，且输出多了一个 `max=` 前缀！

例 4 | 完全背包问题：思路与建模图解



- 为什么同一份样例，完全背包答案还是 12？因为 $(4,5)$ 拿 2 件重 8 价值 10，剩 2 只能拿 $(2,1)$ 得 11；反而不如拿 1 件 $(3,3)$ 和 1 件 $(7,9)$ 得 12。
- 如果把 $(4,5)$ 改成 $(4,6)$，完全背包拿 2 件得 12 再加 $(2,1)$ 就能拿到 13，立刻超过 0-1 背包！

本页要点

只需把 0-1 背包的内层倒序改成正序 `j = w..m`，就能无缝支持无限件拿取。

例 4 | 完全背包问题：原题样例手推跟踪表

输入 $M = 10, N = 4$，物品依次为 $(2,1),(3,3),(4,5),(7,9)$。正序循环下 `dp[0..10]` 逐轮变化：

轮次 \ 容量 j	0	1	2	3	4	5	6	7	8	9	10
初始	0	0	0	0	0	0	0	0	0	0	0
第 1 种 $(2,1)$ 正序	0	0	1	1	2	2	3	3	4	4	5
第 2 种 $(3,3)$ 正序	0	0	1	3	3	4	6	6	7	9	9
第 3 种 $(4,5)$ 正序	0	0	1	3	5	5	6	8	10	10	11
第 4 种 $(7,9)$ 正序	0	0	1	3	5	5	6	9	10	10	**12**

本页要点

对比第 3 行 `j=8`：0-1 背包是 8，完全背包因为正序允许拿 2 个 $(4,5)$ 变成了 10！

例 4 | 完全背包问题：核心步骤与相对模板改动

步骤	按顺序计算	纸上得到	核对问题
1	读入 m, n	容量 10，种数 4	先 m 后 n 顺序正确吗？
2	初始化 `dp(m+1, 0)`	全 0 数组	是否用了 `long long`？
3	每种正序 `j = w..m`	逐轮允许复用当前物品	循环方向是否为 `++j`？
4	按要求带前缀输出	`max=12`	是否漏写了 `"max="`？

相对基础模板改了哪里？

只需在输出语句加上题目要求的 `"max="` 前缀：`cout << "max=" << dp[m] << '\n';`。

例 4 | 完全背包问题：完整核心代码

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
    ios::sync_with_stdio(false); cin.tie(nullptr);
    int m, n; if (!(cin >> m >> n)) return 0;   // 先读背包容量 m，再读物品种数 n
    vector<long long> dp(m + 1, 0);
    for (int i = 0; i < n; ++i) {
        int w, c; cin >> w >> c;
        for (int j = w; j <= m; ++j)            // 完全背包铁律：正序循环 j
            dp[j] = max(dp[j], dp[j - w] + c);
    }
    cout << "max=" << dp[m] << '\n';            // 对应源码 code/knapsack_complete.cpp
    return 0;
}
```

本页要点：注意第 13 行严格按题面格式输出 `max=`（对应 `code/knapsack_complete.cpp`）。

例 4 | 完全背包问题：易错点与 3 组自测数据

组别	自测输入	预期输出	检验目的
倒正序区分	`6 1 / 2 3`	`max=9`	拿 3 次，若写错成倒序只得 3
最小边界	`2 1 / 5 10`	`max=0`	单件超重时跳过循环输出 `max=0`
高性价比小件	`10 2 / 3 4 / 7 9`	`max=13`	1 个 $(3,4)$ + 1 个 $(7,9)$ = 13

常见易错

- 格式 **WA**：很多同学代码逻辑全对，却因为直接输出 `dp[m]` 而没带 `max=` 导致丢分！交题前务必再看一遍“输出格式”。
- 死记硬背写反方向：牢记“0-1 倒序、完全正序”，用 `6 1 / 2 3` 这组最小数据一秒验真假。

线性 DP 与背包高频错误对照与定位指南

题目 / 场景	常见错误写法	正确做法与原因
0-1 背包正序循环	写 `for (int j = w; j <= m; ++j)`	变成同一件物品被无限拿取（**WA**）
完全背包倒序循环	写 `for (int j = m; j >= w; --j)`	每种物品最多只拿了 1 次（**WA**）
容量循环下界写错	写 `j >= 0` 却没判 `j >= w`	读 `dp[j - w]` 产生负下标（**RE**）
LIS 初值遗漏	`dp` 初始全 0 而非 1	单元素自身也算长度 1 的子序列（**WA**）
输出格式漏前缀	《完全背包》直接输出数字	题面要求输出 `max=` 前缀（**WA/PE**）

- 快速定位：**WA** 90% 查三件事——循环方向（倒序/正序）、数组初值（1 / 0 / $-\infty$）、输出格式与 `long long`。

本页要点

“0-1 倒着防重复，完全正着允复用；初值看清恰好还是至多”。

提交前检查清单（六必查）

1. 滚动一维背包时，0-1 背包内层容量循环是不是严格从 `m` 倒序到 `w`？
2. 完全背包（物品无限）内层容量循环是不是严格从 `w` 正序到 `m`？
3. 如果输入的物品重量 w 大于背包容量 M，你的循环边界会不会自然跳过而不越界？
4. LIS 问题中，看清了“严格上升（<)”还是“不下降（<=)”吗？dp 初值全设成 `1` 了吗？
5. 价值总和或方案数在最大值域下会不会爆 `int`？是不是已经开了 `long long`？
6. 题目有没有要求打印 `max=` 前缀或还原完整选择路径（`pre[]` + `reverse`)？

最后一步

交题前用 $M = 4$、唯一物品 $(w = 2, c = 3)$ 手算一遍：0-1 只能得 3，完全才能得 6！

本章全部练习与完成标准

题目	难度	对应源码	核心思路提要
数字金字塔 (1258)	[入门]	`code/pyramid_dp.cpp`	自底向上原地合并左下与右下最大值
求最长不下降 (1259)	[基础]	`code/lis_path.cpp`	以 $a[i]$ 结尾转移 + `pre[]` 倒爬还原
01 背包问题 (1267)	[基础]	`code/knapsack_01.cpp`	一维滚动数组倒序枚举容量 `m..w`
完全背包问题 (1268)	[基础]	`code/knapsack_complete.cpp`	一维滚动数组正序枚举容量 `w..m` + `max=`
庆功会 (1269)	[提高]	—	多重背包：二进制拆分成 0-1 背包求解

完成标准

能一秒说出 0-1 倒序与完全正序的区别；能默写 `pre[]` 路径还原；独立 AC 上述精选习题。

课后安排与下一章预告

- 复习：合上代码再画一遍 0-1 背包“倒序用旧值、正序覆盖成新值”的状态图。
- 练习：把 `code/` 下 4 份源码默写打通，并练习“恰好装满”时 `dp[0]=0`、其余 `-INF` 的初始化写法。
- 下一章预告：第 2 章《区间与环形 DP》——不再从左往右一个元素一个元素推，而是从小区间合并成大区间！遇到首尾相连的圆环又怎么“破环成链”！

本章一句话

写好状态含义就成功了一半，剩下靠无后效性的转移方向把小答案汇成大海。

谢谢！