动态规划

把大问题拆成小状态，按顺序填表

HGNU-ACM 实验室算法组

截图配套：23 道例题，27 道练习题





本章目录 | 知识板块

动态规划基础

线性动态规划

背包动态规划

区间与划分动态规划

本章目录 | 后半

二维表动态规划

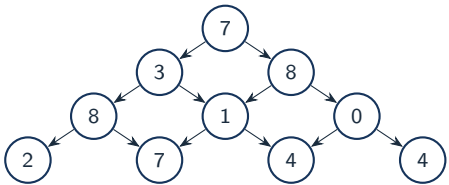
记忆化搜索

易错点与通用模板

本章全部练习题索引

先看一个数字三角形

从顶部出发，每次只能走到左下方或右下方的点，一直走到最底层。把沿途的数字相加，怎样走和最大？



- 逐层枚举：第 1 层有 1 个点，第 i 层有 2^{i-1} 条路线。
- 30 层就有约 10^9 条路线，暴力枚举一定超时。
- 同一个位置会被很多条路线经过，能不能只算一次？

要点：把“从顶点走到某个位置的最大和”记下来，后面就可以直接复用。

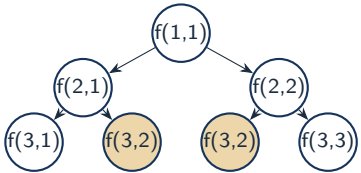
暴力为什么慢：同一个子问题被反复计算

把“从顶点走到 (i, j) 的最大和”写成递归函数：

$$f_{i,j} = a_{i,j} + \max\left(f_{i-1,j-1},\ f_{i-1,j}\right).$$

- 暴力枚举“路线”，不是“状态”。
- 同一位置被多条路线经过，存一次就够。

层数越多重复越严重，总调用次数接近 2^n。



金色结点 $f(3,2)$ 被算了两次。

要点：动态规划 $=$ 不重复计算：每个状态只求一次，结果存进数组。

动态规划的一句话定义

定义

动态规划（Dynamic Programming，DP）把一个大问题拆成若干相互重叠的小问题，先求小问题并把答案存进表里，再用小问题的答案按顺序推出大问题的答案。

- 关键词一：重叠。小问题会被多次用到，所以值得存。
- 关键词二：表。表的一格就是一个“状态”，格子里的数就是该状态的答案。
- 关键词三：顺序。必须先算“更小”的状态，再算依赖它的状态。

要点：DP 不是一种具体算法，而是一种“设计状态、组织计算顺序”的思想。

DP 的四要素总览



状态	用下标把“处理进度”说清楚，例如 $f(i,j)$：走到第 i 层第 j 个位置。
转移	当前状态由哪些更小的状态得到，例如 $f(i,j) = a_{i,j} + \max(f(i-1,j-1), f(i-1,j))$。
边界	最小的状态能直接写出，例如 $f(1,1) = a_{1,1}$。
答案	最终输出的状态，例如 $\max_j f(n,j)$。

要点：写代码前，先用四句话把这四要素各写一行；写不出就说明还没想清楚。

要素一：状态——给“进度”起一个名字

状态是 DP 的表的一格。它要回答两个问题：

- 我现在处理到哪了？（下标）
- 到这个状态为止，最有用的信息是什么？（值）

数字三角形中，$f(i,j)$ 的下标 (i,j) 就是“第 i 层第 j 个位置”，值是从顶点走到这里的最大和。

$$f(i,j) = a_{i,j} + \max\big(f(i-1,j-1), f(i-1,j)\big).$$

	$j=1$	$j=2$	$j=3$	$j=4$
$i=1$	$f(1,1)$			
$i=2$	$f(2,1)$	$f(2,2)$		
$i=3$	$f(3,1)$	$f(3,2)$	$f(3,3)$	
$i=4$	$f(4,1)$	$f(4,2)$	$f(4,3)$	$f(4,4)$

表的一格就是一个状态；金色格子是 $f(3,2)$。

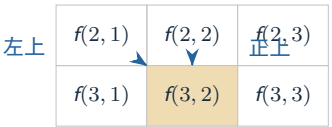
要点：状态的下标通常就是“进度条”：序列下标、区间端点、容量、网格坐标。

要素二：转移——枚举“最后一步”

转移回答：当前状态可以由哪些更小的状态得到？

- 数字三角形：走到 (i,j) 的最后一步，只能从左上 $(i-1,j-1)$ 或正上 $(i-1,j)$ 来。
- 两类来源互不重叠，且覆盖全部可能。
- 取最大值，就得到转移式。

$$f(i,j) = a_{i,j} + \max\big(f(i-1,j-1), f(i-1,j)\big).$$



$f(3,2)$ 的两个前驱：$f(2,1)$ 与 $f(2,2)$。

要点：转移式的写法：把“最后一步”分类，每类对应一个更小状态，再按题目要求取 max / min / 求和。

要素三：边界——递归到哪里停

边界是最小的、可以直接写出答案的状态。

- 数字三角形：$f(1,1) = a_{1,1}$，顶点没有前驱。
- LCS：$f(0,j) = f(i,0) = 0$，空串与任何串的公共子序列长度为 0。
- 背包：$f(0,j) = 0$，没有物品时价值为 0。
- 区间 DP：$f(i,i) = 0$，单个元素不需要合并。

0	0	0	0
0	$f(1,1)$	$f(1,2)$	$f(1,3)$
0	$f(2,1)$	$f(2,2)$	$f(2,3)$
0	$f(3,1)$	$f(3,2)$	$f(3,3)$

蓝色是第一行、第一列；它们是表格的“地基”。

边界写错，后面所有格子都会跟着错。

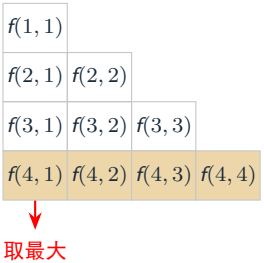
要点：初始化不是随手写 0：最大值、最小值、计数、恰好装满，四种情况的边界各不相同。

要素四：答案——最后输出哪一格

答案不一定是 $f(n, m)$，要按题意确定：

- 数字三角形：底层所有位置的最大值 $\max_j f(n, j)$。
- LIS：所有 $f(i)$ 的最大值，不是 $f(n)$。
- LCS：右下角 $f(n, m)$。
- 背包：$f(M)$（容量不超过 M）。
- 方案题：还要沿 pre 数组回溯。

先确定答案，再决定最后一行代码输出什么。



数字三角形的答案是底层一行的最大值。

要点：答案可能是一个格子、一行/一列的最值，或所有状态的最值；先写清楚再编码。

四要素串起来：数字三角形的四行答案

把四要素写成四行

状态	$f(i,j)$：从顶点走到第 i 层第 j 个位置的最大和。
转移	$f(i,j) = a_{i,j} + \max\big(f(i-1,j-1), f(i-1,j)\big)$。
边界	$f(1,1) = a_{1,1}$；越界位置按 0 处理。
答案	$\max_{1 \le j \le n} f(n,j)$。

- 这四行就是程序的骨架：数组维度、循环体、初始化、输出。
- 如果题目要求输出路径，再增加一个 $pre(i,j)$ 记录“从哪个前驱来”。
- 四行写不出来，说明状态或转移还没想清楚，不要急着写代码。

要点：先有四行，再有代码；代码只是把四行翻译成循环。

什么时候能用 DP：三个条件

最优子结构	无后效性	重叠子问题
大问题的最优解由小问题的最优解组成	状态确定后，未来只依赖状态	同一个状态会被多次用到

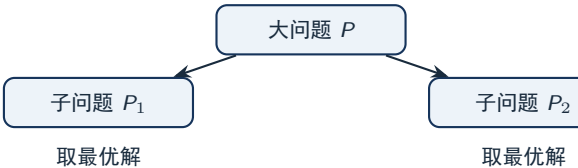
- 三个条件要同时满足，才适合用 DP。
- 最优子结构保证“用小问题的最优解能推出大问题的最优解”。
- 无后效性保证“状态可以替代历史”，状态不用记录完整路线。
- 重叠子问题保证“存表”有收益；如果子问题从不重复，分治更合适。

要点：判断一道题能不能 DP，先逐条检查这三个条件。

条件一：最优子结构

最优子结构：大问题的最优解，一定包含子问题的最优解。

- 数字三角形：整体最大路径在某个位置分开后，上半段也必须是到该位置的最大和；否则换一条更大的上半段，整体会更优。
- $0/1$ 背包：最优方案对前 $i-1$ 件物品的选取，也必须是相应容量下的最优。
- 证明方法：反证法——若子问题不是最优，替换后大问题变优，矛盾。



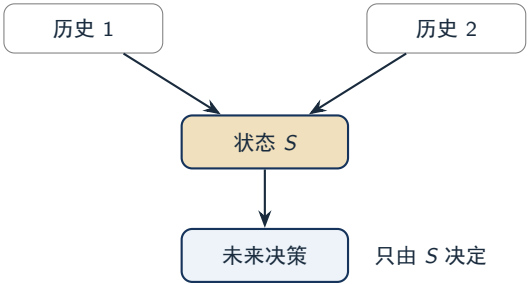
子问题的最优解可以直接拼成大问题的最优解。

要点：如果“子问题最优”拼不出“大问题最优”，就不能直接套 DP。

条件二：无后效性

无后效性：状态一旦确定，后面的决策只依赖这个状态，不依赖“是怎样走到这里”的具体历史。

- 数字三角形：走到 (i,j) 的和只与 (i,j) 有关；从左上还是右上走来，已经包含在 $f(i,j)$ 里。
- 反例：只记录“已用容量”，却忘记“处理到第几件物品”，就无法判断下一件能不能选。
- 判断方法：两个不同的历史，只要状态相同，后续最优选择是否一定相同？



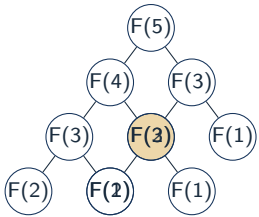
不同历史汇入同一状态后，未来必须走同一条路。

要点：状态要“总结过去、决定未来”；漏掉影响未来的信息，就会有后效性。

条件三：重叠子问题

重叠子问题：同一个状态会在不同的计算路径中被多次用到。

- 数字三角形的 $f(3,2)$ 会被两条路径各调用一次。
- 斐波那契的 $F(5)$ 会反复调用 $F(3)$、$F(2)$。
- 有重叠，存表才有意义；每个状态只算一次，复杂度从指数降到状态数乘转移数。
- 若子问题互不重复（如归并排序的左右两半），用分治，不需要 DP。



金色与未着色的 $F(3)$ 是同一个子问题。

要点：重叠 + 存表 = 每个状态只算一次；这就是 DP 相对暴力的核心优势。

三个条件缺一不可：对照表

条件	满足时	不满足时	典型例子
最优子结构	可直接用子问题最优解	拼不出全局最优	最长简单路径
无后效性	状态可以替代历史	要记录更多历史	需记录“用过哪些”
重叠子问题	存表有收益	分治更快	归并排序

- 三个条件都满足：设计状态，写转移，填表。
- 缺最优子结构：考虑搜索、贪心或其它模型。
- 缺无后效性：给状态加维度。
- 缺重叠：直接用分治或递归。

要点：DP 的适用性判断，比背模板更重要。

状态设计方法：先找“进度”

一个序列：$f(i)$ 前 i 个	两个序列：$f(i,j)$ 前 i、前 j 个
一个区间：$f(l,r)$ 区间 $[l,r]$	背包：$f(i,j)$ 前 i 件、容量 j
网格：$f(i,j)$ 走到 (i,j)	DAG：$f(u)$ 到 u 或从 u 出发

- 状态的下标就是“进度条”：每走一步，至少有一个下标增加。
- 先问“我处理到哪了”，再问“还需要记住什么”；常见维度是位置、前缀长度、区间端点、容量、是否持有。

要点：状态设计的第一步不是写公式，而是找“进度”。

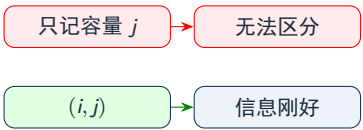
状态设计：完整，但不冗余

完整性：影响未来决策的信息不能丢。

- 0/1 背包只记容量不够，还要记“处理到第几件”。
- 股票买卖还要记“当前是否持有股票”。

冗余性：不影响未来的信息不要留。

- 多一个维度，状态数就乘一个量级。
- 能由其它下标算出的量，不再单独开一维。



上：状态漏信息；下：状态刚好够用。

要点：状态要“刚好够用”：能决定未来，又不重复记录过去。

状态设计：从题意到下标（对照）

题目特征	状态	下标含义
数字三角形	$f(i,j)$	第 i 层第 j 个位置
最长不下降子序列	$f(i)$	以第 i 个数结尾
最长公共子序列	$f(i,j)$	X 的前 i 个、Y 的前 j 个
0/1 背包	$f(i,j)$	前 i 件物品、容量 j
合并石子	$f(l,r)$	区间 $[l,r]$
方格取数	$f(s,i_1,i_2)$	总步数、两条路径的行号

- 先确定“进度”有几个自由度，就有几个下标。
- 下标范围决定数组大小和循环边界；能互相推出的下标只留一个。

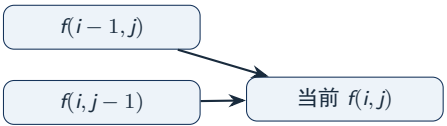
要点：把题意翻译成“进度”，再把“进度”翻译成下标。

转移方向：从“最后一步”倒推

写转移式时，不要从第一步往后想，而是问：最后一步做了什么？

- 0/1 背包：最后一件选或不选。
- LCS：最后一个字符相等或不相等。
- 合并石子：最后一次合并的分界点 k。
- 数字三角形：从左上或正上来。

分类要互不重叠、覆盖全部，否则会重复或漏解。



每个前驱对应“最后一步”的一类选择。

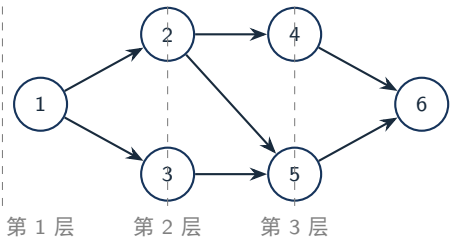
要点：正向想不清时，倒着想“最后一步”；这是写转移式最常用的方法。

循环顺序：按状态依赖图的拓扑序

把每个状态看成一个点，转移看成一条有向边。所有边都必须从“先算”指向“后算”，这张图必须是有向无环图（DAG）。

- 数字三角形：按层从上到下。
- 区间 DP：按区间长度从小到大。
- 0/1 背包：先物品、后容量；容量倒序。

循环顺序就是拓扑序；写反了，用到的是旧值或未算的值。



按层编号 $1 \to 2 \to 3 \to \cdots \to 6$ 计算，所有前驱都已算好。

要点：依赖图有环，说明状态设计有问题；无环才能按拓扑序填表。

推与拉：两种等价的转移写法

拉（**pull**）：站在当前状态，回头看前驱。

$$f(i,j) = \max\big(f(i-1,j), f(i,j-1)\big) + a_{i,j}.$$

适合“每个状态只由少数前驱决定”的题，代码直观。

推（**push**）：站在当前状态，向前更新后继。

$$f(i+1,j) \leftarrow \max\big(f(i+1,j), f(i,j) + a_{i+1,j}\big).$$

适合“一个状态影响很多后继”的题，例如 DAG 上的松弛。

```
// 拉：先算 f[i][j] 的前驱，再填自己
f[i][j] = max(f[i - 1][j], f[i][j - 1]) + a[i][j];

// 推：先算好 f[i][j]，再去更新它能到达的状态
f[i + 1][j] = max(f[i + 1][j], f[i][j] + a[i + 1][j]);
```

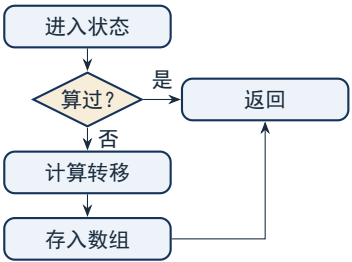
要点：两种写法等价，选一种并保持循环顺序一致；不要在同一道题里混用。

记忆化搜索：先写递归，再把答案存表

有些状态图是 DAG，但转移方向不直观（例如网格上向四个方向滑），这时可以写递归函数。

- 第一次进入某状态时计算，结果存进数组。
- 再次进入直接返回；每个状态只算一次。

递归三件事：出口 $\rightarrow$ 查表 $\rightarrow$ 枚举转移并保存。



查表命中就返回；没命中才计算，算完立刻存。

要点：记忆化搜索 $=$ 递归 $+$ 备忘录；它把“填表顺序”交给递归调用栈。

记忆化搜索模板：数字三角形

```
int a[105][105], f[105][105];

int solve(int i, int j) {
    if (i == 1) {                       // 边界：顶点
        return a[1][1];
    }
    if (f[i][j] != 0) {                 // 查表：算过就返回
        return f[i][j];
    }
    int left = solve(i - 1, j - 1);  // 左上
    int up = solve(i - 1, j);        // 正上
    f[i][j] = a[i][j] + max(left, up); // 计算并保存
    return f[i][j];
}
```

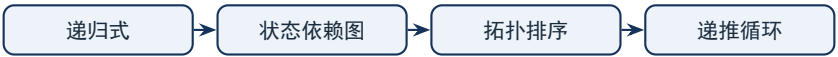
要点：函数返回前把结果写进 `f[i][j]`；下一次进入同一状态时直接取。

记忆化搜索模板：三个要点

- 递归函数只负责“从当前状态往前找”，不需要事先想好循环顺序。
- 数组初值为 0 表示“没算过”；若答案可能为 0，要另设标记数组。
- 递归深度就是状态依赖链的长度，太深时要改成递推。

要点：记忆化适合“转移方向不直观”的题；递推适合“顺序容易确定”的题。

记忆化与递推的转换



记忆化（自顶向下）	递推（自底向上）
从大状态递归到小状态	从小状态循环到大状态
不需要显式确定顺序	必须保证前驱先算
有递归调用和栈开销	没有递归开销，常数更小
可能重复进入但被备忘录挡住	每个状态恰好算一次
适合稀疏状态图	适合稠密、规则的表

- 转换步骤：写出递归式 $\rightarrow$ 画出依赖边 $\rightarrow$ 拓扑排序 $\rightarrow$ 用循环按序填表。
- 两种写法必须使用同一组状态和转移；答案与边界完全一致。

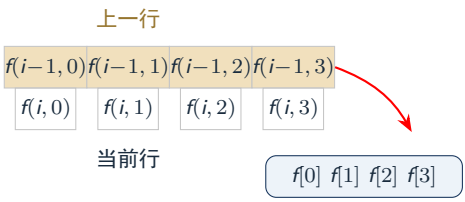
要点：先写记忆化验证正确性，再改递推优化常数，是常用的开发顺序。

空间优化：滚动数组的原理

如果第 i 层的状态只依赖第 $i-1$ 层，就不需要保存整张二维表，只保留“上一行”即可。

- 数字三角形：$f(i,j)$ 只依赖 $f(i-1,\cdot)$。
- LCS：$f(i,j)$ 只依赖第 $i-1$ 行和第 i 行。
- 0/1 背包：$f(i,j)$ 只依赖 $f(i-1,\cdot)$。

条件：依赖必须局限在“上一层”或“固定窗口”；若依赖很多历史层，就不能只留一行。



只保留上一行，当前行直接覆盖到一维数组。

要点：滚动数组的前提：当前层只依赖上一层；否则会丢失还需要的历史。

滚动数组：方向就是语义

0/1 背包：容量倒序



j 从大到小，$f[j-w]$ 还没被本轮更新，物品只用一次。

完全背包：容量正序



j 从小到大，$f[j-w]$ 已被本轮更新，物品可重复用。

一句话

0/1 背包倒序，表示“每件最多一次”；完全背包正序，表示“每件可以无限次”。循环方向就是物品数量的含义。

要点：写一维背包前，先问自己：这件物品允许用几次？答案决定循环方向。

空间优化：常见问题的空间对比

问题	原空间	优化后	做法
数字三角形	$O(n^2)$	$O(n)$	只留上一层，从下往上覆盖
LCS	$O(nm)$	$O(m)$	两行滚动，注意左上角旧值
0/1 背包	$O(nM)$	$O(M)$	一维数组，容量倒序
完全背包	$O(nM)$	$O(M)$	一维数组，容量正序
编辑距离	$O(nm)$	$O(m)$	两行滚动，保留左上角
区间 DP	$O(n^2)$	通常不压	依赖多个短区间，压了会丢值

要点：空间优化是“用空间换时间”之外的另一种取舍；前提是依赖关系允许。

空间优化：三个注意点

- 滚动数组会覆盖旧值；若转移还需要旧值，先用临时变量存起来。
- 两行滚动时，更新当前行前要保留“左上角”的旧值，否则 LCS/编辑距离会错。
- 先写二维版本保证正确，再改一维版本；不要一上来就压空间。

要点：能压就压，但先保证正确：二维版本是调试时的“标准答案”。

复杂度：状态数乘转移数

- 时间复杂度的常用估算：

$$\text{时间} = \text{状态总数} \times \text{每个状态的转移次数}.$$

- 数字三角形有 $\frac{n(n+1)}{2}$ 个状态，每个状态看两个前驱，所以是 $O(n^2)$。
- 空间复杂度通常等于状态总数；如果第 i 层只依赖第 $i-1$ 层，可以滚动数组压到 $O(n)$。
- 先估复杂度再写代码：若状态数是 10^7，每个状态再枚举 10^3 个转移，就已经不可接受。

要点：DP 的核心工作是“设计状态”，代码只是把转移式翻译成循环。

常见模型的复杂度

模型	状态数	转移	时间
数字三角形	$O(n^2)$	$O(1)$	$O(n^2)$
LIS（朴素）	$O(n)$	$O(n)$	$O(n^2)$
LCS	$O(nm)$	$O(1)$	$O(nm)$
0/1 背包	$O(nM)$	$O(1)$	$O(nM)$
区间 DP	$O(n^2)$	$O(n)$	$O(n^3)$

- 先数状态，再数每个状态的转移；两者相乘就是时间复杂度。
- 空间看需要保留多少层；能滚动就滚动。

要点：复杂度估算写在代码之前，能避免“写完才发现超时”。

常见错误清单

错误	后果	检查方法
状态含义不清	转移式写错、答案取错	用一句话写出 f 的完整含义
边界漏写	整张表跟着错	手算 $n=1$、空串、容量 0
初始化用错	不可行状态被当成可行	最大值/0、最小值/$+\infty$、计数/1
循环方向反	$0/1$ 变完全、完全变 $0/1$	问“这件物品允许用几次”
下标越界	运行时错误	数组多开 2 格，循环边界写清楚
数值溢出	答案错误	乘积、方案数、前缀和用 `long long`
答案取错	输出不是题目要的	先写答案状态，再写输出语句
忘记前驱	方案无法输出	更新最优值时同步记录 *pre*

要点：DP 的错误大多不是语法错误，而是状态含义、边界或循环顺序错了。

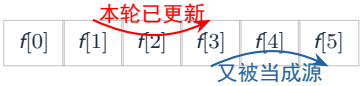
错误示意：0/1 背包写成正序

错误写法：容量 j 从小到大。

$$f[j] = \max\big(f[j],\ f[j-w] + c\big).$$

计算 $f[j]$ 时，$f[j-w]$ 可能已经在同一轮被当前物品更新过，于是这件物品被选了多次，0/1 背包退化成了完全背包。

- 检查方法：问“这件物品允许用几次”。
- 0/1 背包：j 从 M 递减到 w。
- 完全背包：j 从 w 递增到 M。



正序时，左边的源可能已经是“选了当前物品”的新值。

要点：循环方向就是物品数量的语义；写反了，样例可能过，大数据会错。

调试 DP 的三步法

1. 打印表格：把 dp 数组按行打印，和手算的小表逐格对照，先定位第一个不一致的格子。
2. 暴力对拍：写一个 $n \le 10$ 的枚举/搜索，随机生成小数据，比较两个程序的输出。
3. 边界测试：$n = 1$、全部相等、全为 0、负数、最大规模、无解、多解，逐项检查。

小技巧

把转移式中的每个前驱用不同颜色标在纸上；如果某个格子的值不对，沿着它的前驱一路往回查，通常三步内能找到错误。

要点：先保证小数据正确，再优化复杂度；不要用大数据调试。

线性 DP：状态排成一条线或一张表

- 线性 DP 的状态通常写成 $f(i)$ 或 $f(i,j)$，转移只从前一个、前几个或上一行来。
- 常见模型：最长不下降子序列、最长公共子序列、数字三角形、方格取数、编辑距离。
- 做题顺序：先把样例在草稿纸上填成一张表，确认每个格子由哪些格子得到，再写循环。
- 本节从数字金字塔开始，逐题训练“状态、转移、边界、答案”四要素。

要点：先把表填对，再把表翻译成代码。

例 9.2 | 数字金字塔：题意与状态

题意

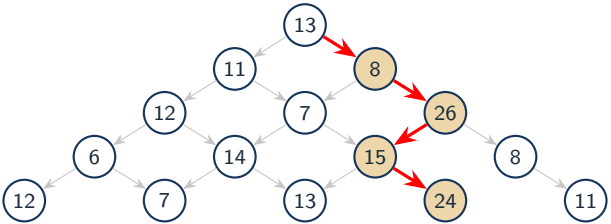
给出一个 R 层的数字金字塔（$1 \le R \le 1000$），从顶点出发，每次走到左下方或右下方的点，直到最底层。求路径上数字之和的最大值。

- 样例：5 层，最大路径 $13 \to 8 \to 26 \to 15 \to 24$，和为 86。
- 定义 $f(i,j)$：从顶点走到第 i 层第 j 个位置时，能得到的最大和。
- 第 (i,j) 个位置只能从 $(i-1,j-1)$ 或 $(i-1,j)$ 走来。

$$f(i,j) = a_{i,j} + \max\big(f(i-1,j-1),\ f(i-1,j)\big).$$

要点：边界 $f(1,1) = a_{1,1}$；越界位置按 0 处理即可，因为所有数非负。

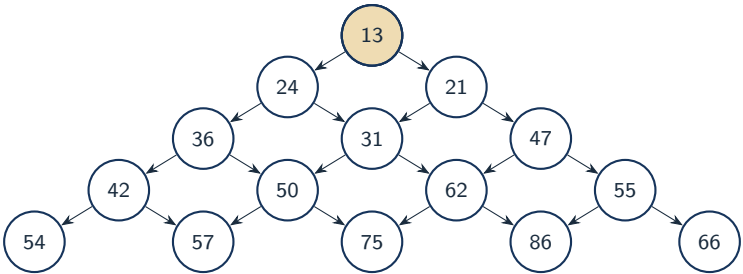
例 9.2 | 样例的最优路径



最优路径 $13 \to 8 \to 26 \to 15 \to 24$，和为 $13+8+26+15+24=86$。

要点：所有路线共享子路径；先算每个位置的最大和，最优路线自然浮出来。

例 9.2 | 把样例逐层填出来



每个圆圈里写的是 $f(i,j)$。例如 $f(3,2) = 7 + \max(24,21) = 31$，$f(4,3) = 15 + \max(31,47) = 62$。最后一层的最大值 86 就是答案。

要点：填表顺序：从上到下、每层从左到右。每个格子用到的是上一层的两个格子。

例 9.2 | 完整程序（上）：读入与边界

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

int main() {
    int R;
    cin >> R;
    vector<vector<int>> a(R + 1, vector<int>(R + 1, 0));
    for (int i = 1; i <= R; ++i) {
        for (int j = 1; j <= i; ++j) {
            cin >> a[i][j];          // 第 i 层第 j 个数
        }
    }
    vector<vector<int>> f(R + 2, vector<int>(R + 2, 0));
    f[1][1] = a[1][1];               // 边界：顶点
```

要点：数组多开两行两列，越界位置自然是 0，省去额外判断。

例 9.2 | 完整程序（下）：填表与答案

```
    for (int i = 2; i <= R; ++i) {
        for (int j = 1; j <= i; ++j) {
            f[i][j] = a[i][j] + max(f[i - 1][j - 1], f[i - 1][j]);
        }
    }
    int ans = 0;
    for (int j = 1; j <= R; ++j) {
        ans = max(ans, f[R][j]);      // 底层最大值
    }
    cout << ans << '\n';
    return 0;
}
```

要点：填表顺序：从上到下、每层从左到右；每个格子用到上一层的两个格子。

例 9.2 | 复杂度、优化与易错点

- 状态数 $\frac{R(R+1)}{2}$，每个状态 $O(1)$ 转移，时间 $O(R^2)$。
- 空间 $O(R^2)$；若只保留上一层，可压成 $O(R)$。
- 最大和不超过 $1000 \times 100 = 10^5$，`int` 足够，但习惯上用 `long long` 更稳。
- 易错：把 $f(i,j)$ 定义成“从 (i,j) 到底层的最大和”也可以，但此时要从下往上填，答案是 $f(1,1)$，不要混用两种定义。

一维写法

从倒数第二层往上，令 `f[j] = a[i][j] + max(f[j], f[j+1])`，最后 `f[1]` 就是答案。这样空间从 $O(R^2)$ 降到 $O(R)$。

例 9.3 | 最长不下降序列：题意与状态

题意

给定 n 个整数（$1 \le n \le 200$），求一个最长的子序列，使下标递增且数值不下降（$b_{i_1} \le b_{i_2} \le \cdots$）。输出长度和其中一组方案。

- 样例有长度为 8 的序列：$7, 9, 16, 18, 19, 21, 22, 63$。
- 定义 $f(i)$：以第 i 个数结尾的最长不下降子序列长度。
- “以 i 结尾”这个限制很重要，它让转移只依赖前面的位置。

$$f(i) = 1 + \max\{f(j) \mid 1 \le j < i,\ b_j \le b_i\}.$$

要点：边界 $f(i) = 1$（只选自己）；答案 $\max_i f(i)$。

例 9.3 | 手推样例并记录前驱

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14
b_i	13	7	9	16	38	24	37	18	44	19	21	22	63	15
$f(i)$	1	1	2	3	4	4	5	4	6	5	6	7	8	3
$pre(i)$	0	0	2	3	4	4	6	4	7	8	10	11	12	3

- $f(3) = 2$：前面只有 $7 \le 9$，接在 7 后面。
- $f(8) = 4$：$7, 9, 16$ 后面接 18，所以 $pre(8) = 4$。
- 最大值 8 在 $i = 13$，沿 pre 回溯得到 $7, 9, 16, 18, 19, 21, 22, 63$。

要点：样例对答案不唯一，题目采用特殊评测；只要输出任意一组最长方案即可。

例 9.3 | 完整程序（上）：输入与初始化

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

int main() {
    int n;
    cin >> n;
    vector<int> b(n + 1);
    vector<int> f(n + 1, 1);          // 以 i 结尾的最长长度
    vector<int> pre(n + 1, 0);        // 最优前驱
    for (int i = 1; i <= n; ++i) {
        cin >> b[i];
    }
    int best = 1;
    int pos = 1;
```

要点：$f(i)$ 的初值是 1：只选自己也是一条合法子序列。

例 9.3 | 完整程序（中）：转移与前驱

```
for (int i = 1; i <= n; ++i) {
    for (int j = 1; j < i; ++j) {
        if (b[j] <= b[i] && f[j] + 1 > f[i]) {
            f[i] = f[j] + 1;
            pre[i] = j;           // 记录前驱
        }
    }
    if (f[i] > best) {
        best = f[i];
        pos = i;                  // 记录最优终点
    }
}
```

要点：条件 $b_j \le b_i$ 对应“不下降”；若要求严格上升，改成 $b_j < b_i$。

例 9.3 | 完整程序（下）：输出方案

```
    cout << "max=" << best << '\n';
    vector<int> ans;
    for (int i = pos; i != 0; i = pre[i]) {
        ans.push_back(b[i]);
    }
    reverse(ans.begin(), ans.end());
    for (size_t i = 0; i < ans.size(); ++i) {
        if (i > 0) {
            cout << ' ';
        }
        cout << ans[i];
    }
    cout << '\n';
    return 0;
}
```

要点：沿 *pre* 从终点回溯到 0，反转后就是正向方案；输出格式是 `max=8`。

例 9.3 | 复杂度与易错点

- 时间 $O(n^2)$：每个 i 扫描所有 $j < i$；$n \le 200$ 时约 4×10^4 次比较。
- 空间 $O(n)$：b, f, pre 三个一维数组。
- 题目说“不下降”，所以判断条件是 $b_j \le b_i$，不能写成 $<$。
- 若要求“最长上升子序列”，把条件改成 $b_j < b_i$ 即可，框架完全一样。
- 记录 pre 时用“严格大于”更新，可以保留最后一次的最优前驱，样例正好得到题面给出的方案。

进阶

用二分加贪心可以把最长上升子序列做到 $O(n \log n)$，但那个方法不记录方案，且“不下降”要用 `upper_bound`。本章先掌握 $O(n^2)$ 的 DP 框架。

例 9.4 | 拦截导弹：两个问题，两个模型

题意

导弹依次飞来，高度不超过 30000，数量不超过 1000。一套系统第一发可以打到任意高度，之后每一发都不能高于前一发。求：

1. 一套系统最多能拦截多少枚导弹；
2. 若要拦截全部导弹，最少需要几套系统。

- 一套系统拦截的导弹高度构成一个不上升子序列。
- 第一问：最长不上升子序列的长度。
- 第二问：把整个序列划分成最少的不上升子序列，由 Dilworth 定理，它等于最长严格上升子序列的长度。

要点：同一个输入，两个方向相反的序列问题。

例 9.4 | 第一问：最长不上升子序列

样例高度：$389, 207, 155, 300, 299, 170, 158, 65$。

$$f(i) = 1 + \max\{f(j) \mid j < i,\ h_j \geq h_i\}.$$

- $389, 300, 299, 170, 158, 65$ 长度为 6，这是第一问的答案。
- 注意条件 $h_j \geq h_i$：后一枚可以等于前一枚，所以是“不上升”，不是“严格下降”。
- 状态定义与例 9.3 完全一样，只把比较符号反过来。

要点：第一问答案 6。$O(n^2)$ 足以处理 $n \leq 1000$。

例 9.4 | 第二问：为什么等于最长严格上升子序列

- 每个拦截系统对应一条不上升的链；要把所有导弹分成若干条这样的链。
- 最长严格上升子序列中的任意两枚导弹不能在同一条不上升链上，所以系统数至少是 LIS 长度。
- 反过来，用贪心：维护每套系统最后一次拦截的高度；新导弹来时，选“最后一发高度 $\geq$ 新高度”中最低的那套，若没有就新开一套。这个贪心恰好用 LIS 长度套系统。

$$g(i) = 1 + \max\{g(j) \mid j < i,\ h_j < h_i\}, \qquad \text{答案} = \max_i g(i).$$

样例的严格上升子序列如 $155, 300$，长度为 2，所以最少 2 套。

要点：第一问用 $\geq$，第二问用 $<$；两个符号不要写反。

例 9.4 | 完整程序（上）：读入与初始化

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

int main() {
    vector<int> h(1);                   // h[0] 占位
    int x;
    while (cin >> x) {                  // 读到文件结束
        h.push_back(x);
    }
    int n = (int)h.size() - 1;
    vector<int> down(n + 1, 1);         // 最长不上升
    vector<int> up(n + 1, 1);           // 最长严格上升
```

要点：输入没有给出导弹数量，用 `while (cin >> x)` 读到文件结束。

例 9.4 | 完整程序（中）：两个 DP

```
for (int i = 1; i <= n; ++i) {
    for (int j = 1; j < i; ++j) {
        if (h[j] >= h[i]) {        // 不上升：允许相等
            down[i] = max(down[i], down[j] + 1);
        }
        if (h[j] < h[i]) {         // 严格上升：不允许相等
            up[i] = max(up[i], up[j] + 1);
        }
    }
}
```

要点：两个转移只差一个比较符号，这是本题最容易写反的地方。

例 9.4 | 完整程序（下）：输出两个答案

```
    int ans1 = 0;
    int ans2 = 0;
    for (int i = 1; i <= n; ++i) {
        ans1 = max(ans1, down[i]);
        ans2 = max(ans2, up[i]);
    }
    cout << ans1 << '\n';
    cout << ans2 << '\n';
    return 0;
}
```

要点：第一行是最长不上升长度，第二行是最长严格上升长度（最少系统数）。

例 9.4 | 复杂度与易错点

- 时间 $O(n^2)$，空间 $O(n)$；$n \le 1000$ 时约 5×10^5 次比较。
- 输入是“一行若干个数”，不要只读一个数就结束。
- 第一问允许相等，第二问不允许相等，这是本题最容易错的地方。
- 若用贪心求第二问，必须选“能接住的系统中最后一发最低的”，不能随便选一套。

拓展

最长上升子序列可以用树状数组或二分做到 $O(n \log n)$；但本题要同时求两个方向，$O(n^2)$ 的写法更直观，也足够快。

例 9.5 | 城市交通路网：DAG 上的最短路

题意

给出 N 个城市之间的单向费用矩阵，$a_{i,j} > 0$ 表示从 i 到 j 有一条边，0 表示没有边。路线只能从编号小的城市走向编号大的城市。求 1 到 N 的最小费用和路径。

- 定义 $dp(j)$：从城市 1 到城市 j 的最小费用。
- 所有边都从小指向大，所以按 $j = 2, 3, \ldots, N$ 的顺序计算即可。

$$dp(j) = \min_{i<j,\ a_{i,j}>0} \big(dp(i) + a_{i,j}\big), \qquad dp(1) = 0.$$

要点：这是“DAG 上的最短路径”：没有环，不需要 Dijkstra，按拓扑序（编号顺序）推一遍即可。

例 9.5 | 样例手推：每个城市的最省费用

j	1	2	3	4	5	6	7	8	9	10
$dp(j)$	0	2	5	1	11	13	9	14	18	19
$pre(j)$	0	1	1	1	3	4	3	5	6	8

- $dp(5) = \min(2+12, 5+6, 1+13) = 11$，从城市 3 来。
- $dp(8) = \min(11+3, 13+6) = 14$，从城市 5 来。
- $dp(10) = \min(14+5, 18+2) = 19$，从城市 8 来。
- 沿 *pre* 回溯：$10 \to 8 \to 5 \to 3 \to 1$，输出 `1 3 5 8 10`。

要点：输出格式是 `minlong=19`，第二行是路径编号，不要输出费用矩阵。

例 9.5 | 完整程序（上）：读入矩阵

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

int main() {
    int n;
    cin >> n;
    vector<vector<int>> a(n + 1, vector<int>(n + 1, 0));
    for (int i = 1; i <= n; ++i) {
        for (int j = 1; j <= n; ++j) {
            cin >> a[i][j];          // 0 表示没有边
        }
    }
```

要点：矩阵中的 0 表示“没有边”，不是“费用为 0 的边”。

例 9.5 | 完整程序（中）：按编号递推

```
const int INF = 1e9;
vector<int> dp(n + 1, INF);
vector<int> pre(n + 1, 0);
dp[1] = 0;                          // 边界：起点
for (int j = 2; j <= n; ++j) {
    for (int i = 1; i < j; ++i) {
        if (a[i][j] > 0 && dp[i] + a[i][j] < dp[j]) {
            dp[j] = dp[i] + a[i][j];
            pre[j] = i;             // 记录前驱城市
        }
    }
}
```

要点：边只从小城市指向大城市，所以按 j 从小到大计算即可。

例 9.5 | 完整程序（下）：输出费用与路径

```
    cout << "minlong=" << dp[n] << '\n';
    vector<int> path;
    for (int v = n; v != 0; v = pre[v]) {
        path.push_back(v);
    }
    reverse(path.begin(), path.end());
    for (size_t i = 0; i < path.size(); ++i) {
        if (i > 0) {
            cout << ' ';
        }
        cout << path[i];
    }
    cout << '\n';
    return 0;
}
```

要点：从 N 沿 *pre* 回溯到 1，反转后输出；第一行带前缀 `minlong=`。

例 9.5 | 复杂度与易错点

- 时间 $O(N^2)$（每个 j 扫描所有 $i<j$），空间 $O(N^2)$ 存矩阵，*dp* 与 *pre* 为 $O(N)$。
- 矩阵中的 0 表示“没有边”，不是“费用为 0 的边”；判断时必须写 $a_{i,j}>0$。
- 只能从编号小的城市走向编号大的城市，所以循环顺序是 j 从小到大，内层 $i<j$。
- 回溯时从 N 开始，沿 *pre* 走到 1，再反转输出。
- 若题目不保证连通，$dp(N)$ 可能仍是 *INF*，要另行判断；本题保证有路。

要点：看到“单向、无环、按编号递增”，先想 DAG 上的 DP。

例 9.6 | 挖地雷：带点权的 DAG 最长路

题意

n 个地窖（$n \le 200$），第 i 个地窖有 w_i 个地雷。给出若干条单向边 $x \to y$，且 $x < y$。可以从任意地窖开始，沿边一直走，求能挖到的最多地雷数和一条路线。

- 定义 $dp(i)$：从某个起点走到地窖 i 时，能挖到的最多地雷数（必须包含 i）。
- 起点可以任意，所以 $dp(i)$ 至少是 w_i。

$$dp(i) = w_i + \max\Big(0,\ \max_{j \to i} dp(j)\Big).$$

要点：与例 9.5 的区别：这里求最长路，且每个点都有收益；初始化 $dp(i) = w_i$。

例 9.6 | 样例手推：路线 $3 \to 4 \to 5 \to 6$

i	1	2	3	4	5	6
w_i	5	10	20	5	4	5
$dp(i)$	5	15	20	25	29	34
$pre(i)$	0	1	0	3	4	5

- $dp(4) = 5 + \max(dp(1), dp(2), dp(3)) = 5 + 20 = 25$，从 3 来。
- $dp(5) = 4 + dp(4) = 29$；$dp(6) = 5 + \max(dp(4), dp(5)) = 5 + 29 = 34$。
- 最大值 34 在 6，回溯得到 $3 - 4 - 5 - 6$。

要点：注意输出第一行是路线，第二行才是地雷总数；路线用减号连接。

例 9.6 | 完整程序（上）：读入与建图

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
    int n;
    cin >> n;
    vector<int> w(n + 1), dp(n + 1), pre(n + 1, 0);
    for (int i = 1; i <= n; ++i) {
        cin >> w[i];
    }
    vector<vector<int>> g(n + 1, vector<int>(n + 1, 0));
    int x, y;
    while (cin >> x >> y && (x != 0 || y != 0)) {
        g[x][y] = 1;                    // 有向边 x -> y
    }
    dp = w;                             // 每个点至少挖自己
```

要点：读边以 `0 0` 结束；*dp* 初始化为 *w*，表示从自己开始挖。

例 9.6 | 完整程序（中）：转移

```
for (int i = 1; i <= n; ++i) {
    for (int j = 1; j < i; ++j) {
        if (g[j][i] && dp[j] + w[i] > dp[i]) {
            dp[i] = dp[j] + w[i];
            pre[i] = j;          // 记录前驱地窖
        }
    }
}
int pos = 1;
for (int i = 2; i <= n; ++i) {
    if (dp[i] > dp[pos]) {
        pos = i;                 // 答案可以在任意地窖结束
    }
}
```

要点：边只从小指向大，所以 i 从小到大就能保证 $dp(j)$ 已经算好。

例 9.6 | 完整程序（下）：输出路线与总数

```
    vector<int> path;
    for (int v = pos; v != 0; v = pre[v]) {
        path.push_back(v);
    }
    reverse(path.begin(), path.end());
    for (size_t i = 0; i < path.size(); ++i) {
        if (i > 0) {
            cout << '-';
        }
        cout << path[i];
    }
    cout << '\n' << dp[pos] << '\n';
    return 0;
}
```

要点：第一行是路线（减号连接），第二行才是地雷总数。

例 9.6 | 复杂度与易错点

- 用邻接矩阵存边，时间 $O(n^2)$，空间 $O(n^2)$；$n \le 200$ 完全够用。
- dp 初始化为 w，表示“从 i 自己开始挖”；不要初始化为 0。
- 读边时以 `0 0` 结束，不能只读固定条数。
- 若图中有多个入边，取最大的前驱；若没有入边，$pre(i) = 0$，路线从 i 开始。
- 答案取所有 $dp(i)$ 的最大值，不是 $dp(n)$，因为可以在任意地窖结束。

例 9.7 | 友好城市：把“不交叉”变成“上升”

题意

南北两岸各有 N 个位置不同的城市（$N \le 5000$），每对友好城市用一条直线航道连接。要求任意两条航道不交叉，求最多能批准多少对。

- 把每对城市看成平面上的点 (s_i, n_i)，s_i 是南岸坐标，n_i 是北岸坐标。
- 两条航道交叉，当且仅当两对坐标的大小顺序相反。
- 先把所有点按南岸坐标 s_i 从小到大排序；此时只要北岸坐标 n_i 递增，航道就不交叉。

要点：排序后，问题变成：在北岸坐标序列中求最长严格上升子序列。

例 9.7 | 样例：排序后求 LIS

样例的 7 对城市按南岸坐标排序：

s_i	2	4	9	10	15	17	22
n_i	6	2	8	3	12	17	4

北岸序列 $6, 2, 8, 3, 12, 17, 4$ 的最长严格上升子序列例如

$$2, 3, 12, 17,$$

长度为 4，所以最多批准 4 对。

- 若按北岸排序，则对南岸求 LIS，结果一样；选一个方向即可。
- 坐标互不相同，所以用严格上升 $<$；若有重复坐标，要按题目要求判断能否相等。

要点：答案 4。$N = 5000$ 时 $O(N^2)$ 的 LIS 约 2.5×10^7 次比较，可以接受。

例 9.7 | 完整程序（上）：排序与 LIS

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

int main() {
    int n;
    cin >> n;
    vector<pair<int, int>> c(n);
    for (int i = 0; i < n; ++i) {
        cin >> c[i].first >> c[i].second;
    }
    sort(c.begin(), c.end());          // 按南岸坐标排序
    vector<int> f(n, 1);
    int ans = 1;
```

要点：pair 默认先比第一关键字，再比第二关键字；本题南岸坐标互不相同。

例 9.7 | 完整程序（下）：北岸 LIS

```
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < i; ++j) {
            if (c[j].second < c[i].second) {
                f[i] = max(f[i], f[j] + 1);
            }
        }
        ans = max(ans, f[i]);
    }
    cout << ans << '\n';
    return 0;
}
```

要点：排序后，“航道不交叉”等价于北岸坐标严格上升。

例 9.7 | 复杂度与易错点

- 排序 $O(N\log N)$，LIS 的 $O(N^2)$ 是主要开销；$N=5000$ 时约 1.25×10^7 对比较。
- 必须先排序再求 LIS；直接对原顺序求 LIS 没有意义。
- `pair` 默认按第一关键字、第二关键字排序；本题南岸坐标互不相同，不需要额外处理并列。
- 若把“不交叉”误判成“北岸坐标不下降”，在坐标有重复时会出错；本题坐标各不相同。

拓展

LIS 可用二分优化到 $O(N\log N)$：维护每个长度对应的最小结尾，替换第一个大于等于当前值的位置。

例 9.8 | 合唱队形：一个峰值，两个方向

题意

N 位同学站成一排（$N \le 100$），要请最少的同学出列，使剩下的身高满足“先严格上升、再严格下降”。求最少出列人数。

- 等价于：保留一个最长的“山峰形”子序列，答案为 $N -$ 最长长度。
- 枚举峰值位置 i：左边是以 i 结尾的最长严格上升子序列，右边是以 i 开头的最长严格下降子序列。

$$\text{keep}(i) = inc(i) + dec(i) - 1, \qquad \text{答案} = N - \max_i \text{keep}(i).$$

要点：峰值 i 在两个方向中都被算了一次，所以要减 1。

例 9.8 | 样例手推：峰值在 200

身高：$186, 186, 150, 200, 160, 130, 197, 220$。

i	1	2	3	4	5	6	7	8
$inc(i)$	1	1	1	2	2	1	3	4
$dec(i)$	3	3	2	3	2	1	1	1
$inc + dec - 1$	3	3	2	4	3	1	3	4

- 峰值 4（身高 200）：左边 $186, 200$，右边 $200, 160, 130$，保留 4 人。
- 峰值 8（身高 220）：$inc(8) = 4$，对应上升段 $150, 160, 197, 220$；$dec(8) = 1$，保留 4 人。
- 最多保留 4 人，所以最少出列 $8 - 4 = 4$ 人。

要点：答案 4。两个方向都必须用严格不等号。

例 9.8 | 完整程序（上）：读入

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    int n;
    cin >> n;
    vector<int> t(n + 1);
    vector<int> inc(n + 1, 1), dec(n + 1, 1);
    for (int i = 1; i <= n; ++i) {
        cin >> t[i];
    }
```

要点：$inc(i)$ 表示以 i 结尾的最长严格上升子序列，$dec(i)$ 表示从 i 开始的最长严格下降子序列。

例 9.8 | 完整程序（中）：两个方向

```
for (int i = 2; i <= n; ++i) {   // 从左往右：上升段
    for (int j = 1; j < i; ++j) {
        if (t[j] < t[i]) {
            inc[i] = max(inc[i], inc[j] + 1);
        }
    }
}
for (int i = n - 1; i >= 1; --i) { // 从右往左：下降段
    for (int j = i + 1; j <= n; ++j) {
        if (t[j] < t[i]) {
            dec[i] = max(dec[i], dec[j] + 1);
        }
    }
}
```

要点：两个方向都必须用严格不等号 $<$；上升段从左往右，下降段从右往左。

例 9.8 | 完整程序（下）：答案

```
    int best = 0;
    for (int i = 1; i <= n; ++i) {
        best = max(best, inc[i] + dec[i] - 1);
    }
    cout << n - best << '\n';
    return 0;
}
```

要点：峰值 i 被两个方向各算一次，所以要减 1；答案是 $N - best$。

例 9.8 | 复杂度与易错点

- 两个方向各 $O(N^2)$，总时间 $O(N^2)$，空间 $O(N)$；$N \le 100$ 很小。
- 上升段和下降段都要求严格，判断用 $<$，不能写 $\le$。
- 峰值必须同时属于两段；$inc(i) + dec(i) - 1$ 中的 -1 不能漏。
- 若最长“山峰”就是整段，答案是 0；若 $N = 2$，最多保留 1 或 2 人，注意边界。
- 也可以枚举峰值后分别向左右扩展，但用两个 DP 数组更清晰。

例 9.9 | 最长公共子序列：二维状态

题意

给定两个长度不超过 1000 的字符串 X, Y，求它们的最长公共子序列（LCS）长度。子序列不要求连续。

- 定义 $f(i,j)$：X 的前 i 个字符与 Y 的前 j 个字符的最长公共子序列长度。
- 只看两个串的最后一个字符 X_i 与 Y_j。

$$f(i,j) = \begin{cases} f(i-1,j-1)+1, & X_i = Y_j, \\ \max\big(f(i-1,j),\ f(i,j-1)\big), & X_i \neq Y_j. \end{cases}$$

要点：边界 $f(0,j) = f(i,0) = 0$；答案 $f(n,m)$。

例 9.9 | 把样例填成表

		B	D	C	A	B	A
	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	1	1	1	1	2	2	
C	1	1	2	2	2	2	
B	1	1	2	2	3	3	
D	1	2	2	2	3	3	
A	1	2	2	3	3	4	
B	1	2	2	3	4	4	

$X =$ ABCBDAB，$Y =$ BDCABA，右下角 $f(7,6) = 4$。$X_i = Y_j$ 时从左上角加 1，否则取上方和左方的较大值。

要点：表中每个格子只依赖左、上、左上三个格子。

例 9.9 | 完整程序（上）：读入与边界

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
    string a, b;
    cin >> a >> b;
    int n = a.size(), m = b.size();
    a = " " + a;                        // 下标从 1 开始
    b = " " + b;
    vector<vector<int>> f(n + 1, vector<int>(m + 1, 0));
```

要点：第一行、第一列初始为 0，对应空串的 LCS 长度。

例 9.9 | 完整程序（下）：转移与输出

```
    for (int i = 1; i <= n; ++i) {
        for (int j = 1; j <= m; ++j) {
            if (a[i] == b[j]) {
                f[i][j] = f[i - 1][j - 1] + 1; // 匹配
            } else {
                f[i][j] = max(f[i - 1][j], f[i][j - 1]);
            }
        }
    }
    cout << f[n][m] << '\n';
    return 0;
}
```

要点：答案在右下角；若只要长度，可以把二维数组改成两行滚动。

例 9.9 | 复杂度与易错点

- 状态数 $n \times m$，每个状态 $O(1)$ 转移，时间 $O(nm)$，空间 $O(nm)$。
- 两串长度都不超过 1000，1001×1001 的 `int` 约 4 MB，可以接受。
- 若只需要长度，可以用两行滚动数组，空间降到 $O(m)$。
- 子序列不要求连续；若要求连续，状态定义要改成“以 i, j 结尾”，转移也不同。
- 当 $X_i = Y_j$ 时直接取左上角加 1，不要再与上方、左方取最大；否则会重复计算。

例 9.10 | 机器分配：分组决策

题意

有 N 个分公司（$N \le 10$）和 M 台设备（$M \le 15$）。第 i 个公司分到 j 台设备可获利润 $a_{i,j}$。每家公司可分 0 到 M 台，总台数不超过 M。求最大利润及分配方案。

- 定义 $f(i,j)$：前 i 个公司共分到 j 台设备时的最大利润。
- 枚举第 i 个公司分到 k 台，剩下的 $j-k$ 台分给前 $i-1$ 个公司。

$$f(i,j) = \max_{0 \le k \le j} \big(f(i-1, j-k) + a_{i,k} \big).$$

要点：边界 $f(0,j) = 0$；答案 $f(N,M)$。记录每个状态的 k 以便回溯。

例 9.10 | 样例手推：三家公司分三台

利润矩阵：

	0	1	2	3
1	0	30	40	50
2	0	20	30	50
3	0	20	25	30

$f(i,j)$ 表：

$f(i,j)$	$j=0$	1	2	3
$i=0$	0	0	0	0
$i=1$	0	30	40	50
$i=2$	0	30	50	60
$i=3$	0	30	50	70

$f(2,2)=f(1,1)+20=50$；$f(3,3)=f(2,2)+20=70$，对应分配 $1,1,1$。

要点：最大利润 70。本题最优分配可能不唯一，输出任意一组即可；下面的程序在并列时取较大的 k。

例 9.10 | 完整程序（上）：读入与初始化

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

int main() {
    int n, m;
    cin >> n >> m;
    vector<vector<int>> a(n + 1, vector<int>(m + 1, 0));
    for (int i = 1; i <= n; ++i) {
        for (int j = 1; j <= m; ++j) {
            cin >> a[i][j];          // 第 i 公司分 j 台的利润
        }
    }
    vector<vector<int>> f(n + 1, vector<int>(m + 1, -1e9));
    vector<vector<int>> ch(n + 1, vector<int>(m + 1, 0));
```

要点：$a_{i,0} = 0$ 由数组初始化保证，表示“不分设备”。

例 9.10 | 完整程序（中）：分组决策

```
for (int j = 0; j <= m; ++j) {
    f[0][j] = 0;                    // 边界：没有公司
}
for (int i = 1; i <= n; ++i) {
    for (int j = 0; j <= m; ++j) {
        for (int k = 0; k <= j; ++k) {
            int val = f[i - 1][j - k] + a[i][k];
            if (val >= f[i][j]) {
                f[i][j] = val;
                ch[i][j] = k;       // 记录分配台数
            }
        }
    }
}
```

要点：枚举第 i 家公司分到 k 台，剩下的 $j-k$ 台分给前 $i-1$ 家。

例 9.10 | 完整程序（下）：回溯输出

```
    cout << f[n][m] << '\n';
    vector<int> give(n + 1);
    int rest = m;
    for (int i = n; i >= 1; --i) {
        give[i] = ch[i][rest];
        rest -= give[i];
    }
    for (int i = 1; i <= n; ++i) {
        cout << i << ' ' << give[i] << '\n';
    }
    return 0;
}
```

要点：回溯从最后一个公司往前推：`ch[i][rest]` 是第 i 家分到的台数。

例 9.10 | 复杂度与易错点

- 状态数 $N(M+1)$，每个状态枚举 $0 \le k \le j$，时间 $O(NM^2)$，空间 $O(NM)$。
- $N \le 10$、$M \le 15$，计算量很小；即使 M 增大到几百也能承受。
- $a_{i,0}$ 必须为 0，表示不分配设备；输入矩阵没有第 0 列，代码中已初始化为 0。
- 回溯时 *rest* 要同步减去当前公司拿走的台数，否则下标会错位。
- 若题目要求“每个公司至少一台”，要把 k 的下界改为 1，并检查 $j \ge i$。

背包问题：容量就是状态的一维

有 n 件物品，每件有重量 w_i 和价值 c_i，背包容量为 M。选一些物品装入背包，使总重量不超过 M，总价值最大。

- 定义 $f(i,j)$：只考虑前 i 件物品、容量为 j 时的最大价值。
- 对第 i 件物品，只有“不选”和“选”两种决策。

$$f(i,j)=\max\big(f(i-1,j),\ f(i-1,j-w_i)+c_i\big).$$

	0	1	2	3	4	5
0 件	0	0	0	0	0	0
$(2,1)$	0	0	1	1	1	1
$(3,3)$	0	0	1	3	4	4
$(4,5)$	0	0	1	3	5	5

右下角

$f(3,5)=\max(f(2,5),f(2,1)+5)=\max(4,5)=5$。

要点：先写二维表，理解“选/不选”；再用一维滚动数组省空间。

为什么 0/1 背包的一维循环必须倒序

- 一维写法：`f[j] = max(f[j], f[j-w] + c)`。
- 若 j 从小到大：计算 $f(j)$ 时，$f(j-w)$ 可能已经在本轮被当前物品更新过，相当于同一件物品被选了多次。
- 若 j 从大到小：$f(j-w)$ 还没被本轮更新，仍然是“前 $i-1$ 件物品”的答案。

一句话

0/1 背包倒序，表示“每件最多一次”；完全背包正序，表示“每件可以无限次”。循环方向就是物品数量的含义。

背包家族的五个常见变体

1. **0/1** 背包：每件最多选一次，容量维倒序。
2. 完全背包：每件可选无限次，容量维正序。
3. 多重背包：第 i 件最多选 s_i 次，可以拆成 s_i 件 0/1 物品，或用二进制拆分。
4. 二维费用背包：有容量、重量两个限制，状态变成 $f(j,k)$。
5. 分组背包：每组最多选一件，先枚举组，再倒序枚举容量，最后枚举组内物品。

要点：识别题目属于哪一类，比背代码更重要。

例 9.11 | 01 背包：状态与转移

题意

背包容量 $M \le 200$，物品数 $N \le 30$。第 i 件物品重 w_i、价值 c_i，每件最多选一次。求最大总价值。样例答案 12。

- 定义 $f(j)$：容量为 j 时能获得的最大价值。
- 初始化 $f(j) = 0$，表示什么也不装。
- 对每件物品，容量 j 从 M 倒序枚举到 w_i。

$$f(j) \leftarrow \max\big(f(j),\ f(j - w_i) + c_i\big).$$

要点：答案 $f(M)$；倒序是 $0/1$ 背包的标志。

例 9.11 | 背包表逐行填表

j	0	1	2	3	4	5
0 件	0	0	0	0	0	0
$(2,1)$	0	0	1	1	1	1
$(3,3)$	0	0	1	3	3	4

前 2 件填完

加入第 3 件

j	0	1	2	3	4	5
0 件	0	0	0	0	0	0
$(2,1)$	0	0	1	1	1	1
$(3,3)$	0	0	1	3	3	4
$(4,5)$	0	0	1	3	5	5

第 3 件只改容量 $4,5$

每一行只依赖上一行：$f(i,j) = \max(f(i-1,j), f(i-1,j-w_i)+c_i)$。填完第 i 行，第 $i-1$ 行就可以丢掉，这就是滚动数组的伏笔。

要点：先画表再写代码：横轴是容量，纵轴是物品件数，每个格子只由上一行决定。

例 9.11 | 完整程序（上）：状态与读入

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
    int M, N;
    cin >> M >> N;
    vector<int> f(M + 1, 0);          // f[j]: 容量 j 的最大价值
    for (int i = 1; i <= N; ++i) {
        int w, c;
        cin >> w >> c;
```

要点：$f(j)$ 表示容量不超过 j 时的最大价值；初值 0 表示什么也不装。

例 9.11 | 完整程序（下）：倒序更新

```
        for (int j = M; j >= w; --j) { // 倒序：每件最多一次
            f[j] = max(f[j], f[j - w] + c);
        }
    }
    cout << f[M] << '\n';
    return 0;
}
```

要点：内层循环从 M 递减到 w，不能写成 $j = w$ 到 M。

例 9.11 | 样例手推与复杂度

物品：$(2,1),(3,3),(4,5),(7,9)$，容量 10。

- 第 $1,2,3$ 件：重量 $2+3+4=9$，价值 $1+3+5=9$。
- 第 $2,4$ 件：重量 $3+7=10$，价值 $3+9=12$。
- 第 $1,3,4$ 件：重量 $2+4+7=13>10$，不可行。
- 第 $3,4$ 件：重量 $4+7=11>10$，不可行。

最优是第 2、4 件，价值 12。逐件填表时，每件物品只更新“容量足够”的格子，最后 $f(10)=12$。

要点：01 背包是“每件最多一次”，所以第 4 件只能在容量 ≥ 7 的格子里被选中。

例 9.11 | 复杂度与易错点

- 时间 $O(NM)$，空间 $O(M)$；$N \le 30$、$M \le 200$ 时非常小。
- 内层循环条件 $j \ge w$，避免下标为负。
- 若题目要求“恰好装满”，初始化 $f(0) = 0$、其余为 $-\infty$；本题是“不超过容量”，全部初始化为 0。
- 价值与重量都可能是 0，读入时不要用“重量为 0”作为结束标志。

例 9.12 | 完全背包：每件可以反复选

题意

背包容量 $M \le 200$，物品种数 $N \le 30$。每种物品有重量 w_i、价值 c_i，每种可以选无限件。求最大总价值。样例输出 `max=12`。

- 状态仍然是 $f(j)$：容量为 j 时的最大价值。
- 与 $0/1$ 背包唯一的区别：容量 j 从小到大枚举。
- 正序时 $f(j-w_i)$ 可能已经包含当前物品，正好允许继续选。

$$f(j) \leftarrow \max\big(f(j),\ f(j-w_i)+c_i\big), \qquad j = w_i, w_i+1, \ldots, M.$$

要点：正序是完全背包的标志；输出带前缀 `max=`。

例 9.12 | 完整程序（上）：状态与读入

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
    int M, N;
    cin >> M >> N;
    vector<int> f(M + 1, 0);          // f[j]: 容量 j 的最大价值
    for (int i = 1; i <= N; ++i) {
        int w, c;
        cin >> w >> c;
```

要点：状态与 0/1 背包完全相同，唯一的区别在容量循环的方向。

例 9.12 | 完整程序（下）：正序更新

```
        for (int j = w; j <= M; ++j) { // 正序：每件可以无限次
            f[j] = max(f[j], f[j - w] + c);
        }
    }
    cout << "max=" << f[M] << '\n';
    return 0;
}
```

要点：把上一题的 `for (j=M; j>=w; --j)` 改成 `for (j=w; j<=M; ++j)` 即可。

例 9.12 | 样例手推与易错点

物品：$(2,1),(3,3),(4,5),(7,9)$，容量 10。

- 选 5 件第 1 种：重量 10，价值 5。
- 选 2 件第 3 种：重量 8，价值 10。
- 选 1 件第 4 种加 1 件第 2 种：重量 10，价值 12。
- 最优值 12，与 $0/1$ 背包样例相同，但可选方案更多。

易错

完全背包不能用倒序；倒序会退化成 $0/1$ 背包。若把循环方向写反，样例可能仍然过，但大数据会错。

例 9.13 | 庆功会：有限件的多重背包

题意

有 n 种奖品（$n \le 500$），总金额 $m \le 6000$。第 i 种奖品价格 v_i、价值 w_i、最多买 s_i 件（$s_i \le 10$）。求最大价值。

- 这是多重背包：每种物品有数量上限 s_i。
- 最简单写法：在 0/1 背包的容量循环内，再枚举买 k 件（$1 \le k \le s_i$）。
- 因为容量倒序，$f(j - kv_i)$ 仍然来自“前 $i-1$ 种物品”，不会重复使用当前种。

$$f(j) \leftarrow \max_{1 \le k \le s_i,\ kv_i \le j} \big(f(j - kv_i) + kw_i\big).$$

要点：时间 $O(nms)$，最大约 $500 \times 6000 \times 10 = 3 \times 10^7$，可以接受。

例 9.13 | 完整程序（上）：读入与容量循环

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

int main() {
    int n, m;
    cin >> n >> m;
    vector<int> f(m + 1, 0);
    for (int i = 0; i < n; ++i) {
        int v, w, s;
        cin >> v >> w >> s;
        for (int j = m; j >= v; --j) { // 倒序枚举容量
```

要点：每种奖品有数量上限 s，在 0/1 背包的容量循环内再枚举件数。

例 9.13 | 完整程序（下）：枚举件数

```
            for (int k = 1; k <= s && k * v <= j; ++k) {
                f[j] = max(f[j], f[j - k * v] + k * w);
            }
        }
    }
    cout << f[m] << '\n';
    return 0;
}
```

要点：容量倒序保证 $f[j-kv]$ 仍来自“前 $i-1$ 种”，不会重复使用当前种。

例 9.13 | 样例与复杂度

样例中 5 种奖品，金额 1000，最优价值 1040。

- 每种奖品数量都不超过 10，直接枚举件数即可。
- 时间 $O(nms)$，空间 $O(m)$；$n = 500$、$m = 6000$、$s = 10$ 时约 3×10^7 次更新。
- 若 s_i 很大（例如 10^9），要把 s_i 二进制拆成 $1, 2, 4, \ldots$ 个 $0/1$ 物品，复杂度降到 $O(nm \log s)$。
- 若某种奖品价格为 0，要特殊处理，否则内层循环会死循环或下标不变。

要点：本题 $s_i \le 10$，不必二进制拆分；但要知道这个优化。

例 9.14 | 混合背包：三种类型放在一起

题意

背包容量 $M \le 200$，物品数 $N \le 30$。每件物品给出 w_i, c_i, p_i：$p_i = 0$ 表示可以选无限件，$p_i > 0$ 表示最多选 p_i 件。求最大价值。

- 读入时按 p_i 分类，不需要把物品真的分成三个数组。
- $p_i = 0$：完全背包，容量正序。
- $p_i > 0$：把 p_i 件当成 p_i 件 $0/1$ 物品，容量倒序重复 p_i 次。

要点：同一个 f 数组可以依次处理三类物品，因为背包问题满足“阶段可叠加”。

例 9.14 | 完整程序（上）：分类处理

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

int main() {
    int m, n;
    cin >> m >> n;
    vector<int> f(m + 1, 0);
    for (int i = 0; i < n; ++i) {
        int w, c, p;
        cin >> w >> c >> p;
        if (p == 0) {                    // 完全背包
            for (int j = w; j <= m; ++j) {
                f[j] = max(f[j], f[j - w] + c);
            }
```

要点：$p_i = 0$ 表示无限件，容量正序；同一个 f 数组可以依次处理各类物品。

例 9.14 | 完整程序（下）：多重背包

```
        } else {                        // 多重背包
            for (int k = 1; k <= p; ++k) {
                for (int j = m; j >= w; --j) {
                    f[j] = max(f[j], f[j - w] + c);
                }
            }
        }
    }
    cout << f[m] << '\n';
    return 0;
}
```

要点：$p_i > 0$ 时把 p_i 件当成 p_i 件 $0/1$ 物品，容量倒序重复 p_i 次。

例 9.14 | 样例与易错点

样例：容量 10，物品 $(2,1,0)$、$(3,3,1)$、$(4,5,4)$。

- 第 1 种无限件：可以选 1 件，价值 1。
- 第 3 种最多 4 件：选 2 件，重量 8，价值 10。
- 合计重量 10，价值 11，与样例输出一致。

易错

$p=1$ 时既是 $0/1$ 又是多重背包，按 $0/1$ 处理即可；不要因为 $p>0$ 就一律用完全背包。容量循环的方向必须按类型分别写。

例 9.15 | 潜水员：二维费用的“至少”背包

题意

潜水员需要至少 m 升氧和 n 升氮（$m \le 21$，$n \le 79$）。有 k 个气缸（$k \le 1000$），第 i 个提供 a_i 升氧、b_i 升氮，重量为 c_i。每个气缸只能用一次。求满足需求的最小总重量。

- 定义 $f(i,j)$：至少获得 i 升氧、j 升氮时的最小重量。
- 对每个气缸，可以选或不选；选的时候两种气体同时增加。
- 超过需求的部分没有意义，统一“截断”到 m 和 n。

$$f\big(\min(m, i+a), \min(n, j+b)\big) \leftarrow \min\big(\cdots,\ f(i,j)+c\big).$$

要点：边界 $f(0,0)=0$，其余为 $+\infty$；答案 $f(m,n)$。

例 9.15 | 样例手推：249 是怎么来的

需求 5 升氧、60 升氮。五个气缸：

i	1	2	3	4	5
	3	10	5	1	4
	36	25	50	45	20
	120	129	250	130	119

- 气缸 $1+2$：氧 $13 \geq 5$，氮 $61 \geq 60$，重量 $120+129=249$。
- 气缸 $4+5$：氧 $1+4=5$，氮 $45+20=65 \geq 60$，重量 $130+119=249$。
- 气缸 $1+5$：氮 $36+20=56<60$，不够；气缸 $2+4$：重量 259。

要点：最优值 249。注意是“至少”，不是“恰好”；氮可以超过 60。

例 9.15 | 完整程序（上）：初始化与读入

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

int main() {
    int m, n;
    cin >> m >> n;
    int k;
    cin >> k;
    const int INF = 1e9;
    vector<vector<int>> f(m + 1, vector<int>(n + 1, INF));
    f[0][0] = 0;                        // 边界：什么都不选
    for (int t = 0; t < k; ++t) {
        int a, b, c;
        cin >> a >> b >> c;
```

要点：初始化 $f(0,0)=0$，其余为 $+\infty$；答案是 $f(m,n)$，不是 $f(0,0)$。

例 9.15 | 完整程序（中）：倒序更新

```
        for (int i = m; i >= 0; --i) {
            for (int j = n; j >= 0; --j) {
                if (f[i][j] == INF) {
                    continue;        // 不可达，跳过
                }
                int ni = min(m, i + a);
                int nj = min(n, j + b);
                f[ni][nj] = min(f[ni][nj], f[i][j] + c);
            }
        }
    }
```

要点：两层容量循环都倒序，保证每个气缸最多用一次；超出需求的部分截断到 m, n。

例 9.15 | 完整程序（下）：输出

```
    cout << f[m][n] << '\n';
    return 0;
}
```

要点：输出最小总重量；若 $f(m,n)$ 仍为 INF，说明无解（本题保证有解）。

例 9.15 | 复杂度与易错点

- 状态数 $(m+1)(n+1)$，每个气缸更新一遍，时间 $O(kmn)$，空间 $O(mn)$。
- 两层容量循环都必须倒序，保证每个气缸最多用一次；只倒序一层会重复使用气缸。
- 超过需求的氧、氮要截断到 m, n，否则数组越界，而且没有额外收益。
- 初始化 $f(0,0)=0$，其他为 *INF*；不要把所有 f 初始化为 0，否则“至少”会变成“不选也满足”。
- 输出最小重量，不是气缸个数；c_i 可能很大，用 `int` 足够（总和不超过 8×10^5）。

例 9.16 | 分组背包：每组最多选一件

题意

背包容量 $V \le 200$，物品数 $N \le 30$，物品分成 $T \le 10$ 组。每组内的物品互相冲突，最多选一件。求最大价值。

- 状态仍是 $f(j)$：容量为 j 时的最大价值。
- 外层枚举组 t，中层倒序枚举容量 j，内层枚举组内物品。
- 倒序保证同一组、同一件物品不会被重复选；组内枚举保证每组最多选一件。

$$f(j) \leftarrow \max\big(f(j),\ f(j-w)+c\big), \qquad (w,c) \in \text{组} t.$$

要点：三层循环的顺序：组 $\rightarrow$ 容量（倒序）$\rightarrow$ 组内物品。

例 9.16 | 完整程序（上）：分组

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

int main() {
    int V, N, T;
    cin >> V >> N >> T;
    vector<vector<pair<int, int>>> g(T + 1);
    for (int i = 0; i < N; ++i) {
        int w, c, p;
        cin >> w >> c >> p;
        g[p].push_back({w, c});       // 放入第 p 组
    }
    vector<int> f(V + 1, 0);
```

要点：用邻接表按组存物品，避免为每组开固定大小的数组。

例 9.16 | 完整程序（下）：组内选一件

```
    for (int t = 1; t <= T; ++t) {
        for (int j = V; j >= 0; --j) { // 容量倒序
            for (auto [w, c] : g[t]) {
                if (j >= w) {
                    f[j] = max(f[j], f[j - w] + c);
                }
            }
        }
    }
    cout << f[V] << '\n';
    return 0;
}
```

要点：三层循环顺序：组 → 容量（倒序）→ 组内物品；容量循环必须在组内物品外层。

例 9.16 | 样例与易错点

样例分组：$\{(2,1),(3,3)\}$、$\{(4,8),(6,9)\}$、$\{(2,8),(3,9)\}$，容量 10。

- 第 1 组选 $(3,3)$，第 2 组选 $(4,8)$，第 3 组选 $(3,9)$：重量 10，价值 20。
- 若第 1 组选 $(2,1)$，总重量 9，价值 18，不是最优。

易错

容量循环必须在组内物品循环的外面，并且倒序；如果写成“先枚举所有物品，再枚举组”，同一组可能选出两件。

例 9.17 | 货币系统：完全背包计数

题意

有 n 种面值（$n \le 20$），每种面值可以使用无限次。求组成面值 m（$m \le 4000$）的方案数。方案不考虑顺序。

- 定义 $f(j)$：组成面值 j 的方案数。
- 不选任何货币也算一种方案，所以 $f(0) = 1$。
- 对每种面值 v，容量 j 从 v 正序枚举到 m：$f(j) \mathrel{+}= f(j-v)$。

$$f(0) = 1, \qquad f(j) \leftarrow f(j) + f(j-v) \quad (j = v, v+1, \ldots, m).$$

要点：外层枚举面值、内层正序，保证同一种面值可以反复使用，且不会把不同顺序重复计数。

例 9.17 | 完整程序（上）：边界与面值循环

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

int main() {
    int n, m;
    cin >> n >> m;
    vector<long long> f(m + 1, 0);
    f[0] = 1;                          // 边界：凑 0 有一种方案
    for (int i = 0; i < n; ++i) {
        int v;
        cin >> v;
        for (int j = v; j <= m; ++j) { // 正序：可用无限次
```

要点：外层枚举面值、内层正序，保证同一种面值可反复使用，且不会重复计数。

例 9.17 | 完整程序（下）：累加方案数

```
            f[j] += f[j - v];
        }
    }
    cout << f[m] << '\n';
    return 0;
}
```

要点：外层枚举面值、内层正序；方案数可能超过 `long long`，更大范围需 `__int128`。

例 9.17 | 样例与易错点

面值 $1, 2, 5$，组成 10 的方案共 10 种：

$$5+5,\quad 5+2+2+1,\quad 5+2+1+1+1,\quad 5+1+\cdots+1,$$
$$2+2+2+2+2,\quad 2+2+2+2+1+1,\quad 2+2+2+1+1+1+1,$$
$$2+2+1+\cdots+1,\quad 2+1+\cdots+1,\quad 1+\cdots+1.$$

- 若把面值循环写在内层、金额写在外层，会把 $1+2$ 和 $2+1$ 算成两种，方案数偏大。
- 方案数增长很快；本题极限数据可能超过 `long long`，正式提交应使用 `__int128` 或高精度。
- 若 $f(0)=0$，所有方案数都会变成 0。
- 正序枚举金额表示“同一种面值可以继续用”，这是完全背包计数的标志。

区间 DP：先算短区间，再算长区间

区间 DP 的状态写成 $f(i,j)$，表示区间 $[i,j]$ 上的答案。大区间由两个更短的区间合并而来，所以必须按区间长度从小到大计算。

$$f(i,j) = \min_{i \le k < j} \big(f(i,k) + f(k+1,j) \big) + w(i,j).$$

- 先枚举长度 $len = 2, 3, \ldots, n$。
- 再枚举左端点 i，右端点 $j = i + len - 1$。
- 最后枚举分界点 k。

	1	2	3	4
1	0	5	10	19
2		0	3	9
3			0	5
4				0

$a = 4, 1, 2, 3$。

$f(1,4) = \min(0+9,\ 5+5,\ 10+0) + 10 = 19$。

要点：填表顺序：长度 $1 \to n$；每个大区间只依赖比它短的区间。

例 9.18 | 合并石子：最后一次合并

题意

一排 N 堆石子（$N \le 100$），每次只能合并相邻的两堆，得分为合并后新堆的石子数。求合并成一堆的最小总得分。样例答案 239。

- 定义 $f(i,j)$：把第 i 到第 j 堆合并成一堆的最小得分。
- 最后一次合并一定把 $[i,j]$ 分成 $[i,k]$ 和 $[k+1,j]$ 两堆。
- 这次合并的得分就是区间 $[i,j]$ 的石子总数。

$$f(i,j) = \min_{i \le k < j} \big(f(i,k) + f(k+1,j)\big) + \sum_{t=i}^{j} a_t.$$

要点：边界 $f(i,i) = 0$；答案 $f(1,N)$。

例 9.18 | 样例与填表顺序

样例 7 堆：$13, 7, 8, 16, 21, 4, 18$，答案是 239。

- 长度为 2 的区间：$f(1,2) = 20$，$f(2,3) = 15$，$f(3,4) = 24$，$f(4,5) = 37$，$f(5,6) = 25$，$f(6,7) = 22$。
- 长度为 3 的区间，例如 $f(1,3) = \min(0 + 15, 20 + 0) + 28 = 43$。
- 长度继续增大，最后 $f(1,7) = 239$。

为什么不能按左端点直接填

$f(1,7)$ 需要 $f(1,k)$ 和 $f(k+1,7)$，其中 $f(k+1,7)$ 的左端点大于 1。如果只按 i 从小到大循环，这些区间还没算。按长度循环可以保证所有短区间先算完。

例 9.18 | 完整程序（上）：前缀和与区间长度

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

int main() {
    int n;
    cin >> n;
    vector<int> a(n + 1), s(n + 1, 0);
    for (int i = 1; i <= n; ++i) {
        cin >> a[i];
        s[i] = s[i - 1] + a[i];       // 前缀和
    }
    vector<vector<long long>> f(n + 2, vector<long long>(n + 2, 0));
    for (int len = 2; len <= n; ++len) { // 区间长度递增
        for (int i = 1; i + len - 1 <= n; ++i) {
            int j = i + len - 1;
```

要点：按长度从小到大填表，保证所有更短的区间已经算好。

例 9.18 | 完整程序（下）：枚举分界点

```
            f[i][j] = LLONG_MAX;
            for (int k = i; k < j; ++k) {
                long long val = f[i][k] + f[k + 1][j];
                val += s[j] - s[i - 1];
                f[i][j] = min(f[i][j], val);
            }
        }
    }
    cout << f[1][n] << '\n';
    return 0;
}
```

要点：最后一次合并的得分是区间总和，用前缀和 $O(1)$ 求出。

例 9.18 | 复杂度、变式与易错点

- 状态数 $O(N^2)$，每个区间枚举 $O(N)$ 个分界点，时间 $O(N^3)$，空间 $O(N^2)$。
- $N \le 100$ 时约 10^6 次转移，完全够用；若 N 到 1000，需要四边形不等式优化。
- 前缀和 s 让区间和 $O(1)$ 求出，不要在内层再写循环求和。
- 若石子围成一圈（环形合并），把数组复制一倍，答案为 $\min_i f(i, i+n-1)$。
- 初始化 $f(i,i)=0$；求最小值时其余格子先设为 $+\infty$。

例 9.19 | 乘积最大：把数字串切成 $K+1$ 段

题意

一个长度为 N 的数字串（$6 \le N \le 10$），插入 K 个乘号（$1 \le K \le 6$），把它分成 $K+1$ 段，使各段乘积最大。样例 1231、$K=2$，最优 $31 \times 2 = 62$。

- 定义 $f(i,t)$：前 i 个数字中插入 t 个乘号能得到的最大乘积。
- 最后一段是第 $j+1$ 到第 i 个数字，前面 j 个数字插入 $t-1$ 个乘号。

$$f(i,t) = \max_{t \le j < i} \big(f(j, t-1) \times \mathrm{num}(j+1, i) \big).$$

要点：边界 $f(i,0) = \mathrm{num}(1,i)$；答案 $f(N,K)$。

例 9.19 | 样例手推（上）：不插与插一个乘号

先算不插乘号：

$$f(1,0)=1,\ f(2,0)=12,\ f(3,0)=123,\ f(4,0)=1231.$$

插一个乘号：

$$\begin{aligned}
f(2,1) &= 1\times 2 = 2,\\
f(3,1) &= \max(1\times 23,\ 12\times 3) = 36,\\
f(4,1) &= \max(1\times 231,\ 12\times 31,\ 123\times 1) = 372.
\end{aligned}$$

要点：$f(i,1)$ 枚举“最后一个乘号”放在第 j 个数字后面。

例 9.19 | 样例手推（下）：插两个乘号

插两个乘号：

$$
\begin{aligned}
f(3,2) &= f(2,1) \times 3 = 6, \\
f(4,2) &= \max\big(f(2,1) \times 31,\ f(3,1) \times 1\big) \\
&= \max(62, 36) = 62.
\end{aligned}
$$

- $62 = 1 \times 2 \times 31$：先在前 2 个数字中插一个乘号，再在最后一段前插第二个。
- 若把两个乘号都放在末尾，会得到 123×1 一类的方案，乘积更小。

要点：答案 62，对应 $1 \times 2 \times 31$：在第 $1, 2$ 个数字后各切一刀。

例 9.19 | 完整程序（上）：子串转整数

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
    int n, k;
    string s;
    cin >> n >> k >> s;
    s = " " + s;                          // 下标从 1 开始
    auto num = [&](int l, int r) {
        return stoll(s.substr(l, r - l + 1));
    };
```

要点：stoll 把子串转成 long long；$num(l, r)$ 表示第 l 到第 r 个数字组成的整数。

例 9.19 | 完整程序（中）：填表

```
vector<vector<long long>> f(n + 1, vector<long long>(k + 1, 0));
for (int i = 1; i <= n; ++i) {
    f[i][0] = num(1, i);          // 边界：不插乘号
}
for (int t = 1; t <= k; ++t) {
    for (int i = t + 1; i <= n; ++i) {
        for (int j = t; j < i; ++j) {
            long long v = f[j][t - 1] * num(j + 1, i);
            f[i][t] = max(f[i][t], v);
        }
    }
}
```

要点：t 个乘号至少需要 $t+1$ 个数字，所以 i 从 $t+1$ 开始，j 从 t 开始。

例 9.19 | 完整程序（下）：输出

```
    cout << f[n][k] << '\n';
    return 0;
}
```

要点：答案 $f(N, K)$：前 N 个数字插入 K 个乘号的最大乘积。

例 9.19 | 复杂度与易错点

- 状态数 NK，每个状态枚举 $O(N)$ 个切点，时间 $O(KN^2)$，空间 $O(NK)$。
- $N \le 10$、$K \le 6$，最大乘积不超过 $9^{10} \approx 3.5 \times 10^9$，超过 `int`，必须用 `long long`。
- 切点 j 必须满足 $j \ge t$（前 j 个数字至少能放 $t-1$ 个乘号），否则状态不合法。
- 数字串中可能有 0，乘积会变成 0；初始化 $f(i,0)$ 时直接转成整数即可。
- 输出的是最大乘积，不是方案；若要输出方案，需要额外记录每个状态的切点。

二维表 DP：两个下标各表示什么

- 二维表 DP 的状态写成 $f(i,j)$，两个下标通常表示“处理到第一个序列的第 i 位、第二个序列的第 j 位”，或者“第 i 行第 j 列”。
- 转移一般只依赖左边、上边、左上三个格子，所以按行从上到下、每行从左到右填表即可。
- 编辑距离、方格取数、复制书稿、橱窗布置都属于这一类；它们的共同点是“两个维度的进度”。

通用检查

写代码前先画一张 5×5 的小表，标出 $f(i,j)$ 由哪些格子得到。若依赖左上、上、左，三重循环的顺序就确定了。

要点：二维 DP 的关键不是代码，而是把“两个下标”的含义说清楚。

例 9.20 | 编辑距离：三种操作对应三个方向

题意

给定字符串 A, B（长度均小于 2000），每次可以删除、插入或修改一个字符。求把 A 变成 B 的最少操作次数。样例答案 4。

- 定义 $f(i,j)$：把 A 的前 i 个字符变成 B 的前 j 个字符的最少操作数。
- 若 $A_i = B_j$，不用操作：$f(i,j) = f(i-1,j-1)$。
- 否则，三种操作分别对应左、上、左上三个格子。

$$f(i,j) = 1 + \min\big(f(i-1,j),\ f(i,j-1),\ f(i-1,j-1)\big).$$

要点：边界 $f(i,0) = i$（删光），$f(0,j) = j$（插入）；答案 $f(n,m)$。

例 9.20 | 样例：删 s、改 q、删 x, b

$A =$ sfdqxbw，$B =$ gfdgw。

	s	f	d	q	x	b	w
	del	f	d	rep	del	del	w
		f	d	g			w

一种操作方案：

1. 删除 s；
2. f, d 保留；
3. 把 q 改成 g；
4. 删除 x、b；
5. w 保留。

共 $1 + 1 + 2 = 4$ 次，与样例输出一致。

例 9.20 | 完整程序（上）：读入

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    string a, b;
    cin >> a >> b;
    int n = a.size(), m = b.size();
    a = " " + a;
    b = " " + b;
    vector<vector<int>> f(n + 1, vector<int>(m + 1, 0));
```

要点：字符串前补空格，让下标从 1 开始；$f(i,j)$ 表示 A 前 i 个变成 B 前 j 个的最少操作数。

例 9.20 | 完整程序（中）：边界

```
for (int i = 0; i <= n; ++i) {
    f[i][0] = i;                    // 边界：删光
}
for (int j = 0; j <= m; ++j) {
    f[0][j] = j;                    // 边界：插入
}
```

要点：边界不能漏：$f(i,0)=i$ 表示删除 i 个字符，$f(0,j)=j$ 表示插入 j 个字符。

例 9.20 | 完整程序（下）：三种操作与输出

```
    for (int i = 1; i <= n; ++i) {
        for (int j = 1; j <= m; ++j) {
            if (a[i] == b[j]) {
                f[i][j] = f[i - 1][j - 1]; // 不用操作
            } else {
                int del = f[i - 1][j];   // 删除
                int ins = f[i][j - 1];   // 插入
                int rep = f[i - 1][j - 1]; // 修改
                f[i][j] = 1 + min({del, ins, rep});
            }
        }
    }
    cout << f[n][m] << '\n';
    return 0;
}
```

要点：删除看上方、插入看左方、修改看左上；三个方向对应三种操作。

例 9.20 | 复杂度与易错点

- 状态数 nm，每个状态 $O(1)$ 转移，时间 $O(nm)$，空间 $O(nm)$。
- 两串长度都小于 2000，2001^2 个 `int` 约 16 MB，接近常见内存限制；可以用两行滚动数组降到 $O(m)$。
- 边界不能漏：$f(i,0)=i$、$f(0,j)=j$，否则删除和插入的代价会算错。
- 三种操作与三个方向一一对应：删除看上方，插入看左方，修改看左上。
- 若 $A_i=B_j$，直接取左上，不要加 1；这是最常见的错误。

例 9.21 | 方格取数：两条路径同时走

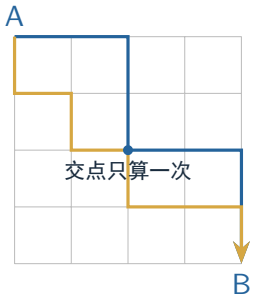
题意

$N \times N$ 的方格（$N \le 10$），某些格子有正整数。从左上角走到右下角，只能向右或向下；走两次，取走格子里的数（取走后变 0）。求两条路径取得的数字和的最大值。样例答案 67。

- 让两个人同时从 $(1,1)$ 出发，每人走一步；总步数 $s = i + j$ 相同。
- 定义 $f(s, i_1, i_2)$：两人都走了 $s-2$ 步，分别在第 i_1、i_2 行时的最大和。
- 列坐标由 $j_1 = s - i_1$、$j_2 = s - i_2$ 算出。

要点：如果两人走到同一格，这个格子的数只能算一次。

例 9.21 | 两条路径与“同一格只算一次”



蓝色和金色是两条路径。它们在中间格相交时，这个格子的数不能加两次。转移时用 `i1 != i2` 判断是否在同一格。

要点：两条路径的步数相同，所以可以用一个步数维度 s 同时表示两个人的进度。

例 9.21 | 完整程序（上）：输入与步数

```
#include <bits/stdc++.h>
using namespace std;
int a[11][11], f[22][11][11];
int main() {
    int n;
    cin >> n;
    int x, y, v;
    while (cin >> x >> y >> v && (x != 0 || y != 0 || v != 0)) {
        a[x][y] = v;
    }
    for (int s = 2; s <= 2 * n; ++s) {
        for (int i1 = 1; i1 <= n; ++i1) {
```

要点：用总步数 $s = i + j$ 同步两条路径；s 从 2 走到 $2N$。

例 9.21 | 完整程序（中）：第二条路径与前驱

```
            int j1 = s - i1;
            if (j1 < 1 || j1 > n) {
                continue;              // 列越界
            }
            for (int i2 = 1; i2 <= n; ++i2) {
                int j2 = s - i2;
                if (j2 < 1 || j2 > n) {
                    continue;          // 列越界
                }
                int gain = a[i1][j1];
                if (i1 != i2) {
                    gain += a[i2][j2]; // 不同格才加两次
                }
```

要点：同一格只算一次：若 $i_1 = i_2$，则 $j_1 = j_2$，只加一次。

例 9.21 | 完整程序（下）：四种前驱与输出

```
                int best = f[s - 1][i1 - 1][i2 - 1];
                best = max(best, f[s - 1][i1 - 1][i2]);
                best = max(best, f[s - 1][i1][i2 - 1]);
                best = max(best, f[s - 1][i1][i2]);
                f[s][i1][i2] = best + gain;
            }
        }
    }
    cout << f[2 * n][n][n] << '\n';
    return 0;
}
```

要点：四个前驱对应“两人分别从上方/左方来”的四种组合。

例 9.21 | 复杂度与易错点

- 状态数 $O(N^3)$（步数 $2N$、两个行号 N^2），每个状态 $O(1)$ 转移，时间 $O(N^3)$。
- $N \le 10$，状态最多约 2×10^3，非常小；空间 $O(N^3)$。
- 同一格只算一次：如果 $i_1 = i_2$，则 $j_1 = j_2$，只加 a_{i_1,j_1}。
- 起点和终点也只算一次；初始化 $f(2,1,1) = a_{1,1}$，最后输出 $f(2N, N, N)$。
- 输入以 `0 0 0` 结束，不是固定 N 行。

例 9.22 | 复制书稿：最小化最大工作量

题意

m 本书（$m \le 500$）分给 k 个人（$k \le m$）抄写，每人抄连续的一段，不能把一本书拆开。复制时间是抄得最多的人所用的时间。求最短时间，并输出一种分配；多解时让前面的人尽量少抄。

- 定义 $f(i,j)$：前 i 本书分给前 j 个人时，最大工作量的最小值。
- 枚举第 j 个人抄第 $p+1$ 到第 i 本。

$$f(i,j) = \min_{j-1 \le p < i} \max\big(f(p,j-1),\ \mathrm{sum}(p+1,i)\big).$$

要点：边界 $f(i,1) = \mathrm{sum}(1,i)$；答案 $f(m,k)$。这是“最小化最大值”的典型 DP。

例 9.22 | 样例：为什么是 17

9 本书页数 $1, 2, \ldots, 9$，分给 3 人。

- 方案 $1-5,\ 6-7,\ 8-9$：工作量 $15, 13, 17$，最大 17。
- 方案 $1-4,\ 5-7,\ 8-9$：工作量 $10, 18, 17$，最大 18。
- 方案 $1-5,\ 6-8,\ 9$：工作量 $15, 21, 9$，最大 21。

最短时间是 17。为了“让前面的人少抄”，从最后一个人往前贪心：在不超过 17 的前提下，尽量把后面的书分给后面的人。

$$\underbrace{1, \ldots, 5}_{15} \mid \underbrace{6, 7}_{13} \mid \underbrace{8, 9}_{17}.$$

要点：先 DP 求最短时间，再从后往前构造方案。

例 9.22 | 完整程序（上）：前缀和与边界

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
    int m, k;
    cin >> m >> k;
    vector<int> a(m + 1), s(m + 1, 0);
    for (int i = 1; i <= m; ++i) {
        cin >> a[i];
        s[i] = s[i - 1] + a[i];
    }
    const long long INF = 1e18;
    vector<vector<long long>> f(m + 1, vector<long long>(k + 1, INF));
    for (int i = 1; i <= m; ++i) {
        f[i][1] = s[i];              // 边界：一个人抄完
    }
```

要点：$f(i,1)$ 表示前 i 本书全由第一个人抄，工作量就是前缀和。

例 9.22 | 完整程序（中）：枚举最后一个人

```
    for (int j = 2; j <= k; ++j) {
        for (int i = j; i <= m; ++i) {
            for (int p = j - 1; p < i; ++p) {
                long long last = (long long)s[i] - s[p];
                long long val = max(f[p][j - 1], last);
                f[i][j] = min(f[i][j], val);
            }
        }
    }
    long long limit = f[m][k];
```

要点：最后一个人抄第 $p+1$ 到第 i 本，工作量是区间和；取两人的较大值。

例 9.22 | 完整程序（下）：从后往前分配

```
vector<int> L(k + 1), R(k + 1);
int end = m;
for (int person = k; person >= 1; --person) {
    int start = end;
    long long sum = 0;
    while (start >= person && sum + a[start] <= limit) {
        sum += a[start];
        --start;
    }
    L[person] = start + 1;
    R[person] = end;
    end = start;                    // 下一个人从这里往前
}
```

要点：从最后一个人往前贪心，在不超过最短时间的前提下尽量多抄。

例 9.22 | 完整程序（续）：输出方案

```
    for (int i = 1; i <= k; ++i) {
        cout << L[i] << ' ' << R[i] << '\n';
    }
    return 0;
}
```

要点：循环条件 `start >= person` 保证前面每个人至少还能分到一本书。

例 9.22 | 复杂度与易错点

- 时间 $O(km^2)$，空间 $O(mk)$；$m \le 500$、$k \le m$ 时约 1.25×10^8，需要较快的实现（本题常见做法是二分加贪心，$O(m \log \sum a_i)$）。
- 若用 DP，注意 $f(p, j-1)$ 中的 $p \ge j-1$，否则前面的人不够分。
- 输出要求“前面的人少抄”，所以构造时必须从最后一个人往前推，不能从第一个人往后推。
- 每个人的区间是闭区间 $[L_i, R_i]$，输出 k 行，编号从小到大。
- 书页总数可能较大，前缀和与 f 用 `long long`。

例 9.23 | 橱窗布置：花束顺序不能变

题意

F 束花（$F \le 100$）按编号 $1 \dots F$ 的顺序放入 V 个花瓶（$F \le V \le 100$），每束花只能放一个花瓶，花瓶编号必须严格递增。第 i 束花放在第 j 个花瓶的美学值为 $A_{i,j}$（可为负）。求最大美学值及方案。

- 定义 $f(i,j)$：前 i 束花放好，且第 i 束花放在第 j 个花瓶时的最大美学值。
- 第 $i-1$ 束花必须放在 j 左边的某个花瓶 $p < j$。

$$f(i,j) = A_{i,j} + \max_{i-1 \le p < j} f(i-1,p).$$

要点：边界 $f(1,j) = A_{1,j}$；答案 $\max_{j \ge F} f(F,j)$。记录 $pre(i,j) = p$ 以便回溯。

例 9.23 | 样例手推：53 的方案

美学矩阵：

	1	2	3	4	5
1	7	23	-5	-24	16
2	5	21	-4	10	23
3	-21	5	-4	-20	20

- 花 1 放瓶 2：23。
- 花 2 放瓶 4：$23 + 10 = 33$。
- 花 3 放瓶 5：$33 + 20 = 53$。

其他方案：花 1 放瓶 1、花 2 放瓶 2、花 3 放瓶 5，美学值 $7 + 21 + 20 = 48$。最优为 53，花瓶编号 $2, 4, 5$。

要点：花瓶可以有空位，但花束之间的相对顺序不能变。

例 9.23 | 完整程序（上）：读入

```
#include <bits/stdc++.h>
using namespace std;
int a[105][105], f[105][105], pre[105][105];
int main() {
    int F, V;
    cin >> F >> V;
    for (int i = 1; i <= F; ++i) {
        for (int j = 1; j <= V; ++j) {
            cin >> a[i][j];
        }
    }
```

要点：美学值可能为负，后面不能用 0 表示“不可达”，要用很小的负数。

例 9.23 | 完整程序（中）：初始化与边界

```
const int NEG = -1e9;
for (int i = 0; i <= F; ++i) {
    for (int j = 0; j <= V; ++j) {
        f[i][j] = NEG;            // 不可达
    }
}
for (int j = 1; j <= V; ++j) {
    f[1][j] = a[1][j];            // 边界：第 1 束花
}
```

要点：$f(1,j) = A_{1,j}$：第 1 束花可以直接放在任意花瓶。

例 9.23 | 完整程序（下）：转移

```
for (int i = 2; i <= F; ++i) {
    for (int j = i; j <= V; ++j) {
        for (int p = i - 1; p < j; ++p) {
            if (f[i - 1][p] + a[i][j] > f[i][j]) {
                f[i][j] = f[i - 1][p] + a[i][j];
                pre[i][j] = p;   // 记录花瓶
            }
        }
    }
}
```

要点：第 i 束花只能放在第 i 到第 $V-(F-i)$ 个花瓶之间，循环下界是 i。

例 9.23 | 完整程序（下）：找最优

```
int best = NEG, pos = 0;
for (int j = F; j <= V; ++j) {
    if (f[F][j] > best) {
        best = f[F][j];
        pos = j;
    }
}
cout << best << '\n';
```

要点：最后一束花可以放在任意 $j \geq F$ 的花瓶，所以在这些位置中取最大。

例 9.23 | 完整程序（续）：回溯输出

```
    vector<int> ans(F + 1);
    for (int i = F; i >= 1; --i) {
        ans[i] = pos;
        pos = pre[i][pos];
    }
    for (int i = 1; i <= F; ++i) {
        if (i > 1) {
            cout << ' ';
        }
        cout << ans[i];
    }
    cout << '\n';
    return 0;
}
```

要点：从第 F 束花开始沿 *pre* 往前找每一束花的花瓶，再反转输出。

例 9.23 | 复杂度与易错点

- 状态数 FV，每个状态枚举 $O(V)$ 个前驱，时间 $O(FV^2)$，空间 $O(FV)$。
- $F, V \leq 100$，约 10^6 次转移，完全够用。
- 美学值可以是负数，不能把 f 初始化为 0；要用很小的负数（如 -10^9）表示“不可达”。
- 第 i 束花只能放在第 i 到第 $V-(F-i)$ 个花瓶之间，循环下界是 i，上界可以到 V。
- 题目要求输出任意最优方案；本程序在并列时保留先找到的方案。

记忆化搜索：递归写转移，数组存答案

- 有些状态图是 DAG，但转移方向不直观（例如网格上向四个方向滑）。这时可以写递归函数，按“从当前状态出发”的方向计算。
- 递归第一次进入某状态时计算，并把结果存进数组；再次进入时直接返回。
- 每个状态只算一次，复杂度等于状态数乘转移数，与递推填表相同。

递归三件事

出口（最小状态）$\rightarrow$ 如果算过就返回 $\rightarrow$ 枚举所有转移，取最优并保存。

要点：记忆化搜索 $=$ 递归 $+$ 备忘录；它把“填表顺序”交给递归调用栈。

例 9.24 | 滑雪：网格上的最长下降路径

题意

给出 $R \times C$ 的高度矩阵（$R, C \le 100$）。从任意点出发，每次可以走到上下左右相邻且高度更低的点。求最长滑坡经过的点数。样例答案 25。

- 定义 $f(i,j)$：从 (i,j) 出发能滑过的最多点数（包含起点）。
- 只能走向高度更低的邻居，所以路径不会成环，状态图是 DAG。
- 如果四个邻居都不更低，$f(i,j) = 1$。

$$f(i,j) = 1 + \max_{\text{邻居}(x,y),\ h_{x,y} < h_{i,j}} f(x,y).$$

要点：答案是所有 $f(i,j)$ 的最大值，不是 $f(1,1)$。

例 9.24 | 样例：25 个点的一条滑坡

1	2	3	4	5
16	17	18	19	6
15	24	25	20	7
14	23	22	21	8
13	12	11	10	9

从 25 出发，沿着 $25 \to 24 \to 23 \to \cdots \to 1$ 的螺旋路径，可以经过全部 25 个格子。每一步高度都严格下降，所以这条路径合法。

要点：最长路径不一定从最大值开始，但本题的最大值 25 恰好能绕完整个网格。

例 9.24 | 完整程序（上）：记忆化 DFS

```
#include <bits/stdc++.h>
using namespace std;
int R, C, h[105][105], f[105][105];
int dx[4] = {1, -1, 0, 0};
int dy[4] = {0, 0, 1, -1};
int dfs(int x, int y) {
    if (f[x][y]) {
        return f[x][y];                 // 备忘录
    }
    f[x][y] = 1;                        // 至少包含自己
    for (int d = 0; d < 4; ++d) {
        int nx = x + dx[d], ny = y + dy[d];
        if (nx < 1 || nx > R || ny < 1 || ny > C) {
            continue;                   // 越界
        }
```

要点：`f[x][y]` 非 0 表示已经算过；这个判断就是“备忘录”。

例 9.24 | 完整程序（中）：向更低处滑

```
        if (h[nx][ny] < h[x][y]) {
            f[x][y] = max(f[x][y], 1 + dfs(nx, ny));
        }
    }
    return f[x][y];
}
```

要点：只能走向高度严格更低的邻居，所以路径不会成环，状态图是 DAG。

例 9.24 | 完整程序（下）：主函数

```
int main() {
    cin >> R >> C;
    for (int i = 1; i <= R; ++i) {
        for (int j = 1; j <= C; ++j) {
            cin >> h[i][j];
        }
    }
    int ans = 0;
    for (int i = 1; i <= R; ++i) {
        for (int j = 1; j <= C; ++j) {
            ans = max(ans, dfs(i, j));
        }
    }
    cout << ans << '\n';
}
```

要点：答案取所有起点的最大值；时间 $O(RC)$。

写 DP 前的四要素检查表

状态	每个下标表示什么？是否包含了影响后续决策的全部信息？ 两个不同的历史，只要状态相同，后续最优选择是否一定相同？
转移	最后一步有哪些互不重叠的类别？每个类别由哪个更小状态得到？ 计数题相加，最值题取 max 或 min，是否漏掉某一类？
边界	最小状态能否直接写出？初始化用 0、1 还是 $-\infty$？ “不选”“空串”“单个元素”这些情况是否覆盖？
答案	输出某个 $f(i)$，还是 $\max_i f(i)$？下标范围是否写对？ 是否需要记录前驱来输出方案？

要点：这四行写不出来，说明还没想清楚，不要急着写代码。

初始化：0、1 还是 $-\infty$

- 最大值且允许不选：全部初始化为 0。例如 0/1 背包、LIS 的 $f(i) = 1$。
- 最小值：初始化为 $+\infty$，但 $f(0) = 0$ 或 $f(i, i) = 0$ 等合法边界要单独写。
- 恰好装满/恰好用完：$f(0) = 0$，其余为 $-\infty$；答案取 $f(M)$，不能取 $\max_j f(j)$。
- 计数：$f(0) = 1$，表示“空方案也是一种方案”；其余为 0。
- 二维表：第一行、第一列要按含义初始化，不能一律置 0。

检查方法

把 $n = 1$、容量为 0、字符串为空这些最小情况代入程序，看输出是否符合题意。

循环顺序：方向就是语义

0/1 背包	容量 j 从 M 递减到 w，表示每件最多一次。
完全背包	容量 j 从 w 递增到 M，表示每件可以无限次。
多重背包	容量倒序，再枚举件数 k；或二进制拆分后按 0/1 做。
区间 DP	长度 len 从小到大，再枚举左端点和分界点。
二维表 DP	若依赖上、左、左上，按行从上到下、每行从左到右。
DAG 上的 DP	按拓扑序；若边都从小指向大，就按编号从小到大。
两条路径	按总步数 s 从小到大，保证前一步已经算好。

要点：循环写反时，程序可能仍然跑出样例，但含义已经变了。

数值范围、取模与调试

- `int` 约到 2.1×10^9；乘积、方案数、前缀和常用 `long long`。
- 用 $-\infty$ 初始化时，不要直接做加法，否则会溢出；先判断是否可达。
- 取模题每步取模，减法后加模数再取模，保证结果非负。
- 调试第一步：把 *dp* 表打印出来，和手算的小表逐格对照。
- 调试第二步：写一个 $n \le 10$ 的暴力枚举，随机生成小数据对拍。
- 调试第三步：测试 $n = 1$、全部相等、全为 0、负数、最大规模等边界。

要点：DP 的错误大多不是语法错误，而是状态含义或初始化错了；打印表格最快。

通用模板：从题意到代码的五步

1. 用一句话写出 f 的含义（含下标）。
2. 写出转移式，并检查类别互斥、完整。
3. 写边界和初始化。
4. 确定循环顺序，保证前驱先算。
5. 确定答案并处理输出格式。

```
vector<long long> f(m + 1, 0);          // 1. 状态
for (int i = 1; i <= n; ++i) {          // 2. 阶段
    for (int j = m; j >= w[i]; --j) { // 3. 顺序
        f[j] = max(f[j], f[j - w[i]] + c[i]); // 4. 转移
    }
}
cout << f[m] << '\n';                   // 5. 答案
```

要点：模板只能帮你记结构，状态和转移必须按题目重新设计。

练习地图：27 道题，按模型分三组

线性与序列（9 道）

- 最长上升子序列、最大子矩阵
- 登山、摘花生
- 最大上升子序列和
- 怪盗基德的滑翔翼
- 最低通行费
- 三角形最佳路径问题
- 拦截导弹

背包（8 道）

- 采药、数字组合
- 宠物小精灵之收服
- 买书、Charm Bracelet
- 装箱问题、开餐馆
- 高效工作

区间与二维（10 道）

- 公共子序列
- 计算字符串距离
- 糖果、鸡蛋的硬度
- 大盗阿福、股票买卖
- 鸣人的影分身、数的划分
- Maximum sum
- 最长公共子上升序列

要点：建议顺序：先做第 $1, 3, 4, 8$ 题，再做背包第 $10, 11, 13, 14$ 题，最后做区间与二维题。

本章练习题索引 | 1–7 / 27

1. P5278 | 最长上升子序列（一本通 1281）
2. P5279 | 最大子矩阵（一本通 1282）
3. P5280 | 登山（一本通 1283）
4. P5281 | 摘花生（一本通 1284）
5. P5282 | 最大上升子序列和（一本通 1285）
6. P5283 | 怪盗基德的滑翔翼（一本通 1286）
7. P5284 | 最低通行费（一本通 1287）

点击蓝色题名进入学校 OJ；本章 27 道练习全部列在索引中。

本章练习题索引 | 8–14 / 27

8. P5285 | 三角形最佳路径问题（一本通 1288）
9. P5271 | 拦截导弹（一本通 1289）
10. P5294 | 采药（一本通 1290）
11. P5295 | 数字组合（一本通 1291）
12. P5296 | 宠物小精灵之收服（一本通 1292）
13. P5297 | 买书（一本通 1293）
14. P5298 | Charm Bracelet（一本通 1294）

点击蓝色题名进入学校 OJ；本章 27 道练习全部列在索引中。

本章练习题索引 | 15–21 / 27

15. P5299 | 装箱问题（一本通 1295）
16. P5300 | 开餐馆（一本通 1296）
17. 高效工作（一本通 2003） P4982
18. P5308 | 公共子序列（一本通 1297）
19. P5309 | 计算字符串距离（一本通 1298）
20. P5310 | 糖果（一本通 1299）
21. P5311 | 鸡蛋的硬度（一本通 1300）

点击蓝色题名进入学校 OJ；“题包”表示该题暂未上学校 OJ。

本章练习题索引 | 22–27 / 27

22. P5312 | 大盗阿福（一本通 1301）
23. P5313 | 股票买卖（一本通 1302）
24. P5314 | 鸣人的影分身（一本通 1303）
25. P5315 | 数的划分（一本通 1304）
26. P5316 | Maximum sum（一本通 1305）
27. P5317 | 最长公共子上升序列（一本通 1306）

点击蓝色题名进入学校 OJ；本章 27 道练习全部列在索引中。