递推与递归

从小问题推出大问题：顺推填表、调用栈帧与记忆化

HGNU-ACM 实验室算法组

配套：一维/二维递推 + 栈帧与调用树 + 记忆化搜索 + 4 道真题精讲

本章目录 | 前半

顺推递推与边界初始化

二维棋盘递推与阻挡处理

递归与调用栈帧机制

终止条件与指数级重复子问题

承接第 5 章《贪心策略》：当“只看眼前一步”不够用时，学会顺推填表与自顶向下递归拆解。

本章目录｜后半

记忆化搜索：递归加缓存打通递推

四大递推与递归一本通真题精讲

常见错误、检查清单与练习安排

用“记忆化”打通递归与递推，精讲《上台阶》《过河卒》《汉诺塔》《数的计数》与避坑清单。

承接第 5 章：为什么有时“只看眼前一步”会翻车？

回顾《贪心策略》：硬币 $\{1,5,11\}$ 凑 15 元

- 贪心每次拿最大面额 11，结果 $11+1+1+1+1$ 用了 5 枚；真正最优是 $5+5+5$，只需 3 枚。

- 当“只看眼前一步”行不通时，换一条路：把每个小规模的答案依次算出来。
- 想求规模为 n 的答案，先问自己：能不能直接由更小的 $n-1$、$n-2$ 的答案算出来？
- 如果可以，先填好最小几项，再用循环从小到大把整张表填完——这就是递推。

本页要点

贪心是“看当前直接做决定”；递推是“先算好小问题，再按顺序推出大问题”。

先手算一个小例子：走楼梯每次跨 1 或 2 级

生活场景

共有 n 级台阶，从第 0 级出发每次可往上跨 1 级或 2 级，到第 n 级有几种走法？

先别急着写式子，在纸上把 $n = 1, 2, 3, 4$ 的所有走法列出来：

- $n = 1$：1 $\rightarrow$ **1** 种；　　$n = 2$：1+1，2 $\rightarrow$ **2** 种；
- $n = 3$：1+1+1，1+2，2+1 $\rightarrow$ **3** 种；
- $n = 4$：1+1+1+1，1+1+2，1+2+1，2+1+1，2+2 $\rightarrow$ **5** 种。

观察最后一步

想踩上第 4 级，最后一步只能从第 3 级跨 1 级上来，或从第 2 级跨 2 级上来，故 $f(4) = f(3) + f(2) = 5$！

本页要点

数方案数时，盯紧“最后一步从哪里来”，就能把大问题拆成几个小问题之和。

新术语大白话：状态、转移方程与边界初值

刚才的手算过程，对应递推的三个核心名词：

- 状态（**State**）：用数组表示某个小问题的答案。如 $f[i]$ 表示“走到第 i 级台阶的走法总数”。
- 状态转移方程（**Transition**）：大状态由前面小状态算出来的公式。如 $f[i] = f[i-1] + f[i-2]$。
- 边界初值（**Base Case**）：不能再往前推、必须人工填好的最小几项。如 $f[1] = 1,\ f[2] = 2$。

写递推前先回答三个问题

1. $f[i]$ 代表什么？　2. $f[i]$ 由哪几个更小的项算出来？　3. 循环开始前要提前填好哪几项？

本页要点

“状态”是格子的含义，“转移方程”是填格子的公式，“边界初值”是启动循环的第一块砖。

为什么不用暴力枚举所有走法？算一算操作次数

暴力做法

从第 0 级出发，每一步试探“跨 1 级”或“跨 2 级”，把每条走法一条条走到头并计数。

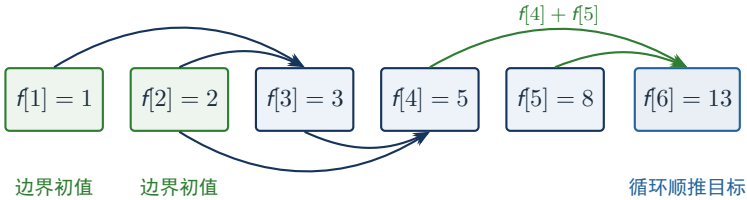
- 每走一步最多分裂成 2 个分支，走到第 n 级大约产生 $f(n)$ 条不同路径（约 1.618^n 呈指数增长）。
- 当 $n = 45$ 时，路径总数超过 1.8×10^9 条；当 $n = 70$ 时，路径数高达 10^{14} 级以上。
- 评测机 1 秒只能执行约 10^8 次基本运算，逐条数出来需要几万年，必然 **TLE**！
- 递推的威力：后续走法只关心“此刻在第几级”，把到达同一点的方案数直接相加，只需 $O(n)$ 步！

本页要点

暴力是“一条一条把路径数出来”（指数级）；递推是“把到达同一点的路径数直接相加”（$O(n)$）。

TikZ 图解：一维顺推依赖链与边界启动

以斐波那契型递推 $f[i] = f[i-1] + f[i-2]$ 为例，观察数组从左到右逐格填满的过程：



- 绿框 $f[1], f[2]$ 人工装入；从 $i = 3$ 起，每格只看左边紧邻的两格。
- 若往前要依赖 k 项（如 $i-1, i-2, i-3$），就必须先填好 k 个边界，避免 $i-k$ 越界读到负数下标。

本页要点

递推的方向必须“被依赖的格子先算完，当前格子后计算”。

手算跟踪表：计算 $n = 6$ 时每轮循环的变化

设 `f[1]=1`，`f[2]=2`，执行 `for (int i = 3; i <= 6; ++i)`，逐步填写下表：

判断前 i	$i \le 6$?	读取前两项	计算新值 $f[i]$	更新后 i	说明
3	真	$f[2]=2,\ f[1]=1$	$f[3]=3$	4	第一次进入循环体
4	真	$f[3]=3,\ f[2]=2$	$f[4]=5$	5	依赖刚算好的 $f[3]$
5	真	$f[4]=5,\ f[3]=3$	$f[5]=8$	6	顺推无须回溯
6	真	$f[5]=8,\ f[4]=5$	$f[6]=13$	7	得到目标答案 13
7	假	不读取	不执行	不更新	条件为假，退出循环

- 注意最后一行：$i = 7$ 时条件 $i \le 6$ 为假，循环终止，答案留在 `f[6]` 中。

本页要点

手算跟踪表一定要把“条件为假退出的那一行”写出来，才能确认循环不多算也不少算。

另一类经典一维递推：错排问题 D_n

生活例子：装错信封

n 封信和 n 个信封，每封信都不能装进自己的信封里（称为错排），共有 D_n 种装法？

- 手算边界：$n = 1$ 时无处可错放，$D_1 = \mathbf{0}$；$n = 2$ 时只能互换，$D_2 = \mathbf{1}$。
- 推导 D_n：第 n 封信放进第 k 个信封（k 有 $n-1$ 种选法），看第 k 封信去哪：
  1. 第 k 封恰好装进第 n 个信封：剩下 $n-2$ 封信自己错排，有 D_{n-2} 种；
  2. 第 k 封不装进第 n 个信封：把“第 n 个信封”临时看成“第 k 个信封”，等价于 $n-1$ 封错排，有 D_{n-1} 种。
- 合起来：$D_n = (n-1) \times (D_{n-1} + D_{n-2})$！验证：$D_3 = 2 \times (1+0) = 2$，$D_4 = 3 \times (2+1) = 9$。

本页要点

递推不止是简单相加，关键是“按某一个关键元素的去向分类讨论”。

正确性论证：为什么递推算出来一定不重不漏？

用“循环不变量”三步检验 $f[i] = f[i-1] + f[i-2]$（$f[i]$ 等于走到第 i 级的全部方案数）：

检验阶段	论证内容
初始化	进入循环（$i = 3$）前，$f[1] = 1$、$f[2] = 2$ 已由枚举确认，对 $1, 2$ 级完全正确。
保持	进入第 i 轮时，假定 $f[1] \dots f[i-1]$ 都已正确。所有到第 i 级的走法按最后一步分为两类：最后跨 1 级（来自 $i-1$，共 $f[i-1]$ 种）与最后跨 2 级（来自 $i-2$，共 $f[i-2]$ 种）。两类最后一步不同故绝不重复，且没有第三种跨法故绝不遗漏，相加后 $f[i]$ 仍正确。
终止	当 $i = n+1$ 时条件为假退出，此时 $f[1] \dots f[n]$ 全部满足不变量，$f[n]$ 即为所求。

本页要点

判定递推公式对不对，就看分类是否同时满足“互不重复（最后一步不同）”与“没有遗漏”。

模板代码：一维顺推（上台阶 1 或 2 级）

```
long long f[85] = {0};                    // 开 long long 防方案数溢出
int n;
cin >> n;
f[1] = 1; f[2] = 2;                       // 边界初值：前两阶直接人工填入
for (int i = 3; i <= n; ++i) {            // 从第 3 阶起按顺序从小推到大
    f[i] = f[i - 1] + f[i - 2];           // 最后跨 1 阶 + 最后跨 2 阶
}
cout << f[n] << '\n';                     // 输出第 n 阶的最终方案数
```

为什么这样写

1. 把数组开到 85 覆盖最大规模，全局或局部清零；2. 若输入可能为 $n = 1$ 或 $n = 2$，循环 $i \le n$ 一次也不执行，直接输出填好的 $f[1]$ 或 $f[2]$，天然兼容边界！

本页要点

边界初值先装好，主循环自然跳过小规模输入，不用额外写一堆 `if` 特判。

上机验证与反事实追问

三组测试数据（输入 → 预期输出）

- 最小边界：输入 1 → 预期输出 1（循环 $i = 3 \le 1$ 为假不执行，直接输出 f[1]）；
- 普通样例：输入 6 → 预期输出 13（与手算跟踪表一致）；
- 大边界值：输入 80 → 预期输出 37889062373143906（需 long long，不溢出）。

反事实追问

问：如果在上面模板里只初始化 f[1] = 1，忘了写 f[2] = 2 而从 i = 2 开始循环，会变成什么结果？

答：$i = 2$ 时会算成 f[2] = f[1] + f[0] = 1 + 0 = 1（错把 2 级台阶算成只有 1 种走法），进而导致后面 $f[3], f[4], \ldots$ 全部偏小，提交直接 **WA**！

本页要点

递推前面缺一个初值，后面整条链都会多米诺骨牌式全错。

一维递推常见易错对照

错误写法	触发结果	正确写法与原因
`int f[85];`	**WA**（溢出变负数）	当 $n \geq 46$ 时 $f[n] > 2.1 \times 10^9$，必须用 `long long`
未清空局部数组 `f`	**WA**（随机垃圾值）	局部数组自带乱码，必须写成 `= {0}` 或开成全局数组
循环从 `i=1` 开始算 `f[i-2]`	**RE / WA**（越界读 `f[-1]`）	先人工填入所有边界项，循环必须从首个合法项起步
只特判 `n==1` 没特判 `f[2]`	**WA**（数组越界或错值）	依赖几项就至少填好几项，或确认 $f[0]=1$ 语义是否自洽

本页要点

写完一维递推先自查三件套：数组清空了吗？初值够吗？会不会爆 `int`？

生活引入：棋盘街区路线遇到“施工封路”怎么办？

问题场景

在方格街道中，从左上角 $(0,0)$ 出发，每次只能向下走一格或向右走一格，要走到 $(3,3)$ 有几条路？如果 $(1,1)$ 正在修路不能走呢？

- 一维状态只用 i 表示“第几级台阶”；现在位置由行 i 和列 j 共同决定，需要二维数组 $dp[i][j]$ 表示“从 $(0,0)$ 走到 (i,j) 的路径条数”。
- 为什么不能直接用数学排列公式 $\binom{n+m}{n}$？因为路上有坑（阻挡点）！一旦某个格子被封住，所有经过它的路线都要作废，数学公式很难扣干净。
- 递推怎么绕开？盯紧最后一步：能走到 (i,j)，上一步只能是从正上方 $(i-1,j)$ 下来，或从正左方 $(i,j-1)$ 过来！

本页要点

状态有几个决定因素，数组就开几维；遇坑直接把该格方案数设为 0 即可。

为什么不能暴力一条条路径搜？

暴力 DFS 搜索所有路径

站在 $(0,0)$ 尝试向下或向右走一步，每步分裂成 2 个分支，直到走到 (n,m) 给计数加 1。

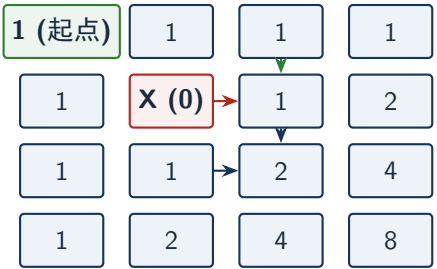
- 当棋盘为 20×20（没有封锁时），从 $(0,0)$ 到 $(20,20)$ 共有 $\binom{40}{20}\approx 1.378\times 10^{11}$ 条不同路径！
- 即使每秒跑 10^8 步，把这 1378 亿条路径走完也要将近 23 分钟，必然 **TLE**。
- 二维递推的工作量：不管路径有多少亿条，棋盘格子一共只有 $21\times 21=$ **441** 个！每个格子只做一次加法，一瞬间就算完。

本页要点

路径总数是指数级的，但棋盘格子数只有 $O(n\times m)$，递推把“千亿条路”压缩成“四百个格子的加法”。

TikZ 图解：二维棋盘填表方向与阻挡绕开

下面展示 4×4 格子（坐标 $0 \sim 3$），假设格子 $(1,1)$ 被封锁不可经过（标红 X）：



$(1,1)$ 是阻挡，强制 $dp[1][1] = 0$；
上方 $(0,2)$ 贡献 1，左方 $(1,1)$ 贡献 0，
故 $dp[1][2] = 1 + 0 = 1$！

$(2,2)$ 从上方 $(1,2)$ 得 1，
从左方 $(2,1)$ 得 1，合计 2。

本页要点

第 0 行只能从左边来（没有上方），第 0 列只能从上边来（没有左方）；内部格一律“上加左”。

手算跟踪表：按行扫描填满二维表

对上图 4×4 棋盘（$(1,1)$ 封锁），展示双层循环按行 $i=0\to3$、按列 $j=0\to3$ 扫描过程：

当前格子 (i,j)	是否封锁/起点	上方 $dp[i-1][j]$	左方 $dp[i][j-1]$	写入 $dp[i][j]$ 与备注
$(0,0)$	起点且未封	无（$i=0$）	无（$j=0$）	预先设为 1，循环跳过
$(0,1)\sim(0,3)$	未封	无（$i=0$）	$1,1,1$	依次得 $1,1,1$
$(1,0)$	未封	1	无（$j=0$）	$dp[1][0]=1$
$(1,1)$	封锁点	跳过计算	跳过计算	保持初值 0（自动断流）
$(1,2)$	未封	$dp[0][2]=1$	$dp[1][1]=0$	$1+0=1$
$(2,2)\to(3,3)$	未封	$1\to4$	$1\to4$	$dp[2][2]=2$，终点 $dp[3][3]=8$
$i=4$	条件为假	不访问	不访问	$i\le3$ 为假，外层循环退出

本页要点

扫描顺序是“从上到下、从左到右”，确保算到 (i,j) 时，它的正上方和正左方一定早已算好。

模板代码：二维网格递推（含阻挡处理）

```
vector<vector<long long>> dp(n + 1, vector<long long>(m + 1, 0));
if (!blocked[0][0]) dp[0][0] = 1;           // 起点未被封锁才初始化为 1
for (int i = 0; i <= n; ++i) {
    for (int j = 0; j <= m; ++j) {
        if (blocked[i][j] || (i == 0 && j == 0)) continue;
        if (i > 0) dp[i][j] += dp[i - 1][j]; // 仅当 i>0 时加上方格
        if (j > 0) dp[i][j] += dp[i][j - 1]; // 仅当 j>0 时加左方格
    }
}
cout << dp[n][m] << '\n';
```

本页要点

为什么这样写： 用 `if (i > 0)` 与 `if (j > 0)` 保护下标，自然兼顾第 0 行和第 0 列； 遇 `blocked` 直接 `continue` 让该格保持 0。

上机验证与反事实追问

三组验证用例（格式：n m 与阻挡位置 → 路径数）

- 最小边界：0 5，无阻挡 → 1；　普通样例：3 3，(1,1) 被挡 → 8；
- 特殊边界：3 3，起点 (0,0) 被挡 → 0（一步都迈不出去，全盘皆 0）。

反事实追问

问：如果把模板第 5 行写成 `if (blocked[i][j]) continue;`，忘了写 `|| (i == 0 && j == 0)` 会怎样？

答：虽然 $i=0, j=0$ 不满足 `> 0`，但一旦有人习惯写覆盖式 `dp[i][j] = dp[i-1][j] + dp[i][j-1]`，就会把起点 `dp[0][0]=1` 清零成 0！保留 `(i==0 && j==0)` 跳过最安全。

本页要点

写完循环后，一定要专门检查 $(0,0)$ 这一格有没有被循环体误改写！

二维递推常见易错对照

错误写法	触发结果	正确写法与原因
`dp[0][0] = 1` 不判封锁	**WA**（起点被挡却输出非零）	写成 `if (!blocked[0][0]) dp[0][0] = 1;`
直接写 `dp[i-1][j]`	**RE / WA**（$i=0$ 时下标 -1）	加上 `if (i > 0)` 与 `if (j > 0)` 保护边界
阻挡坐标直接赋值不判范围	**RE**（段错误崩溃）	标记马或障碍前先判 `0<=x<=n && 0<=y<=m`
使用 `int` 存 20×20 路径	**WA**（整数溢出）	$\binom{40}{20} \approx 1.37 \times 10^{11}$，必须使用 `long long`

本页要点

二维递推三大防线：偏移量先判边界防 **RE**，起点判封锁防 **WA**，结果开 `long long` 防溢出。

衔接第 6 章《函数》：为什么要让函数“自己调用自己”？

回顾 01 篇第 6 章

你已经学过函数的基本概念和最简单的 `sumTo(n) = n + sumTo(n-1)`。但在汉诺塔、多层嵌套等问题里，硬用 `for` 循环顺推很难追踪层数和柱子角色。

- 递推：从 1 往 n 顺着算（适合状态编号清晰、一维/二维数组能装下的题）。
- 递归：从 n 往 1 倒着拆——假装“小一号的自己”已经算好，我只需在外面收个尾！
- 这节我们要看清黑盒：函数自己调自己时，电脑内存里到底发生了什么？

本页要点

递推是“人手动写循环顺着推”；递归是“借助系统调用栈，先一路拆到底再逐层往回算”。

新术语大白话：递归、调用栈与栈帧

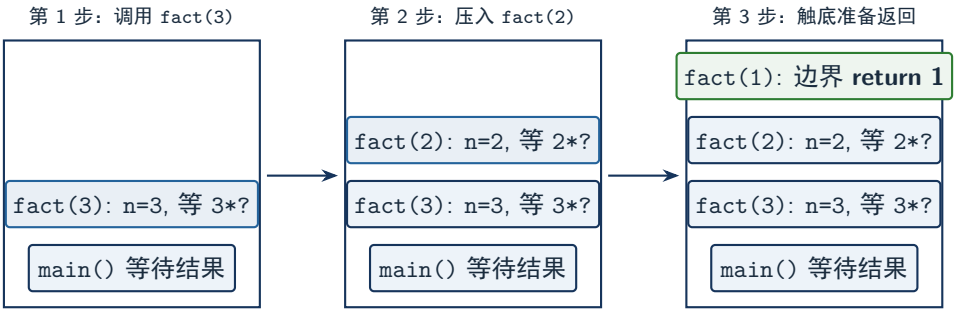
- 递归（**Recursion**）：函数在自己的函数体内部直接或间接调用自己，每次把问题规模缩小一点。
- 调用栈（**Call Stack**）：操作系统给程序准备的一摞“便签本”，遵循后进先出（Last-In, First-Out）：最后被调用的函数最先结束。
- 栈帧（**Stack Frame**）：每调用一次函数，系统就往这摞便签最顶上压一张新便签，里面只记三样东西：
  1. 本次传入的参数值（例如这一层里的 `n = 3`）；
  2. 本次内部的局部变量；
  3. 返回地址（内层算完回来后，接着执行哪一行）。

本页要点

每一层递归都有自己独立的“便签纸（栈帧）”，所以第 3 层的 `n=3` 绝对不会被第 2 层的 `n=2` 覆盖！

TikZ 图解：计算 fact(3) 时的“入栈（压栈）”过程

看阶乘函数 `fact(n) = n * fact(n - 1)`（边界 `n == 1` 返回 1），从 `main` 调用 `fact(3)` 一路向下压栈：

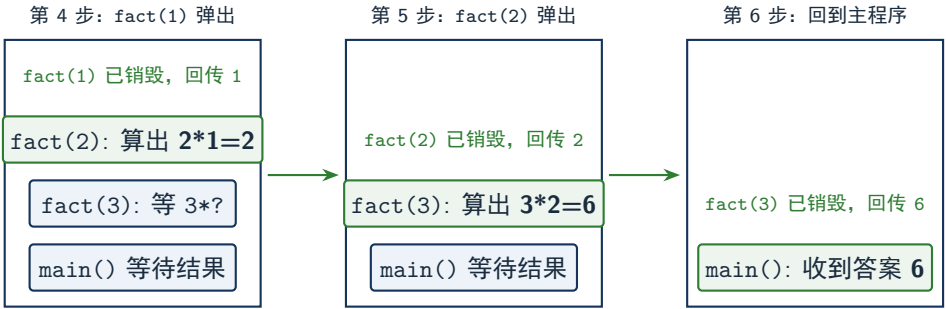


本页要点

压栈阶段只负责“一层层把任务挂起来”，真正的乘法计算此刻还没开始做！

TikZ 图解：触底后的“出栈（逐层返回）”过程

fact(1) 遇到终止边界返回 1 后，栈顶便签一张张撕掉，结果沿着栈往下层传递并完成乘法：



本页要点

先一路向下压栈，触底后再一路向上出栈结算——这就是递归在电脑里的真实运行轨迹。

手算跟踪表：调用 fact(4) 的完整栈帧变化

下面用表格按时间顺序记录 fact(4) 从入栈到清空的全部 8 个关键时刻：

步骤	动作	当前栈顶及栈内帧（从底到顶）	说明与返回结果
1	入栈	main → fact(4)	需要等 fact(3)
2	入栈	main → fact(4) → fact(3)	需要等 fact(2)
3	入栈	main → fact(4) → fact(3) → fact(2)	需要等 fact(1)
4	触底	main → ... → fact(1)	命中 n <= 1，直接返回 **1**
5	出栈	main → fact(4) → fact(3) → fact(2)	fact(2) 算 2 * 1 = 2 并返回
6	出栈	main → fact(4) → fact(3)	fact(3) 算 3 * 2 = 6 并返回
7	出栈	main → fact(4)	fact(4) 算 4 * 6 = 24 并返回
8	终止	main（所有递归栈帧已清空）	收到结果 **24**，继续执行主程序

本页要点

栈的最大高度（这里是 4 帧）决定了递归消耗的内存空间；深度过大会撑爆系统栈。

正确性论证：为什么写递归不需要人脑展开十几层？

判断一个递归函数对不对，只需用数学归纳法检查两件事：

证明阶段	在递归函数中如何检查
基础步（终止边界）	当输入达到最小规模（如 `n == 0` 或 `n == 1`）时，代码是否不调自己、直接给出正确答案？
归纳步（递归关系）	先相信更小规模 `f(n-1)` 一定能算出正确结果；在此前提下，当前层用 `n * f(n-1)` 拼出的结果是不是恰好等于 $n!$？
收敛性（参数变小）	每次调用传递的新参数是不是严格更接近终止边界（不会原地打转或越跑越远）？

- 写代码时，把 `f(n-1)` 当成一个“别人已经写好且百分百正确的现成函数”来调用！

本页要点

用归纳法信任小规模调用，人脑只看“当前这一层”和“最小边界”，再深也不用人肉展开！

模板代码：标准递归结构（带边界保护）

```
long long fact(int n) {
    if (n <= 1) return 1;                  // 1. 终止边界：最小情况直接返回
    long long sub = fact(n - 1);           // 2. 缩小规模：信任 sub 等于 (n-1)!
    return n * sub;                        // 3. 拼合答案：利用 sub 算出 n!
}
int main() {
    int n;
    cin >> n;
    cout << fact(n) << '\n';
    return 0;
}
```

本页要点

为什么这样写： `n <= 1` 放第一行防止 $n = 0$ 跌成负数； 调试时可临时打印缩进观察压栈出栈。

上机验证、反事实追问与栈帧易错对照

三组测试（输入 $n \to$ 预期输出 $n!$）

- 最小边界：$0 \to 1$；　普通样例：$4 \to 24$；　较大值：$20 \to 2432902008176640000$（long long 内最大值）。

反事实追问

问：如果在 fact(n) 里把第一行终止条件写成 if (n == 1) return 1;，输入 n = 0 时会发生什么？

答：$n = 0$ 跳过 n == 1，去调用 fact(-1) $\to$ fact(-2) $\to \ldots$ 永远停不下来，不断往系统栈压入新便签，约几十万层后耗尽系统默认几 MB 栈空间，触发 **RE**（栈溢出 / 段错误）！

本页要点

边界写 n <= 1 永远比 n == 1 更安全；一旦无限递归就会栈溢出 **RE**。

三种常见的递归爆栈（RE）写法对照

即使代码能通过编译，下面三种写法一旦运行几乎必报 **RE（Runtime Error）**：

错误类型	典型错误代码	为什么会爆栈崩溃
1. 漏写边界	`int f(int n){ return n+f(n-1); }`	没有 `if (...) return;` 出口，无限压栈到死
2. 边界写太窄	`if (n == 0) return 1;` 配 `f(n-2)`	传入奇数 $n = 5 \to 3 \to 1 \to -1 \to -3$ 直接跨过 0
3. 参数未变小	`if (n==1) return 1; return f(n);`	传给自己的还是 n，死循环原地压栈

- 此外，系统调用栈一般只有几 MB 到几十 MB，如果递推能用循环写到 $n = 10^6$，改写成递归深达 10^6 层照样会爆栈 **RE**！

本页要点

写完递归先拿笔测两件事：奇偶/负数会不会跳过边界？递归深度会不会超过 10^5？

纯递归的另一颗暗雷：为什么算个 fib(45) 要卡好几秒？

手算看看发生了什么

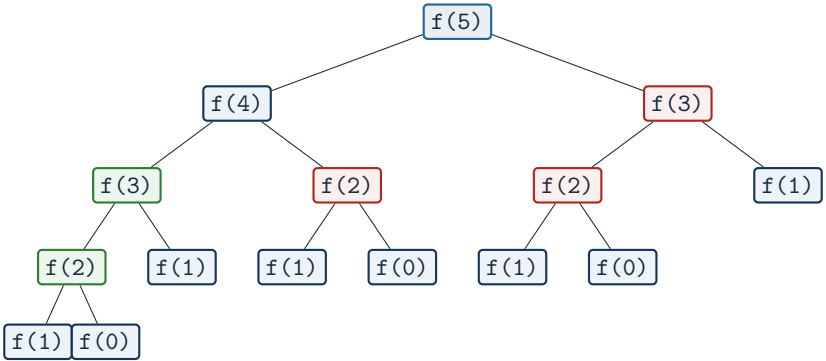
假设用最直白的公式写斐波那契递归：`fib(n) = fib(n-1) + fib(n-2)`（边界 `n <= 1` 返回 `n`）。
求 `fib(5)` 时，实际上我们只关心 $f(0), f(1), f(2), f(3), f(4), f(5)$ 这 6 个数；
但用纯递归跑一遍，函数到底被调用了多少次？

- `fib(5)` 拆出 `fib(4)` 和 `fib(3)`；
- `fib(4)` 又拆出 `fib(3)` 和 `fib(2)`——注意！`fib(3)` 刚刚明明已经安排算了一遍，这里又从头再算一遍！
- 结果算一个区区的 `fib(5)`，函数一共被调用了 15 次！

本页要点

“同一道子问题被反复从头算”叫重叠子问题，它是导致纯递归 **TLE** 的头号元凶。

TikZ 图解：看看 fib(5) 的调用树里有多少次重复计算！



本页要点

左半边绿框已经把 f(3) 和 f(2) 辛苦算完了，右半边红框却毫无记忆，又把整棵子树重跑一遍！

跟踪对比表：纯递归调用次数怎样从 15 次爆炸到四百亿次？

设 $C(n)$ 为算一次 `fib(n)` 触发的函数调用总次数，显然 $C(n) = 1 + C(n-1) + C(n-2)$：

规模 n	真正不同的子问题数	纯递归调用次数 $C(n)$	在评测机（10^8 次/秒）上的表现
5	6 个（$0 \sim 5$）	15 次	瞬间完成
10	11 个	177 次	瞬间完成
20	21 个	21 891 次	不到 1 毫秒
30	31 个	2 692 537 次	约 20 毫秒
40	41 个	331 160 281 次	约 3 秒（已经超时 **TLE**！）
50	51 个	40 730 022 147 次	约 6 分钟（严重 **TLE**！）

本页要点

同样算 51 个数，循环递推只要 50 步，纯递归却要调用 407 亿次！

反面教材代码：千万别直接交题的纯指数递归

```cpp
long long fib_bad(int n) {
    if (n <= 1) return n;                    // 终止条件看起来没毛病
    return fib_bad(n - 1) + fib_bad(n - 2); // 但每次分裂成两半，毫无记忆！
}
int main() {
    int n;
    cin >> n;
    cout << fib_bad(n) << '\n';              // n = 45 时就会明显卡死 TLE
    return 0;
}
```

本页要点

问题出在哪：代码虽短，但每层分裂成两支且不缓存，复杂度呈指数级 $O(2^n)$，$n > 40$ 必超时 **TLE**！

上机实测追问：为什么是 TLE 而不是爆栈 RE/MLE？

三组对照实验（实测 fib_bad(n) 的运行现象）

- `n = 5` → 瞬间输出 5；　`n = 20` → 瞬间输出 6765；　`n = 45` → 卡顿数秒才输出 1134903170。

反事实追问

问：刚才说 `fib_bad(50)` 调用了 407 亿次，那它的系统调用栈是不是同时压了 407 亿层而爆栈 RE/MLE 呢？
答：不会！因为左子树算完返回时，栈帧就弹出销毁并回收了空间；右子树复用同一片栈空间。整棵树同一时刻在栈里的最大深度只有 50 层（只占几 KB 内存）。树的最大深度决定空间（栈），树的总节点数决定时间（**TLE**）！

本页要点

最大深度 50 不会爆栈，但 407 亿个节点全走一遍会耗光时间——这就是 **TLE**。

纯递归什么时候能直接用？什么时候必须优化？

场景特征	能否直接用纯递归？	原因与对策
每层只调用 1 次自己（如阶乘、进制转换）	可以（前提深度 $< 10^5$）	没有重叠子问题，时间是线性的 $O(n)$
分叉但规模不大且无重复（如汉诺塔 $n < 20$）	可以	本身就要输出 $2^n - 1$ 步，没有重叠子问题
多个分支且反复走进同一参数（如斐波那契、数的计数）	绝对不行（必 **TLE**）	必须加记忆化数组，或改成循环递推

本页要点

判断要不要加记忆化，就问一句话：“不同的分支会不会再次走进相同的参数状态？”

生活引入：把算过的答案抄在草稿本上，下次直接查

查字典对比重推导

你在考场算数学题：第 1 步费劲算出了 $f(3)=2$；过两分钟第 2 步又需要用到 $f(3)$，你会重新埋头算一遍，还是直接看草稿纸上写好的 $f(3)=2$？

- 纯递归会超时，唯一的原因就是它“没有草稿本”，算过的结果转眼就忘。
- 只需要全局开两个小数组当草稿本：
  1. `memo[n]`：用来记住状态 `n` 算出来的答案；
  2. `vis[n]`：用来标记状态 `n` 有没有算过（`true/false`）。
- 每次进函数，第一件事先看 `vis[n]`：如果已经算过，直接返回 `memo[n]`，不再往下调用！

本页要点

记忆化 = 递归函数 + 两个缓存数组（`vis` 标记是否算过 + `memo` 存结果）。

新术语大白话：记忆化搜索与“自顶向下 vs 自底向上”

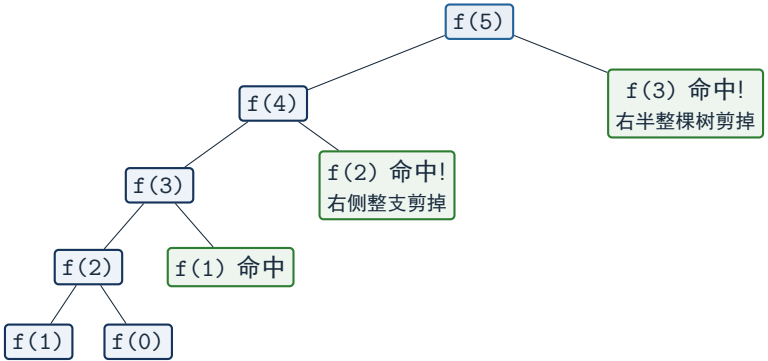
- 记忆化搜索（**Memoization**）：在递归函数入口先查缓存，命中直接返回；没命中才计算，算完立刻存进缓存再返回。
- 自底向上（递推）：从最小规模 $1, 2, 3$ 顺着用 `for` 循环填表填到 n。
- 自顶向下（记忆化递归）：直接问目标 n，需要谁就临时去算谁，算完填进 `memo` 表里。



本页要点

递推和记忆化搜索是同一枚硬币的两面：填的都是同一张状态表，只是填表顺序不同！

TikZ 图解：加上 memo 之后，fib(5) 的调用树瘦成一条链！



本页要点

左支算出 $f(0..4)$ 记入 memo，后面绿框直接 $O(1)$ 查表返回并剪掉整棵子树，复杂度降为 $O(n)$！

手算跟踪表：记忆化求 fib(5) 的真实执行轨迹

按函数实际被调用的先后顺序（先走左支再走右支），跟踪 vis 和 memo 的变化：

调用顺序	vis[k]	执行动作	写入结果	已缓存状态
f(5)->f(4)->f(3)->f(2)	均假	压栈到 f(2)	待返回	无
f(1), f(0)	均假	触发边界存缓存	memo[1]=1, [0]=0	$\{0,1\}$
回到 f(2)	刚算完	$1+0=1$	memo[2]=1	$\{0,1,2\}$
f(3) 查 f(1)	命中!	读 memo[1]=1	memo[3]=2	$\{0..3\}$
f(4) 查 f(2)	命中!	读 memo[2]=1	memo[4]=3	$\{0..4\}$
f(5) 查 f(3)	命中!	读 memo[3]=2	memo[5]=5	$\{0..5\}$ 完成

本页要点

总共只调用了 9 次函数（其中 3 次是一步到位的缓存命中），再无整棵子树的重复展开！

正确性论证：为什么加了缓存不会改变答案？

论证要素	为什么结果绝对等价
1. 纯函数性质	只要没有外部随机修改，同一个输入参数 `n` 每次算出来的返回值永远唯一确定。
2. 无环依赖（**DAG**）	每次递归参数严格变小（如 $n \to n-1, n-2$），不可能在缓存还没写入时自己绕回来查自己。
3. 写入不变量	当且仅当状态 `n` 的完整结果算完并写入 `memo[n]` 时，才将 `vis[n]` 置为 `true`。此后任何地方查到 `vis[n]==true`，读到的必是完整准确的最终答案。

- 复杂度账本：共有 $n+1$ 个不同状态，每个状态体内的计算只跑 1 次，故总时间为 $O(n)$；数组占 $O(n)$ 空间。

本页要点

记忆化能成立的前提是：同样的参数算出的结果永远相同，且依赖关系不会兜圈成环！

模板代码：记忆化递归标准写法

```
long long memo[1005];                          // 存放已经算出来的状态结果
bool vis[1005];                                // 标记某个状态是否算过（初始全 false）

long long solve(int n) {
    if (n <= 1) return n;                      // 1. 终止边界放在最前面
    if (vis[n]) return memo[n];                // 2. 查草稿本：算过直接 O(1) 返回！
    long long ans = solve(n - 1) + solve(n - 2); // 3. 没算过才拆成子问题
    vis[n] = true;                             // 4. 算完再打标记，写入缓存
    return memo[n] = ans;                      // 5. 保存并返回结果
}
```

本页要点

记忆化模板五步：判边界 → 查 vis → 算子问题 → 标 vis=true（子问题全算完后）→ 存 memo 返回。

上机验证与反事实追问

三组测试（用记忆化模板算 fib(n)）

- 最小边界：1 → 1；　普通样例：10 → 55；　原超时数据：50 → 秒出 12586269025。

反事实追问

问：如果懒得开 `bool vis[]`，只用 `if (memo[n] != 0) return memo[n];`，在所有题里都安全吗？
答：非常危险！若某个合法状态的答案恰好等于 0（如错排 $D_1 = 0$），`memo[n] == 0` 会被误判成“还没算过”，导致它反复重算退化成 **TLE**！务必单开 `vis[]`，或初始化为不可能的值 `-1`。

本页要点

千万别拿可能出现的合法答案（比如 0）充当“未计算”标记，老老实实开 `vis[]` 最稳！

记忆化易错对照与顺推递推选型表

易错场景 / 对比维度	常见坑点 / 递推特点	正确做法 / 记忆化特点
只查 vis 忘了存 memo	只写 return ans; 没写 memo[n]=ans	下次命中 vis 会返回初始化的 0（全 **WA**）
多组测试忘记清空缓存	第二组数据直接读到上一组的 memo	若题意依赖输入参数变化，每组需 memset 清零
什么时候选循环递推	状态转移规则整齐（一维/二维网格）	速度快、零递归栈开销，能到 $n = 10^6$ 不爆栈
什么时候选记忆化	状态转移方向不规则（如树的计数/区间）	不用费心找填表顺序，只算真正被访问到的状态

本页要点

状态顺序好排用递推，状态转移分支复杂、不想操心填表顺序用记忆化递归。

例 1 | 上台阶（一本通 1190）[入门]：先读懂要求

大白话题意

楼梯共有 n 阶（$0 < n < 71$）。每次上楼可以跨 1 阶、2 阶或 3 阶，求一共有多少种不同的走法。

- 输入：若干行，每行一个正整数 n；最后一行是 0，表示输入结束（0 不输出）。
- 处理：对每个 n 算出到达第 n 阶的走法总数（注意多组询问，最大到 $n = 70$）。
- 输出：每个询问占一行，输出对应的走法总数。

原题样例

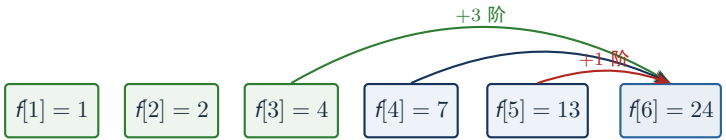
输入：1、2、3、4、0（每行一个） $\Longrightarrow$ 输出：1、2、4、7（每行一个）。

本页要点

先读清结束标志：读到 0 立即停下，不要对 0 输出一行！

例 1 | 怎么想到的？从暴力到三步汇聚图解

- 暴力为什么不行？每一步有 3 种跨法，$n = 70$ 时路径数超过 2×10^{18} 条，挨个模拟走一遍肯定 **TLE**。
- 关键性质：要踩上第 i 阶，最后一步只可能是从 $i-1$ 跨 1 阶、从 $i-2$ 跨 2 阶，或者从 $i-3$ 跨 3 阶。



- 转移方程：$f[i] = f[i-1] + f[i-2] + f[i-3]$（$i \geq 4$）。

本页要点

把一维台阶模板的“前 2 项相加”拓展成“前 3 项相加”，同时提前把 $1 \sim 70$ 一次算好！

例 1 | 样例与小边界手推表

人工先数清楚前 3 阶初值，再逐项推进到第 6 阶，同时核对 $i = 71$ 退出预处理循环：

台阶 i	依赖哪些项	计算算式 $f[i]$	走法或状态说明
1	边界初值	**1**	`1`
2	边界初值	**2**	`1+1`，`2`
3	边界初值	**4**	`1+1+1`，`1+2`，`2+1`，`3`
4	$f[3], f[2], f[1]$	$4+2+1=$ **7**	与原题样例第 4 行完全一致
5	$f[4], f[3], f[2]$	$7+4+2=$ **13**	每次只看紧邻前 3 项
6	$f[5], f[4], f[3]$	$13+7+4=$ **24**	增长很快，$n=70$ 达 2.07×10^{18}
71	条件为假	不计算	$i \le 70$ 为假，预处理循环停止

本页要点

前 3 项必须一个不漏人工填对 $(1, 2, 4)$；如果误把 $f[3]$ 写成 3，后面全盘皆错！

例 1 | 核心步骤与相对模板的改动

按顺序计算	纸上得到	核对问题
1. 设 `f[1]=1, f[2]=2, f[3]=4`	装好前 3 阶底座	$i=4$ 访问 $f[i-3]$ 下标为 1，不越界吗？
2. 循环 $i=4 \to 70$ 算 `f[i]`	整张表 $f[1\ldots70]$ 填满	开到 75 了吗？类型是 `long long` 吗？
3. `while(cin>>n && n!=0)`	每次直接输出 `f[n]`	读到 0 直接退出，不输出任何数吗？

本页要点

相对“一维模板”改动： 边界需填 3 项； 提前一次预处理 $1 \sim 70$，多组询问 $O(1)$ 查表； 开 `long long`。

例 1 | 完整可编译代码（code/stairs.cpp）

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
    ios::sync_with_stdio(false); cin.tie(nullptr);
    long long f[75] = {0};                      // 方案数超 2e9，必须用 long long
    f[1] = 1; f[2] = 2; f[3] = 4;               // 边界初值：前 3 阶直接填好
    for (int i = 4; i <= 70; ++i)
        f[i] = f[i - 1] + f[i - 2] + f[i - 3]; // 最后一步跨 1/2/3 阶三类相加
    int n;
    while (cin >> n && n != 0) cout << f[n] << '\n'; // 遇 0 终止，每次 O(1) 查表
    return 0;
}
```

本页要点

`while (cin >> n && n != 0)` 干净拦截结尾 0，预处理后每次询问仅 $O(1)$！

例 1 | 易错点与 3 组自测边界

常见易错

- 易错 **1**（溢出 **WA**）：开 `int f[75]`，$n \geq 37$ 即溢出变负数；必须用 `long long`；
- 易错 **2**（多输出一行 **WA**）：用 `do-while` 把结尾 `0` 也当成有效询问输出了一个 `0`；
- 易错 **3**（数组太小 **RE**）：开成 `f[70]` 导致访问 `f[70]` 越界（下标只到 69）。

三组自测数据（输入 $\rightarrow$ 预期输出）

- 最小边界：`1 / 0` $\rightarrow$ `1`；　　原题样例：`1 / 2 / 3 / 4 / 0` $\rightarrow$ `1 / 2 / 4 / 7`；
- 最大边界：`70 / 0` $\rightarrow$ `2073693258389777176`（亲测不溢出 `int`）。

本页要点

交题前一定跑一组 `n = 70`，看输出是不是正整数 2073693258389777176！

例 2 | 过河卒（一本通 1314，NOIP 2002）[基础]：先读懂要求

大白话题意

棋盘左上角 $A(0,0)$ 有一个卒，要走到右下角 $B(n,m)$（$0 \le n, m \le 20$），每次只能向下（$i+1$）或向右（$j+1$）走一格。棋盘上 $C(c_x, c_y)$ 放着一匹马，马本身以及它跳“日”字一步能到达的最多 8 个点都是封锁点，卒绝对不能经过。问共有多少条路径？

- 输入：一行四个整数 n, m, c_x, c_y（保证 $C \neq A$ 且 $C \neq B$）。
- 输出：一行一个整数，表示从 $(0,0)$ 到 (n,m) 的合法路径数。

原题样例

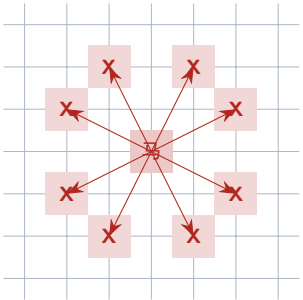
输入：`8 6 0 4` $\Longrightarrow$ 输出：`1617`。

本页要点

注意马自己的位置 C 也算控制点，加上 8 个“日”字跳点，一共最多封锁 9 个格子！

例 2 | 怎么想到的？马控 9 点方向与棋盘建模图解

- 有了马的 9 个控制点挖坑，排列组合无法绕开；直接套用二维网格递推，先标记 blocked 数组！



方向偏移表（共 8 组）：

```
dx = {1,1,-1,-1,2,2,-2,-2}
dy = {2,-2,2,-2,1,-1,1,-1}
```

马贴边时某些跳点会落在棋盘外，
赋值前必须先检查越界！

本页要点

用 dx[8]，dy[8] 偏移数组写循环标记马控点，比手写 8 条 if 整洁且绝不漏点。

例 2 | 原题样例 8 6 0 4 手推表（前 4 行细节 + 终点汇总）

对原题样例 $n=8, m=6$，马在 $(0,4)$（封锁 $(0,4),(1,2),(1,6),(2,3),(2,5)$），逐行递推结果如下：

行 i \ 列 j	0	1	2	3	4	5	6	关键格子计算说明
0	**1**	1	1	1	X	0	0	$(0,4)$ 是马，$dp[0][5..6]$ 截断为 0
1	1	2	X	1	1	1	X	$(1,3)$ 从上得 1；$(1,4)$ 从左得 1 绕过马
2	1	3	3	X	1	X	0	$(2,4) = dp[1][4] + dp[2][3] = 1+0 = 1$
3	1	4	7	7	8	8	8	第 3 行起无阻挡，路径重新铺满整行
$4 \to 8$	1	…	…	…	…	…	**1617**	逐行上加左，终行第 6 列得 **1617**

本页要点

看 $(1,4)$ 和 $(2,4)$ 怎么把“断流”的右半边重新接通——递推自然绕过所有封锁点！

例 2 | 核心步骤与相对模板的改动

按顺序计算	纸上得到	核对问题
1. 用 dx,dy 标出 blocked	$0 \sim 9$ 个格子被标为 1	加上 0<=x<=n && 0<=y<=m 判越界了吗？
2. 若 !blocked[0][0] 设起点 1	确定起点是否被马咬死	万一马在 $(1,2)$ 控住 $(0,0)$，起点为 0 吗？
3. 双重循环填 dp[i][j]	填满 $(n+1)\times(m+1)$ 表	用 long long 了吗？输出 dp[n][m] 吗？

本页要点

相对“二维模板”改动：主循环一字不差，只在前面新增一段“用马的 8 个日字偏移生成 blocked”的预处理。

例 2 | 完整可编译代码 (1/2)：标记马的 9 个控制点

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
    ios::sync_with_stdio(false); cin.tie(nullptr);
    int n, m, cx, cy;
    if (!(cin >> n >> m >> cx >> cy)) return 0;
    vector<vector<char>> blocked(n + 1, vector<char>(m + 1, 0));
    const int dx[8] = {1, 1, -1, -1, 2, 2, -2, -2};
    const int dy[8] = {2, -2, 2, -2, 1, -1, 1, -1};
    if (0 <= cx && cx <= n && 0 <= cy && cy <= m) blocked[cx][cy] = 1;
    for (int k = 0; k < 8; ++k) {
        int x = cx + dx[k], y = cy + dy[k];
        if (0 <= x && x <= n && 0 <= y && y <= m) blocked[x][y] = 1;
    }
```

本页要点

第 13 行先判范围再写入 `blocked[x][y]`，彻底杜绝负数或超界 **RE**。

例 2 | 完整可编译代码 (2/2)：二维填表与输出（code/pawn_crossing.cpp）

```
    vector<vector<long long>> dp(n + 1, vector<long long>(m + 1, 0));
    if (!blocked[0][0]) dp[0][0] = 1;          // 若马控制了起点，路径直接为 0
    for (int i = 0; i <= n; ++i) {
        for (int j = 0; j <= m; ++j) {
            if (blocked[i][j] || (i == 0 && j == 0)) continue;
            if (i > 0) dp[i][j] += dp[i - 1][j]; // 从正上方走下来
            if (j > 0) dp[i][j] += dp[i][j - 1]; // 从正左方走过来
        }
    }
    cout << dp[n][m] << '\n';
    return 0;
}
```

本页要点

用 vector 动态大小 $(n+1)\times(m+1)$ 且自动全清 0，既省空间又无溢出风险。

例 2 | 易错点与 3 组自测边界

常见易错

- 易错 **1**（越界 **RE**）：没判 x, y 范围直接 `blocked[cx+dx[k]][cy+dy[k]]=1`，负数下标直接崩溃；
- 易错 **2**（忘判起点 **WA**）：写死 `dp[0][0]=1`，马在 $(1,2)/(2,1)$ 时控住了起点却没归零；
- 易错 **3**（溢出 **WA**）：20×20 方案数约 1.37×10^{11}，必须使用 `long long`。

三组自测数据（输入 $\rightarrow$ 预期输出）

- 原题样例：`8 6 0 4` $\rightarrow$ `1617`；　　马控起点：`5 5 1 2` $\rightarrow$ `0`；
- 满范围大数：`20 20 10 10` $\rightarrow$ `21388094780`（检验 `long long`）。

本页要点

测试过河卒一定要测一组“马在 $(1,2)$ 控住起点”和一组“$n = m = 20$ 大答案”。

例 3 | 汉诺塔问题（一本通 1205）[基础]：先读懂要求

大白话题意

有三根杆子，最左杆从上到下由小到大串着 n 个圆盘（编号 $1 \sim n$）。要求把所有圆盘全部移到中间那根杆上，一次只能移一个盘，且任何时候不能把大盘压在小盘上。打印每一步移动记录。

- 输入：一个整数 n（$1 \le n < 20$），后跟三个单字符杆名（如 `a b c`）。
- 处理：将第 1 根杆上的 n 个盘，借助第 3 根杆，全部搬到第 2 根杆。
- 输出：每次移动一行，格式为 起始杆-> 盘号-> 目标杆（如 `a->3->b`）。

原题样例

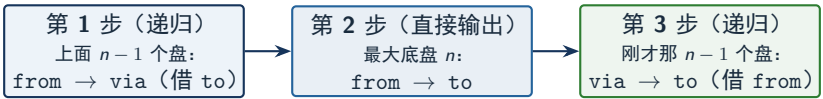
输入：`2 a b c` $\Longrightarrow$ 输出三行：`a->1->c`、`a->2->b`、`c->1->b`。

本页要点

特别盯紧题目要求：起点是第 1 个字符 `a`，目标是第 2 个字符 `b`，辅助杆才是 `c`！

例 3 | 怎么想到的？把 n 个盘拆成“$n-1$ + 底盘 n”三步图解

假设无论多少个盘，只要上面 $n-1$ 个盘都比最底下的大盘 n 小，那么它们就永远不会压坏 n：



- 这正是递归的“外包思想”：我们不需要关心那 $n-1$ 个盘子到底怎么一点点搬过去，只要调用 `hanoi(n-1, ...)` 外包给小一号自己就行！
- 当只剩 $n=0$ 个盘子时，什么都不用做，直接 `return`。

本页要点

记住三步口诀：上 $n-1$ 去辅助 $\rightarrow$ 底盘 n 直奔目标 $\rightarrow$ 上 $n-1$ 从辅助回目标。

例 3 | 样例 2 a b c 递归调用与返回手推表

定义函数 hanoi(n, from, to, via)（把 n 个盘从 from 借 via 移到 to），手推样例 2 a b c：

阶段	当前调用帧	执行动作	屏幕输出或返回
1	hanoi(2, a, b, c)	先执行第 1 步：调 hanoi(1, a, c, b)	压栈等待
2	hanoi(1, a, c, b)	调 hanoi(0, ...) 空返回 → 移盘 1	输出 a->1->c
3	hanoi(1, a, c, b)	调 hanoi(0, ...) 空返回	本帧结束，弹出
4	回到 hanoi(2, a, b, c)	执行第 2 步：直接把 2 号底盘移到 b	输出 a->2->b
5	hanoi(2, a, b, c)	执行第 3 步：调 hanoi(1, c, b, a)	移 1 号盘并输出 c->1->b
6	回到 main	所有栈帧清空	共输出 3 步，与样例一致

本页要点

递归执行顺序恰好就是“中序遍历”：左子树（先移 $n-1$）→ 根（移第 n 盘）→ 右子树（再移 $n-1$）。

例 3 | 核心步骤与三根杆参数防混淆法则

按顺序操作	参数对应含义	核对问题
1. 设计形参签名	`(int n, char from, char to, char via)`	位置固定为:(盘数, 起点, 终点, 辅助)
2. 主程序入口调用	`hanoi(n, a, b, c)`	题面要求移到中间杆 `b`，第二参是 `b`!
3. 两次递归传参	`(n-1,from,via,to)` 与 `(n-1,via,to,from)`	第 1 步把 `via` 当终点，第 3 步当起点

本页要点

永远按“从哪来 **(from)**、到哪去 **(to)**、借谁转 **(via)**”三个语义角色传参，绝不去死背字母 `a,b,c`!

例 3 | 完整可编译代码 (1/2)：递归函数体

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

void hanoi(int n, char from, char to, char via) {
    if (n == 0) return;                          // 终止边界：0 个盘直接结束
    hanoi(n - 1, from, via, to);                 // 1. 上面 n-1 个借 to 移到 via
    cout << from << "->" << n << "->" << to << '\n'; // 2. 底盘 n 从 from 直奔 to
    hanoi(n - 1, via, to, from);                 // 3. 把 via 上 n-1 个借 from 移到
    to
}
```

本页要点

注意输出盘号是当前层的 n（不是 1），因为正在移的是当前子问题的最底那个盘。

例 3 | 完整可编译代码 (2/2)：主程序读入（code/hanoi.cpp）

```cpp
int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n; char a, b, c;
    if (cin >> n >> a >> b >> c) hanoi(n, a, b, c);
    return 0;
}
```

为什么这样写

第 5 行严格按题面传入 (n, a, b, c)：起点 a、目标中间杆 b、辅助杆 c；同时关闭流同步避免大数据输出卡顿。

本页要点

即使题目某次把输入三杆字母换成 x y z，只要按读入顺序传入第 $1, 2, 3$ 个字符就永远正确！

例 3 | 易错点与 3 组自测边界

常见易错

- 易错 **1**（整题全 **WA**）：主程序错传 `hanoi(n, a, c, b)`，移去了第 3 根杆而非中间杆 `b`；
- 易错 **2**（输出格式 **WA**）：中间输出把盘号写死成 `1`，或漏掉盘号（应为 `a->2->b`）；
- 易错 **3**（无边界 **RE**）：漏写 `if (n == 0) return;`，一直递归到负数爆栈崩溃。

三组自测数据（输入 $\rightarrow$ 预期输出）

- 最小边界：`1 a b c` $\rightarrow$ `a->1->b;`　　原题样例：`2 a b c` $\rightarrow$ `a->1->c / a->2->b / c->1->b;`
- 换字母测试：`3 x y z` $\rightarrow$ 共 $2^3 - 1 = 7$ 步，第 4 步为 `x->3->y`，最终全落在 `y`。

本页要点

n 个盘的汉诺塔总步数恒为 $2^n - 1$；本题 $n < 20$，最多约 5×10^5 步，务必用 `'\n'` 输出。

例 4 | 数的计数（一本通 1316，NOIP 2001）[提高]：先读懂要求

大白话题意

给你一个自然数 n（$1 \le n \le 1000$）。你可以对它做以下操作（算上原数 n 本身）：1. 不加任何数，直接结束；2. 在它左边添上一个不超过原数一半（$\le \lfloor n/2 \rfloor$）的自然数（至少为 1），变成一个新数，然后对这个刚添上去的左边数字继续重复同样规则，直到不能再加为止。求一共能构造出多少个不同的数？

- 输入：一个自然数 n（$1 \le n \le 1000$）。　输出：满足条件的数的总个数。

原题样例

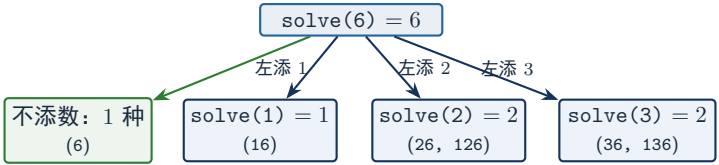
输入：`6` $\Longrightarrow$ 输出：`6`（分别是 `6`，`16`，`26`，`126`，`36`，`136`）。

本页要点

关键规则看清楚：下次添数时，只能看“最新添在最左边的那个数的一半”，不是看整个多位数！

例 4 | 怎么想到的？发现“只看左端新数”的子问题树图解

- 在 6 的左边放了一个 i（$1 \le i \le 3$）之后，接下来还能往左边接什么，完全只取决于 i 本身，跟右边的 6 毫无关系！
- 这意味着：“以 i 开头接出多少种后缀”就是原问题的缩小版 $f(i)$！



本页要点

转移式一眼看穿：$f(n) = 1 + \sum_{i=1}^{\lfloor n/2 \rfloor} f(i)$；但直接递归没有缓存的话，$n = 1000$ 必超时！

例 4 | 样例手推表（从 $n=1$ 推到 $n=6$）

按公式 $f(n)=1+\sum_{i=1}^{\lfloor n/2\rfloor} f(i)$ 从小到大核对：

n	可添范围	计算算式	构造出的合法数字（共 $f(n)$ 个）
1	无（0）	1	1（仅自身，**1** 个）
2	1	$1+f(1)=\mathbf{2}$	2，12
3	1	$1+f(1)=\mathbf{2}$	3，13
4	1, 2	$1+f(1)+f(2)=\mathbf{4}$	4，14，24，124
5	1, 2	$1+f(1)+f(2)=\mathbf{4}$	5，15，25，125
6	1, 2, 3	$1+f(1)+f(2)+f(3)=\mathbf{6}$	6，16，26，126，36，136（与样例一致）

本页要点

规律：$f(2)=f(3), f(4)=f(5)$，偶数与它后面的奇数左侧可添上限完全相同！

例 4 | 核心步骤与“记忆化 vs 递推”视角打通

按顺序执行	纸上得到	核对问题
1. 进 `solve(n)` 先查 `vis[n]`	算过直接 $O(1)$ 读 `memo[n]`	漏掉这一行，$n = 1000$ 必超时吗？
2. 设初值 `ans = 1`	计入“不添数”本身	错设成 0 会导致全零吗？
3. 枚举 $i = 1 \to n/2$ 累加 `solve(i)`	汇总所有左添前缀方案	循环条件写的是 `i <= n / 2` 吗？
4. 标记 `vis[n]` 并存 `memo[n]`	写入备忘录再返回	也可以写成顺推循环填 `f[1..n]`

本页要点

加记忆化后每个 $1 \dots n$ 只算一次，总计 $\approx \frac{n^2}{4} \approx 2.5 \times 10^5$ 次；与顺推填 `f[1..n]` 等价！

例 4 | 完整可编译代码 (1/2)：记忆化搜索核心

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

long long memo[1005];
bool vis[1005];

long long solve(int n) {
    if (vis[n]) return memo[n];                  // 查草稿本：算过直接返回 O(1)
    vis[n] = true;                               // 标记已访问（此处无环，算完会写入）
    long long ans = 1;                           // 1 种：不在左边添任何数
    for (int i = 1; i <= n / 2; ++i) ans += solve(i); // 枚举左侧添加的数 i
    return memo[n] = ans;                        // 保存进缓存并返回
}
```

本页要点

当 $n = 1$ 时，`i <= 1/2` 即 `i <= 0` 不执行循环，直接返回 1，天然兼顾最小边界！

例 4 | 完整可编译代码 (2/2)：主程序调用（code/count_nums.cpp）

```cpp
int main() {
    int n;
    if (cin >> n) cout << solve(n) << '\n';
    return 0;
}
```

为什么这样写

1. 全局数组 memo 与 vis 默认全清零，覆盖 $1 \le n \le 1000$；
2. solve(n) 内只向严格更小的 $i \le n/2$ 递归，绝无环路，提前写 vis[n] = true 也完全安全。

本页要点

记住：多组数据题若要复用全局 vis，这里单组直接 solve(n) 一次搞定。

例 4 | 易错点与 3 组自测边界

常见易错

- 易错 **1**（忘加记忆化 **TLE**）：没开 `vis/memo` 直接递归，$n = 100$ 就已经慢到卡死，更别提 $n = 1000$；
- 易错 **2**（漏掉边界 **WA**）：循环条件写成 `i < n / 2` 少了等于号，例如 $n = 6$ 时漏了左添 3；
- 易错 **3**（漏算自身 **WA**）：把 `long long ans = 1;` 写成了 `0`，导致 `solve(1)=0` 一路全零。

三组自测数据（输入 $\rightarrow$ 预期输出）

- 最小边界：`1` $\rightarrow$ `1`；　　原题样例：`6` $\rightarrow$ `6`；
- 中等自测：`10` $\rightarrow$ `14`；　　最大边界：`1000` $\rightarrow$ `1981471878`（一毫秒内秒出）。

本页要点

交题前一定亲手测一组 `1000`：既能检验有没有漏写记忆化超时，也能确认输出为 `1981471878`。

递推与递归高频错误对照表

题目 / 场景	常见错误写法	正确做法与原因
方案数溢出	`int f[75]` 或 `int dp[21][21]`	上台阶 $n = 70$ 和 20×20 过河卒远超 2×10^9，用 `long long`
忘记起点封锁	过河卒写死 `dp[0][0]=1` 不判马控	若马恰好在 $(1,2)/(2,1)$ 会控住起点 $(0,0)$，应先判 `!blocked[0][0]`
马控坐标越界	直接 `blocked[cx+dx][cy+dy]=1`	可能算出负数或 $> n$ 下标崩溃 **RE**，必须先判范围
递归忘记记忆化	不开 `memo/vis` 直接深搜	相同子问题重复展开达指数次 **TLE**
汉诺塔传错杆	主函数传成 `hanoi(n, a, c, b)`	本题要求移到中间杆 `b`，读清输入顺序再写参数
未计算哨兵错用	用 `memo[n]!=0` 判断已算过	若答案本身可能为 0 会失效，单开 `bool vis[]` 最稳

本页要点

递推盯紧“初值 + 防越界 + long long”，递归盯紧“边界 + 记忆化不重算”。

看懂评测结果：CE / WA / RE / TLE / MLE 定位指南

提交后没 AC 先别慌，按评测状态直奔对应排查点：

结果	本章最常见元凶	快速排查步骤
CE	函数先调用后定义、头文件缺失	检查递归函数有没有写在 `main` 上方（或加声明）
WA	方案数爆 `int`、边界初值少填、杆名传反	代 $n = 1, 2$ 小数据手算对比；对大范围测溢出
RE	递归无边界/跨过边界爆栈、马控越界	查终止条件是否用 `<=`；查数组下标是否先判 ≥ 0
TLE	重叠子问题忘加记忆化、或用暴力路径搜	检查同一状态是否被调用多次，立刻加上 `vis+memo`
MLE	递推深度/二维数组盲目开过大（如 $10^4 \times 10^4$）	算清内存大小，递归深度 $> 10^5$ 改循环递推

本页要点

看到 **RE** 优先查无限递归和负数下标；看到 **TLE** 优先查是不是忘了写记忆化！

提交前检查清单（六必查）

1. 是否在写递推循环前，先人工确认了最小 $2 \sim 3$ 项边界初值（如 $f[1], f[2], f[3]$ 或 $dp[0][0]$）？
2. 方案总数、路径总数在最大值域下会不会超出 2×10^9？是否用了 `long long`？
3. 二维格子填表或加减偏移量时，有没有对数组边界（< 0 或 $> n$）做好越界拦截？
4. 写的递归函数是否存在至少一条必定能走到的 `if (...) return;` 终止边界，且参数每次严格变小？
5. 当递归会从多个分支走进同一个状态时，是否加上了 `vis` / `memo` 记忆化缓存？
6. 题目如有指定输出格式、结束标记（如 `0` 结束）或杆名顺序，是否与样例完全对齐？

最后一步

造一组最小边界（$n = 1$ 或起点被控）和一组满范围测试（如 $n = 70$ 或 $n = 1000$），验证不超时、不溢出再提交！

本章全部练习与完成标准

题号	题目名称	难度	核心思路与代码路径
1190	上台阶	[入门]	$f[i] = f[i-1] + f[i-2] + f[i-3]$，查表，code/stairs.cpp
1314	过河卒（NOIP 2002）	[基础]	标出马控 9 点，二维递推，code/pawn_crossing.cpp
1312	昆虫繁殖	[基础]	成虫 $a[i]$ 与产卵 $b[i]$ 双数组顺推
1205	汉诺塔问题	[基础]	三步递归，目标杆为第 2 参 b，code/hanoi.cpp
1316	数的计数（NOIP 2001）	[提高]	$f(n) = 1 + \sum_{i=1}^{\lfloor n/2 \rfloor} f(i)$ 记忆化，code/count_nums.cpp
1199	全排列	[基础]	递归填第 dep 位，used[] 标记按序输出

本页要点

完成标准：能手算前 4 项推递推式、画栈帧、把指数递归改记忆化，独立 AC 以上 6 题。

课后安排与下一章预告

- 复习：合上代码再推一遍《过河卒》二维转移式，重点记忆马控 8 点的越界判断写法。
- 练习：用循环递推和记忆化递归两种写法分别完成《数的计数》，对比代码结构和运行速度。
- 下一章预告：第 7 章《分治算法》——当拆出来的子问题互不重叠时，用对半切开 $+$ 合并解锁快速幂 $O(\log p)$、循环赛倍增与快速选择均摊 $O(n)$！

本章一句话

递推是从小向大顺着填表，递归是从大向小拆解问题，记忆化让两者合二为一。

谢谢！