基础算法 · 第四章

递归算法

把大问题交给同名的自己

HGNU-ACM 实验室算法组

截图配套：2 道例题，14 道练习题





本章目录 | 前半

从函数调用说起

递归怎样执行

递归的三要素

递归树与记忆化

回溯与恢复现场

本章目录｜后半

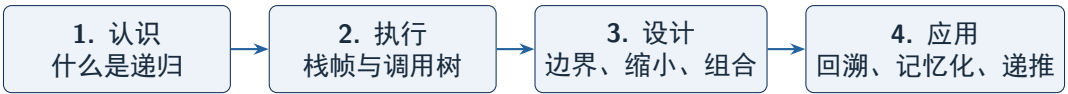
例 4.5：集合的划分

例 4.6：数的计数

递归与递推的关系

易错点、模板与练习

本章学习地图：先懂执行，再学设计



看不懂执行过程，
就写不对递归

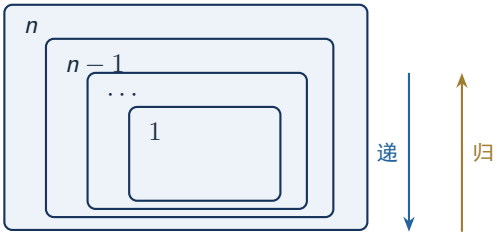
先有正确递归，
再谈优化

- 本章前一半讲基础：递归是什么、函数调用栈怎样工作、三要素怎样落地。
- 后一半再进入例题：集合划分、数的计数，以及递归与递推的互相转换。
- 每讲一个知识点，先给过程和适用条件，再给例子；不要只背模板。

本页要点

递归不是“高级循环”，而是一种用函数调用描述问题结构的方法。

什么是递归：一层套一层的同一个问题



- 生活中的例子：俄罗斯套娃，打开一层，里面还是同样的娃娃。
- 查字典时，解释里又出现需要再查的词，也是“同类问题套同类问题”。
- 程序里的递归函数：函数直接调用自己，或经过几个函数间接调用自己。
- 它由两部分组成：自己做的那一步和交给更小问题的同一函数。

一句话定义

递归就是：把规模为 n 的问题，转化成规模更小的同类问题；当规模小到可以直接回答时，返回答案。

递归函数的三块拼图

```
long long solve(int n) {
    if (n == 0) {                    // 1. 边界
        return 0;
    }
    long long sub = solve(n - 1);    // 2. 规模缩小
    return n + sub;                  // 3. 返回与组合
}
```

1. 边界

什么时候不再往下调用？最小输入是什么？

2. 规模缩小

下一层的参数为什么一定会靠近边界？

3. 返回与组合

子问题的答案怎样变成当前答案？

本页要点

写递归前必须回答：边界、缩小、组合。

第一个递归：求 $1+2+\cdots+n$

```
long long sum(int n) {
    if (n == 0) {                    // 边界
        return 0;
    }
    return n + sum(n - 1);           // 当前项 + 子问题
}
```

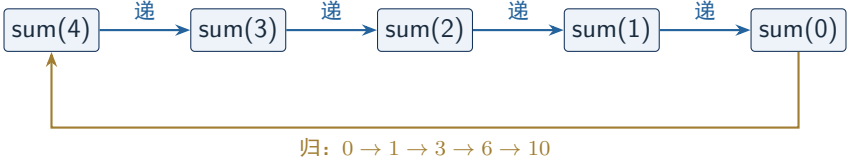
思路：$S(n) = n + S(n-1)$，且 $S(0) = 0$。

适用条件

问题能写成“当前这一步 + 同类子问题”，子问题规模严格变小，并且答案可以组合。

调用	等待的运算	返回值
sum(4)	$4+(\cdot)$	10
sum(3)	$3+(\cdot)$	6
sum(2)	$2+(\cdot)$	3
sum(1)	$1+(\cdot)$	1
sum(0)	无，到达边界	0

先“递”下去，再“归”回来

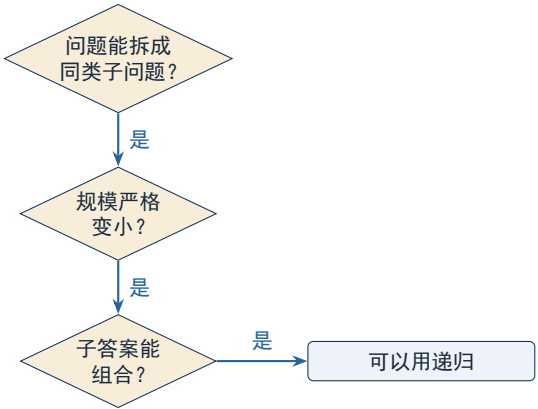


- 递：函数一层层调用自己，参数不断变小，暂时不计算结果。
- 归：到达边界后，最内层先返回，外层再用返回值完成自己挂起的运算。
- 所以递归的输出顺序、计算顺序，与代码的书写顺序并不相同。

本页要点

递归的执行顺序是“先一路向下，再从最底层逐层向上计算”。

递归的适用条件：什么时候值得用



- 问题结构本身是树或图：遍历、搜索、分治。
- 需要枚举所有方案并撤销选择：全排列、八皇后、迷宫。
- 重复子问题多、规模又不大：递归加记忆化。

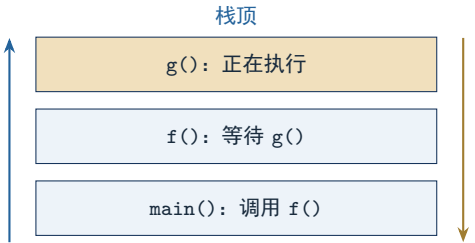
什么时候不要直接用递归

- 简单线性重复：1 到 n 累加、阶乘，用循环更省，也没有调用开销。
- 递归深度过大：链式深搜达到 10^5 层时，先估算栈空间，必要时改显式栈或递推。
- 重复子问题多又不记忆化：朴素菲波那契会指数级超时。
- 状态空间太大、记忆化也存不下：需要换状态设计或换算法。

本页要点

递归适合“结构自然、深度可控、子问题可组合”的场合；三者有一个不满足，就要考虑替代方案。

函数调用栈：每次调用都压入一个栈帧

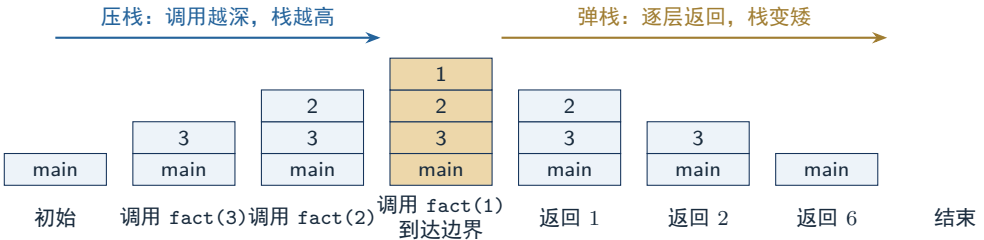


- 程序运行时，系统用一段内存维护“调用栈”。
- 每调用一个函数，就把这个函数的一层“活动记录”压到栈顶。
- 函数返回时，栈顶那一层被弹出，控制权回到调用者。
- 栈是后进先出：最后被调用的函数，最先返回。

本页要点

递归之所以能“回到上一层”，靠的就是调用栈保存了每一层的现场。

压栈与弹栈：fact(3) 的完整过程

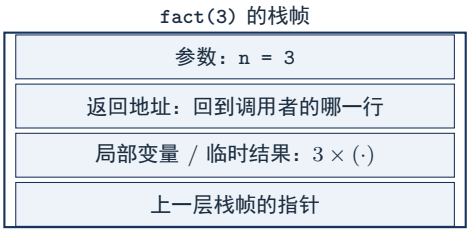


- 栈的高度就是当前递归深度；最高点出现在最内层调用。
- 每层只知道自己那一层的参数，返回后这层数据就被销毁。

本页要点

递归函数的执行过程，就是栈帧不断压入、再不断弹出的过程。

栈帧里到底存了什么



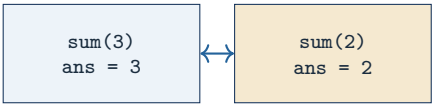
- 参数和局部变量：每层各有一份，互不覆盖。
- 返回地址：函数返回后，程序知道从哪里继续。
- 临时结果：例如阶乘里挂起的 $n \times (\cdot)$。
- 保存的寄存器、上一层帧指针等，具体内容由编译器决定。

本页要点

递归深度越大，同时存活的栈帧越多，占用的栈空间越大。

每层都有自己的变量

```
long long sum(int n) {
    // 本层变量
    long long ans = n;
    // 边界
    if (n == 0) {
        return 0;
    }
    // 加上子问题的答案
    ans += sum(n - 1);
    return ans;
}
```

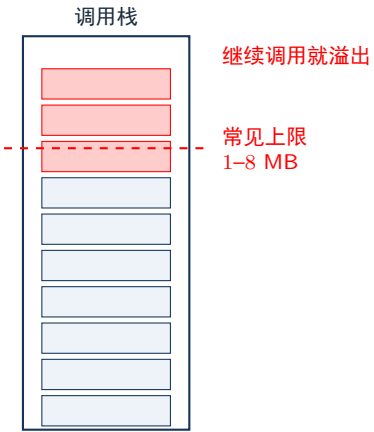


- sum(3) 的 ans 和 sum(2) 的 ans 是两个不同的变量。
- 内层修改自己的 ans，不会改掉外层的那一份。

本页要点

不要把递归中的局部变量想成全局共享；每一层都是独立的一份。

递归深度：栈空间是有限资源



- 每层栈帧通常占几十到上百字节，具体取决于参数、局部变量和编译器。
- 常见环境的栈空间约为 1–8 MB，递归 10^5 层就可能栈溢出。
- 深搜一条长链、递归处理 10^5 个结点时，要优先考虑显式栈或递推。

粗略估算

若每层约 64 字节：10^4 层约 640 KB，10^5 层约 6.4 MB，已经接近或超过常见上限。

栈溢出的原因：没有边界或深度过大

错误：没有边界

```
long long bad(int n) {
    // 永远停不下来
    return bad(n + 1);
}
```

危险：深度过大

```
void dfs(int u) {
    for (int v : g[u]) {
        // 链很长时会溢出
        dfs(v);
    }
}
```

- 没有边界会一直压栈；有边界但深度达到 10^5 以上，也可能超过栈空间。

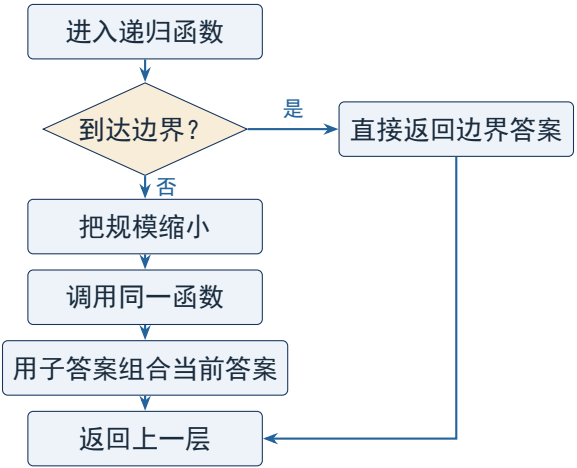
栈溢出的对策：先估算，再改写

- 改成显式栈：用数组或 vector 手动模拟递归，把栈放在堆上。
- 改成自底向上的递推，按依赖顺序填表，彻底消除递归深度。
- 缩小状态规模或提前剪枝，降低搜索树的最大深度。
- 不要依赖“把系统栈调大”这种不可移植的做法。
- 写代码前估算最大深度与每层字节数：深度 $\times$ 每层字节就是大致栈用量。

本页要点

先保证会停，再保证停得足够浅；这两件事都要在写代码前想清楚。

写递归之前，先回答三个问题



三要素各管什么

1. 边界

什么时候可以直接给出答案？例如 $n = 0$、空串、区间只剩一个元素。

2. 规模缩小

每次调用的参数必须朝边界靠近，否则永远停不下来。

3. 返回与组合

子问题的答案怎样变成当前答案？相加、相乘、取最大，还是只负责输出？

- 边界决定“什么时候停”，规模缩小决定“会不会停”，组合决定“答案对不对”。
- 三者缺一不可：没有边界会死递归，规模不缩小会死递归，不会组合就得不到答案。

边界：所有不需要再递归的情况都要覆盖

阶乘	求和	字符串	区间
$n = 0$ 或 $n = 1$	$n = 0$	空串	$l = r$
✓	✓	✓	✓

- 边界不是“随便写一个 $n = 0$”，而是所有不需要再递归的情况。
- 求 $n!$：规定 $0! = 1$，就把 $n = 0$ 和 $n = 1$ 都覆盖了。
- 全排列：当已经填满 n 个位置时输出，这是边界。
- 常见错误：只写了 $n = 1$，输入 0 时继续调用 $n - 1$，造成越界或死循环。

检查方法

把最小、次小的输入代进去，问自己：函数会在有限步内停下来吗？返回值与题目要求一致吗？

边界漏掉最小输入：一个真实的坑

只写了 $n = 1$

```
long long fact(int n) {
    // 只覆盖 n=1
    if (n == 1) {
        return 1;
    }
    // n=0 时还会往下
    return n * fact(n - 1);
}
```

- 输入 $n = 0$：调用 `fact(0)`，不满足 $n = 1$，继续调用 `fact(-1)`。
- 参数越来越小，却永远碰不到 $n = 1$，最终栈溢出或答案错误。
- 改成 `if (n <= 1)` 就能同时覆盖 0 和 1。
- 写边界时，先把参数的最小值、次小值列出来，再决定用等号还是小于等于。

本页要点

边界要覆盖“所有可以直接回答的输入”，不是只覆盖你最先想到的那一个。

规模缩小：参数必须真的向边界靠近

正确：每次减一

```
long long f(int n) {
    if (n <= 1) {
        // 边界
        return 1;
    }
    // 参数变小
    return n * f(n - 1);
}
```

错误：参数没有变小

```
long long bad(int n) {
    if (n == 0) {
        // 边界
        return 1;
    }
    // 参数没变小
    return bad(n) + bad(n+1);
}
```

- 每次递归调用的参数，必须能用一个“递减量”衡量，并且最终到达边界。
- 分治类递归把区间一分为二，区间长度也在变小。
- 回溯类递归让“已填位置数”增加，同样在向边界靠近。

规模缩小的三种常见方式



每次减一
线性递归

到达边界
停止

线性缩小

$n \to n-1$：阶乘、求和、汉诺塔。递归深度为 $O(n)$。

折半缩小

$[l, r] \to [l, mid]$ 和 $[mid+1, r]$：归并排序、二分。深度 $O(\log n)$。

按决策缩小

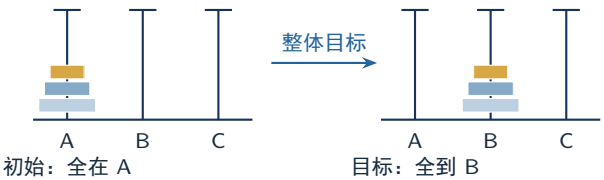
已填位置 $pos \to pos+1$：全排列、八皇后。深度等于决策步数。

返回与组合：子问题的答案怎样变成当前答案



- 求和：`return n + sum(n-1);`，组合方式是加法。
- 阶乘：`return n * fact(n-1);`，组合方式是乘法。
- 最大值：`return max(a[n], best(n-1));`
- 计数：把各分支的方案数累加。
- 只输出：组合方式可能只是“先递归左边，再输出自己，再递归右边”。
- 组合方式写错，递归结构再对，答案也会错。

三要素落地：汉诺塔

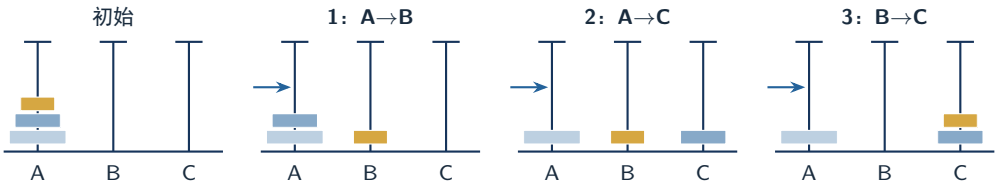


- 边界：$n = 0$ 时不用搬，直接返回。
- 缩小：先把上面 $n - 1$ 个从 A 搬到 C（借助 B）。
- 组合：搬最大盘 n 从 A 到 B，再把 $n - 1$ 个从 C 搬到 B（借助 A）。

移动次数

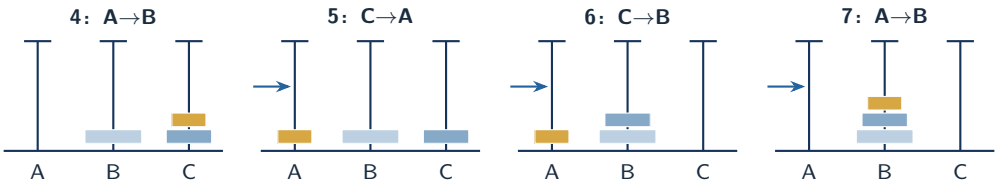
$T(n) = 2T(n-1) + 1$，且 $T(0) = 0$，解得 $T(n) = 2^n - 1$。$n = 64$ 时约为 1.84×10^{19}。

汉诺塔全过程（一）：前 3 步



- 前三步只关心“把上面 2 个从 A 搬到 C”，最大的盘 3 暂时不动。
- 第 2 步把盘 2 放到空杆 C，第 3 步才能把盘 1 放到盘 2 上。
- 每一步都必须遵守“小盘只能放在大盘上面”。

汉诺塔全过程（二）：后 4 步



- 第 4 步把最大的盘 3 从 A 搬到 B，这是整个任务的“中点”。
- 后三步把刚才暂存在 C 的 2 个盘搬回 B，过程与前三步对称。
- 递归的“组合”就落在第 4 步：先解决左边，再搬最大盘，再解决右边。

汉诺塔代码：把三要素翻译成函数

```
void hanoi(int n, char from, char via, char to) {
    if (n == 0) {                    // 边界：没有盘子
        return;
    }
    hanoi(n - 1, from, to, via);    // 上面 n-1 个先移走
    cout << from << " -> " << to << '\n';
    hanoi(n - 1, via, from, to);    // 再移回来
}
int main() {
    int n;
    cin >> n;
    hanoi(n, 'A', 'B', 'C');
    return 0;
}
```

- 参数 `via` 表示“可以借助的杆”，它让同一个函数能处理不同角色。
- 两次递归的 `from / via / to` 顺序不同，这是本题最容易写错的地方。

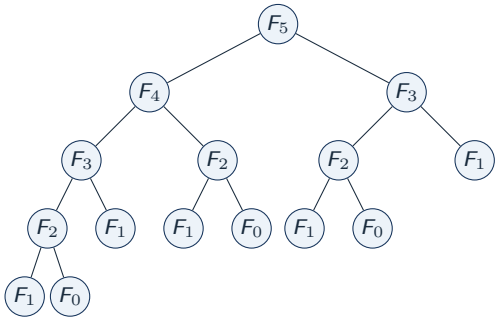
汉诺塔：复杂度与常见错误

- 时间：$T(n) = 2T(n-1) + 1$，展开得 $T(n) = 2^n - 1$，所以时间复杂度 $O(2^n)$。
- 空间：递归深度为 n，栈空间 $O(n)$；没有额外数组。
- $n = 64$ 时约 1.84×10^{19} 次移动，按每秒 10^8 次估算也要约 1845 亿秒（约 5850 年）。
- 常见错误一：两次递归的 `via` 没有交换，导致盘子搬到错误的杆。
- 常见错误二：先搬最大盘再递归，违反“小盘只能放在大盘上”。
- 常见错误三：边界写成 $n = 1$，虽然本题通常从 $n \geq 1$ 开始，但统一写 $n = 0$ 更稳。

本页要点

汉诺塔的价值不在“搬盘子”，而在于它把“两次同类子问题 + 一次当前操作”展示得非常清楚。

把递归调用画成一棵树



- 每个结点是一次函数调用，父子边表示“我调用了谁”。
- 叶子是到达边界、直接返回的调用。
- 树的高度就是最大递归深度。
- 结点总数就是总的调用次数。
- 同一层结点越多，这一层的工作量越大。

本页要点

递归树把“看不见的调用过程”变成了可以数、可以比较的图形。

数一数：朴素菲波那契为什么爆炸

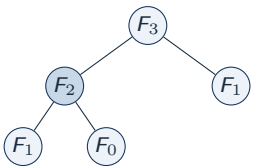
- 定义：$F_0 = 0$，$F_1 = 1$，$F_n = F_{n-1} + F_{n-2}$。
- 每次调用分裂成两个子调用，树几乎是一棵满二叉树。
- 调用次数满足 $C(n) = C(n-1) + C(n-2) + 1$。
- 解得 $C(n) = 2F_{n+1} - 1$，随 n 指数增长。

n	调用次数
5	15
10	177
20	21891
30	2692537
40	约 3.3×10^8

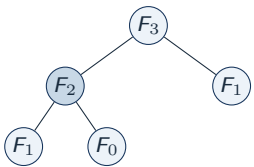
$$C(n) = C(n-1) + C(n-2) + 1, \qquad C(0) = C(1) = 1.$$

重复子问题：同一棵子树被算了多次

第一次计算 F_3



第二次又算 F_3



- 两棵子树的根都是 F_3，参数完全一样，答案也完全一样，却从头算了两遍。
- 在 F_5 的递归树里，F_3 出现 2 次，F_2 出现 3 次。
- 越往下的状态，重复次数越多；重复是指数级时间的根源。

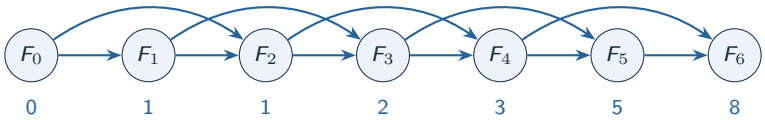
记忆化：算过的答案存起来

```
long long memo[100];
bool done[100];

long long fib(int n) {
    if (n <= 1) {                    // 边界
        return n;
    }
    if (done[n]) {                   // 已经算过
        return memo[n];
    }
    done[n] = true;
    memo[n] = fib(n - 1) + fib(n - 2);
    return memo[n];
}
```

- 每个状态只真正计算一次，之后用 $O(1)$ 的时间直接取答案。
- 时间从 $O(2^n)$ 降到 $O(n)$，空间增加一个记忆数组。

记忆化的执行过程：从树变成一条链



每个状态只算一次：$0, 1, 1, 2, 3, 5, 8$

- 先算 F_0, F_1，再算 F_2，依次向右；每个结点只被填入一次。
- 弧线表示 F_n 依赖 F_{n-2}，直线表示依赖 F_{n-1}，两种依赖都已经算好。
- 记忆化把“指数棵调用树”压成了“n 个状态的有向无环图”。

本页要点

记忆化的本质：用空间换时间，让每个状态只解决一次。

记忆化的适用条件与写法

必须满足

1. 状态由参数唯一确定：同样的参数，答案必须相同。
2. 状态之间没有循环依赖：递归必须能到达边界。
3. 状态数量可以接受：数组或哈希表存得下。
4. 每个状态的答案与“怎么走到这里”无关。

常见写法

- 数组记忆化：参数是小整数、范围固定。
- 哈希表记忆化：状态稀疏或参数较大。
- `done[]` 标记是否算过；也可用特殊值表示未算。
- 多参数状态：把参数组合映射成一维下标，或用多维数组。

- 如果状态数本身是指数级，记忆化也救不了；这时要换状态设计或算法。
- 记忆化不改变递归结构，只避免重复计算，所以先保证递归式正确。

用递归树估算复杂度：看每层做了多少事

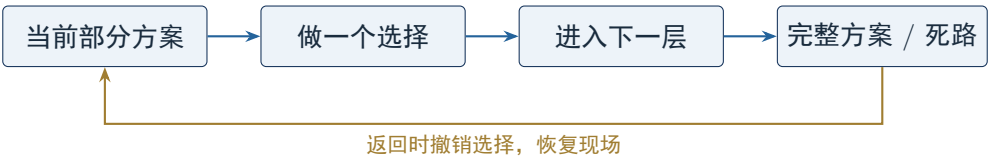
- 第一步：写出递归式，例如 $T(n) = 2T(n/2) + O(n)$。
- 第二步：画出每一层的结点数与单结点代价。
- 第三步：把各层代价相加；等比数列看公比，等差看层数。

算法	递归式	复杂度
朴素菲波那契	$T(n) = T(n-1) + T(n-2) + O(1)$	$O(2^n)$
记忆化菲波那契	每个状态一次	$O(n)$
归并排序	$T(n) = 2T(n/2) + O(n)$	$O(n \log n)$
二分查找	$T(n) = T(n/2) + O(1)$	$O(\log n)$
二叉树遍历	$T(n) = 2T(n/2) + O(1)$	$O(n)$
汉诺塔	$T(n) = 2T(n-1) + O(1)$	$O(2^n)$

本页要点

递归树不仅帮助理解，也是推导复杂度的正式工具。

回溯：在搜索树上“走一步，退一步”

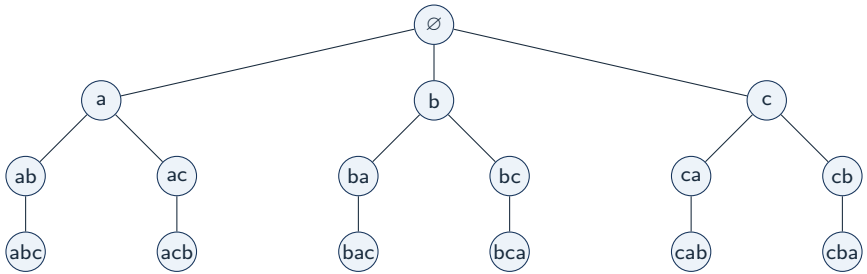


- 回溯是一种带“撤销”的递归：每层做一个选择，进入下一层；返回时把选择撤掉。
- 它适合解决“枚举所有方案”“找一个可行解”的问题，例如全排列、八皇后、迷宫。
- 搜索树上的每个结点是一个“部分方案”，叶子是一个完整方案或一条死路。
- 恢复现场是回溯的灵魂：不恢复，后面的分支就会带着错误状态继续。

本页要点

递归负责往下走，恢复现场负责回来时把棋盘擦干净。

全排列的搜索树：以 abc 为例



- 第 1 层选第一个字母，第 2 层选第二个，第 3 层选第三个；填满 3 个位置就到叶子。
- 根到叶子的每一条路径是一个排列，一共 $3! = 6$ 条路径。
- 按 a、b、c 的顺序扩展，叶子自然按字典序出现。

全排列代码（一）：全局状态与边界

```
#include <bits/stdc++.h>
using namespace std;
string s, cur;
bool used[10];

void dfs(int pos) {
    if (pos == (int)s.size()) {    // 边界：填满
        cout << cur << '\n';
        return;
    }
```

- pos 表示“正在填第几个位置”，它从 0 开始，每次加 1。
- cur 保存已经选出的前缀；到达边界时，它就是一个完整排列。
- used[i] 表示第 i 个字符是否已经在当前前缀中。
- 用 (int)s.size() 是为了避免有符号数与无符号数比较的警告。

全排列代码（二）：枚举分支与恢复现场

```
    for (int i = 0; i < (int)s.size(); ++i) {
        if (used[i]) {              // 已经用过
            continue;
        }
        used[i] = true;             // 做选择
        cur.push_back(s[i]);
        dfs(pos + 1);               // 进入下一层
        cur.pop_back();             // 恢复现场
        used[i] = false;            // 恢复现场
    }
}
```

- 进入下一层前：标记 used，把字符加入 cur。
- 从下一层返回后：按相反顺序撤销，先弹出字符，再清除标记。
- 这两行恢复缺一不可；否则下一个分支会看到上一个分支的残留状态。

全排列代码（三）：主函数与调用

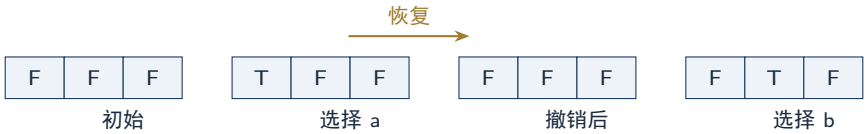
```
int main() {
    cin >> s;
    dfs(0);                          // 从第 0 个位置开始填
    return 0;
}
```

- 输入一个字符串，调用 `dfs(0)`，从空前缀开始搜索。
- 程序不需要在 `main` 里恢复任何状态；恢复发生在每一层返回时。
- 如果题目要求按字典序输出，输入本身应当有序，或先对字符串排序。

本页要点

回溯代码的结构：边界输出 → 枚举候选 → 做选择 → 递归 → 撤销选择。

恢复现场：为什么这一行不能省



前三个状态：a 分支返回后必须回到全 F

若不恢复，b 分支会看到 T，导致少枚举方案

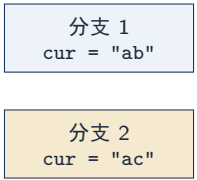
- 从 `dfs` 返回后，当前层的 `used[i]` 必须恢复为 `false`，否则下一个分支会误以为字母已被占用。
- 数组、标记、累加和、栈，只要是“这一层改过、下一层还要用”的状态，返回时都要还原。
- 如果状态是按值传递的（例如把当前字符串作为参数复制下去），就不需要手动恢复。

按值传递：复制一份状态，省去恢复

```
void dfs(int pos, string cur) {
    if (pos == n) {                 // 边界
        cout << cur << '\n';
        return;
    }
    for (int i = 0; i < n; ++i) {
        if (!used[i]) {
            used[i] = true;
            dfs(pos + 1, cur + s[i]);
            used[i] = false;        // 标记仍要恢复
        }
    }
}
```

- 参数 cur 按值传递，每次调用都会复制一份；返回时副本自动销毁，不需要手动撤销。
- 注意：全局 used 仍然必须恢复。

按值传递的代价：复制也有成本



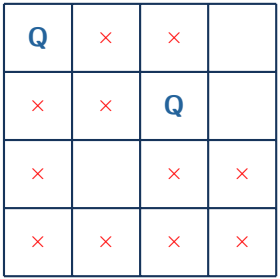
两个独立的字符串副本

- 每次调用都复制当前状态，天然隔离了不同分支，不需要在返回时手动撤销。
- 复制的代价是 $O(n)$；状态很大时，按值传递可能比“引用 + 恢复现场”更慢。
- 选择标准：状态小、复制便宜就按值；状态大、修改点多就用引用并认真恢复。

本页要点

“不用恢复”不是免费的，只是把恢复的成本换成了复制的成本。

剪枝：少走不可能的分支



4 皇后的一步部分棋盘

```
bool ok(int r, int c) {
    for (int i = 0; i < r; ++i) {
        if (qc[i] == c) {
            // 同列
            return false;
        }
        if (abs(qc[i] - c) == r - i) {
            // 同对角线
            return false;
        }
    }
    return true;
}
```

- `qc[i]` 是第 i 行皇后所在列；放第 r 行前检查同列和对角线，冲突就剪枝。

回溯模板：五个动作

```
void dfs(int pos) {
    if (pos == n) {                 // 1. 边界
        output();
        return;
    }
    for (int v : candidates) {
        if (!can_choose(v)) {
            continue;               // 2. 剪枝
        }
        choose(v);                  // 3. 做选择
        dfs(pos + 1);               // 4. 进入下一层
        undo(v);                    // 5. 恢复现场
    }
}
```

- 边界、候选集合、可行性判断、做选择、撤销选择，是回溯代码的五个固定位置。

回溯的复杂度：搜索树有多大

问题	搜索树规模	常见复杂度
全排列	n 层，每层最多 n 个分支	$O(n \cdot n!)$
子集枚举	每个元素选或不选	$O(2^n)$
八皇后	每行选一列，带剪枝	最坏 $O(n!)$
迷宫方案数	每个格子最多四个方向	最坏 $O(4^{nm})$

- 搜索树的结点数决定运行时间；输出本身也要花时间，全排列要输出 $n!$ 个长度为 n 的串。
- 剪枝能把“最坏情况”变小，但通常不改变最坏复杂度的量级。
- 估算时先问：一共有多少个方案？每个方案多长？这两者的乘积是输出的下界。

本页要点

回溯题先估搜索树大小，再决定是否需要剪枝、记忆化或换算法。

例 4.5 | 题意：把 n 个元素放进 k 个无标号盒子

题目

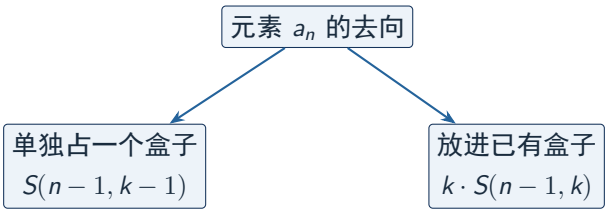
设 S 是有 n 个元素的集合，把它划分成 k 个非空、互不相交、并为 S 的子集 $S_1, \ldots, S_k$。求划分数 $S(n, k)$，其中 $0 < k \le n < 30$。

- 盒子无标号：$\{a_1, a_2\} \cup \{a_3\}$ 与 $\{a_3\} \cup \{a_1, a_2\}$ 是同一种划分。
- 每个盒子非空：不能有盒子空着。
- 这就是第二类斯特林数，记作 $S(n, k)$ 或 $\left\{ {n \atop k} \right\}$。

本页要点

先确认“盒子有没有编号、允许不允许空”，这两个条件决定递推式怎么写。

例 4.5 | 思路：看最后一个元素 a_n 去哪



- 情况一：a_n 单独占一个盒子。剩下的 $n-1$ 个元素要放进 $k-1$ 个非空盒，方案数 $S(n-1,k-1)$。
- 情况二：a_n 加入已有盒子。先把前 $n-1$ 个元素分成 k 个非空盒，有 $S(n-1,k)$ 种；再选 k 个盒子之一放 a_n，乘 k。
- 两种情况互不重叠、覆盖全部，所以相加。

本页要点

递推的关键：按最后一个元素的去向分类。

例 4.5 | 递推式、边界与手推

$$S(n,k) = S(n-1,k-1) + kS(n-1,k),$$
$$S(n,1) = S(n,n) = 1, \qquad S(0,0) = 1.$$

- 手算 $S(4,2) = 1 + 2S(3,2) = 7$，与直接枚举 $\{1,2,3,4\}$ 分成两块的结果一致。
- 题目样例 $S(10,6) = 22827$。

$n\backslash k$	1	2	3	4
1	1			
2	1	1		
3	1	3	1	
4	1	7	6	1

- $n < 30$ 时最大答案约 1.81×10^{22}，超过 `long long`；必须用 128 位整数或高精度。

例 4.5 | 代码（一）：128 位整数

```
#include <bits/stdc++.h>
using namespace std;
using i128 = __int128_t;
i128 S[31][31];

void print128(i128 x) {
    if (x == 0) {                    // 特判 0
        cout << '0';
        return;
    }
    string s;
```

- `__int128_t` 是 GCC 提供的 128 位整数，能存约 1.7×10^{38}。

例 4.5 | 代码（一，续）：逐位输出

```
    while (x > 0) {
        s.push_back(char('0' + x % 10));
        x /= 10;
    }
    reverse(s.begin(), s.end());
    cout << s;
}
```

- C++ 标准库没有直接输出 128 位整数的运算符，所以要自己转成字符串。

例 4.5 | 代码（二）：二维递推

```
int main() {
    int n, k;
    cin >> n >> k;
    S[0][0] = 1;
    for (int i = 1; i <= n; ++i) {
        for (int j = 1; j <= i; ++j) {
            S[i][j] = (i128)j * S[i - 1][j]
                    + S[i - 1][j - 1];
        }
    }
    print128(S[n][k]);
    cout << '\n';
    return 0;
}
```

- 循环顺序保证计算 $S(i,j)$ 时，$S(i-1,j)$ 和 $S(i-1,j-1)$ 已经算好。
- 强制转换 `(i128)j` 可以避免乘法在 64 位范围内溢出。

例 4.5 | 复杂度与易错点

- 状态数 $n \times k$，每个状态做一次加法和一次乘法：时间 $O(nk)$，空间 $O(nk)$。
- 最大 $S(29, 11) \approx 1.81 \times 10^{22}$，必须用 128 位整数或高精度；`long long` 会溢出。
- 盒子无标号，所以是 $kS(n-1, k)$，不是 $k!$ 或 k^k。
- 忘记 $S(n, n) = 1$ 或 $S(n, 1) = 1$，小数据就会错。
- 循环从 $i = 1$ 到 n、$j = 1$ 到 i，保证 $S(i-1, \cdot)$ 已算好。

本页要点

先判断答案范围，再决定用 `long long` 还是 128 位整数。

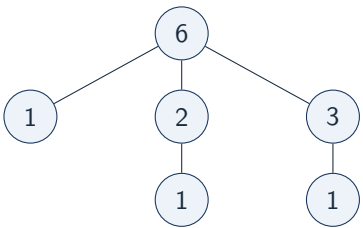
例 4.6 | 题意：不断在左边加数

题目（Noip2001）

给出自然数 $n \le 1000$。对 n 反复执行：在它的左边加上一个自然数，但这个数不能超过“当前原数”的一半；直到不能再加为止。问一共能得到多少个不同的数（包括 n 本身）。

- 准确理解规则：当前最左边的数是 x（初始 $x = n$）；若 $x \ge 2$，可以在左边写一个 y，满足 $1 \le y \le \lfloor x/2 \rfloor$。
- 写完 y 后，y 成为下一轮的“当前原数”，继续按同样的规则往左加。
- 例如 $n = 6$：先可加 $1, 2, 3$，得到 $16, 26, 36$；对 26 而言当前最左数是 2，还可加 1，得到 126。
- 问的是数的个数，不是操作步数；不同的加数路径可能得到同一个数吗？本题不会，因为数字串唯一确定路径。

例 4.6 | 把过程画成一棵“加数树”



- 根是 6；它的孩子是允许加在左边的 $1, 2, 3$。
- 2 的孩子是 1（因为 $\lfloor 2/2 \rfloor = 1$）；3 的孩子也是 1。
- 把根到某个结点路径上的数字从右往左拼起来，就得到一个数。

例 4.6 | 路径与数值的对应

路径	得到的数
6	6
$6 \to 1$	16
$6 \to 2$	26
$6 \to 2 \to 1$	126
$6 \to 3$	36
$6 \to 3 \to 1$	136

本页要点

树的结点总数就是答案；$n = 6$ 时共有 6 个结点。

例 4.6 | 递推式：以最左数为状态

- 定义 $f(x)$：当前最左数是 x 时，能得到的数的个数（包括 x 本身）。
- 左边可以接 $1, 2, \ldots, \lfloor x/2 \rfloor$；接上 i 之后，剩下的方案数正好是 $f(i)$。
- 所以

$$f(x) = 1 + \sum_{i=1}^{\lfloor x/2 \rfloor} f(i), \qquad f(1) = 1.$$

- 手算 $f(6) = 1 + f(1) + f(2) + f(3) = 1 + 1 + 2 + 2 = 6$。
- 其中 $f(2) = 1 + f(1) = 2$（对应 $2, 12$），$f(3) = 1 + f(1) = 2$（对应 $3, 13$）。

前缀和优化

记 $s(m) = \sum_{i=1}^{m} f(i)$，则 $f(x) = 1 + s(\lfloor x/2 \rfloor)$。从小到大计算 f，同时维护 s，可把 $O(n^2)$ 降到 $O(n)$。

例 4.6 | 记忆化递归

```
long long f[1005];
bool vis[1005];

long long solve(int n) {
    if (vis[n]) {                   // 算过就直接返回
        return f[n];
    }
    vis[n] = true;
    long long ans = 1;              // n 本身
    for (int i = 1; i <= n / 2; ++i) {
        ans += solve(i);            // 左边接上 i
    }
    return f[n] = ans;
}
```

- 状态是“当前最左数”，参数 n 唯一决定答案，所以可以记忆化。
- 边界是隐式的：$n = 1$ 时循环不执行，返回 1；$n = 0$ 不会被递归到。

例 4.6 | 主函数与答案范围

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

- 输入只有一个整数 n，输出方案数。
- $f(1000) = 1981471878$，在 int 范围内，但中间量用 long long 更稳。
- 记忆数组大小 1005 足够覆盖 $n \le 1000$。

为什么答案这么大？

$n = 1000$ 时可以接出的数远多于 1000 个；答案接近 2×10^9，已经接近 int 上限 2.147×10^9。

本页要点

先估答案范围，再决定用 int 还是 long long；不要等溢出后才改。

例 4.6 | 复杂度、优化与易错点

- 状态有 n 个，每个状态枚举到 $n/2$：时间 $O(n^2)$，空间 $O(n)$；$n \le 1000$ 时约 5×10^5 次加法。
- 用前缀和可优化到 $O(n)$：记 $s(m) = \sum_{i=1}^{m} f(i)$，则 $f(n) = 1 + s(\lfloor n/2 \rfloor)$。
- 易错一：把“不能超过一半”理解成“不能超过最初输入的 n 的一半”；每一轮的限制对象是当前最左数。
- 易错二：把“新数的一半”当成限制；例如 26 的限制来自刚加的 2，不是 26。
- 易错三：忘记把 n 本身计入；$f(1) = 1$ 是边界。

本页要点

递归式写对后，先看能不能用前缀和把重复的求和省掉。

同一个式子，两种实现

递归：自顶向下

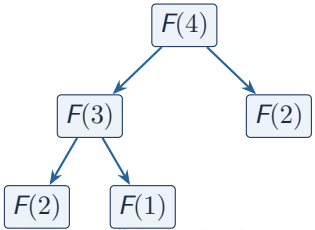
```
long long fact(int n) {
    if (n <= 1) {
        return 1;
    }
    return n * fact(n - 1);
}
```

递推：自底向上

```
long long fact(int n) {
    long long r = 1;
    for (int i = 2; i <= n; ++i) {
        r *= i;
    }
    return r;
}
```

- 递归把“大问题需要什么”写在代码里，结构清晰，但有调用开销和栈深度限制。
- 递推把“小问题先算好”写在循环里，速度快、空间省，但需要先想清楚计算顺序。

自顶向下与自底向上：方向相反



递归：先向下问
到达边界再向上算



递推：从小下标开始
算好一个填一个

- 递归的箭头方向是“调用”：从大问题指向小问题；返回值沿相反方向回来。
- 递推的箭头方向是“依赖”：先算 f_0, f_1，再算 $f_2, f_3, \ldots$。
- 两者描述的是同一个递推关系，只是计算顺序和实现手段不同。

从递归到递推：四步转换

1. 写出递归式与边界，例如 $f(n) = 1 + \sum_{i=1}^{\lfloor n/2 \rfloor} f(i)$，$f(1) = 1$。
2. 确定依赖方向：$f(n)$ 只依赖比它小的下标，所以从小到大计算。
3. 用数组保存每个状态，循环填表；必要时用前缀和、滚动数组优化。
4. 用样例和小规模手算对照，确认边界和下标没有差一。

判断依赖方向

如果 $f(n)$ 只依赖 $f(1), \ldots, f(n-1)$，就可以从小到大递推；如果依赖更大的下标，就要换状态定义或换计算顺序。

本页要点

递归式是“关系”，递推是“按关系排好顺序计算”。

从递归到递推：数的计数的循环写法

```
long long f[1005], s[1005];

int main() {
    int n;
    cin >> n;
    f[1] = 1;
    s[1] = 1;
    for (int i = 2; i <= n; ++i) {
        f[i] = 1 + s[i / 2];        // 前缀和代替内层循环
        s[i] = s[i - 1] + f[i];
    }
    cout << f[n] << '\n';
    return 0;
}
```

- s[i] 保存 $f(1) + \cdots + f(i)$，所以 $f(i)$ 可以 $O(1)$ 求出。
- 时间复杂度 $O(n)$，空间复杂度 $O(n)$；也可以只保留前缀和数组。

递归与递推对照表

维度	递归	递推
思考方向	自顶向下，先问子问题	自底向上，先算小状态
代码结构	函数调用自己	循环填表
额外开销	栈帧、调用、返回	一般没有
深度风险	可能栈溢出	无递归深度
重复计算	不记忆化会重复	每个状态只算一次
适用场景	树、分治、回溯、状态转移自然	依赖顺序清晰的计数、DP
调试难度	调用树较难跟踪	下标和顺序较直观

本页要点

不要问“递归和递推哪个更高级”；先看问题结构，再看深度、重复计算和实现成本。

什么时候优先用递归

优先递归

- 问题结构本身是树或图：二叉树遍历、搜索。
- 需要枚举所有方案并撤销选择：全排列、八皇后、迷宫。
- 分治：把区间分成两半分别处理，再合并答案。
- 递归式自然，但直接递推的下标顺序难想。

优先递推

- 简单的一维/二维递推，依赖顺序清晰。
- 递归深度可能很大，栈空间不够。
- 子问题大量重复，记忆化后状态仍然很多。
- 需要滚动数组、前缀和等空间优化。

- 递归 + 记忆化是把“指数级搜索”变成“多项式级动态规划”的常用桥梁。

递归的五个高频错误

1. 没有边界，或边界漏掉最小输入（$n = 0$、空串、空区间）。
2. 参数没有向边界靠近，造成无限递归。
3. 回溯后忘记恢复现场，后面的分支带着脏状态。
4. 递归深度过大，栈溢出；或答案超过 `int`/`long long`。
5. 重复子问题没有记忆化，指数级时间超时。

排查顺序

先写小数据手推，再在函数入口打印“参数 + 深度”，最后检查边界与恢复现场。

本页要点

五个错误里，前三个是“正确性”问题，后两个是“资源”问题；先保证正确，再保证跑得动。

调试递归（一）：让调用过程可见

```
void dfs(int pos, int depth) {
    for (int i = 0; i < depth; ++i) {
        cerr << "  ";
    }
    cerr << "dfs(" << pos << ")\n";  // 缩进显示层级
    if (pos == n) {                  // 边界
        return;
    }
```

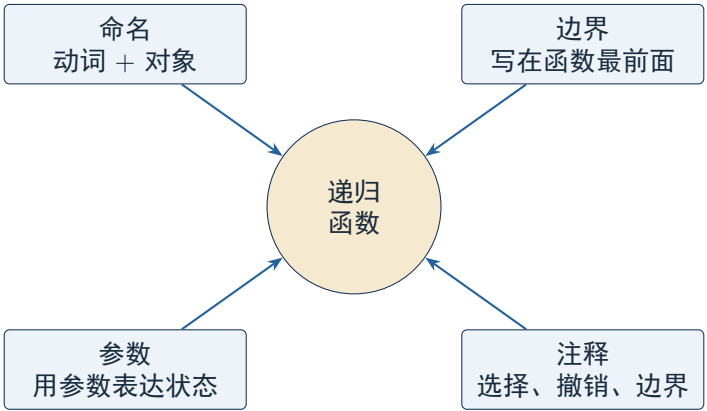
- 只看最终输出很难判断递归在哪一层走错，缩进日志能直接看出调用树。
- 用 cerr 输出调试信息，不会混进标准输出的答案里。
- 数据小时可以手动画出递归树，把每层的参数写在结点里。

调试递归（二）：枚举与恢复的日志

```
    for (int i = 0; i < n; ++i) {
        if (used[i]) {
            continue;
        }
        used[i] = true;
        dfs(pos + 1, depth + 1);
        used[i] = false;
    }
}
```

- 在 `used[i] = true` 和 `false` 旁边各打印一次，能看出“进入哪个分支、从哪个分支返回”。
- 如果日志里同一层的 `used` 没有回到进入前的值，就说明恢复现场漏了。
- 调试信息在上交前要删掉，避免输出超时或格式错误。

递归代码规范：四个位置



- 命名说明“做什么”；边界写在最前面；注释写选择、撤销和边界。

常用模板（一）：计数与判断

计数 / 求和

```
long long dfs(int n) {
    if (n == 0) {
        return 1;
    }
    long long ans = 1;
    for (int v : next(n)) {
        ans += dfs(v);
    }
    return ans;
}
```

判断存在性

```
bool dfs(int n) {
    if (n == target) {
        return true;
    }
    if (n > target) {
        return false;
    }
    if (dfs(2 * n + 1)) {
        return true;
    }
    return dfs(3 * n + 1);
}
```

常用模板（二）：回溯枚举

```
void dfs(int pos) {
    if (pos == n) {                  // 1. 边界
        output();
        return;
    }
    for (int v : candidates) {
        if (!can_choose(v)) {
            continue;                // 2. 剪枝
        }
        choose(v);                   // 3. 做选择
        dfs(pos + 1);                // 4. 进入下一层
        undo(v);                     // 5. 恢复现场
    }
}
```

- 边界、候选、判断、选择、撤销，是回溯代码的五个固定位置。

常用模板（三）：记忆化

```
T dfs(int n) {
    if (vis[n]) {                   // 查表
        return f[n];
    }
    vis[n] = true;                  // 标记
    T ans = init();
    for (int v : next(n)) {
        ans = merge(ans, dfs(v));
    }
    return f[n] = ans;              // 写回表中
}
```

- 先查表，再标记，计算完写回；每个状态只真正计算一次。

复杂度速查表

问题	递归关系	复杂度
阶乘	$T(n) = T(n-1) + O(1)$	$O(n)$
朴素菲波那契	$T(n) = T(n-1) + T(n-2)$	$O(2^n)$
记忆化菲波那契	每个状态一次	$O(n)$
汉诺塔	$T(n) = 2T(n-1) + 1$	$O(2^n)$
全排列	n 层，每层 n 个分支	$O(n \cdot n!)$
集合的划分	二维状态 $S(n,k)$	$O(nk)$
数的计数	$f(n) = 1 + \sum_{i \le n/2} f(i)$	$O(n^2)$，前缀和 $O(n)$
归并排序	$T(n) = 2T(n/2) + O(n)$	$O(n \log n)$
二分查找	$T(n) = T(n/2) + O(1)$	$O(\log n)$

本页要点

写出递归式后，先估复杂度，再决定是否需要记忆化或改递推。

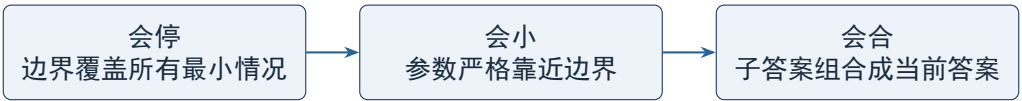
提交前的递归自查清单

1. 边界是否覆盖所有最小输入？返回值和题目要求一致吗？
2. 每层参数是否严格向边界靠近？有没有可能死递归？
3. 回溯状态是否在返回后恢复？多个分支之间是否互相污染？
4. 递归深度是否在栈空间允许范围内？需要改显式栈吗？
5. 是否存在重复子问题？需要记忆化吗？
6. 答案范围是否需要 `long long`、128 位整数或高精度？
7. 调试输出、计时输出是否已经删除？数组大小是否足够？

提交前最后一眼

拿最小的 3 组输入手推：$n = 0$ 或 1、边界附近、普通小数据；三组都对，再提交。

本章总结：三句话记住递归



- 执行模型：每次调用压一个栈帧，返回时弹栈；深度就是同时存活的栈帧数。
- 性能模型：递归树看调用次数；重复子问题用记忆化；深度过大改递推或显式栈。
- 回溯模型：做选择 $\rightarrow$ 进入下一层 $\rightarrow$ 撤销选择；恢复现场是正确性的底线。
- 工程习惯：边界优先、参数化状态、关键行注释、先小数据手推再提交。

本页要点

递归是把“问题的结构”直接写进代码；三要素保证它正确，调用栈和递归树帮助我们分析它。

本章练习题索引 | 1–7 / 14

1. P5197 | 逆波兰表达式（一本通 1198）
2. P5198 | 全排列（一本通 1199）
3. P5199 | 分解因数（一本通 1200）
4. P5200 | 菲波那契数列（一本通 1201）
5. P5186 | Pell 数列（一本通 1202）
6. P5201 | 扩号匹配问题（一本通 1203）
7. P5202 | 爬楼梯（一本通 1204）

点击蓝色题名进入学校 OJ；本章截图中的练习全部列在索引中。

本章练习题索引 | 8–14 / 14

8. P5203 | 汉诺塔问题（一本通 1205）
9. P5204 | 放苹果（一本通 1206）
10. P5205 | 求最大公约数问题（一本通 1207）
11. P5206 | 2 的幂次方表示（一本通 1208）
12. P5207 | 分数求和（一本通 1209）
13. P5208 | 因子分解（一本通 1210）
14. P5209 | 判断元素是否存在（一本通 1211）

点击蓝色题名进入学校 OJ；本章截图中的练习全部列在索引中。