DAMONTHRONE / CSP-SROUND1

0x3D

CSP-S 第一轮

提高组初赛专项教材

选择题・程序阅读・完善程序

S1

目录

第一编 考试方法 **1**

0x01 CSP-S 第一轮考什么 2

0x02 选择题怎么做 6

0x03 程序阅读怎么读 11

0x04 完善程序怎么补 20

第二编 复杂度与程序行为 **28**

0x05 时间复杂度 29

0x06 递归与递推 42

0x07 分治 54

0x08 二分 64

0x09 排序与比较次数 75

第三编 数学与计数 **85**

0x0A 进制与数位 86

0x0B 排列与组合 103

0x0C 容斥 115

0x0D 概率与期望 126

0x0E 递推数列 137

0x0F 模运算与逆元 152

0x10 gcd 与 lcm 168

第四编 数据结构 **182**

0x11 栈与队列 183

0x12 堆 198

0x13 哈希表 212

0x14 二叉树 228

0x15 二叉搜索树 248

0x16 Trie 266

0x17 线段树 281

0x18 RMQ 与 ST 表 298

第五编 树与图 **315**

0x19 图的基本概念 316

0x1A 连通与强连通 325

0x1B 欧拉图与二分图 . . . 334
0x1C DAG 与拓扑排序 . . . 342
0x1D 最小生成树 . . . 352
0x1E 最短路 . . . 362
0x1F 状态最短路与次短路 . . . 374
0x20 LCA . . . 385
0x21 树的重心与树形结构 . . . 394

第六编 字符串与序列 . . . 405

0x22 字符串匹配 . . . 406
0x23 KMP . . . 416
0x24 字符串哈希 . . . 427
0x25 LCS 与子序列 . . . 438
0x26 编码类程序阅读 . . . 449
0x27 相邻交换与逆序思想 . . . 462

第七编 DP、贪心与搜索 . . . 472

0x28 DP 的状态与转移 . . . 473
0x29 背包 . . . 487
0x2A 最大子段和 . . . 500
0x2B 子序列 DP . . . 512
0x2C 贪心 . . . 523
0x2D DFS 与回溯 . . . 536
0x2E 折半搜索 . . . 552

第八编 程序阅读专项 . . . 564

0x2F 变量追踪 . . . 565
0x30 循环程序 . . . 575
0x31 递归程序 . . . 588
0x32 数组与字符串程序 . . . 603
0x33 树图程序 . . . 616
0x34 综合程序阅读 . . . 636

第九编 完善程序专项 . . . 657

0x35 初始化与边界 . . . 658
0x36 二分与查找 . . . 672
0x37 排序与归并 . . . 689
0x38 DP 程序填空 . . . 707
0x39 图论程序填空 . . . 724
0x3A 搜索与递归填空 . . . 746

第十编 综合训练 . . . 767

0x3B 选择题分类训练 . . . 768
0x3C 程序阅读分类训练 . . . 776

0x3D 完善程序分类训练 783

附录 **792**

附录 A Linux 命令 793

附录 B C++ 运算符优先级速查 796

附录 C 复杂度速查 799

附录 D 数据结构速查 803

附录 E 图论速查 808

附录 F 数学公式速查 813

附录 G 考前检查表 817

第一编 考试方法

0x01 CSP-S 第一轮考什么

CSP-S 第一轮不是单纯考会不会写代码，也不是只考背过多少算法名。它更像一次纸面上的程序能力检查：给出概念、代码、过程、边界和选项，要求考生在有限时间内判断程序会怎样运行、算法会怎样变化、答案应当落在哪个选项里。

第二轮主要看能否把完整算法写成程序；第一轮则更重视另一种能力：能否在不运行程序的情况下读懂代码、还原过程、识别陷阱、手算结果。这些能力并不比写程序轻松。很多学生在第二轮能写出一些题，但第一轮仍然会丢分，原因常常不是不会算法，而是没有形成纸面推演的习惯。

本书从这一章开始，先把第一轮到底考什么讲清楚。后面的章节会按知识点和题型逐步展开，本章只负责建立整体认识。

第一轮的三类任务

近年 CSP-S 第一轮的题型大体稳定，主要由三部分组成：

题型	常见分值	主要考查
单项选择题	约 30 分	概念、计算、复杂度、数据结构、数学、图论、程序行为
程序阅读题	约 40 分	读代码、跟踪变量、判断输出、分析修改代码后的影响
完善程序题	约 30 分	根据题意和代码框架补全关键语句

三类题看起来差别很大，底层要求却有共通处：都要求考生把 “文字描述”“代码结构” 和 “算法过程” 连起来。

选择题常常把知识点压缩成一个小问题。例如给出哈希函数和插入顺序，要求判断最后落在哪个位置；给出递归式，要求判断复杂度；给出一个图，要求判断最短路、拓扑排序或连通性。题面短，计算量不一定小。

程序阅读题会给出一整段程序。题目不要求写代码，而是要求看懂代码在做什么。判断题可能问某个输入下输出是否正确，选择题可能问某个变量的范围、某行删除后的影响、某个函数的时间复杂度。它考的是 “在纸上运行程序” 的能力。

完善程序题通常先给出问题背景，再给出一段接近完整的代码，其中若干位置留空。选项可能只差一个下标、一个比较符号、一次更新顺序或一个边界条件。填空时不能只盯空格附近的一行，而要看清楚整个流程。

这三类题不是彼此独立的。选择题中的二分、分治、DP、图论，可能在程序阅读题中变成一段代码；程序阅读中的变量含义和边界判断，也可能在完善程序中成为填空关键。第一轮复习不能只背结论，要训练 “看见代码后能还原意图”。

选择题考的不是零散常识

选择题分值看起来不高，但覆盖面很广。它常见的考查方式可以分成四类。

第一类是基础概念题。例如 Linux 命令、数据结构定义、排序稳定性、二叉树性质、图的概念。这类题不难在于题面长短，而在于概念必须准确。把 “队列先进先出” 和 “栈后进先出” 混淆，把 “强连通” 和 “完全图” 混淆，都会直接失分。

第二类是手算题。例如进制转换、排列组合、容斥计数、期望、相邻交换次数、哈希线性探查、堆的插入删除、二叉搜索树遍历。这类题通常不需要写程序，但需要把过程列清楚。只凭感觉选答案，常常会被相近选项干扰。

第三类是复杂度题。例如双重循环、递归式、分治过程、二分查找、快速幂、朴素递归和动态规划的区别。复杂度题的关键不是记住某个算法名称，而是看清楚每一步执行了多少次，规模怎样变化。

第四类是算法性质题。例如 DAG 是否存在拓扑序、欧拉图的条件、二分图染色、最小生成树、最短路、LCA、树的重心、字符串匹配、LCS。这类题通常不会展开成完整复赛题，但会要求考生知道基本性质，并能在小规模图或序列上推演。

选择题最怕两种做法：一种是只背结论，不会用；另一种是每题都从头乱算，没有方法。后面的章节会把高频知识点拆开讲，每个知识点都尽量给出适合纸面考试的手算方式。

程序阅读考的是运行过程

程序阅读题的代码往往并不短，但它不会无缘无故给出一大段程序。每段程序都有核心结构，读题时要尽快找到它。

有些程序围绕循环展开。此时要关注循环变量的初值、终止条件、每轮更新、数组下标是否越界、某个判断会不会跳过一轮。

有些程序围绕递归展开。此时要关注递归入口、终止条件、参数如何变化、返回值如何合并，以及全局变量是否在递归前后被修改。

有些程序围绕数组或字符串处理展开。此时要关注下标从 0 开始还是从 1 开始，长度变量代表字符数还是最后下标，循环是否包含右端点，字符串是否按块处理。

有些程序围绕数据结构展开。例如栈、队列、堆、哈希表、树、图。此时要先确定数据结构中保存了什么，再看每次插入、删除、查询对状态造成什么影响。

程序阅读不是把代码从第一行读到最后一行就结束。读完之后，还要能回答三个问题：

问题	作用
这段程序输入什么，输出什么？	判断程序整体功能
哪些变量决定程序状态？	找到追踪重点
哪些语句改变答案或流程？	找到设题位置

例如遇到 DFS 程序，不能只说“它是搜索”。要进一步看它枚举了什么、哪些选择被禁止、递归到哪一层计数、回溯时恢复了哪些变量。题目如果问删去某一行是否影响答案，就要看这行是否负责恢复状态、剪枝或更新计数。

程序阅读题的高分来自稳定的跟踪方法。后面第 3 章会先讲通用读法，第八编还会专门训练变量追踪、循环程序、递归程序、数组字符串程序和树图程序。

完善程序考的是代码意图

完善程序题常被误解为“选项题”。其实它考的不是碰运气选一项，而是能否从题意和已有代码中推出空缺处应该做什么。

一个空缺位置通常承担下列某一种职责：

空缺职责	常见表现
初始化	给数组、队列、答案或边界赋初值
判断	决定循环是否继续，或某个状态是否合法
转移	从旧状态推出新状态
更新	改变答案、距离、计数、指针或集合
输出	选择最终答案或还原方案

做完善程序题时，先不要急着比较选项。更可靠的顺序是：

先读题意，确认程序要解决什么问题；再读变量定义，给每个关键变量写出含义；接着找主循环或核心函数，判断算法流程；最后回到空缺处，看它在流程中承担哪一种职责。

例如一个二分程序的空缺，可能只差 `<` 和 `<=`；一个图论程序的空缺，可能只差 `d[u]` 和 `d[v]`；一个 DP 程序的空缺，可能只差上一层状态和当前层状态。选项看似相近，其实只有一个能和上下文完全对齐。

完善程序题还有一个特点：题面常常先给出算法描述，再给代码。算法描述不是装饰，它通常会提示状态、转移、边界或答案形式。读代码时要不断把两者对照：

题意中的信息	代码中的对应
起点、终点、初始状态	初始化语句
每次操作、每条边、每个选择	循环或转移
最小、最大、计数、可行性	答案变量或状态含义
限制条件	判断语句
输出要求	最后一段代码

如果代码框架和题意对不上，空缺处就很难填对。后面第 4 章会先讲完善程序的通用方法，第九编再按初始化与边界、二分与查找、排序与归并、DP、图论、搜索递归分别训练。

第一轮需要哪些能力

CSP-S 第一轮需要的知识很多，但不能把复习变成“见一个名词背一个名词”。更重要的是建立几种稳定能力。

第一是概念辨析能力。能分清相近概念，例如子序列和子串、连通和强连通、稳定排序和不稳定排序、完全二叉树和满二叉树、最短路和最小生成树。

第二是纸面计算能力。能在小规模数据上算出哈希位置、堆变化、遍历序列、组合数量、复杂度量级、DP 表的一部分结果。

第三是代码跟踪能力。能在程序中找到关键变量，按输入逐步跟踪状态变化，而不是被代码长度吓住。

第四是边界判断能力。能看出循环是否少算或多算一个位置，二分边界是否闭合，数组下标是否匹配，递归终止条件是否正确。

第五是算法意图识别能力。看到一段代码时，能判断它大致属于二分、分治、搜索、DP、贪心、图论或字符串处理中的哪一类，并进一步说明它维护了什么。

第六是选项排除能力。第一轮有大量选择题，完全手算每个选项会消耗很多时间。正确的排除不是猜，而是利用单位、范围、奇偶性、上下界、变量含义和特殊输入快速排除不可能选项。

本书后面的知识点章节，都会尽量围绕这些能力来写。该讲性质时讲性质，该训练手算时训练手算，该读代码时读代码，不把复赛模板原封不动搬进第一轮教材。

本书怎么用

本书的顺序不是随意排列的。

第一编先讲考试方法。第 1 章建立整体认识，第 2 章讲选择题，第 3 章讲程序阅读，第 4 章讲完善程序。先学方法，是为了避免后面只看知识点却不会用在试卷上。

第二编到第七编按知识点展开。复杂度、递归、分治、二分、数学、数据结构、树图、字符串、DP、贪心、搜索，都是第一轮反复出现的内容。每章不会追求把算法讲到复赛最高难度，而是围绕第一轮会怎样考来组织。

第八编和第九编是专项训练。到这里时，前面的知识点已经学过，重点不再是重新讲一遍算法，而是训练题型能力：如何读程序，如何补程序。

第十编是综合训练。它用于把分散知识重新合并起来，适合在完成主体学习后使用。

附录用于考前速查，不替代正文学习。运算符优先级、复杂度表、数据结构操作、图论概念、数学公式，可以在复习后期快速翻看；如果正文没有学扎实，只看附录很难解决程序阅读和完善程序中的综合问题。

第一轮复习要避免两种偏差。

一种偏差是只刷选择题。这样短期内可能记住一些结论，但遇到程序阅读和完善程序时仍然缺少推演能力。

另一种偏差是按复赛方式学习所有算法。第一轮并不要求写出每个算法的完整模板，也不要求掌握过深的变形。对于很多知识点，知道基本性质、能手算小例子、能读懂核心代码，比背一份长模板更有用。

比较合适的学习节奏是：先读正文，理解知识点；再做例题，检查是否会手算；再做变式题，辨析细节；最后回到真题，看这个知识点在试卷中如何出现。

第一轮考试的难点，不在于每道题都很深，而在于知识点多、题型变化快、细节要求高。复习时要把“会一点”变成“能在纸上稳定做对”。这也是本书后面所有章节的写作目标。

0x02 选择题怎么做

选择题是 CSP-S 第一轮的第一部分。它的题面通常不长，分值也比较分散，但并不只是“送分题”。选择题的难点在于覆盖面广、节奏快、选项相近，很多题只要一个概念混淆、一个下标算错、一个边界漏掉，就会选到干扰项。

做选择题时，目标不是把每道题都写成完整题解，而是在较短时间内给出稳定判断。稳定判断来自三件事：看清题目类型，留下必要的手算过程，用选项帮助校验结果。

本章先讲选择题的通用方法。具体知识点会在后面各章展开，这里只解决“拿到选择题时怎么下手”。

先判断题目类型

选择题不能都用同一种做法。遇到题目后，先判断它属于哪一类，再决定作答方式。

常见选择题大致可以分为五类。

类型	题面特征	作答重点
概念判断	问“以下说法正确/错误的是”	抓关键词，排除混淆概念
纸面计算	给出小规模数据，要求算答案	逐步记录过程，避免心算跳步
复杂度分析	给出代码、递归式或算法过程	看循环次数、规模变化、合并代价
程序片段	给出几行代码，问结果或行为	按运算顺序和变量变化推演
算法性质	涉及树、图、字符串、DP、贪心等	用定义和小例子验证

题目类型不同，错误来源也不同。

概念判断题常错在“差一个词”。例如“完全图”“连通图”“强连通图”“欧拉图”都有固定条件，不能只凭印象判断。

纸面计算题常错在“漏一步”。哈希线性探查、堆插入删除、相邻交换、二叉搜索树遍历，都需要按过程走完。

复杂度题常错在“只看外形”。双重循环不一定是 $O(n^2)$，递归调用两次也不一定是 $O(2^n)$，要看循环变量怎样变化、递归规模怎样缩小。

程序片段题常错在“运算顺序”。逻辑运算、位运算、括号、短路、整数除法、取模，这些细节会直接影响结果。

算法性质题常错在“把结论用错范围”。例如最小生成树只适用于连通无向图，拓扑排序只适用于有向无环图，二分查找要求序列满足可判定的单调性。

读题后的第一步可以很短：在心里给题目贴一个标签。它是概念题、计算题、复杂度题、代码题，还是性质题？标签一旦确定，解题路线就会清楚很多。

概念题抓住限定词

概念题不一定考深，但要求准确。很多选项不是完全胡说，而是把一个概念中的限定条件换掉。

例如：

- “队列先进先出”不能换成“后进先出”；
- “二叉树每个结点最多有两个子结点”不能换成“每个结点恰好有两个子结点”；
- “DAG 没有有向环”不能换成“没有无向环”；

- “稳定排序保持相等元素的相对顺序”不能换成“排序后结果唯一”。

做概念题时，不要先看选项中哪个最熟悉，而要先回到定义。定义里通常有几个关键限定词：

限定词	需要注意
任意	不能只举一个成立例子
存在	找到一个例子即可
至少、至多	边界方向不能反
有向、无向	图论题中差别很大
连通、强连通	适用对象不同
稳定、不稳定	只讨论相等元素的相对顺序
最坏、平均、最好	复杂度场景不同

有些题问“不正确的是”，有些题问“一定正确的是”，这两个问法差别很大。看到否定词时，应当在题干上做标记。纸质考试中，建议把“不”“不一定”“至少”“至多”“最坏”等词圈出来，防止读题时自动忽略。

概念题还可以用反例排除。若选项说“某性质一定成立”，只要能构造一个不成立的小例子，就能排除。反例不需要大，通常 3 到 5 个点、3 到 6 个数就够。

计算题要留下痕迹

选择题中的计算题不适合全靠心算。心算太快时，错了也不知道错在哪。纸面上留下少量痕迹，可以显著减少失误。

例题 1：线性探查

有一个长度为 7 的哈希表，下标为 0 到 6，哈希函数为 $h(x) = x \bmod 7$。采用线性探查解决冲突，依次插入 $10, 17, 24, 3$，问 3 最终放在哪个位置？

先算初始位置：

数	初始位置
10	3
17	3
24	3
3	3

再按顺序插入：

步骤	操作	表中被占位置
插入 10	放入 3	3
插入 17	3 冲突，放入 4	$3, 4$
插入 24	$3, 4$ 冲突，放入 5	$3, 4, 5$
插入 3	$3, 4, 5$ 冲突，放入 6	$3, 4, 5, 6$

答案是 6。

这类题如果只在脑中想“都冲突往后走”，很可能少走一次。表格不必完整画出所有空位，只要记下已经占用的位置即可。

例题 2：相邻交换

把字符串 `DACFEB` 变成 `ABCDEF`，每次只能交换相邻两个字符，问最少需要多少次？

相邻交换次数可以转成逆序数。把原字符串中每个字符按目标串中的位置编号：

字符	D	A	C	F	E	B
目标位置	4	1	3	6	5	2

于是要计算序列 $4,1,3,6,5,2$ 的逆序数。

逐个数：

- 4 后面比它小的有 $1,3,2$，共 3 个；
- 1 后面没有比它小的；
- 3 后面比它小的有 2，共 1 个；
- 6 后面比它小的有 $5,2$，共 2 个；
- 5 后面比它小的有 2，共 1 个。

总数为 $3+1+2+1=7$。

这类题的关键不是模拟每一次交换，而是看清“相邻交换最少次数”和“逆序数”的对应关系。后面讲相邻交换与逆序思想时会系统处理。

复杂度题先看规模怎样变

复杂度题不能只看代码有几层缩进。更可靠的做法是问两个问题：

第一，单层工作量是多少？

第二，规模变成了什么？

例如递归式 $T(n)=2T(n/2)+O(n)$，每一层总工作量大致是 $O(n)$，层数是 $O(\log n)$，总复杂度是 $O(n\log n)$。

如果递归式变成 $T(n)=2T(n/2)+O(n^2)$，每层的合并代价不再一样。第一层是 n^2，下一层两份 $(n/2)^2$，合起来是 $n^2/2$，再下一层是 $n^2/4$。这些量相加后仍由前几层主导，总复杂度是 $O(n^2)$。

这说明，不能看到“分成两半”就直接选 $O(n\log n)$，也不能看到“两次递归”就直接选指数级。

例题 3：循环变量变化

判断下面代码的时间复杂度：

```
int cnt = 0;
for (int i = 1; i <= n; i *= 2) {
    for (int j = 1; j <= n; j++) {
        cnt++;
    }
}
```

外层 i 依次为 $1,2,4,8,\ldots$，执行 $O(\log n)$ 次。内层每次执行 n 次，所以总复杂度是 $O(n\log n)$。

若改成：

```
int cnt = 0;
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= i; j *= 2) {
        cnt++;
    }
}
```

外层执行 n 次。对于固定的 i，内层执行 $O(\log i)$ 次，总次数是

$$\log 1 + \log 2 + \cdots + \log n.$$

在选择题中不需要精确求和，只要知道它不超过 $n \log n$，且大体也有这个量级，就可以判断为 $O(n \log n)$。

复杂度题中，常见的规模变化有这些：

形式	常见量级
每次 $+1$	线性次数
每次 $\times 2$	对数次数
每次除以 2	对数层数
两层都到 n	常见为 $O(n^2)$
内层到 i	常见为 $O(n^2)$
排序	常见为 $O(n \log n)$
枚举所有子集	常见为 $O(2^n)$
枚举全排列	常见为 $O(n!)$

这些只是初步判断，具体题目仍要结合代码。第一轮常把边界、异常循环、重复递归调用放进题目里，不能只靠外形判断。

代码题按顺序推演

选择题中有时会给一小段 C++ 代码，问输出、变量值或表达式结果。这类题应按 C++ 的规则推演，不能按数学表达式的习惯处理。

需要特别注意的细节包括下面几类：

- 整数除法：`5 / 2` 的结果是 `2`。
- 取模：负数取模要按 C++ 表达式推演。
- 自增自减：注意前置和后置差别。
- 逻辑短路：`&&` 左侧为假时，右侧不再执行；`||` 左侧为真时，右侧不再执行。
- 运算优先级：不确定时先加括号分析。
- 数组下标：看清从 0 开始还是从 1 开始。
- 局部变量遮蔽：内层变量可能与外层同名，但不是同一个变量。

例题 4：变量遮蔽

```cpp
int sum = 0;
for (int i = 1; i <= 5; i += 2) {
    sum += i;
    int sum = 0;
}
cout << sum;
```

循环中有两次与 `sum` 有关的语句。第一句 `sum += i` 使用的是循环外的 `sum`，因为此时内层的 `sum` 还没有声明。第二句 `int sum = 0` 声明了一个新的局部变量，它只在本轮循环的花括号内部有效，不会把外层 `sum` 清零。

外层 `sum` 依次加上 $1, 3, 5$，最终输出 9。

这类题如果只看见里面有 `int sum = 0`，就可能误以为总和每轮都清零。代码题要按语句执行顺序处理。

选项不是摆设

选择题的选项很有用。它们不仅用于最后选择，也可以反过来帮助判断解题方向。

如果四个选项差距很大，可以先估范围。例如计数题选项是 $6, 30, 120, 720$，题目又要求“不相邻”“不区分颜色”等，通常不需要一开始就精确计算，可以先判断是否接近全排列、组合数还是插空法结果。

如果四个选项很接近，例如 $13, 14, 15, 16$，就说明题目可能在某个边界上设陷阱。此时要少用估算，多做完整过程。

如果选项中有两个表达式只差一个等号，例如 `< k` 和 `<= k`，题目多半考边界。此时应找一个刚好等于边界的小例子，代入检验。

不确定时先保分

第一轮考试时间有限，不可能每道选择题都反复推敲。遇到不确定的题，要学会保分。

第一步，先判断是否有明显错误选项。概念题中，凡是把适用范围说得过满的选项，要重点检查。计算题中，凡是数量级不可能的选项，可以先排除。

第二步，找特殊情况。若题目问“一定成立”，就试最小规模、空结构、链状结构、完全结构、全相等、全递增、全递减等情况。很多错误选项会在特殊情况下暴露。

第三步，看单位和范围。例如概率不能超过 1，计数不能为负，最短路不能小于所有边权下界，树上结点数和边数有固定关系，哈希地址不能超过表长范围。

第四步，做标记后继续。若一道题已经花了较长时间仍不能确定，不要继续拖住整张卷子。可以在题号旁做标记，先完成后面题目，再回来处理。程序阅读和完善程序分值更高，不能让一道选择题占用过多时间。

保分不是乱猜，而是把不确定题的损失控制住。第一轮成绩往往由许多小分组成，稳定拿下会做的题，比在某一道难题上耗太久更重要。

训练时要按类别复盘

选择题训练不能只统计对错。做完一组题后，应当给错题分类。

错误类型	复盘方式
概念混淆	回到定义，写出相近概念差异
手算失误	补上过程表，找漏算位置
边界错误	构造最小例子，检查等号方向
复杂度误判	写出循环次数或递归层数
代码规则不熟	查清 C++ 语法规则，补一个小例子
读题失误	圈出被忽略的关键词

错题复盘不要只写“粗心”。“粗心”这个说法太宽，不能指导下一次改进，更具体的记录才有用。

选择题的提升，靠的是把同类错误逐渐消掉。错题越具体，下一次越能避免。

小结

选择题不是整张试卷中最难的部分，但它是进入状态的第一部分。开头做得稳，后面的程序阅读和完善程序也会更从容。后面几章会分别讲程序阅读和完善程序的方法；从第二编开始，再把选择题中反复出现的知识点逐章拆开。

0x03 程序阅读怎么读

程序阅读题要求考生在不能运行程序的情况下，判断程序的行为。它不是单纯考语法，也不是让考生把每一行代码都翻译成中文。真正要做的事情，是在纸面上还原程序的执行过程：输入怎样进入变量，循环怎样推进，递归怎样展开，数组怎样变化，最终输出怎样得到。

很多学生做程序阅读题时，会有两种不好的习惯。

一种是从第一行开始逐字读，到最后已经忘记前面变量的作用。另一种是看到熟悉的代码片段就急着判断它属于某个算法，忽略了题目中改过的边界、变量和输出。程序阅读题常常就考这些细节：同样像二分，可能问的是边界；同样像 DFS，可能问的是回溯；同样像 DP，可能问的是初始化。

程序阅读要有顺序。先看程序整体，再看关键变量；先读结构，再追数据；先判断功能，再处理具体小问。这样读代码，速度不一定最慢，出错反而会少很多。

先看程序做什么

拿到一段程序，不要急着从头模拟。先用较短时间判断它大致在做什么。

可以按下面的顺序看：

- 看输入：读入了几个数、几个数组、几个字符串，是否有多组数据。
- 看输出：输出一个数、一个字符串、两行结果，还是输出中间过程。
- 看全局变量：哪些变量在多个函数中被使用。
- 看核心函数：主函数调用了哪些函数，函数名和参数能提供哪些信息。
- 看主循环或递归入口：程序主要时间花在哪里。
- 看答案变量：哪个变量最后被输出，在哪里被修改。

这一步的目标不是把全部细节都弄清楚，而是给程序建立一个框架。

例如看到下面这段程序：

```
int ans = 0;
bool vis[20];

void dfs(int k) {
    if (k == n + 1) {
        ans++;
        return;
    }
    for (int i = 1; i <= n; i++) {
        if (vis[i]) continue;
        vis[i] = true;
        dfs(k + 1);
        vis[i] = false;
    }
}
```

先不需要立刻展开所有递归。读到这里，可以判断：

- `dfs(k)` 表示正在填第 k 个位置；

- `vis[i]` 表示数字 i 是否已经用过；
- 到 `k == n + 1` 时说明已经填完一种方案；
- `ans` 统计方案数；
- `vis[i] = false` 是回溯恢复。

有了这个框架，再去看题目问 `n=3` 时输出多少，或删掉某一行会怎样，才不会乱。

程序阅读的第一步，是把代码从“一堆语句”变成“几个部分”：输入、状态、过程、答案。

给变量写含义

程序阅读题中，变量名不一定友好。真题代码里经常出现 `i,j,k,l,r,t,ans,cnt,vis,f,g` 这样的变量。只看名字，很难知道它们的作用。读代码时要主动给变量写含义。

建议在草稿纸上列一个简短的变量表。变量表不必完整，但关键变量一定要写。

变量	含义	何时改变
`n`	数据规模	输入时确定
`ans`	当前答案或方案数	满足条件时更新
`i`	当前枚举对象	循环中改变
`vis[i]`	第 i 个对象是否已使用	选择和回溯时改变
`f[i]`	第 i 个状态的值	转移时改变

表里的“何时改变”很重要。程序阅读题经常问某个变量的范围、某行删除后的影响、某个输入下答案是多少。只知道变量含义还不够，还要知道它在哪里被修改。

同一个变量在不同程序中含义可能完全不同。`cnt` 可能是计数，也可能是数组长度；`p` 可能是父亲数组，也可能是排列；`f` 可能是函数，也可能是 DP 数组；`k` 可能是输入参数，也可能是递归层数。不要把以前见过的含义直接套进当前程序。

例题 1：变量含义先于模拟

阅读下面程序片段：

```
int n, ans;
int p[10];
bool used[10];

void dfs(int k) {
    if (k == n + 1) {
        ans++;
        return;
    }
    for (int i = 1; i <= n; i++) {
        if (used[i]) continue;
        if (k > 1 && i == p[k - 1] + 1) continue;
        p[k] = i;
        used[i] = true;
        dfs(k + 1);
        used[i] = false;
    }
}
```

先写变量含义：

变量	含义
`k`	当前要填写的位置
`i`	尝试放入当前位置的数字
`p[k]`	第 k 个位置放的数字
`used[i]`	数字 i 是否已经用过
`ans`	满足限制的排列数量

限制条件有两条：

- 一个数字不能重复使用；
- 如果不是第一个位置，当前位置不能放 `p[k - 1] + 1`。

因此这段程序不是单纯统计全排列，而是在统计满足相邻限制的排列。若没有先写变量含义，很容易只看到 `dfs` 和 `used` 就误判为 $n!$。

追踪要选小输入

程序阅读题常给出具体输入，要求判断输出。此时需要模拟，但不能把所有分支都铺开。应当选择合适的来追踪。

追踪时有三个原则。

第一，输入规模小，就完整列状态。比如 `n=3`、字符串长度为 4、数组长度为 5，可以逐步写出变量变化。

第二，输入规模稍大，就只追关键变量。比如排序、二分、哈希、堆操作，不必写出所有无关变量，只记录数组、边界、堆顶或当前位置。

第三，程序结构复杂时，先追一次主流程，再回答小问。不要每个小问都从头读一遍程序。

例题 2：跟踪递归计数

沿用上一节的 DFS 程序，设 $n = 3$，统计满足限制的排列数量。

所有 $1, 2, 3$ 的排列共有 6 个：

排列	是否满足限制	原因
$1, 2, 3$	否	$2 = 1 + 1$
$1, 3, 2$	是	相邻位置没有出现后一项等于前一项加 1
$2, 1, 3$	是	满足限制
$2, 3, 1$	否	$3 = 2 + 1$
$3, 1, 2$	否	$2 = 1 + 1$
$3, 2, 1$	是	满足限制

所以答案为 3。

这里没有必要画出完整递归树。因为 $n = 3$ 很小，直接列出排列更清楚。程序阅读不是必须按程序的真实展开形式记录，只要记录方式与程序逻辑一致，并且不漏情况即可。

若 $n = 4$，完整列出 24 个排列就比较费时。此时可以按第一位分组，或写一个简短的递归计数表。追踪方法要随规模调整。

循环看边界

循环程序是程序阅读题最常见的结构。读循环时，要看四个位置：

- 循环变量初值；
- 循环继续条件；
- 每轮结束后的更新；
- 循环体中是否修改了循环相关变量。

很多错误判断都来自边界。比如 `i < n` 和 `i <= n`，`j = i + 1` 和 `j = i`，`while (l < r)` 和 `while (l <= r)`，差别都很大。

读循环时，可以先写出循环变量实际取过哪些值。

例如：

```
for (int i = 1; i <= n; i *= 2) {
    cnt++;
}
```

当 $n = 10$ 时，`i` 取 $1, 2, 4, 8$，不取 16。

再如：

```
for (int i = n; i >= 1; i -= 2) {
    cnt++;
}
```

当 $n = 6$ 时，`i` 取 $6, 4, 2$；当 $n = 5$ 时，`i` 取 $5, 3, 1$。奇偶不同，次数可能不同。

例题 3：循环里改指针

```
int s = 0;
for (int i = 0, j = 0; i < n; i++) {
    while (a[i] - a[j] > m) j++;
    s += i - j;
}
```

这段代码中，`i` 每轮加 1，`j` 只在 `while` 中增加，不会减少。读它时，不能把内层 `while` 当成每次都从 0 开始。`j` 是延续上一轮的值。

因此，若数组 `a` 已经有序，这段代码常用于维护一个滑动窗口：对每个右端点 `i`，找到最小的左端点 `j`，使得区间内差值不超过 m。`s += i - j` 统计以 `i` 为右端点的某类二元组数量。

这种程序的关键变量不是只有 `i`，而是 `i,j,s` 三个变量之间的关系。若题目问第某行执行次数或函数复杂度，就要注意 `j` 总共最多增加 n 次，而不是每轮重新增加 n 次。

递归看入口和回收

递归程序要先看入口，再看终止条件，最后看递归之后有没有恢复操作。

常见递归有几类：

类型	关键问题
分治递归	每次分成几段，合并做了什么
DFS 枚举	每层选什么，何时计数
树上递归	子树结果怎样合并到父结点
记忆化搜索	状态是否已算过，返回值代表什么

读递归时，要特别注意递归前后变量是否恢复。例如：

```
used[i] = true;
dfs(k + 1);
used[i] = false;
```

这里的恢复不是可有可无。若删掉 `used[i] = false`，当前分支用过的数字会影响后面的分支，答案通常会改变。

但不是所有递归后都需要恢复。如果修改的是局部变量，或每次递归传的是值而不是引用，就可能不需要手动恢复。判断时要看变量的作用范围和传参方式。

例题 4：按值传递和按引用传递

```
void dfs1(int x) {
    x++;
    dfs1(x);
}
```

这里 `x` 是按值传递。每次调用有自己的 `x`，修改当前 `x` 不会改变上一层的 `x`。

再看：

```
void dfs2(vector<int>& a) {
    a.push_back(1);
    dfs2(a);
}
```

这里 `a` 是引用。递归中对 `a` 的修改会保留在同一个容器里。如果程序需要枚举不同分支，通常要在递归返回后恢复。

程序阅读题中，参数里出现 `&`，要立刻注意它表示引用。数组作为参数传递时，也要把它看成会影响原数组的对象。

数组和字符串看下标

数组和字符串题的错误常发生在下标上。读这类程序时，要先确认三件事：

- 下标从 0 开始还是从 1 开始；
- 循环右端点是否包含；
- 变量表示长度还是最后一个位置。

例如 C++ 字符串 `s` 的长度是 `s.size()`，合法下标是 0 到 `s.size() - 1`。如果循环写成：

```
for (int i = 0; i < s.size(); i++)
```

它访问的是所有字符。

如果写成：

```
for (int i = 1; i <= n; i++)
```

就要看数组是否按 1 到 n 存储。

字符串程序还常出现“窗口”“前缀”“后缀”“匹配位置”等概念。遇到这类程序，建议画出字符串下标：

```
下标：0 1 2 3 4 5
字符：a b a c a b
```

画出下标后，`s[i]`、`s[i + 1]`、`s[i + m]` 是否越界，匹配从哪里开始，就会清楚很多。

例题 5：子序列匹配

```
int cnt = 0;
int l1 = s1.size(), l2 = s2.size();
for (int i = 0, j = 0; i < l2; i++) {
    if (s2[i] == s1[j]) {
        cnt++;
        j++;
    }
}
if (cnt == l1) cout << "similar";
```

这段程序用 `i` 扫描长字符串 `s2`，用 `j` 表示短字符串 `s1` 中下一个要匹配的位置。每当字符相同，就让 `j` 后移。

它判断的是 `s1` 是否为 `s2` 的子序列，而不是子串。子序列只要求相对顺序一致，不要求连续。

读这段代码时要注意一个细节：若 `j` 已经等于 `l1`，继续访问 `s1[j]` 会超出短字符串范围。真实考试中题目可能保证输入不触发这个问题，也可能把它作为判断点。程序阅读不能只看算法意图，还要看边界是否被代码正确处理。

输出题先别急着算答案

程序阅读题经常问“输入为某某时，输出是什么”。做这类题时，先确认程序到底输出几行、每行输出什么格式，再算具体数值。

需要注意：

- 是否先输出调试信息；
- 是否输出空格；
- 是否输出换行；
- 是否固定小数位数；
- 是否输出两个函数的结果；
- 是否某些分支没有输出；
- 是否提前 `return`。

有些程序会在 `check` 函数里输出中间信息，有些程序会先输出一个数组，再输出答案。若只盯最后的答案变量，就可能漏掉前面的输出。

例如：

```
bool check(int x) {
    cout << "check " << x << endl;
    return x >= k;
}
```

若主函数多次调用 `check`，输出中会包含多行 `check x`。题目若问完整输出，就不能只写最终判断结果。

不过 CSP-S 第一轮的程序阅读选择题，很多时候问的是“猜测次数”“某一行输出”“某个变量值”，不一定要求写完整输出。读题时要看清楚问题范围。

修改代码题看影响范围

程序阅读题常问“删除某一行”“替换某个表达式”“修改某个条件”后，结果是否改变。做这类题时，不要只看被修改的那一行，要看它影响哪些变量和后续流程。

常见修改影响包括：

- 删除回溯恢复语句，导致后续分支状态被污染；
- 改变比较符号，导致边界多算或少算；
- 改变排序语句，导致后续双指针或二分失效；
- 改变初始化，导致默认状态不同；
- 改变数组下标，导致取到相邻位置或错误状态；
- 改变传参方式，导致函数内修改是否保留发生变化。

判断修改是否影响结果，可以用两步。

第一步，看这行语句是否会被执行。若某行在给定输入下不会执行，那么对这个输入的输出可能不受影响。

第二步，看被修改变量后面是否还会被读取。若修改了一个后面不再使用的变量，对输出可能没有影响；若修改了答案变量、状态数组、边界变量，通常要谨慎。

例题 6：删掉排序

```
sort(a.begin(), a.end());
int l = 0;
for (int r = 0; r < n; r++) {
    while (a[r] - a[l] > m) l++;
    ans += r - l;
}
```

如果删掉排序，后面的双指针逻辑通常就不能保证正确。因为 `a[r] - a[l]` 的变化依赖数组有序，`l` 只增不减也依赖有序性。

这种题不是问 `sort` 本身有没有输出，而是问后续代码是否依赖排序后的性质。读修改题时，要找“后面哪一段代码需要这一行提供的条件”。

判断题要特别小心

程序阅读中的判断题常见选项是“正确”或“错误”。它看起来比选择题简单，其实更容易被绝对化表述误导。

遇到判断题，重点看这些词：

- 一定；
- 总是；
- 不会；
- 可能；
- 任意；
- 所有；
- 至少；
- 最多；
- 不影响。

如果题目说“程序总是输出两行相等的数”，就要确认是否所有合法输入都成立，而不是只找一个成立例子。

如果题目说“删除某行不会影响输出”，就要考虑是否存在某个输入会受影响。只要存在一个输入使输出改变，这个判断就是错的。

判断题常用反例解决。反例要尽量小，便于纸面验证。例如：

- 要否定“所有数组都满足”，试长度 1、长度 2、全相等、递增、递减；

- 要否定“所有图都满足”，试孤立点、链、环、两个连通块；
- 要否定“所有字符串都满足”，试空匹配、长度为 1、重复字符、完全不匹配。

在第一轮里，判断题分值不一定高，但数量多。判断题稳，整张卷子的底分会更稳。

读不懂时先找结构

有些程序第一次看会很陌生，比如编码、哈希、复杂 DP、图上状态、折半搜索。遇到这种程序，不要因为不认识算法就放弃。程序阅读题的小问并不总是要求完全理解算法背景。

可以先找结构：

- 是否有初始化函数；
- 是否有预处理数组；
- 是否有排序；
- 是否有二分；
- 是否有递归；
- 是否有队列或优先队列；
- 是否有两段相似代码；
- 是否有一个变量最后被输出。

很多问题只需要局部理解。

例如一个编码程序很长，但题目问 `encode` 函数的时间复杂度，就只需要看它对字符串扫描了几次；一个图论程序很复杂，但题目问优先队列里存的是什么，就只需要看 `push` 和 `pop` 附近；一个 DP 程序状态很多，但题目问某个初始化是否正确，就先看初始状态的含义。

读不懂整段程序时，先拿小问切入。小问往往会提示程序的关键点。做完几个小问后，程序整体含义也会逐渐清楚。

草稿纸怎么用

程序阅读题的草稿纸不要写得太满。过大的表格会浪费时间，也容易把自己绕进去。

比较实用的记录有四类。

第一类是变量表。记录变量含义和修改位置。

第二类是状态表。用于小输入模拟，例如循环中 `i,j,ans` 的变化。

第三类是递归树或分组计数。用于 DFS、分治、递归计数，但只画必要部分。

第四类是数组下标图。用于字符串、数组、前缀和、双指针、区间处理。

例如追踪一个循环，可以写成：

轮次	`i`	`j`	`ans`	说明
1	0	0	0	初始
2	1	0	1	满足条件
3	2	1	2	`j` 后移

如果某一列始终不影响答案，就不要继续写。草稿的目标是辅助判断，不是复刻程序运行日志。

小结

程序阅读的能力不是靠背题得到的，而是靠一次次有顺序地读程序建立起来的。读代码时少一点跳步，多一点状态记录；少一点凭印象判断，多一点按语句推演。这样进入后面的递归、分治、二分、数据结构和图论程序时，才有稳定的抓手。

0x04 完善程序怎么补

完善程序题和程序阅读题很像，都给出一段较长代码，都要求考生读懂程序行为。但完善程序题多了一层要求：不仅要看懂已有代码，还要判断空缺处应该填入什么。

很多学生做完善程序题时，会先看空缺，再看选项，然后凭局部感觉作答。这种做法很容易掉进陷阱。完善程序题中的选项往往只差一个变量、一个下标、一个等号或一次更新顺序。若没有读懂整段程序的意图，只盯空缺附近几行，很难稳定选对。

完善程序题的核心不是“猜这一空像什么”，而是“判断这一空在整个算法流程里承担什么职责”。空缺可能负责初始化，可能负责判断，可能负责转移，可能负责更新，也可能负责输出。先判断职责，再比较选项，答案才会有依据。

本章只讲完善程序的通用方法。二分、排序、DP、图论、搜索等具体知识，会在后文分章展开。

先读题意，再读代码

完善程序题通常有一段较长题意，后面给出代码。题意不是背景介绍，而是代码的说明书。读代码之前，先把题意中的关键信息抓出来。

要抓四类信息：

- 输入是什么；
- 要求输出什么；
- 允许哪些操作或状态；
- 题目要求最小值、最大值、计数、判断可行性，还是还原方案。

例如题目说“求从起点到终点的最小费用”，代码中大概率会出现距离数组、松弛操作和优先队列。题目说“统计方案数”，代码中通常会出现计数变量或 DP 加法。题目说“找第 k 小”，代码中可能会出现二分答案、排名计算或归并过程。

读完题意后，再看代码。此时不要急着补空，而要先确认题意和代码如何对应。

可以在草稿纸上写一个简短对应表：

题意信息	代码位置
初始状态	初始化语句
一次操作	循环或转移
限制条件	if 判断
当前最优值	数组或答案变量
最终答案	输出语句

如果题意和代码没有对应起来，空缺就会变成猜选项。完善程序题一定要先让代码“有意义”。

给空缺定职责

每个空缺都在程序中承担某种职责。补空前，应先判断它属于哪一类。

常见空缺可以分为六类。

空缺类型	常见位置	判断重点
初始化	主循环之前	初始状态是否和题意一致
循环条件	while、for 中	是否会多做或少做一步
分支判断	if 中	条件方向和边界是否正确
状态转移	DP、搜索、图论松弛中	新状态从哪些旧状态来
变量更新	答案、指针、距离、计数	更新对象和更新顺序
输出结果	程序末尾	输出哪个状态或哪个答案

先定职责，再看选项。例如空缺位于初始化部分，就不该从“转移公式”角度乱选；空缺位于输出部分，就要回到题目要求判断最终答案形式。

例题 1：初始化空缺

某程序判断从起点 s 到其他点的最短距离，距离数组 `d[i]` 初始表示“尚未到达”。程序片段如下：

```
const int INF = 1000000000;
int d[N];

for (int i = 1; i <= n; i++) d[i] = INF;
_____ ;
```

选项：

A. `d[s] = INF`

B. `d[s] = 0`

C. `d[1] = 0`

D. `d[n] = 0`

空缺在初始化之后。根据题意，起点到自己的距离为 0，其他点暂时不可达。因此应填 `d[s] = 0`。

这道题不需要展开最短路算法。只要判断空缺职责是初始化，再对应“起点距离为 0”，就能选出答案。

看变量含义，不看变量名字

完善程序题中的变量名未必清楚，有时甚至会刻意使用很短的名字。变量名字只能提供参考，不能作为依据。补空时要从代码行为中确定变量含义。

例如下面代码：

```
int l = 0, r = n;
while (l < r) {
    int mid = (l + r) / 2;
    if (check(mid)) r = mid;
    else l = mid + 1;
}
cout << l << endl;
```

这里的 `l` 和 `r` 不只是普通左右端点。结合更新方式可知，程序在寻找最小的满足 `check(x)` 的位置。`check(mid)` 为真时保留 `mid` 和左侧，为假时排除 `mid` 及左侧。

若某个空缺出现在 `if (_____) r = mid; else l = mid + 1;` 中，就要判断 `check(mid)` 的含义是否表示“mid 已经可行”。若表示可行，条件应直接使用它；若表示“mid 还不够”，条件方向就要反过来。

变量表在完善程序题里很有用。可以写得很短：

变量	含义
`l`	当前答案下界
`r`	当前答案上界，可能成为答案
`mid`	本轮测试位置
`check(mid)`	`mid` 是否满足要求

空缺处填什么，取决于这些含义是否一致。

不要只看空缺附近

有些空缺附近的代码看起来很像某个模板，但真正答案要看上下文。

例题 2：二分中的边界

有两个有序数组 `a` 和 `b`，程序要找最小的 `x`，使得不超过 `x` 的配对和数量至少为 `k`。函数 `rank(x)` 返回配对和 `<= x` 的数量。

```
int l = 0, r = a[n - 1] + b[n - 1];
while (l < r) {
    int mid = ((long long)l + r) >> 1;
    if (_____) {
        l = mid + 1;
    } else {
        r = mid;
    }
}
return l;
```

选项：

A. `rank(mid) < k`

B. `rank(mid) <= k`

C. `rank(mid) > k`

D. `rank(mid) >= k`

程序寻找最小的 `x`，使得 `rank(x) >= k`。如果 `rank(mid) < k`，说明 `mid` 太小，答案必须在右边，所以执行 `l = mid + 1`。否则 `mid` 已经可能成为答案，执行 `r = mid`。

因此空缺应填 `rank(mid) < k`。

这道题不能只背“二分里常写 `check(mid)`”。要看 `rank(mid)` 返回的是什么，以及当前分支保留哪一边。

判断等号属于哪一边

完善程序题很喜欢考等号。`<`、`<=`、`>`、`>=` 的差别，往往决定答案。

判断等号时，可以问一句：当前情况“刚好相等”时，应该算成功还是失败？

例如：

- 找第一个满足条件的位置，刚好满足时要保留；
- 统计不超过 x 的数量，等于 x 的元素应计入；
- 判断是否超过限制，刚好达到限制可能仍合法；
- 判断数组下标是否越界，等于长度通常已经越界；
- 判断是否还有剩余，剩余为 0 时通常应停止。

例题 3：上界函数

```
int upper_bound(int *a, int *an, int x) {
    int l = 0, r = an - a;
    while (l < r) {
        int mid = (l + r) >> 1;
        if (_____) r = mid;
        else l = mid + 1;
    }
    return a + l;
}
```

这个函数应返回第一个大于 x 的位置。若 a[mid] > x，mid 可能就是答案，应保留左侧和 mid；否则 a[mid] <= x，应去右侧。

空缺应填：

```
a[mid] > x
```

这里等号不能放进条件。如果写成 a[mid] >= x，函数就变成了返回第一个大于等于 x 的位置，含义改变。

完善程序题中，函数名有时会给提示，但不能只靠函数名。还要看这个函数的返回值在后面怎样使用。

从输出倒推状态

有些完善程序题从前往后读很难补，但从输出往回看会清楚很多。

比如程序末尾有：

```
cout << min(d[t][0], d[t][1]) << endl;
```

这说明 d[t][0] 和 d[t][1] 都表示到达终点 t 的某类费用，只是状态不同。再回头看数组定义：

```
long long d[N][2];
```

如果题目中有“某个操作最多使用一次”，那么第二维很可能表示这个操作是否已经使用。0 表示未使用，1 表示已使用。

此时遇到类似空缺：

```
if (used == 0) {
    if (d[u][used] < _____) {
        _____ = d[u][used];
        pq.push({_____, v, 1});
    }
}
```

就能判断：这里是在从“未使用免费机会”转移到“已使用免费机会”，且费用不增加，目标状态应是 d[v][1]。

从输出倒推时，要抓住答案变量的维度和含义。二维数组、三维数组、多个答案变量，通常都对应题意中的状态分类。

从选项差异反推考点

完善程序题的选项通常很接近。与其逐项读完整，不如先看它们差在哪里。

如果选项差在变量名，例如 d[u] 和 d[v]，考点多半是“从谁转移到谁”。

如果选项差在下标，例如 `i` 和 `i - 1`，考点多半是“当前位置和前一位置”。

如果选项差在比较符号，例如 `<` 和 `<=`，考点多半是“边界是否包含”。

如果选项差在函数名，例如 `next_permutation` 和 `prev_permutation`，考点多半是“枚举顺序和初始排列”。

如果选项差在是否加 1，考点多半是“当前对象本身是否计入答案”。

例题 4：选项只差是否加一

某 DAG 中，`f[u]` 表示从点 `u` 出发的路径数量，路径至少包含一个点。代码片段如下：

```
for (int u : order) {
    f[u] = 1;
    for (int v : E[u]) {
        f[u] = _____;
    }
}
```

选项：

A. `min(f[u] + f[v], LIM)`

B. `min(f[u] + f[v] + 1, LIM)`

C. `min(f[u] * f[v], LIM)`

D. `min(f[u] * (f[v] + 1), LIM)`

题目已经说路径至少包含一个点。`f[u] = 1` 已经把只包含 `u` 的路径计入。对于每条边 `u -> v`，从 `u` 出发并走向 `v` 的路径数量，等于从 `v` 出发的路径数量，每一条都可以在前面接上 `u`。因此应加上 `f[v]`，不再额外加 1。

答案是 A。

这类题不能看到“路径数量”就乱乘。路径是从当前点选择一个后继继续走，不是把所有后继方案任意组合。

代码框架会暴露算法意图

完善程序题经常给出一个较完整的代码框架。这个框架本身会暴露算法意图。

看到这些结构时，可以先作初步判断：

代码结构	可能意图
`sort` 后接双指针	有序序列上的统计或匹配
`while (l < r)`	二分答案或二分位置
`priority_queue` 加距离数组	最短路或贪心取最小状态
`dfs` 加 `vis`	搜索、枚举或图遍历
`f[i][j]`	DP 状态
`pre` 或 `fa`	记录前驱或父结点
`reverse(order)`	DAG 逆拓扑处理
`stack` 或单调比较	单调栈相关处理

这只是初步判断，不能代替读题。比如 `priority_queue` 不一定总是 Dijkstra，也可能是不断取当前最大值；`dfs` 不一定是搜索答案，也可能只是遍历树。判断后还要看变量含义和题意对应。

例题 5：优先队列和状态

```
struct State {
    long long dist;
    int u;
    int used;
    bool operator>(const State &other) const {
        return dist > other.dist;
    }
};

priority_queue<State, vector<State>, greater<State>> pq;
```

这里优先队列每次取出 `dist` 最小的状态。状态中除了点 `u`，还有一个 `used`，说明同一个点可能以不同状态出现。若题意中有“某种机会最多使用一次”，那么 `used` 很可能表示是否已经用过这次机会。

补空时，就不能只写普通的一维距离 `d[v]`，而要判断应该更新 `d[v][0]` 还是 `d[v][1]`。

还原方案题看前驱

有些完善程序题不仅要求输出答案，还要输出方案。此时代码中常出现 `pre`、`from`、`fa` 等数组，用来记录前驱。

读还原方案代码时，要看两件事：

- 前驱记录的是上一个点、上一个状态，还是上一次操作；
- 输出时是递归回溯，还是从终点倒着走再反转。

例如：

```
void print(int u) {
    if (u != s) print(pre[u]);
    cout << u << ' ';
}
```

这段代码说明 `pre[u]` 表示到达 `u` 前的上一个点。输出时先递归输出前驱，再输出当前点，所以最终顺序是从起点到终点。

若状态被拆成两层，例如 `u` 和 `n + u` 表示同一个点的两种状态，那么前驱也要记录状态编号，而不只是点编号。补空时要判断是 `pre[u]`、`pre[n + u]`，还是取模后的某个位置。

还原方案题的空缺容易出现在输出函数里。此时不要只看输出格式，要回到前面“前驱是怎样记录的”。

复杂程序先分块

完善程序题的代码可能很长。遇到长程序，先分块，不要逐行硬读。

可以按功能分成：

- 输入与建图、建表；
- 预处理；
- 主算法；
- 查询或输出；
- 辅助函数。

每一块只问一个问题：它做了什么？

例如一个 RMQ 或 LCA 相关程序，可能包含建树、DFS、预处理、查询四部分。补空可能落在建树，也可能落在查询。若空缺在建树部分，就不必先完全掌握查询细节；若空缺在查询部分，就要知道预处理数组的含义。

长程序的选项也会提供线索。若某空选项全是 `son[0]`、`son[1]`、`top--`，说明它考的是建树过程；若选项全是 `dep`、`dfn`、`min`，说明它考的是深度或遍历序相关的判断。

排除错误选项

完善程序题不是只能正向推出答案。有时可以先排除明显不可能的选项。

常见排除依据如下：

- 类型不匹配：需要整数却给出容器或布尔值；
- 下标越界：访问了不可能合法的位置；
- 状态方向反了：从未来状态转移到过去状态；
- 变量未定义或含义不对：选项中变量不属于当前流程；
- 破坏单调性：二分或双指针所需条件被破坏；
- 漏掉当前状态：最小值、最大值或计数少算一类情况；
- 重复计数：已经初始化计入，又在转移中再加一次。

排除时要有理由。不要因为某个选项“看起来复杂”就排除，也不要因为某个选项“像模板”就选择。完善程序题的干扰项常常就长得很像模板。

例题 6：排除状态方向错误

```
if (d[u][0] < d[v][1]) {
    d[v][1] = d[u][0];
}
```

这表示从点 u 的未使用状态，经过某种操作，到达点 v 的已使用状态。若选项给出：

A. `d[v][0] = d[u][0]`

B. `d[v][1] = d[u][0]`

C. `d[u][1] = d[v][0]`

D. `d[u][0] = d[v][1]`

C 和 D 都把更新方向写反了。程序正在处理从 u 出发到 v 的转移，不应更新回 u。A 没有改变使用状态。只有 B 与“使用一次机会后进入已使用状态”一致。

完善程序的作答顺序

一题完善程序通常有多个空。不要把每个空孤立处理。更合适的顺序是：

第一步，读题意，确定问题类型。

第二步，读变量定义，写出关键变量含义。

第三步，读主流程，判断算法大致结构。

第四步，按代码顺序看空缺，但遇到输出空、答案空时，可以从末尾倒推。

第五步，比较选项差异，判断每个空的考点。

第六步，把填入后的程序在小样例上走一遍，检查是否自洽。

如果某一空暂时无法确定，不要卡住整题。可以先做后面的空。后面的空有时会暴露前面变量的含义。例如输出处使用 `f[u]`，说明前面 `f[u]` 必须表示某种答案；还原方案处使用 `pre`，说明前面更新答案时必须同时记录前驱。

小样例验证

填完空后，尽量用一个小样例验证。小样例不一定是题目给的样例，也可以自己构造。

好的小样例应满足：

- 规模小，便于手算；
- 覆盖边界；
- 能区分相近选项；
- 不需要复杂计算。

例如二分题可以用答案刚好等于 `mid` 的情况，检查等号方向。DP 题可以用 $n = 1$ 或 $n = 2$，检查初始化。图论题可以用两个点一条边，检查起点、终点和状态转移。搜索题可以用只有一种方案或没有方案的情况，检查终止条件。

小样例验证不是重新解整题，而是检查“填入的语句是否和程序流程一致”。

常见失分点

完善程序题的失分点相对集中。

第一，没读题意。只看代码像什么算法，没有看题目要求“最大”“最小”“第 k 小”“至少”“至多”等关键词。

第二，变量含义写反。比如把 `d[u][0]` 当成已经使用机会，把 `d[u][1]` 当成未使用机会；把 `l` 当成答案上界，把 `r` 当成答案下界。

第三，等号方向错。二分、计数、边界判断里尤其常见。

第四，数组下标错。`i`、`i-1`、`i+1`，`j`、`j-mid`，`u`、`v`，这些差别都可能成为考点。

第五，初始化漏掉。DP 或最短路中，初始状态错了，后面转移再正确也没有用。

第六，更新顺序错。有些程序先用旧值再更新，有些程序必须先更新再使用。特别是滚动数组、双指针和递归回溯，要看清楚顺序。

第七，输出状态错。程序中可能有多个状态都到达终点，最终答案需要取最小、最大、求和或选择某一类状态。

把这些失分点提前记住，做题时就会更敏感。

小结

完善程序题考的不是记忆模板，而是把题意、代码和选项连接起来。读懂题意，确定变量含义，判断空缺职责，再处理边界和状态方向，才是稳定作答的路径。

完成这一章后，第一编的考试方法已经讲完。后面从复杂度开始进入具体知识点。学习后面章节时，要不断把知识点放回这三类题型中：选择题怎样问，程序阅读怎样写成代码，完善程序会在哪些位置留空。

第二编 复杂度与程序行为

0x05 时间复杂度

时间复杂度题考的不是程序实际跑了多少秒，而是随着输入规模变大，程序的基本执行次数大致怎样增长。

在 CSP-S 第一轮中，时间复杂度常以三种形式出现：给一段代码，判断复杂度；给一个递归式，判断复杂度；给几个算法复杂度表达式，比较哪个更小。题面通常不长，但很容易在细节处出错。循环变量是否真的变化、递归是否重复调用、排序是否在循环内部、指针是否会反复回退，都会影响答案。

本章是第二编的开篇，先讲第一轮中最常用的复杂度判断方法。递归与递推、分治、二分和排序会在后面几章继续展开。本章只把复杂度判断的主线搭起来。

问题

时间复杂度描述的是“规模增长时，执行次数的增长级别”。

例如，一个程序对长度为 n 的数组求最大值：

```
int mx = a[1];
for (int i = 2; i <= n; i++) {
    if (a[i] > mx) mx = a[i];
}
```

循环从 2 到 n，执行大约 n 次。输入规模扩大一倍，比较次数也大约扩大一倍。因此它是 $O(n)$。

如果程序枚举所有二元组：

```
int cnt = 0;
for (int i = 1; i <= n; i++) {
    for (int j = i + 1; j <= n; j++) {
        cnt++;
    }
}
```

内层不是每次都执行 n 次，但总执行次数是

$$(n-1)+(n-2)+\cdots+1=\frac{n(n-1)}{2}.$$

最高次项是 n^2，所以复杂度是 $O(n^2)$。

时间复杂度关注增长级别，不关注常数。$3n+5$、$10n$、$n/2$ 在复杂度记号中都属于 $O(n)$；n^2+n 属于 $O(n^2)$；$n\log n+100n$ 属于 $O(n\log n)$。

这并不是说常数完全没有意义，而是在第一轮纸面判断中，通常先看增长级别。

常见量级

第一轮选择题和程序阅读中，常见复杂度大致有下面几类。

复杂度	常见来源	增长特点
$O(1)$	固定次数运算、数组下标访问	与 n 无关
$O(\log n)$	二分、每次乘二或除二	增长较慢
$O(n)$	单层遍历、双指针整体推进	与规模成正比
$O(n \log n)$	排序、分治合并、外层线性内层对数	第一轮高频
$O(n^2)$	双重枚举、两两比较	规模翻倍，次数约变为四倍
$O(n^3)$	三重枚举、矩阵类朴素算法	增长较快
$O(2^n)$	枚举所有子集	每增加一个元素，情况数翻倍
$O(n!)$	枚举全排列	增长很快

复杂度比较时，只看 n 很大时谁增长更快。常见顺序为：

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) < O(n!).$$

在第一轮中，不必把所有数学细节展开到很深，但要能快速识别常见来源。

单层循环

单层循环通常是线性的，但要看循环变量怎样变化。

例题 1：普通线性循环

```
int s = 0;
for (int i = 1; i <= n; i++) {
    s += i;
}
```

`i` 从 1 到 n，每次加 1，循环执行 n 次，复杂度为 $O(n)$。

若写成：

```
for (int i = 0; i < n; i++) {
    s += i;
}
```

循环执行次数仍是 n 次，复杂度仍为 $O(n)$。下标从 0 开始还是从 1 开始，不影响量级，但会影响具体执行次数和数组访问位置。

例题 2：每次加常数

```
for (int i = 1; i <= n; i += 3) {
    s += i;
}
```

`i` 每次增加 3，循环次数大约是 $n/3$，复杂度仍为 $O(n)$。

复杂度不关心常数倍，所以 `i += 2`、`i += 5`、`i += 10`，只要步长是固定常数，都是 $O(n)$。

例题 3：每次乘二

```
int cnt = 0;
for (int i = 1; i <= n; i *= 2) {
    cnt++;
}
```

i 依次为 $1, 2, 4, 8, \ldots$。设执行了 k 次，则大约有 $2^k \le n$，所以 $k \le \log_2 n$。复杂度为 $O(\log n)$。

遇到“每次乘一个大于 1 的常数”或“每次除以一个大于 1 的常数”，通常对应对数级别。

变式：循环变量不推进

```
for (int j = 0; j < n; j *= 2) {
    cnt++;
}
```

这里 j 初值为 0，每次执行 j *= 2 后仍然是 0。若 $n > 0$，循环条件 j < n 一直成立，程序无法正常结束。

这类题不能看到 j *= 2 就直接写 $O(\log n)$。要先检查循环变量是否真的发生变化。第一轮考的是代码实际行为，不是代码外形。

双重循环

双重循环不一定都是 $O(n^2)$，但多数题可以先从“总共执行多少次内层语句”入手。

例题 4：完整双重循环

```
int cnt = 0;
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        cnt++;
    }
}
```

外层执行 n 次，每次内层执行 n 次，总执行次数为 n^2，复杂度为 $O(n^2)$。

例题 5：三角形枚举

```
int cnt = 0;
for (int i = 1; i <= n; i++) {
    for (int j = i + 1; j <= n; j++) {
        cnt++;
    }
}
```

当 $i = 1$ 时，内层执行 $n - 1$ 次；当 $i = 2$ 时，内层执行 $n - 2$ 次；最后加起来是：

$$(n-1) + (n-2) + \cdots + 1 = \frac{n(n-1)}{2}.$$

复杂度仍为 $O(n^2)$。

注意，内层不是每次都执行 n 次，但总量仍是平方级。很多“两两比较”“枚举点对”“枚举区间左右端点”的程序，都是这种形式。

例题 6：外层线性，内层对数

```
int cnt = 0;
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j *= 2) {
        cnt++;
    }
}
```

外层 n 次。内层每次 j 乘 2，执行 $O(\log n)$ 次。总复杂度是 $O(n \log n)$。

这类题要把外层次数和内层次数相乘。

例题 7：内层依赖外层

```
int cnt = 0;
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= i; j *= 2) {
        cnt++;
    }
}
```

当外层为 i 时，内层执行 $O(\log i)$ 次。总次数为：

$$\log 1 + \log 2 + \cdots + \log n.$$

在第一轮中，可以把它判断为 $O(n \log n)$。因为每一项都不超过 $\log n$，一共有 n 项，所以总量不超过 $n \log n$；同时后半部分的很多项也在这个级别附近，量级就是 $O(n \log n)$。

不是所有嵌套都要相乘

看到嵌套循环时，不能机械地把每层次数相乘。有些内层变量不会每次从头开始，而是整体只前进有限次。

例题 8：双指针

```
int ans = 0;
for (int i = 1, j = 1; i <= n; i++) {
    while (j <= n && a[j] - a[i] <= k) {
        j++;
    }
    ans += j - i - 1;
}
```

这里外层 `i` 从 1 到 n。内层 `while` 中的 `j` 不会在每轮外层重新变回 1，它只会不断增加，最多从 1 增加到 $n+1$。

因此，虽然代码是循环套循环，但 `i` 总共增加 n 次，`j` 总共增加不超过 n 次，总复杂度是 $O(n)$。

这类题的关键是看指针是否单调移动。若两个指针都只向一个方向走，总步数通常是线性的。

变式：内层重新开始

```
int ans = 0;
for (int i = 1; i <= n; i++) {
    int j = i;
```

```
    while (j <= n && a[j] - a[i] <= k) {
        j++;
    }
    ans += j - i - 1;
}
```

这次 `j` 在每轮外层都重新赋值为 `i`。最坏情况下，内层可能反复扫描很多位置，总复杂度可能达到 $O(n^2)$。

同样是 `while` 嵌套在 `for` 里，是否重新初始化，复杂度差别很大。

排序和复杂度

第一轮常把排序放进代码中，要求判断整体复杂度。常见比较排序的复杂度是 $O(n \log n)$，例如 C++ 中常用的 `sort` 通常按这个量级判断。

如果程序是：

```
sort(a + 1, a + n + 1);
for (int i = 1; i <= n; i++) {
    ans += a[i];
}
```

排序是 $O(n \log n)$，后面的循环是 $O(n)$。合起来取较大的量级，整体是 $O(n \log n)$。

如果程序是：

```
for (int i = 1; i <= n; i++) {
    sort(a + 1, a + n + 1);
}
```

排序在循环里执行了 n 次，每次 $O(n \log n)$，整体是 $O(n^2 \log n)$。

判断整体复杂度时，要看高代价操作是在循环外，还是循环内。排序只出现一次，和出现 n 次，差别很大。

递归的初步判断

递归复杂度会在第 6 章和第 7 章继续讲。本章先处理第一轮中常见的基础判断。

递归复杂度要看：

- 每次调用产生几个子调用；
- 子问题规模怎样变化；
- 除递归外，本层还做了多少工作；
- 是否重复计算了相同问题。

例题 9：线性递归

```
int f(int n) {
    if (n == 1) return 1;
    return f(n - 1) + n;
}
```

每次只调用一次 `f(n - 1)`，规模减 1。从 n 到 1，总共约 n 层，每层做常数次运算，复杂度为 $O(n)$。

例题 10：二分递归

```
int f(int n) {
    if (n == 1) return 1;
    return f(n / 2) + 1;
}
```

每次只调用一次 `f(n/2)`，规模不断减半。层数为 $O(\log n)$，每层常数工作，复杂度为 $O(\log n)$。

例题 11：重复递归

```
int fib(int n) {
    if (n <= 2) return 1;
    return fib(n - 1) + fib(n - 2);
}
```

每次调用会继续调用 `fib(n - 1)` 和 `fib(n - 2)`，其中大量子问题会被重复计算。例如 `fib(n - 2)` 会在不同分支中反复出现。因此朴素写法是指数级复杂度。

若改成从小到大递推：

```
f[1] = f[2] = 1;
for (int i = 3; i <= n; i++) {
    f[i] = f[i - 1] + f[i - 2];
}
```

每个 `f[i]` 只计算一次，复杂度为 $O(n)$。

这里差别不在于 “递归” 这两个字本身，而在于是否重复计算同一批状态。

分治递归与主定理

有些递归来自分治：把一个规模为 n 的问题拆成若干个规模较小的问题，递归求解，再合并结果。它们经常写成下面的形式：

$$T(n) = aT(n/b) + f(n).$$

其中，a 表示每次产生几个子问题，n/b 表示每个子问题的规模，$f(n)$ 表示这一层除递归之外还要做的工作，例如划分、合并、扫描数组等。

主定理可以快速判断这类递归式的复杂度。第一轮中先掌握速用版即可。

设

$$g(n) = n^{\log_b a}.$$

把 $f(n)$ 和 $g(n)$ 比较：

情况	含义	结论
$f(n)$ 比 $g(n)$ 小一个多项式级别	递归产生的子问题占主导	$T(n) = O(g(n))$
$f(n)$ 与 $g(n)$ 同级	每层工作量大致相同	$T(n) = O(g(n)\log n)$
$f(n)$ 比 $g(n)$ 大一个多项式级别	当前层和靠上的层占主导	$T(n) = O(f(n))$

这里的“同级”可以理解为量级相同，例如 $f(n)=O(n)$，$g(n)=n$；“小一个多项式级别”可以粗略理解为指数低一些，例如 n 比 n^2 小；“大一个多项式级别”则相反。

主定理在第一轮中的作用，是快速处理标准分治递推式。它不代替读代码。拿到实际程序时，仍要先判断递归式是否真的是 $T(n)=aT(n/b)+f(n)$。

例题 12：归并排序式递归

$$T(n)=2T(n/2)+O(n).$$

这里 $a=2,b=2$，所以

$$g(n)=n^{\log_2 2}=n.$$

又因为 $f(n)=O(n)$，与 $g(n)$ 同级，所以

$$T(n)=O(n\log n).$$

这对应很多“分成两半，递归处理，再线性合并”的程序。

也可以按层理解：

层数	子问题数量	每个规模	本层总代价
第 0 层	1	n	n
第 1 层	2	$n/2$	n
第 2 层	4	$n/4$	n
……	……	……	……

共有 $O(\log n)$ 层，每层 $O(n)$，所以总复杂度是 $O(n\log n)$。

例题 13：本层代价为平方

$$T(n)=2T(n/2)+O(n^2).$$

这里 $a=2,b=2$，仍有

$$g(n)=n^{\log_2 2}=n.$$

而 $f(n)=O(n^2)$，比 $g(n)$ 大一个多项式级别，所以

$$T(n)=O(n^2).$$

这类题很容易误选 $O(n^2\log n)$。错误原因是只看到了递归有 $O(\log n)$ 层，却没有比较每层工作量。实际上第一层代价是 n^2，下一层总代价约为 $n^2/2$，再下一层约为 $n^2/4$，总量由前几层主导。

例题 14：子问题更多

$$T(n)=4T(n/2)+O(n).$$

这里 $a=4,b=2$，所以

$$g(n)=n^{\log_2 4}=n^2.$$

而 $f(n) = O(n)$，比 $g(n)$ 小一个多项式级别，所以

$$T(n) = O(n^2).$$

虽然每一层只做线性额外工作，但子问题数量增长得更快，最终复杂度由递归展开出的子问题数量主导。

例题 15：三分递归

$$T(n) = 3T(n/3) + O(n).$$

这里 $a = 3, b = 3$，所以

$$g(n) = n^{\log_3 3} = n.$$

$f(n) = O(n)$，与 $g(n)$ 同级，所以

$$T(n) = O(n \log n).$$

它和 $T(n) = 2T(n/2) + O(n)$ 的复杂度相同，原因是每一层的总工作量都大致为 $O(n)$，层数都是对数级。

例题 16：只有一个子问题

$$T(n) = T(n/2) + O(n).$$

这里 $a = 1, b = 2$，所以

$$g(n) = n^{\log_2 1} = 1.$$

$f(n) = O(n)$，比 $g(n)$ 大一个多项式级别，因此

$$T(n) = O(n).$$

按层看也很好理解：第一层做 n，下一层做 $n/2$，再下一层做 $n/4$，加起来是

$$n + \frac{n}{2} + \frac{n}{4} + \cdots,$$

总量是 $O(n)$。

主定理不能乱套

主定理只适合形如 $T(n) = aT(n/b) + f(n)$ 的标准分治递归式，其中 a, b 是常数，且子问题规模相同或可以近似看成相同。

下面这些情况不要直接套：

- $T(n) = T(n-1) + O(1)$，规模不是按比例缩小；
- $T(n) = T(n/2) + T(n/3) + O(n)$，子问题规模不相同；
- 程序中两个递归调用是否都会执行还不确定；
- 代码中有记忆化，重复子问题不会重复计算；
- 递归之外还有循环嵌套调用整个递归函数。

第一轮中，主定理主要用于快速处理标准分治递归式。若式子不符合标准形式，就回到递归树、按层统计或直接数执行次数。

重复调用和保存结果

程序阅读题中，递归式不能只看数学结果，还要看程序是否真的重复调用。

例题 17：两个相同递归调用

```
int f(int n) {
    if (n == 1) return 1;
    return f(n / 2) + f(n / 2);
}
```

从数学结果看，两个 `f(n/2)` 的值相同；但程序会调用两次。若没有把结果保存下来，执行次数仍要算两次递归。递归式是：

$$T(n) = 2T(n/2) + O(1).$$

用主定理，$a = 2, b = 2$，$g(n) = n$，$f(n) = O(1)$ 比 $g(n)$ 小，所以复杂度是 $O(n)$。

若写成：

```
int f(int n) {
    if (n == 1) return 1;
    int t = f(n / 2);
    return t + t;
}
```

这才只调用一次递归。递归式变成：

$$T(n) = T(n/2) + O(1),$$

复杂度为 $O(\log n)$。

程序阅读题中，不能把 “两个表达式值相同” 直接当成 “只算一次”。

复杂度表达式比较

选择题中常要求比较几个复杂度。当图中 $m = \Theta(n)$，或 $m = \Theta(n^2)$ 时，还需要把 m 换成 n 的量级。

例题 18：稀疏图中的表达式比较

设图有 n 个点、m 条边，且 $m = \Theta(n)$。比较：

- $O(m \log n)$；
- $O(n^2 + m)$；
- $O(m + n \log n)$。

因为 $m = \Theta(n)$，可以把 m 看成和 n 同级。

于是：

- $O(m\log n)=O(n\log n)$；
- $O(n^2+m)=O(n^2)$；
- $O(m+n\log n)=O(n+n\log n)=O(n\log n)$。

前两个中，$O(n\log n)$ 小于 $O(n^2)$。第一项和第三项量级相同，若题目还比较常数，第一轮通常不会要求；若选项中只有一个最小量级，就按量级判断。

例题 19：稠密图中的表达式比较

若 $m=\Theta(n^2)$，则：

- $O(m\log n)=O(n^2\log n)$；
- $O(n^2+m)=O(n^2)$；
- $O(m+n\log n)=O(n^2)$。

此时含 m 的表达式可能和 n^2 同级。图论复杂度题一定要看清楚题目给的是稀疏图还是稠密图。

取最大项

一个程序分成几段时，整体复杂度通常取其中增长最快的一段。

例如：

```
for (int i = 1; i <= n; i++) {
    a[i] = i;
}

for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        cnt++;
    }
}
```

第一段是 $O(n)$，第二段是 $O(n^2)$，整体是 $O(n^2)$。

如果程序是：

```
sort(a + 1, a + n + 1);

for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        cnt++;
    }
}
```

排序是 $O(n\log n)$，双重循环是 $O(n^2)$，整体是 $O(n^2)$。

当多个部分顺序执行时，把它们的复杂度相加，再保留增长最快的项。

多个参数

有些题不只有 n，还有 m、k、字符集大小、图的边数等。此时不要强行只写成 n。

例如：

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= m; j++) {
        cnt++;
    }
}
```

复杂度是 $O(nm)$。只有题目额外说明 m 和 n 的关系时，才进一步化简。

再如：

```
for (int i = 1; i <= n; i++) {
    for (int c = 0; c < 26; c++) {
        cnt++;
    }
}
```

内层 26 是常数，所以复杂度是 $O(n)$，不是 $O(26n)$。如果字符集大小不是固定常数，而是输入给出的 k，那复杂度就应写成 $O(nk)$。

判断多参数复杂度时，先保留所有输入规模，再看题目是否给出关系。

程序阅读中的复杂度

程序阅读题问复杂度时，不一定要求完全识别算法名称。常见做法是找出最耗时的部分。

读复杂度时，可以按下面顺序：

- 看是否有排序；
- 看是否有多重循环；
- 看循环变量是否单调推进；
- 看递归调用次数和规模变化；
- 看是否可以套主定理；
- 看是否枚举子集或排列；
- 看是否每次循环里又调用了高代价函数。

例题 20：函数调用藏在循环里

```
int calc(int x) {
    int s = 0;
    for (int i = 1; i <= x; i++) {
        s += i;
    }
    return s;
}

int ans = 0;
for (int i = 1; i <= n; i++) {
    ans += calc(n);
}
```

`calc(n)` 是 $O(n)$。外层调用它 n 次，总复杂度为 $O(n^2)$。

如果只看主函数中的外层循环，会误判为 $O(n)$。程序阅读题中，函数调用也要算进去。

变式：参数变化

```
int ans = 0;
for (int i = 1; i <= n; i++) {
    ans += calc(i);
}
```

这时总次数是 $1+2+\cdots+n$，仍为 $O(n^2)$。

选择题中的常见陷阱

时间复杂度题常见陷阱有以下几类。

陷阱	说明
循环变量不变	例如 `j=0` 后执行 `j*=2`，无法推进
内层变量不重置	双指针整体线性，不是平方
高代价函数在循环内	排序、查找、递归调用可能被重复执行
两个递归表达式重复调用	值相同不代表只算一次
主定理套错形式	不是 $aT(n/b)+f(n)$ 的式子不要硬套
只看层数不看每层代价	分治递归不能只套 $\log n$
忽略输入参数关系	$m=\Theta(n)$ 和 $m=\Theta(n^2)$ 会改变比较
把常数当变量	固定 26、固定 10 不改变复杂度
把变量当常数	输入给出的 k 不能随便忽略

做题时，一旦选项中出现很接近的复杂度，例如 $O(n)$ 和 $O(n\log n)$、$O(n^2)$ 和 $O(n^2\log n)$，就要重点检查这些陷阱。

原创例题

例题 21：判断复杂度

求下面程序的时间复杂度。

```
int ans = 0;
for (int i = 1; i <= n; i++) {
    for (int j = i; j <= n; j += i) {
        ans++;
    }
}
```

对于固定的 i，内层枚举 $i, 2i, 3i, \ldots$，执行大约 n/i 次。总次数为：

$$\frac{n}{1}+\frac{n}{2}+\frac{n}{3}+\cdots+\frac{n}{n}.$$

也就是：

$$n\left(1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n}\right).$$

第一轮中，可以记住这个常见结果：$1+\frac{1}{2}+\cdots+\frac{1}{n}$ 的量级是 $O(\log n)$。所以总复杂度为 $O(n\log n)$。

这类循环常出现在倍数枚举、约数统计、筛法相关程序中。

变式 1

```
int ans = 0;
for (int i = 1; i <= n; i++) {
    for (int j = 1; j * i <= n; j++) {
        ans++;
    }
}
```

内层条件 `j * i <= n` 等价于 `j <= n / i`。对固定 i，内层仍执行约 n/i 次，所以复杂度仍为 $O(n \log n)$。

变式 2

```
int ans = 0;
for (int i = 1; i <= n; i++) {
    for (int j = i; j <= n; j++) {
        ans++;
    }
}
```

这次内层不是按倍数跳，而是每次加 1。总次数为：

$$n + (n-1) + (n-2) + \cdots + 1.$$

复杂度为 $O(n^2)$。

两个程序只差 `j += i` 和 `j++`，复杂度从 $O(n \log n)$ 变成 $O(n^2)$。判断复杂度时，循环变量更新方式非常关键。

小结

遇到时间复杂度题，可以按下面顺序检查。

步骤	检查问题
看循环	循环变量初值、条件、更新是否正确
算次数	内层语句总共执行多少次
看指针	指针是否每轮重置，还是整体单调推进
看函数	循环内是否调用排序、递归或其他高代价函数
看递归	每次产生几个子调用，规模怎样变化
看主定理	是否符合 $T(n) = aT(n/b) + f(n)$
看分层	每层总代价是否相同，还是逐层变小
看参数	是否有 m, k 等多个输入规模
取最大项	顺序执行的多段程序保留增长最快的一段
查陷阱	是否存在死循环、重复调用、边界误判

时间复杂度题的关键不是背很多结论，而是把执行次数数清楚。先看循环是否真的推进，再看总次数；先看递归调用是否重复，再看规模变化；能套主定理时再套主定理，不能套时回到递归树或按层统计。后面学习递归、分治、二分、排序和数据结构时，这些判断会反复用到。

0x06 递归与递推

递归和递推都是用 “已经知道的较小问题” 来解决 “当前问题”。它们看起来接近，但在程序阅读中表现很不一样。

递归是函数调用自己。读递归程序时，要看入口、终止条件、参数变化、返回值和全局变量变化。

递推是按某种顺序逐步计算。读递推程序时，要看初值、转移关系、计算顺序和数组含义。

CSP-S 第一轮中，递归与递推常以三种方式出现：选择题中问递归式和复杂度，程序阅读中给出一段递归代码让考生判断输出，完善程序中要求补出边界或转移。会写递归不等于会读递归，能背递推公式也不等于能看懂程序里的状态变化。本章重点讲纸面考试中怎样判断。

问题

先看一个最简单的递归程序：

```
int f(int n) {
    if (n == 1) return 1;
    return f(n - 1) + n;
}
```

它的含义是：

- 当 $n = 1$ 时，答案是 1；
- 当 $n > 1$ 时，先求 $f(n-1)$，再加上 n。

所以 $f(4)$ 的计算过程是：

```
f(4)
= f(3) + 4
= f(2) + 3 + 4
= f(1) + 2 + 3 + 4
= 1 + 2 + 3 + 4
= 10
```

这段程序做的是求 $1 + 2 + \cdots + n$。

同样的问题也可以用递推写：

```
f[1] = 1;
for (int i = 2; i <= n; i++) {
    f[i] = f[i - 1] + i;
}
```

递归从当前问题往更小问题走，递推从小问题一步步往大问题算。两者本质联系很紧，但读代码时关注点不同。

递归要问：函数什么时候停？每次调用参数怎样变？返回值怎样合并？

递推要问：初值是什么？转移公式是什么？计算顺序是否保证旧状态已经算好？

递归的三件事

读递归程序，先抓三件事。

第一，终止条件。没有终止条件，或者终止条件永远到不了，递归就无法正常结束。

第二，递归方向。参数要朝终止条件靠近。例如 `n - 1`、`n / 2` 都可能使规模变小；如果参数没有变小，就要警惕。

第三，合并方式。递归调用返回后，当前层怎样得到结果。可能是加法、取最大值、计数，也可能只是修改全局变量。

例题 1：阶乘递归

```
int fac(int n) {
    if (n == 0) return 1;
    return fac(n - 1) * n;
}
```

终止条件是 `n == 0`。递归方向是从 `n` 变成 `n - 1`。合并方式是把较小问题的答案乘上当前的 `n`。

所以：

```
fac(4)
= fac(3) * 4
= fac(2) * 3 * 4
= fac(1) * 2 * 3 * 4
= fac(0) * 1 * 2 * 3 * 4
= 24
```

这类递归每层只调用一次自身，层数为 $n+1$，复杂度为 $O(n)$。

变式：终止条件不同

```
int fac(int n) {
    if (n == 1) return 1;
    return fac(n - 1) * n;
}
```

如果输入保证 $n \geq 1$，这段程序也能计算阶乘。但若输入 $n = 0$，它不会到达 `n == 1`，会继续调用 `fac(-1)`、`fac(-2)` 等。程序阅读题中，必须结合输入范围判断终止条件是否足够。

第一轮常把“输入保证”写在题干里。读递归边界时，要看题目保证了什么，不要只看代码表面。

递归调用顺序

递归不是把所有公式同时展开。程序执行时，先进入更深层，等深层返回后，再继续当前层后面的语句。

看下面程序：

```
void f(int n) {
    if (n == 0) return;
    cout << n << ' ';
    f(n - 1);
}
```

输入 `f(3)` 时，输出顺序是：

```
3 2 1
```

因为每一层都是先输出当前 n，再递归。

若代码改成：

```
void f(int n) {
    if (n == 0) return;
    f(n - 1);
    cout << n << ' ';
}
```

输出顺序变成：

```
1 2 3
```

同样是递归到 n - 1，输出语句在递归前还是递归后，结果完全不同。

例题 2：两次输出

```
void f(int n) {
    if (n == 0) return;
    cout << n << ' ';
    f(n - 1);
    cout << n << ' ';
}
```

调用 f(3)。

按执行顺序写：

```
f(3): 输出 3，调用 f(2)，返回后输出 3
f(2): 输出 2，调用 f(1)，返回后输出 2
f(1): 输出 1，调用 f(0)，返回后输出 1
f(0): 返回
```

所以输出为：

```
3 2 1 1 2 3
```

读递归输出题时，不要只看递归参数变化，还要看输出语句在递归调用前还是后。

全局变量与回溯

递归程序中常出现全局变量，例如 ans、vis、p、cnt。这些变量不会随函数返回自动恢复，除非程序显式恢复。

例题 3：排列枚举

```
int n, ans;
int p[10];
bool vis[10];

void dfs(int k) {
    if (k == n + 1) {
```

```
        ans++;
        return;
    }
    for (int i = 1; i <= n; i++) {
        if (vis[i]) continue;
        p[k] = i;
        vis[i] = true;
        dfs(k + 1);
        vis[i] = false;
    }
}
```

变量含义如下：

变量	含义
k	当前填写第 k 个位置
i	当前尝试放入的数
p[k]	第 k 个位置的数
vis[i]	数字 i 是否已经使用
ans	已经找到的排列数量

`vis[i] = true` 表示进入一个分支前做选择。`vis[i] = false` 表示递归返回后撤销选择，让后面的分支还能使用数字 i。

若删掉最后的 `vis[i] = false`，当前分支用过的数字会一直保留，后续分支会被错误限制，答案会改变。

程序阅读题常问 “删除某一行是否影响答案”。遇到递归搜索，先检查被删的是不是回溯恢复语句。

变式：局部变量不需要恢复

```
void dfs(int k, int sum) {
    if (k == n + 1) {
        ans = max(ans, sum);
        return;
    }
    dfs(k + 1, sum);
    dfs(k + 1, sum + a[k]);
}
```

这里 `sum` 是按值传递。每次递归调用都有自己的 `sum`，不会改变上一层的 `sum`。因此递归返回后不需要写 “恢复 `sum`” 的语句。

判断是否需要恢复，要看变量是否会影响其他分支。全局数组、引用参数、指针指向的对象通常要特别小心；普通按值传递的局部变量通常不需要手动恢复。

递归的复杂度

递归复杂度要看调用树。不同递归形式差别很大。

一次递归调用

```
int f(int n) {
    if (n == 1) return 1;
    return f(n - 1) + n;
}
```

每层只调用一次 `f(n - 1)`，层数约为 n，每层做常数工作，所以复杂度为 $O(n)$。

```
int g(int n) {
    if (n == 1) return 1;
    return g(n / 2) + 1;
}
```

每层只调用一次 `g(n / 2)`，规模每次减半，层数为 $O(\log n)$，所以复杂度为 $O(\log n)$。

两次递归调用

```
int h(int n) {
    if (n == 1) return 1;
    return h(n - 1) + h(n - 1);
}
```

每一层会产生两个相同规模的递归调用。虽然两个调用的返回值相同，但程序没有保存结果，会算两次。调用次数按层翻倍，复杂度是指数级。

如果写成：

```
int h(int n) {
    if (n == 1) return 1;
    int t = h(n - 1);
    return t + t;
}
```

这时每层只调用一次 `h(n - 1)`，复杂度变为 $O(n)$。

程序阅读中，“表达式相同” 不等于 “只执行一次”。要数函数调用出现了几次。

斐波那契递归

```
int fib(int n) {
    if (n <= 2) return 1;
    return fib(n - 1) + fib(n - 2);
}
```

这段程序会反复计算相同的值。例如求 `fib(5)` 时：

```
fib(5)
= fib(4) + fib(3)
= fib(3) + fib(2) + fib(3)
```

`fib(3)` 已经出现了两次。规模更大时，重复会更多，所以朴素递归是指数级。

若改成递推：

```
f[1] = f[2] = 1;
for (int i = 3; i <= n; i++) {
    f[i] = f[i - 1] + f[i - 2];
}
```

每个状态只计算一次，复杂度是 $O(n)$。

这类题的考点不是 “递归一定慢，递推一定快”，而是 “是否重复计算同一批状态”。

递推的初值

递推程序首先看初值。初值错了，后面的转移再正确也会错。

例题 4：斐波那契递推

```
f[0] = 0;
f[1] = 1;
for (int i = 2; i <= n; i++) {
    f[i] = f[i - 1] + f[i - 2];
}
```

这里的定义是 $F(0)=0, F(1)=1$。如果题目定义为 $F(1)=1, F(2)=1$，程序初值就要相应改变：

```
f[1] = 1;
f[2] = 1;
for (int i = 3; i <= n; i++) {
    f[i] = f[i - 1] + f[i - 2];
}
```

第一轮中，斐波那契数列常被用来考递归复杂度、递推初值和数组下标。要看清题目从 $F(0)$ 开始还是从 $F(1)$ 开始。

例题 5：递推初值影响答案

定义 $a_1=2$，$a_2=3$，且 $a_i=a_{i-1}+a_{i-2}$。求 a_5。

按顺序计算：

i	a_i
1	2
2	3
3	5
4	8
5	13

如果误把初值写成 $a_0=2, a_1=3$，结果就会错。递推题中，初值和下标是一体的，不能只记公式。

递推的顺序

递推要求用到的旧值已经算好。若计算顺序不对，就可能读到未更新或被覆盖的值。

例题 6：从小到大

```
f[1] = 1;
for (int i = 2; i <= n; i++) {
    f[i] = f[i - 1] + i;
}
```

计算 `f[i]` 时，只需要 `f[i - 1]`，而 `f[i - 1]` 已经在前一轮算好，所以从小到大是正确的。

例题 7：倒序更新

```
for (int i = n; i >= 1; i--) {
    f[i] = f[i + 1] + a[i];
}
```

这里计算 `f[i]` 时需要 `f[i + 1]`，所以要从大到小计算。若改成从小到大，就会先用到还没算好的 `f[i + 1]`。

读递推程序时，不要只看公式，还要看循环方向。循环方向决定转移是否成立。

递推与前缀和

递推不只出现在数列中，也常出现在前缀和、计数和状态更新中。

例题 8：前缀和

```
s[0] = 0;
for (int i = 1; i <= n; i++) {
    s[i] = s[i - 1] + a[i];
}
```

这里 `s[i]` 表示前 i 个数的和，即：

$$s[i] = a[1] + a[2] + \cdots + a[i].$$

区间 $[l, r]$ 的和可以写成：

$$s[r] - s[l-1].$$

读这类程序时，要看清 `s[0]` 的初值。若数组从 1 开始，`s[0]=0` 很重要；若数组从 0 开始，公式可能变成另一种写法。

例题 9：计数递推

```
f[0] = 1;
for (int i = 1; i <= n; i++) {
    f[i] = 0;
    if (i >= 1) f[i] += f[i - 1];
    if (i >= 2) f[i] += f[i - 2];
}
```

可以把它理解为：走到位置 i，最后一步可能从 $i-1$ 走来，也可能从 $i-2$ 走来。

因此 `f[i]` 表示到达位置 i 的方案数。`f[0]=1` 表示“什么都不走”也算一种初始方式。若把 `f[0]` 设成 0，所有后续方案数都会被影响。

完善程序题中，初值常是空缺。做这类题时，要先判断状态含义，再判断空状态是否应该算一种方案。

递归转递推

有些递归可以直接转成递推。关键是看递归依赖哪些更小状态。

例如：

```
int f(int n) {
    if (n <= 2) return 1;
    return f(n - 1) + f(n - 2);
}
```

它依赖 $n-1$ 和 $n-2$，所以可以从小到大递推：

```
f[1] = f[2] = 1;
for (int i = 3; i <= n; i++) {
    f[i] = f[i - 1] + f[i - 2];
}
```

再看：

```
int g(int n) {
    if (n == 1) return 1;
    return 5 * g(n - 1) % mod;
}
```

递推写法为：

```
g[1] = 1;
for (int i = 2; i <= n; i++) {
    g[i] = 5 * g[i - 1] % mod;
}
```

递归转递推时，要做三件事：

步骤	内容
找边界	递归中的 `return` 边界对应递推初值
找依赖	当前状态依赖哪些更小状态
定顺序	让被依赖的状态先算好

这也是很多 DP 的基础思想。后面讲动态规划时，会在更复杂的状态上继续使用这套方法。

递推转滚动变量

如果递推只依赖少数几个旧值，不一定要保存整个数组。

例如斐波那契递推：

```
f[1] = f[2] = 1;
for (int i = 3; i <= n; i++) {
    f[i] = f[i - 1] + f[i - 2];
}
```

只需要前两项，可以写成：

```
int a = 1, b = 1, c;
for (int i = 3; i <= n; i++) {
    c = a + b;
    a = b;
    b = c;
}
```

这里 `a` 表示上一轮的较早项，`b` 表示上一轮的较晚项，`c` 是当前项。每轮更新后，窗口向后移动一格。

程序阅读中，滚动变量比数组更难看懂。遇到这种写法，要把每个变量对应到数列项上。

例如开始时：

变量	含义
a	F_{i-2}
b	F_{i-1}
c	F_i

更新 `a = b; b = c;` 后，下一轮仍保持这个关系。

滚动变量题常考更新顺序。如果先写 `a = b`，再写 `b = a + b`，此时 `a` 已经变了，含义会错。遇到这类程序，要按语句顺序追踪，不能只按公式想。

递归程序阅读

递归程序阅读题常见问题包括：

- 给定输入，问输出；
- 判断某个变量的取值范围；
- 判断删去某行是否影响结果；
- 判断函数调用次数或复杂度；
- 判断递归函数的含义。

做递归阅读题，可以按下面顺序：

1. 找终止条件；
2. 写出参数含义；
3. 看递归调用的位置和次数；
4. 看递归前后的语句；
5. 看全局变量是否改变；
6. 用小输入追踪一遍。

例题 10：统计调用次数

```
int cnt = 0;

void f(int n) {
    cnt++;
    if (n == 0) return;
    f(n - 1);
}
```

调用 `f(3)` 时，函数会依次进入 `f(3),f(2),f(1),f(0)`，共调用 4 次，所以 `cnt=4`。

若改成：

```
void f(int n) {
    cnt++;
    if (n == 0) return;
    f(n - 1);
    f(n - 1);
}
```

调用次数就会按树状展开。设 $C(n)$ 表示调用 `f(n)` 的总次数，则：

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

这类递推会快速增长。第一轮中，不一定要求写出精确通项，但要能判断它不是线性。

例题 11：递归前后修改变量

```
int x = 0;

void f(int n) {
    if (n == 0) return;
    x += n;
    f(n - 1);
    x -= n;
}
```

调用 `f(3)` 结束后，`x` 的值仍为 0。因为每一层递归前加了 `n`，递归后又减了同一个 `n`。

但如果程序在中途输出 `x`，输出的值会受到递归深度影响。判断最终值和判断中间输出，是两件不同的事。

递推程序阅读

递推程序阅读题常见问题包括：

- 某个输入下数组最终内容；
- 输出的是哪个状态；
- 初值改变后是否影响答案；
- 循环方向改变后是否仍正确；
- 时间复杂度和空间复杂度。

读递推程序时，要先给数组写含义。

例题 12：数组含义

```
f[0] = 1;
for (int i = 1; i <= n; i++) {
    f[i] = f[i - 1] * i;
}
cout << f[n];
```

由转移可知：

- `f[0]=1`；
- `f[1]=1`；
- `f[2]=2`；
- `f[3]=6`。

所以 `f[i]` 表示 $i!$。

如果题目问 “`f[5]` 是多少”，可以直接算 $5!=120$。如果题目问复杂度，循环执行 n 次，是 $O(n)$。

例题 13：状态被覆盖

```
for (int i = 1; i <= n; i++) {
    f[i] = f[i] + f[i - 1];
}
```

这段代码不是单纯“由 `f[i - 1]` 推出 `f[i]`”，因为 `f[i]` 原来已经有值。它是在原值基础上累加前一项。读递推程序时，要看左边变量在本轮之前是否已经存有含义。

在完善程序中，若空缺处是 `f[i] = ...`，要判断是赋值还是累加。方案计数常用 `+=`，最值转移常用 `max` 或 `min`，初始化时常用直接赋值。

原创例题

例题 14：递归输出

阅读程序：

```
void f(int n) {
    if (n == 0) return;
    cout << n;
    f(n - 1);
    if (n % 2 == 0) cout << n;
}
```

调用 `f(4)`，输出什么？

先看递归前输出：进入 `f(4),f(3),f(2),f(1)` 时，依次输出 `4,3,2,1`。

递归返回后，只有偶数层会再次输出：

- `f(1)` 返回后，1 不是偶数，不输出；
- `f(2)` 返回后，输出 `2`；
- `f(3)` 返回后，不输出；
- `f(4)` 返回后，输出 `4`。

所以完整输出为：

```
432124
```

这类题要分清“进入递归时输出”和“递归返回后输出”。

变式 1

如果把最后一句改成：

```
if (n % 2 == 1) cout << n;
```

调用 `f(4)` 时，前半部分仍输出 `4321`。返回时输出奇数层：先输出 `1`，再输出 `3`。所以结果是：

```
432113
```

变式 2

如果把递归调用提前：

```
void f(int n) {
    if (n == 0) return;
    f(n - 1);
    cout << n;
    if (n % 2 == 0) cout << n;
}
```

调用 f(4) 时，会先深入到 f(0)，再从小到大输出。各层返回时：

- n=1 输出 1；
- n=2 输出 22；
- n=3 输出 3；
- n=4 输出 44。

所以输出为：

```
122344
```

递归调用位置改变后，输出顺序随之改变。

小结

递归与递推是后面许多知识点的基础。分治会用递归拆分问题，搜索会用递归枚举状态，动态规划会用递推计算状态。第一轮中，先把递归调用顺序、边界条件、复杂度和递推初值看准，后面读更复杂的程序才不会失去方向。

0x07 分治

分治是一种把大问题拆成小问题的方法。它通常包含三步：分解、解决、合并。

“分解” 是把原问题拆成若干个规模较小的问题；“解决” 是递归处理这些小问题；“合并” 是把小问题的结果组合成原问题的结果。许多看起来复杂的递归程序，本质上都在做这三件事。

CSP-S 第一轮中，分治常以程序阅读、复杂度判断和完善程序的形式出现。题目不一定要求写出完整分治算法，但会要求考生看懂递归边界、区间划分、返回值含义、合并过程和递归式。

上一章讲了递归与递推，本章在此基础上讨论 “递归怎样拆问题”。下一章会讲二分，二分也有 “每次缩小规模” 的特征，但它通常只保留一侧；分治则常常要处理多个子问题，并把结果合并。

问题

先看一个求区间最大值的递归程序：

```
int solve(int l, int r) {
    if (l == r) return a[l];
    int mid = (l + r) / 2;
    int x = solve(l, mid);
    int y = solve(mid + 1, r);
    return max(x, y);
}
```

这个程序要求数组 `a[l..r]` 中的最大值。

它的过程是：

- 若区间只剩一个元素，答案就是这个元素；
- 否则把区间分成左半段和右半段；
- 分别求出左半段最大值和右半段最大值；
- 两者取较大值，作为整个区间的最大值。

这就是一个非常直接的分治程序。

如果输入区间是 $[1,8]$，递归结构大致如下：

```
solve(1,8)
├─ solve(1,4)
│  ├─ solve(1,2)
│  └─ solve(3,4)
└─ solve(5,8)
   ├─ solve(5,6)
   └─ solve(7,8)
```

每个区间继续拆成两半，直到长度为 1。之后从小区间逐层返回，把最大值合并上来。

读分治程序时，要抓住四个问题：

- 当前函数解决哪个问题？
- 递归边界是什么？

- 子问题怎么划分？
- 子问题答案怎么合并？

这四个问题回答清楚，很多分治程序就能看懂。

区间写法

分治程序经常处理区间。区间边界写错，整个程序就会出问题。

常见写法有两种。

第一种是闭区间 `[l,r]`，表示包含 `l` 和 `r`。

```
int solve(int l, int r) {
    if (l == r) return a[l];
    int mid = (l + r) / 2;
    int x = solve(l, mid);
    int y = solve(mid + 1, r);
    return max(x, y);
}
```

这里左半段是 `[l,mid]`，右半段是 `[mid+1,r]`。两个区间不重叠，也不遗漏。

第二种是左闭右开区间 `[l,r)`，表示包含 `l`，不包含 `r`。

```
int solve(int l, int r) {
    if (l + 1 == r) return a[l];
    int mid = (l + r) / 2;
    int x = solve(l, mid);
    int y = solve(mid, r);
    return max(x, y);
}
```

这里左半段是 `[l,mid)`，右半段是 `[mid,r)`。它们拼起来正好是 `[l,r)`。

两种写法都可以，但不能混用。闭区间常用终止条件 `l == r`，左闭右开区间常用终止条件 `l + 1 == r`。如果把两种写法混在一起，常见错误是漏掉 `mid`，或让 `mid` 被两个子问题重复处理。

例题 1：判断区间是否覆盖完整

下面代码使用闭区间：

```
int solve(int l, int r) {
    if (l == r) return a[l];
    int mid = (l + r) / 2;
    int x = solve(l, mid - 1);
    int y = solve(mid + 1, r);
    return max(x, y);
}
```

这段代码有问题。因为 `mid` 位置既不在左区间 `[l,mid-1]` 中，也不在右区间 `[mid+1,r]` 中，被漏掉了。

例如区间 `[1,3]`，`mid=2`。左区间是 `[1,1]`，右区间是 `[3,3]`，位置 2 没有被处理。

正确的闭区间拆法通常是：

```
solve(l, mid);
solve(mid + 1, r);
```

或者：

```
solve(l, mid - 1);
solve(mid, r);
```

但后者需要保证边界不会出问题，考试中更常见的是前一种。

递归边界

分治程序一定要有边界。边界通常对应“问题已经小到可以直接回答”。

对于区间分治，常见边界有：

区间写法	常见边界	含义
[l,r]	l == r	区间只有一个元素
[l,r)	l + 1 == r	区间只有一个元素
[l,r]	l > r	空区间
[l,r)	l >= r	空区间

有些分治程序需要处理空区间，有些不需要。要看函数是否可能被传入空区间。

例题 2：最大子段和中的空区间

下面程序用分治求最大子段和，允许空子段，空子段和为 0。

```
int solve(int l, int r) {
    if (l > r) return 0;
    if (l == r) return max(a[l], 0);
    int mid = (l + r) / 2;
    int left = solve(l, mid);
    int right = solve(mid + 1, r);
    int bestl = 0, s = 0;
    for (int i = mid; i >= l; i--) {
        s += a[i];
        bestl = max(bestl, s);
    }
    int bestr = 0;
    s = 0;
    for (int i = mid + 1; i <= r; i++) {
        s += a[i];
        bestr = max(bestr, s);
    }
    return max(max(left, right), bestl + bestr);
}
```

这里 l > r 返回 0，是因为题目允许空子段。若题目要求子段必须非空，这个边界就不能简单返回 0，否则全是负数时会得到错误答案。

分治边界不是固定模板，要和题目要求一致。

合并

分治的难点往往不在“拆”，而在“合并”。

求最大值时，合并很简单：

```
return max(x, y);
```

求区间和时，合并也很简单：

```
return x + y;
```

但有些问题只看左右子问题答案还不够，还要考虑“跨过中点”的情况。

例题 3：最大子段和

给定数组，求一段连续子数组的最大和。可以用分治处理区间 `[l,r]`。

设 `mid=(l+r)/2`。最大子段有三种可能：

- 完全在左半段；
- 完全在右半段；
- 跨过中点。

前两种可以递归求。第三种必须额外计算。

跨过中点的最大子段，一定由“以 `mid` 结尾的最大后缀”和“以 `mid+1` 开头的最大前缀”拼成。

代码如下：

```
int solve(int l, int r) {
    if (l == r) return a[l];

    int mid = (l + r) / 2;
    int left = solve(l, mid);
    int right = solve(mid + 1, r);

    int bestl = a[mid], s = 0;
    for (int i = mid; i >= l; i--) {
        s += a[i];
        bestl = max(bestl, s);
    }

    int bestr = a[mid + 1];
    s = 0;
    for (int i = mid + 1; i <= r; i++) {
        s += a[i];
        bestr = max(bestr, s);
    }

    int cross = bestl + bestr;
    return max(max(left, right), cross);
}
```

这段程序的合并部分不是简单地 `left + right` 或 `max(left,right)`。因为答案可能跨过中点，必须单独处理。

做分治题时，最关键的一问是：原问题的答案是否可能由多个子问题共同组成？如果可能，合并时就要补上跨区间或跨子问题的情况。

变式：允许空子段

若题目允许选择空子段，最大子段和至少为 0。边界和合并都要调整。单点区间可以返回 `max(a[l], 0)`，跨中点的前缀、后缀也可以从 0 开始。

题目要求“非空”还是“可以为空”，会影响边界和初始化。完善程序题常把这些地方设为空缺。

分治复杂度

分治程序的复杂度通常可以写成递归式。

例题 4：求最大值

```
int solve(int l, int r) {
    if (l == r) return a[l];
    int mid = (l + r) / 2;
    return max(solve(l, mid), solve(mid + 1, r));
}
```

设区间长度为 n。每次拆成两个长度约为 $n/2$ 的子问题，合并只做常数次比较。

递归式为：

$$T(n) = 2T(n/2) + O(1).$$

用第 5 章的主定理，$a = 2, b = 2$，$g(n) = n$，$f(n) = O(1)$ 比 $g(n)$ 小，所以复杂度为 $O(n)$。

这也符合直觉：每个元素最终都会作为单点区间被访问一次，总量是线性的。

例题 5：最大子段和分治

上一节最大子段和分治中，每层除了递归处理左右两半，还要从中点向左、向右扫描一遍，合并代价为 $O(n)$。

递归式为：

$$T(n) = 2T(n/2) + O(n).$$

由主定理可得复杂度为 $O(n \log n)$。

这不是最大子段和的最优做法，但它很适合说明分治合并的复杂度：每层都要花 $O(n)$ 合并，一共有 $O(\log n)$ 层。

例题 6：本层代价不同

如果某个分治程序满足：

$$T(n) = 2T(n/2) + O(n^2),$$

则复杂度是 $O(n^2)$，不是 $O(n^2 \log n)$。原因已经在第 5 章讲过：每层总代价逐层减半，总量由上层主导。

复杂度题不能只看 “有 $\log n$ 层”，还要看每层总代价如何变化。

分治和普通递归

不是所有递归都是分治。

```
int f(int n) {
    if (n == 1) return 1;
    return f(n - 1) + n;
}
```

这个递归每次只把问题规模减 1，没有把问题拆成多个子问题，也没有合并多个子问题答案。它更像普通递归，不是典型分治。

```
int solve(int l, int r) {
    if (l == r) return a[l];
    int mid = (l + r) / 2;
    int x = solve(l, mid);
    int y = solve(mid + 1, r);
    return max(x, y);
}
```

这个程序把区间拆成两个子区间，分别递归，再合并结果，才是典型分治。

判断是否是分治，不看函数是否递归调用自己，而看它是否把一个问题拆成若干个规模较小、结构相同的子问题。

分治和二分

分治和二分都常出现 `mid`，但它们不是一回事。

分治通常会处理多个子问题。例如：

```
solve(l, mid);
solve(mid + 1, r);
```

二分通常只保留一边。例如：

```
if (check(mid)) r = mid;
else l = mid + 1;
```

分治关心“左右两边的答案怎样合并”；二分关心“答案在哪一边”。

下一章会系统讲二分。本章只要先分清：看到 `mid` 不一定是二分，也可能是分治；看到递归拆成两半，也不一定是在查找答案，可能是在合并区间信息。

程序阅读中的分治

读分治程序，可以按下面顺序。

第一，确定函数含义。比如 `solve(l,r)` 是求区间最大值、区间和、最大子段和，还是统计某种数量。

第二，确定边界。看 `l == r`、`l + 1 == r`、`l > r` 等条件分别表示什么。

第三，确定区间写法。闭区间和左闭右开区间不能混。

第四，确定递归调用。看子区间是否覆盖完整，是否重叠，是否遗漏中点。

第五，确定合并。看左右答案怎样组合，是否需要处理跨中点情况。

第六，判断复杂度。写出递归式，再用主定理或按层统计。

例题 7：读懂返回值

```
int solve(int l, int r) {
    if (l == r) return 1;
    int mid = (l + r) / 2;
    int x = solve(l, mid);
```

```
    int y = solve(mid + 1, r);
    return x + y;
}
```

调用 `solve(1,n)`，返回值是多少？

这个函数不看数组，只看区间。单点区间返回 1。非单点区间返回左右子区间返回值之和。

因此它返回的是区间中的元素个数。`solve(1,n)` 返回 n。

这种题不需要把递归树全部画完。只要看懂返回值含义，就能判断。

例题 8：子区间是否重叠

```
int solve(int l, int r) {
    if (l == r) return a[l];
    int mid = (l + r) / 2;
    int x = solve(l, mid);
    int y = solve(mid, r);
    return max(x, y);
}
```

这段代码有风险。使用闭区间时，左区间 `[l,mid]` 和右区间 `[mid,r]` 都包含 `mid`。当 `l+1==r` 时，`mid=l`，右区间变成 `[l,r]`，和原区间一样，递归规模没有变小，程序无法正常结束。

闭区间常用拆法是：

```
solve(l, mid);
solve(mid + 1, r);
```

左闭右开区间常用拆法是：

```
solve(l, mid);
solve(mid, r);
```

看到 `solve(l, mid)` 和 `solve(mid, r)`，要先判断函数使用的是哪种区间写法。

完善程序中的分治

完善程序题中，分治常在这些位置留空：

- 递归边界；
- `mid` 的计算；
- 左右子区间；
- 跨中点的合并；
- 返回值；
- 复杂度相关判断。

做这类题时，不要先套模板。先判断 `solve(l,r)` 的含义，再看空缺处承担什么职责。

例题 9：补子区间

下面程序使用闭区间求区间和。

```
int solve(int l, int r) {
    if (l == r) return a[l];
    int mid = (l + r) / 2;
    int x = solve(l, mid);
    int y = solve(_____);
    return x + y;
}
```

空缺应填：

```
mid + 1, r
```

因为左半段已经处理 `[l,mid]`，右半段应处理 `[mid+1,r]`。若填 `mid,r`，会重复处理 `mid`，并且在区间长度为 2 时可能无法缩小规模。

例题 10：补合并式

下面程序求区间最大值。

```
int solve(int l, int r) {
    if (l == r) return a[l];
    int mid = (l + r) / 2;
    int x = solve(l, mid);
    int y = solve(mid + 1, r);
    return _____;
}
```

空缺应填：

```
max(x, y)
```

因为整个区间最大值要么在左半段，要么在右半段。这个问题不需要跨中点额外合并。

若题目变成最大子段和，返回值就不能只写 `max(x,y)`，因为答案可能跨过中点。完善程序题的关键是先确定函数到底返回什么。

原创例题

例题 11：分治求数组和

阅读程序：

```
int solve(int l, int r) {
    if (l == r) return a[l];
    int mid = (l + r) / 2;
    return solve(l, mid) + solve(mid + 1, r);
}
```

若数组为：

```
a[1]=3, a[2]=1, a[3]=4, a[4]=2
```

调用 `solve(1,4)` 的返回值是多少？

递归拆分为：

```
solve(1,4)
= solve(1,2) + solve(3,4)
= (solve(1,1)+solve(2,2)) + (solve(3,3)+solve(4,4))
= 3 + 1 + 4 + 2
= 10
```

所以返回 10。

这个程序的函数含义是“求区间和”。只要这个含义确定，后面很多问题都能直接回答。

变式 1：改变边界

```
int solve(int l, int r) {
    if (l > r) return 0;
    if (l == r) return a[l];
    int mid = (l + r) / 2;
    return solve(l, mid - 1) + a[mid] + solve(mid + 1, r);
}
```

这段程序仍然求区间和。它把中点单独拿出来，左边处理 `[l,mid-1]`，右边处理 `[mid+1,r]`。因为允许空区间，所以需要 `l > r` 返回 0。

这说明分治拆法不止一种。只要子问题和单独处理的部分合起来正好覆盖原区间，就可以。

变式 2：复杂度判断

上面变式中，每次递归处理左右两边，中点只做常数次处理。递归式仍可看成：

$$T(n) = 2T(n/2) + O(1).$$

复杂度为 $O(n)$。

虽然递归层数是 $O(\log n)$，但所有单点元素最终各被处理一次，整体仍是线性级别。

真题式训练

例题 12：分治合并中的跨区间

下面程序用于求非空最大子段和的一部分，空缺处应填什么？

```
int solve(int l, int r) {
    if (l == r) return a[l];
    int mid = (l + r) / 2;
    int ans = max(solve(l, mid), solve(mid + 1, r));

    int s = 0, bestl = a[mid];
    for (int i = mid; i >= l; i--) {
        s += a[i];
        bestl = max(bestl, s);
    }

    s = 0;
    int bestr = a[mid + 1];
    for (int i = mid + 1; i <= r; i++) {
        s += a[i];
        bestr = max(bestr, s);
```

```
    }

    return max(ans, _____);
}
```

这里 ans 已经处理了完全在左半段或完全在右半段的答案。还缺少跨过中点的答案。跨中点的最大子段由左侧最大后缀和右侧最大前缀组成，所以空缺应填：

```
bestl + bestr
```

若填 max(bestl, bestr)，就只取了一侧，没有形成跨中点子段；若填 ans + bestl + bestr，会把已经求出的答案重复加入。

这类题的关键是把三种情况分清楚：左边、右边、跨中点。

常见错误

分治题常见错误集中在以下几类。

第一，边界写错。闭区间和左闭右开区间混用，导致漏元素、重复元素或递归无法缩小。

第二，只递归不合并。分治不是把问题拆完就结束，必须把子问题答案组合回来。

第三，漏掉跨中点情况。最大子段和、逆序对统计、区间相关计数中，答案可能横跨左右两半。

第四，复杂度误判。看到递归层数是 $O(\log n)$ 就直接乘上最大层代价，可能得到错误结论。要看每层总代价如何变化。

第五，把二分当分治。二分通常只保留一个方向，分治通常处理多个子问题。

第六，没有看返回值含义。同样是 solve(l,r)，可能返回最大值、和、方案数、结构信息。返回值含义不同，合并方式不同。

小结

遇到分治题，可以按下面顺序检查。

步骤	检查问题
函数含义	solve(l,r) 解决什么问题
区间写法	是闭区间，还是左闭右开区间
递归边界	区间多小时直接返回
子问题划分	左右区间是否覆盖完整，是否重叠
合并方式	子问题答案怎样组成原问题答案
跨区间情况	是否有答案横跨中点
复杂度	能否写成 $T(n)=aT(n/b)+f(n)$
程序行为	递归规模是否一定变小
完善程序	空缺处负责边界、划分、合并，还是返回

分治的重点不在“写出递归”这件事本身，而在于拆得完整、边界正确、合并准确。第一轮中，只要能抓住函数含义、区间边界和合并方式，大多数分治程序阅读和填空题都会清楚很多。

0x08 二分

二分是一种不断缩小范围的方法。它每次取中间位置，通过一次判断排除一半左右的范围。由于范围缩小很快，二分的复杂度通常是 $O(\log n)$ 或 $O(\log V)$，其中 V 是答案范围大小。

CSP-S 第一轮中，二分常以三种方式出现：选择题中判断复杂度或边界，程序阅读中给出一段二分代码要求判断输出，完善程序中要求补出判断条件或左右端点更新。二分题的难点通常不在“会不会想到二分”，而在边界是否写对、等号属于哪一边、最终返回哪个变量。

本章默认优先使用左闭右开区间 `[l,r)`。这种写法在纸面分析时比较统一：`l` 可以成为答案，`r` 不在当前候选范围内；循环结束时 `l == r`，这个位置就是第一个满足条件的位置。为了读懂真题代码，本章也会说明闭区间写法的判断方法，但主线使用左闭右开。

问题

先看一个数组查找问题。

给定一个单调不降数组 `a[0..n-1]`，要求找出第一个大于等于 `x` 的位置。如果不存在，返回 `n`。

例如：

```
a = 1 3 3 5 8
x = 4
```

第一个大于等于 4 的数是 5，位置为 3。

如果从头扫一遍，复杂度是 $O(n)$。二分可以把范围不断减半：

```
int lower_bound_pos(int a[], int n, int x) {
    int l = 0, r = n;
    while (l < r) {
        int mid = l + (r - l) / 2;
        if (a[mid] >= x) r = mid;
        else l = mid + 1;
    }
    return l;
}
```

这里维护的是左闭右开区间 `[l,r)`。初始时答案可能在 `[0,n)` 中，也可能是 n。把 `r` 设成 `n`，就能表示“所有元素都小于 `x`”时答案为 `n`。

循环结束时 `l == r`，返回 `l`。

二分的关键不是背代码，而是理解每次判断后为什么可以删掉一半范围。

单调性

二分需要单调性。所谓单调性，就是答案范围可以被某个判断切成两段。

对于“找第一个大于等于 `x` 的位置”，判断条件是：

```
a[mid] >= x
```

因为数组单调不降，所以如果 `a[mid] >= x` 成立，那么 `mid` 右边的位置也都不小于 `x`，答案不会在 `mid` 右边更靠后的位置第一次出现；答案可能是 `mid`，也可能在左边。因此保留 `[l,mid]`，在左闭右开写法中写成：

```
r = mid;
```

如果 `a[mid] < x`，说明 `mid` 及其左边都不可能是答案，答案只能在右边，所以写：

```
l = mid + 1;
```

可以把判断结果想成一串布尔值：

```
false false false true true true
```

二分要找的是第一个 `true`。

许多二分题都可以转成这个模型：前面一段不满足，后面一段满足，找分界点。

左闭右开写法

左闭右开区间 `[l,r)` 的含义是：当前候选答案在 `l,l+1,...,r-1` 中。`r` 本身不属于候选范围。

寻找第一个满足条件的位置时，常用模板是：

```
int l = L, r = R;
while (l < r) {
    int mid = l + (r - l) / 2;
    if (check(mid)) r = mid;
    else l = mid + 1;
}
return l;
```

这里要求 `check(pos)` 的结果满足：

```
false false false true true true
```

也就是一旦某个位置满足，右边也满足。

这个模板的含义是：

- `check(mid)` 为真，`mid` 可能是答案，不能丢掉，所以令 `r = mid;`
- `check(mid)` 为假，`mid` 不可能是答案，左边也不可能，所以令 `l = mid + 1`。

例题 1：第一个大于等于

数组：

```
a = 1 3 3 5 8
x = 3
```

找第一个大于等于 3 的位置。

初始 `l=0,r=5`。

轮次	l	r	mid	判断	更新
1	0	5	2	a[2]=3 >= 3	r=2
2	0	2	1	a[1]=3 >= 3	r=1
3	0	1	0	a[0]=1 < 3	l=1

循环结束，l=r=1，答案为 1。

注意答案不是位置 2，因为题目要求第一个大于等于 3 的位置。等号属于“满足”这一边，所以遇到 a[mid] == x 时要向左收缩。

例题 2：不存在时返回 n

数组：

```
a = 1 3 3 5 8
x = 10
```

所有元素都小于 10。每次判断 a[mid] >= 10 都为假，l 会不断右移，最后得到 l=5。返回 5，表示不存在这样的位置。

这也是左闭右开写法常把 r 初始化为 n 的原因：答案可以落在数组末尾之后。

找第一个大于

找第一个大于 x 的位置，只需要把判断条件换成：

```
a[mid] > x
```

代码为：

```
int upper_bound_pos(int a[], int n, int x) {
    int l = 0, r = n;
    while (l < r) {
        int mid = l + (r - l) / 2;
        if (a[mid] > x) r = mid;
        else l = mid + 1;
    }
    return l;
}
```

与“第一个大于等于”相比，唯一变化是等号属于哪一边。

例题 3：等号的区别

数组：

```
a = 1 3 3 5 8
x = 3
```

第一个大于等于 3 的位置是 1。

第一个大于 3 的位置是 3。

对应条件分别是：

```
a[mid] >= x
a[mid] > x
```

这两个条件只差一个等号，但答案可能相差很多。完善程序题中，如果选项只差 `>` 和 `>=`，多半就在考边界归属。

找最后一个满足

很多题问的是最后一个小于等于 `x` 的位置。可以直接写闭区间二分，也可以继续用左闭右开，把它转成“第一个大于 `x` 的位置减一”。

```
int p = upper_bound_pos(a, n, x) - 1;
```

如果 `p >= 0`，那么 `p` 就是最后一个小于等于 `x` 的位置；如果 `p == -1`，说明没有元素小于等于 `x`。

这种转化很适合纸面判断，因为仍然使用“找第一个 true”的统一模型。

例题 4：最后一个小于等于

数组：

```
a = 1 3 3 5 8
x = 3
```

第一个大于 3 的位置是 3，所以最后一个小于等于 3 的位置是 2。

若数组为：

```
a = 5 6 7
x = 3
```

第一个大于 3 的位置是 0，减一后得到 −1，表示不存在。

二分题中，若目标是“最后一个满足”，可以先思考能否转成“第一个不满足”再减一。

闭区间写法

有些真题或程序片段使用闭区间 `[l,r]`。这种写法中，`l` 和 `r` 都在候选范围内。

常见写法如下：

```
int l = 0, r = n - 1, ans = n;
while (l <= r) {
    int mid = l + (r - l) / 2;
    if (a[mid] >= x) {
        ans = mid;
        r = mid - 1;
    } else {
        l = mid + 1;
    }
}
return ans;
```

这里 `ans` 负责保存当前找到的可行位置。若 `a[mid] >= x`，`mid` 是一个候选答案，但还要继续向左找，所以 `ans = mid`，再令 `r = mid - 1`。若不满足，就向右找。

闭区间写法能写对，但需要额外维护 ans，并且循环条件是 l <= r。左闭右开写法通常在循环结束时直接返回 l，结构更统一。

读真题代码时，要先判断它是哪种区间：

写法	候选范围	循环条件	常见返回
左闭右开	[l,r)	l < r	l
闭区间	[l,r]	l <= r	ans 或 l

不要把两种写法混在一起。比如左闭右开中写 r = mid - 1，就可能跳过答案；闭区间中写 r = mid，若配合 l <= r，可能无法正确缩小范围。

二分答案

二分不只用于有序数组。只要答案具有单调性，就可以二分答案。

典型形式是：给定一个值 x，可以用 check(x) 判断 x 是否可行。如果 x 可行时，更大的值也都可行，那么可以找最小可行值。

例题 5：最小可行值

有 n 个任务，按顺序分成若干段，每段任务量之和不能超过 M。问至少需要多少段。若要求最多分成 k 段，求最小的 M。

对于一个给定的 M，可以从左到右贪心分段：

```
bool check(int m) {
    int cnt = 1;
    int sum = 0;
    for (int i = 1; i <= n; i++) {
        if (a[i] > m) return false;
        if (sum + a[i] <= m) {
            sum += a[i];
        } else {
            cnt++;
            sum = a[i];
        }
    }
    return cnt <= k;
}
```

如果某个 M 可行，那么更大的 M 也可行，因为每段容量变大，不会让分段变困难。于是 check(M) 的结果呈现：

```
false false false true true true
```

可以二分最小的可行 M：

```
int l = 0, r = 0;
for (int i = 1; i <= n; i++) {
    l = max(l, a[i]);
    r += a[i];
}

while (l < r) {
    int mid = l + (r - l) / 2;
    if (check(mid)) r = mid;
    else l = mid + 1;
}
```

```
cout << l << endl;
```

这里左边界至少是单个任务的最大值，右边界可以取所有任务量之和。因为一个段必须能放下最大任务，而把所有任务放进一段时容量为总和。

这类题的核心不是数组有序，而是答案判断具有单调性。

最大可行值

有些题要找最大可行值。若 check(x) 呈现：

```
true true true false false false
```

可以转成找第一个不可行值，再减一。

例如要找最大的 x，使得 check(x) 为真。可以写：

```
int l = L, r = R + 1;
while (l < r) {
    int mid = l + (r - l) / 2;
    if (!check(mid)) r = mid;
    else l = mid + 1;
}
int ans = l - 1;
```

这里二分的是第一个 false。循环结束时 l 是第一个不可行值，所以最大可行值是 l-1。

也可以写成偏右取中点的闭区间形式，但在第一轮教材中，建议先掌握“找第一个分界点”的统一方法。最大可行值转成第一个不可行值，边界会更清楚。

check 函数

二分答案题中，check 函数决定了整个程序的含义。读二分程序时，不要只看 while，还要看 check。

要问三件事：

- check(x) 为真表示什么？
- 若 x 变大，check(x) 会怎样变化？
- 程序是在找第一个真，还是第一个假？

例题 6：判断方向

下面程序：

```
while (l < r) {
    int mid = l + (r - l) / 2;
    if (check(mid)) r = mid;
    else l = mid + 1;
}
```

它找的是第一个使 check(mid) 为真的位置。因此 check 的结果应当是：

```
false false false true true true
```

如果实际 check 的含义是“当前值仍然可行，值越大越不容易可行”，结果呈现：

```
true true true false false false
```

就不能直接套这段代码。要么取反，找第一个不可行；要么改写二分更新方式。

完善程序题中，经常把 `check(mid)`、`!check(mid)`、`rank(mid) < k`、`rank(mid) >= k` 放成选项。判断时要先写出 `check` 的真假序列。

第 k 小问题中的二分

有一类常见题：要求第 k 小的数，但这些数没有直接列出来。做法常常是二分答案 `x`，计算“小于等于 `x` 的数有多少个”。

如果 `cnt(x)` 表示小于等于 `x` 的数的个数，那么 `x` 越大，`cnt(x)` 不会变小。要求第 k 小，就是找最小的 `x`，使得：

```
cnt(x) >= k
```

二分代码为：

```
while (l < r) {
    int mid = l + (r - l) / 2;
    if (cnt(mid) >= k) r = mid;
    else l = mid + 1;
}
```

例题 7：乘法表第 k 小

有一个 $n \times n$ 的乘法表，第 i 行第 j 列为 $i \cdot j$。要求第 k 小的数。

对于给定的 `x`，第 `i` 行中不超过 `x` 的数有：

```
min(n, x / i)
```

个。因此：

```
long long cnt(int x) {
    long long s = 0;
    for (int i = 1; i <= n; i++) {
        s += min(n, x / i);
    }
    return s;
}
```

然后找最小的 `x`，使得 `cnt(x) >= k`。

这类题中，`cnt(x) == k` 时不能直接返回 `x`，因为可能有重复值，也可能更小的数已经让计数达到 k。稳妥做法是继续向左找最小可行值。

手算二分过程

程序阅读题有时给出具体数组和目标值，要求判断二分过程或输出。此时要按代码实际更新过程追踪。

例题 8：追踪代码

```cpp
int l = 0, r = 6;
while (l < r) {
    int mid = (l + r) / 2;
    if (a[mid] >= x) r = mid;
    else l = mid + 1;
}
cout << l << endl;
```

数组：

```
下标：0 1 2 3 4 5
数值：2 4 4 7 9 9
```

令 `x=5`。

追踪：

轮次	l	r	mid	a[mid]	更新
1	0	6	3	7	r=3
2	0	3	1	4	l=2
3	2	3	2	4	l=3

结束时输出 3。对应第一个大于等于 5 的位置，值为 7。

手算二分时，表格不必很大，但 `l,r,mid` 三列最好写清楚。很多错误都来自某一轮更新后忘记新边界。

二分的复杂度

二分每次把区间长度大约缩小一半。若初始范围长度为 n，经过 t 次后长度约为 $n/2^t$。当长度变为 1 时，有：

$$\frac{n}{2^t} \le 1.$$

所以 $t = O(\log n)$。

数组二分中，每次判断通常是 $O(1)$，所以总复杂度是 $O(\log n)$。

二分答案中，每次判断可能需要额外计算。若答案范围大小为 V，`check` 的复杂度为 $O(C)$，总复杂度通常为：

$$O(C \log V).$$

例如上一节分段任务的 `check` 需要扫一遍数组，是 $O(n)$；答案范围是从最大任务量到总和，记为 V。总复杂度是 $O(n \log V)$。

复杂度题中，要把二分次数和每次检查的代价乘起来。不要只看到二分就写 $O(\log n)$。

程序阅读中的二分

读二分程序，可以按下面顺序：

1. 看初始 `l,r`，判断搜索范围；
2. 看循环条件，判断区间写法；
3. 看 `mid` 的计算方式；

4. 看判断条件，写出真假序列；
5. 看更新语句，判断找第一个真、最后一个真，还是其他分界点；
6. 看返回值或输出值。

例题 9：判断程序含义

```
int f(int a[], int n, int x) {
    int l = 0, r = n;
    while (l < r) {
        int mid = l + (r - l) / 2;
        if (a[mid] > x) r = mid;
        else l = mid + 1;
    }
    return l;
}
```

循环条件是 `l < r`，初始 `r=n`，是左闭右开写法。

判断条件是 `a[mid] > x`。为真时保留左边和 `mid`，为假时丢掉 `mid` 及左边。它找的是第一个大于 `x` 的位置。

若题目问“它是否等价于找第一个大于等于 `x` 的位置”，答案是否定的。等号被放在了不满足的一侧。

完善程序中的二分

完善程序题中，二分常在以下位置留空：

- 初始左右边界；
- `while` 条件；
- `mid` 计算；
- 判断条件；
- `l` 或 `r` 更新；
- 最终返回值。

做这类题时，要先判断目标：找第一个满足，最后一个满足，还是最小可行答案。

例题 10：补判断条件

给定单调不降数组，找第一个大于等于 `x` 的位置。

```
int l = 0, r = n;
while (l < r) {
    int mid = l + (r - l) / 2;
    if (_____) r = mid;
    else l = mid + 1;
}
return l;
```

空缺应填：

```
a[mid] >= x
```

因为这段代码在条件为真时保留 `mid`，说明 `mid` 可能是答案。对于“第一个大于等于 `x`”，当 `a[mid] >= x` 时，`mid` 确实可能成为答案。

例题 11：补更新语句

找最小可行答案。`check(mid)` 为真表示 `mid` 可行，且更大的值也可行。

```
while (l < r) {
    int mid = l + (r - l) / 2;
    if (check(mid)) {
        _____;
    } else {
        _____;
    }
}
```

应填：

```
r = mid
l = mid + 1
```

因为 `mid` 可行时，答案可能就是 `mid`，不能排除；`mid` 不可行时，`mid` 和更小的值都不可行，应从右边继续找。

常见错误

二分题的错误很集中。

第一，条件方向反了。`check(mid)` 为真时，到底应该向左找还是向右找，要看题目是找最小可行还是最大可行。

第二，等号放错。`>=` 和 `>`，`<` 和 `<=` 会改变边界位置。

第三，区间写法混用。左闭右开写法中常用 `r=mid`，闭区间写法中常用 `r=mid-1`。混用容易漏答案或循环无法结束。

第四，返回值错。左闭右开找第一个满足通常返回 `l`；闭区间写法可能返回 `ans`，也可能返回 `l`，要看代码维护的含义。

第五，二分答案时没有证明单调。若 `check(x)` 的真假不是连续两段，二分就不能保证正确。

第六，只算二分次数，漏掉 `check` 复杂度。二分答案总复杂度要乘上每次检查的代价。

第七，最大值问题没有转化清楚。找最大可行值时，可以找第一个不可行值再减一，避免边界混乱。

原创例题

例题 12：最小速度

有 n 段路程，第 i 段长度为 `a[i]`。每天最多走 v 的路程，必须按顺序走，走完一段后才能走下一段。若要求在不超过 k 天内走完，求最小的 v。

对于给定的 v，可以计算需要多少天：

```
bool check(int v) {
    int day = 1;
    int sum = 0;
    for (int i = 1; i <= n; i++) {
        if (a[i] > v) return false;
        if (sum + a[i] <= v) {
            sum += a[i];
        } else {
            day++;
            sum = a[i];
```

```
        }
    }
    return day <= k;
}
```

若速度 v 可行，那么更大的速度也可行。因此可以二分最小可行速度：

```
int l = 0, r = 0;
for (int i = 1; i <= n; i++) {
    l = max(l, a[i]);
    r += a[i];
}

while (l < r) {
    int mid = l + (r - l) / 2;
    if (check(mid)) r = mid;
    else l = mid + 1;
}
cout << l << endl;
```

这里 `l` 从最大单段长度开始，`r` 从总长度开始，都是有意义的边界。若从 `l=0` 开始也能借助 `check` 排除一部分，但边界含义不如现在清楚。

变式 1：最多天数变少

如果把要求改成不超过 $k-1$ 天，答案不会变小。因为限制更严格，需要的最小速度可能变大。

这说明 `check(v)` 的单调方向仍然是：速度越大越容易满足。

变式 2：问最大允许天数

如果题目改问“给定速度 v，最多能允许多少天”，就不是二分答案问题，而是直接模拟计算所需天数。不是所有带“最大”“最小”的题都要二分，必须先看是否存在需要寻找的单调分界点。

小结

遇到二分题，可以按下面顺序检查。

步骤	检查问题
目标	找第一个满足、最后一个满足，还是最小可行答案
单调性	判断结果是否能分成连续两段
区间	使用左闭右开还是闭区间
条件	`check(mid)` 为真表示什么
等号	刚好相等时应算满足还是不满足
更新	`mid` 是否可能成为答案，能否被丢掉
返回	循环结束后应该返回 `l`、`r`、`l-1` 还是 `ans`
复杂度	二分次数乘上 `check` 代价
读程序	按代码实际过程追踪 `l,r,mid`

二分的代码很短，但边界要求很高。学习时不要只背模板，要反复问：当前区间表示什么，`mid` 能不能成为答案，等号属于哪一侧。只要这三个问题答清楚，二分题的程序阅读和完善程序都会稳定很多。

0x09 排序与比较次数

排序是 CSP-S 第一轮中很常见的考点，但它不一定以“写排序程序”的形式出现。更多时候，题目会问排序算法的稳定性、最坏复杂度、比较次数，或者把排序作为程序阅读的一部分，让考生判断排序后的数组、某个变量的值、某段代码是否仍然正确。

本章不把排序算法写成模板合集。第一轮更需要掌握的是：不同排序方法的性质，比较次数怎样估算，排序在程序中改变了哪些条件。后面讲数据结构、图论、字符串和 DP 时，排序也会作为预处理或贪心依据反复出现。

排序题考什么

排序相关题目常见考法有五类。

考法	典型问题
稳定性	哪些排序稳定，哪些不稳定
复杂度	最好、平均、最坏时间复杂度
比较次数	至少需要比较多少次，某种算法比较几次
程序行为	排序后数组变成什么，循环还能否正确
排序思想	归并、快速、堆、插入、选择、冒泡的过程差异

提高组第一轮不会要求把所有排序算法的代码细节背下来，但会要求学生分清它们的基本性质。比如：

- 插入排序、冒泡排序、归并排序通常是稳定的；
- 选择排序、快速排序、堆排序通常是不稳定的；
- 归并排序最坏复杂度为 $O(n \log n)$；
- 朴素快速排序在特殊输入下可能退化到 $O(n^2)$；
- 比较排序的最坏比较次数有下界，不能无限低。

这里的“通常”是指常见教材实现。稳定性和具体实现有关，但第一轮题目一般按常见实现考查。

稳定性

排序稳定性只讨论“关键字相等的元素”的相对顺序。

若排序前有两个元素 x, y，它们的关键字相等，并且 x 在 y 前面；排序后 x 仍在 y 前面，则这对元素的相对顺序被保持。若所有相等关键字元素都能保持原相对顺序，这个排序就是稳定的。

例如有一组学生记录，只按分数排序：

```
排序前：
A:90  B:80  C:90  D:70
```

若按分数从小到大排序，A 和 C 的分数相同。稳定排序的结果中，A 仍应在 C 前面：

```
D:70  B:80  A:90  C:90
```

若排序后变成：

```
D:70  B:80  C:90  A:90
```

则 A 和 C 的相对顺序被改变，这种实现就不是稳定的。

稳定性只和相等关键字有关。对于关键字不同的元素，排序本来就可能改变它们的位置，不能据此判断稳定性。

常见排序稳定性

排序方法	常见稳定性	说明
冒泡排序	稳定	只在相邻逆序时交换，相等时不交换
插入排序	稳定	插入时相等元素不越过原有相等元素
归并排序	稳定	合并时相等元素先取左边
选择排序	不稳定	一次交换可能跨过相等元素
快速排序	不稳定	分区交换可能改变相等元素相对顺序
堆排序	不稳定	堆调整可能远距离交换
计数排序	可稳定	按稳定写法可保持相对顺序
基数排序	依赖稳定子排序	每一位排序通常要求稳定

这张表适合第一轮记忆。若题目给出具体代码，则要按代码判断，不能只按算法名判断。

例题 1：选择排序为什么可能不稳定

数组中每个元素写成“关键字 + 编号”：

```
2A  2B  1C
```

按关键字从小到大排序。选择排序第一轮会找到最小元素 `1C`，并把它和第一个位置交换：

```
1C  2B  2A
```

原来 `2A` 在 `2B` 前面，排序后 `2B` 在 `2A` 前面。两个关键字同为 2 的元素相对顺序改变，所以这种选择排序不稳定。

判断不稳定时，只要给出一个反例即可。

例题 2：冒泡排序为什么稳定

冒泡排序常见写法只在相邻元素逆序时交换：

```
if (a[j] > a[j + 1]) swap(a[j], a[j + 1]);
```

若两个元素关键字相等，条件 `a[j] > a[j + 1]` 不成立，它们不会因为彼此相等而交换。相等元素只能跟别的更大或更小元素交换位置，但相等元素之间不会互相越过，所以这种写法是稳定的。

若代码把条件写成：

```
if (a[j] >= a[j + 1]) swap(a[j], a[j + 1]);
```

相等元素也会交换，相对顺序可能改变，稳定性就不再成立。程序阅读题中要看具体比较符号。

比较排序的次数

很多排序算法通过比较元素大小来决定顺序，这类排序称为比较排序。插入排序、选择排序、冒泡排序、归并排序、快速排序、堆排序都属于比较排序。

对于比较排序，第一轮常考两类问题：

- 某个具体算法做了多少次比较；
- 找最大值、最小值，或者同时找最大值和最小值，至少需要多少次比较。

先看最基础的问题：在 n 个数中找最大值。

```
int mx = a[1];
for (int i = 2; i <= n; i++) {
    if (a[i] > mx) mx = a[i];
}
```

每个后来的数都要和当前最大值比较一次，共比较 $n-1$ 次。这个次数也是最坏情况下至少需要的比较次数。因为如果某个元素从来没有参与比较并输给别人，就无法确认它不是最大值。

所以，找最大值至少需要 $n-1$ 次比较；找最小值也至少需要 $n-1$ 次比较。

同时找最大值和最小值

若分别找最大值和最小值，需要：

```
(n - 1) + (n - 1) = 2n - 2
```

次比较。

但可以更省。把元素两两分组，先在每组内部比较一次，较大的去竞争最大值，较小的去竞争最小值。

以 n 为偶数为例：

- 组内比较：$n/2$ 次；
- 每组较大者共有 $n/2$ 个，找最大值需要 $n/2-1$ 次；
- 每组较小者共有 $n/2$ 个，找最小值需要 $n/2-1$ 次。

总比较次数为：

$$\frac{n}{2}+\left(\frac{n}{2}-1\right)+\left(\frac{n}{2}-1\right)=\frac{3n}{2}-2.$$

若题目说“有 $2n$ 个数，同时找最大值和最小值”，代入上式得到：

$$\frac{3\cdot 2n}{2}-2=3n-2.$$

这是第一轮很常见的选择题考点。

例题 3：8 个数同时找最大和最小

有 8 个数，同时找最大值和最小值，最少需要多少次比较？

按两两分组：

- 组内比较 4 次；
- 4 个较大者找最大值，需要 3 次；
- 4 个较小者找最小值，需要 3 次。

总次数为 $4+3+3=10$。

若分别找最大和最小，则需要 $7+7=14$ 次。分组方法节省了 4 次比较。

常见排序的比较次数

第一轮不要求精确掌握所有排序在所有输入下的比较次数，但几个典型结果要熟悉。

选择排序

选择排序每一轮从未排序部分中找最小值，放到当前位置。

第 1 轮要在 n 个数中找最小值，需要 $n-1$ 次比较；第 2 轮在剩下 $n-1$ 个数中找最小值，需要 $n-2$ 次比较；一直到最后。

总比较次数为：

$$(n-1)+(n-2)+\cdots+1=\frac{n(n-1)}{2}.$$

选择排序的比较次数与原数组是否有序关系不大，常见实现中每一轮仍要扫描剩余部分。

复杂度为 $O(n^2)$。

冒泡排序

普通冒泡排序每一轮比较相邻元素，把较大的元素逐步换到后面。

若不加入提前结束标记，第 1 轮比较 $n-1$ 次，第 2 轮比较 $n-2$ 次，总比较次数也是：

$$\frac{n(n-1)}{2}.$$

若加入“本轮没有交换则提前结束”的优化，则在数组已经有序时，只需要一轮扫描，比较 $n-1$ 次，复杂度为 $O(n)$；但最坏情况下仍为 $O(n^2)$。

插入排序

插入排序从左到右维护已排序前缀，把当前元素插入到前面合适位置。

若数组已经有序，每个元素只需和前一个位置比较一次左右，复杂度为 $O(n)$。

若数组逆序，第 i 个元素要一路向前移动，比较次数大致为：

$$1+2+\cdots+(n-1)=\frac{n(n-1)}{2}.$$

最坏复杂度为 $O(n^2)$。

插入排序的表现和原数组有序程度关系很大。第一轮中如果问“已排序数组上插入排序表现如何”，不要直接答 $O(n^2)$，要看是否是最好情况。

归并排序

归并排序把数组分成两半，分别排序，然后合并。

复杂度满足：

$$T(n)=2T(n/2)+O(n).$$

根据主定理，复杂度为 $O(n\log n)$。

归并排序的比较次数也在 $O(n\log n)$ 级别。它的一个重要性质是最坏情况下仍能保持 $O(n\log n)$，不像朴素快速排序那样可能退化到 $O(n^2)$。

快速排序

快速排序选择一个基准，把数组分成小于基准和大于基准的两部分，再递归排序。

如果每次划分比较均衡，递归深度为 $O(\log n)$，每层处理总量为 $O(n)$，复杂度为 $O(n\log n)$。

如果每次基准都选到最小值或最大值，划分会变成一边为空、一边规模减 1，递归式接近：

$$T(n)=T(n-1)+O(n).$$

于是复杂度为 $O(n^2)$。

例如数组已经有序，而快速排序总是选择第一个元素作为基准，就可能出现这种退化。这是 CSP-S 第一轮中的高频判断点。

堆排序

堆排序利用堆维护当前最大值或最小值。建堆后，每次取出堆顶并调整堆。

常见复杂度为 $O(n\log n)$，最坏情况下也保持这个量级。堆排序通常不稳定。

第一轮中，堆排序常和堆结构一起考查：插入、删除堆顶、堆顶元素、调整后的数组形态等。这里先记住它的排序性质，具体堆操作会在数据结构部分讲。

排序算法速查

排序方法	平均复杂度	最坏复杂度	常见稳定性	第一轮常考点
冒泡排序	$O(n^2)$	$O(n^2)$	稳定	相邻交换、提前结束
选择排序	$O(n^2)$	$O(n^2)$	不稳定	比较次数固定
插入排序	$O(n^2)$	$O(n^2)$	稳定	有序时较快
归并排序	$O(n\log n)$	$O(n\log n)$	稳定	分治、合并
快速排序	$O(n\log n)$	$O(n^2)$	不稳定	基准选择、退化
堆排序	$O(n\log n)$	$O(n\log n)$	不稳定	堆调整
计数排序	$O(n+V)$	$O(n+V)$	可稳定	值域大小
基数排序	与位数和值域有关	与位数和值域有关	依赖稳定子排序	按位处理

表中的 V 表示值域大小。计数排序不是比较排序，它的复杂度依赖值域。如果值域很大，计数排序未必适合。

第一轮中，若题目问“以下排序最坏复杂度低于 $O(n^2)$ 的是”，归并排序、堆排序通常是正确方向；快速排序要看问的是平均还是最坏。

排序后的程序行为

排序常常不是题目的最终目标，而是后续算法的前提。程序阅读题中，如果删去排序语句，后面的逻辑可能失效。

例如：

```
sort(a + 1, a + n + 1);

int ans = 0;
for (int i = 1, j = 1; i <= n; i++) {
    while (j <= n && a[j] - a[i] <= k) j++;
    ans += j - i - 1;
}
```

这里双指针依赖数组有序。`j` 能一直向右移动，是因为 `a[j] - a[i]` 随着 `j` 增大不会变小。若删掉排序，`j` 的单调推进就没有依据，答案可能错误。

再看：

```
sort(a + 1, a + n + 1);
cout << a[n] << endl;
```

这段代码排序后输出最大值。若只想输出最大值，其实不需要完整排序，线性扫描也可以。但程序已经排序，复杂度就是 $O(n \log n)$，不能按最优思路写成 $O(n)$。

读程序时，要区分“这个问题可以怎样更好地做”和“这段程序实际做了什么”。

排序和逆序数

相邻交换次数常与逆序数有关。虽然本书后面会专门讲相邻交换与逆序思想，这里先给出排序章节中必须掌握的基本结论。

对于一个排列，逆序对是满足 $i < j$ 且 $a_i > a_j$ 的二元组。

例如：

```
3 1 2
```

逆序对有：

- $(3, 1)$；
- $(3, 2)$。

共有 2 个。

如果每次只能交换相邻两个元素，那么把一个排列变成升序排列所需的最少交换次数，等于它的逆序对个数。

原因是：一次相邻交换最多只改变一对相邻元素的相对顺序，也就是使逆序数改变 1。要把逆序数从初始值降到 0，至少需要这么多次交换；通过不断交换相邻逆序元素，可以做到这个次数。

例题 4：字符串相邻交换

把 `DACFEB` 变成 `ABCDEF`，每次只能交换相邻字符，问最少需要多少次？

把原字符串中的字符转换成目标位置：

字符	D	A	C	F	E	B
目标位置	4	1	3	6	5	2

得到序列：

```
4 1 3 6 5 2
```

计算逆序数：

- 4 后面比它小的有 $1, 3, 2$，共 3 个；
- 1 后面没有比它小的；
- 3 后面比它小的有 2，共 1 个；
- 6 后面比它小的有 $5, 2$，共 2 个；
- 5 后面比它小的有 2，共 1 个。

总数为 $3 + 1 + 2 + 1 = 7$，所以最少需要 7 次相邻交换。

这类题不需要真的一步步交换。先转成目标位置序列，再数逆序对，通常更稳。

比较排序下界

比较排序的一个重要事实是：仅靠比较大小来排序 n 个互不相同的元素，最坏情况下需要 $\Omega(n \log n)$ 次比较。

第一轮中通常不会要求证明得很严密，但要知道它的含义：比较排序不可能在最坏情况下做到 $O(n)$。若某个排序声称只靠比较，却能对任意 n 个数在线性时间内完成，就不符合这个下界。

可以用决策树作直观理解。

n 个互不相同的元素，可能的排列有 $n!$ 种。排序算法必须通过比较，把每一种输入排列区分开。每次比较只有两种结果：小于或大于。若最坏情况下比较 h 次，最多区分 2^h 种情况。因此必须有：

$$2^h \geq n!.$$

所以 h 至少是 $\log_2(n!)$ 量级，而 $\log_2(n!)$ 是 $\Omega(n \log n)$ 级别。

这个结论只适用于比较排序。计数排序、基数排序利用了值域或数字位数信息，不只依赖两两比较，所以不受这个下界限制。

原创例题

例题 5：稳定性判断

有如下数组，按第一个数字从小到大排序：

```
(2,A) (1,B) (2,C) (1,D)
```

若排序结果为：

```
(1,B) (1,D) (2,A) (2,C)
```

这个结果是否符合稳定排序的要求？

关键字为 1 的元素中，排序前 `B` 在 `D` 前，排序后仍是 `B` 在 `D` 前。关键字为 2 的元素中，排序前 `A` 在 `C` 前，排序后仍是 `A` 在 `C` 前。

所以这个结果符合稳定排序要求。

若排序结果为：

```
(1,D) (1,B) (2,A) (2,C)
```

则关键字为 1 的两个元素相对顺序改变，不符合稳定排序要求。

例题 6：比较次数

选择排序处理 6 个元素，比较次数是多少？

选择排序每轮从剩余元素中选最小值：

```
第 1 轮：比较 5 次
第 2 轮：比较 4 次
第 3 轮：比较 3 次
第 4 轮：比较 2 次
第 5 轮：比较 1 次
```

总次数为 $5+4+3+2+1=15$。

若题目问冒泡排序的普通版本，在没有提前结束优化时，比较次数也是 15。若题目给出有提前结束标记的代码，则要结合输入判断，不能直接套固定次数。

例题 7：快速排序退化

下面快速排序每次选择第一个元素作为基准。若输入数组已经从小到大有序，它的复杂度是什么？

```
void qsort(int l, int r) {
    if (l >= r) return;
    int pivot = a[l];
    int p = l;
    for (int i = l + 1; i <= r; i++) {
        if (a[i] < pivot) {
            p++;
            swap(a[p], a[i]);
        }
    }
    swap(a[l], a[p]);
    qsort(l, p - 1);
    qsort(p + 1, r);
}
```

数组已经有序时，`pivot=a[l]` 是当前区间最小值。循环中没有元素小于它，最后 `p=l`。递归变成：

```
qsort(l+1, r)
```

也就是每次只减少一个元素。每层还要扫描当前区间一次，所以总时间为：

$$n+(n-1)+(n-2)+\cdots+1=O(n^2).$$

这类题不能只记快速排序平均 $O(n\log n)$，要看基准选择和输入形态。

程序阅读中的排序

排序程序阅读题可以按下面顺序处理。

第一，判断排序范围。是整个数组，还是某个区间；下标从 0 开始还是从 1 开始。

第二，判断排序依据。是按数值排序，还是按结构体中的某个字段排序；是升序还是降序。

第三，判断是否稳定。若题目涉及相等元素的顺序，要看算法或比较函数是否会改变它们。

第四，判断排序后提供了什么性质。后续程序可能依赖单调性、相邻关系、最大最小位置，或相同元素聚集。

第五，判断复杂度。排序在循环外还是循环内，排序规模是 n 还是某个子数组长度。

例题 8：排序范围

```
sort(a + 1, a + n);
```

这行代码排序的是 a[1] 到 a[n-1]，不包括 a[n]。

如果要排序 a[1] 到 a[n]，应写：

```
sort(a + 1, a + n + 1);
```

C++ 中 sort(first, last) 排序的是左闭右开区间 [first,last)。程序阅读题中，这种边界非常常见。

例题 9：比较函数

```
bool cmp(Node x, Node y) {
    return x.score > y.score;
}
sort(a + 1, a + n + 1, cmp);
```

这个比较函数表示分数高的排在前面，所以是按 score 降序排序。

若分数相等时 cmp(x,y) 和 cmp(y,x) 都返回 false，sort 不保证相等元素保持原顺序。需要稳定性时，应使用稳定排序，或者在比较函数中加入第二关键字。

第一轮中，不需要展开 C++ 标准库细节，但要知道：sort 通常不保证稳定；stable_sort 才强调稳定性。

完善程序中的排序

排序相关完善程序常考以下位置：

- 比较函数；
- 排序区间；
- 归并过程；
- 计数排序中的前缀和；
- 快速排序中的分区；
- 排序后双指针或二分的判断条件。

做这类题时，先看排序要提供什么条件。若后续要从小到大扫描，比较函数就不能写成降序；若后续要按结束时间贪心，排序关键字就应是结束时间；若后续要让相同元素连续出现，只要按对应关键字排序即可。

例题 10：补比较函数

有若干活动，每个活动有开始时间 l 和结束时间 r。程序后续要按结束时间从小到大选择活动，比较函数应如何写？

```
bool cmp(Node x, Node y) {
    return _____;
}
```

应填：

```
x.r < y.r
```

因为后续要让结束时间早的活动排在前面。

若填 `x.l < y.l`，排序依据变成开始时间；若填 `x.r > y.r`，则变成结束时间降序，都会破坏后续贪心流程。

常见错误

排序与比较次数题常见错误有以下几类。

错误	说明
混淆稳定性	只要相等关键字顺序可能改变，就不是稳定
忽略实现差异	冒泡中 `>` 和 `>=` 会影响稳定性
快速排序只记平均	特殊输入可能退化到 $O(n^2)$
排序范围看错	`sort(a+1,a+n)` 不包含 `a[n]`
忘记排序在循环内	总复杂度可能多乘一层
删除排序后仍按有序处理	双指针、二分可能失效
把比较排序下界用于非比较排序	计数排序、基数排序不只靠比较
相邻交换乱模拟	最少次数应看逆序数

做题时要把“排序后有什么性质”说清楚。排序不是单独存在的操作，它常为后续判断提供单调性、相邻性或分组效果。

小结

排序是很多算法的前置步骤，也是第一轮常见的独立考点。掌握稳定性、比较次数、复杂度和排序后的性质，比背很多排序模板更重要。第二编到这里结束，后面进入数学与计数部分，选择题中的手算比例会明显增加。

第三编 数学与计数

0x0A 进制与数位

进制题在 CSP-S 第一轮中经常以选择题和程序阅读题出现。它的题面通常不长，但很容易因为一个位权、一个余数顺序、一个十六进制字母值而失分。

进制的核心并不复杂：每个数字所在的位置有不同的权值。十进制中，从右往左依次是 $1, 10, 100, 1000, \ldots$；二进制中，从右往左依次是 $1, 2, 4, 8, \ldots$。小数点右侧也有位权，只不过位权变成了负次幂，例如十进制小数点右侧依次是 $10^{-1}, 10^{-2}, 10^{-3}, \ldots$。

本章重点讲整数进制、简单小数进制、数位拆分、二进制与十六进制的转换，以及程序阅读中常见的数位处理。小数进制转换不是 CSP-S 第一轮的重点，本章只讲最常见的判断方法，不展开成复杂专题。

位权

一个 b 进制数，每一位只能使用 0 到 $b-1$ 之间的数字。若 $b > 10$，还需要用字母表示更大的数字。例如十六进制中，A, B, C, D, E, F 分别表示 $10, 11, 12, 13, 14, 15$。

以十进制数 345 为例：

$$345 = 3 \times 10^2 + 4 \times 10^1 + 5 \times 10^0.$$

以二进制数 10110_2 为例：

$$10110_2 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0.$$

计算得到：

$$10110_2 = 16 + 4 + 2 = 22.$$

如果有小数点，小数点左侧的位权是非负次幂，小数点右侧的位权是负次幂。例如：

$$101.101_2 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}.$$

也就是：

$$101.101_2 = 4 + 1 + \frac{1}{2} + \frac{1}{8} = 5.625.$$

这里下标 2 表示这个数是二进制。书写进制题时，建议给非十进制数加上下标，避免把 10110_2 看成十进制的一万零一百一十。

常见进制如下。

进制	基数	可用数字	例子
二进制	2	$0, 1$	10110_2
八进制	8	0 到 7	735_8
十进制	10	0 到 9	345
十六进制	16	0 到 9，A 到 F	$2A7_{16}$

进制题的第一步，是确认题目中的数到底是哪一种进制。不要看到数字串就默认它是十进制。

转成十进制

任意进制转十进制，方法是按位权展开。

例题 1：二进制转十进制

求 110101_2 的十进制值。

从右往左写位权：

数位	1	1	0	1	0	1
位权	2^5	2^4	2^3	2^2	2^1	2^0

所以：

$$110101_2 = 1 \times 32 + 1 \times 16 + 0 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1 = 53.$$

答案是 53。

手算时可以不写乘以 0 的项：

$$110101_2 = 32 + 16 + 4 + 1 = 53.$$

例题 2：十六进制转十进制

求 $2A7_{16}$ 的十进制值。

十六进制中 $A = 10$，因此：

$$2A7_{16} = 2 \times 16^2 + 10 \times 16^1 + 7 \times 16^0.$$

计算：

$$2 \times 256 + 10 \times 16 + 7 = 512 + 160 + 7 = 679.$$

答案是 679。

十六进制题常见错误是把 A 当成 11，或把 F 当成 16。要记住：A 从 10 开始，F 是 15。

例题 3：二进制小数转十进制

求 101.101_2 的十进制值。

小数点左侧：

$$101_2 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 5.$$

小数点右侧：

$$.101_2 = 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} = \frac{1}{2} + \frac{1}{8} = 0.625.$$

所以：

$$101.101_2 = 5.625.$$

小数点右侧第一位的位权是 2^{-1}，不是 2^1。这是小数进制题最容易错的地方。

例题 4：十六进制小数转十进制

求 $A.F_{16}$ 的十进制值。

十六进制中 $A = 10, F = 15$，因此：

$$A.F_{16} = 10 \times 16^0 + 15 \times 16^{-1}.$$

计算：

$$10 + \frac{15}{16} = 10.9375.$$

所以：

$$A.F_{16} = 10.9375.$$

十六进制小数点右侧第一位表示“十六分之几”，不是“十分之几”。

十进制转其他进制

十进制整数转成 b 进制，使用“除 b 取余，余数倒序”的方法。

原因可以这样理解：一个数除以 b 的余数，就是它在 b 进制下的最低位；商再除以 b 的余数，就是倒数第二位。不断除下去，直到商为 0，再把余数倒过来写。

例题 5：十进制转二进制

把 53 转成二进制。

被除数	除以 2 的商	余数
53	26	1
26	13	0
13	6	1
6	3	0
3	1	1
1	0	1

余数从下往上读，得到：

$$53 = 110101_2.$$

这里余数不能顺着写成 101011_2。除基取余得到的是从低位到高位的顺序，最后必须倒序。

例题 6：十进制转十六进制

把 679 转成十六进制。

被除数	除以 16 的商	余数
679	42	7
42	2	10
2	0	2

余数 10 在十六进制中写作 A。从下往上读：

$$679 = 2A7_{16}.$$

十六进制转换时，余数可能是 10 到 15，要及时换成 A 到 F。

十进制小数转其他进制

十进制小数转成 b 进制小数，常用 “乘 b 取整” 的方法。

具体做法是：

- 把小数部分乘以 b；
- 乘积的整数部分就是当前位；
- 保留新的小数部分继续乘；
- 若小数部分变成 0，转换结束；
- 若一直不为 0，可能得到循环小数，只按题目要求保留若干位。

例题 7：十进制小数转二进制

把 0.625 转成二进制小数。

当前小数	乘以 2	取出的整数位	剩余小数
0.625	1.25	1	0.25
0.25	0.5	0	0.5
0.5	1.0	1	0

所以：

$$0.625 = 0.101_2.$$

注意，小数部分的取位顺序不需要倒过来。第一次乘出来的整数位就是小数点右侧第一位。

例题 8：带整数部分的小数转换

把 10.625 转成二进制。

整数部分 10 转成二进制：

$$10 = 1010_2.$$

小数部分 0.625 转成二进制：

$$0.625 = 0.101_2.$$

合起来：

$$10.625 = 1010.101_2.$$

处理带小数的数时，整数部分和小数部分分开转换。整数部分用“除基取余，倒序”；小数部分用“乘基取整，顺序”。

例题 9：不一定能有限表示

把 0.1 转成二进制时，小数部分不会很快变成 0。它会得到循环二进制小数。

这说明：一个十进制有限小数，在二进制中不一定也是有限小数。程序中用浮点数保存这类数时，也可能出现精度误差。

CSP-S 第一轮通常不会要求复杂的小数进制长计算。遇到小数进制题，重点掌握三点：

- 小数点右侧位权是负次幂；
- 转十进制按位权展开；
- 十进制小数转其他进制用乘基取整。

二进制、八进制、十六进制互转

二进制、八进制、十六进制之间有很方便的分组方法。

因为：

$$8 = 2^3, \quad 16 = 2^4.$$

所以每 3 个二进制位对应 1 个八进制位，每 4 个二进制位对应 1 个十六进制位。

例题 10：二进制转十六进制

把 110101101_2 转成十六进制。

从右往左每 4 位一组，不足 4 位在左边补 0：

```
0001 1010 1101
```

分别转成十六进制：

```
0001 = 1
1010 = A
1101 = D
```

所以：

$$110101101_2 = 1AD_{16}.$$

不要从左往右随便分组。二进制的最低位必须和十六进制的最低位对齐，所以要从右往左分组。

例题 11：十六进制转二进制

把 $3F2_{16}$ 转成二进制。

每一位十六进制数展开成 4 位二进制：

```
3 = 0011
F = 1111
2 = 0010
```

所以：

$$3F2_{16} = 001111110010_2.$$

若写成普通二进制数，最前面的多余 0 可以去掉：

$$3F2_{16} = 1111110010_2.$$

但在表示固定长度编码时，前导 0 可能有意义。题目若强调“补足 8 位”或“补足 16 位”，就不能随意删掉前导 0。

例题 12：二进制转八进制

把 10111011_2 转成八进制。

从右往左每 3 位一组：

```
010 111 011
```

转成八进制：

```
010 = 2
111 = 7
011 = 3
```

所以：

$$10111011_2 = 273_8.$$

八进制分组是 3 位，十六进制分组是 4 位。不要混用。

小数部分的分组

若二进制数带小数点，整数部分从右往左分组，小数部分从左往右分组。

例如把 101.1011_2 转成十六进制。

整数部分 101 从右往左补 0：

```
0101 = 5
```

小数部分 1011 从左往右正好一组：

```
1011 = B
```

所以：

$$101.1011_2 = 5.B_{16}.$$

若小数部分不足一组，在右侧补 0。例如：

$$0.11_2 = 0.1100_2 = 0.C_{16}.$$

整数部分补 0 在左边，小数部分补 0 在右边，这一点不要弄反。

进制中的加法

进制加法的原则和十进制一样：逢基数进一。

例题 13：二进制加法

计算：

$$101101_2 + 1110_2.$$

先对齐右端：

```
  101101
+ 001110
```

从低位加起：

```
  101101
+ 001110
= 111011
```

所以：

$$101101_2 + 1110_2 = 111011_2.$$

也可以转成十进制检查：

$$101101_2 = 45, \quad 1110_2 = 14, \quad 45 + 14 = 59.$$

而：

$$111011_2 = 32 + 16 + 8 + 2 + 1 = 59.$$

两边一致。

例题 14：十六进制加法

计算：

$$A7_{16} + 3D_{16}.$$

低位：$7 + D = 7 + 13 = 20$，$20 = 1 \times 16 + 4$，写 4，进 1。

高位：$A + 3 + 1 = 10 + 3 + 1 = 14$，写 E。

所以：

$$A7_{16} + 3D_{16} = E4_{16}.$$

进制加法题容易错在字母值。建议先把 A 到 F 对应的十进制值写在草稿纸边上：

```
A=10, B=11, C=12, D=13, E=14, F=15
```

数位拆分

数位题经常出现在程序阅读中。常见操作有取个位、去掉个位、取某一位、求数位和。

对十进制整数 x：

```
x % 10
```

得到个位；

```
x / 10
```

去掉个位。

例如 x=3457：

```
3457 % 10 = 7
3457 / 10 = 345
```

若反复执行，就可以从低位到高位取出所有十进制数位。

```
while (x > 0) {
    int d = x % 10;
    x /= 10;
}
```

这里得到的数位顺序是从低位到高位。若要按原顺序输出，需要先存下来再倒着输出，或使用字符串处理。

例题 15：求数位和

```
int sum_digit(int x) {
    int s = 0;
    while (x > 0) {
        s += x % 10;
        x /= 10;
    }
```

```
    return s;
}
```

输入 $x = 2025$。

过程为：

当前 x	x % 10	新的 x	s
2025	5	202	5
202	2	20	7
20	0	2	7
2	2	0	9

返回 9。

注意中间的数位 0 也要处理。虽然它不改变和，但循环不能跳过。

变式：输入为 0

上面的程序若输入 $x = 0$，循环一次也不执行，返回 0。这对“数位和”来说是合理的，因为 0 的数位和为 0。

但如果程序要统计数位个数，输入 0 时要单独处理：

```
int count_digit(int x) {
    if (x == 0) return 1;
    int cnt = 0;
    while (x > 0) {
        cnt++;
        x /= 10;
    }
    return cnt;
}
```

数位题中，0 经常是边界。

任意进制下的数位

十进制下用 `% 10` 和 `/ 10`，在 b 进制下就用 `% b` 和 `/ b`。

```
while (x > 0) {
    int d = x % b;
    x /= b;
}
```

得到的是 x 在 b 进制下从低位到高位的各个数位。

例题 16：二进制中 1 的个数

下面程序统计正整数 x 的二进制表示中有多少个 1。

```
int count_one(int x) {
    int cnt = 0;
    while (x > 0) {
        if (x % 2 == 1) cnt++;
        x /= 2;
    }
    return cnt;
```

```
}
```

输入 $x = 13$。

13 的二进制是 1101_2，其中有 3 个 1。

按程序追踪：

当前 x	x % 2	新的 x	cnt
13	1	6	1
6	0	3	1
3	1	1	2
1	1	0	3

返回 3。

这段程序没有直接写二进制字符串，但它等价于逐位检查二进制数位。

数位与字符串

进制题有时用字符串表示数字，特别是十六进制。程序需要把字符转成对应数值。

常见写法如下：

```
int val(char c) {
    if ('0' <= c && c <= '9') return c - '0';
    return c - 'A' + 10;
}
```

若 c 是 '7'，返回 7；若 c 是 'A'，返回 10；若 c 是 'F'，返回 15。

把一个十六进制字符串转成十进制，可以这样写：

```
int to_dec(string s) {
    int x = 0;
    for (int i = 0; i < s.size(); i++) {
        x = x * 16 + val(s[i]);
    }
    return x;
}
```

这段程序从左到右读入数位。每读一个新数位，相当于原来的数整体左移一位，即乘以 16，再加上当前数位。

例题 17：字符串转十进制

输入字符串 2A7。

程序过程：

已读部分	当前字符	当前值
空	2	$0 \times 16 + 2 = 2$
2	A	$2 \times 16 + 10 = 42$
2A	7	$42 \times 16 + 7 = 679$

所以返回 679。

这种写法不仅适用于十六进制，也适用于任意 b 进制：

```
x = x * b + 当前数位值;
```

C++ 中的整数写法

C++ 中，整数常量可以用不同形式表示。

- 普通写法如 `123` 是十进制；
- 以 `0` 开头的整数常量通常表示八进制；
- 以 `0x` 或 `0X` 开头表示十六进制；
- C++14 支持以 `0b` 或 `0B` 开头的二进制整数常量。

例如：

```cpp
int a = 10;
int b = 012;
int c = 0xA;
int d = 0b1010;
```

这里四个变量的值都是十进制下的 10。

程序阅读题中，`012` 很容易被误认为十进制的 12。但它是八进制，等于十进制的 10。看到前导 0 时要特别小心。

例题 18：整数常量

判断下面程序输出什么。

```cpp
#include <iostream>
using namespace std;

int main() {
    int a = 010;
    int b = 10;
    int c = 0x10;
    cout << a << ' ' << b << ' ' << c << endl;
    return 0;
}
```

`010` 是八进制，等于十进制 8。

`10` 是十进制，等于 10。

`0x10` 是十六进制，等于十进制 16。

因此输出：

```
8 10 16
```

这类题考的不是输出格式，而是整数常量的进制识别。

负进制的阅读

负进制不常作为选择题单独考查，但可能出现在程序阅读中。读负进制程序时，不需要先背很多性质，重点是看代码如何处理余数和商。

例如下面程序把整数 `n` 转成 $-k$ 进制，其中 $k > 0$：

```
while (n) {
    int d = (n % k + k) % k;
    a[++cnt] = d;
    n = (d - n) / k;
}
```

这里 d 被处理成 0 到 $k-1$ 之间的余数。更新式来自：

$$n = d + (-k) \times q.$$

移项得到：

$$q = \frac{d-n}{k}.$$

所以新一轮的 n 是 (d - n) / k。

读这类程序时，最稳的方式是用小输入追踪，不要凭普通正进制除法去想。

例题 19：负进制追踪

设 $k=2, n=6$。程序每轮为：

当前 n	d=(n%2+2)%2	新的 n=(d-n)/2
6	0	-3
-3	1	2
2	0	-1
-1	1	1
1	1	0

得到的低位到高位数位是：

```
0 1 0 1 1
```

倒序输出为：

```
11010
```

检查：

$$11010_{-2} = 1\times(-2)^4 + 1\times(-2)^3 + 0\times(-2)^2 + 1\times(-2)^1 + 0 = 16 - 8 - 2 = 6.$$

负进制题的关键是：程序记录的余数仍然要倒序输出；只是商的更新方式和普通正进制不同。

真题式例题

例题 20：十六进制数位和

定义函数 $f(n)$ 为 n 的十六进制表示中各位数字之和。例如：

$$2A7_{16}$$

的数位和为 $2+10+7=19$。

求 $f(2023)$。

先把 2023 转成十六进制。

被除数	除以 16 的商	余数
2023	126	7
126	7	14
7	0	7

余数 14 是 E，所以：

$$2023 = 7E7_{16}.$$

数位和为：

$$7+14+7=28.$$

所以 $f(2023)=28$。

这类题可以不写出完整十六进制字符串，也可以在除法过程中直接累加余数。但若题目还要继续对 $f(n)$ 操作，把表示写出来更不容易错。

变式 1：反复取数位和

定义 $f(n)$ 为十六进制数位和。若从 $n=2023$ 开始不断执行 $n=f(n)$，直到不再变化，最后得到多少？

上面已经算出：

$$2023 = 7E7_{16}, \quad f(2023)=28.$$

继续算 28 的十六进制表示：

$$28 = 1C_{16}.$$

所以：

$$f(28)=1+12=13.$$

继续算：

$$13 = D_{16}, \quad f(13)=13.$$

因此最后停在 13。

这类题中，“不再变化” 的意思是 $f(n)=n$。对于十六进制的一位数 0 到 15，它的数位和就是自己，所以都会不再变化。

程序阅读中的进制题

进制程序阅读题可以按下面顺序处理。

第一，确认基数。看代码里是 `% 10`、`% 2`、`% 16`，还是变量 `% k`。

第二，确认取位顺序。`x % b` 取的是最低位，循环得到的顺序通常是从低位到高位。

第三，确认是否倒序输出。进制转换程序常把余数存入数组，再从后往前输出。

第四，确认字符和数值转换。`'0'` 到 `'9'`，`'A'` 到 `'F'` 的处理要分开看。

第五，确认边界。输入 0、前导 0、十六进制字母、负数处理都可能影响答案。

例题 21：输出顺序

```
int n;
cin >> n;
while (n > 0) {
    cout << n % 2;
    n /= 2;
}
```

输入 13 时，程序输出什么？

13 的二进制是 1101_2。但程序每次输出最低位，没有倒序处理。

过程为：

当前 n	输出
13	1
6	0
3	1
1	1

所以程序输出：

```
1011
```

它不是正常顺序的二进制表示，而是二进制数位从低到高的顺序。

例题 22：倒序输出

```
int n, a[40], cnt = 0;
cin >> n;
while (n > 0) {
    a[++cnt] = n % 2;
    n /= 2;
}
for (int i = cnt; i >= 1; i--) {
    cout << a[i];
}
```

输入 13 时，数组依次存入：

```
a[1]=1, a[2]=0, a[3]=1, a[4]=1
```

倒序输出为：

```
1101
```

这才是 13 的正常二进制表示。

完善程序中的进制题

完善程序题中，进制相关空缺常出现在以下位置：

- 取当前最低位；
- 更新商；
- 倒序输出；
- 字符转数值；
- 数值转字符；
- 判断循环是否结束；
- 处理输入为 0 的边界；
- 小数转换中取整数部分和保留小数部分。

例题 23：补全十进制转二进制

```
int n, a[40], cnt = 0;
cin >> n;
if (n == 0) {
    cout << 0;
} else {
    while (n > 0) {
        a[++cnt] = _____;
        n = _____;
    }
    for (int i = cnt; i >= 1; i--) cout << a[i];
}
```

空缺应填：

```
n % 2
n / 2
```

第一空取当前最低二进制位，第二空去掉当前最低位。最后从 `cnt` 到 1 输出，是因为余数存入数组的顺序是从低位到高位。

例题 24：补全十六进制输出

```
char trans(int x) {
    if (x < 10) return _____;
    return _____;
}
```

把 0 到 15 的数转换成十六进制字符。

若 $x < 10$，应返回数字字符：

```
char('0' + x)
```

若 $x \geq 10$，10 对应 `A`，11 对应 `B`，所以应返回：

```
char('A' + x - 10)
```

完整写法：

```
char trans(int x) {
    if (x < 10) return char('0' + x);
    return char('A' + x - 10);
}
```

字符题中，不要把数值 10 直接输出成字符。数值和字符之间需要转换。

例题 25：补全十进制小数转二进制的一步

下面代码处理 $0 \le x < 1$ 的小数部分，输出前 m 位二进制小数。

```
for (int i = 1; i <= m; i++) {
    x = x * 2;
    int d = _____;
    cout << d;
    x = _____;
}
```

每次乘以 2 后，整数部分就是当前二进制小数位，小数部分留给下一轮。因此空缺应填：

```
int(x)
x - d
```

完整片段为：

```
for (int i = 1; i <= m; i++) {
    x = x * 2;
    int d = int(x);
    cout << d;
    x = x - d;
}
```

如果某一轮后 x 变成 0，后面继续输出的位都是 0。若题目只要求输出前 m 位，这样处理即可。

常见错误

进制与数位题的错误很集中。

错误	说明
忘记倒序	整数部分除基取余得到的是低位到高位
小数也倒序	小数部分乘基取整得到的是从左到右顺序
位权错位	最高位对应的幂次数数错
小数位权错	小数点右侧是负次幂
十六进制字母值错	$A = 10$，$F = 15$
前导 0 判断错	固定长度表示中前导 0 可能不能删
把 `010` 当十进制	C++ 中前导 0 的整数常量要小心
输入 0 没处理	`while (n > 0)` 不会执行
分组方向错	整数部分从右往左，小数部分从左往右
字符数值混淆	`'9'` 不是数值 9，需要减 `'0'`
负进制套普通除法	负进制程序要按代码追踪商和余数

做进制题时，草稿纸上最好写清楚三个东西：基数、位权、余数顺序。只要这三点不乱，大多数题都能稳住。

小结

进制题不需要复杂算法，但非常考验细节。位权、余数顺序、字符转换、前导 0、小数点两侧的处理方向，都是第一轮常见失分点。下一章进入排列与组合，计数题会更多，数位中的 “按位分类” 和 “逐位处理” 思想仍会继续用到。

0x0B 排列与组合

排列与组合是 CSP-S 第一轮中最常见的计数内容之一。它们经常出现在选择题中，题面通常很短：选人、排座位、发奖牌、组成字符串、构造号码、从集合中取元素。真正容易错的地方不是公式本身，而是题目中的“是否区分顺序”“是否允许重复”“是否必须选满”“对象是否相同”。

做计数题时，不要急着套 A_n^m 或 C_n^m。先问两件事：选出来的对象有没有顺序？同一个对象能不能重复选？这两个问题判断清楚，很多题就不会错。

本章先讲排列、组合和乘法原理。容斥会在下一章系统讲，本章只在少数例题中使用简单补集思想。

先看顺序

排列和组合最大的区别，在于是否区分顺序。

如果选出的人或物有位置、名次、编号、先后顺序，那么通常是排列。

如果只关心选了哪些人或物，不关心内部顺序，那么通常是组合。

例如，从 10 名选手中选出金、银、铜牌获得者。金牌、银牌、铜牌不同，选手 A, B, C 分别获得金银铜，与 B, A, C 分别获得金银铜，是两种不同结果。因此这是排列。

从 8 人中选出 2 人组成一个小组。若不区分组内角色，选出 A, B 和选出 B, A 是同一种小组。因此这是组合。

可以先用下面这张表判断：

题目说法	是否区分顺序	常用方法
排成一列	区分	排列
前三名、金银铜	区分	排列
密码、车牌、字符串	按位置区分	乘法原理或排列
选出若干人组成小组	不区分	组合
从集合中取若干元素	通常不区分	组合
分配到不同岗位	区分岗位	排列或乘法原理

有些题不会直接写“顺序”两个字。看到“第几位”“第几个位置”“金银铜”“编号”“岗位”“字符串”，就要把它们看成有顺序或有位置。

乘法原理

如果一件事要分几步完成，第 1 步有 a_1 种方法，第 2 步有 a_2 种方法，……，且每一步都要做，那么总方法数为：

$$a_1 a_2 \cdots a_k.$$

这就是乘法原理。

例题 1：车牌号

某车牌后 5 位中，前 2 位必须是大写英文字母，后 3 位必须是数字。字母可以重复，数字也可以重复。共有多少种号码？

前两位每位有 26 种选择，后三位每位有 10 种选择。由于每一位都要填，所以用乘法原理：

$$26 \times 26 \times 10 \times 10 \times 10 = 676000.$$

答案是 676000。

这道题不能写成 $26 \times 25 \times 10 \times 9 \times 8$，因为题目允许重复。判断是否允许重复，是这类题的关键。

例题 2：三位密码

密码由 3 位数字组成，每位可以是 0 到 9，允许重复。共有多少种密码？

每一位都有 10 种选择，所以共有：

$$10^3 = 1000$$

种。

如果题目说“三位正整数”，就不能让第一位为 0。第一位有 9 种选择，后两位各有 10 种，总数为 $9 \times 10 \times 10 = 900$。

同样是三位数字，是否允许前导 0 会改变答案。

加法原理

如果一件事可以分成几类完成，且这些类别互不重叠，第 1 类有 a_1 种，第 2 类有 a_2 种，……，那么总方法数为：

$$a_1 + a_2 + \cdots + a_k.$$

这就是加法原理。

乘法原理对应“几步都要做”，加法原理对应“几类选一种”。

例题 3：分类选择

从编号 1 到 20 的整数中选一个数，要求它是 3 的倍数或 5 的倍数，且先不使用容斥，改用分类计数。

可以先列出满足条件的数：

```
3, 5, 6, 9, 10, 12, 15, 18, 20
```

共有 9 个。

如果按“3 的倍数”和“5 的倍数”直接相加，会把 15 重复计算。这是下一章容斥要处理的问题。本章先记住：使用加法原理时，分类之间最好互不重叠；若有重叠，就要额外处理。

例题 4：首位分类

用数字 $0, 1, 2, 3$ 组成没有重复数字的三位数，问共有多少个？

可以直接用乘法原理：百位不能为 0，有 3 种；十位从剩下 3 个数字中选；个位从剩下 2 个数字中选。总数为：

$$3 \times 3 \times 2 = 18.$$

也可以按百位分类。百位为 $1, 2, 3$ 三类，每类中十位、个位有 $3 \times 2 = 6$ 种，总数为：

$$6 + 6 + 6 = 18.$$

两种方法结果一致。分类法适合题目条件在不同类别下不一样的情况。

排列

从 n 个不同对象中，选出 m 个并按顺序排列，方法数记作 A_n^m，也常写作 $P(n, m)$。

计算公式为：

$$A_n^m = n(n-1)(n-2)\cdots(n-m+1).$$

也可以写成：

$$A_n^m = \frac{n!}{(n-m)!}.$$

其中：

$$n! = 1 \times 2 \times 3 \times \cdots \times n.$$

特别地，$0! = 1$。这是为了让很多公式在边界处保持一致，例如 $A_n^n = \frac{n!}{0!} = n!$。

例题 5：颁奖方式

有 10 名选手，前三名分别获得金、银、铜牌。没有并列，且每名选手最多获得一枚奖牌。共有多少种颁奖方式？

金牌有 10 种选择。选定金牌后，银牌有 9 种选择。再选铜牌，有 8 种选择。

总数为：

$$10 \times 9 \times 8 = 720.$$

也就是：

$$A_{10}^3 = 720.$$

这不是组合。因为金、银、铜是不同位置，顺序不同表示不同颁奖方式。

变式 1：只选前三名名单

如果题目改成“选出进入前三名的 3 名选手，不区分名次”，则只关心是哪 3 人，不关心金银铜顺序。此时应使用组合：

$$C_{10}^3 = 120.$$

同样选 3 人，是否区分名次决定使用排列还是组合。

变式 2：允许一人多奖

如果题目改成“每个奖项从 10 名选手中选一人，允许同一人获得多个奖项”，那么金、银、铜各有 10 种选择，总数为：

$$10^3 = 1000.$$

这时不再是 $10 \times 9 \times 8$，因为对象可以重复选择。

组合

从 n 个不同对象中选出 m 个，不考虑顺序，方法数记作 C_n^m，也常写作 $\binom{n}{m}$。

公式为：

$$C_n^m = \frac{n!}{m!(n-m)!}.$$

也可以理解为：先按顺序选出 m 个，有 A_n^m 种；但同一组选出的人内部可以有 $m!$ 种排列，这些排列都对应同一个组合，所以要除以 $m!$：

$$C_n^m = \frac{A_n^m}{m!}.$$

例题 6：两人组队

共有 8 人，要求选出 2 人组成团队，不区分两人在团队中的角色。共有多少种组队方案？

如果先按顺序选，第一个人有 8 种选择，第二个人有 7 种选择，共 56 种。但同一个团队会被算两次，例如 A, B 和 B, A 是同一个团队。

因此答案为：

$$\frac{8 \times 7}{2} = 28.$$

也就是：

$$C_8^2 = 28.$$

例题 7：选委员会

从 12 人中选出 4 人组成委员会，不区分职位。共有多少种选法？

这是组合：

$$C_{12}^4 = \frac{12 \times 11 \times 10 \times 9}{4 \times 3 \times 2 \times 1} = 495.$$

计算组合数时，常把分子分母同步约分，避免中间数太大。

判断排列还是组合

很多错题都来自这一步。可以用下面的问题检查：

- 交换两个被选对象的位置，结果是否改变？
- 题目中是否有名次、位置、岗位、编号？
- 是否只问“选出哪些对象”？
- 选出的对象内部是否有角色差异？

例题 8：座位问题

从 6 人中选 3 人坐到一排 3 个座位上，有多少种坐法？

先选第 1 个座位的人，有 6 种；第 2 个座位有 5 种；第 3 个座位有 4 种。

总数为：

$$6 \times 5 \times 4 = 120.$$

这是排列，因为座位不同。A, B, C 坐在第 $1, 2, 3$ 个座位，与 B, A, C 坐在第 $1, 2, 3$ 个座位，是不同坐法。

例题 9：选代表

从 6 人中选 3 人作为代表，不区分职位，有多少种选法？

这是组合：

$$C_6^3 = \frac{6 \times 5 \times 4}{3 \times 2 \times 1} = 20.$$

同样是从 6 人中取 3 人，座位问题是排列，代表问题是组合。

允许重复的选择

有些题允许重复选择。例如每一位密码可以相同，每次投票可以投给同一个候选人，字符串中字符可以重复。

若每个位置都有相同的选择数，通常用乘法原理。

例题 10：字符串数量

由小写字母组成长度为 5 的字符串，允许字符重复，共有多少个？

每个位置都有 26 种选择，共有：

$$26^5$$

个。

这里不能用 A_{26}^5，因为字符可以重复。

例题 11：无重复字符串

若改成长度为 5，且字符互不相同，则第 1 位有 26 种，第 2 位有 25 种，依次为：

$$26 \times 25 \times 24 \times 23 \times 22 = A_{26}^5.$$

“允许重复” 和 “不允许重复” 差别很大。题目中若写 “可以相同”“互不相同”“每个对象最多使用一次”，都要圈出来。

全排列

把 n 个不同对象全部排成一列，方法数为：

$$n!.$$

例题 12：排列数字

数字 $1, 2, 3, 4$ 排成一列，共有多少种不同排列？

共有：

$$4! = 24$$

种。

若要求第一个位置不能是 1，可以先选第一个位置：有 3 种选择；剩下 3 个数字任意排列，有 3! 种。总数为：

$$3 \times 3! = 18.$$

也可以用补集：所有排列 4! 种，第一个位置是 1 的排列有 3! 种，所以答案是：

$$4! - 3! = 24 - 6 = 18.$$

补集思想在“不能”“至少一个不满足”等题中很常用。下一章讲容斥时会继续使用。

圆排列的初步认识

有些题把对象围成一圈。围成一圈时，整体旋转后如果看作同一种情况，就不能直接用 $n!$。

n 个不同对象围成一个圆圈，若只把旋转视为相同，而不把翻转视为相同，则方法数为：

$$(n-1)!.$$

可以理解为：先固定一个对象的位置，剩下 $n-1$ 个对象排成一列。

例题 13：围桌入座

5 个人围一张圆桌入座，若只关心相对顺序，旋转后视为同一种坐法，共有多少种？

固定其中一个人位置，剩下 4 个人排列：

$$4! = 24.$$

如果题目还把顺时针和逆时针看成同一种，则需要再除以 2。但这种情况题目会特别说明，不能自己默认。

圆排列在 CSP-S 第一轮中不算最高频，但在图论和环相关计数中可能出现。本章只需要先知道：圆上的“整体旋转”通常会造成重复计数。

二项式系数

组合数 C_n^m 也叫二项式系数，因为它出现在二项式展开中：

$$(a+b)^n = \sum_{k=0}^{n} C_n^k a^{n-k} b^k.$$

例如：

$$(a+b)^4 = a^4 + 4a^3b + 6a^2b^2 + 4ab^3 + b^4.$$

其中系数 $1, 4, 6, 4, 1$ 就是：

$$C_4^0, C_4^1, C_4^2, C_4^3, C_4^4.$$

第一轮中，二项式系数常用于选择题中的系数判断，也会和递推数组联系起来。

常用性质

组合数有几个常用性质。

第一，对称性：

$$C_n^m = C_n^{n-m}.$$

从 n 个对象中选 m 个，等价于决定哪 $n-m$ 个不选。

第二，递推式：

$$C_n^m = C_{n-1}^m + C_{n-1}^{m-1}.$$

它可以这样理解：考虑第 n 个对象。若不选它，需要从前 $n-1$ 个对象中选 m 个；若选它，需要从前 $n-1$ 个对象中再选 $m-1$ 个。

第三，边界：

$$C_n^0 = C_n^n = 1.$$

一个都不选有 1 种方式，全部选也有 1 种方式。

这些性质在程序阅读和完善程序中经常出现。

杨辉三角

组合数可以用递推式计算：

```
for (int i = 0; i <= n; i++) {
    c[i][0] = c[i][i] = 1;
    for (int j = 1; j < i; j++) {
        c[i][j] = c[i - 1][j] + c[i - 1][j - 1];
    }
```

```
}
```

这段程序计算的是杨辉三角。

以 $n=5$ 为例：

n	组合数
0	1
1	1 1
2	1 2 1
3	1 3 3 1
4	1 4 6 4 1
5	1 5 10 10 5 1

读这类程序时，要看清楚边界：

```
c[i][0] = c[i][i] = 1;
```

表示 $C_i^0 = C_i^i = 1$。

内部转移：

```
c[i][j] = c[i - 1][j] + c[i - 1][j - 1];
```

表示 $C_i^j = C_{i-1}^j + C_{i-1}^{j-1}$。

例题 14：程序含义

```
int c[10][10];

for (int i = 0; i <= 5; i++) {
    c[i][0] = c[i][i] = 1;
    for (int j = 1; j < i; j++) {
        c[i][j] = c[i - 1][j] + c[i - 1][j - 1];
    }
}
cout << c[5][2] << endl;
```

`c[5][2]` 表示 C_5^2，值为：

$$C_5^2 = \frac{5 \times 4}{2 \times 1} = 10.$$

所以输出 10。

这类程序阅读题不需要把整个数组都算完。认出组合数递推后，直接计算对应组合数即可。

分类计数

排列组合题中，题目条件复杂时，常需要分类。

分类的原则是：每种合法情况只属于一类，不重复，不遗漏。

例题 15：至少一个女生

有 5 名男生、4 名女生，从中选 3 人，要求至少有 1 名女生。共有多少种选法？

方法一：按女生人数分类。

女生人数可以是 $1, 2, 3$。

- 选 1 名女生、2 名男生：$C_4^1 C_5^2$；
- 选 2 名女生、1 名男生：$C_4^2 C_5^1$；
- 选 3 名女生、0 名男生：$C_4^3 C_5^0$。

总数为：

$$C_4^1 C_5^2 + C_4^2 C_5^1 + C_4^3 C_5^0 = 4 \times 10 + 6 \times 5 + 4 \times 1 = 74.$$

方法二：用补集。

所有选法为 C_9^3。不含女生，就是从 5 名男生中选 3 人，有 C_5^3 种。

所以答案为：

$$C_9^3 - C_5^3 = 84 - 10 = 74.$$

两种方法结果一致。遇到“至少一个”“至少一次”“不全是”这类说法，补集常常更简洁。下一章会把这种思想扩展到多个条件的容斥。

变式：恰好一个女生

若题目改成“恰好有 1 名女生”，就只能用第一类：

$$C_4^1 C_5^2 = 40.$$

“至少一个”和“恰好一个”不能混淆。

程序阅读中的排列组合

排列组合不只出现在选择题，也会藏在程序阅读中。

例题 16：双重循环计数

```
int cnt = 0;
for (int i = 1; i <= n; i++) {
    for (int j = i + 1; j <= n; j++) {
        cnt++;
    }
}
cout << cnt << endl;
```

这段程序枚举所有满足 $1 \le i < j \le n$ 的二元组。每一对不同元素只被枚举一次，所以：

$$cnt = C_n^2 = \frac{n(n-1)}{2}.$$

若内层改成：

```
for (int j = 1; j <= n; j++)
```

就枚举所有有序二元组 (i, j)，共有 n^2 个。

若内层改成：

```
for (int j = 1; j <= n; j++) {
    if (i == j) continue;
    cnt++;
}
```

就枚举所有 $i \neq j$ 的有序二元组，共有 $n(n-1)$ 个，也就是 A_n^2。

循环边界体现了“是否有顺序”“是否允许相等”。

例题 17：三重循环

```
int cnt = 0;
for (int i = 1; i <= n; i++) {
    for (int j = i + 1; j <= n; j++) {
        for (int k = j + 1; k <= n; k++) {
            cnt++;
        }
    }
}
```

这里枚举所有 $1 \le i < j < k \le n$ 的三元组。每组选出的 3 个数只按升序出现一次，所以：

$$cnt = C_n^3.$$

如果三个循环都从 1 到 n，并且要求 i, j, k 互不相同，那么枚举的是有序三元组，数量为：

$$n(n-1)(n-2) = A_n^3.$$

程序阅读中，不要只看循环层数，要看循环变量之间的大小关系。

完善程序中的组合数

完善程序题中，组合数常出现在两类位置：初始化和递推。

例题 18：补全杨辉三角

```
for (int i = 0; i <= n; i++) {
    c[i][0] = c[i][i] = 1;
    for (int j = 1; j < i; j++) {
        c[i][j] = _____;
    }
}
```

空缺应填：

```
c[i - 1][j] + c[i - 1][j - 1]
```

因为：

$$C_i^j = C_{i-1}^j + C_{i-1}^{j-1}.$$

这里不能填 `c[i - 1][j] + c[i][j - 1]`。那样右边使用了当前行的值，含义已经变成另一种递推。

例题 19：一维滚动计算

```
c[0] = 1;
for (int i = 1; i <= n; i++) {
    for (int j = i; j >= 1; j--) {
        c[j] = c[j] + c[j - 1];
    }
}
```

这段程序也在计算组合数。外层处理第 i 个对象，内层倒序更新，避免 `c[j-1]` 被当前轮提前覆盖。

处理完第 i 轮后，`c[j]` 表示 C_i^j。

如果内层从小到大更新：

```
for (int j = 1; j <= i; j++)
```

那么 `c[j - 1]` 可能已经变成当前轮的新值，结果会错。滚动数组题要特别注意循环方向。

常见错误

排列组合题的错误很集中。

错误	说明
把排列当组合	有名次、位置、岗位时仍除以阶乘
把组合当排列	只选人组队时仍区分顺序
忽略重复	题目允许重复，却用了 A_n^m
忽略不重复	题目要求互不相同，却用了幂
前导 0 处理错	数字和字符串的限制不同
分类重叠	加法原理中同一情况被算多次
漏分类	“至少”“至多”类问题少算一种情况
等价情况没除	圆排列、无角色小组等存在重复表示
循环含义看错	`i<j` 与 `i!=j` 对应不同计数

做题时，不要只写公式，要先写一句“是否区分顺序”。这句话能避免很多错误。

原创例题

例题 20：排队限制

6 名学生排成一列，其中甲必须站在乙前面。共有多少种排法？

不加限制时有 $6!$ 种排法。

对于任意一个排列，甲和乙的相对顺序只有两种：甲在乙前，或乙在甲前。两种情况数量相同。因此甲在乙前的排法数为：

$$\frac{6!}{2} = 360.$$

也可以先把甲乙位置选出来，再安排其他人，但对称思想更快。

变式 1：甲乙相邻

6 名学生排成一列，要求甲乙相邻。共有多少种排法？

把甲乙看成一个整体，与其他 4 人一起排，共 5! 种。甲乙内部可以是“甲乙”或“乙甲”，有 2 种。

总数为：

$$5! \times 2 = 240.$$

“相邻”常用捆绑法。

变式 2：甲乙不相邻

不相邻可以用补集。

所有排法 6! 种，相邻排法 240 种，所以不相邻排法为：

$$6! - 240 = 720 - 240 = 480.$$

遇到“不相邻”“至少一个不满足”等条件，补集常常比直接分类更稳。

小结

遇到排列组合题，可以按下面顺序检查。

步骤	检查问题
看对象	被选择或排列的对象是否互不相同
看顺序	交换两个对象的位置，结果是否改变
看重复	同一个对象能否被重复选择
看位置	是否有座位、名次、编号、字符串位置
看限制	是否有前导 0、相邻、不相邻、至少、至多
选公式	排列、组合、幂，还是乘法原理
查分类	分类是否互不重叠，是否覆盖全部情况
查补集	条件复杂时，反面是否更好数
看程序	循环边界对应有序还是无序计数
查边界	C_n^0、C_n^n、0! 是否处理正确

排列组合题看起来短，但最怕题意判断错。先判断顺序，再判断重复；先判断位置和角色，再决定是否除以阶乘。把这些问题想清楚，比记很多公式更有用。下一章进入容斥，重点会从“单一条件计数”转向“多个条件重叠时怎样不重不漏地计数”。

0x0C 容斥

容斥用来解决“重复计算”的问题。

在排列组合中，我们常用加法原理把情况分成几类再相加。但如果这些类别之间有重叠，直接相加就会把重叠部分算多。容斥的作用，就是把多算的部分减掉，再把被减多的部分加回来。

CSP-S 第一轮中的容斥题通常不会很长，但细节要求很高。题目可能问若干范围内满足条件的整数个数，也可能问字符串、集合、方案数中“至少一个条件成立”的数量，还可能给出一段程序，让考生判断它在统计什么。

本章只讲第一轮常用的容斥方法。更复杂的数论容斥、莫比乌斯反演等内容不属于本书主线。

两个集合

先看最基本的情况。

若有两个集合 A, B，要求 $A \cup B$ 中元素个数。如果直接算 $|A| + |B|$，那么同时属于 A 和 B 的元素会被算两次。因此要减去一次交集：

$$|A \cup B| = |A| + |B| - |A \cap B|.$$

这就是两个集合的容斥公式。

例题 1：倍数个数

求 1 到 100 中，是 3 的倍数或 5 的倍数的整数有多少个。

设：

- A 为 3 的倍数集合；
- B 为 5 的倍数集合。

1 到 100 中，3 的倍数有 $\lfloor 100/3 \rfloor = 33$ 个，5 的倍数有 $\lfloor 100/5 \rfloor = 20$ 个。

同时是 3 和 5 的倍数，就是 15 的倍数，有 $\lfloor 100/15 \rfloor = 6$ 个。

所以答案为：

$$33 + 20 - 6 = 47.$$

这里不能直接算 $33 + 20 = 53$，因为 $15, 30, 45, 60, 75, 90$ 被算了两次。

变式 1：既不是 3 的倍数，也不是 5 的倍数

求 1 到 100 中，既不是 3 的倍数，也不是 5 的倍数的整数有多少个。

上一题已经算出“是 3 的倍数或 5 的倍数”的有 47 个。总共有 100 个数，所以既不是的个数为：

$$100 - 47 = 53.$$

遇到“都不满足”“没有一个满足”这类问法，补集常常更快。

变式 2：是 3 的倍数但不是 5 的倍数

求 1 到 100 中，是 3 的倍数但不是 5 的倍数的整数有多少个。

先数 3 的倍数，共 33 个。再去掉同时是 3 和 5 的倍数，也就是 15 的倍数，共 6 个。

答案为：

$$33-6=27.$$

这不是完整的容斥公式，但本质仍是从一个集合中去掉重叠部分。

三个集合

若有三个集合 A,B,C，要求至少属于其中一个集合的元素个数，公式为：

$$|A\cup B\cup C|=|A|+|B|+|C|-|A\cap B|-|A\cap C|-|B\cap C|+|A\cap B\cap C|.$$

为什么最后要加回三者交集？

同时属于 A,B,C 的元素，在 $|A|+|B|+|C|$ 中被算了 3 次；在减去三个两两交集时，又被减了 3 次。这样它变成被算 0 次。它本来应该被算 1 次，所以要再加回来一次。

可以记成：

```
单个集合：加
两个集合交集：减
三个集合交集：加
```

例题 2：2、3、5 的倍数

求 1 到 100 中，至少是 $2,3,5$ 中一个数的倍数的整数有多少个。

设：

- A：2 的倍数；
- B：3 的倍数；
- C：5 的倍数。

单个集合：

$$|A|=50,\quad |B|=33,\quad |C|=20.$$

两两交集：

$$|A\cap B|=\left\lfloor\frac{100}{6}\right\rfloor=16,$$

$$|A\cap C|=\left\lfloor\frac{100}{10}\right\rfloor=10,$$

$$|B \cap C| = \left\lfloor \frac{100}{15} \right\rfloor = 6.$$

三者交集：

$$|A \cap B \cap C| = \left\lfloor \frac{100}{30} \right\rfloor = 3.$$

所以答案为：

$$50 + 33 + 20 - 16 - 10 - 6 + 3 = 74.$$

这里 $2, 3, 5$ 两两互质，所以共同倍数用乘积表示。若几个数不互质，应使用最小公倍数。最小公倍数会在第 16 章系统讲，这里只要知道 “同时是若干数的倍数” 要看它们的共同周期。

至少与恰好

容斥题常见关键词有 “至少”“恰好”“至多”“都不”。这些词对应不同计数方式。

“至少一个满足” 通常可以用容斥。

“都不满足” 通常可以用补集：总数减去至少一个满足。

“恰好一个满足” 不能直接套 “至少一个满足”。它要排除满足多个条件的情况。

例题 3：恰好一个倍数条件

求 1 到 30 中，恰好满足 “是 2 的倍数” 和 “是 3 的倍数” 之一的整数有多少个。

先数 2 的倍数，有 15 个；3 的倍数，有 10 个；同时是两者倍数，即 6 的倍数，有 5 个。

恰好满足一个条件，包含：

- 是 2 的倍数但不是 3 的倍数；
- 是 3 的倍数但不是 2 的倍数。

可以写成：

$$(15 - 5) + (10 - 5) = 15.$$

也可以写成：

$$|A| + |B| - 2|A \cap B| = 15 + 10 - 2 \times 5 = 15.$$

注意，求 “至少一个” 时减一次交集；求 “恰好一个” 时要把交集从两个集合里都去掉，所以减两次。

例题 4：恰好两个条件

求 1 到 60 中，恰好是 $2, 3, 5$ 中两个数的倍数的整数有多少个。

设三个条件分别为：

- A：是 2 的倍数；
- B：是 3 的倍数；

- C：是 5 的倍数。

恰好满足 A, B，但不满足 C 的数，是 6 的倍数但不是 30 的倍数：

$$\left\lfloor \frac{60}{6} \right\rfloor - \left\lfloor \frac{60}{30} \right\rfloor = 10 - 2 = 8.$$

恰好满足 A, C，但不满足 B：

$$\left\lfloor \frac{60}{10} \right\rfloor - \left\lfloor \frac{60}{30} \right\rfloor = 6 - 2 = 4.$$

恰好满足 B, C，但不满足 A：

$$\left\lfloor \frac{60}{15} \right\rfloor - \left\lfloor \frac{60}{30} \right\rfloor = 4 - 2 = 2.$$

总数为：

$$8 + 4 + 2 = 14.$$

这类题不要只把两两交集相加。因为同时满足三个条件的数会出现在每个两两交集中，必须去掉。

补集

当直接计数困难时，可以数反面。

补集适合这些说法：

- 至少一个；
- 不全是；
- 没有相邻；
- 至少一次出现；
- 至少一个位置满足；
- 至少一个条件不满足。

例题 5：至少一个女生

有 5 名男生、4 名女生，从中选 3 人，要求至少有 1 名女生。共有多少种选法？

总选法为：

$$C_9^3 = 84.$$

不含女生，就是从 5 名男生中选 3 人：

$$C_5^3 = 10.$$

所以至少有 1 名女生的选法为：

$$84 - 10 = 74.$$

这题也可以按女生人数为 $1,2,3$ 分类，但补集更短。

例题 6：至少有一个 0

长度为 4 的十进制数字串，每一位可以是 0 到 9，允许前导 0。要求至少有一位是 0。共有多少个？

总数字串有：

$$10^4$$

个。

没有 0 的数字串，每一位只能是 1 到 9，有：

$$9^4$$

个。

所以答案为：

$$10^4 - 9^4 = 10000 - 6561 = 3439.$$

注意这里是数字串，允许前导 0。若题目说 “四位正整数”，第一位不能为 0，总数和补集都会改变。

变式：四位正整数至少有一个 0

四位正整数的第一位不能为 0。

总数为：

$$9 \times 10^3 = 9000.$$

没有 0 的四位正整数，每一位都不能为 0，第一位到第四位各有 9 种选择：

$$9^4 = 6561.$$

所以至少有一个 0 的四位正整数有：

$$9000 - 6561 = 2439.$$

同样是 “四位”，数字串和正整数的前导 0 规则不同。

多条件容斥

当条件更多时，可以使用一般容斥思想：

- 加上所有单个条件的数量；
- 减去所有两个条件同时满足的数量；
- 加上所有三个条件同时满足的数量；
- 继续交替加减。

若有 m 个条件，至少满足一个条件的数量可以写成：

$$\sum |A_i| - \sum |A_i \cap A_j| + \sum |A_i \cap A_j \cap A_k| - \cdots.$$

第一轮中，通常不会要求手算很多个集合的完整公式，但会给出程序，让考生判断程序含义。

例题 7：三个条件的程序

下面程序统计 1 到 n 中，至少被 $2,3,5$ 中一个数整除的数的个数。

```
int ans = 0;
ans += n / 2;
ans += n / 3;
ans += n / 5;
ans -= n / 6;
ans -= n / 10;
ans -= n / 15;
ans += n / 30;
```

前三行加单个集合，接下来三行减两两交集，最后加三者交集。

这段程序的含义是：

```
2 的倍数 或 3 的倍数 或 5 的倍数
```

不是“同时被 $2,3,5$ 整除”。若问同时被三者整除，只需要 `n / 30`。

例题 8：统计不满足任何条件

若要统计 1 到 n 中，不被 $2,3,5$ 中任何一个数整除的数，可以在上面结果基础上取补集：

```
int bad = 0;
bad += n / 2;
bad += n / 3;
bad += n / 5;
bad -= n / 6;
bad -= n / 10;
bad -= n / 15;
bad += n / 30;
int ans = n - bad;
```

这里 `bad` 是至少满足一个“被整除”条件的数量。`ans` 是总数减去 `bad`。

程序阅读题中，变量名未必叫 `bad`，要看最后是否有 `n - ans` 这样的操作。

容斥和二进制枚举

多个条件的容斥可以用二进制状态枚举子集。虽然完整的位运算技巧不是本章重点，但第一轮程序阅读中可能出现这类结构，需要能读懂基本含义。

假设有 m 个条件，用一个整数 `s` 表示选中了哪些条件。若第 i 个条件被选中，就把它放入交集。

常见代码结构如下：

```
long long ans = 0;

for (int s = 1; s < (1 << m); s++) {
```

```
    int cnt = 0;
    long long ways = calc(s);

    for (int i = 0; i < m; i++) {
        if (s & (1 << i)) cnt++;
    }

    if (cnt % 2 == 1) ans += ways;
    else ans -= ways;
}
```

这段程序的符号由 `cnt` 的奇偶决定：

- 选中条件数为奇数，加；
- 选中条件数为偶数，减。

这正是容斥的“单个加、两个减、三个加、四个减”的规则。

这里 `calc(s)` 表示同时满足 `s` 中所有条件的方案数。不同题目中，`calc(s)` 的具体含义不同。读这种程序时，重点不在位运算本身，而在“它正在枚举条件子集，并按子集大小交替加减”。

程序阅读中的容斥

容斥程序通常有几个特征：

- 先把若干条件的数量加起来；
- 再减去重叠部分；
- 或者用子集枚举，根据选中条件数量奇偶加减；
- 最后可能用总数减去“至少一个条件成立”的数量。

读这类程序时，可以按下面顺序：

1. 找出全集大小；
2. 找出每个条件代表什么；
3. 看程序统计的是至少一个条件，还是没有条件；
4. 看交集部分是否使用共同周期、共同限制或同时满足；
5. 看最后有没有取补集。

例题 9：判断程序输出

```
int n = 100;
int ans = 0;
for (int i = 1; i <= n; i++) {
    if (i % 2 == 0 || i % 3 == 0) ans++;
}
cout << ans << endl;
```

这段程序逐个枚举 1 到 100，统计能被 2 或 3 整除的数。

用容斥计算：

$$\left\lfloor \frac{100}{2} \right\rfloor + \left\lfloor \frac{100}{3} \right\rfloor - \left\lfloor \frac{100}{6} \right\rfloor = 50 + 33 - 16 = 67.$$

所以输出 67。

这里程序是枚举写法，容斥是手算方法。程序阅读题中，可以用容斥帮助快速算结果。

例题 10：从代码还原条件

```
int ans = n;
ans -= n / 2;
ans -= n / 3;
ans += n / 6;
```

这段代码统计什么？

先看：

$$n/2 + n/3 - n/6$$

表示 1 到 n 中，能被 2 或 3 整除的数。

程序用 n 减去这部分，所以统计的是：

```
既不能被 2 整除，也不能被 3 整除的数
```

也就是与 6 互质的部分中的一类简单情况。这里不需要展开互质概念，只要看懂补集即可。

完善程序中的容斥

完善程序题中，容斥常在这些位置留空：

- 两个集合的交集项；
- 三个集合最后加回的项；
- 取补集时的 总数 - 已统计数量；
- 子集枚举时根据奇偶决定加减；
- 共同倍数对应的除数。

例题 11：补交集项

统计 1 到 n 中，能被 a 或 b 整除的数。已知 a, b 互质。

```
long long ans = 0;
ans += n / a;
ans += n / b;
ans -= _____;
```

因为 a, b 互质，同时被 a, b 整除就是被 ab 整除，所以空缺应填：

```
n / (a * b)
```

若 a, b 不互质，就不能直接用 ab，而应使用最小公倍数。第 16 章会专门讲 `gcd` 和 `lcm`。

例题 12：补三集合容斥

统计 1 到 n 中，能被 $2, 3, 5$ 至少一个整除的数。

```
long long ans = 0;
ans += n / 2 + n / 3 + n / 5;
ans -= n / 6 + n / 10 + n / 15;
ans += _____;
```

三者共同倍数是 30 的倍数，所以空缺应填：

```
n / 30
```

最后一项必须加回来，而不是继续减。因为三者交集在前面被加了三次、减了三次，已经变成没有被计入，需要再加一次。

例题 13：补奇偶符号

```
if (cnt % 2 == 1) ans += ways;
else _____;
```

这是子集枚举容斥。若选中条件数为奇数，则加；若为偶数，则减。所以空缺应填：

```
ans -= ways
```

不要把 `cnt` 的奇偶和条件编号混淆。这里看的是当前子集中选中了多少个条件。

常见错误

容斥题的错误很集中。

错误	说明
直接相加	忘记交集被重复计算
多减交集	求恰好一个和至少一个时减法次数不同
忘记加回三者交集	三集合容斥最后一项应加
补集方向错	总数减去的是“至少一个满足”的数量
分类重叠	使用加法原理时类别没有互斥
把“至少”看成“恰好”	至少一个包含多个条件同时满足
把“都不”看成“都满足”	两者含义相反
共同倍数算错	不互质时不能直接相乘
子集枚举符号错	子集大小为奇数加，偶数减

做容斥题时，最好先写一句：现在统计的是“至少一个满足”“恰好一个满足”，还是“没有一个满足”。这句话能避免很多方向错误。

原创例题

例题 14：数字串中至少出现一个指定数字

长度为 5 的十进制数字串，允许前导 0，要求至少出现一个 7。共有多少个？

总数字串有 10^5 个。

没有 7 的数字串，每一位有 9 种选择，所以有 9^5 个。

答案为：

$$10^5 - 9^5 = 100000 - 59049 = 40951.$$

这题用补集最简洁。若直接按“第一个 7 出现在哪里”分类也可以，但要小心后面位置是否还能出现 7。

变式 1：至少出现一个 7 和至少出现一个 8

长度为 5 的十进制数字串，允许前导 0，要求至少出现一个 7，且至少出现一个 8。共有多少个？

用总数减去不满足条件的情况。

设：

- A：没有 7；
- B：没有 8。

总数为 10^5。

没有 7 的有 9^5 个；没有 8 的也有 9^5 个；既没有 7 也没有 8 的，每位只有 8 种选择，有 8^5 个。

不满足“至少一个 7 且至少一个 8”的情况是 $A \cup B$，数量为：

$$9^5 + 9^5 - 8^5.$$

所以答案为：

$$10^5 - 2 \cdot 9^5 + 8^5.$$

这道题中，条件是两个“至少”。直接正向数会比较乱，用补集加容斥更稳。

变式 2：恰好出现一种指定数字

长度为 5 的十进制数字串，允许前导 0，要求在数字 7 和 8 中，恰好有一种出现过。共有多少个？

分两类：

第一类：出现过 7，但没有出现 8。

没有 8 时，每位可选 9 种；既没有 7 也没有 8 时，每位可选 8 种。所以出现过 7 且没有 8 的数量为：

$$9^5 - 8^5.$$

第二类：出现过 8，但没有出现 7。同理，也是：

$$9^5 - 8^5.$$

总数为：

$$2(9^5 - 8^5).$$

“恰好一种出现”和“两种都至少出现”不同，不能混用上一题答案。

小结

遇到容斥题，可以按下面顺序检查。

步骤	检查问题
全集	总共有多少种对象或方案
条件	每个集合分别表示什么
目标	至少一个、恰好一个、都不满足，还是同时满足
交集	哪些情况被重复计算
符号	单个加，两个减，三个加
补集	直接数困难时，反面是否更容易
分类	分类是否互斥，是否完整
程序	是否在枚举条件子集并按奇偶加减
边界	是否存在前导 0、重复对象、不互质等细节

容斥的难点不在公式本身，而在于看清“谁和谁重叠”。先定义集合，再判断目标，再决定加减顺序。只要每一步都和题意对上，容斥题通常可以在纸面上稳定完成。

0x0D 概率与期望

概率题和计数题关系很近。很多第一轮概率题，表面问“概率是多少”，实际第一步仍然是数方案：总共有多少种情况，其中有多少种满足条件。第 11 章和第 12 章讲的排列组合、容斥，在这一章会继续用到。

期望题看起来比概率题更抽象一些，但第一轮中常见的期望题并不复杂。重点是理解“平均意义下的结果”，以及会用期望的线性性，把一个总量拆成若干个小量分别计算。

本章不把概率与期望讲成完整数学课程，只保留 CSP-S 第一轮中常见、适合纸面考试的内容：等可能事件、补集、独立事件、条件概率的基本判断、期望定义、期望线性性，以及程序阅读中常见的概率模拟和计数程序。

概率先数总数

若所有基本情况等可能，事件 A 的概率为：

$$P(A)=\frac{\text{满足}A\text{的情况数}}{\text{所有情况数}}.$$

这里最重要的是“等可能”。如果每种情况出现的机会相同，就可以直接数个数；如果机会不同，就不能只数个数。

例题 1：掷骰子

掷一枚均匀的六面骰子，点数为偶数的概率是多少？

所有可能结果为：

```
1, 2, 3, 4, 5, 6
```

共 6 种，等可能。偶数结果为：

```
2, 4, 6
```

共 3 种。因此概率为：

$$\frac{3}{6}=\frac{1}{2}.$$

例题 2：两枚硬币

同时抛两枚均匀硬币，至少有一枚正面朝上的概率是多少？

所有结果可以写成：

```
正正，正反，反正，反反
```

共 4 种，等可能。至少有一枚正面的结果有 3 种：

```
正正，正反，反正
```

所以概率为：

$$\frac{3}{4}.$$

这里 “正反” 和 “反正” 是两种结果，因为两枚硬币可以区分为第一枚和第二枚。若题目只说 “正面个数”，那只是把结果合并后的描述，不能把 “一个正面一个反面” 只当作一种基本情况。

等可能要小心

概率题中最常见的错误，是把不等可能的情况当成等可能。

例题 3：两枚骰子的点数和

掷两枚均匀骰子，点数和为 7 的概率是多少？

不能只看点数和可能是 $2, 3, \dots, 12$，共 11 种，然后认为每种和的概率都是 $1/11$。这些点数和并不等可能。

正确的基本情况是有序点对 (a, b)，其中 a, b 分别表示第一枚和第二枚骰子的点数。共有 $6 \times 6 = 36$ 种等可能结果。

点数和为 7 的有：

```
(1,6), (2,5), (3,4), (4,3), (5,2), (6,1)
```

共 6 种。所以概率为：

$$\frac{6}{36} = \frac{1}{6}.$$

这道题的关键是选择正确的样本空间。点数和虽然有 11 种取值，但它们不是等可能基本结果。

补集

遇到 “至少一个”“不是全部失败”“出现过” 这类题，补集常常更简洁。

若事件 A 的反面事件记作 $\overline{A}$，则：

$$P(A) = 1 - P(\overline{A}).$$

例题 4：至少一次正面

连续抛 5 次均匀硬币，至少出现一次正面的概率是多少？

直接数 “至少一次正面” 要分出现 $1, 2, 3, 4, 5$ 次正面。用补集更简单。

反面事件是 “一次正面都没有”，也就是 5 次全是反面，概率为：

$$\left(\frac{1}{2}\right)^5 = \frac{1}{32}.$$

所以至少一次正面的概率为：

$$1 - \frac{1}{32} = \frac{31}{32}.$$

变式：至少一次成功

某操作每次成功概率为 p，失败概率为 $1-p$。独立进行 n 次，至少成功一次的概率为：

$$1-(1-p)^n.$$

这里用的是补集：“至少成功一次”的反面是“全部失败”。

这类公式不要死记。只要能把反面说清楚，就能重新推出来。

概率加法

若事件 A 和事件 B 互不重叠，也就是不能同时发生，则：

$$P(A\cup B)=P(A)+P(B).$$

若事件可能同时发生，就要减去重叠部分：

$$P(A\cup B)=P(A)+P(B)-P(A\cap B).$$

这和容斥完全对应。

例题 5：骰子点数

掷一枚均匀骰子，点数是 2 的倍数或 3 的倍数的概率是多少？

点数是 2 的倍数：

```
2, 4, 6
```

点数是 3 的倍数：

```
3, 6
```

直接相加会把 6 算两次。满足“是 2 的倍数或 3 的倍数”的点数为：

```
2, 3, 4, 6
```

共 4 种，所以概率为：

$$\frac{4}{6}=\frac{2}{3}.$$

用公式写：

$$P(A\cup B)=\frac{3}{6}+\frac{2}{6}-\frac{1}{6}=\frac{4}{6}.$$

概率里的容斥和计数里的容斥是同一件事，只是最后除以总情况数。

概率乘法

如果两个事件相互独立，事件 A 发生的概率不影响事件 B 发生的概率，则：

$$P(A \cap B) = P(A)P(B).$$

例如连续抛两次均匀硬币，每次是否为正面相互独立。两次都是正面的概率为：

$$\frac{1}{2} \times \frac{1}{2} = \frac{1}{4}.$$

例题 6：连续成功

某操作每次成功概率为 0.8，每次之间互不影响。连续做 3 次，全部成功的概率是多少？

每次成功概率都是 0.8，且相互独立，所以：

$$0.8^3 = 0.512.$$

例题 7：不独立的抽球

袋中有 3 个红球、2 个蓝球。不放回地连续抽两次，求两次都抽到红球的概率。

第一次抽到红球的概率为 3/5。在第一次已经抽到红球的条件下，袋中剩下 2 个红球、2 个蓝球，第二次抽到红球的概率为 2/4。

所以：

$$\frac{3}{5} \times \frac{2}{4} = \frac{3}{10}.$$

这里两次抽球不是独立事件。第二次的概率受到第一次结果影响。不能写成 $\frac{3}{5} \times \frac{3}{5}$。

条件概率

条件概率表示 “在某个条件已经发生的前提下，另一个事件发生的概率”。

记作：

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)}.$$

读作 “在 B 发生的条件下，A 发生的概率”。

第一轮中，不需要把条件概率展开得很深，但要会判断题意中的 “已知”“在……条件下”。

例题 8：已知点数为偶数

掷一枚均匀骰子，已知点数为偶数，求点数大于 3 的概率。

已知点数为偶数后，可能结果只剩：

```
2，4，6
```

在这 3 种等可能结果中，大于 3 的是：

```
4，6
```

所以概率为：

$$\frac{2}{3}.$$

不能用所有 6 个点数作为分母。条件已经把样本空间缩小了。

期望是什么

期望可以理解为“长期平均值”。如果一个随机变量 X 可能取值 $x_1, x_2, \dots, x_k$，对应概率为 $p_1, p_2, \dots, p_k$，那么：

$$E(X) = x_1p_1 + x_2p_2 + \cdots + x_kp_k.$$

例题 9：一枚骰子的期望点数

掷一枚均匀骰子，点数 X 的期望是多少？

每个点数 1 到 6 的概率都是 $1/6$，所以：

$$E(X) = 1 \times \frac{1}{6} + 2 \times \frac{1}{6} + \cdots + 6 \times \frac{1}{6}.$$

计算：

$$E(X) = \frac{1+2+3+4+5+6}{6} = \frac{21}{6} = 3.5.$$

期望不一定是某个可能出现的结果。骰子点数不可能是 3.5，但长期平均点数可以接近 3.5。

例题 10：抽奖收益

某抽奖活动有 $1/10$ 的概率获得 100 分，$9/10$ 的概率获得 0 分。设获得分数为 X，求 $E(X)$。

$$E(X) = 100 \times \frac{1}{10} + 0 \times \frac{9}{10} = 10.$$

期望为 10 分。它不是说每次都能拿到 10 分，而是多次重复后平均每次约为 10 分。

期望线性性

期望最常用的性质是线性性：

$$E(X + Y) = E(X) + E(Y).$$

更一般地：

$$E(X_1 + X_2 + \cdots + X_n) = E(X_1) + E(X_2) + \cdots + E(X_n).$$

这个性质不要求 X_i 相互独立。第一轮中，很多期望题都可以把总量拆成若干个指示变量来处理。

指示变量是一种只取 0 或 1 的随机变量。若事件发生，变量为 1；否则为 0。它的期望就是事件发生的概率。

例题 11：正面次数期望

连续抛 n 次均匀硬币，求正面出现次数的期望。

设第 i 次是否出现正面为 X_i。若第 i 次是正面，$X_i = 1$；否则 $X_i = 0$。

那么正面总次数为：

$$X = X_1 + X_2 + \cdots + X_n.$$

每次出现正面的概率为 $1/2$，所以：

$$E(X_i) = \frac{1}{2}.$$

由期望线性性：

$$E(X) = E(X_1) + E(X_2) + \cdots + E(X_n) = \frac{n}{2}.$$

例题 12：随机排列中的固定点

把 $1, 2, \ldots, n$ 随机打乱，求满足 “第 i 个位置上的数正好是 i” 的位置个数的期望。

设 X_i 表示第 i 个位置是否是固定点。若第 i 个位置上的数等于 i，则 $X_i = 1$，否则为 0。

对于任意位置 i，数 i 出现在这个位置的概率为 $1/n$，所以：

$$E(X_i) = \frac{1}{n}.$$

固定点总数为：

$$X = X_1 + X_2 + \cdots + X_n.$$

所以：

$$E(X) = n \times \frac{1}{n} = 1.$$

无论 n 是多少，随机排列固定点个数的期望都是 1。

这道题的关键不是枚举所有排列，而是把总固定点数拆成每个位置是否固定。

几何型等待

有一类常见期望：每次尝试成功概率为 p，各次相互独立，问第一次成功前期望尝试多少次。

答案是：

$$\frac{1}{p}.$$

第一轮中可以把它作为常用结论记住，但最好知道它的含义：成功概率越大，期望等待次数越少；若每次成功概率为 $1/2$，平均需要 2 次；若每次成功概率为 $1/6$，平均需要 6 次。

例题 13：等到第一次正面

反复抛均匀硬币，直到第一次出现正面为止，期望抛多少次？

每次出现正面的概率为 $1/2$，所以期望次数为：

$$\frac{1}{1/2} = 2.$$

例题 14：等到掷出 6

反复掷均匀骰子，直到第一次掷出 6 为止，期望掷多少次？

每次掷出 6 的概率为 $1/6$，所以期望次数为：

$$\frac{1}{1/6} = 6.$$

如果题目中每次成功概率会变化，就不能直接套这个结论。比如不放回抽球时，每次抽到某种球的概率会随剩余球数改变。

期望和概率的区别

概率问“某件事发生的可能性有多大”。期望问“某个数值平均是多少”。

例如抛 10 次硬币：

- “正面次数恰好为 5 的概率是多少”是概率问题；
- “正面次数的期望是多少”是期望问题。

前者需要计算某一种或某几种情况的概率，后者可以直接用线性性得到 $10 \times \frac{1}{2} = 5$。

期望题不一定要求知道每一种取值的完整分布。很多时候只要知道每个小变量的期望即可。

程序阅读中的概率

程序阅读题中，概率常以随机模拟、枚举计数、分数计算的形式出现。

例题 15：枚举概率

```
int cnt = 0;
for (int a = 1; a <= 6; a++) {
    for (int b = 1; b <= 6; b++) {
        if (a + b == 7) cnt++;
    }
}
cout << cnt << "/" << 36 << endl;
```

这段程序枚举两枚骰子的所有有序点对，共 36 种。满足点数和为 7 的有 6 种，所以输出：

```
6/36
```

若题目问化简后的概率，则为 $1/6$。但若程序直接输出分子和分母，就要按程序输出回答，不要自动化简成另一种格式。

例题 16：统计频率不是精确概率

```
int cnt = 0;
for (int i = 1; i <= n; i++) {
    int x = rand() % 6 + 1;
    if (x == 6) cnt++;
}
cout << cnt << endl;
```

这段程序模拟掷骰子 n 次，输出出现 6 的次数。它的期望是 $n/6$，但一次运行的输出不一定等于 $n/6$。

程序阅读题如果给出随机程序，通常不会要求预测某一次真实随机输出，而会问期望、可能范围或程序含义。要看题目问的是“输出一定是多少”，还是“期望是多少”。

程序阅读中的期望

期望程序常见结构是把每种结果的贡献乘以概率，再累加。

例题 17：数组平均值

```
double e = 0;
for (int i = 1; i <= n; i++) {
    e += a[i] * p[i];
}
cout << e << endl;
```

如果 `p[i]` 表示取到 `a[i]` 的概率，且所有 `p[i]` 之和为 1，那么程序计算的是随机变量的期望：

$$E = \sum_{i=1}^{n} a_i p_i.$$

读这类程序时，要确认 `p[i]` 是概率，还是次数、权重、比例。若 `p[i]` 是次数，还需要除以总次数才是概率。

例题 18：指示变量思想

```
double ans = 0;
for (int i = 1; i <= n; i++) {
    ans += p[i];
}
```

若 `p[i]` 表示第 i 个事件发生的概率，而程序要求事件发生总个数的期望，那么这段代码是合理的。因为每个事件对应一个指示变量，期望等于它发生的概率。

即使这些事件之间不独立，求总个数期望时仍然可以相加。这里用的是期望线性性，不是概率乘法。

完善程序中的概率与期望

完善程序题中，概率与期望常在这些位置留空：

- 总情况数；
- 有利情况数；
- 补集概率；
- 独立事件概率乘积；

- 期望累加式；
- 指示变量概率；
- 等待次数公式。

例题 19：补全期望

某随机变量取值为 `x[i]` 的概率为 `p[i]`。补全求期望的程序。

```
double ans = 0;
for (int i = 1; i <= n; i++) {
    ans += _____;
}
```

应填：

```
x[i] * p[i]
```

期望是“取值乘概率”的和，不是单纯把取值相加。

例题 20：补全集合概率

事件 A 发生概率为 `p`，独立重复 n 次。求至少发生一次的概率。

```
double ans = 1 - _____;
```

一次都不发生的概率是 $(1-p)^n$，所以空缺应填：

```
pow(1 - p, n)
```

完整含义是：

$$1-(1-p)^n.$$

若题目不允许使用浮点函数，可能会用循环乘法实现，但数学含义一样。

原创例题

例题 21：随机选两人

班上有 6 名男生、4 名女生，从 10 人中等可能选出 2 人。求选出的两人性别不同的概率。

总选法为：

$$C_{10}^{2}=45.$$

一男一女的选法为：

$$C_{6}^{1}C_{4}^{1}=24.$$

所以概率为：

$$\frac{24}{45}=\frac{8}{15}.$$

这里选出的两人不区分顺序，所以用组合。如果题目改成“先后随机抽出两人且记录顺序”，也可以用有序方法算，分子分母会同时扩大，结果仍为 $8/15$。

变式 1：两人都是女生

两人都是女生的选法为：

$$C_4^2 = 6.$$

概率为：

$$\frac{6}{45} = \frac{2}{15}.$$

变式 2：至少一名女生

可以用补集。没有女生就是两人都是男生，选法为 $C_6^2 = 15$。

所以至少一名女生的概率为：

$$1 - \frac{15}{45} = \frac{2}{3}.$$

这比按“一名女生、两名女生”分类更简洁。

例题 22：期望得分

某题答对得 5 分，答错得 0 分。某学生答对概率为 0.7。求该题得分期望。

设得分为 X，则：

$$E(X) = 5 \times 0.7 + 0 \times 0.3 = 3.5.$$

如果有 10 道互不相同的题，第 i 题分值为 s_i，答对概率为 p_i，总得分期望为：

$$\sum_{i=1}^{10} s_i p_i.$$

这里不需要知道每两道题之间是否独立。总分期望可以直接相加。

常见错误

概率与期望题的错误集中在下面几类。

错误	说明
样本空间选错	把不等可能结果当成等可能
顺序判断错	两枚骰子、两次抽取常常是有序结果
忘记补集	“至少一次”直接分类过多，容易漏
独立性误判	不放回抽取通常不是独立事件
条件概率分母错	已知条件会缩小样本空间
把概率相加乱用	事件重叠时需要减交集
把期望当最可能值	期望是平均值，不一定能实际取到
线性性不敢用	求总数期望时可拆成指示变量
把概率乘法当线性性	概率乘法需要独立或条件处理
程序输出误判	随机模拟输出不等于期望值

做概率题时，先问“哪些基本情况等可能”。做期望题时，先问“能否把总量拆成若干个小变量”。这两个问题能解决大部分第一轮题目。

小结

遇到概率与期望题，可以按下面顺序检查。

步骤	检查问题
样本空间	基本结果是否等可能
分子分母	有利情况数和总情况数分别是多少
顺序	结果是否区分先后或位置
补集	“至少一次”“至少一个”能否反面计数
交集	多个事件是否重叠，需要容斥吗
独立性	概率能否直接相乘
条件	是否已经给出某个条件，样本空间是否缩小
期望定义	是否按取值乘概率求和
线性性	总量能否拆成若干个指示变量
程序	程序输出的是概率、次数、频率，还是期望

概率与期望并不要求把所有情况背下来。概率题先数清基本情况，期望题先找可拆分的小变量。把计数、补集、容斥和线性性结合起来，就能处理第一轮中大多数概率与期望问题。

0x0E 递推数列

递推数列是用前面的若干项来定义后面的项。它和第 6 章中的递归、递推关系很近，但本章更偏向数学与计数：给出初值和递推式，要求计算某一项、判断增长规律、读懂程序中的数组含义，或在完善程序中补出转移语句。

CSP-S 第一轮中，递推数列经常以选择题和程序阅读题出现。题目可能给出一个公式，也可能给出一段循环程序，让考生判断输出。很多题并不难在公式，而难在初值、下标、循环顺序和取模。

本章只讲第一轮常见递推。动态规划中的状态设计会在后面专章讲，本章重点是“数列项之间怎样推出来”。

初值和递推式

一个递推数列通常由两部分确定：初值和递推式。

例如：

$$a_1 = 1, \quad a_n = a_{n-1} + 2 \quad (n \geq 2).$$

这里 $a_1 = 1$ 是初值，$a_n = a_{n-1} + 2$ 是递推式。由它可以依次算出：

$$\begin{aligned} a_1 &= 1, \\ a_2 &= a_1 + 2 = 3, \\ a_3 &= a_2 + 2 = 5, \\ a_4 &= a_3 + 2 = 7. \end{aligned}$$

若只给递推式，不给初值，数列通常不能唯一确定。例如 $a_n = a_{n-1} + 2$，若 $a_1 = 1$，得到 $1, 3, 5, 7, \ldots$；若 $a_1 = 10$，得到 $10, 12, 14, 16, \ldots$。

程序中也是一样。下面两段程序的递推式相同，但初值不同，输出会不同。

```
a[1] = 1;
for (int i = 2; i <= n; i++) {
    a[i] = a[i - 1] + 2;
}
```

```
a[1] = 10;
for (int i = 2; i <= n; i++) {
    a[i] = a[i - 1] + 2;
}
```

读递推程序时，第一步不是看循环，而是看初值。初值错了，后面每一项都会跟着错。

等差数列

如果相邻两项的差固定不变，这个数列就是等差数列。

例如：

```
3, 7, 11, 15, 19, ...
```

相邻两项都相差 4。若首项为 a_1，公差为 d，则第 n 项为：

$$a_n = a_1 + (n-1)d.$$

前 n 项和为：

$$S_n = \frac{n(a_1 + a_n)}{2}.$$

也可以写成：

$$S_n = \frac{n(2a_1 + (n-1)d)}{2}.$$

例题 1：求第 n 项

数列满足 $a_1 = 5$，$a_n = a_{n-1} + 3$。求 a_{10}。

这是等差数列，首项 $a_1 = 5$，公差 $d = 3$。所以：

$$a_{10} = 5 + (10-1) \times 3 = 32.$$

答案是 32。

也可以逐项递推，但选择题中若 n 较大，直接使用通项更快。

例题 2：求前 n 项和

数列 $2, 5, 8, 11, \ldots$ 的前 20 项和是多少？

首项 $a_1 = 2$，公差 $d = 3$。第 20 项为：

$$a_{20} = 2 + 19 \times 3 = 59.$$

前 20 项和为：

$$S_{20} = \frac{20(2 + 59)}{2} = 610.$$

等差数列求和题要注意项数。第 1 项到第 20 项一共有 20 项，不是 19 项。

等比数列

如果相邻两项的比值固定不变，这个数列就是等比数列。

例如：

```
3, 6, 12, 24, 48, ...
```

每一项都是前一项乘以 2。若首项为 a_1，公比为 q，则第 n 项为：

$$a_n = a_1 q^{n-1}.$$

若 $q \neq 1$，前 n 项和为：

$$S_n = a_1 \frac{q^n - 1}{q - 1}.$$

若 $q = 1$，每一项都等于 a_1，前 n 项和为 na_1。

例题 3：求第 n 项

数列满足 $a_1 = 2$，$a_n = 3a_{n-1}$。求 a_6。

这是等比数列，首项 2，公比 3。

$$a_6 = 2 \times 3^5 = 486.$$

答案是 486。

例题 4：增长速度比较

比较两个数列：

```
a_n = 100n
b_n = 2^n
```

当 n 很小时，a_n 可能比 b_n 大。例如 $n = 5$ 时，$a_5 = 500$，$b_5 = 32$。

但随着 n 增大，2^n 的增长会超过 $100n$。复杂度比较中，指数级增长会超过线性增长。

这一点和第 5 章的复杂度判断相通。递推数列不只用于算数值，也常用于判断增长级别。

斐波那契型递推

斐波那契数列是最常见的二阶递推：

$$F_1 = 1, \quad F_2 = 1, \quad F_n = F_{n-1} + F_{n-2} \quad (n \geq 3).$$

前几项为：

```
1, 1, 2, 3, 5, 8, 13, 21, ...
```

它的当前项依赖前两项。程序写法为：

```
f[1] = 1;
f[2] = 1;
for (int i = 3; i <= n; i++) {
    f[i] = f[i - 1] + f[i - 2];
}
```

这里有两个初值，因为递推式需要用到前两项。

例题 5：计算斐波那契数列

求 F_7。

按顺序计算：

i	F_i
1	1
2	1
3	2
4	3
5	5
6	8
7	13

所以 $F_7 = 13$。

变式：初值改变

若数列满足：

$$a_1 = 2, \quad a_2 = 3, \quad a_n = a_{n-1} + a_{n-2}.$$

求 a_5。

逐项计算：

i	a_i
1	2
2	3
3	5
4	8
5	13

结果仍是 13。但这只是前几项碰巧接近常见斐波那契结果，不能直接把它当作标准 F_n。初值一变，整个数列含义就变了。

递推的下标

递推题中，下标是高频失分点。最常见的错误有两种。

第一，把从 0 开始的数列看成从 1 开始。

第二，循环起点写错，导致使用了未定义的项。

例如下面这段程序：

```
f[0] = 0;
f[1] = 1;
for (int i = 2; i <= n; i++) {
    f[i] = f[i - 1] + f[i - 2];
}
```

这里定义的是：

$$F_0 = 0, \quad F_1 = 1.$$

此时 $F_2 = 1, F_3 = 2, F_4 = 3$。

而下面这段程序：

```
f[1] = 1;
f[2] = 1;
for (int i = 3; i <= n; i++) {
    f[i] = f[i - 1] + f[i - 2];
}
```

定义的是：

$$F_1 = 1, \quad F_2 = 1.$$

此时 $F_3 = 2, F_4 = 3$。

两种写法都常见，但输入为 0、1、2 时输出不同。程序阅读题中，一定要看清题目和代码从哪个下标开始。

例题 6：输出第几项

```
int f[10];
f[0] = 1;
f[1] = 1;
for (int i = 2; i <= 6; i++) {
    f[i] = f[i - 1] + f[i - 2];
}
cout << f[6] << endl;
```

逐项计算：

i	$f[i]$
0	1
1	1
2	2
3	3
4	5
5	8
6	13

输出 13。

这里 `f[6]` 是从 `f[0]` 开始数的第 7 项。若题目写“第 6 项”，要看它的定义是否从 0 开始。

递推和程序顺序

递推程序必须保证被依赖的项已经算好。

例如：

```
for (int i = 3; i <= n; i++) {
    f[i] = f[i - 1] + f[i - 2];
}
```

计算 `f[i]` 时，需要 `f[i-1]` 和 `f[i-2]`。从小到大循环可以保证这两个值已经算好。

若写成：

```
for (int i = n; i >= 3; i--) {
    f[i] = f[i - 1] + f[i - 2];
}
```

当计算 `f[n]` 时，`f[n-1]` 和 `f[n-2]` 可能还没有被正确计算。除非它们已经由别的方式给出，否则这个顺序不对。

例题 7：反向递推

有些递推必须从后往前算。例如：

```
f[n] = a[n];
for (int i = n - 1; i >= 1; i--) {
    f[i] = f[i + 1] + a[i];
}
```

这里 `f[i]` 表示从 i 到 n 的后缀和：

$$f[i] = a[i] + a[i+1] + \cdots + a[n].$$

计算 `f[i]` 时需要 `f[i+1]`，所以必须从大到小。

读递推程序时，不要先判断循环方向“看起来是否正常”，要看当前项依赖哪一项。依赖前面就从前往后，依赖后面就从后往前。

一阶线性递推

有些递推形如：

$$a_n = pa_{n-1} + q.$$

这类递推常在程序阅读题中出现。若只要求算小规模，直接逐项计算即可；若题目要求判断增长规律，可以把它看成“上一项乘一个数，再加一个固定数”。

例题 8：逐项计算

数列满足 $a_1 = 1$，$a_n = 2a_{n-1} + 1$。求 a_5。

逐项计算：

n	a_n
1	1
2	$2 \times 1 + 1 = 3$
3	$2 \times 3 + 1 = 7$
4	$2 \times 7 + 1 = 15$
5	$2 \times 15 + 1 = 31$

所以 $a_5 = 31$。

这个数列其实是 $a_n = 2^n - 1$。但第一轮中，若只问小项，逐项递推更简单。

例题 9：程序输出

```
int x = 1;
for (int i = 1; i <= 4; i++) {
    x = 2 * x + 1;
}
cout << x << endl;
```

初始 `x=1`。循环执行 4 次：

```
第 1 次：x = 3
第 2 次：x = 7
第 3 次：x = 15
第 4 次：x = 31
```

输出 31。

这里循环次数和数列下标要区分。`x` 初始值已经相当于第一项，循环 4 次后到达第五个值。

二阶线性递推

斐波那契数列是一种二阶线性递推。更一般地，可以写成：

$$a_n = pa_{n-1} + qa_{n-2}.$$

第一轮中，常见考法是给定初值和递推式，要求计算若干项，或判断程序对应的递推。

例题 10：二阶递推

数列满足：

$$a_1 = 1, \quad a_2 = 2, \quad a_n = 2a_{n-1} + a_{n-2}.$$

求 a_5。

逐项计算：

n	a_n
1	1
2	2
3	$2 \times 2 + 1 = 5$
4	$2 \times 5 + 2 = 12$
5	$2 \times 12 + 5 = 29$

所以 $a_5 = 29$。

这类题不需要急着找通项。CSP-S 第一轮中，大多数二阶递推手算题的规模不会太大，表格更可靠。

计数递推

递推数列经常来自计数问题。题目不一定直接给公式，而是给一个过程，要求考生自己写出递推。

例题 11：爬楼梯

每次可以上 1 级或 2 级台阶。设 $f[n]$ 表示走到第 n 级台阶的方法数。求递推式。

最后一步有两种可能：

- 从第 $n-1$ 级走 1 级；
- 从第 $n-2$ 级走 2 级。

所以：

$$f[n] = f[n-1] + f[n-2].$$

初值可以取：

$$f[0] = 1, \quad f[1] = 1.$$

这里 $f[0] = 1$ 表示“什么都不走”也是一种起点方式。于是：

```
f[2]=2
f[3]=3
f[4]=5
```

这和斐波那契数列非常接近，但下标定义不同。

例题 12：铺砖

用 1×2 的小砖铺满 $2 \times n$ 的长方形。设方案数为 $f[n]$。

看最左边：

- 若放一块竖砖，占掉 2×1，剩下 $2 \times (n-1)$，有 $f[n-1]$ 种；
- 若放两块横砖，占掉 2×2，剩下 $2 \times (n-2)$，有 $f[n-2]$ 种。

所以：

$$f[n] = f[n-1] + f[n-2].$$

初值为：

$$f[1] = 1, \quad f[2] = 2.$$

这说明不同问题可能得到相同形式的递推式，但初值和状态含义可能不同。

前缀和递推

前缀和也是一种递推。

```
s[0] = 0;
for (int i = 1; i <= n; i++) {
    s[i] = s[i - 1] + a[i];
}
```

这里：

$$s[i] = a[1] + a[2] + \cdots + a[i].$$

递推式为：

$$s[i] = s[i-1] + a[i].$$

用它可以快速求区间和：

$$a[l] + a[l+1] + \cdots + a[r] = s[r] - s[l-1].$$

例题 13：程序阅读

```
int a[6] = {0, 3, 1, 4, 1, 5};
int s[6];
s[0] = 0;
for (int i = 1; i <= 5; i++) {
    s[i] = s[i - 1] + a[i];
}
cout << s[4] - s[1] << endl;
```

先算前缀和：

i	$a[i]$	$s[i]$
0	—	0
1	3	3
2	1	4
3	4	8
4	1	9
5	5	14

`s[4] - s[1]` 表示 $a[2] + a[3] + a[4]$，值为 $1 + 4 + 1 = 6$。

所以输出 6。

前缀和题中，`s[r]-s[l-1]` 和 `s[r]-s[l]` 只差一个下标，但区间含义不同。

取模递推

数列增长很快时，程序中常对某个模数取模。

```
const int mod = 1000000007;

f[1] = 1;
f[2] = 1;
for (int i = 3; i <= n; i++) {
    f[i] = (f[i - 1] + f[i - 2]) % mod;
}
```

这段程序计算的是 F_i 对 `mod` 取模后的值。

取模递推常见性质是：加法、减法、乘法可以在每一步取模，不影响最终对模数的余数。例如：

$$(a+b) \bmod m = ((a \bmod m) + (b \bmod m)) \bmod m.$$

因此递推中每次取模是合理的。

例题 14：小模数递推

设 $mod = 10$，$f_1 = 1, f_2 = 1$，$f_i = (f_{i-1} + f_{i-2}) \bmod 10$。求 f_8。

逐项计算：

i	原和	f_i
1	—	1
2	—	1
3	$1+1$	2
4	$1+2$	3
5	$2+3$	5
6	$3+5$	8
7	$5+8=13$	3
8	$8+3=11$	1

所以 $f_8 = 1$。

注意第 7 项开始已经取模。如果继续用未取模的斐波那契数列，再最后取模，在数学结果上也一致；但程序阅读题要按程序中的每一步来算。

周期

有些取模递推会出现周期。第一轮中通常不会要求证明复杂周期，但可能要求观察小规模状态。

例如：

```
f[0] = 0;
f[1] = 1;
for (int i = 2; i <= n; i++) {
    f[i] = (f[i - 1] + f[i - 2]) % 2;
}
```

前几项为：

i	$f[i]$
0	0
1	1
2	1
3	0
4	1
5	1
6	0

从 $0, 1, 1$ 开始循环，周期为 3。

为什么会出现周期？因为当前项只由前两项决定，而在模 2 下，可能的相邻状态只有有限种：

```
(0,0), (0,1), (1,0), (1,1)
```

一旦某个相邻状态再次出现，后面的数列就会重复。

程序阅读题中，如果看到“小模数 + 递推 + 很大的 n”，可以尝试找周期。但本书不在这里展开复杂周期算法。

滚动变量

有些递推只依赖前一项或前几项，不需要保存整个数组。

例如斐波那契递推：

```
int a = 1, b = 1, c;
for (int i = 3; i <= n; i++) {
    c = a + b;
    a = b;
    b = c;
}
```

这里 a 和 b 分别保存前两项，c 是当前项。每一轮后，a、b 向后移动。

例题 15：判断输出

```
int a = 1, b = 2, c;
for (int i = 3; i <= 6; i++) {
    c = a + b;
    a = b;
    b = c;
}
cout << b << endl;
```

追踪：

轮次	i	c=a+b	更新后 a	更新后 b
初始	—	—	1	2
1	3	3	2	3
2	4	5	3	5
3	5	8	5	8
4	6	13	8	13

输出 13。

滚动变量题要按语句顺序看。若把更新顺序写错，结果会变。

变式：错误更新顺序

```
a = b;
b = a + b;
```

这两句执行后，第二句中的 a 已经不是旧的 a，而是旧的 b。如果本来想计算新项为“旧 a + 旧 b”，这种写法就错了。

安全写法是先用临时变量保存新值：

```
c = a + b;
a = b;
b = c;
```

从程序还原递推式

程序阅读中，常见问题是给一段循环，问它对应什么递推。

例题 16：还原递推

```
f[0] = 2;
for (int i = 1; i <= n; i++) {
    f[i] = 3 * f[i - 1] + 1;
}
```

对应递推为：

$$f_0 = 2, \quad f_i = 3f_{i-1} + 1 \quad (i \geq 1).$$

若要求 f_3，逐项计算：

```
f_0 = 2
f_1 = 3 * 2 + 1 = 7
f_2 = 3 * 7 + 1 = 22
f_3 = 3 * 22 + 1 = 67
```

答案为 67。

例题 17：二维循环中的递推

```
c[0][0] = 1;
for (int i = 1; i <= n; i++) {
    c[i][0] = c[i][i] = 1;
    for (int j = 1; j < i; j++) {
        c[i][j] = c[i - 1][j] + c[i - 1][j - 1];
    }
}
```

这正是第 11 章中讲过的组合数递推：

$$C_i^j = C_{i-1}^j + C_{i-1}^{j-1}.$$

读程序时，要先看边界 `c[i][0]=c[i][i]=1`，再看内部转移。边界和转移一起决定数组含义。

完善程序中的递推

递推数列在完善程序中常见空缺包括：

- 初值；
- 循环起点；
- 转移公式；

- 取模；
- 滚动变量更新顺序；
- 最终输出项。

例题 18：补初值

下面程序计算标准斐波那契数列 $F_1 = F_2 = 1$。

```
f[1] = _____;
f[2] = _____;
for (int i = 3; i <= n; i++) {
    f[i] = f[i - 1] + f[i - 2];
}
```

空缺应填：

```
1
1
```

如果题目定义从 $F_0 = 0, F_1 = 1$ 开始，代码就应相应改变。完善程序题必须以题面定义为准。

例题 19：补循环起点

```
f[1] = 1;
f[2] = 1;
for (int i = _____; i <= n; i++) {
    f[i] = f[i - 1] + f[i - 2];
}
```

因为 `f[1]` 和 `f[2]` 已经给出，下一项从 `f[3]` 开始算，所以空缺应填：

```
3
```

若从 `i=2` 开始，程序会重新计算 `f[2]`，并且用到 `f[0]`，含义不对。

例题 20：补滚动更新

```
int a = 1, b = 1, c;
for (int i = 3; i <= n; i++) {
    c = a + b;
    _____;
    _____;
}
cout << b << endl;
```

若 `a` 表示上一轮的较早项，`b` 表示上一轮的较晚项，那么计算出当前项 `c` 后，应当让窗口右移：

```
a = b;
b = c;
```

这两句顺序不能反。如果先写 `b = c`，再写 `a = b`，那么 `a` 也会变成 `c`，旧的 `b` 丢失。

常见错误

递推数列题的错误集中在下面几类。

错误	说明
忘记初值	只看递推式，没有看初始项
下标错一位	从 0 开始还是从 1 开始没有分清
循环起点错	已知前两项却从第 2 项开始递推
递推方向错	当前项依赖后面的项，却从前往后算
取模时机看错	程序每一步取模，手算时漏取模
滚动更新错	更新顺序导致旧值被覆盖
通项乱套	初值改变后仍套标准公式
把计数含义丢掉	只看公式，不看状态表示什么
输出项错	输出 `f[n]`、`f[n-1]` 或滚动变量时混淆

做这类题时，草稿纸上最好写出前几项。前几项一写出来，很多下标和初值错误会立即暴露。

原创例题

例题 21：递推输出

阅读程序：

```
int f[20];
f[1] = 2;
f[2] = 5;
for (int i = 3; i <= 8; i++) {
    f[i] = f[i - 1] + 2 * f[i - 2];
}
cout << f[8] << endl;
```

逐项计算：

i	$f[i]$
1	2
2	5
3	$5+2\times 2=9$
4	$9+2\times 5=19$
5	$19+2\times 9=37$
6	$37+2\times 19=75$
7	$75+2\times 37=149$
8	$149+2\times 75=299$

所以输出 299。

这道题不需要找通项。按表格从初值往后算，最稳。

变式 1：循环少一轮

如果循环条件改成：

```
for (int i = 3; i < 8; i++)
```

那么最后只计算到 `f[7]`，`f[8]` 没有被这段循环赋值。若 `f` 是全局数组，`f[8]` 初始为 0；若是局部数组，值不确定。程序阅读题中要看变量定义位置和循环边界。

变式 2：加入取模

若递推改成：

```
f[i] = (f[i - 1] + 2 * f[i - 2]) % 100;
```

则从每一步开始都要保留后两位。以上序列到 $f[8]$ 时：

```
f[1]=2
f[2]=5
f[3]=9
f[4]=19
f[5]=37
f[6]=75
f[7]=(75+74)%100=49
f[8]=(49+150)%100=99
```

输出变为 99。

取模后不能继续沿用未取模序列的中间值做程序追踪。

小结

递推数列的题目通常不需要复杂技巧，但对初值和下标很敏感。把前几项写清楚，确认每一项从哪些项来，再看循环边界和输出位置，递推题就会稳定很多。下一章进入模运算与逆元，递推中的取模会在那里继续出现。

0x0F 模运算与逆元

模运算是 CSP-S 第一轮中很常见的数学工具。它经常出现在选择题、程序阅读和完善程序中：判断表达式余数、读懂取模递推、分析循环节、补全快速幂或逆元代码。

很多学生对模运算的第一印象是 “防止数太大”。这只说对了一部分。模运算更重要的作用，是把整数按余数分类。两个很大的数，只要它们除以同一个模数的余数相同，在许多运算中就可以看成同一类。

本章先讲同余和取模规则，再讲 C++ 中 `%` 的细节，最后讲逆元。逆元本质上是在模意义下做除法，它比普通取模更容易出错，需要格外注意适用条件。

余数

给定正整数 m，一个整数 a 除以 m 后，可以写成：

$$a = qm + r, \quad 0 \le r < m.$$

这里 r 就是 a 除以 m 的余数，记作：

$$a \bmod m = r.$$

例如：

$$17 = 3 \times 5 + 2,$$

所以：

$$17 \bmod 5 = 2.$$

又如：

$$36 = 4 \times 9 + 0,$$

所以：

$$36 \bmod 9 = 0.$$

取模题中，模数 m 通常为正数。余数的标准范围是 0 到 $m-1$。

例题 1：简单取模

计算：

$$12345 \bmod 7.$$

可以直接除：

$$12345 = 1763 \times 7 + 4.$$

所以：

$$12345 \bmod 7 = 4.$$

也可以逐步处理：

$$12345 = 12000 + 345.$$

若知道 $12000 \bmod 7$ 和 $345 \bmod 7$，也可以合并。大数取模时，逐位取模更常用，后面会讲。

同余

如果两个整数 a, b 除以 m 的余数相同，就说 a 与 b 模 m 同余，记作：

$$a \equiv b \pmod{m}.$$

例如：

$$17 \bmod 5 = 2, \quad 12 \bmod 5 = 2,$$

所以：

$$17 \equiv 12 \pmod{5}.$$

同余也可以理解为 $a - b$ 能被 m 整除。因为 $17 - 12 = 5$，可以被 5 整除。

例题 2：判断同余

判断 2025 和 25 是否模 8 同余。

计算余数：

$$2025 \bmod 8 = 1, \quad 25 \bmod 8 = 1.$$

所以：

$$2025 \equiv 25 \pmod{8}.$$

也可以看差：

$$2025 - 25 = 2000.$$

因为 2000 能被 8 整除，所以两者模 8 同余。

模运算的加减乘

模运算中，加法、减法、乘法可以先取模再计算。

若：

$$a \equiv b \pmod{m}, \quad c \equiv d \pmod{m},$$

则：

$$a + c \equiv b + d \pmod{m},$$

$$a - c \equiv b - d \pmod{m},$$

$$ac \equiv bd \pmod{m}.$$

程序中常用这些规则：

$$(a + b) \bmod m = ((a \bmod m) + (b \bmod m)) \bmod m,$$

$$(a - b) \bmod m = ((a \bmod m) - (b \bmod m)) \bmod m,$$

$$(ab) \bmod m = ((a \bmod m)(b \bmod m)) \bmod m.$$

例题 3：分步取模

计算：

$$(123456789 \times 987654321) \bmod 10.$$

只看个位即可：

$$123456789 \bmod 10 = 9,$$

$$987654321 \bmod 10 = 1.$$

所以：

$$(123456789 \times 987654321) \bmod 10 = (9 \times 1) \bmod 10 = 9.$$

不需要把乘积完整算出来。

例题 4：加法取模

计算：

$$(2025 + 9876) \bmod 7.$$

先分别取模：

$$2025 \bmod 7 = 2,$$

$$9876 \bmod 7 = 6.$$

所以：

$$(2025 + 9876) \bmod 7 = (2 + 6) \bmod 7 = 1.$$

答案是 1。

减法取模

减法取模容易出现负数。数学上，余数通常要求在 0 到 $m - 1$ 之间。

例如：

$$3 - 5 = -2.$$

在模 7 意义下，-2 和 5 同余，因为：

$$-2 + 7 = 5.$$

所以：

$$(3 - 5) \bmod 7 = 5.$$

程序中常写：

```
ans = (a - b + mod) % mod;
```

若 a 和 b 已经在 0 到 $mod - 1$ 之间，这样可以避免结果为负。

如果一次加 mod 仍可能不够，例如 a-b 很小，可以写成：

```
ans = ((a - b) % mod + mod) % mod;
```

第一轮中，常见情况是两个已经取模的数相减，加一次 mod 就够。

例题 5：减法取模

设 $mod = 11$，计算：

$$(4-9) \bmod 11.$$

有：

$$4-9=-5.$$

加上 11：

$$-5+11=6.$$

所以结果为 6。

程序写法：

```
int ans = (4 - 9 + 11) % 11;
```

得到 6。

C++ 中的 %

C++ 中的 % 是整数取余运算。对于正数，它和数学中的余数一致。

```
cout << 17 % 5;
```

输出 2。

但当被取模的数是负数时，C++ 的结果可能是负数。例如：

```
cout << -2 % 7;
```

在 C++ 中结果是 -2，而数学中常用的非负余数是 5。

因此程序中若要保证余数非负，常写：

```
int r = ((x % mod) + mod) % mod;
```

例题 6：程序输出

```
int x = -2;
int mod = 7;
cout << x % mod << endl;
cout << (x % mod + mod) % mod << endl;
```

第一行输出 C++ 取余结果：

```
-2
```

第二行把它调整到 0 到 6 的范围内，输出：

```
5
```

程序阅读题中，看到负数取模，要按 C++ 规则判断；若程序又加了 mod，要看最后是否变成非负余数。

大数逐位取模

如果一个十进制大整数用字符串表示，无法直接放进普通整数变量，可以逐位取模。

例如要计算字符串 `s` 表示的十进制整数对 `mod` 的余数：

```
int r = 0;
for (int i = 0; i < s.size(); i++) {
    r = (r * 10 + s[i] - '0') % mod;
}
```

这和第 10 章中字符串转十进制的过程相同，只是每一步都取模，避免数变大。

例题 7：逐位取模

计算字符串 `12345` 对 7 的余数。

按程序过程：

已读字符	新的 `r`
1	$(0\times 10+1)\bmod 7=1$
2	$(1\times 10+2)\bmod 7=5$
3	$(5\times 10+3)\bmod 7=4$
4	$(4\times 10+4)\bmod 7=2$
5	$(2\times 10+5)\bmod 7=4$

所以余数为 4。

这和直接计算 $12345 \bmod 7=4$ 一致。

取模递推

递推数列中常出现取模。

```
f[1] = 1;
f[2] = 1;
for (int i = 3; i <= n; i++) {
    f[i] = (f[i - 1] + f[i - 2]) % mod;
}
```

这段程序计算的是斐波那契数列对 `mod` 的余数。

由于加法可以先取模，所以每一步只保留余数不会影响最终余数。

例题 8：小模数递推

设 $mod=5$，$f_1=1, f_2=1$，且：

$$f_i=(f_{i-1}+f_{i-2}) \bmod 5.$$

求 f_8。

逐项计算：

i	原和	f_i
1	—	1
2	—	1
3	$1+1$	2
4	$1+2$	3
5	$2+3$	0
6	$3+0$	3
7	$0+3$	3
8	$3+3$	1

所以 $f_8 = 1$。

手算时要每一步都按程序取模，不能只算普通数列后随意截断中间过程。

快速幂

快速幂用于计算：

$$a^b \bmod m.$$

若直接乘 b 次，复杂度是 $O(b)$。快速幂利用二进制拆分指数，把复杂度降到 $O(\log b)$。

常见代码如下：

```
long long qpow(long long a, long long b, long long mod) {
    long long res = 1;
    a %= mod;
    while (b > 0) {
        if (b % 2 == 1) res = res * a % mod;
        a = a * a % mod;
        b /= 2;
    }
    return res;
}
```

读这段程序时，可以这样理解：

- `res` 保存已经选中的幂的乘积；
- `a` 每轮平方，依次表示 $a, a^2, a^4, a^8, \ldots$；
- `b` 每轮除以 2，相当于从低位到高位查看指数的二进制位；
- 当前二进制位为 1 时，把当前的 `a` 乘进 `res`。

例题 9：快速幂手算

计算 $3^{13} \bmod 7$。

指数 13 的二进制为：

$$13 = 8 + 4 + 1.$$

所以：

$$3^{13} = 3^8 \cdot 3^4 \cdot 3^1.$$

逐步取模：

$$3^1 \bmod 7 = 3,$$

$$3^2 \bmod 7 = 9 \bmod 7 = 2,$$

$$3^4 \bmod 7 = 2^2 \bmod 7 = 4,$$

$$3^8 \bmod 7 = 4^2 \bmod 7 = 2.$$

于是：

$$3^{13} \bmod 7 = (2 \times 4 \times 3) \bmod 7 = 24 \bmod 7 = 3.$$

答案是 3。

例题 10：程序追踪

```
long long res = 1;
long long a = 3;
long long b = 5;
long long mod = 7;

while (b > 0) {
    if (b % 2 == 1) res = res * a % mod;
    a = a * a % mod;
    b /= 2;
}
cout << res << endl;
```

追踪过程：

轮次	b	是否乘入 res	更新后 res	更新后 a	更新后 b
1	5	是	3	2	2
2	2	否	3	4	1
3	1	是	5	2	0

输出 5。这对应：

$$3^5 = 243, \quad 243 \bmod 7 = 5.$$

程序阅读题中，快速幂变量变化很适合用表格追踪。

模意义下的除法

在普通整数中，若要计算：

$$\frac{a}{b},$$

可以直接除。但在模运算中，不能简单写成：

$$(a \bmod m)/(b \bmod m).$$

例如在模 5 意义下：

$$6 \equiv 1 \pmod 5.$$

但：

$$6/2 = 3,$$

而：

$$1/2$$

在整数中并不是整数。模意义下的“除以 b”，需要转成“乘以 b 的逆元”。

如果存在整数 x，满足：

$$bx \equiv 1 \pmod m,$$

那么 x 就叫作 b 在模 m 意义下的逆元，记作 b^{-1}。

有了逆元，模意义下除以 b 就可以写成乘以 b^{-1}：

$$\frac{a}{b} \equiv a \cdot b^{-1} \pmod m.$$

例题 11：找逆元

求 3 在模 7 意义下的逆元。

需要找 x，使得：

$$3x \equiv 1 \pmod 7.$$

试几个数：

x	$3x \bmod 7$
1	3
2	6
3	2
4	5
5	1

当 $x = 5$ 时满足条件，所以 3 在模 7 意义下的逆元是 5。

检查：

$$3 \times 5 = 15 \equiv 1 \pmod 7.$$

因此，在模 7 下，除以 3 等价于乘以 5。

逆元不一定存在

不是每个数都有逆元。整数 a 在模 m 意义下有逆元，需要 a 和 m 没有除 1 以外的公共因子。这个条件会在第 16 章用 gcd 系统说明。

先看一个反例。

求 2 在模 6 意义下的逆元，也就是找 x 使得：

$$2x \equiv 1 \pmod 6.$$

枚举 $x = 0, 1, 2, 3, 4, 5$：

x	$2x \bmod 6$
0	0
1	2
2	4
3	0
4	2
5	4

结果只可能是 $0, 2, 4$，不会得到 1。所以 2 在模 6 意义下没有逆元。

第一轮中，如果题目问“是否一定可以除以某个数”，要想到逆元存在条件。模运算中不能随便约分或除法。

质数模数下的逆元

如果模数 p 是质数，且 a 不是 p 的倍数，那么 a 在模 p 意义下有逆元。

常用公式是：

$$a^{-1} \equiv a^{p-2} \pmod p.$$

这个公式来自费马小定理。第一轮中，通常只需要会用，不需要证明。

因此可以用快速幂求逆元：

```
long long inv(long long a, long long p) {
    return qpow(a, p - 2, p);
}
```

前提是：p 是质数，且 a 不能被 p 整除。

例题 12：用快速幂求逆元

求 3 在模 7 意义下的逆元。

因为 7 是质数，且 3 不是 7 的倍数，所以：

$$3^{-1} \equiv 3^{7-2} = 3^5 \pmod 7.$$

计算：

$$3^2 \bmod 7 = 2,$$

$$3^4 \bmod 7 = 4,$$

$$3^5 \bmod 7 = 3^4 \cdot 3 \bmod 7 = 4 \times 3 \bmod 7 = 5.$$

所以逆元为 5，和前面枚举结果一致。

组合数取模中的逆元

排列组合题中经常出现除法。例如：

$$C_n^k = \frac{n!}{k!(n-k)!}.$$

如果要求对质数 mod 取模，不能直接做整数除法。常用方法是把除法改成乘逆元：

$$C_n^k \bmod mod = n! \cdot (k!)^{-1} \cdot ((n-k)!)^{-1} \bmod mod.$$

程序中可以预处理阶乘和逆元阶乘：

```
fac[0] = 1;
for (int i = 1; i <= n; i++) {
    fac[i] = fac[i - 1] * i % mod;
}

ifac[n] = qpow(fac[n], mod - 2, mod);
for (int i = n; i >= 1; i--) {
    ifac[i - 1] = ifac[i] * i % mod;
}
```

然后：

```
long long C(int n, int k) {
    if (k < 0 || k > n) return 0;
    return fac[n] * ifac[k] % mod * ifac[n - k] % mod;
}
```

第一轮中，可能不要求完整写出组合数取模代码，但程序阅读和完善程序会考这些语句的含义。

例题 13：组合数取模

求 $C_5^2 \bmod 7$。

普通计算：

$$C_5^2 = 10.$$

所以：

$$10 \bmod 7 = 3.$$

用逆元写也可以：

$$C_5^2 = \frac{5!}{2!3!}.$$

在模 7 下：

$$5! \bmod 7 = 120 \bmod 7 = 1,$$

$$2! = 2, \quad 3! = 6.$$

2 在模 7 下的逆元是 4，6 在模 7 下的逆元是 6。于是：

$$C_5^2 \bmod 7 = 1 \times 4 \times 6 \bmod 7 = 24 \bmod 7 = 3.$$

结果一致。

约分要谨慎

在模运算中，不能随便把等式两边除以同一个数。

例如：

$$2x \equiv 2 \pmod 6.$$

如果像普通方程那样两边除以 2，会得到：

$$x \equiv 1 \pmod 6.$$

但实际上原式的解不止这一类。检查：

$$x = 1 \Rightarrow 2x = 2 \equiv 2 \pmod 6,$$

$$x = 4 \Rightarrow 2x = 8 \equiv 2 \pmod 6.$$

所以 $x \equiv 4 \pmod 6$ 也满足原式。

问题在于 2 在模 6 下没有逆元，不能直接除以 2。

如果要在模 m 下除以 a，必须先确认 a 有逆元。第一轮中，一旦看到“模意义下除法”或“约掉某个因子”，都要检查条件。

程序阅读中的模运算

模运算程序阅读常见问题有：

- 输出某个表达式的余数；
- 判断负数取模结果；
- 判断取模递推的某一项；
- 读懂快速幂；
- 判断逆元代码是否适用；
- 判断组合数取模程序中的数组含义。

例题 14：表达式追踪

```
int mod = 7;
int x = 1;
for (int i = 1; i <= 5; i++) {
    x = x * 3 % mod;
}
cout << x << endl;
```

这段程序计算的是：

$$3^5 \bmod 7.$$

逐项计算：

i	x
初始	1
1	3
2	2
3	6
4	4
5	5

输出 5。

例题 15：数组含义

```
fac[0] = 1;
for (int i = 1; i <= n; i++) {
    fac[i] = fac[i - 1] * i % mod;
}
```

这里 `fac[i]` 表示：

$$i! \bmod mod.$$

如果后面出现：

```
fac[n] * ifac[k] % mod * ifac[n - k] % mod
```

就要想到它在计算组合数 C_n^k 对 mod 的余数。

完善程序中的模运算

模运算完善程序常见空缺包括：

- 加法后取模；
- 减法防负；
- 乘法取模；
- 快速幂中指数奇偶判断；
- 快速幂中底数平方；
- 逆元的指数 $mod-2$；
- 组合数公式中的逆元阶乘。

例题 16：补全快速幂

```
long long qpow(long long a, long long b, long long mod) {
    long long res = 1;
    a %= mod;
    while (b > 0) {
        if (_____) res = res * a % mod;
        a = _____;
        b /= 2;
    }
    return res;
}
```

第一空判断当前指数最低位是否为 1，应填：

```
b % 2 == 1
```

第二空更新底数平方，应填：

```
a * a % mod
```

完整含义是：若当前二进制位为 1，就把当前幂乘进答案；然后底数平方，指数右移一位。

例题 17：补全逆元

若 mod 是质数，且 a 不是 mod 的倍数：

```
long long inv(long long a) {
    return qpow(a, _____, mod);
}
```

空缺应填：

```
mod - 2
```

因为：

$$a^{-1} \equiv a^{mod-2} \pmod{mod}.$$

这里的前提很重要：mod 是质数，且 a 与 mod 没有公共因子。若题目没有这些条件，就不能直接使用这段代码。

例题 18：补全减法取模

```
ans = (x - y + _____) % mod;
```

若已知 x 和 y 都在 0 到 $mod - 1$ 之间，为了避免负数，应填：

```
mod
```

这样 x-y+mod 至少为 0，再 % mod 得到标准余数。

常见错误

模运算与逆元题常见错误如下。

错误	说明
负数取模误判	C++ 中负数 % 可能得到负数
减法忘记加模数	(a-b)%mod 可能为负
乘法中间溢出	大数相乘时要用更大的整数类型
把除法直接取模	模意义下除法要乘逆元
逆元条件漏掉	不是所有数都有逆元
质数公式乱用	a^{p-2} 公式要求模数为质数且 a 不为 p 的倍数
快速幂更新错	a=a*a%mod 和 b/=2 缺一不可
组合数取模直接除	不能写 fac[n] / fac[k] / fac[n-k] % mod
约分不检查条件	没有逆元时不能随便约掉因子

第一轮中，模运算题常把错误藏在“看起来像普通算术”的地方。只要出现除法、负数、逆元，就要放慢一步。

原创例题

例题 19：程序输出

阅读程序：

```
int mod = 9;
int x = 0;
for (int i = 1; i <= 6; i++) {
    x = (x * 10 + i) % mod;
}
cout << x << endl;
```

这段程序依次构造数字 123456，但每一步都对 9 取模。因此输出的是：

$$123456 \bmod 9.$$

因为一个数对 9 的余数等于它各位数字和对 9 的余数，可以快速算：

$$1+2+3+4+5+6=21, \quad 21 \bmod 9 = 3.$$

也可以按程序追踪：

i	x=(x*10+i)%9
1	1
2	3
3	6
4	1
5	6
6	3

输出 3。

变式 1：改成模 7

若把 mod 改成 7，就不能直接使用“数位和模 9”的规则。按程序追踪：

i	x
1	1
2	5
3	4
4	2
5	4
6	4

输出 4。

同样的程序，模数不同，能用的简化规则也不同。

例题 20：逆元判断

判断下面说法是否正确：在模 8 意义下，3 的逆元是 3。

检查：

$$3 \times 3 = 9 \equiv 1 \pmod 8.$$

所以说法正确。

再判断：在模 8 意义下，2 有逆元。

枚举：

$$2 \times 0, 2 \times 1, 2 \times 2, \ldots$$

对 8 的余数只可能是 $0, 2, 4, 6$，不可能是 1。所以 2 没有逆元。

这说明模数不一定要是质数才存在逆元，但如果模数是质数，判断会更简单。一般条件与 gcd 有关，下一章会继续讲。

小结

模运算的重点不是把数变小这么简单，而是理解“同余类”下哪些运算仍然成立。加、减、乘可以稳定处理，除法必须依赖逆元。第 16 章会接着讲 gcd 与 lcm，用来补上逆元存在条件和共同倍数判断的基础。

0x10 gcd 与 lcm

最大公约数和最小公倍数是数论中很基础的两个概念。它们经常出现在 CSP-S 第一轮的选择题、程序阅读题和完善程序题中：判断两个数是否互质，求分数约分结果，计算循环周期，判断模意义下逆元是否存在，读懂辗转相除法程序。

本章只讲第一轮常见范围。扩展欧几里得、线性同余方程等内容不作为本章主线。需要记住的是：`gcd` 处理“共同除数”，`lcm` 处理“共同倍数”；二者关系紧密，但不能混用。

最大公约数

若正整数 d 同时整除 a 和 b，则 d 是 a, b 的公约数。所有公约数中最大的一个，叫作 a, b 的最大公约数，记作：

$$\gcd(a, b).$$

例如 12 的正因数为：

$$1, 2, 3, 4, 6, 12.$$

18 的正因数为：

$$1, 2, 3, 6, 9, 18.$$

共同因数为：

$$1, 2, 3, 6.$$

所以：

$$\gcd(12, 18) = 6.$$

最大公约数反映的是两个数“共同能被多大的数整除”。

例题 1：求最大公约数

求 $\gcd(84, 60)$。

先列因数也可以，但更常用的是分解质因数：

$$84 = 2^2 \times 3 \times 7,$$

$$60 = 2^2 \times 3 \times 5.$$

共同部分为 $2^2 \times 3$，所以：

$$\gcd(84, 60) = 2^2 \times 3 = 12.$$

答案是 12。

分解质因数适合小数手算；程序中更常用辗转相除法。

最小公倍数

若正整数 m 同时是 a 和 b 的倍数，则 m 是 a, b 的公倍数。所有正公倍数中最小的一个，叫作 a, b 的最小公倍数，记作：

$$\operatorname{lcm}(a, b).$$

例如 12 的正倍数有：

$$12, 24, 36, 48, 60, 72, \ldots$$

18 的正倍数有：

$$18, 36, 54, 72, \ldots$$

共同倍数中最小的是 36，所以：

$$\operatorname{lcm}(12, 18) = 36.$$

最小公倍数常用于 “两个周期什么时候同时出现”“同时是几个数的倍数” 这类题。

例题 2：求最小公倍数

求 $\operatorname{lcm}(84, 60)$。

分解质因数：

$$84 = 2^2 \times 3 \times 7,$$

$$60 = 2^2 \times 3 \times 5.$$

最小公倍数要包含两个数中每个质因子的最高次数：

$$\operatorname{lcm}(84, 60) = 2^2 \times 3 \times 5 \times 7 = 420.$$

答案是 420。

最大公约数取 “共同部分”，最小公倍数取 “合并后足够包含两边的部分”。

gcd 与 lcm 的关系

对于正整数 a, b，有一个非常常用的关系：

$$\gcd(a,b) \times \operatorname{lcm}(a,b) = ab.$$

因此：

$$\operatorname{lcm}(a,b) = \frac{ab}{\gcd(a,b)}.$$

程序中通常写成：

```
long long lcm(long long a, long long b) {
    return a / gcd(a, b) * b;
}
```

先除再乘，是为了减少中间乘积过大的风险。

例题 3：由 gcd 求 lcm

已知 $a = 84, b = 60$，且 $\gcd(84,60) = 12$。求 $\operatorname{lcm}(84,60)$。

由公式：

$$\operatorname{lcm}(84,60) = \frac{84 \times 60}{12} = 420.$$

和上一节分解质因数得到的结果一致。

变式：已知乘积和 gcd

若两个正整数 a, b 的乘积为 360，且 $\gcd(a,b) = 6$，求 $\operatorname{lcm}(a,b)$。

由：

$$\gcd(a,b) \times \operatorname{lcm}(a,b) = ab,$$

得到：

$$6 \times \operatorname{lcm}(a,b) = 360.$$

所以：

$$\operatorname{lcm}(a,b) = 60.$$

这类题不需要知道 a, b 分别是多少。

互质

如果两个正整数的最大公约数为 1，就称它们互质。

例如：

$$\gcd(8,15)=1,$$

所以 8 和 15 互质。

但互质不要求两个数本身都是质数。8 和 15 都不是质数，却互质。

例题 4：判断互质

判断 35 和 64 是否互质。

分解：

$$35=5\times 7,$$

$$64=2^6.$$

它们没有公共质因子，所以：

$$\gcd(35,64)=1.$$

因此 35 和 64 互质。

互质与逆元

上一章讲过，整数 a 在模 m 意义下有逆元，当且仅当：

$$\gcd(a,m)=1.$$

例如 3 在模 7 下有逆元，因为 $\gcd(3,7)=1$；2 在模 6 下没有逆元，因为 $\gcd(2,6)=2$。

第一轮中，如果题目问“模 m 下能否除以 a”，不要直接做除法，要先检查 a 和 m 是否互质。

辗转相除法

求最大公约数最常用的方法是辗转相除法，也叫欧几里得算法。

核心性质是：

$$\gcd(a,b)=\gcd(b,a \bmod b)\quad (b\neq 0).$$

不断把较大的数对较小的数取余，直到余数为 0。最后非零的那个数就是最大公约数。

例题 5：手算辗转相除法

求 $\gcd(84,60)$。

按辗转相除：

$$84=1\times 60+24,$$

$$60 = 2 \times 24 + 12,$$

$$24 = 2 \times 12 + 0.$$

最后非零余数是 12，所以：

$$\gcd(84, 60) = 12.$$

这里的每一步都把问题变小。原来的 $\gcd(84, 60)$ 变成 $\gcd(60, 24)$，再变成 $\gcd(24, 12)$，最后得到 12。

例题 6：另一组数

求 $\gcd(2025, 756)$。

辗转相除：

$$2025 = 2 \times 756 + 513,$$

$$756 = 1 \times 513 + 243,$$

$$513 = 2 \times 243 + 27,$$

$$243 = 9 \times 27 + 0.$$

所以：

$$\gcd(2025, 756) = 27.$$

这类题用辗转相除比列因数更稳。

gcd 程序

辗转相除法可以写成循环。

```
int gcd(int a, int b) {
    while (b != 0) {
        int r = a % b;
        a = b;
        b = r;
    }
    return a;
}
```

也可以写成递归。

```
int gcd(int a, int b) {
    if (b == 0) return a;
    return gcd(b, a % b);
}
```

两种写法含义一样。

例题 7：程序追踪

阅读程序：

```
int gcd(int a, int b) {
    while (b != 0) {
        int r = a % b;
        a = b;
        b = r;
    }
    return a;
}
```

调用 `gcd(84,60)`。

追踪：

轮次	a	b	r=a%b
初始	84	60	—
1	60	24	24
2	24	12	12
3	12	0	0

循环结束，返回 12。

程序阅读题中，辗转相除法很适合用表格追踪。只要记录 `a,b,r` 三列即可。

为什么辗转相除法正确

第一轮不需要长证明，但需要知道它为什么可以这样换。

设 $a = qb + r$，其中 $r = a \bmod b$。

如果某个数 d 同时整除 a 和 b，那么 d 也整除 $a - qb$，也就是整除 r。

反过来，如果 d 同时整除 b 和 r，那么 d 也整除 $qb + r$，也就是整除 a。

因此，a, b 的公约数集合和 b, r 的公约数集合相同，最大公约数也相同：

$$\gcd(a, b) = \gcd(b, r).$$

这就是辗转相除法的基础。

特殊值

在第一轮中，通常处理正整数。但程序中可能出现 0。

常用约定是：

$$\gcd(a,0)=a \quad (a>0),$$

$$\gcd(0,a)=a \quad (a>0).$$

如果 $a=b=0$，有些数学语境下不定义；程序题中一般会避免这种输入，或单独规定。

最小公倍数通常讨论正整数。若其中一个数为 0，有些程序会约定 `lcm(0,a)=0`，但第一轮选择题若未说明，通常不要把 0 的情况随意套进正整数公式。

例题 8：含 0 的 gcd

调用下面函数：

```
int gcd(int a, int b) {
    if (b == 0) return a;
    return gcd(b, a % b);
}
```

求 `gcd(15,0)` 的返回值。

一进入函数，`b == 0` 成立，直接返回 `a`，即 15。

所以返回值为 15。

再看 `gcd(0,15)`：

```
gcd(0,15) -> gcd(15,0) -> 15
```

返回值也是 15。

用 gcd 约分

分数约分时，需要用分子和分母的最大公约数同时除掉。

若分数为：

$$\frac{a}{b},$$

令：

$$g=\gcd(a,b).$$

约分后为：

$$\frac{a/g}{b/g}.$$

例题 9：分数约分

把 $\frac{84}{60}$ 约分。

前面已经知道：

$$\gcd(84, 60) = 12.$$

所以：

$$\frac{84}{60} = \frac{84/12}{60/12} = \frac{7}{5}.$$

约分后的分子和分母互质：

$$\gcd(7, 5) = 1.$$

程序中常写：

```
int g = gcd(a, b);
a /= g;
b /= g;
```

如果分母可能为负数，还要统一符号。但 CSP-S 第一轮的基础题通常会避开复杂符号处理。

lcm 与周期

最小公倍数常用来处理“多个周期同时发生”。

例题 10：两个周期同时出现

甲事件每 6 天发生一次，乙事件每 8 天发生一次。若今天同时发生，下一次同时发生在多少天后？

下一次同时发生的时间，必须同时是 6 和 8 的倍数。最小的正时间是：

$$\operatorname{lcm}(6, 8) = 24.$$

所以 24 天后再次同时发生。

例题 11：多个周期

三个事件周期分别为 $4, 6, 10$。若今天同时发生，下一次同时发生在多少时间单位后？

先求：

$$\operatorname{lcm}(4, 6) = 12.$$

再求：

$$\operatorname{lcm}(12, 10) = 60.$$

所以答案为 60。

多个数的最小公倍数可以逐个合并：

```
ans = lcm(ans, x);
```

但要注意中间结果可能较大。

多个数的 gcd

多个数的最大公约数也可以逐个合并：

$$\gcd(a,b,c) = \gcd(\gcd(a,b),c).$$

程序写法：

```
int g = a[1];
for (int i = 2; i <= n; i++) {
    g = gcd(g, a[i]);
}
```

例题 12：多个数的最大公约数

求 $24, 36, 60$ 的最大公约数。

先求：

$$\gcd(24,36) = 12.$$

再求：

$$\gcd(12,60) = 12.$$

所以：

$$\gcd(24,36,60) = 12.$$

变式：多个数的最小公倍数

求 $6, 10, 15$ 的最小公倍数。

先求：

$$\operatorname{lcm}(6,10) = 30.$$

再求：

$$\operatorname{lcm}(30,15) = 30.$$

所以答案为 30。

最大公约数和质因数

用质因数分解看 gcd 和 lcm，会更容易理解它们的区别。

设：

$$a = 2^3 \times 3^1 \times 5^2,$$

$$b = 2^1 \times 3^2 \times 7^1.$$

最大公约数取公共质因子的较小指数：

$$\gcd(a, b) = 2^1 \times 3^1.$$

最小公倍数取出现过的质因子的较大指数：

$$\operatorname{lcm}(a, b) = 2^3 \times 3^2 \times 5^2 \times 7^1.$$

可以记成：

```
gcd：共同拥有的部分，指数取小
lcm：合并覆盖两边，指数取大
```

这不是程序中最常用的方法，但在选择题手算中很有用。

gcd 与互质计数

第一轮中有时会问“某些数中与 n 互质的数有多少个”。如果范围很小，可以直接用 gcd 判断。

例题 13：小范围互质计数

求 1 到 10 中，与 10 互质的正整数个数。

逐个判断：

数	与 10 的 gcd	是否互质
1	1	是
2	2	否
3	1	是
4	2	否
5	5	否
6	2	否
7	1	是
8	2	否
9	1	是
10	10	否

共有 4 个：$1, 3, 7, 9$。

更系统的欧拉函数不在本书主线中展开。对于第一轮小范围题，会用 gcd 判断即可。

程序阅读中的 gcd

gcd 程序阅读常见问题包括：

- 判断函数返回值；

- 判断循环次数；
- 判断 a,b 的变化；
- 判断某个条件是否表示互质；
- 判断 lcm 是否可能计算错误；
- 判断分数约分程序含义。

例题 14：判断函数含义

```
int f(int a, int b) {
    while (b) {
        int t = a % b;
        a = b;
        b = t;
    }
    return a;
}
```

这个函数就是辗转相除法，返回 $\gcd(a,b)$。

如果后面出现：

```
if (f(x, y) == 1) cout << "YES";
```

它判断的是 x 和 y 是否互质。

例题 15：lcm 程序

```
long long gcd(long long a, long long b) {
    while (b) {
        long long t = a % b;
        a = b;
        b = t;
    }
    return a;
}

long long lcm(long long a, long long b) {
    return a / gcd(a, b) * b;
}
```

`lcm` 函数先除以最大公约数，再乘以另一个数。它返回的是最小公倍数。

若写成：

```
return a * b / gcd(a, b);
```

数学上结果相同，但中间的 `a*b` 可能更大。程序阅读题中，如果数据范围较大，要注意中间结果是否可能超过变量类型范围。

完善程序中的 gcd 与 lcm

完善程序题中，gcd 与 lcm 常见空缺包括：

- 辗转相除中的余数；
- 更新 a,b 的顺序；

- 递归终止条件；
- lcm 公式中的除法位置；
- 判断互质；
- 分数约分。

例题 16：补全 gcd 函数

```
int gcd(int a, int b) {
    while (b != 0) {
        int r = _____;
        a = b;
        b = r;
    }
    return a;
}
```

空缺应填：

```
a % b
```

因为辗转相除法要把 $\gcd(a,b)$ 转成 $\gcd(b, a \bmod b)$。

例题 17：补全递归 gcd

```
int gcd(int a, int b) {
    if (_____) return a;
    return gcd(b, a % b);
}
```

空缺应填：

```
b == 0
```

当第二个数为 0 时，最大公约数就是第一个数。

例题 18：补全 lcm

```
long long lcm(long long a, long long b) {
    return _____;
}
```

应填：

```
a / gcd(a, b) * b
```

这比 `a * b / gcd(a,b)` 更适合程序实现，因为先除可以减小中间结果。

例题 19：补全约分

```
int g = gcd(a, b);
a = _____;
b = _____;
```

约分时，分子分母都要除以最大公约数，所以应填：

```
a / g
b / g
```

如果只除分子或只除分母，分数值会改变。

常见错误

gcd 与 lcm 题的错误集中在下面几类。

错误	说明
混淆 gcd 和 lcm	一个处理共同除数，一个处理共同倍数
lcm 忘记除 gcd	直接写 ab 会重复计算共同因子
辗转相除更新顺序错	应把 (a,b) 变成 $(b,a \bmod b)$
递归边界错	`b==0` 时返回 `a`
互质判断错	互质是 gcd 为 1，不是两个数都是质数
模逆元条件漏掉	a 在模 m 下有逆元要求 $\gcd(a,m)=1$
分数约分只除一边	分子分母都要除以 gcd
lcm 中间结果过大	程序里应尽量先除再乘
多个数 lcm 乱乘	应逐步取 lcm，而不是直接相乘

做这类题时，先判断题目要的是“共同除数”还是“共同倍数”。如果是周期同时出现，多半用 lcm；如果是约分、互质、逆元条件，多半用 gcd。

原创例题

例题 20：程序输出

阅读程序：

```
int gcd(int a, int b) {
    if (b == 0) return a;
    return gcd(b, a % b);
}

int main() {
    int a = 48, b = 18;
    int g = gcd(a, b);
    cout << g << ' ' << a / g << ' ' << b / g << endl;
    return 0;
}
```

先求：

$$\gcd(48,18).$$

辗转相除：

$$48=2\times 18+12,$$

$$18 = 1 \times 12 + 6,$$

$$12 = 2 \times 6 + 0.$$

所以 $g = 6$。

程序输出：

$$6, \quad 48/6 = 8, \quad 18/6 = 3.$$

因此输出为：

```
6 8 3
```

这段程序先求最大公约数，再输出约分后的两个数。

变式 1：改成 lcm

若主函数改成：

```
cout << a / gcd(a, b) * b << endl;
```

则输出：

$$48/6 \times 18 = 8 \times 18 = 144.$$

也就是 $\mathrm{lcm}(48, 18)$。

变式 2：判断逆元

若题目问 18 在模 48 意义下是否有逆元，由于：

$$\gcd(18, 48) = 6 \neq 1,$$

所以没有逆元。

如果问 5 在模 48 意义下是否有逆元，由于：

$$\gcd(5, 48) = 1,$$

所以有逆元。

小结

最大公约数和最小公倍数并不复杂，但它们连接了很多第一轮知识点：约分、周期、互质、逆元、容斥中的共同倍数。到这一章结束，第三编“数学与计数”的主体内容已经完成。下一编开始进入数据结构，题目会更多地转向栈、队列、堆、哈希表和树结构的程序行为。

第四编 数据结构

0x11 栈与队列

栈和队列是最基础、也最容易在程序阅读中出题的数据结构。它们的操作不复杂，但一旦放进循环、条件判断、递归模拟、括号匹配或序列判断中，就很容易因为进出顺序看错而失分。

栈的规则是后进先出。最后放进去的元素，最先被取出来。

队列的规则是先进先出。最早放进去的元素，最早被取出来。

这两句话很短，但第一轮考查的重点并不是背定义，而是看懂程序中元素何时进入、何时离开、当前结构中还剩什么、下一次取出的会是谁。本章重点讲纸面追踪方法、合法序列判断、程序阅读和完善程序中的常见空缺。

栈

栈可以想成一摞盘子。新盘子只能放在最上面，取盘子也只能从最上面取。

常见操作如下。

操作	含义
`push(x)`	把元素 x 放入栈顶
`pop()`	删除栈顶元素
`top()`	查看栈顶元素
`empty()`	判断栈是否为空
`size()`	查看栈中元素个数

例如依次执行：

$$\text{push}(1),\ \text{push}(2),\ \text{push}(3)$$

栈底到栈顶依次为：

$$1,\ 2,\ 3.$$

此时 `top()` 得到 3。执行一次 `pop()` 后，3 被删除，栈顶变成 2。

栈的核心是：先入栈的元素会被后来入栈的元素压在下面，所以要等上面的元素都出栈后，它才能出栈。

例题 1：栈操作追踪

初始栈为空，依次执行：

$$\text{push}(1),\ \text{push}(2),\ \text{pop},\ \text{push}(3),\ \text{push}(4),\ \text{pop}.$$

问最后栈中从底到顶的元素是什么？

按步骤追踪：

步骤	操作	栈中元素（底到顶）
1	`push(1)`	1
2	`push(2)`	$1, 2$
3	`pop()`	1
4	`push(3)`	$1, 3$
5	`push(4)`	$1, 3, 4$
6	`pop()`	$1, 3$

最后栈中从底到顶为：

$$1,\ 3.$$

栈题追踪时，不需要画很复杂的图。写一列“底到顶”的序列，遇到入栈就在右侧加入，遇到出栈就在右侧删除即可。

队列

队列可以想成排队买票。新来的排在队尾，服务时从队头开始。

常见操作如下。

操作	含义
`push(x)`	把元素 x 放到队尾
`pop()`	删除队头元素
`front()`	查看队头元素
`back()`	查看队尾元素
`empty()`	判断队列是否为空
`size()`	查看队列中元素个数

例如依次执行：

$$\text{push}(1),\ \text{push}(2),\ \text{push}(3)$$

队头到队尾依次为：

$$1,\ 2,\ 3.$$

此时 `front()` 得到 1，`back()` 得到 3。执行一次 `pop()` 后，1 被删除，新的队头是 2。

例题 2：队列操作追踪

初始队列为空，依次执行：

$$\text{push}(1),\ \text{push}(2),\ \text{pop},\ \text{push}(3),\ \text{pop},\ \text{push}(4).$$

问最后队列中从队头到队尾的元素是什么？

追踪如下：

步骤	操作	队列元素（头到尾）
1	`push(1)`	1
2	`push(2)`	$1, 2$
3	`pop()`	2
4	`push(3)`	$2, 3$
5	`pop()`	3
6	`push(4)`	$3, 4$

最后队列从头到尾为：

$$3,\ 4.$$

队列题追踪时，左侧表示队头，右侧表示队尾。入队在右侧加，出队在左侧删。

栈和队列的区别

栈和队列最容易混淆的地方，在于 `push` 都表示加入，但取出位置不同。

数据结构	加入位置	取出位置	规则
栈	栈顶	栈顶	后进先出
队列	队尾	队头	先进先出

看下面两个程序。

```
stack<int> s;
s.push(1);
s.push(2);
s.push(3);
while (!s.empty()) {
    cout << s.top() << ' ';
    s.pop();
}
```

输出顺序为：

$$3,\ 2,\ 1.$$

因为最后入栈的 3 最先出来。

再看队列：

```
queue<int> q;
q.push(1);
q.push(2);
q.push(3);
while (!q.empty()) {
    cout << q.front() << ' ';
    q.pop();
}
```

输出顺序为：

$$1,\ 2,\ 3.$$

因为最先入队的 1 最先出来。

程序阅读题中，看到 `stack` 要盯 `top()`，看到 `queue` 要盯 `front()`。不要把二者的取出位置看反。

空结构不能取元素

无论栈还是队列，都不能在空结构上直接取栈顶、队头或删除元素。

例如：

```
stack<int> s;
s.pop();
```

这段代码试图从空栈中删除元素，程序行为不正确。

再如：

```
queue<int> q;
cout << q.front();
```

这里队列为空，不能访问队头。

第一轮程序阅读中，题目一般会让程序在合法输入下运行，但判断题可能会问“该程序是否对所有输入都正确”。这时要检查是否可能出现空栈 `pop()`、空队列 `front()` 等问题。

例题 3：括号匹配中的空栈

```
bool check(string s) {
    stack<char> st;
    for (int i = 0; i < s.size(); i++) {
        if (s[i] == '(') st.push(s[i]);
        else {
            if (st.empty()) return false;
            st.pop();
        }
    }
    return st.empty();
}
```

这段程序判断只含左右括号的字符串是否匹配。

遇到左括号就入栈；遇到右括号时，需要有一个左括号与之匹配，所以先判断栈是否为空。若为空，说明右括号太多或出现得太早，返回 `false`。

循环结束后，还要判断栈是否为空。若栈中还剩左括号，说明左括号太多，也不匹配。

例如字符串 (()())：

字符	操作	栈中左括号个数
(	入栈	1
(	入栈	2
)	出栈	1
(	入栈	2
)	出栈	1
)	出栈	0

最后栈为空，匹配成功。

若字符串为 ())(，扫描到第三个字符时，栈已经为空却遇到右括号，应返回 `false`。

括号匹配

括号匹配是栈的典型应用。若有多种括号，例如 `()`、`[]`、`{}`，栈中要保存左括号的种类。

```
bool match(char l, char r) {
    if (l == '(' && r == ')') return true;
    if (l == '[' && r == ']') return true;
    if (l == '{' && r == '}') return true;
    return false;
}

bool check(string s) {
    stack<char> st;
    for (int i = 0; i < s.size(); i++) {
        char c = s[i];
        if (c == '(' || c == '[' || c == '{') {
            st.push(c);
        } else {
            if (st.empty()) return false;
            if (!match(st.top(), c)) return false;
            st.pop();
        }
    }
    return st.empty();
}
```

这里不能只统计左括号和右括号数量。数量相等不代表匹配。

例如：

(])

左括号和右括号数量都对，但顺序不对。扫描到 `)` 时，栈顶是 `[`，不能和 `)` 匹配，所以失败。

例题 4：判断括号串

判断字符串 ([]) 是否匹配。

逐步追踪：

字符	操作	栈中内容（底到顶）
(	入栈	(
[	入栈	(, [
{	入栈	(, [, {
}	与 { 匹配，出栈	(, [
]	与 [匹配，出栈	(
)	与 (匹配，出栈	空

最后栈为空，匹配成功。

判断字符串 (]) 时，扫描到 `)`，栈顶是 `[`，不匹配，所以失败。

括号匹配题的关键是“最近的未匹配左括号”必须和当前右括号匹配，这正是栈的后进先出规则。

栈序列

栈序列题常问：给定入栈顺序，某个出栈顺序是否可能？

假设入栈顺序固定为：

$$1, 2, 3, \dots, n.$$

每个元素必须先入栈，再出栈。入栈和出栈操作可以交替进行。问目标序列是否可能成为出栈序列。

判断方法是模拟。

从 1 到 n 依次入栈。每次入栈后，只要栈顶等于目标出栈序列当前需要的元素，就立刻出栈。最后如果目标序列全部匹配，则合法；否则不合法。

例题 5：合法出栈序列

入栈顺序为 $1, 2, 3, 4, 5$。判断出栈序列 $2, 1, 4, 5, 3$ 是否可能。

模拟：

操作	栈中内容（底到顶）	已输出
入栈 1	1	空
入栈 2	1, 2	空
出栈 2	1	2
出栈 1	空	2, 1
入栈 3	3	2, 1
入栈 4	3, 4	2, 1
出栈 4	3	2, 1, 4
入栈 5	3, 5	2, 1, 4
出栈 5	3	2, 1, 4, 5
出栈 3	空	2, 1, 4, 5, 3

目标序列可以得到，所以它是合法出栈序列。

例题 6：非法出栈序列

入栈顺序为 $1, 2, 3, 4, 5$。判断出栈序列 $3, 1, 2, 5, 4$ 是否可能。

要先输出 3，必须依次入栈 $1, 2, 3$，然后弹出 3。此时栈中从底到顶为：

$$1,\ 2.$$

下一个目标是 1，但栈顶是 2。由于 2 压在 1 上面，若想取出 1，必须先弹出 2。可是目标序列要求先输出 1，所以不可能。

因此 $3, 1, 2, 5, 4$ 不是合法出栈序列。

栈序列题中，只看数字大小不够。要看某个元素是否被后来的元素压住。

单调栈的影子

单调栈会在一些程序阅读题中出现。它不是本章的主线，但需要先能读懂基本结构。

下面程序维护一个从栈底到栈顶单调递增的栈：

```
for (int i = 1; i <= n; i++) {
    while (!st.empty() && st.top() > a[i]) {
        st.pop();
    }
    st.push(a[i]);
}
```

每来一个新元素 a[i]，如果栈顶比它大，就弹出栈顶，直到栈空或栈顶不大于它，再把 a[i] 入栈。

这类程序的特点是：每个元素最多入栈一次、出栈一次，所以整体复杂度通常是 $O(n)$，不是 $O(n^2)$。

例题 7：单调栈追踪

设数组为：

$$3,\ 1,\ 4,\ 2.$$

执行上面的程序。

i	$a[i]$	弹出情况	栈中内容（底到顶）
1	3	无	3
2	1	弹出 3	1
3	4	无	1, 4
4	2	弹出 4	1, 2

最终栈中为：

$$1,\ 2.$$

第一轮遇到这类代码时，重点看 while 条件和弹栈规则，不要把它当作普通栈随便追。

队列模拟

队列常用于模拟按先后顺序处理的过程。

例如有一批任务，按到达顺序进入队列，每次处理队头任务：

```
queue<int> q;

for (int i = 1; i <= n; i++) {
    q.push(i);
}

while (!q.empty()) {
    int x = q.front();
    q.pop();
    cout << x << ' ';
}
```

输出为：

$$1,\ 2,\ 3, \ldots, n.$$

因为队列保持先进先出。

例题 8：循环报数

有 5 个人编号 1 到 5，按顺序排成队列。每次取出队头的人，若当前报数不是 3，则把他放回队尾；若当前报数是 3，则他出局。报数从 1 开始循环。问出局顺序。

这是队列模拟的典型过程。追踪如下：

当前队列（头到尾）	报数	操作	出局
$1,2,3,4,5$	1	1 到队尾	—
$2,3,4,5,1$	2	2 到队尾	—
$3,4,5,1,2$	3	3 出局	3
$4,5,1,2$	1	4 到队尾	—
$5,1,2,4$	2	5 到队尾	—
$1,2,4,5$	3	1 出局	1
$2,4,5$	1	2 到队尾	—
$4,5,2$	2	4 到队尾	—
$5,2,4$	3	5 出局	5
$2,4$	1	2 到队尾	—
$4,2$	2	4 到队尾	—
$2,4$	3	2 出局	2
4	1	最后一人	4

出局顺序为：

$$3,\ 1,\ 5,\ 2,\ 4.$$

这种题不要只凭感觉跳步。队列从头到尾写清楚，操作就不容易错。

循环队列

用数组实现队列时，常用两个指针 `head` 和 `tail`。一种常见写法是：

```
int q[N], head = 1, tail = 0;

void push(int x) {
    q[++tail] = x;
}

void pop() {
    head++;
}

int front() {
    return q[head];
}

bool empty() {
    return head > tail;
}
```

这里队列中的元素位于：

$$q[head], q[head+1], \dots, q[tail].$$

当 `head > tail` 时，队列为空。

如果数组空间固定，又要反复入队出队，可以让下标循环使用，这就是循环队列。第一轮中常见的是判断 `head`、`tail` 的变化，不要求写复杂实现。

一个常见循环队列写法如下：

```
int q[N], head = 0, tail = 0;

void push(int x) {
    q[tail] = x;
    tail = (tail + 1) % N;
}

void pop() {
    head = (head + 1) % N;
}

bool empty() {
    return head == tail;
}
```

这段代码中，`tail` 指向下一个可插入位置，`head` 指向当前队头位置。为了区分空和满，真实程序通常还需要额外处理队列满的情况，或空出一个位置不用。

第一轮读循环队列时，重点看：

- `head` 指向队头还是队头前一位；
- `tail` 指向队尾还是下一个插入位置；
- 空队列条件是什么；
- 下标是否取模回绕。

例题 9：数组队列追踪

```
int q[10], head = 1, tail = 0;

q[++tail] = 4;
q[++tail] = 7;
head++;
q[++tail] = 9;

cout << q[head] << ' ' << q[tail] << endl;
```

追踪：

操作	head	tail	队列内容
初始	1	0	空
入队 4	1	1	4
入队 7	1	2	4, 7
出队	2	2	7
入队 9	2	3	7, 9

所以：

$$q[head] = q[2] = 7, \quad q[tail] = q[3] = 9.$$

输出：

```
7 9
```

数组队列题中，`head` 和 `tail` 的含义必须先确定。不同写法下，`tail` 可能表示队尾，也可能表示下一个空位。

双端队列

双端队列可以在两端插入和删除。C++ 中常用 `deque` 表示。

常见操作如下。

操作	含义
`push_front(x)`	从队头插入
`push_back(x)`	从队尾插入
`pop_front()`	删除队头
`pop_back()`	删除队尾
`front()`	查看队头
`back()`	查看队尾

双端队列兼有栈和队列的一些特点。若只在一端进出，它像栈；若从一端进、另一端出，它像队列。

例题 10：双端队列操作

初始双端队列为空，依次执行：

$$\text{push_back}(1),\ \text{push_back}(2),\ \text{push_front}(3),\ \text{pop_back},\ \text{push_front}(4).$$

追踪：

步骤	操作	内容（头到尾）
1	`push_back(1)`	1
2	`push_back(2)`	$1, 2$
3	`push_front(3)`	$3, 1, 2$
4	`pop_back()`	$3, 1$
5	`push_front(4)`	$4, 3, 1$

最后从头到尾为：

$$4,\ 3,\ 1.$$

双端队列题要看清操作端点。`front` 和 `back` 一旦看反，后面全会错。

STL 与手写结构

C++ 中可以直接使用 `stack`、`queue`、`deque`。第一轮程序阅读常出现这些容器。

```
#include <stack>
#include <queue>
#include <deque>
```

常见区别如下。

容器	主要访问位置	常见用途
stack	top()	后进先出
queue	front()、back()	先进先出
deque	front()、back()	两端操作

要注意：stack 没有 front()；queue 没有 top()。程序阅读题中如果给出标准容器代码，要按容器接口判断。

不过 CSP-S 第一轮也可能给出手写数组结构。此时不要强行套 STL 名称，要看 head、tail、top 等变量实际如何变化。

例题 11：手写栈

```
int st[100], top = 0;

st[++top] = 5;
st[++top] = 8;
top--;
st[++top] = 3;

cout << st[top] << endl;
```

追踪：

操作	top	栈中内容
初始	0	空
入栈 5	1	5
入栈 8	2	5, 8
出栈	1	5
入栈 3	2	5, 3

最后 st[top]=3，输出 3。

手写栈中，top 通常表示栈顶位置。++top 后赋值表示先移动栈顶，再放入元素。

程序阅读中的栈与队列

读栈与队列程序，可以按下面顺序。

第一，判断结构类型。是栈、队列、双端队列，还是手写数组结构。

第二，确定两端含义。栈看栈顶；队列看队头和队尾；双端队列看两端。

第三，追踪操作。入栈、出栈、入队、出队要逐步记录。

第四，检查空结构。看是否存在空栈取顶、空队列取头。

第五，判断程序目的。括号匹配、栈序列、队列模拟、单调结构，追踪重点不同。

例题 12：混合操作程序

```
stack<int> s;
queue<int> q;

for (int i = 1; i <= 3; i++) {
    s.push(i);
    q.push(i);
}
```

```
cout << s.top() << ' ' << q.front() << endl;
```

栈中从底到顶为：

$$1,\ 2,\ 3.$$

所以 `s.top()` 是 3。

队列中从头到尾为：

$$1,\ 2,\ 3.$$

所以 `q.front()` 是 1。

输出：

```
3 1
```

同样的插入顺序，栈和队列取出的元素不同。

完善程序中的栈与队列

完善程序题中，栈与队列常见空缺包括：

- 括号匹配中的空栈判断；
- 左括号入栈；
- 右括号与栈顶匹配；
- BFS 或队列模拟中的入队、出队；
- 手写队列中的 `head`、`tail` 更新；
- 栈序列判断中的循环条件；
- 单调栈中的弹栈条件。

本章不讲 BFS 图搜索细节，那会在后面搜索和图论章节处理。这里先看栈与队列本身的空缺。

例题 13：补全括号匹配

```
bool check(string s) {
    stack<char> st;
    for (int i = 0; i < s.size(); i++) {
        if (s[i] == '(') {
            _____;
        } else {
            if (st.empty()) return false;
            _____;
        }
    }
    return st.empty();
}
```

只含左右括号。遇到左括号应入栈，遇到右括号应弹出一个左括号。

所以空缺应填：

```
st.push(s[i])
st.pop()
```

最后 `return st.empty()` 用来确认没有多余左括号。

例题 14：补全队列出队

```
queue<int> q;
q.push(1);
q.push(2);
q.push(3);

while (!q.empty()) {
    int x = _____;
    q.pop();
    cout << x << ' ';
}
```

队列查看队头使用 `front()`，所以空缺应填：

```
q.front()
```

不能填 `q.back()`。`back()` 是队尾元素，如果每次输出队尾再删除队头，输出和删除的不是同一个元素。

例题 15：补全手写队列

```
int q[N], head = 1, tail = 0;

void push(int x) {
    q[_____] = x;
}

void pop() {
    _____;
}
```

若采用 `head=1, tail=0` 的写法，入队时应先增加 `tail` 再赋值：

```
++tail
```

出队时让 `head` 后移：

```
head++
```

完整片段为：

```
void push(int x) {
    q[++tail] = x;
}

void pop() {
    head++;
}
```

这种写法下，队列为空的条件是 `head > tail`。

常见错误

栈与队列题的错误集中在下面几类。

错误	说明
栈队列规则混淆	栈后进先出，队列先进先出
top 和 front 混用	栈看 top，队列看 front
空结构取元素	没有先判断 empty()
括号只数数量	数量相等不代表顺序匹配
栈序列只看大小	关键是是否被后入栈元素压住
队列端点看反	入队在尾，出队在头
手写队列含义不清	tail 是队尾还是下一个空位要看代码
循环队列忘记取模	下标到末尾后要回绕
双端队列操作端点看错	front 与 back 方向不同
单调栈复杂度误判	每个元素最多入栈出栈一次

做栈与队列题时，最稳的方法是画状态表。状态表不需要大，只要记录当前结构中元素的顺序和关键指针，就能避免多数错误。

原创例题

例题 16：栈与队列输出

阅读程序：

```
stack<int> s;
queue<int> q;

for (int i = 1; i <= 4; i++) {
    s.push(i);
    q.push(i);
}

s.pop();
q.pop();

s.push(5);
q.push(5);

cout << s.top() << ' ' << q.front() << endl;
```

栈的变化：

操作	栈中内容（底到顶）
入栈 $1,2,3,4$	$1,2,3,4$
出栈	$1,2,3$
入栈 5	$1,2,3,5$

所以 s.top() 为 5。

队列的变化：

操作	队列内容（头到尾）
入队 $1,2,3,4$	$1,2,3,4$
出队	$2,3,4$
入队 5	$2,3,4,5$

所以 `q.front()` 为 2。

输出：

```
5 2
```

变式 1：输出队尾

如果最后一行改成：

```
cout << s.top() << ' ' << q.back() << endl;
```

队列队尾为 5，所以输出：

```
5 5
```

`front()` 和 `back()` 差别很大，程序阅读题中要逐字看清。

变式 2：改用双端队列

```
deque<int> d;
d.push_back(1);
d.push_back(2);
d.push_front(3);
d.pop_back();
cout << d.front() << ' ' << d.back() << endl;
```

追踪：

操作	内容（头到尾）
`push_back(1)`	1
`push_back(2)`	$1,2$
`push_front(3)`	$3,1,2$
`pop_back()`	$3,1$

所以输出：

```
3 1
```

小结

栈与队列的基本规则很简单，但程序阅读中的变化很多。只要坚持按操作顺序追踪，不把栈顶和队头混淆，遇到括号匹配、栈序列和队列模拟时就能稳住。下一章进入堆，它同样支持插入和删除，但取出的不再是最早或最后进入的元素，而是当前优先级最高的元素。

0x12 堆

堆是一种支持 “快速取出当前最大值或最小值” 的数据结构。它常被用来实现优先队列。

第 17 章讲的普通队列遵守先进先出：先进入队列的元素先被取出。堆不按进入时间取元素，而是按优先级取元素。如果用大根堆，每次取出当前最大值；如果用小根堆，每次取出当前最小值。

CSP-S 第一轮中，堆常以三种方式出现：选择题中问堆的性质、插入删除后的状态；程序阅读中给出 `priority_queue` 或手写堆代码，要求判断输出；完善程序中要求补出上浮、下沉或比较条件。本章只讲第一轮需要掌握的堆，不展开成复杂数据结构专题。

堆是什么

堆通常是一棵完全二叉树，并且满足堆序性质。

大根堆满足：

每个结点的值都不小于它的子结点的值。

所以大根堆的根结点是整个堆中的最大值。

小根堆满足：

每个结点的值都不大于它的子结点的值。

所以小根堆的根结点是整个堆中的最小值。

注意，堆不要求整棵树从左到右有序。它只要求父子之间满足大小关系。

例如下面是一棵大根堆：

```
      9
    /   \
   7     8
  / \   /
 3   5 2
```

根结点 9 是最大值。7 大于它的子结点 3,5，8 大于它的子结点 2。但是同一层中，7 和 8 的左右顺序没有固定要求。

例题 1：判断是否为大根堆

判断下面这棵树是否为大根堆。

```
      10
    /    \
   6      9
  / \    /
 4   7  3
```

检查父子关系：

- $10 \geq 6$，$10 \geq 9$；
- $6 \geq 4$，但 $6 < 7$；
- $9 \geq 3$。

因为结点 6 小于它的子结点 7，堆序性质被破坏，所以它不是大根堆。

判断堆时，不需要比较所有结点对，只要检查每条父子边。

数组表示

堆通常用数组存储。若数组下标从 1 开始，则结点 i 的位置关系很简单：

位置	下标
父结点	$\lfloor i/2 \rfloor$
左儿子	$2i$
右儿子	$2i+1$

如果堆中有 n 个元素，只有下标不超过 n 的结点存在。

例如数组：

$$h[1..6] = 9,\ 7,\ 8,\ 3,\ 5,\ 2$$

对应的树为：

```
       h[1]=9
      /      \
   h[2]=7    h[3]=8
    /   \      /
 h[4]=3 h[5]=5 h[6]=2
```

这正是一棵大根堆。

若使用从 0 开始的数组，下标关系会变成：

位置	下标
父结点	$\lfloor (i-1)/2 \rfloor$
左儿子	$2i+1$
右儿子	$2i+2$

CSP-S 第一轮中，两种写法都可能出现。读代码时先看数组从 0 还是从 1 开始，不要把下标公式混用。

例题 2：数组是否表示大根堆

给定数组：

$$h[1..7] = 20,\ 15,\ 18,\ 8,\ 10,\ 12,\ 9.$$

判断它是否表示大根堆。

检查非叶子结点即可。对于 $n = 7$，非叶子结点为 $1, 2, 3$。

- $h[1] = 20$，子结点为 $h[2] = 15, h[3] = 18$，满足；

- $h[2]=15$，子结点为 $h[4]=8, h[5]=10$，满足；
- $h[3]=18$，子结点为 $h[6]=12, h[7]=9$，满足。

所以这是一个大根堆。

不需要检查 $h[4], h[5], h[6], h[7]$ 的子结点，因为它们没有儿子。

插入：上浮

向堆中插入一个新元素时，先把它放在数组末尾，也就是完全二叉树的下一个位置。然后不断与父结点比较，若破坏堆序，就交换它和父结点。这个过程叫上浮。

以大根堆为例，若新元素比父结点大，就需要向上交换。

例题 3：大根堆插入

已有大根堆数组：

$$h[1..6] = 9,\ 7,\ 8,\ 3,\ 5,\ 2.$$

插入 10。

第一步，把 10 放到末尾：

$$9,\ 7,\ 8,\ 3,\ 5,\ 2,\ 10.$$

新元素下标为 7，父结点下标为 3，值为 8。因为 $10 > 8$，交换：

$$9,\ 7,\ 10,\ 3,\ 5,\ 2,\ 8.$$

此时 10 的下标为 3，父结点下标为 1，值为 9。因为 $10 > 9$，继续交换：

$$10,\ 7,\ 9,\ 3,\ 5,\ 2,\ 8.$$

10 已到根结点，插入结束。

最终堆为：

$$10,\ 7,\ 9,\ 3,\ 5,\ 2,\ 8.$$

插入操作的关键是：新元素只沿着从末尾到根的一条路径移动，不会影响整棵树的所有结点。

手写上浮代码

大根堆插入可以写成：

```
void push(int x) {
    h[++siz] = x;
    int p = siz;
    while (p > 1 && h[p] > h[p / 2]) {
        swap(h[p], h[p / 2]);
        p /= 2;
    }
```

```
}
```

这里 `siz` 表示堆中元素个数。`p` 是当前新元素所在位置。

完善程序题中，常见空缺是：

- `h[++siz] = x;`
- `p > 1;`
- `h[p] > h[p / 2];`
- `p /= 2`。

若是小根堆，比较条件要改成 `h[p] < h[p / 2]`。

删除堆顶：下沉

堆最常见的删除操作是删除堆顶。对于大根堆，堆顶是最大值；对于小根堆，堆顶是最小值。

删除堆顶时，不能直接把根结点空出来。常见做法是：

1. 用最后一个元素覆盖根结点；
2. 堆大小减一；
3. 从根结点开始下沉，恢复堆序。

以大根堆为例，下沉时要把当前结点和较大的子结点比较。如果当前结点小于这个子结点，就交换；否则停止。

例题 4：大根堆删除堆顶

已有大根堆：

$$h[1..7] = 10,\ 7,\ 9,\ 3,\ 5,\ 2,\ 8.$$

删除堆顶 10。

第一步，用最后一个元素 8 覆盖根结点，并删除最后位置：

$$8,\ 7,\ 9,\ 3,\ 5,\ 2.$$

现在根结点 8 的两个儿子为 7 和 9。较大的儿子是 9，因为 $8 < 9$，交换：

$$9,\ 7,\ 8,\ 3,\ 5,\ 2.$$

此时 8 在下标 3，它的左儿子为下标 6，值为 2，没有右儿子。因为 $8 \geq 2$，停止。

删除后的堆为：

$$9,\ 7,\ 8,\ 3,\ 5,\ 2.$$

手写下沉代码

大根堆删除堆顶可以写成：

```
void pop() {
    h[1] = h[siz--];
    int p = 1;
    while (p * 2 <= siz) {
        int c = p * 2;
        if (c + 1 <= siz && h[c + 1] > h[c]) c++;
        if (h[p] >= h[c]) break;
        swap(h[p], h[c]);
        p = c;
    }
}
```

这里 c 表示当前要比较的子结点。对于大根堆，要选择两个儿子中较大的那个：

```
if (c + 1 <= siz && h[c + 1] > h[c]) c++;
```

若当前结点已经不小于较大儿子，就说明堆序恢复：

```
if (h[p] >= h[c]) break;
```

小根堆的比较方向全部相反：选择较小儿子，并在当前结点不大于较小儿子时停止。

建堆

给定一组元素，可以逐个插入建堆，也可以从最后一个非叶子结点开始下沉建堆。

逐个插入比较直观：

```
siz = 0;
for (int i = 1; i <= n; i++) {
    push(a[i]);
}
```

每次插入代价是 $O(\log n)$，总复杂度是 $O(n \log n)$。

从下往上建堆写法如下：

```
for (int i = n / 2; i >= 1; i--) {
    down(i);
}
```

其中 `down(i)` 表示从位置 i 开始下沉。因为下标大于 $n/2$ 的结点都是叶子，不需要下沉，所以从 $n/2$ 开始。

第一轮中，如果只考纸面过程，通常会让学生追踪少量元素的插入、删除；如果考复杂度，要知道这种从下往上建堆可以在线性时间内完成。若暂时不熟悉证明，可以先记住这个结论；程序阅读题更多关注 `down` 的方向和比较条件。

例题 5：最后一个非叶子结点

堆有 $n = 10$ 个元素，数组下标从 1 开始。最后一个非叶子结点下标是多少？

最后一个非叶子结点是 $\lfloor n/2 \rfloor$，所以为：

$$\lfloor 10/2 \rfloor = 5.$$

下标 6 到 10 都是叶子结点。

若 $n=9$，最后一个非叶子结点为：

$$\lfloor 9/2 \rfloor = 4.$$

因为下标 4 的左儿子是 8，右儿子是 9；下标 5 的左儿子是 10，已经不存在。

堆的复杂度

堆的高度是 $O(\log n)$。因为堆是一棵完全二叉树，元素数量每增加一层大约翻倍。

常见操作复杂度如下。

操作	复杂度	原因
查看堆顶	$O(1)$	堆顶在根结点
插入元素	$O(\log n)$	最多上浮一条根到叶的路径
删除堆顶	$O(\log n)$	最多下沉一条根到叶的路径
逐个插入建堆	$O(n \log n)$	插入 n 次
从下往上建堆	$O(n)$	许多结点高度较小

程序阅读题中，若代码对每个元素执行一次 `push`，复杂度通常是 $O(n \log n)$。若代码先建堆，再执行 k 次删除堆顶，总复杂度常写成 $O(n + k \log n)$，具体还要看建堆方式。

priority_queue

C++ 中的 `priority_queue` 是优先队列，通常用堆实现。默认情况下，它是大根堆，也就是 `top()` 返回当前最大元素。

基本用法：

```
priority_queue<int> q;

q.push(3);
q.push(1);
q.push(5);

cout << q.top() << endl;
```

输出为：

```
5
```

因为默认优先队列取最大值。

删除堆顶：

```
q.pop();
```

再次 `top()` 得到 3。

例题 6：priority_queue 输出

阅读程序：

```
priority_queue<int> q;

q.push(4);
q.push(1);
q.push(7);
q.push(3);

while (!q.empty()) {
    cout << q.top() << ' ';
    q.pop();
}
```

默认是大根堆，所以每次输出当前最大值。

输出为：

```
7 4 3 1
```

这段程序会把元素按从大到小的顺序输出。

小根堆写法

若要让 `priority_queue` 每次取出最小值，可以写：

```
priority_queue<int, vector<int>, greater<int> > q;
```

例如：

```
priority_queue<int, vector<int>, greater<int> > q;

q.push(4);
q.push(1);
q.push(7);
q.push(3);

while (!q.empty()) {
    cout << q.top() << ' ';
    q.pop();
}
```

输出为：

```
1 3 4 7
```

这里 `greater<int>` 使较小的元素优先出现在堆顶。程序阅读题中，看到 `greater<int>`，要想到它通常是小根堆。

自定义比较

堆中的元素不一定是整数，也可以是结构体。此时需要定义比较方式。

例如有若干任务，每个任务有编号 `id` 和优先级 `p`，要求优先级大的先处理：

```
struct Node {
    int id, p;
};

struct Cmp {
```

```
    bool operator()(Node a, Node b) {
        return a.p < b.p;
    }
};

priority_queue<Node, vector<Node>, Cmp> q;
```

这段比较函数的含义容易看错。对于 `priority_queue`，可以先不深究比较器的内部规则，而用小例子判断：

若 `a.p < b.p` 时返回 `true`，则优先级更大的结点会排在堆顶。因此这是按 `p` 从大到小取出。

若要优先级小的先处理，可以写：

```
struct Cmp {
    bool operator()(Node a, Node b) {
        return a.p > b.p;
    }
};
```

第一轮中，自定义比较常出现在程序阅读题里。稳妥方法是取两个元素代入比较函数，看看谁会更靠前。

例题 7：结构体优先队列

```
struct Node {
    int id, p;
};

struct Cmp {
    bool operator()(Node a, Node b) {
        return a.p < b.p;
    }
};

priority_queue<Node, vector<Node>, Cmp> q;

q.push({1, 30});
q.push({2, 50});
q.push({3, 40});

cout << q.top().id << endl;
```

三个任务的优先级分别为 $30, 50, 40$。比较器让优先级大的排在前面，所以堆顶是 `id=2` 的任务。

输出：

```
2
```

若比较函数改成 `return a.p > b.p;`，堆顶会变成优先级最小的任务，即 `id=1`。

堆排序

堆可以用来排序。以大根堆为例：

1. 把所有元素放入大根堆；
2. 每次取出堆顶最大值；
3. 不断删除堆顶，得到从大到小的序列。

如果想得到从小到大排序，可以使用小根堆逐个取出，也可以在数组中原地堆排序。第一轮中，更多是判断取出顺序和复杂度。

例题 8：堆排序输出

把 $5, 1, 4, 2$ 依次插入大根堆，然后不断取出堆顶并输出。

大根堆每次取当前最大值，所以输出为：

$$5,\ 4,\ 2,\ 1.$$

若使用小根堆，则输出为：

$$1,\ 2,\ 4,\ 5.$$

这类题不一定要画出每次堆的形态。若只问完整输出顺序，知道每次取当前最大或最小即可。

复杂度

堆排序的复杂度通常为 $O(n \log n)$。因为每次删除堆顶代价为 $O(\log n)$，删除 n 次总共 $O(n \log n)$。如果从下往上建堆，建堆部分为 $O(n)$，但后续删除仍是主要部分。

堆排序通常不稳定。因为堆调整中可能发生较远距离交换，相等关键字元素的相对顺序可能改变。

Top k 问题

堆常用于处理 “前 k 大” 或 “前 k 小” 问题。

若要在 n 个数中找最大的 k 个数，一种方法是维护一个大小为 k 的小根堆。堆中保存当前见过的最大 k 个数，堆顶是这 k 个数中最小的那个。

处理新数 x 时：

- 若堆大小不足 k，直接插入；
- 若 x 大于堆顶，说明 x 应进入前 k 大，弹出堆顶，再插入 x；
- 否则忽略 x。

这样每次操作最多 $O(\log k)$，总复杂度 $O(n \log k)$。

例题 9：维护前三大

依次读入：

$$5,\ 1,\ 9,\ 3,\ 7,\ 6.$$

要求维护前三大的数。使用大小为 3 的小根堆。

读入	操作后堆中元素	说明
5	5	不足 3 个，插入
1	1, 5	不足 3 个，插入
9	1, 5, 9	不足 3 个，插入
3	3, 5, 9	$3 > 1$，替换堆顶
7	5, 7, 9	$7 > 3$，替换堆顶
6	6, 7, 9	$6 > 5$，替换堆顶

最终前三大为：

$$6,\ 7,\ 9.$$

注意堆中不一定按从小到大完整排列。小根堆只保证堆顶是最小的那个。若要输出有序结果，还需要继续取堆顶或排序。

合并果子模型

堆的另一个常见模型是：每次取出当前最小的两个数，合并后再放回。这个过程常用小根堆实现。

例题 10：合并代价

有 4 堆果子，数量分别为：

$$1,\ 2,\ 5,\ 9.$$

每次合并两堆，代价为两堆数量之和，合并后的新堆再参与后续合并。若每次都合并当前最小的两堆，总代价是多少？

使用小根堆追踪：

步骤	取出	新堆	本次代价	当前堆
1	1, 2	3	3	3, 5, 9
2	3, 5	8	8	8, 9
3	8, 9	17	17	17

总代价为：

$$3 + 8 + 17 = 28.$$

这类题中，不能随便选两堆合并。每一步都要取当前最小的两堆。程序阅读中若看到小根堆反复 `pop` 两次、再 `push` 和，通常就是这种流程。

程序阅读中的堆

读堆相关程序，可以按下面顺序。

第一，判断是大根堆还是小根堆。`priority_queue<int>` 默认大根堆；带 `greater<int>` 通常是小根堆；手写堆要看比较条件。

第二，确定 `top()` 的含义。它是当前最大值、最小值，还是按自定义优先级排在最前的元素。

第三，追踪 push 和 pop。若只问输出顺序，通常不必画完整堆，只要知道每次取出当前优先级最高者。

第四，若是手写堆，要看上浮、下沉的比较方向。

第五，判断复杂度。每次插入、删除堆顶通常是 $O(\log n)$。

例题 11：程序输出

```
priority_queue<int, vector<int>, greater<int> > q;

q.push(6);
q.push(2);
q.push(8);
q.push(4);

cout << q.top() << ' ';
q.pop();
q.push(1);
cout << q.top() << ' ';
q.pop();
cout << q.top() << endl;
```

这是小根堆。

初始插入 $6, 2, 8, 4$ 后，堆顶为 2。输出 2，弹出 2。

插入 1 后，堆中有 $1, 4, 6, 8$，堆顶为 1。输出 1，弹出 1。

此时堆顶为 4。输出 4。

完整输出为：

```
2 1 4
```

例题 12：手写堆判断

```
void push(int x) {
    h[++siz] = x;
    int p = siz;
    while (p > 1 && h[p] < h[p / 2]) {
        swap(h[p], h[p / 2]);
        p /= 2;
    }
}
```

这里新元素若小于父结点，就向上交换。因此这是小根堆插入。插入后，堆顶 `h[1]` 是当前最小值。

若比较符号改成 `>`，就是大根堆插入。

完善程序中的堆

堆相关完善程序常见空缺包括：

- 上浮中的父结点下标；
- 上浮比较条件；
- 下沉中的左儿子、右儿子下标；
- 选择较大儿子或较小儿子；
- 删除堆顶时用最后元素覆盖根；

- `priority_queue` 的比较器；
- 小根堆模板中的 `greater<int>`。

例题 13：补全大根堆插入

```
void push(int x) {
    h[++siz] = x;
    int p = siz;
    while (p > 1 && _____) {
        swap(h[p], h[p / 2]);
        p = _____;
    }
}
```

大根堆中，若当前结点大于父结点，就需要上浮。所以第一空为：

```
h[p] > h[p / 2]
```

交换后当前位置变成父结点位置，所以第二空为：

```
p / 2
```

完整关键部分为：

```
while (p > 1 && h[p] > h[p / 2]) {
    swap(h[p], h[p / 2]);
    p = p / 2;
}
```

例题 14：补全大根堆下沉

```
void down(int p) {
    while (p * 2 <= siz) {
        int c = p * 2;
        if (c + 1 <= siz && _____) c++;
        if (_____) break;
        swap(h[p], h[c]);
        p = c;
    }
}
```

大根堆下沉时，要选择两个儿子中较大的那个。若右儿子存在且右儿子更大，应令 `c++`，所以第一空为：

```
h[c + 1] > h[c]
```

若当前结点已经不小于较大的儿子，就不需要交换，所以第二空为：

```
h[p] >= h[c]
```

完整关键部分为：

```
if (c + 1 <= siz && h[c + 1] > h[c]) c++;
if (h[p] >= h[c]) break;
```

例题 15：补全小根堆优先队列

```
priority_queue<int, vector<int>, _____ > q;
```

若要求每次取出最小值，应填：

```
greater<int>
```

完整写法为：

```
priority_queue<int, vector<int>, greater<int> > q;
```

注意模板右侧的两个 > 之间保留空格，在 C++14 中这样写更稳妥。

常见错误

堆相关题目的错误集中在下面几类。

错误	说明
把堆当成有序数组	堆只保证父子关系，不保证整体有序
大根堆小根堆混淆	`priority_queue<int>` 默认取最大值
`greater<int>` 看反	常用于小根堆
删除堆顶步骤错	应用最后元素覆盖根，再下沉
下沉时选错儿子	大根堆选较大儿子，小根堆选较小儿子
上浮比较方向错	大根堆用 >，小根堆用 <
数组下标公式混用	从 1 开始和从 0 开始公式不同
以为 `top()` 删除元素	`top()` 只查看，`pop()` 才删除
Top k 堆类型选错	求前 k 大常维护小根堆
复杂度误判	插入、删除堆顶通常是 $O(\log n)$

做堆题时，先写清楚“堆顶代表最大还是最小”。这一步确认后，再追踪 `push`、`pop`、`top`，就不容易乱。

原创例题

例题 16：优先队列输出

阅读程序：

```
priority_queue<int> q;

int a[6] = {0, 4, 9, 1, 7, 3};

for (int i = 1; i <= 5; i++) {
    q.push(a[i]);
}

q.pop();
q.push(6);

while (!q.empty()) {
    cout << q.top() << ' ';
    q.pop();
}
```

`priority_queue<int>` 默认是大根堆。

插入 $4, 9, 1, 7, 3$ 后，堆顶是 9。执行 `q.pop()` 删除 9。

再插入 6，堆中元素为：

$$4,\ 1,\ 7,\ 3,\ 6.$$

不断输出并删除当前最大值，顺序为：

$$7,\ 6,\ 4,\ 3,\ 1.$$

因此输出：

```
7 6 4 3 1
```

变式 1：改成小根堆

若第一行改成：

```
priority_queue<int, vector<int>, greater<int> > q;
```

插入 $4, 9, 1, 7, 3$ 后，堆顶是 1。先删除 1，再插入 6。堆中元素为：

$$3,\ 4,\ 6,\ 7,\ 9.$$

不断取最小值，输出：

```
3 4 6 7 9
```

变式 2：只看不删

如果把循环中的 `q.pop()` 删除：

```
while (!q.empty()) {
    cout << q.top() << ' ';
}
```

那么队列状态不会改变，`q.empty()` 永远不会变为真，程序无法正常结束。

`top()` 只是查看堆顶，不会删除堆顶。要删除必须调用 `pop()`。

小结

堆的规则比栈和队列多一些，但核心仍然是“堆顶代表当前优先级最高的元素”。大根堆取最大，小根堆取最小。只要先判断堆顶含义，再按插入、删除过程追踪，堆相关程序阅读和完善程序就会清楚很多。下一章进入哈希表，重点会从“优先级”转向“如何快速定位和查找”。

0x13 哈希表

哈希表用于快速判断一个元素是否出现过，或快速找到某个键对应的值。它的目标很直接：给定一个键，尽量用接近 $O(1)$ 的时间完成查找、插入和删除。

第 17 章的栈与队列强调进出顺序，第 18 章的堆强调优先级。哈希表关注的不是顺序，也不是最大最小，而是“能不能快速定位”。在 CSP-S 第一轮中，哈希表常以选择题、程序阅读和完善程序形式出现：判断平均复杂度、处理冲突、读懂 `unordered_set` 和 `unordered_map` 的行为、统计出现次数、判断重复元素。

本章只讲哈希表的基础用法和纸面考法。字符串哈希会在第 36 章系统讲，本章只把字符串作为一种可以放入哈希表的键来看待。

哈希表解决什么问题

先看一个简单问题：给出 n 个整数，再给出 q 次询问，每次问某个整数是否在这 n 个数中出现过。

如果每次询问都从头扫描数组，一次查询是 $O(n)$，q 次查询是 $O(nq)$。

如果用哈希表，可以先把所有数插入表中，再对每个询问直接查找。平均意义下，每次查找接近 $O(1)$，总复杂度可以看作 $O(n+q)$。

例如：

```
unordered_set<int> s;

for (int i = 1; i <= n; i++) {
    int x;
    cin >> x;
    s.insert(x);
}

for (int i = 1; i <= q; i++) {
    int x;
    cin >> x;
    if (s.count(x)) cout << "YES\n";
    else cout << "NO\n";
}
```

这里 `unordered_set<int>` 存的是一批不重复的整数。`s.count(x)` 用来判断 x 是否在集合中。

哈希表适合解决这类问题：

- 判断某个值是否出现过；
- 统计某个值出现了多少次；
- 维护键到值的对应关系；
- 快速去重；
- 根据编号、字符串、状态查找对应信息。

它不适合直接回答“第几小”“前驱后继”“按大小顺序遍历”这类问题。因为哈希表本身不维护有序性。

哈希函数

哈希表的核心是哈希函数。哈希函数把一个键转换成一个数组下标或桶编号。

例如，我们想把整数放入大小为 m 的表中，可以使用：

$$h(x) = x \bmod m.$$

若 $m = 7$，则：

x	$h(x) = x \bmod 7$
10	3
17	3
25	4
31	3

这里 $10, 17, 31$ 都被映射到编号 3 的位置。不同的键映射到同一个位置，这就叫冲突。

哈希函数的目标是尽量把键分散到不同位置。但只要可能的键很多、表的大小有限，冲突就无法完全避免。

第一轮中，哈希函数常以简单取模形式出现。读题时要看清楚：

- 表大小是多少；
- 哈希函数怎样计算；
- 发生冲突时怎样处理；
- 最终查找过程是否和插入过程一致。

冲突

冲突是哈希表必须处理的问题。若两个不同元素被哈希函数映射到同一个位置，就要有办法同时保存它们。

例如表大小为 7，哈希函数为 $h(x) = x \bmod 7$。插入 $10, 17, 24$。

它们的哈希值都是 3：

$$10 \bmod 7 = 3, \quad 17 \bmod 7 = 3, \quad 24 \bmod 7 = 3.$$

如果表中每个位置只能放一个元素，就无法同时保存三者。因此需要冲突处理。

常见处理方法有两类：

- 拉链法；
- 开放寻址法。

CSP-S 第一轮不要求掌握很多工程细节，但要知道冲突存在，并能读懂基本处理过程。

拉链法

拉链法的做法是：每个桶后面挂一条链表或一个列表，所有哈希值相同的元素都放在同一个桶里。

例如表大小为 7，插入 $10, 17, 25, 31$。

哈希值为：

元素	哈希值
10	3
17	3
25	4
31	3

表中可以表示为：

桶编号	元素
0	空
1	空
2	空
3	$10, 17, 31$
4	25
5	空
6	空

查找 17 时，先计算 $17 \bmod 7 = 3$，只需要在桶 3 的列表中查找。

若哈希函数分布较好，每个桶里的元素很少，查找就很快。若很多元素都进入同一个桶，查找就会变慢，最坏可以退化成线性扫描。

例题 1：拉链法查找

表大小为 5，哈希函数为 $h(x) = x \bmod 5$。依次插入：

$$12,\ 7,\ 18,\ 20,\ 25,\ 31.$$

计算哈希值：

元素	哈希值
12	2
7	2
18	3
20	0
25	0
31	1

所以各桶为：

桶编号	元素
0	$20, 25$
1	31
2	$12, 7$
3	18
4	空

查找 25 时，先到桶 0，再在 $20, 25$ 中找到 25。

查找 22 时，$22 \bmod 5 = 2$，到桶 2，检查 $12, 7$，没有 22，所以查找失败。

开放寻址法

开放寻址法不把冲突元素挂到链表中，而是在数组中继续寻找下一个可用位置。

最简单的是线性探测。若位置 $h(x)$ 已被占用，就依次尝试：

$$h(x)+1,\ h(x)+2,\ h(x)+3,\ldots$$

必要时对表大小取模回到开头。

例如表大小为 7，哈希函数 $h(x)=x \bmod 7$，依次插入 $10, 17, 24$。

- 插入 10，$10 \bmod 7=3$，位置 3 空，放入位置 3；
- 插入 17，$17 \bmod 7=3$，位置 3 已占用，尝试位置 4，放入位置 4；
- 插入 24，$24 \bmod 7=3$，位置 $3,4$ 已占用，尝试位置 5，放入位置 5。

最终：

下标	内容
0	空
1	空
2	空
3	10
4	17
5	24
6	空

查找时必须使用相同探测规则。查找 24 时，从位置 3 开始，依次看 $10, 17, 24$，直到找到。

例题 2：线性探测插入

表大小为 10，哈希函数为 $h(x)=x \bmod 10$。依次插入：

$$12,\ 22,\ 32,\ 15.$$

插入过程：

元素	初始位置	插入位置	说明
12	2	2	位置 2 空
22	2	3	位置 2 被占，向后找
32	2	4	位置 $2,3$ 被占，向后找
15	5	5	位置 5 空

最终位置：

下标	0	1	2	3	4	5	6	7	8	9
内容	空	空	12	22	32	15	空	空	空	空

查找 32 时，从下标 2 开始，检查 $12, 22, 32$，第三次比较找到。

删除的细节

开放寻址法中，删除元素比插入和查找更麻烦。若直接把某个位置标为空，可能会破坏后续查找。

例如表大小为 10，线性探测后有：

下标	2	3	4
内容	12	22	32

查找 32 时，从 2 开始，经过 $12, 22$，再到 32。

如果删除 22 后直接把下标 3 标为空，那么查找 32 时：

- 先看下标 2，不是；
- 再看下标 3，发现空位；
- 可能误以为后面不再有相关元素，查找失败。

因此开放寻址删除时常需要“删除标记”，表示这个位置曾经有元素但已被删除。查找时遇到删除标记不能停止，插入时可以考虑复用它。

第一轮中，如果题目出现开放寻址删除，要重点看它是否区分“从未使用”和“已删除”。若没有删除操作，追踪会简单很多。

平均复杂度和最坏复杂度

哈希表常说查找、插入、删除平均为 $O(1)$。这建立在哈希函数分布较好、冲突较少的前提下。

若大量元素落入同一个桶，拉链法中这个桶会变长；开放寻址法中探测距离也会变长。最坏情况下，查找可能需要检查很多元素，复杂度退化为 $O(n)$。

因此，第一轮中遇到哈希表复杂度，要分清：

- 平均情况通常按 $O(1)$；
- 最坏情况可能是 $O(n)$；
- 若题目明确构造大量冲突，就不能再按常数处理；
- 若使用 STL 的 `unordered_map`、`unordered_set`，选择题常按平均 $O(1)$ 理解，但要知道最坏退化存在。

例题 3：冲突导致退化

表大小为 10，哈希函数为 $h(x) = x \bmod 10$。插入：

$$10,\ 20,\ 30,\ 40,\ 50.$$

这些数哈希值都为 0。若使用拉链法，它们都会进入桶 0。

查找 50 时，可能需要在桶 0 中依次检查多个元素。此时查找不再是“看一个位置就结束”。

这说明哈希表效率依赖哈希分布。平均快，不代表任何输入下都只需一步。

unordered_set

`unordered_set` 表示无序集合。它只关心某个键是否存在，不保存重复元素。

常见操作：

操作	含义
`insert(x)`	插入元素 x
`count(x)`	判断 x 是否存在，返回 0 或 1
`find(x)`	查找 x，找不到则返回 `end()`
`erase(x)`	删除元素 x
`size()`	元素个数

例如：

```
unordered_set<int> s;

s.insert(3);
s.insert(5);
s.insert(3);

cout << s.size() << endl;
```

虽然插入了两次 3，集合中只保存一个 3，所以元素为 $3, 5$，大小为 2。

例题 4：unordered_set 去重

阅读程序：

```
unordered_set<int> s;
int a[6] = {0, 4, 1, 4, 2, 1};

for (int i = 1; i <= 5; i++) {
    s.insert(a[i]);
}

cout << s.size() << endl;
```

数组中出现的不同元素有：

$$4,\ 1,\ 2.$$

所以 `s.size()` 为 3，输出：

```
3
```

`unordered_set` 不保存重复元素。如果题目要统计出现次数，不能只用集合大小。

例题 5：判断出现

```
unordered_set<int> s;
s.insert(10);
s.insert(20);

cout << s.count(10) << ' ' << s.count(15) << endl;
```

`10` 存在，所以 `s.count(10)` 为 1。`15` 不存在，所以 `s.count(15)` 为 0。

输出：

```
1 0
```

对 unordered_set 来说，count(x) 只可能是 0 或 1。

unordered_map

unordered_map 表示键到值的映射。它不仅判断键是否存在，还能保存这个键对应的值。

常见用途是统计次数。

```
unordered_map<int, int> cnt;

for (int i = 1; i <= n; i++) {
    int x;
    cin >> x;
    cnt[x]++;
}
```

这里 cnt[x] 表示整数 x 出现的次数。

例题 6：统计次数

阅读程序：

```
unordered_map<int, int> cnt;
int a[7] = {0, 2, 3, 2, 5, 3, 2};

for (int i = 1; i <= 6; i++) {
    cnt[a[i]]++;
}

cout << cnt[2] << ' ' << cnt[3] << ' ' << cnt[4] << endl;
```

数组中：

- 2 出现 3 次；
- 3 出现 2 次；
- 4 没有出现。

因此输出：

```
3 2 0
```

这里要注意：cnt[4] 若原本不存在，会被创建并赋默认值 0。这正是 operator[] 的一个重要行为。

operator[] 的影响

unordered_map 的 mp[key] 不只是查询。若 key 不存在，它会新建这个键，并把对应值初始化为默认值。

例如：

```
unordered_map<int, int> mp;

cout << mp[10] << endl;
cout << mp.size() << endl;
```

第一行访问 `mp[10]` 时，键 10 原本不存在，于是被插入，值为 0。所以第一行输出 0，第二行输出 1。

如果只想判断键是否存在，不希望创建新键，应使用 `count` 或 `find`。

例题 7：访问导致插入

```
unordered_map<int, int> mp;

mp[1] = 5;
cout << mp.count(2) << ' ';
cout << mp[2] << ' ';
cout << mp.count(2) << endl;
```

开始时只有键 1。

第一次 `mp.count(2)` 为 0。

执行 `mp[2]` 时，键 2 被创建，值为 0，所以输出 0。

之后 `mp.count(2)` 变为 1。

完整输出为：

```
0 0 1
```

程序阅读题中，看到 `mp[x]` 要注意它可能改变哈希表内容。

find 和 count

`count(x)` 常用于判断是否存在。`find(x)` 返回迭代器，若找不到，则返回 `end()`。

常见写法：

```
if (mp.find(x) != mp.end()) {
    cout << "found";
}
```

与：

```
if (mp.count(x)) {
    cout << "found";
}
```

在判断是否存在时含义接近。

但是对于 `unordered_map`，如果还要读取对应值，`find` 可以避免重复查找，也避免误插入。

```
auto it = mp.find(x);
if (it != mp.end()) {
    cout << it->second;
}
```

第一轮中，不需要深入迭代器语法，但要知道 `find(x) == end()` 表示没找到。

有序性误区

`unordered_set` 和 `unordered_map` 的名字中有 `unordered`，表示不按键的大小顺序存储。

例如：

```
unordered_set<int> s;
s.insert(3);
s.insert(1);
s.insert(2);

for (int x : s) {
    cout << x << ' ';
}
```

不能保证输出是：

```
1 2 3
```

也不能保证输出是插入顺序：

```
3 1 2
```

遍历顺序与内部桶、哈希函数和实现有关，不应作为题目推理依据。若题目要求按从小到大输出，应使用排序，或使用有序容器。

例题 8：不能依赖遍历顺序

下面说法是否正确：向 `unordered_set<int>` 中插入 $3, 1, 2$，再遍历输出，一定得到 $1, 2, 3$。

这个说法错误。

`unordered_set` 不维护升序。若要保证升序输出，需要把元素放入数组后排序，或使用能维护顺序的结构。本章只强调：哈希表适合查找，不适合维护顺序。

哈希表和数组计数

若键的范围很小，可以直接用数组计数，不一定需要哈希表。

例如所有数都在 0 到 1000 之间：

```
int cnt[1001];

for (int i = 1; i <= n; i++) {
    int x;
    cin >> x;
    cnt[x]++;
}
```

这种做法下标直接对应键，查找是 $O(1)$，实现也简单。

若键的范围很大，例如可能达到 10^9，但实际只有 n 个数，就不适合开一个巨大数组。此时可以用哈希表：

```
unordered_map<int, int> cnt;
```

例题 9：选择存储方式

有 n 个整数，范围为 0 到 10^6，需要统计每个数出现次数。若内存允许，可以使用数组 `cnt[1000001]`。

若整数范围为 -10^9 到 10^9，但 n 只有 10^5，用数组按值开空间不合适，可以使用 `unordered_map<int,int>`。

选择结构时，要看键的范围和实际元素数量。范围小，用数组很直接；范围大但元素少，哈希表更合适。

哈希表去重

去重是哈希表的常见用途。

```
unordered_set<int> s;
for (int i = 1; i <= n; i++) {
    int x;
    cin >> x;
    s.insert(x);
}
cout << s.size() << endl;
```

这段程序输出不同整数的个数。

例题 10：不同元素个数

输入序列：

$$5,\ 2,\ 5,\ 3,\ 2,\ 8,\ 8.$$

插入 unordered_set 后，保留的不同元素为：

$$5,\ 2,\ 3,\ 8.$$

不同元素个数为 4。

如果题目问“出现次数最多的元素”，只用 unordered_set 不够，需要 unordered_map 统计次数。

哈希表统计频率

统计频率常用 unordered_map。

```
unordered_map<string, int> cnt;

for (int i = 1; i <= n; i++) {
    string s;
    cin >> s;
    cnt[s]++;
}
```

这里键是字符串，值是出现次数。

例题 11：字符串频率

阅读程序：

```
unordered_map<string, int> cnt;

cnt["cat"]++;
cnt["dog"]++;
cnt["cat"]++;
cnt["bird"]++;
cnt["dog"]++;

cout << cnt["cat"] << ' ' << cnt["dog"] << ' ' << cnt["fish"] << endl;
```

统计结果：

- `"cat"` 出现 2 次；
- `"dog"` 出现 2 次；
- `"fish"` 没有出现，但访问 `cnt["fish"]` 会创建它，值为 0。

输出：

```
2 2 0
```

若后面再输出 `cnt.size()`，要记得 `"fish"` 已经被插入。

哈希表与两数和

哈希表常用于“查找是否存在某个配对元素”的问题。

例如给定数组，判断是否存在两个不同位置的数之和为 T。

可以从左到右扫描。处理当前数 x 时，检查之前是否出现过 $T-x$。

```cpp
unordered_set<int> s;
bool ok = false;

for (int i = 1; i <= n; i++) {
    int x;
    cin >> x;
    if (s.count(T - x)) ok = true;
    s.insert(x);
}
```

这里先查找，再插入当前数，可以避免同一个位置被使用两次。

例题 12：两数和判断

数组为：

$$4,\ 1,\ 6,\ 3$$

目标和 $T=7$。

扫描过程：

当前数	查找目标	已出现集合	是否找到
4	3	空	否
1	6	4	否
6	1	$4,1$	是
3	4	$4,1,6$	是

当扫描到 6 时，之前出现过 1，所以存在两数和为 7。

若程序先插入再查找，在 $T=2x$ 时可能错误地把当前元素和自己配对。读程序时要看查找和插入的先后顺序。

哈希表和离散化的区别

哈希表和离散化都能处理“大值域中的少量元素”，但用途不同。

哈希表直接根据键查找值：

```
cnt[x]++;
```

离散化则把出现过的值排序、去重，映射成较小编号：

$$\text{原值} \to \text{编号}.$$

例如原值为：

$$1000000000,\ -5,\ 1000000000,\ 7.$$

离散化后不同值为：

$$-5,\ 7,\ 1000000000.$$

可以映射为：

$$-5 \to 1, \quad 7 \to 2, \quad 1000000000 \to 3.$$

哈希表适合快速查找原键；离散化适合后续需要按大小顺序、排名、数组下标处理的场景。

本章只需要先分清：哈希表不保证有序，离散化通常会排序。

程序阅读中的哈希表

读哈希表程序，可以按下面顺序。

第一，判断容器类型。是 `unordered_set`，还是 `unordered_map`。

第二，判断键和值。集合只有键；映射有键和值。

第三，判断是否会插入。`insert` 会插入，`mp[x]` 也可能插入，`count` 和 `find` 通常只查询。

第四，判断重复元素如何处理。集合不保存重复元素，映射可以累加次数。

第五，判断输出是否依赖顺序。若遍历 `unordered_set` 或 `unordered_map`，不要默认升序或插入顺序。

第六，判断复杂度。平均常数，最坏可能线性。

例题 13：程序输出

```
unordered_map<int, int> mp;

mp[2]++;
mp[3]++;
mp[2]++;
mp[5] = mp[2] + mp[3];

cout << mp[2] << ' ' << mp[3] << ' ' << mp[5] << endl;
```

逐步计算：

- `mp[2]++` 后，`mp[2]=1`；

- `mp[3]++` 后，`mp[3]=1`；
- 再次 `mp[2]++` 后，`mp[2]=2`；
- `mp[5]=mp[2]+mp[3]=3`。

输出：

```
2 1 3
```

这道题的关键是：未出现过的键在使用 `mp[x]++` 时会先创建，初值为 0，再加 1。

例题 14：集合大小

```
unordered_set<int> s;

for (int i = 1; i <= 5; i++) {
    s.insert(i % 3);
}

cout << s.size() << endl;
```

i 从 1 到 5 时，`i % 3` 依次为：

$$1,\ 2,\ 0,\ 1,\ 2.$$

不同值为 $0, 1, 2$，所以集合大小为 3。

输出：

```
3
```

完善程序中的哈希表

哈希表完善程序常见空缺包括：

- 插入元素；
- 判断元素是否存在；
- 统计次数；
- 访问键对应的值；
- 查找配对元素；
- 避免把当前元素与自己配对；
- 处理重复出现的键。

例题 15：补全去重计数

```
unordered_set<int> s;

for (int i = 1; i <= n; i++) {
    int x;
    cin >> x;
    _____;
}

cout << s.size() << endl;
```

若要统计不同整数个数，空缺应填：

```
s.insert(x)
```

`unordered_set` 会自动忽略重复元素。

例题 16：补全频率统计

```
unordered_map<int, int> cnt;

for (int i = 1; i <= n; i++) {
    int x;
    cin >> x;
    _____;
}
```

若要统计每个整数出现次数，空缺应填：

```
cnt[x]++
```

如果 `x` 第一次出现，`cnt[x]` 会先变成 0，再加 1。

例题 17：补全两数和

```
unordered_set<int> s;
bool ok = false;

for (int i = 1; i <= n; i++) {
    int x;
    cin >> x;
    if (_____) ok = true;
    s.insert(x);
}
```

若目标和为 `T`，需要判断之前是否出现过 `T - x`。空缺应填：

```
s.count(T - x)
```

注意这里先查找再插入当前数，保证配对元素来自之前的位置。若题目允许同一个值出现两次，后续相同值会通过第二次扫描处理。

常见错误

哈希表题的错误集中在下面几类。

错误	说明
忘记冲突	不同键可能映射到同一桶
把平均复杂度当绝对	最坏情况可能退化为 $O(n)$
依赖遍历顺序	`unordered_set` 和 `unordered_map` 不保证升序
集合统计频率	`unordered_set` 只能判断是否存在，不保存次数
`mp[x]` 误当纯查询	访问不存在的键会创建它
`count` 含义看错	在 `unordered_set` 中结果只有 0 或 1
查找插入顺序错	两数和中可能把当前元素和自己配对
数组和哈希表乱用	小值域可用数组，大值域更适合哈希表
开放寻址删除处理错	删除标记和空位不能混淆
字符串哈希混淆	`unordered_map<string,int>` 不是第 36 章的滚动字符串哈希

做哈希表题时，先明确“键是什么，值是什么”。如果只有存在性，用集合；如果要统计次数或保存对应值，用映射。

原创例题

例题 18：访问导致大小变化

阅读程序：

```
unordered_map<int, int> mp;

mp[1] = 3;
mp[2] = 5;

cout << mp.size() << ' ';
cout << mp[3] << ' ';
cout << mp.size() << endl;
```

开始时有键 $1,2$，所以 `mp.size()` 为 2。

访问 `mp[3]` 时，键 3 不存在，会被创建，默认值为 0。所以第二个输出是 0。

此时哈希表中有键 $1,2,3$，大小变为 3。

完整输出为：

```
2 0 3
```

变式 1：使用 count

如果改成：

```
cout << mp.size() << ' ';
cout << mp.count(3) << ' ';
cout << mp.size() << endl;
```

`count(3)` 只判断是否存在，不会创建键 3。因此输出为：

```
2 0 2
```

`mp[x]` 和 `mp.count(x)` 的副作用不同，这是第一轮很常见的程序阅读点。

例题 19：查找重复元素

阅读程序：

```
unordered_set<int> s;
bool ok = false;

int a[6] = {0, 4, 2, 5, 2, 7};

for (int i = 1; i <= 5; i++) {
    if (s.count(a[i])) ok = true;
    s.insert(a[i]);
}

cout << ok << endl;
```

扫描过程：

i	$a[i]$	之前是否出现	ok
1	4	否	false
2	2	否	false
3	5	否	false
4	2	是	true
5	7	否	true

最终输出：

```
1
```

在 C++ 中，`true` 用 `cout` 输出通常显示为 `1`，`false` 显示为 `0`。

变式 2：先插入再判断

若循环改成：

```
for (int i = 1; i <= 5; i++) {
    s.insert(a[i]);
    if (s.count(a[i])) ok = true;
}
```

那么第一次循环中，插入 4 后，`s.count(4)` 已经为真，`ok` 立刻变为 `true`。这段代码不能用来判断是否出现重复元素。

顺序很重要：判断重复时，应先查旧集合，再插入当前元素。

小结

哈希表的优势是快速定位，但它不维护顺序，也不能避免所有最坏情况。第一轮中，只要分清集合与映射、存在性与频率、查询与插入的区别，就能处理大多数哈希表程序阅读和完善程序题。下一章进入二叉树，结构会从“按键定位”转向“按父子关系组织元素”。

0x14 二叉树

二叉树是一种每个结点最多有两个子结点的树结构。它在 CSP-S 第一轮中经常以选择题的形式出现：判断结点数量关系，计算树的高度，写出前序、中序、后序遍历结果，根据遍历序列还原树，或者读懂递归遍历代码。

本章只讲二叉树本身。二叉搜索树会在下一章系统讲，那里会加入 “左子树小、右子树大” 这样的有序性质。本章先把二叉树的结构和遍历顺序讲清楚。

结点和子树

二叉树由若干个结点组成。每个结点最多有两个子结点，通常称为左儿子和右儿子。

下面是一棵二叉树：

```
      A
    /   \
   B     C
  / \     \
 D   E     F
```

在这棵树中：

- A 是根结点；
- B 是 A 的左儿子，C 是 A 的右儿子；
- D 和 E 是 B 的子结点；
- F 是 C 的右儿子；
- D,E,F 没有子结点，称为叶子结点。

对于任意一个结点，它的左儿子连同下面所有结点构成左子树；右儿子连同下面所有结点构成右子树。

例如对结点 A 来说，左子树是：

```
   B
  / \
 D   E
```

右子树是：

```
C
 \
  F
```

二叉树中的 “左” 和 “右” 是有意义的。即使两个结点的值一样，左儿子和右儿子交换后，也可能表示不同的二叉树。

深度和高度

树题中常见两个量：深度和高度。

本书采用下面的约定：

- 根结点深度为 1；
- 一个结点的儿子深度比它大 1；
- 树的高度是所有结点深度的最大值。

例如：

```
      A
    /   \
   B     C
  / \     \
 D   E     F
```

各结点深度为：

结点	深度
A	1
B,C	2
D,E,F	3

所以这棵树的高度为 3。

有些资料会把根结点深度记为 0，此时高度也会相应少 1。做题时要以题目定义为准。如果题目没有特别说明，第一轮中通常按层数理解更方便，也就是根在第 1 层。

例题 1：计算高度

给出二叉树：

```
      1
     / \
    2   3
   /
  4
 /
5
```

从根到最深叶子的路径为：

$$1 \to 2 \to 4 \to 5.$$

这条路径上有 4 个结点，所以树的高度为 4。

若某题把根深度定义为 0，则最深结点深度为 3。同一棵树，数值可能不同，原因是定义不同。

满二叉树与完全二叉树

第一轮常考两种特殊二叉树：满二叉树和完全二叉树。

满二叉树可以理解为：每一层都排满了结点。高度为 h 的满二叉树结点数为：

$$1 + 2 + 4 + \cdots + 2^{h-1} = 2^h - 1.$$

例如高度为 3 的满二叉树：

```
      A
    /   \
   B     C
  / \   / \
 D   E F   G
```

共有 7 个结点。

完全二叉树可以理解为：除了最后一层，前面的层都排满；最后一层的结点从左到右连续排列，中间不能空。

例如：

```
      A
    /   \
   B     C
  / \   /
 D   E F
```

这是完全二叉树。最后一层有 `D,E,F`，它们从左到右连续出现。

而下面这棵树不是完全二叉树：

```
      A
    /   \
   B     C
  /       \
 D         F
```

因为最后一层中，`D` 后面的位置空了，却又在更右边出现了 `F`。

堆通常用完全二叉树表示。上一章讲堆时，数组下标关系正是依赖完全二叉树的结构。

二叉树的性质

二叉树常用性质不多，但要记准确。

高度为 h 的二叉树，最多有：

$$2^h - 1$$

个结点。

第 k 层最多有：

$$2^{k-1}$$

个结点。

若一棵二叉树有 n_0 个叶子结点，n_2 个度为 2 的结点，则：

$$n_0 = n_2 + 1.$$

这里 “度” 表示一个结点的子结点个数。度为 0 的结点就是叶子，度为 1 的结点只有一个儿子，度为 2 的结点有两个儿子。

例题 2：叶子数关系

某二叉树中，度为 2 的结点有 6 个。问叶子结点有多少个？

由性质：

$$n_0 = n_2 + 1.$$

所以叶子结点数为：

$$6 + 1 = 7.$$

这条性质和度为 1 的结点数量无关。无论有多少个只有一个儿子的结点，叶子数都比度为 2 的结点数多 1。

性质的简单说明

设二叉树中：

- 叶子结点数为 n_0；
- 度为 1 的结点数为 n_1；
- 度为 2 的结点数为 n_2。

总结点数为：

$$n = n_0 + n_1 + n_2.$$

除根结点外，每个结点都有一条来自父结点的边，所以边数为 $n - 1$。

另一方面，从父结点往儿子数边，度为 1 的结点贡献 1 条边，度为 2 的结点贡献 2 条边，所以边数也等于：

$$n_1 + 2n_2.$$

因此：

$$n_0 + n_1 + n_2 - 1 = n_1 + 2n_2.$$

整理得到：

$$n_0 = n_2 + 1.$$

这就是叶子数关系的来源。

链式存储

二叉树最直接的存储方式，是每个结点保存左儿子和右儿子。

在程序中常写成数组：

```
int lch[N], rch[N];
```

其中 `lch[u]` 表示结点 u 的左儿子，`rch[u]` 表示结点 u 的右儿子。若某个儿子不存在，可以记为 0 或 -1，具体要看题目约定。

例如：

```
lch[1] = 2; rch[1] = 3;
lch[2] = 4; rch[2] = 5;
lch[3] = 0; rch[3] = 6;
```

表示：

```
      1
    /   \
   2     3
  / \     \
 4   5     6
```

这种存储方式适合一般二叉树。树的形状不要求完整，也不要求结点编号满足堆那样的父子下标关系。

例题 3：根据数组画树

已知：

结点	左儿子	右儿子
1	2	3
2	4	0
3	5	6
4	0	0
5	0	0
6	0	0

画出二叉树。

结点 1 的左右儿子是 $2, 3$；结点 2 只有左儿子 4；结点 3 有左右儿子 $5, 6$。树为：

```
      1
    /   \
   2     3
  /     / \
 4     5   6
```

其中 $4, 5, 6$ 是叶子结点。

读链式存储时，先从根开始，根据左右儿子表逐层画出结构，通常最稳。

数组存储

完全二叉树可以用数组顺序存储。

若下标从 1 开始，结点 i 的关系为：

位置	下标
父结点	$\lfloor i/2 \rfloor$
左儿子	$2i$
右儿子	$2i+1$

例如数组：

$$a[1..6] = A,\ B,\ C,\ D,\ E,\ F.$$

对应完全二叉树：

```
      A
    /   \
   B     C
  / \   /
 D   E F
```

这和堆的数组表示相同。

但一般二叉树不一定适合这样存。如果树很稀疏，用数组下标硬存会浪费很多位置。

例如：

```
A
 \
  B
   \
    C
     \
      D
```

若按完全二叉树下标存储，D 的下标会很大，而中间很多位置为空。

因此，数组下标公式主要用于完全二叉树、堆或题目明确给出的顺序存储二叉树。

前序遍历

二叉树遍历是第一轮高频内容。遍历就是按某种顺序访问树中所有结点。

前序遍历的顺序是：

$$\text{根} \to \text{左子树} \to \text{右子树}.$$

对应递归代码：

```cpp
void preorder(int u) {
    if (u == 0) return;
    cout << u << ' ';
    preorder(lch[u]);
    preorder(rch[u]);
}
```

先访问当前根结点，再遍历左子树，最后遍历右子树。

例题 4：写出前序遍历

给定二叉树：

```
      A
    /   \
   B     C
  / \     \
 D   E     F
```

前序遍历：

先访问根 A；再遍历左子树 B,D,E；再遍历右子树 C,F。

所以序列为：

$$A,\ B,\ D,\ E,\ C,\ F.$$

不要把“先访问根”理解成只看第一层。访问 B 后，还要先进入 B 的左子树，再进入 B 的右子树。

中序遍历

中序遍历的顺序是：

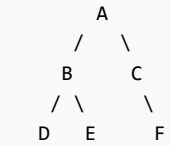
$$\text{左子树} \to \text{根} \to \text{右子树}.$$

对应递归代码：

```
void inorder(int u) {
    if (u == 0) return;
    inorder(lch[u]);
    cout << u << ' ';
    inorder(rch[u]);
}
```

例题 5：写出中序遍历

仍看这棵树：



中序遍历左子树 B 时，先访问 D，再访问 B，再访问 E，得到：

$$D,\ B,\ E.$$

然后访问根 A。

右子树 C 没有左子树，先访问 C，再访问右子树 F，得到：

$$C,\ F.$$

所以整棵树的中序遍历为：

$$D,\ B,\ E,\ A,\ C,\ F.$$

中序遍历最容易错在根的位置。根不是最先访问，也不是最后访问，而是在左子树之后、右子树之前访问。

后序遍历

后序遍历的顺序是：

$$\text{左子树} \to \text{右子树} \to \text{根}.$$

对应递归代码：

```
void postorder(int u) {
    if (u == 0) return;
    postorder(lch[u]);
    postorder(rch[u]);
    cout << u << ' ';
}
```

例题 6：写出后序遍历

对于：

```
      A
    /   \
   B     C
  / \     \
 D   E     F
```

左子树 B 的后序遍历为：

$$D,\ E,\ B.$$

右子树 C 的后序遍历为：

$$F,\ C.$$

最后访问根 A。

所以整棵树的后序遍历为：

$$D,\ E,\ B,\ F,\ C,\ A.$$

后序遍历常用于“先处理子结点，再处理当前结点”的程序，例如计算子树大小、释放树、求表达式树的值等。

层序遍历

层序遍历按照从上到下、从左到右的顺序访问结点。它通常用队列实现。

对于：

```
      A
    /   \
   B     C
  / \     \
 D   E     F
```

层序遍历为：

$$A,\ B,\ C,\ D,\ E,\ F.$$

队列实现思路：

```
void bfs(int root) {
    queue<int> q;
    q.push(root);

    while (!q.empty()) {
        int u = q.front();
        q.pop();

        cout << u << ' ';

        if (lch[u]) q.push(lch[u]);
        if (rch[u]) q.push(rch[u]);
    }
}
```

过程是：先把根入队。每次取出队头结点，访问它，再把它的左儿子、右儿子依次入队。由于队列先进先出，结点会按层被访问。

例题 7：层序追踪

给定：

```
       1
     /   \
    2     3
   /     / \
  4     5   6
```

层序遍历过程：

步骤	取出结点	新入队结点	队列内容
初始	—	1	1
1	1	2, 3	2, 3
2	2	4	3, 4
3	3	5, 6	4, 5, 6
4	4	无	5, 6
5	5	无	6
6	6	无	空

层序遍历序列为：

$$1,\ 2,\ 3,\ 4,\ 5,\ 6.$$

层序遍历题要注意左右儿子的入队顺序。若先入右儿子，再入左儿子，同一层的访问顺序会改变。

根据遍历序列还原树

二叉树可以由某些遍历序列唯一确定，但不是任意一种遍历都够。

常见结论：

已知序列	是否能唯一确定一般二叉树
前序 + 中序	可以
后序 + 中序	可以
前序 + 后序	通常不可以
只有一种遍历	不可以

前序遍历的第一个结点是根。中序遍历中，根左侧是左子树，根右侧是右子树。因此 “前序 + 中序” 可以递归还原树。

后序遍历的最后一个结点是根。配合中序，也可以递归还原树。

例题 8：由前序和中序还原

已知某二叉树：

$$\text{前序} = A,\ B,\ D,\ E,\ C,\ F,$$

$$\text{中序} = D,\ B,\ E,\ A,\ C,\ F.$$

求这棵树。

前序第一个结点是根，所以根为 A。

在中序序列中，A 左侧为：

$$D,\ B,\ E.$$

右侧为：

$$C,\ F.$$

所以左子树有 3 个结点，右子树有 2 个结点。

前序中根 A 后面的 3 个结点属于左子树：

$$B,\ D,\ E.$$

剩下属于右子树：

$$C,\ F.$$

左子树：

- 前序为 B, D, E；
- 中序为 D, B, E。

左子树根为 B。中序中 B 左侧是 D，右侧是 E，所以 D 是左儿子，E 是右儿子。

右子树：

- 前序为 C, F；
- 中序为 C, F。

右子树根为 C。中序中 C 左侧为空，右侧是 F，所以 F 是 C 的右儿子。

最终树为：

```
      A
    /   \
   B     C
  / \     \
 D   E     F
```

这类题的步骤要固定：先找根，再用中序切左右子树，再按子树大小切另一个序列。

例题 9：为什么前序和后序不够

两棵不同的二叉树：

```
  A
 /
B
```

和：

```
A
 \
  B
```

它们的前序遍历都是：

$$A,\ B.$$

后序遍历也都是：

$$B,\ A.$$

但树的形状不同。第一棵中 B 是左儿子，第二棵中 B 是右儿子。

所以只有前序和后序，通常不能唯一确定一般二叉树。

递归遍历程序

二叉树遍历程序常用递归写。读这类程序时，要看输出语句相对于递归调用的位置。

```
void dfs(int u) {
    if (u == 0) return;
    cout << u << ' ';
    dfs(lch[u]);
    dfs(rch[u]);
}
```

输出在两个递归调用之前，是前序遍历。

```
void dfs(int u) {
    if (u == 0) return;
    dfs(lch[u]);
    cout << u << ' ';
    dfs(rch[u]);
}
```

输出在左递归之后、右递归之前，是中序遍历。

```
void dfs(int u) {
    if (u == 0) return;
    dfs(lch[u]);
    dfs(rch[u]);
    cout << u << ' ';
}
```

输出在两个递归调用之后，是后序遍历。

例题 10：判断程序输出

已知：

```
lch[1] = 2; rch[1] = 3;
lch[2] = 4; rch[2] = 5;
lch[3] = 0; rch[3] = 6;
```

程序：

```
void dfs(int u) {
    if (u == 0) return;
    dfs(lch[u]);
    cout << u << ' ';
    dfs(rch[u]);
}
```

这段程序是中序遍历。

树为：

```
       1
     /   \
    2     3
   / \     \
  4   5     6
```

中序遍历为：

$$4,\ 2,\ 5,\ 1,\ 3,\ 6.$$

所以输出：

```
4 2 5 1 3 6
```

程序阅读题中，先判断遍历类型，再写序列，通常比逐行展开递归更快。

子树大小

二叉树程序中常见一个递归函数，用来计算每个结点的子树大小。

```
int siz[N];

void dfs(int u) {
    if (u == 0) return;
    siz[u] = 1;
    dfs(lch[u]);
    dfs(rch[u]);
    siz[u] += siz[lch[u]] + siz[rch[u]];
}
```

若 `siz[0]=0`，则这段程序计算每个结点子树中的结点数。

更常见的写法是让函数直接返回大小：

```
int calc(int u) {
    if (u == 0) return 0;
    return calc(lch[u]) + calc(rch[u]) + 1;
}
```

这里当前子树大小等于左子树大小，加右子树大小，再加根结点自己。

例题 11：计算子树大小

树为：

```
      1
    /   \
   2     3
  / \     \
 4   5     6
```

求各结点子树大小。

叶子结点 $4,5,6$ 的子树大小都是 1。

结点 2 的子树包含 $2,4,5$，大小为 3。

结点 3 的子树包含 $3,6$，大小为 2。

根结点 1 的子树包含所有结点，大小为 6。

结点	子树大小
1	6
2	3
3	2
4	1
5	1
6	1

后序思路适合计算子树信息，因为必须先知道左右子树结果，才能得到当前结点结果。

树的高度递归

树高也可以递归计算。

```
int height(int u) {
    if (u == 0) return 0;
    return max(height(lch[u]), height(rch[u])) + 1;
}
```

空树高度为 0。非空树高度等于左右子树高度的较大值加 1。

例题 12：程序含义

```
int f(int u) {
    if (u == 0) return 0;
    int x = f(lch[u]);
    int y = f(rch[u]);
    return max(x, y) + 1;
}
```

这段函数返回以 u 为根的子树高度。

若树为：

```
    1
   /
  2
   \
    3
```

则：

- f(3)=1;
- f(2)=2;
- f(1)=3。

所以整棵树高度为 3。

若把 max 改成加法：

```
return x + y + 1;
```

函数就变成计算子树结点数，而不是高度。完善程序题常把这类位置设为空缺。

表达式树

二叉树可以表示表达式。叶子结点是操作数，内部结点是运算符。

例如表达式：

$$(a+b)\times(c-d)$$

可以表示为：

```
      *
    /   \
   +     -
  / \   / \
 a   b c   d
```

前序遍历得到前缀表达式：

$$*,\ +,\ a,\ b,\ -,\ c,\ d.$$

中序遍历得到：

$$a,\ +,\ b,\ *,\ c,\ -,\ d.$$

这个中序序列若不加括号，可能产生歧义。表达式树的中序输出通常需要根据结构加括号。

后序遍历得到后缀表达式：

$$a,\ b,\ +,\ c,\ d,\ -,\ *.$$

例题 13：计算表达式树

给定表达式树：

```
      +
    /   \
   3     *
        / \
       4   5
```

它表示：

$$3 + 4 \times 5.$$

计算时，先算右子树：

$$4 \times 5 = 20.$$

再算根：

$$3 + 20 = 23.$$

若用后序遍历计算，顺序为：

$$3,\ 4,\ 5,\ *,\ +.$$

先把操作数压入栈，遇到运算符就取出两个数计算，也能得到 23。本章只要求能把表达式树和遍历顺序对应起来，不展开后缀表达式求值细节。

二叉树与普通树

普通树中，一个结点可以有任意多个儿子；二叉树中，每个结点最多两个儿子。普通树有时可以转成二叉树表示，常见方法是 “左儿子右兄弟” 表示法。

在这种表示中：

- 左指针指向第一个儿子；
- 右指针指向下一个兄弟。

例如普通树：

```
A
├─ B
├─ C
└─ D
```

可以表示为：

```
A
/
B
 \
  C
   \
    D
```

这里 B 是 A 的第一个儿子，C 是 B 的兄弟，D 是 C 的兄弟。

第一轮中，左儿子右兄弟表示法可能出现在程序阅读中。读这种结构时，不要把右指针误认为“右子树”含义完全相同，它在这种表示里表示兄弟关系。

程序阅读中的二叉树

读二叉树程序，可以按下面顺序。

第一，确定存储方式。是 lch/rch 链式存储，还是数组下标完全二叉树存储。

第二，找到根。题目可能直接给根，也可能需要找没有父结点的那个结点。

第三，判断空结点标记。是 0、−1，还是其他值。

第四，判断遍历类型。输出在递归前、中、后，分别对应前序、中序、后序。

第五，判断函数返回值。返回的是高度、结点数、叶子数，还是某种计算结果。

第六，追踪左右儿子顺序。左右顺序改变，遍历结果也会改变。

例题 14：寻找根

有 5 个结点，左右儿子如下：

结点	左儿子	右儿子
1	2	3
2	0	0
3	4	5
4	0	0
5	0	0

根结点是哪个？

根结点没有父结点。先标记所有出现过的儿子：$2,3,4,5$ 都是别人的儿子。只有 1 没有作为儿子出现，所以根是 1。

程序中常用：

```
bool vis[N];

for (int i = 1; i <= n; i++) {
    if (lch[i]) vis[lch[i]] = true;
    if (rch[i]) vis[rch[i]] = true;
}

for (int i = 1; i <= n; i++) {
    if (!vis[i]) root = i;
}
```

这段代码的含义是找没有父结点的结点。

完善程序中的二叉树

二叉树完善程序常见空缺包括：

- 递归边界；
- 前序、中序、后序输出位置；
- 计算高度时使用 max；
- 计算子树大小时左右子树相加；
- 层序遍历中的入队顺序；
- 根据遍历序列还原树时切分左右子树；
- 找根时标记儿子。

例题 15：补全高度函数

```
int height(int u) {
    if (u == 0) return 0;
    int x = height(lch[u]);
    int y = height(rch[u]);
    return _____;
}
```

高度等于左右子树高度较大者加 1，所以空缺应填：

```
max(x, y) + 1
```

若填 `x + y + 1`，得到的是结点数，不是高度。

例题 16：补全子树大小

```
int calc(int u) {
    if (u == 0) return 0;
    return _____;
}
```

若 `calc(u)` 表示以 `u` 为根的子树结点数，则空缺应填：

```
calc(lch[u]) + calc(rch[u]) + 1
```

这里加 1 是因为要计入当前结点。

例题 17：补全中序遍历

```
void dfs(int u) {
    if (u == 0) return;
    _____;
    cout << u << ' ';
    _____;
}
```

中序遍历为左子树、根、右子树，所以空缺应填：

```
dfs(lch[u])
dfs(rch[u])
```

若输出语句放在第一空前，就是前序遍历；若输出语句放在两个递归之后，就是后序遍历。

常见错误

二叉树题的错误集中在下面几类。

错误	说明
左右儿子看反	左右顺序会影响遍历结果
把二叉树当完全二叉树	一般二叉树不能乱用 $2i, 2i+1$ 下标
遍历顺序混淆	前序看根在前，中序看根在中间，后序看根在后
中序还原漏切子树	必须用根在中序中的位置分左右
前序后序误判	一般二叉树仅靠前序和后序不能唯一确定
高度和结点数混淆	高度用 `max`，结点数用加法
空结点标记看错	可能是 0，也可能是 -1
根结点找错	根是没有父结点的结点
层序入队顺序错	先左后右和先右后左结果不同
左儿子右兄弟误读	右指针可能表示兄弟关系

做二叉树题时，建议先画树，再写序列。若题目给的是数组或左右儿子表，先把结构画出来，后面的遍历、树高、子树大小都会清楚很多。

原创例题

例题 18：综合遍历

给定二叉树：

```
      1
    /   \
   2     3
    \   / \
     4 5   6
```

分别写出前序、中序、后序、层序遍历。

前序为根、左、右：

$$1,\ 2,\ 4,\ 3,\ 5,\ 6.$$

中序为左、根、右：

$$2,\ 4,\ 1,\ 5,\ 3,\ 6.$$

后序为左、右、根：

$$4,\ 2,\ 5,\ 6,\ 3,\ 1.$$

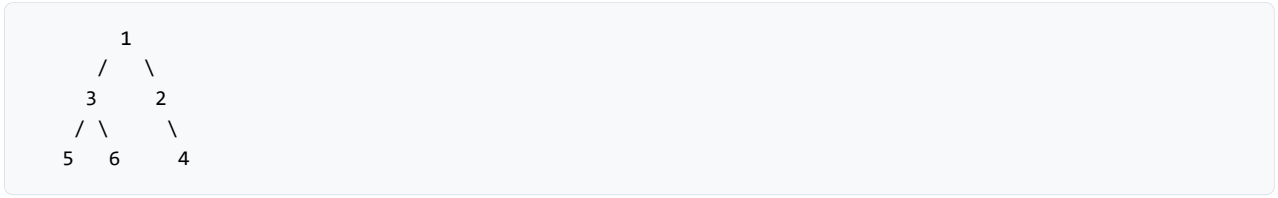
层序为从上到下、从左到右：

$$1,\ 2,\ 3,\ 4,\ 5,\ 6.$$

同一棵树，不同遍历序列差别很大。做题时要先写遍历规则，再按规则走，不要凭印象写。

变式 1：交换左右儿子

若把根结点 1 的左右儿子交换，树变成：



前序变为：

$$1,\ 3,\ 5,\ 6,\ 2,\ 4.$$

中序变为：

$$5,\ 3,\ 6,\ 1,\ 2,\ 4.$$

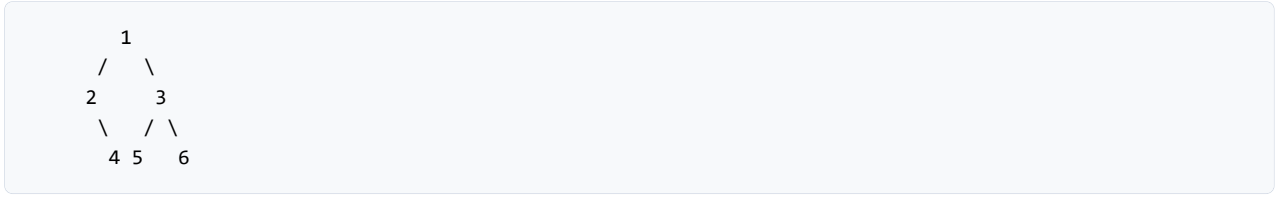
后序变为：

$$5,\ 6,\ 3,\ 4,\ 2,\ 1.$$

这说明二叉树的左右位置不可随意交换。交换后，树已经变了。

变式 2：计算高度和叶子数

原树：



高度为 3。叶子结点为 $4, 5, 6$，共 3 个。度为 2 的结点只有 $1, 3$，共 2 个，满足：

$$n_0 = n_2 + 1.$$

即：

$$3 = 2 + 1.$$

小结

二叉树的重点是结构和顺序。先确定左右儿子，再判断遍历顺序；先画出树，再计算高度、叶子数和子树大小。下一章进入二叉搜索树，会在二叉树结构上加入大小关系，中序遍历也会因此获得新的含义。

0x15 二叉搜索树

二叉搜索树是一种带有大小关系的二叉树。普通二叉树只规定每个结点最多有两个儿子；二叉搜索树还规定：对每个结点来说，左子树中的值都小于它，右子树中的值都大于它。

正是这个性质，让二叉搜索树可以像二分一样查找。查找一个值时，如果目标值比当前结点小，就去左子树；如果比当前结点大，就去右子树；如果相等，就找到了。

CSP-S 第一轮中，二叉搜索树常以选择题和程序阅读形式出现：判断一棵树是否是二叉搜索树，按顺序插入若干数后画出树形，写出中序遍历结果，判断查找路径，分析删除结点后的调整方式。本章只讲普通二叉搜索树，不讲平衡树。平衡树的旋转、红黑树、AVL 树不属于本书主线。

搜索性质

二叉搜索树满足这样的规则：

$$\text{左子树中的所有值} < \text{根结点的值} < \text{右子树中的所有值}.$$

这个规则不仅对根结点成立，也要对树中每一个结点成立。

例如下面是一棵二叉搜索树：

```
      8
    /   \
   3     10
  / \      \
 1   6      14
    / \    /
   4   7  13
```

对根结点 8 来说，左子树中的 $1, 3, 4, 6, 7$ 都小于 8，右子树中的 $10, 13, 14$ 都大于 8。

对结点 3 来说，左子树中的 1 小于 3，右子树中的 $4, 6, 7$ 都大于 3。

对结点 10 来说，右子树中的 $13, 14$ 都大于 10，并且结点 14 的左儿子 13 小于 14。

判断二叉搜索树时，不能只检查一个结点和它的直接儿子。还要保证整棵左子树都小于当前结点，整棵右子树都大于当前结点。

例题 1：判断是否为二叉搜索树

判断下面这棵树是否为二叉搜索树。

```
      8
    /   \
   4     12
  / \    /
 2   9  10
```

只看父子关系时：

- $4 < 8$，$12 > 8$；
- $2 < 4$，$9 > 4$；
- $10 < 12$。

这些直接父子关系都看似合理。

但结点 9 位于根结点 8 的左子树中。二叉搜索树要求根的左子树中所有结点都小于 8，而 $9 > 8$，所以这棵树不是二叉搜索树。

这类题的陷阱就在于：不能只看局部父子大小，还要看祖先结点给出的范围限制。

查找

在二叉搜索树中查找一个值 x，从根开始：

- 若当前结点为空，说明没有找到；
- 若 x 等于当前结点值，查找成功；
- 若 x 小于当前结点值，去左子树；
- 若 x 大于当前结点值，去右子树。

例如在下面的树中查找 7：

```
      8
    /   \
   3     10
  / \      \
 1   6      14
    / \    /
   4   7  13
```

查找路径为：

$$8 \to 3 \to 6 \to 7.$$

因为：

- $7 < 8$，去左子树；
- $7 > 3$，去右子树；
- $7 > 6$，去右子树；
- 找到 7。

查找 5 时，路径为：

$$8 \to 3 \to 6 \to 4.$$

到 4 后，$5 > 4$，应去右子树，但右子树为空，所以 5 不存在。

例题 2：查找路径

在下面二叉搜索树中查找 13。

```
      9
    /   \
   5     15
```

```
   / \    /
  2   7  12
          \
           13
```

从根 9 开始：

- $13 > 9$，去右子树，到 15；
- $13 < 15$，去左子树，到 12；
- $13 > 12$，去右子树，到 13。

查找路径为：

$$9 \to 15 \to 12 \to 13.$$

查找路径体现了二叉搜索树的大小关系。程序阅读题中，如果只问访问了哪些结点，可以直接按比较结果走。

查找程序

二叉搜索树查找可以递归写：

```
bool find(int u, int x) {
    if (u == 0) return false;
    if (val[u] == x) return true;
    if (x < val[u]) return find(lch[u], x);
    return find(rch[u], x);
}
```

也可以循环写：

```
bool find(int root, int x) {
    int u = root;
    while (u != 0) {
        if (val[u] == x) return true;
        if (x < val[u]) u = lch[u];
        else u = rch[u];
    }
    return false;
}
```

两种写法含义一样。

这里 `val[u]` 是结点 `u` 的值，`lch[u]` 和 `rch[u]` 分别是左右儿子编号。空结点用 0 表示。

例题 3：程序输出

已知：

结点	值	左儿子	右儿子
1	8	2	3
2	3	4	5
3	10	0	6
4	1	0	0
5	6	0	0
6	14	0	0

运行：

```
cout << find(1, 6) << ' ' << find(1, 9) << endl;
```

查找 6：

$$8 \to 3 \to 6.$$

能够找到，返回 `true`，输出为 1。

查找 9：

$$8 \to 10.$$

因为 $9 < 10$，接下来应去 10 的左儿子，但左儿子为空，所以找不到，返回 `false`，输出为 0。

因此输出：

```
1 0
```

C++ 中布尔值直接输出时，`true` 通常输出 1，`false` 输出 0。

插入

向二叉搜索树中插入一个新值时，也从根开始比较：

- 若新值小于当前结点值，进入左子树；
- 若新值大于当前结点值，进入右子树；
- 直到走到空位置，把新值放在那里。

例如依次插入：

$$8,\ 3,\ 10,\ 1,\ 6,\ 14,\ 4,\ 7,\ 13.$$

得到的树为：

```
      8
    /   \
   3     10
  / \      \
 1   6      14
    / \    /
   4   7  13
```

插入顺序会影响树的形状。相同的一组数，若插入顺序不同，得到的二叉搜索树可能不同。

例题 4：按顺序插入

依次插入：

$$5,\ 2,\ 8,\ 1,\ 3,\ 7.$$

画出二叉搜索树。

步骤如下：

- 5 成为根；
- $2 < 5$，放到 5 的左边；
- $8 > 5$，放到 5 的右边；
- $1 < 5$，到 2；$1 < 2$，放到 2 的左边；
- $3 < 5$，到 2；$3 > 2$，放到 2 的右边；
- $7 > 5$，到 8；$7 < 8$，放到 8 的左边。

最终树为：

```
      5
    /   \
   2     8
  / \   /
 1   3 7
```

这类题一定要按插入顺序逐个处理，不能先把数排好再画一棵“看起来平衡”的树。

插入程序

插入可以用递归写。若 `u` 是引用，空位置可以直接被修改。

```
void insert(int &u, int x) {
    if (u == 0) {
        u = ++tot;
        val[u] = x;
        lch[u] = rch[u] = 0;
        return;
    }
    if (x < val[u]) insert(lch[u], x);
    else if (x > val[u]) insert(rch[u], x);
}
```

这里默认不插入重复值。若 `x == val[u]`，程序什么也不做。

若题目要求允许重复值，就必须规定重复值放左边还是右边，或者用 `cnt[u]` 记录出现次数。

例题 5：重复值如何处理

下面插入程序：

```
void insert(int &u, int x) {
    if (u == 0) {
        u = ++tot;
        val[u] = x;
```

```
        return;
    }
    if (x < val[u]) insert(lch[u], x);
    if (x > val[u]) insert(rch[u], x);
}
```

依次插入：

$$5,\ 3,\ 5,\ 7.$$

第二个 5 与根结点 5 相等。程序中只有小于时去左边，大于时去右边，相等时不会进入任何分支，所以不会插入第二个 5。

最终树只有 $5, 3, 7$ 三个值：

```
  5
 / \
3   7
```

如果题目说要统计每个值出现次数，这段程序就不够，需要在相等时增加计数。

中序遍历

二叉搜索树最重要的性质之一：中序遍历会得到从小到大的序列。

原因是中序遍历顺序为：

$$\text{左子树} \to \text{根} \to \text{右子树}.$$

而二叉搜索树中，左子树所有值都小于根，右子树所有值都大于根。因此中序遍历正好按从小到大访问所有结点。

例如：

```
       8
     /   \
    3     10
   / \      \
  1   6      14
     / \    /
    4   7  13
```

中序遍历为：

$$1,\ 3,\ 4,\ 6,\ 7,\ 8,\ 10,\ 13,\ 14.$$

这是一个升序序列。

例题 6：由树写中序

给定二叉搜索树：

```
      5
    /   \
   2     8
  / \   /
 1   3 7
```

中序遍历为左、根、右：

左子树得到：

$$1,\ 2,\ 3.$$

访问根：

$$5.$$

右子树得到：

$$7,\ 8.$$

所以完整中序为：

$$1,\ 2,\ 3,\ 5,\ 7,\ 8.$$

如果中序遍历结果不是升序，就说明这棵树不是合法二叉搜索树，或者遍历顺序写错了。

前驱和后继

在有序序列中，一个值的前驱是小于它的最大值，后继是大于它的最小值。

在二叉搜索树中：

- 一个结点的前驱，常是它左子树中最大的结点；
- 一个结点的后继，常是它右子树中最小的结点。

如果没有左子树，前驱可能在祖先中；如果没有右子树，后继可能在祖先中。本章重点掌握最常见的删除用法：用右子树最小值或左子树最大值替换当前结点。

例题 7：找后继

在下面的二叉搜索树中，结点 8 的后继是谁？

```
      8
    /   \
   3     10
  / \      \
 1   6      14
    / \    /
   4   7  13
```

结点 8 有右子树。它的后继是右子树中最小的值。

右子树为：

```
10
  \
   14
  /
 13
```

从右子树根 10 开始，一直往左走。10 没有左儿子，所以最小值就是 10。因此 8 的后继是 10。

结点 10 的后继则是它右子树中最小的值。右子树根为 14，14 的左儿子为 13，所以 10 的后继是 13。

删除

二叉搜索树删除结点时，要保持搜索性质。常见分三种情况。

第一，删除叶子结点。直接删掉即可。

第二，删除只有一个儿子的结点。用它的儿子顶替它的位置。

第三，删除有两个儿子的结点。常用它的后继或前驱替换它，再删除那个后继或前驱。

例题 8：删除叶子结点

树为：

```
      5
    /   \
   2     8
  / \   /
 1   3 7
```

删除 3。

3 是叶子结点，直接删除：

```
      5
    /   \
   2     8
  /     /
 1     7
```

二叉搜索树性质仍然成立。

例题 9：删除只有一个儿子的结点

在下面树中删除 8：

```
      5
    /   \
   2     8
  / \   /
 1   3 7
```

结点 8 只有左儿子 7。删除 8 后，用 7 顶替 8 的位置：

```
      5
    /   \
   2     7
  / \
```

```
  1   3
```

因为 $7 > 5$，放在根的右侧仍然合法。

例题 10：删除有两个儿子的结点

删除下面树中的 5：

```
      5
    /   \
   2     8
  / \   / \
 1   3 7   9
```

结点 5 有两个儿子。可以用右子树中的最小值，也就是 7，替换 5。

先把根值改成 7：

```
      7
    /   \
   2     8
  / \   / \
 1   3 7   9
```

再删除右子树中原来的 7。那个 7 是叶子，删除后得到：

```
      7
    /   \
   2     8
  / \     \
 1   3     9
```

这样仍然是一棵二叉搜索树。

也可以用左子树最大值 3 替换根，再删除原来的 3。两种方法都可以，但程序题要看具体代码使用哪一种。

删除程序的阅读

删除程序比查找、插入更容易出错。第一轮中，如果出现删除代码，通常不会要求写完整复杂实现，但会要求判断某些分支的含义。

下面是一种递归删除写法，默认没有重复值：

```
void erase(int &u, int x) {
    if (u == 0) return;

    if (x < val[u]) {
        erase(lch[u], x);
    } else if (x > val[u]) {
        erase(rch[u], x);
    } else {
        if (lch[u] == 0 && rch[u] == 0) {
            u = 0;
        } else if (lch[u] == 0) {
            u = rch[u];
        } else if (rch[u] == 0) {
            u = lch[u];
```

```
        } else {
            int p = rch[u];
            while (lch[p] != 0) p = lch[p];
            val[u] = val[p];
            erase(rch[u], val[p]);
        }
    }
}
```

最后一种情况是有两个儿子。程序先找到右子树中最小的结点 `p`，用 `val[p]` 替换当前结点值，再从右子树中删除原来的 `val[p]`。

读这段程序时，不要只看 `val[u] = val[p]`。真正删除的是后面那句：

```
erase(rch[u], val[p]);
```

若只替换值不删除原结点，树中会出现重复值。

高度与复杂度

二叉搜索树查找、插入、删除的时间，取决于树的高度。

若树比较平衡，高度约为 $O(\log n)$，操作就比较快。

例如：

```
      4
    /   \
   2     6
  / \   / \
 1   3 5   7
```

查找任意值只需走很短路径。

但若插入序列已经有序，例如依次插入：

$$1,\ 2,\ 3,\ 4,\ 5,$$

普通二叉搜索树会变成：

```
1
 \
  2
   \
    3
     \
      4
       \
        5
```

这时高度为 5，查找 5 要走 5 个结点。操作退化为 $O(n)$。

例题 11：插入顺序影响高度

依次插入 $1, 2, 3, 4$，得到的二叉搜索树为：

```
1
 \
  2
   \
    3
     \
      4
```

高度为 4。

依次插入 $2, 1, 3, 4$，得到：

```
   2
  / \
 1   3
      \
       4
```

高度为 3。

同一组数，插入顺序不同，树高可能不同，查找效率也可能不同。

本书不讲平衡树，但要知道普通二叉搜索树最坏情况下会退化成链。

验证二叉搜索树

判断一棵二叉树是否是二叉搜索树，有两种常见方法。

第一种方法：中序遍历，看得到的序列是否严格递增。

若不允许重复值，则中序序列必须严格递增。若允许重复值，还要看重复值被规定放在哪一侧。

第二种方法：递归维护值域范围。每个结点都必须落在祖先结点限制出的区间内。

只检查父子关系是不够的，前面例题已经展示过。

例题 12：中序验证

判断下面二叉树是否是二叉搜索树。

```
      6
    /   \
   3     8
  / \   /
 2   5 7
```

中序遍历为：

$$2,\ 3,\ 5,\ 6,\ 7,\ 8.$$

序列严格递增，所以这棵树是二叉搜索树。

再看：

```
      6
    /   \
   3     8
  / \   /
```

```
  2   7 5
```

中序遍历为：

$$2,\ 3,\ 7,\ 6,\ 5,\ 8.$$

不是递增序列，所以不是二叉搜索树。

由序列构造二叉搜索树

如果给出一串插入序列，可以唯一确定一棵普通二叉搜索树。方法是按序列依次插入。

例题 13：插入序列和前序

依次插入：

$$4,\ 2,\ 6,\ 1,\ 3,\ 5,\ 7.$$

得到：

```
      4
    /   \
   2     6
  / \   / \
 1   3 5   7
```

这棵树的前序遍历为：

$$4,\ 2,\ 1,\ 3,\ 6,\ 5,\ 7.$$

中序遍历为：

$$1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7.$$

注意：插入序列本身不一定等于前序遍历。这个例子中刚好相同，是因为插入顺序先根、再左子树、再右子树。换一个插入序列，未必相同。

变式：不同插入序列可能得到同一棵树

下面两个插入序列：

$$4,\ 2,\ 6,\ 1,\ 3,\ 5,\ 7$$

和：

$$4,\ 6,\ 2,\ 7,\ 5,\ 3,\ 1$$

都可以得到同一棵树：

```
      4
    /   \
   2     6
  / \   / \
 1   3 5   7
```

因为根 4 先插入后，左子树中的元素和右子树中的元素只要最终按各自规则插入，就可能形成相同结构。

第一轮中若问 “给定插入序列得到什么树”，按序列插入即可；若问 “某棵树可能由哪些序列插入得到”，就要注意序列不唯一。

二叉搜索树和二分

二叉搜索树查找与二分类似：每次比较后，只进入左子树或右子树的一边。

但它和数组二分不同。

数组二分依赖有序数组，范围每次大致减半，所以复杂度是 $O(\log n)$。

普通二叉搜索树若不平衡，左右子树大小可能差很多，每次并不一定减半，最坏可能退化成 $O(n)$。

例如链状树：

```
1
 \
  2
   \
    3
     \
      4
```

查找 4 时，要依次经过所有结点。虽然每次也只往一个方向走，但并没有把规模减半。

因此不能看到 “二叉搜索树查找” 就一律写 $O(\log n)$。要看树高。通常说法是：

$$O(h)$$

其中 h 是树高。平衡时 $h = O(\log n)$，最坏时 $h = O(n)$。

和堆的区别

二叉搜索树和堆都可以用二叉树表示，但性质完全不同。

结构	核心性质	堆顶或根的含义	中序遍历
二叉搜索树	左小右大	根不一定是最大或最小	得到有序序列
大根堆	父结点不小于子结点	根是最大值	不保证有序
小根堆	父结点不大于子结点	根是最小值	不保证有序

例如：

```
    8
  /   \
 3     10
```

这是一棵二叉搜索树，因为左边小、右边大。但它不是大根堆，因为根 8 小于右儿子 10。

再看：

```
      10
    /    \
   8      9
```

这是大根堆，但不是二叉搜索树，因为右儿子 9 小于根 10，不满足右子树大于根。

不要把“二叉树上的大小关系”混为一谈。堆只管父子关系，二叉搜索树管整棵左子树和右子树的范围。

程序阅读中的二叉搜索树

读二叉搜索树程序，可以按下面顺序。

第一，确认是否允许重复值。相等时是忽略、计数，还是放到某一侧。

第二，确认查找或插入的比较方向。小于去左，大于去右；若代码相反，树的规则也可能相反。

第三，判断根结点。插入序列第一个值通常成为根。

第四，追踪路径。查找和插入都只走一条从根到某个位置的路径。

第五，判断中序遍历。若树是二叉搜索树，中序通常给升序序列。

第六，分析复杂度。看树高，而不是只看结点数。

例题 14：程序输出

```cpp
int root = 0, tot = 0;
int val[20], lch[20], rch[20];

void insert(int &u, int x) {
    if (u == 0) {
        u = ++tot;
        val[u] = x;
        return;
    }
    if (x < val[u]) insert(lch[u], x);
    else if (x > val[u]) insert(rch[u], x);
}

void dfs(int u) {
    if (u == 0) return;
    dfs(lch[u]);
    cout << val[u] << ' ';
    dfs(rch[u]);
}

int main() {
    int a[6] = {0, 5, 2, 8, 2, 3};
    for (int i = 1; i <= 5; i++) {
        insert(root, a[i]);
    }
    dfs(root);
    return 0;
}
```

插入序列为：

$$5,\ 2,\ 8,\ 2,\ 3.$$

第二个 2 与已有结点 2 相等，程序不插入重复值。

最终树为：

```
   5
  / \
 2   8
  \
   3
```

`dfs` 是中序遍历，所以输出升序序列：

```
2 3 5 8
```

这里没有第二个 2，因为相等时没有插入。

完善程序中的二叉搜索树

二叉搜索树完善程序常见空缺包括：

- 查找时向左或向右递归；
- 插入新结点时初始化左右儿子；
- 相等时计数或忽略；
- 中序遍历输出；
- 删除时找右子树最小值或左子树最大值；
- 验证二叉搜索树时维护上下界；
- 复杂度判断中使用树高。

例题 15：补全查找

```
bool find(int u, int x) {
    if (u == 0) return false;
    if (val[u] == x) return true;
    if (x < val[u]) return _____;
    return _____;
}
```

二叉搜索树中，若 x 小于当前值，应去左子树；若大于当前值，应去右子树。

所以空缺为：

```
find(lch[u], x)
find(rch[u], x)
```

例题 16：补全插入

```
void insert(int &u, int x) {
    if (u == 0) {
        u = ++tot;
        val[u] = x;
        lch[u] = rch[u] = 0;
        return;
    }
    if (x < val[u]) _____;
    else if (x > val[u]) _____;
```

```
}
```

小于当前值去左子树，大于当前值去右子树，所以填：

```
insert(lch[u], x)
insert(rch[u], x)
```

注意这里参数是引用 `int &u`，因此递归到空儿子位置时，可以直接修改对应儿子编号。

例题 17：补全中序输出

```
void dfs(int u) {
    if (u == 0) return;
    dfs(lch[u]);
    _____;
    dfs(rch[u]);
}
```

若要按从小到大输出二叉搜索树中的所有值，空缺应填：

```
cout << val[u] << ' '
```

因为中序遍历二叉搜索树得到升序序列。

常见错误

二叉搜索树题的错误集中在下面几类。

错误	说明
只看父子大小	必须保证整棵左子树和右子树都满足范围
忘记插入顺序	相同集合不同插入顺序可能得到不同树
把中序写成前序	升序性质来自中序，不是任意遍历
重复值处理不清	相等时忽略、计数、放左或放右要看代码
删除只替换不删除	用后继替换后，还要删掉原后继结点
查找复杂度误判	普通二叉搜索树最坏是 $O(n)$
与堆混淆	堆顶最大或最小，二叉搜索树根不一定极值
验证时不看祖先限制	右孙子、左孙子也要符合祖先范围
`operator` 方向看错	`<` 去左，`>` 去右，代码若相反规则也变
空结点标记忽略	递归边界通常是 `u==0`

做二叉搜索树题时，先抓住一句话：左子树小，右子树大。再结合中序遍历、插入顺序和查找路径，题目就会清楚很多。

原创例题

例题 18：插入、查找与遍历

依次插入：

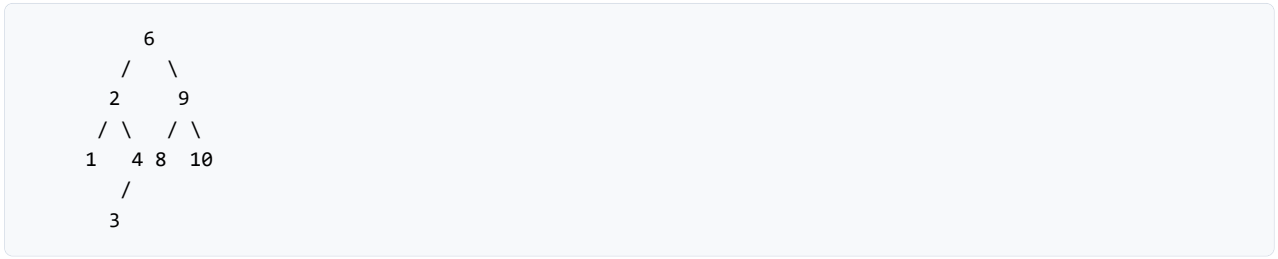
$$6,\ 2,\ 9,\ 1,\ 4,\ 8,\ 10,\ 3.$$

得到怎样的二叉搜索树？中序遍历是什么？查找 3 的路径是什么？

插入过程：

- 6 为根；
- $2 < 6$，在左边；
- $9 > 6$，在右边；
- $1 < 6$，到 2，$1 < 2$，放左边；
- $4 < 6$，到 2，$4 > 2$，放右边；
- $8 > 6$，到 9，$8 < 9$，放左边；
- $10 > 6$，到 9，$10 > 9$，放右边；
- $3 < 6$，到 2，$3 > 2$，到 4，$3 < 4$，放左边。

树为：



中序遍历为：

$$1,\ 2,\ 3,\ 4,\ 6,\ 8,\ 9,\ 10.$$

查找 3 的路径为：

$$6 \to 2 \to 4 \to 3.$$

变式 1：查找不存在的数

在上面的树中查找 5。

路径为：

$$6 \to 2 \to 4.$$

因为 $5 > 4$，应去 4 的右子树，但右子树为空，所以查找失败。

变式 2：删除根结点

若删除根结点 6，并使用右子树最小值替换。

右子树为：



其中最小值为 8。用 8 替换根，再删除原来的 8，得到：

```
      8
    /   \
   2     9
  / \     \
 1   4     10
    /
   3
```

中序遍历仍为：

$$1,\ 2,\ 3,\ 4,\ 8,\ 9,\ 10.$$

删除后仍要保持二叉搜索树性质。

小结

二叉搜索树把二叉树的结构和大小关系结合起来。它的查找路径很像二分，但是否高效取决于树高。中序遍历得到升序序列，是本章最重要的结论之一。下一章讲 Trie，它也用于查找，但不按数值大小分左右，而是按字符或数位逐层向下走。

0x16 Trie

Trie 是一种适合存储字符串集合的树结构。它和二叉搜索树不同：二叉搜索树按大小关系往左或往右走，Trie 按字符一层一层往下走。

如果要保存很多单词，例如：

$$\text{an},\ \text{and},\ \text{ant},\ \text{boy}$$

这些字符串中有些前缀相同。Trie 会把相同前缀共用起来。这样，查找一个字符串时，只需要按它的字符顺序沿边往下走；如果某一步走不下去，就说明这个字符串不在集合中。

CSP-S 第一轮中，Trie 常以程序阅读和完善程序形式出现：判断插入若干字符串后有多少结点，判断某个字符串是否存在，统计有多少字符串具有某个前缀，补全 `insert`、`find` 或数组下标。字符串哈希会在第 36 章讲，本章只讲 Trie 本身。

按字符建树

Trie 的根结点通常表示空前缀。从根出发，每经过一条边，就多读入一个字符。

例如插入：

$$\text{an},\ \text{and},\ \text{ant},\ \text{boy}$$

可以得到下面的结构：

```
(root)
 ├─ a
 │  └─ n
 │     ├─ d
 │     └─ t
 └─ b
    └─ o
       └─ y
```

字符串 `an` 对应路径：

$$root \to a \to n.$$

字符串 `and` 对应路径：

$$root \to a \to n \to d.$$

字符串 `ant` 对应路径：

$$root \to a \to n \to t.$$

an、and、ant 共用了前缀 an。这正是 Trie 的主要特点。

要注意：Trie 的结点通常不直接存整个字符串。一个结点代表从根走到它时得到的前缀。某个字符串是否在集合中，还需要在对应结点上做结束标记。

结束标记

看下面的 Trie：

```
(root)
 └ a
   └ n
     └ d
```

它至少表示路径 a、an、and 都存在。但这不代表三个字符串都被插入过。

如果只插入了 and，那么 a 和 an 只是路径上的前缀，不是完整字符串。

因此，Trie 结点上常有一个结束标记：

```
bool end[N];
```

或出现次数：

```
int cnt[N];
```

当插入一个字符串结束时，在最后一个结点上标记：

```
cnt[u]++;
```

若 `cnt[u] > 0`，表示有字符串在这个结点结束。

例题 1：前缀不等于单词

依次插入字符串：

and, ant.

问 an 是否一定在集合中？

不是。

插入 and 和 ant 时，路径 a -> n 会存在，但如果没有在结点 n 上打结束标记，就说明 an 只是前缀，不是被插入的完整字符串。

如果查询“是否存在前缀 an”，答案是存在；如果查询“字符串 an 是否被插入过”，答案是否定的。

Trie 题中要分清“前缀存在”和“完整字符串存在”。

数组存储

若只处理小写英文字母，可以用数组保存儿子：

```
int ch[N][26];
int cnt[N];
int tot;
```

含义如下。

变量	含义
`ch[u][c]`	结点 `u` 沿字符 `c` 走到的儿子
`cnt[u]`	有多少字符串在结点 `u` 结束
`tot`	已经创建的结点数量
根结点	通常编号为 0 或 1

如果根结点编号为 0，初始时：

```
tot = 0;
```

新建结点时：

```
ch[u][c] = ++tot;
```

如果根结点编号为 1，初始时：

```
tot = 1;
```

新建结点时同样使用 `++tot`。

两种写法都可以。程序阅读题中，必须先看根结点编号是多少。根为 0 和根为 1 时，结点总数的计算会差 1。

字符下标

对于小写字母：

```
int c = s[i] - 'a';
```

这样：

字符	下标
`'a'`	0
`'b'`	1
`'c'`	2
…	…
`'z'`	25

如果处理大写字母，就应使用：

```
int c = s[i] - 'A';
```

如果题目中字符集不是英文字母，例如数字字符 `'0'` 到 `'9'`，则应按对应字符集转换。不要把所有 Trie 都默认写成 26 个儿子。

插入

插入字符串时，从根开始，按字符逐个向下走。

若当前字符对应的儿子不存在，就新建结点；若已经存在，就沿已有边继续走。最后在字符串结束位置打标记。

典型代码如下：

```
void insert(string s) {
    int u = 0;
    for (int i = 0; i < s.size(); i++) {
        int c = s[i] - 'a';
        if (ch[u][c] == 0) ch[u][c] = ++tot;
        u = ch[u][c];
    }
    cnt[u]++;
}
```

这里根结点编号为 0。数组 ch 初始全为 0，表示没有对应儿子。新建结点编号从 1 开始。

例题 2：插入过程

依次插入：

$$\text{an},\ \text{and},\ \text{ant}.$$

根结点记为 root。

插入 an：

$$root \to a \to n.$$

创建两个新结点：a 和 an。

插入 and：

前缀 an 已经存在，只需要在 n 后面创建 d：

$$root \to a \to n \to d.$$

再创建一个新结点。

插入 ant：

前缀 an 仍然存在，只需要在 n 后面创建 t：

$$root \to a \to n \to t.$$

再创建一个新结点。

若根结点不计入新建数量，一共创建 4 个非根结点，分别对应前缀：

$$\text{a},\ \text{an},\ \text{and},\ \text{ant}.$$

若把根也计入 Trie 结点数，则总共有 5 个结点。

计数题中要看清题目是否把根结点算入总数。

查找完整字符串

查找字符串时，也从根开始按字符往下走。

若某个字符对应的儿子不存在，查找失败。若所有字符都走完，还要检查结束标记。

```
bool find_word(string s) {
    int u = 0;
    for (int i = 0; i < s.size(); i++) {
        int c = s[i] - 'a';
        if (ch[u][c] == 0) return false;
        u = ch[u][c];
    }
    return cnt[u] > 0;
}
```

这里返回 `cnt[u] > 0`，表示这个完整字符串曾经被插入过。

例题 3：查找完整字符串

插入：

and, ant.

查询 `and`：

路径存在，并且 `d` 对应结点有结束标记，所以查询成功。

查询 `an`：

路径 `a -> n` 存在，但 `n` 结点没有结束标记，所以完整字符串 `an` 没有被插入过，查询失败。

查询 `any`：

路径 `a -> n` 存在，但从 `n` 沿字符 `y` 走不下去，查询失败。

这三种失败原因不同：一种是缺少结束标记，一种是路径不存在。程序阅读题中要分清。

查找前缀

若题目问“是否存在某个字符串以 `p` 为前缀”，只需要看前缀路径是否存在，不需要检查结束标记。

```
bool find_prefix(string s) {
    int u = 0;
    for (int i = 0; i < s.size(); i++) {
        int c = s[i] - 'a';
        if (ch[u][c] == 0) return false;
        u = ch[u][c];
    }
    return true;
}
```

如果能走完这个前缀，说明 Trie 中至少有某条路径包含它。至于这个前缀本身是不是完整字符串，不影响前缀查询。

例题 4：完整查找和前缀查找

插入：

apple, app, bat.

查询结果如下。

查询	完整字符串查找	前缀查找
app	成功	成功
ap	失败	成功
apple	成功	成功
apply	失败	失败
ba	失败	成功

ap 和 ba 都只是前缀，不是完整字符串。若使用 find_word，它们失败；若使用 find_prefix，它们成功。

统计前缀数量

有时题目要求：已经插入若干字符串，查询有多少字符串以某个前缀开头。

这时可以在每个结点记录经过它的字符串数量：

```
int pass[N];
```

插入时，每走到一个结点，就让它的 pass 加一：

```
void insert(string s) {
    int u = 0;
    for (int i = 0; i < s.size(); i++) {
        int c = s[i] - 'a';
        if (ch[u][c] == 0) ch[u][c] = ++tot;
        u = ch[u][c];
        pass[u]++;
    }
    cnt[u]++;
}
```

查询前缀 p 时，先走到 p 对应的结点。如果路径不存在，答案为 0；否则答案为 pass[u]。

例题 5：前缀统计

插入：

$$\text{app},\ \text{apple},\ \text{ape},\ \text{bat}.$$

统计各前缀数量：

前缀	以它开头的字符串	数量
a	app,apple,ape	3
ap	app,apple,ape	3
app	app,apple	2
apple	apple	1
b	bat	1
ba	bat	1
cat	无	0

这里 cnt[u] 和 pass[u] 含义不同。cnt[u] 只统计有多少字符串在结点 u 结束；pass[u] 统计有多少字符串经过结点 u。

若查询前缀数量，应看 pass；若查询完整字符串出现次数，应看 cnt。

统计完整字符串次数

如果允许重复插入同一个字符串，`cnt[u]` 可以记录它出现了多少次。

例如插入：

$$\text{cat},\ \text{car},\ \text{cat}.$$

则：

- `cnt[cat] = 2`；
- `cnt[car] = 1`；
- `cnt[ca] = 0`，除非也插入过 `ca`。

例题 6：重复插入

执行：

```
insert("to");
insert("tea");
insert("to");
insert("ten");
```

查询完整字符串次数：

字符串	次数
to	2
tea	1
ten	1
te	0

如果查询前缀 `te` 的通过数量，则为 2，因为 `tea` 和 `ten` 都以 `te` 开头。

同一个结点上可能同时有 `cnt` 和 `pass`。做题时先看题目问的是“这个词出现几次”，还是“这个前缀下面有几个词”。

结点数量

Trie 的结点数量常出现在选择题中。插入字符串时，只有不存在的前缀才会创建新结点。

若根结点不计入，Trie 中非根结点数量等于所有不同非空前缀的数量。

例题 7：计算结点数

插入字符串：

$$\text{a},\ \text{ab},\ \text{abc},\ \text{b}.$$

列出不同非空前缀：

- `a` 的前缀：`a`；
- `ab` 的前缀：`a,ab`；
- `abc` 的前缀：`a,ab,abc`；
- `b` 的前缀：`b`。

不同非空前缀为：

$$a,\ ab,\ abc,\ b.$$

共 4 个。因此若不计根结点，Trie 有 4 个结点；若计根结点，则有 5 个结点。

例题 8：共享前缀

插入：

$$abc,\ abd,\ bcd.$$

不同非空前缀包括：

$$a,\ ab,\ abc,\ abd,\ b,\ bc,\ bcd.$$

共 7 个非根结点。加上根结点，总数为 8。

`abc` 和 `abd` 共享前缀 `a`、`ab`，所以不会重复建这些结点。

结点数量题可以逐个插入追踪，也可以列不同前缀。字符串较少时，列前缀更稳。

字典序输出

如果 Trie 的儿子按字符顺序遍历，就可以按字典序输出所有字符串。

对于小写字母，按 `0` 到 `25` 遍历儿子，相当于按 `'a'` 到 `'z'` 遍历。

```
void dfs(int u) {
    if (cnt[u] > 0) {
        cout << 当前路径 << endl;
    }
    for (int c = 0; c < 26; c++) {
        if (ch[u][c]) {
            加入字符 char('a' + c);
            dfs(ch[u][c]);
            删除最后一个字符;
        }
    }
}
```

这段程序的含义是：先输出当前结点对应的完整字符串，再按字符从小到大进入子树。

例题 9：字典序

插入：

$$bat,\ app,\ apple,\ ape.$$

按字典序输出应为：

$$ape,\ app,\ apple,\ bat.$$

为什么 app 在 apple 前面？因为 app 是 apple 的前缀，并且 app 本身是一个完整字符串。字典序中，较短的前缀字符串排在它的延长字符串前面。

如果 DFS 中先遍历儿子，再输出当前结点，则 apple 可能会出现在 app 前面，顺序就错了。输出位置会影响结果。

字典树和排序

Trie 可以按字典序输出字符串，但它不是普通排序算法。它依赖字符集大小和字符串总长度。

若所有字符串总长度为 L，插入它们通常需要 $O(L)$。字典序遍历也与结点数和字符集有关。对于第一轮，掌握“按字符顺序遍历儿子可以得到字典序” 即可。

如果只是少量字符串排序，直接用字符串排序更简单。Trie 更适合大量前缀查询、字符串集合查找、统计前缀数量等任务。

01 Trie

除了字符串，Trie 也可以按二进制位存储整数。这种结构常叫 01 Trie。

对于一个非负整数，把它写成固定长度的二进制表示。从高位到低位插入，每一层只有两个儿子：0 和 1。

```
int ch[N][2];
```

01 Trie 常用于处理按二进制位查找的问题，例如找某个数与集合中某个数的异或最大值。异或相关内容不是本章重点，但第一轮程序阅读中可能会出现按位走 Trie 的代码。

例题 10：按二进制插入

假设只看 3 位二进制。插入整数 5。

5 的三位二进制为：

$$101_2.$$

插入路径为：

$$root \to 1 \to 0 \to 1.$$

插入整数 4，三位二进制为：

$$100_2.$$

路径为：

$$root \to 1 \to 0 \to 0.$$

它们共享前两层 `1 -> 0`，只在最后一位分开。

这和字符串 Trie 很接近。字符串 Trie 按字符走，01 Trie 按二进制位走。

查找最大异或的基本想法

对于两个二进制位，异或规则为：

位 a	位 b	$a \operatorname{xor} b$
0	0	0
0	1	1
1	0	1
1	1	0

若要让异或结果尽量大，应优先让高位变成 1。因此，在 01 Trie 中查找与 x 异或最大的数时，从高位到低位走。当前位为 0，优先找儿子 1；当前位为 1，优先找儿子 0。如果相反儿子不存在，再走相同儿子。

本章只讲这个读代码需要的判断，不展开成完整专题。

例题 11：一位选择

在某一位上，当前数 x 的位为 0。为了让异或结果这一位为 1，应选择集合中数的这一位为 1。

若当前位为 1，则应优先选择对方位为 0。

所以查找时常见代码是：

```
int t = bit ^ 1;
if (ch[u][t]) u = ch[u][t];
else u = ch[u][bit];
```

这里 `bit ^ 1` 表示相反位。若相反位存在，就优先走相反位。

位运算的系统内容不是本章重点，遇到这类程序时，只要知道它在优先制造异或高位上的 1 即可。

Trie 的复杂度

Trie 的插入和查询复杂度通常与字符串长度有关。

若字符串长度为 m，插入需要沿着 m 个字符走一遍，复杂度为 $O(m)$。查找完整字符串或前缀也一样是 $O(m)$。

如果插入很多字符串，总复杂度通常与总字符数成正比。

空间方面，最直接的数组写法是：

```
int ch[N][26];
```

其中 N 是结点数量上限。结点数量最多接近所有字符串长度之和再加根结点。若字符串总长度为 L，结点数通常不超过 $L+1$。

例题 12：复杂度判断

有 n 个字符串，总长度为 L。把它们全部插入 Trie，复杂度是多少？

每个字符在插入时被处理一次。若某条边已经存在，就沿着它走；若不存在，就创建新结点。无论哪种情况，每个字符对应一次常数级处理。

所以总时间复杂度为：

$$O(L).$$

如果每个字符串长度都不超过 m，则 $L \le nm$，也可以写成 $O(nm)$。

程序阅读中的 Trie

读 Trie 程序，可以按下面顺序。

第一，确认根编号。根是 0 还是 1，会影响结点数。

第二，确认字符集。`26` 表示小写字母，`10` 表示数字字符，`2` 表示二进制位。

第三，确认儿子数组含义。`ch[u][c]` 通常表示结点 `u` 沿字符 `c` 的儿子。

第四，区分 `cnt` 和 `pass`。`cnt` 表示字符串在此结束，`pass` 表示多少字符串经过此结点。

第五，判断查询类型。是查完整字符串，还是查前缀。

第六，注意输出顺序。DFS 中输出语句位置和儿子遍历顺序会影响结果。

例题 13：程序输出

```
int ch[100][26], cnt[100], tot = 0;

void insert(string s) {
    int u = 0;
    for (int i = 0; i < s.size(); i++) {
        int c = s[i] - 'a';
        if (!ch[u][c]) ch[u][c] = ++tot;
        u = ch[u][c];
    }
    cnt[u]++;
}

int query(string s) {
    int u = 0;
    for (int i = 0; i < s.size(); i++) {
        int c = s[i] - 'a';
        if (!ch[u][c]) return 0;
        u = ch[u][c];
    }
    return cnt[u];
}

int main() {
    insert("ab");
    insert("abc");
    insert("ab");
    cout << query("a") << ' ' << query("ab") << ' ' << query("abc") << endl;
    return 0;
}
```

插入了两次 `ab`，一次 `abc`。

查询 `a`：路径存在，但没有字符串在 `a` 结点结束，所以 `cnt=0`。

查询 `ab`：插入过两次，所以 `cnt=2`。

查询 `abc`：插入过一次，所以 `cnt=1`。

输出为：

```
0 2 1
```

这里 query 查询的是完整字符串出现次数，不是前缀数量。

例题 14：改成前缀统计

若插入函数改成：

```
void insert(string s) {
    int u = 0;
    for (int i = 0; i < s.size(); i++) {
        int c = s[i] - 'a';
        if (!ch[u][c]) ch[u][c] = ++tot;
        u = ch[u][c];
        pass[u]++;
    }
    cnt[u]++;
}
```

并且查询返回 pass[u]，那么对同样的插入：

```
insert("ab");
insert("abc");
insert("ab");
```

前缀 a 的通过次数为 3，前缀 ab 的通过次数为 3，前缀 abc 的通过次数为 1。

此时查询结果会变成：

```
3 3 1
```

同一棵 Trie，查询 cnt 和查询 pass 的含义完全不同。

完善程序中的 Trie

Trie 完善程序常见空缺包括：

- 字符转下标；
- 不存在儿子时新建结点；
- 沿儿子向下移动；
- 字符串结束时增加 cnt；
- 前缀统计时增加 pass；
- 查询失败时返回 0 或 false；
- 字典序 DFS 中儿子遍历顺序。

例题 15：补全插入

```
void insert(string s) {
    int u = 0;
    for (int i = 0; i < s.size(); i++) {
        int c = _____;
        if (ch[u][c] == 0) ch[u][c] = _____;
        u = _____;
    }
    cnt[u]++;
}
```

若字符串只含小写字母，则第一空为：

```
s[i] - 'a'
```

不存在儿子时，新建结点：

```
++tot
```

沿当前字符向下走：

```
ch[u][c]
```

完整关键部分为：

```
int c = s[i] - 'a';
if (ch[u][c] == 0) ch[u][c] = ++tot;
u = ch[u][c];
```

例题 16：补全完整字符串查询

```
bool find_word(string s) {
    int u = 0;
    for (int i = 0; i < s.size(); i++) {
        int c = s[i] - 'a';
        if (_____) return false;
        u = ch[u][c];
    }
    return _____;
}
```

若当前字符没有对应儿子，路径不存在，应返回失败：

```
ch[u][c] == 0
```

走完整个字符串后，还要确认它是完整字符串，而不只是前缀：

```
cnt[u] > 0
```

所以两个空分别为：

```
ch[u][c] == 0
cnt[u] > 0
```

例题 17：补全前缀查询

```
bool find_prefix(string s) {
    int u = 0;
    for (int i = 0; i < s.size(); i++) {
        int c = s[i] - 'a';
        if (ch[u][c] == 0) return false;
        u = ch[u][c];
    }
    return _____;
}
```

前缀查询只需要路径存在。走完所有字符后，可以直接返回：

```
true
```

不能写成 `cnt[u] > 0`。那会把“前缀存在”误判成“完整字符串存在”。

常见错误

Trie 题的错误集中在下面几类。

错误	说明
前缀和完整字符串混淆	路径存在不代表完整字符串存在
忘记结束标记	插入结束时没有更新 `cnt`
`cnt` 和 `pass` 混用	一个看结束，一个看经过
根编号看错	根为 0 或 1 会影响结点数量
字符下标错	小写字母应减 `'a'`，大写应减 `'A'`
字符集大小错	数字字符、二进制位、小写字母对应大小不同
查询时漏判空儿子	走到不存在的边应失败
字典序输出位置错	前缀字符串应在它的延长字符串前输出
结点数重复计算	共享前缀只建一次
01 Trie 位序错	固定长度整数通常从高位往低位走

做 Trie 题时，先判断查询目标：完整字符串、前缀存在，还是前缀数量。目标不同，最后返回的变量就不同。

原创例题

例题 18：插入与查询

依次插入：

$$\text{he, her, hero, hi, he.}$$

回答下面查询：

1. 完整字符串 `he` 出现几次；
2. 完整字符串 `h` 是否出现；
3. 有多少字符串以前缀 `he` 开头；
4. 有多少字符串以前缀 `hero` 开头；
5. 有多少字符串以前缀 `ha` 开头。

分析：

插入的完整字符串中，`he` 出现 2 次，所以第 1 问答案为 2。

`h` 是所有这些字符串的前缀，但没有作为完整字符串插入，所以第 2 问答案为否。

以前缀 `he` 开头的字符串有：

$$\text{he, her, hero, he.}$$

注意 `he` 插入了两次，所以数量为 4。

以前缀 `hero` 开头的字符串只有：

hero.

数量为 1。

前缀 ha 的路径不存在，数量为 0。

变式 1：只统计不同字符串

如果题目要求统计“不同字符串”中有多少个以前缀 he 开头，那么 he 的重复插入只算一次。不同字符串为：

he, her, hero, hi.

以前缀 he 开头的不同字符串有：

he, her, hero.

答案为 3。

这时插入时不能简单让 pass 每次都加一，或者要先判断这个完整字符串是否已经出现过。题目是否计算重复，需要看清。

变式 2：结点数量

插入不同字符串：

he, her, hero, hi.

不同非空前缀为：

h, he, her, hero, hi.

所以非根结点数为 5。若计入根结点，总结点数为 6。

小结

Trie 的核心是“按字符逐层向下走”。它不按数值大小比较，也不维护普通二叉树的左右关系。完整字符串查 cnt，前缀路径查是否能走到，前缀数量查 pass。把这三件事分清，Trie 的程序阅读和完善程序就会稳定很多。下一章进入线段树，树结构会继续出现，但结点含义会变成区间信息。

0x17 线段树

线段树是一种维护区间信息的数据结构。它把一个大区间不断分成左右两半，每个结点负责一个区间，并保存这个区间的某种信息，例如区间和、区间最大值、区间最小值。

前面讲过的堆也常用数组表示树，但堆的结点表示“一个元素”，并且根结点表示当前最大或最小值。线段树的结点表示“一个区间”，根结点通常表示整个数组区间。

CSP-S 第一轮中，线段树常以程序阅读和完善程序形式出现：给出 `build`、`query`、`update` 函数，要求判断某次查询返回值，补全左右儿子编号、区间中点、合并方式或懒标记下传。第一轮不会要求写很长的模板，但需要看懂每个结点维护的区间和递归过程。

区间信息

给定数组：

$$a[1..8] = 3,\ 1,\ 4,\ 1,\ 5,\ 9,\ 2,\ 6.$$

如果频繁询问区间和，例如：

$$a[3] + a[4] + a[5] + a[6],$$

直接累加需要逐个访问元素。若询问很多次，就会比较慢。

线段树会把区间分成很多段，并提前保存每一段的信息。例如：

结点区间	区间和
$[1, 8]$	31
$[1, 4]$	9
$[5, 8]$	22
$[1, 2]$	4
$[3, 4]$	5
$[5, 6]$	14
$[7, 8]$	8

查询 $[3, 6]$ 时，可以把它拆成 $[3, 4]$ 和 $[5, 6]$ 两段：

$$(4 + 1) + (5 + 9) = 5 + 14 = 19.$$

线段树的意义就在于：把大区间的信息拆成少量已经维护好的小区间信息。

结点表示区间

线段树中，每个结点对应一个区间 $[l, r]$。

根结点对应整个数组，例如 $[1, n]$。若当前结点对应区间 $[l, r]$，且 $l < r$，设：

$$mid = \left\lfloor \frac{l + r}{2} \right\rfloor.$$

左儿子对应：

$$[l, mid],$$

右儿子对应：

$$[mid + 1, r].$$

一直分到 $l = r$，此时结点表示单个元素。

例如 $n = 8$ 时，线段树的区间划分为：

层数	区间
1	$[1, 8]$
2	$[1, 4]$，$[5, 8]$
3	$[1, 2]$，$[3, 4]$，$[5, 6]$，$[7, 8]$
4	$[1, 1]$，$[2, 2]$，$[3, 3]$，$[4, 4]$，$[5, 5]$，$[6, 6]$，$[7, 7]$，$[8, 8]$

叶子结点表示单个位置，非叶子结点表示一段连续区间。

例题 1：判断左右区间

当前结点表示区间 $[3, 9]$，按线段树划分，它的左儿子和右儿子分别表示什么区间？

计算：

$$mid = \left\lfloor \frac{3 + 9}{2} \right\rfloor = 6.$$

所以左儿子表示：

$$[3, 6],$$

右儿子表示：

$$[7, 9].$$

注意右儿子从 $mid + 1$ 开始，不是从 mid 开始。若左右都包含 mid，区间会重叠；若左右都不包含 mid，区间会遗漏。

数组编号

线段树常用数组模拟树。若当前结点编号为 `p`，通常规定：

结点	编号
当前结点	`p`
左儿子	`p * 2`
右儿子	`p * 2 + 1`

根结点编号一般为 1。

例如：

编号	区间
1	$[1,8]$
2	$[1,4]$
3	$[5,8]$
4	$[1,2]$
5	$[3,4]$
6	$[5,6]$
7	$[7,8]$

这里的编号不是数组原下标，而是线段树结点编号。程序阅读题中，`p` 表示线段树结点，`l,r` 表示这个结点负责的原数组区间，两者不能混淆。

线段树数组通常开到 `4 * n` 左右。这是为了容纳递归划分产生的结点编号。第一轮中不需要证明这个界，只要知道 `tr[4*N]` 是线段树常见写法。

建树

建树函数负责把原数组的信息填入线段树。

以区间和为例：

```
void build(int p, int l, int r) {
    if (l == r) {
        sum[p] = a[l];
        return;
    }
    int mid = (l + r) / 2;
    build(p * 2, l, mid);
    build(p * 2 + 1, mid + 1, r);
    sum[p] = sum[p * 2] + sum[p * 2 + 1];
}
```

参数含义如下：

参数	含义
`p`	当前线段树结点编号
`l,r`	当前结点负责的原数组区间
`sum[p]`	当前区间 $[l,r]$ 的元素和

边界是 `l == r`，表示区间只包含一个元素。此时：

```
sum[p] = a[l];
```

非叶子结点要先建左右子树，再合并左右儿子信息：

```
sum[p] = sum[p * 2] + sum[p * 2 + 1];
```

这一步称为向上合并，常写成 `pushup(p)`。

例题 2：建树后的区间和

数组：

$$a[1..4] = 2,\ 7,\ 1,\ 8.$$

建区间和线段树。

叶子结点：

区间	区间和
$[1,1]$	2
$[2,2]$	7
$[3,3]$	1
$[4,4]$	8

合并：

$$sum[1,2] = 2 + 7 = 9,$$

$$sum[3,4] = 1 + 8 = 9,$$

$$sum[1,4] = 9 + 9 = 18.$$

所以根结点保存整个数组和 18。

pushup

为了让代码更清楚，常把合并写成函数：

```
void pushup(int p) {
    sum[p] = sum[p * 2] + sum[p * 2 + 1];
}
```

若维护区间最大值，合并方式会变成：

```
mx[p] = max(mx[p * 2], mx[p * 2 + 1]);
```

若维护区间最小值，合并方式会变成：

```
mn[p] = min(mn[p * 2], mn[p * 2 + 1]);
```

线段树不是只能维护区间和。它能维护什么，取决于结点保存的信息以及左右儿子如何合并。

例题 3：判断维护内容

下面代码维护的是什么？

```
void pushup(int p) {
    tr[p] = max(tr[p * 2], tr[p * 2 + 1]);
}
```

它把当前结点的值设成左右儿子的较大值。因此 `tr[p]` 表示当前区间的最大值。

如果查询函数返回若干结点的 `tr` 并取最大值，那么整个线段树用于区间最大值查询。

区间查询

区间查询是线段树的核心操作。以区间和为例，查询 $[L, R]$ 的和。

函数写法常见如下：

```
long long query(int p, int l, int r, int L, int R) {
    if (L <= l && r <= R) {
        return sum[p];
    }

    int mid = (l + r) / 2;
    long long ans = 0;

    if (L <= mid) ans += query(p * 2, l, mid, L, R);
    if (R > mid) ans += query(p * 2 + 1, mid + 1, r, L, R);

    return ans;
}
```

这里有两组区间：

区间	含义
$[l, r]$	当前结点负责的区间
$[L, R]$	查询区间

若当前区间完全被查询区间包含：

$$L \le l \le r \le R,$$

就可以直接返回 `sum[p]`。

否则，要看查询区间是否与左儿子、右儿子有交集：

- 若 $L \le mid$，说明查询区间碰到左儿子；
- 若 $R > mid$，说明查询区间碰到右儿子。

最后把相关部分的结果相加。

例题 4：查询过程

数组：

$$a[1..8] = 3,\ 1,\ 4,\ 1,\ 5,\ 9,\ 2,\ 6.$$

查询区间 $[3, 6]$ 的和。

根结点是 $[1, 8]$，中点为 4。查询区间 $[3, 6]$ 同时碰到左儿子 $[1, 4]$ 和右儿子 $[5, 8]$，所以分两边查。

左边 $[1, 4]$ 中，查询范围是 $[3, 4]$。结点 $[3, 4]$ 完全被包含，直接取它的和：

$$4 + 1 = 5.$$

右边 $[5, 8]$ 中，查询范围是 $[5, 6]$。结点 $[5, 6]$ 完全被包含，直接取它的和：

$$5 + 9 = 14.$$

合并得到：

$$5 + 14 = 19.$$

所以查询结果为 19。

线段树查询不是把 $3, 4, 5, 6$ 四个位置逐个加起来，而是取出几个已经维护好的区间。

三种相交情况

读查询函数时，可以按当前结点区间 $[l, r]$ 和查询区间 $[L, R]$ 的关系判断。

第一种，完全包含：

$$L \le l \text{ 且} r \le R.$$

直接返回当前结点信息。

第二种，完全不相交。某些写法会显式判断：

$$r < L \text{ 或} R < l.$$

这种情况对区间和返回 0，对区间最大值可能返回一个很小的数。

第三种，部分相交。继续递归左右儿子，把结果合并。

本书前面给出的查询代码没有显式写 “完全不相交”，而是通过：

```
if (L <= mid)
if (R > mid)
```

只递归可能相交的儿子。两种写法都常见。

例题 5：判断是否递归左儿子

当前结点区间为 $[1, 8]$，中点为 4。查询区间为 $[6, 7]$。

因为 $L = 6 > 4$，查询区间完全在右半部分，不需要递归左儿子。

代码中：

```
if (L <= mid)
```

条件为假，所以不会进入左儿子。

右儿子条件：

```
if (R > mid)
```

为真，所以只递归右儿子。

单点修改

单点修改是把某个位置的值改掉，然后更新相关区间信息。

例如把 $a[x]$ 改成 v。只会影响从叶子 $[x,x]$ 到根路径上的结点，其他区间不变。

代码如下：

```
void update(int p, int l, int r, int x, int v) {
    if (l == r) {
        sum[p] = v;
        return;
    }

    int mid = (l + r) / 2;
    if (x <= mid) update(p * 2, l, mid, x, v);
    else update(p * 2 + 1, mid + 1, r, x, v);

    pushup(p);
}
```

先递归到位置 x 对应的叶子结点，把它改成 v。回溯时，沿路重新合并区间和。

例题 6：单点修改影响哪些区间

数组长度为 8，修改位置 6。线段树区间从根到叶子的路径为：

$$[1,8] \to [5,8] \to [5,6] \to [6,6].$$

因此只需要更新这些结点的信息。

区间 $[1,4]$、$[7,8]$ 等不包含位置 6，不会受到影响。

例题 7：修改后查询

数组：

$$a[1..4] = 2,\ 7,\ 1,\ 8.$$

先把 $a[2]$ 改为 5，再查询 $[1,3]$ 的和。

修改后数组为：

$$2,\ 5,\ 1,\ 8.$$

所以 $[1,3]$ 的和为：

$$2 + 5 + 1 = 8.$$

若用线段树看，修改位置 2 会影响 $[2,2]$、$[1,2]$、$[1,4]$。查询 $[1,3]$ 时，会组合 $[1,2]$ 和 $[3,3]$ 的信息。

区间修改

有些题不是修改一个位置，而是把整个区间都加上某个数。例如：

$$a[L], a[L+1], \ldots, a[R]$$

都加上 v。

如果直接把每个位置都改一遍，一次区间修改可能需要 $O(n)$。线段树可以用懒标记处理这类操作。

懒标记的思想是：如果当前结点区间完全被修改区间覆盖，就先修改当前结点保存的信息，并记录一个标记，表示“这个修改还没有下传给儿子”。等以后需要访问儿子时，再把标记传下去。

区间加法的标记

若结点 p 表示区间 $[l,r]$，区间长度为：

$$r-l+1.$$

如果整个区间都加上 v，那么区间和增加：

$$(r-l+1)v.$$

所以可以写：

```
void addtag(int p, int l, int r, long long v) {
    sum[p] += (r - l + 1) * v;
    tag[p] += v;
}
```

这里 `tag[p]` 表示结点 p 的整个区间还有一个加法标记没有下传。

当需要访问左右儿子时，执行下传：

```
void pushdown(int p, int l, int r) {
    if (tag[p] == 0) return;

    int mid = (l + r) / 2;
    addtag(p * 2, l, mid, tag[p]);
    addtag(p * 2 + 1, mid + 1, r, tag[p]);

    tag[p] = 0;
}
```

下传后，父结点的标记清空，因为这份修改已经交给两个儿子保存了。

懒标记查询

有懒标记时，查询函数在进入子结点前，要先下传标记。

```
long long query(int p, int l, int r, int L, int R) {
    if (L <= l && r <= R) {
        return sum[p];
    }

    pushdown(p, l, r);

    int mid = (l + r) / 2;
    long long ans = 0;

    if (L <= mid) ans += query(p * 2, l, mid, L, R);
    if (R > mid) ans += query(p * 2 + 1, mid + 1, r, L, R);
```

```
    return ans;
}
```

为什么要先 pushdown？

因为子结点的信息可能还没有包含父结点积累的修改。如果不下传，就进入子树查询，可能读到旧值。

如果当前结点已经完全被查询区间包含，可以直接返回 sum[p]，不需要下传。因为 sum[p] 已经包含了这整个区间的正确信息。

例题 8：懒标记含义

数组长度为 4，初始全为 0。对区间 $[1,4]$ 全部加 3。

根结点 $[1,4]$ 完全被覆盖。此时可以直接：

$$sum[1] = 4 \times 3 = 12,$$

并记录：

$$tag[1] = 3.$$

这表示根区间的总和已经正确，但左右儿子还没有真正加上 3。

如果接下来查询 $[1,4]$，直接返回根的 sum，答案为 12。

如果接下来查询 $[2,2]$，就必须先把根的标记下传到左右儿子，再继续往下查。

区间修改代码

区间加法修改代码常见如下：

```
void update(int p, int l, int r, int L, int R, long long v) {
    if (L <= l && r <= R) {
        addtag(p, l, r, v);
        return;
    }

    pushdown(p, l, r);

    int mid = (l + r) / 2;
    if (L <= mid) update(p * 2, l, mid, L, R, v);
    if (R > mid) update(p * 2 + 1, mid + 1, r, L, R, v);

    pushup(p);
}
```

这段代码和区间查询很像。区别是：

- 查询函数返回答案；
- 修改函数改变线段树内容；
- 完全覆盖时，修改函数加标记并返回；
- 递归修改子树后，要重新 pushup(p)。

例题 9：区间加后求和

数组：

$$a[1..4] = 1,\ 2,\ 3,\ 4.$$

先对区间 $[2,4]$ 全部加 5，再查询 $[1,3]$ 的和。

修改后数组为：

$$1,\ 7,\ 8,\ 9.$$

所以查询 $[1,3]$ 的答案为：

$$1 + 7 + 8 = 16.$$

用线段树程序追踪时，不一定要展开所有标记。若题目只问最终数组某段和，小规模情况下可以先手动得到修改后的数组，再求答案。若题目问 `sum[p]` 或 `tag[p]` 的值，就必须按线段树过程追踪。

线段树复杂度

线段树把区间不断二分，高度为 $O(\log n)$。

单点修改只沿一条根到叶路径走，复杂度为：

$$O(\log n).$$

区间查询会访问若干个与查询区间有关的结点，常见复杂度为：

$$O(\log n).$$

带懒标记的区间修改也通常是：

$$O(\log n).$$

建树需要处理每个结点，复杂度为：

$$O(n).$$

这里说的是常见线段树在一维数组上的操作。第一轮中，重点是能判断“建树线性，查询和修改对数级”，以及知道每次查询或修改不是扫完整个数组。

与前缀和的区别

前缀和也能快速回答区间和查询：

$$a[L] + a[L+1] + \cdots + a[R] = s[R] - s[L-1].$$

如果数组不修改，前缀和非常适合区间和查询。

但如果数组会修改，前缀和维护起来就不方便。单点修改 $a[x]$ 会影响所有 $s[x], s[x+1], \ldots, s[n]$。区间修改影响更大。

线段树的优势是同时支持修改和查询。

结构	查询区间和	单点修改	适合场景
前缀和	$O(1)$	$O(n)$	数组不修改
线段树	$O(\log n)$	$O(\log n)$	查询和修改交替出现

下一章的 ST 表也可以做区间查询，但它不适合处理修改。本章只要先分清：线段树能处理动态修改，前缀和和 ST 表更偏向静态查询。

维护最大值和最小值

线段树不仅能维护区间和，也能维护区间最大值、最小值。

区间最大值建树：

```
void pushup(int p) {
    mx[p] = max(mx[p * 2], mx[p * 2 + 1]);
}
```

查询时，完全覆盖直接返回 `mx[p]`；部分相交时，对左右查询结果取最大值。

```
int query(int p, int l, int r, int L, int R) {
    if (L <= l && r <= R) return mx[p];

    int mid = (l + r) / 2;
    int ans = -1000000000;

    if (L <= mid) ans = max(ans, query(p * 2, l, mid, L, R));
    if (R > mid) ans = max(ans, query(p * 2 + 1, mid + 1, r, L, R));

    return ans;
}
```

这里初值要足够小。若维护区间最小值，初值要足够大，合并用 `min`。

例题 10：区间最大值

数组：

$$a[1..6] = 4,\ 1,\ 7,\ 2,\ 9,\ 3.$$

查询 $[2, 5]$ 的最大值。

区间 $[2, 5]$ 中的数为：

$$1,\ 7,\ 2,\ 9.$$

最大值为 9。

若用线段树，它会把查询区间拆成若干段，并对这些段的最大值取最大。最终结果仍为 9。

常见程序变量

线段树程序中的变量很多，先把含义分清，代码就不容易乱。

变量	常见含义
p	当前线段树结点编号
l,r	当前结点负责的区间
L,R	操作目标区间
mid	当前区间中点
sum[p]	当前区间和
mx[p]	当前区间最大值
tag[p]	当前结点的懒标记
p*2	左儿子编号
p*2+1	右儿子编号

最容易混淆的是 `l,r` 和 `L,R`。前者属于当前结点，后者属于查询或修改操作。程序阅读时可以把它们读成：

- 小写：我现在站在哪个区间；
- 大写：我要操作哪个区间。

程序阅读中的线段树

读线段树程序，可以按下面顺序。

第一，判断维护的信息。看 `pushup` 是加法、取最大值，还是取最小值。

第二，判断区间写法。通常是闭区间 $[l,r]$，但也可能是左闭右开。

第三，判断左右儿子区间。看右儿子从 `mid+1` 还是 `mid` 开始。

第四，判断完全覆盖条件。常见是：

```
L <= l && r <= R
```

第五，判断部分相交时递归哪些儿子。常见是：

```
if (L <= mid)
if (R > mid)
```

第六，若有懒标记，要看何时 `pushdown`，何时清空标记。

例题 11：判断查询含义

```
int query(int p, int l, int r, int L, int R) {
    if (L <= l && r <= R) return mx[p];

    int mid = (l + r) / 2;
    int ans = -1000000000;

    if (L <= mid) ans = max(ans, query(p * 2, l, mid, L, R));
    if (R > mid) ans = max(ans, query(p * 2 + 1, mid + 1, r, L, R));

    return ans;
}
```

这段代码返回区间 $[L, R]$ 的最大值。

判断依据是：

- 完全覆盖时返回 mx[p]；
- 部分相交时，对左右结果取 max；
- 初值是很小的数。

如果把 max 改成加法，含义就会变成区间和；如果把 mx 改成 mn 且合并用 min，含义会变成区间最小值。

例题 12：判断是否需要下传

```
if (L <= l && r <= R) {
    return sum[p];
}
pushdown(p, l, r);
```

这里完全覆盖时直接返回 sum[p]，不需要 pushdown。因为当前结点保存的区间和已经正确。

只有需要进入左右儿子时，才必须把标记下传，保证儿子信息正确。

这类题常把 pushdown 的位置作为空缺。要记住：进入子结点前下传；完全覆盖直接用当前结点信息。

完善程序中的线段树

线段树完善程序常见空缺包括：

- 建树边界；
- 左右儿子编号；
- 中点；
- pushup 合并方式；
- 查询完全覆盖条件；
- 查询递归左右儿子的条件；
- 单点修改走左还是走右；
- 懒标记更新区间和；
- pushdown 后清空标记。

例题 13：补全建树

```
void build(int p, int l, int r) {
    if (_____) {
        sum[p] = a[l];
        return;
    }
    int mid = (l + r) / 2;
    build(p * 2, l, mid);
    build(p * 2 + 1, mid + 1, r);
    sum[p] = _____;
}
```

区间只有一个元素时是叶子，所以第一空为：

```
l == r
```

区间和线段树的合并为左右儿子相加，所以第二空为：

```
sum[p * 2] + sum[p * 2 + 1]
```

例题 14：补全查询

```
long long query(int p, int l, int r, int L, int R) {
    if (_____) return sum[p];

    int mid = (l + r) / 2;
    long long ans = 0;

    if (L <= mid) ans += query(p * 2, l, mid, L, R);
    if (R > mid) ans += query(p * 2 + 1, mid + 1, r, L, R);

    return ans;
}
```

完全覆盖条件为：

```
L <= l && r <= R
```

表示当前结点区间 $[l, r]$ 完全落在目标查询区间 $[L, R]$ 内。

例题 15：补全单点修改

```
void update(int p, int l, int r, int x, int v) {
    if (l == r) {
        sum[p] = v;
        return;
    }

    int mid = (l + r) / 2;
    if (_____) update(p * 2, l, mid, x, v);
    else update(p * 2 + 1, mid + 1, r, x, v);

    pushup(p);
}
```

若位置 x 在左半区间 $[l, mid]$ 中，应进入左儿子，所以空缺为：

```
x <= mid
```

例题 16：补全懒标记

```
void addtag(int p, int l, int r, long long v) {
    sum[p] += _____;
    tag[p] += v;
}
```

区间 $[l, r]$ 中有 $r - l + 1$ 个元素，每个加 v，区间和增加：

```
(r - l + 1) * v
```

所以空缺为：

```
(r - l + 1) * v
```

常见错误

线段树题的错误集中在下面几类。

错误	说明
混淆结点编号和数组下标	`p` 是线段树结点，`l,r` 才是原数组区间
左右区间写错	闭区间常写 `[l,mid]` 和 `[mid+1,r]`
完全覆盖条件反了	应是 `L <= l && r <= R`
查询左右条件错	左边看 `L <= mid`，右边看 `R > mid`
忘记 pushup	修改子树后父结点信息没有更新
忘记 pushdown	进入子结点前没有下传懒标记
标记下传后不清空	同一个标记可能被重复下传
区间长度算错	闭区间长度是 `r-l+1`
维护内容看错	区间和、最大值、最小值合并方式不同
把线段树当静态表	线段树重点是查询和修改都能处理

做线段树题时，先写清楚每个结点代表的区间，再看这个结点保存什么信息。只要区间和信息含义不乱，代码就不会变成一堆难记的符号。

原创例题

例题 17：查询与修改综合

数组：

$$a[1..5] = 2,\ 4,\ 1,\ 7,\ 3.$$

线段树维护区间和。依次执行：

1. 查询 $[2, 4]$；
2. 把 $a[3]$ 改为 6；
3. 查询 $[2, 4]$；
4. 查询 $[1, 5]$。

第一次查询：

$$a[2] + a[3] + a[4] = 4 + 1 + 7 = 12.$$

修改后数组为：

$$2,\ 4,\ 6,\ 7,\ 3.$$

第二次查询：

$$a[2] + a[3] + a[4] = 4 + 6 + 7 = 17.$$

第三次查询：

$$2+4+6+7+3=22.$$

所以三次查询结果依次为：

$$12,\ 17,\ 22.$$

线段树程序中，第一次查询不改变树；单点修改会改变位置 3 到根路径上的结点；后两次查询基于修改后的信息。

变式 1：改成区间最大值

若线段树维护区间最大值，同样数组：

$$2,\ 4,\ 1,\ 7,\ 3.$$

查询 $[2,4]$ 的结果为：

$$\max(4,1,7)=7.$$

把 $a[4]$ 改为 0 后，数组变成：

$$2,\ 4,\ 1,\ 0,\ 3.$$

再次查询 $[2,4]$：

$$\max(4,1,0)=4.$$

维护内容改变后，`pushup` 和查询合并方式也要改变。

变式 2：区间加法

数组初始为：

$$1,\ 1,\ 1,\ 1.$$

对 $[2,4]$ 加 3 后，数组为：

$$1,\ 4,\ 4,\ 4.$$

查询 $[1,3]$ 的和为：

$$1+4+4=9.$$

若程序中使用懒标记，可能不会立即把每个叶子都改掉，但逻辑上的数组结果就是这样。判断查询答案时，看逻辑结果即可；判断内部 `tag` 时，要按程序过程追踪。

小结

线段树看起来代码长，但核心只有两件事：结点表示区间，父结点信息由左右儿子合并。查询时按区间关系递归，修改时沿受影响的区间更新。把这两件事想清楚，线段树的程序阅读和完善程序就会稳很多。下一章讲 RMQ 与 ST 表，它也处理区间查询，但与线段树相比更偏向静态数组上的快速查询。

0x18 RMQ 与 ST 表

RMQ 是 Range Minimum Query 的缩写，通常指 “区间最小值查询”。在竞赛教材中，也常把区间最大值查询一并放进来讲。给定一个数组，多次询问某个区间中的最小值或最大值，这就是典型 RMQ 问题。

第 23 章讲过线段树。线段树可以处理区间查询，也可以处理修改。ST 表处理的问题更窄：数组不修改，只做静态区间查询。它牺牲了一些空间，用预处理换取快速查询。对于区间最小值、最大值这类问题，ST 表可以做到预处理 $O(n \log n)$，每次查询 $O(1)$。

CSP-S 第一轮中，更常见的是给出一段预处理代码和查询代码，要求判断数组含义、循环边界、查询区间拆分方式，或者区分它和线段树的适用场景。

RMQ 问题

给定数组：

$$a[1..8] = 5,\ 2,\ 8,\ 6,\ 3,\ 7,\ 1,\ 4.$$

若询问区间 $[3, 7]$ 的最小值，就是在：

$$8,\ 6,\ 3,\ 7,\ 1$$

中取最小值，答案为 1。

若询问区间 $[2, 6]$ 的最大值，就是在：

$$2,\ 8,\ 6,\ 3,\ 7$$

中取最大值，答案为 8。

如果只有一次询问，直接扫描区间即可。若有很多次询问，逐个扫描就可能很慢。

RMQ 的核心任务是：数组固定不变，快速回答大量区间最值询问。

例题 1：直接扫描

数组：

$$a[1..6] = 4,\ 9,\ 1,\ 7,\ 3,\ 8.$$

求区间 $[2, 5]$ 的最小值和最大值。

区间 $[2, 5]$ 中的数为：

$$9,\ 1,\ 7,\ 3.$$

最小值为 1，最大值为 9。

直接扫描需要看 4 个数。若数组长度和询问次数都很大，就需要更快的办法。

为什么前缀和不够

区间和可以用前缀和快速计算：

$$a[L] + a[L+1] + \cdots + a[R] = s[R] - s[L-1].$$

但区间最小值不能这样简单相减。

例如：

$$a[1..5] = 5,\ 2,\ 8,\ 1,\ 7.$$

如果记前缀最小值：

$$pre[i] = \min(a[1], a[2], \ldots, a[i]),$$

那么：

$$pre[5] = 1, \quad pre[2] = 2.$$

不能用 $pre[5]$ 和 $pre[2]$ 通过减法得到区间 $[3,5]$ 的最小值。最小值不是可减的量。

这也是 RMQ 需要专门结构的原因。区间和、区间最小值、区间最大值虽然都是“区间查询”，但能用的预处理方式不同。

ST 表的区间含义

ST 表的核心思想是：预处理所有长度为 2^k 的区间信息。

设：

$$st[i][k]$$

表示从位置 i 开始、长度为 2^k 的区间中的最小值。

也就是：

$$st[i][k] = \min(a[i], a[i+1], \ldots, a[i+2^k-1]).$$

例如：

- $st[i][0]$ 表示长度为 1 的区间，即 $a[i]$；
- $st[i][1]$ 表示长度为 2 的区间；
- $st[i][2]$ 表示长度为 4 的区间；
- $st[i][3]$ 表示长度为 8 的区间。

对于数组：

$$a[1..8] = 5,\ 2,\ 8,\ 6,\ 3,\ 7,\ 1,\ 4,$$

有：

$$st[3][2] = \min(a[3], a[4], a[5], a[6]).$$

即：

$$st[3][2] = \min(8, 6, 3, 7) = 3.$$

ST 表中的第二维 k 表示区间长度的指数，不是右端点。

预处理

先处理长度为 1 的区间：

$$st[i][0] = a[i].$$

长度为 2^k 的区间，可以拆成两个长度为 2^{k-1} 的区间：

- 左半段从 i 开始；
- 右半段从 $i + 2^{k-1}$ 开始。

所以递推式为：

$$st[i][k] = \min(st[i][k-1], st[i + 2^{k-1}][k-1]).$$

对应代码：

```
for (int i = 1; i <= n; i++) {
    st[i][0] = a[i];
}

for (int k = 1; (1 << k) <= n; k++) {
    for (int i = 1; i + (1 << k) - 1 <= n; i++) {
        st[i][k] = min(st[i][k - 1], st[i + (1 << (k - 1))][k - 1]);
    }
}
```

内层循环条件：

```
i + (1 << k) - 1 <= n
```

表示从 i 开始，长度为 2^k 的区间不能越过数组末尾。

例题 2：计算一项 ST 表

数组：

$$a[1..6] = 4,\ 9,\ 1,\ 7,\ 3,\ 8.$$

求 $st[2][2]$，其中 ST 表维护区间最小值。

因为 $2^2 = 4$，所以 $st[2][2]$ 表示区间 $[2, 5]$ 的最小值：

$$st[2][2] = \min(a[2], a[3], a[4], a[5]).$$

代入：

$$st[2][2] = \min(9, 1, 7, 3) = 1.$$

也可以用递推式计算：

$$st[2][2] = \min(st[2][1], st[4][1]).$$

其中：

$$st[2][1] = \min(9, 1) = 1,$$

$$st[4][1] = \min(7, 3) = 3.$$

所以：

$$st[2][2] = 1.$$

这说明 $st[i][k]$ 不是随便存一个值，它严格对应从 i 开始、长度为 2^k 的区间。

查询

要求区间 $[L, R]$ 的最小值，先计算区间长度：

$$len = R - L + 1.$$

找到最大的 k，使得：

$$2^k \le len.$$

用两个长度为 2^k 的区间覆盖 $[L, R]$：

- 第一个从 L 开始；
- 第二个以 R 结尾，也就是从 $R - 2^k + 1$ 开始。

答案为：

$$\min(st[L][k], st[R - 2^k + 1][k]).$$

对应代码：

```
int query(int L, int R) {
    int len = R - L + 1;
    int k = lg[len];
    return min(st[L][k], st[R - (1 << k) + 1][k]);
}
```

这里两个区间可能有重叠。对于最小值查询，重叠没有问题。一个元素即使被看了两次，也不会改变最小值。

例题 3：查询过程

数组：

$$a[1..8] = 5,\ 2,\ 8,\ 6,\ 3,\ 7,\ 1,\ 4.$$

求区间 $[3, 7]$ 的最小值。

区间长度为：

$$len = 7 - 3 + 1 = 5.$$

最大的 $2^k \le 5$ 是 4，所以 $k = 2$。

取两个长度为 4 的区间：

第一个：

$$[3, 6],$$

第二个：

$$[4, 7],$$

因为：

$$R - 2^k + 1 = 7 - 4 + 1 = 4.$$

答案为：

$$\min(st[3][2], st[4][2]).$$

其中：

$$st[3][2] = \min(8, 6, 3, 7) = 3,$$

$$st[4][2] = \min(6, 3, 7, 1) = 1.$$

所以：

$$\min(3, 1) = 1.$$

这和直接看 $[3,7]$ 中的数：

$$8,\ 6,\ 3,\ 7,\ 1$$

得到的最小值一致。

为什么可以重叠

ST 表查询最值时，常用两个长度相同的区间覆盖查询区间。这两个区间可能重叠。

例如查询 $[3,7]$ 时，用的是：

$$[3,6] \quad 和 \quad [4,7].$$

它们的重叠部分是：

$$[4,6].$$

对最小值来说，重叠不会影响答案。因为：

$$\min(x,x) = x.$$

同一个元素参加两次最小值计算，结果不变。

最大值也是一样：

$$\max(x,x) = x.$$

所以 ST 表可以用两个重叠区间 $O(1)$ 回答区间最小值和最大值。

但是区间和不能这样做。若把两个重叠区间的和相加，重叠部分会被多算。

例题 4：区间和不能重叠

数组：

$$a[1..5] = 1,\ 2,\ 3,\ 4,\ 5.$$

查询区间 $[2,5]$ 的和为：

$$2+3+4+5 = 14.$$

若用两个长度为 4 的区间覆盖：

$$[2,5] \quad 和 \quad [2,5],$$

相加会得到：

$$14+14 = 28.$$

即使换成两个有部分重叠的区间，重叠部分也会被重复计算。

因此，ST 表的 $O(1)$ 查询方式适合最小值、最大值、gcd 这类重复计算不改变结果的运算，不适合直接用来做区间和的重叠查询。

本章只要求掌握最值查询。gcd 会在数论中出现，本章不展开。

lg 数组

为了快速得到：

$$\lfloor \log_2 len \rfloor ,$$

常预处理一个 lg 数组。

```
lg[1] = 0;
for (int i = 2; i <= n; i++) {
    lg[i] = lg[i / 2] + 1;
}
```

含义是：

$$lg[i] = \lfloor \log_2 i \rfloor .$$

例如：

i	$lg[i]$
1	0
2	1
3	1
4	2
5	2
6	2
7	2
8	3

查询区间长度为 5 时，$lg[5] = 2$，表示可以使用长度 $2^2 = 4$ 的块。

例题 5：计算 lg

按代码：

```
lg[1] = 0;
for (int i = 2; i <= 10; i++) {
    lg[i] = lg[i / 2] + 1;
}
```

逐项得到：

i	$i/2$	$lg[i]$
2	1	1
3	1	1
4	2	2
5	2	2
6	3	2
7	3	2
8	4	3
9	4	3
10	5	3

所以 $lg[10]=3$，因为 $2^3 \le 10 < 2^4$。

程序阅读题中，若看到 `lg[i] = lg[i / 2] + 1`，要想到它在求二进制长度或 $\lfloor \log_2 i \rfloor$。

维护最大值

如果要维护区间最大值，只需要把 `min` 改成 `max`。

预处理：

```
for (int i = 1; i <= n; i++) {
    st[i][0] = a[i];
}

for (int k = 1; (1 << k) <= n; k++) {
    for (int i = 1; i + (1 << k) - 1 <= n; i++) {
        st[i][k] = max(st[i][k - 1], st[i + (1 << (k - 1))][k - 1]);
    }
}
```

查询：

```
int query(int L, int R) {
    int len = R - L + 1;
    int k = lg[len];
    return max(st[L][k], st[R - (1 << k) + 1][k]);
}
```

例题 6：区间最大值查询

数组：

$$a[1..8] = 5,\ 2,\ 8,\ 6,\ 3,\ 7,\ 1,\ 4.$$

求区间 $[2,6]$ 的最大值。

长度为：

$$len = 6 - 2 + 1 = 5,$$

所以 $k=2$，块长为 4。

两个区间为：

$$[2,5] \quad 和 \quad [3,6].$$

答案为：

$$\max(st[2][2], st[3][2]).$$

其中：

$$st[2][2] = \max(2,8,6,3) = 8,$$

$$st[3][2] = \max(8,6,3,7) = 8.$$

所以答案为 8。

若直接看区间 $[2,6]$：

$$2,\ 8,\ 6,\ 3,\ 7,$$

最大值也是 8。

下标从 0 开始的写法

有些代码使用 0 下标数组。此时：

$$a[0..n-1]$$

表示长度为 n 的数组。

ST 表含义可以写成：

$$st[i][k]$$

表示从 i 开始、长度为 2^k 的区间。

预处理代码常写成：

```
for (int i = 0; i < n; i++) {
    st[i][0] = a[i];
}

for (int k = 1; (1 << k) <= n; k++) {
    for (int i = 0; i + (1 << k) <= n; i++) {
        st[i][k] = min(st[i][k - 1], st[i + (1 << (k - 1))][k - 1]);
    }
}
```

注意内层循环条件变成：

```
i + (1 << k) <= n
```

因为最后一个下标是 $n-1$。

查询 $[L,R]$ 时，仍然是：

```
int len = R - L + 1;
int k = lg[len];
return min(st[L][k], st[R - (1 << k) + 1][k]);
```

这里 L,R 是 0 下标下的闭区间端点。

例题 7：0 下标查询

数组：

$$a[0..5] = 4,\ 9,\ 1,\ 7,\ 3,\ 8.$$

查询 $[1,4]$ 的最小值。

区间中的数为：

$$9,\ 1,\ 7,\ 3.$$

最小值为 1。

长度：

$$len = 4 - 1 + 1 = 4,$$

所以 $k=2$，块长为 4。此时两个块刚好都是 $[1,4]$：

$$st[1][2].$$

答案为 1。

0 下标和 1 下标的核心公式相同，但建表边界不同。程序阅读题中，先确认数组下标从哪里开始。

和线段树的区别

ST 表和线段树都能做区间最值查询，但适用场景不同。

结构	预处理	查询	修改	适合场景
线段树	$O(n)$	$O(\log n)$	$O(\log n)$	查询和修改交替
ST 表	$O(n\log n)$	$O(1)$	不适合	数组静态，多次查询

如果数组会修改，ST 表通常不适合。因为一个位置变化后，很多包含它的预处理区间都会受影响，维护代价很高。

如果数组不修改，且询问很多，ST 表查询快，代码也比带修改的线段树短。

例题 8：选择结构

有一个长度为 10^5 的数组，有 10^5 次询问，每次询问区间最大值，数组不修改。适合使用什么结构？

可以使用 ST 表。预处理后，每次询问区间最大值为 $O(1)$。

若题目改成“每次可能修改一个位置，然后询问区间最大值”，ST 表就不合适，应考虑线段树。

选择数据结构时，要先看数组是否会修改。静态查询和动态修改是重要分界。

不是所有区间问题都适合 ST 表

ST 表最适合区间最小值、最大值这类可以重叠合并的操作。

对区间和，也可以用类似预处理保存长度为 2^k 的区间和，但查询时不能用两个重叠区间 $O(1)$ 合并。若要用二进制拆成若干个不重叠区间，查询复杂度会变成 $O(\log n)$。而区间和本来可以用前缀和 $O(1)$ 查询，所以没有必要使用 ST 表。

对区间修改，ST 表也不适合。

第一轮中，如果题目问“以下哪个结构适合静态区间最大值多次查询”，ST 表是合理选项；如果题目包含频繁修改，线段树通常更合适。

原创例题：静态区间最大值

下面程序用于回答静态数组上的区间最大值查询。数组下标从 1 开始。

```
const int N = 100005;
int n, q;
int a[N], lg[N], st[N][20];

void init() {
    lg[1] = 0;
    for (int i = 2; i <= n; i++) {
        lg[i] = lg[i / 2] + 1;
    }

    for (int i = 1; i <= n; i++) {
        st[i][0] = a[i];
    }

    for (int k = 1; (1 << k) <= n; k++) {
        for (int i = 1; i + (1 << k) - 1 <= n; i++) {
            st[i][k] = max(st[i][k - 1], st[i + (1 << (k - 1))][k - 1]);
        }
    }
}

int query(int l, int r) {
    int k = lg[r - l + 1];
    return max(st[l][k], st[r - (1 << k) + 1][k]);
}
```

设：

$$a[1..8] = 5,\ 2,\ 8,\ 6,\ 3,\ 7,\ 1,\ 4.$$

问 `query(3,7)` 的返回值。

区间 $[3,7]$ 的长度为 5，所以：

$$k = lg[5] = 2.$$

块长为 4。查询使用两个区间：

$$[3,6] \quad 和 \quad [4,7].$$

第一个区间最大值为：

$$\max(8,6,3,7)=8.$$

第二个区间最大值为：

$$\max(6,3,7,1)=7.$$

所以返回：

$$\max(8,7)=8.$$

这道题的关键是看懂 `st[i][k]` 的含义，以及 `r-(1<<k)+1` 为什么是第二个块的起点。

变式 1：改成最小值

若程序中的 `max` 全部改成 `min`，则 `query(3,7)` 查询区间最小值。

区间 $[3,7]$ 中的数为：

$$8,\ 6,\ 3,\ 7,\ 1.$$

最小值为 1。

变式 2：边界少写一位

若建表内层循环写成：

```
for (int i = 1; i + (1 << k) <= n; i++)
```

对于 1 下标闭区间，这会漏掉最后一个合法区间。

例如 $n=8,k=2$，长度为 4 的区间最后可以从 $i=5$ 开始，即 $[5,8]$。正确条件：

$$i+4-1\le 8,$$

所以 $i\le 5$。

错误条件：

$$i+4\le 8,$$

所以 $i\le 4$，漏掉了 $[5,8]$。

完善程序题中，循环边界经常是考点。

程序阅读中的 ST 表

读 ST 表程序，可以按下面顺序。

第一，判断维护内容。看递推中使用 `min`、`max`，还是其他合并方式。

第二，判断下标从 0 还是从 1 开始。

第三，判断 `st[i][k]` 的含义。通常表示从 i 开始，长度为 2^k 的区间。

第四，检查建表边界。区间右端不能超过数组末尾。

第五，检查查询长度和 `lg`。查询区间长度是 $R-L+1$。

第六，检查第二个块的起点。闭区间中通常是：

```
R - (1 << k) + 1
```

第七，判断是否存在修改。若数组会修改，ST 表预处理结果通常不能直接继续使用。

例题 9：判断程序含义

```
for (int k = 1; (1 << k) <= n; k++) {
    for (int i = 1; i + (1 << k) - 1 <= n; i++) {
        st[i][k] = min(st[i][k - 1], st[i + (1 << (k - 1))][k - 1]);
    }
}
```

这段程序在预处理区间最小值。

原因是：长度为 2^k 的区间被拆成两个长度为 2^{k-1} 的区间，当前值取两边最小。

若把 `min` 改成 `max`，就变成预处理区间最大值。

例题 10：判断第二块起点

查询区间为 $[10,25]$。若 $k=4$，块长为 16。第二个块的起点是多少？

因为第二个块要以 $R=25$ 结尾，长度为 16，所以起点为：

$$25-16+1=10.$$

两个块都正好是 $[10,25]$。

若查询区间为 $[10,30]$，长度为 21，仍取 $k=4$，块长为 16。第二个块起点为：

$$30-16+1=15.$$

两个块为：

$$[10,25] \quad 和 \quad [15,30].$$

它们有重叠，但用于最值查询没有问题。

完善程序中的 ST 表

ST 表完善程序常见空缺包括：

- `lg` 数组递推；
- `st[i][0]` 初始化；
- 建表中的右端边界；
- 建表递推式；
- 查询区间长度；
- 查询中的 k；
- 第二个块起点；
- `min` 和 `max` 的选择。

例题 11：补全 `lg`

```
lg[1] = 0;
for (int i = 2; i <= n; i++) {
    lg[i] = _____;
}
```

应填：

```
lg[i / 2] + 1
```

因为 $\lfloor \log_2 i \rfloor$ 可以由 $\lfloor \log_2 \lfloor i/2 \rfloor \rfloor$ 加 1 得到。

例题 12：补全建表

```
for (int k = 1; (1 << k) <= n; k++) {
    for (int i = 1; _____; i++) {
        st[i][k] = min(_____, _____);
    }
}
```

若数组下标从 1 开始，且维护区间最小值，内层循环条件为：

```
i + (1 << k) - 1 <= n
```

递推式左右两半为：

```
st[i][k - 1]
st[i + (1 << (k - 1))][k - 1]
```

完整写法为：

```
for (int k = 1; (1 << k) <= n; k++) {
    for (int i = 1; i + (1 << k) - 1 <= n; i++) {
        st[i][k] = min(st[i][k - 1], st[i + (1 << (k - 1))][k - 1]);
    }
}
```

例题 13：补全查询

```
int query(int L, int R) {
    int len = _____;
    int k = lg[len];
    return min(st[L][k], st[_____][k]);
}
```

查询区间长度为：

```
R - L + 1
```

第二个长度为 2^k 的块以 R 结尾，起点为：

```
R - (1 << k) + 1
```

完整写法为：

```
int query(int L, int R) {
    int len = R - L + 1;
    int k = lg[len];
    return min(st[L][k], st[R - (1 << k) + 1][k]);
}
```

常见错误

ST 表题的错误集中在下面几类。

错误	说明
混淆第二维含义	`k` 表示长度 2^k，不是右端点
建表边界少一位	1 下标闭区间应检查 `i+(1<<k)-1<=n`
查询长度写错	闭区间长度是 `R-L+1`
第二块起点错	应为 `R-(1<<k)+1`
把 ST 表用于动态修改	数组修改后预处理信息会失效
把最值和求和混用	重叠块适合最值，不适合区间和
忘记初始化 `st[i][0]`	长度为 1 的区间是所有递推基础
`lg` 数组含义错	`lg[len]` 是不超过 `len` 的最大二次幂指数
下标体系混用	0 下标和 1 下标建表边界不同
与线段树适用场景混淆	ST 表静态快查，线段树支持修改

做 ST 表题时，先写清楚一句话：`st[i][k]` 表示从 i 开始、长度为 2^k 的区间信息。只要这句话不乱，建表和查询公式都能推出来。

原创例题

例题 14：完整查询过程

数组：

$$a[1..10] = 6,\ 4,\ 9,\ 1,\ 7,\ 3,\ 8,\ 2,\ 5,\ 10.$$

ST 表维护区间最小值。求 `query(4,9)` 的结果。

查询区间为：

$$[4,9],$$

其中的数为：

$$1,\ 7,\ 3,\ 8,\ 2,\ 5.$$

直接看答案应为 1。

按 ST 表查询过程：

区间长度：

$$len = 9 - 4 + 1 = 6.$$

最大的 $2^k \le 6$ 是 4，所以 $k = 2$。

两个长度为 4 的区间：

$$[4,7] \quad 和 \quad [6,9].$$

第一个区间最小值：

$$\min(1,7,3,8) = 1.$$

第二个区间最小值：

$$\min(3,8,2,5) = 2.$$

查询结果为：

$$\min(1,2) = 1.$$

变式 1：查询最大值

若 ST 表维护区间最大值，同样查询 $[4,9]$。

两个区间仍为：

$$[4,7] \quad 和 \quad [6,9].$$

第一个区间最大值为：

$$\max(1,7,3,8) = 8.$$

第二个区间最大值为：

$$\max(3,8,2,5) = 8.$$

答案为 8。

变式 2：修改后不能直接查

若把 $a[5]$ 从 7 改为 0，原来的 ST 表不能直接继续使用。

因为很多预处理区间包含位置 5，例如：

$$[5,5],\ [4,5],\ [5,6],\ [1,8],...$$

这些区间的最小值或最大值都可能改变。ST 表不是为动态修改设计的结构。若需要修改后继续查询，应使用线段树等支持修改的数据结构。

小结

RMQ 与 ST 表的重点不是代码长度，而是区间含义。线段树把区间递归拆开，适合修改；ST 表预处理固定长度的区间，适合静态快速查询。到这一章，第四编 “数据结构” 结束。下一编进入树与图，图的基本概念、连通性和最短路等内容会成为重点。

第五编 树与图

0x19 图的基本概念

图是由点和边组成的结构。二叉树、堆、Trie 都可以看作特殊的图，但图比树更自由：边可以有方向，也可以没有方向；两个点之间可能有边，也可能没有边；一张图可能连成一片，也可能分成几块。

CSP-S 第一轮中的图论题，不一定从复杂算法开始。很多题考的是基本概念是否清楚：有向图和无向图的区别，简单图最多有多少条边，点的度数如何计算，路径和环如何判断，图该用邻接矩阵还是邻接表存储。若这些概念混在一起，后面的连通性、拓扑排序、最小生成树和最短路都会读得很吃力。

点和边

一张图通常记作 $G=(V,E)$。其中 V 是点集，也叫顶点集；E 是边集。

例如有 4 个点，点集可以写成 $V=\{1,2,3,4\}$。若边集为 $E=\{(1,2),(2,3),(3,4)\}$，那么这张图中有 4 个点、3 条边。

如果边没有方向，$(1,2)$ 和 $(2,1)$ 表示同一条边，这叫无向图。如果边有方向，$(1,2)$ 表示从 1 指向 2，与 $(2,1)$ 不同，这叫有向图。

在无向图中，边常写成 $\{u,v\}$。在有向图中，边常写成 (u,v)。

第一轮题目里，图经常直接画出来。读图时不要只看点的位置，要看边是否带箭头、边权是否标在边上、是否允许重复边和自环。

图的类型	边的含义
无向图	两个点之间可以双向经过
有向图	只能按箭头方向经过
带权图	每条边有一个权值
无权图	边只表示连接关系
简单图	没有自环，也没有重边

自环是从一个点连回自己的边，例如 (u,u)。重边是两个点之间存在多条相同方向或相同连接关系的边。若题目说“简单无向图”，就默认没有自环和重边。

例题 1：判断边数

一张无向图有 5 个点，边为 $\{1,2\},\{2,3\},\{3,1\},\{3,4\},\{4,5\}$。这张图有多少条边？

边集里列出了 5 条无向边，所以边数为 5。不要把 $\{1,2\}$ 和 $\{2,1\}$ 分开计算。

若同样的连接关系出现在有向图中，$(1,2)$ 和 $(2,1)$ 是两条不同的边。边是否有方向，是计数前必须先确认的信息。

度数

在无向图中，一个点的度数等于与它相连的边数。

例如边集为 $\{1,2\},\{1,3\},\{2,3\},\{3,4\}$。各点度数如下：

点	相连的边	度数
1	$\{1,2\},\{1,3\}$	2
2	$\{1,2\},\{2,3\}$	2
3	$\{1,3\},\{2,3\},\{3,4\}$	3
4	$\{3,4\}$	1

无向图中，所有点的度数之和等于边数的两倍，即 $\sum_{v\in V}\deg(v)=2m$。原因是每条无向边连接两个端点，会给两个端点各贡献 1 次度数。

在有向图中，要区分入度和出度。入度表示有多少条边指向这个点，出度表示有多少条边从这个点指出去。

例如有向边为 $(1,2),(1,3),(2,3),(3,1)$。各点入度、出度如下：

点	入度	出度
1	1	2
2	1	1
3	2	1

有向图中，所有点的入度之和等于边数，所有点的出度之和也等于边数，即 $\sum_{v\in V}indeg(v)=m$，$\sum_{v\in V}outdeg(v)=m$。因为每条有向边都有一个起点和一个终点，给起点贡献一次出度，给终点贡献一次入度。

例题 2：用度数和判断

一张无向图有 8 个点，所有点的度数分别为 $1,2,2,3,3,4,4,5$。这张图有多少条边？

先求度数和：$1+2+2+3+3+4+4+5=24$。无向图中度数和为 $2m$，所以 $2m=24$，边数 $m=12$。

如果度数和是奇数，这样的无向图不存在。因为无向图的度数和一定等于 $2m$，它必须是偶数。

简单图最多有多少条边

简单图没有自环和重边。这个限制常用于选择题计数。

对于 n 个点的简单无向图，每条边由两个不同的点决定，所以最多有 $\binom{n}{2}=\frac{n(n-1)}{2}$ 条边。例如 5 个点的简单无向图最多有 $\binom{5}{2}=10$ 条边。

这张图就是完全图，记作 K_5。完全图中，任意两个不同顶点之间都有一条边。

对于 n 个点的简单有向图，若不允许自环，但允许 $u\to v$ 和 $v\to u$ 同时存在，那么最多有 $n(n-1)$ 条边。因为每条有向边由有序点对 (u,v) 决定，且 $u\neq v$。起点有 n 种选择，终点有 $n-1$ 种选择。

图的类型	最多边数
简单无向图	$\dfrac{n(n-1)}{2}$
无自环简单有向图	$n(n-1)$

第一轮题中，如果题目写的是“简单无向图”，要用 $\binom{n}{2}$；如果题目写的是“有向图”，要重新判断是否区分方向。

真题精讲：非连通简单图

2021 年 CSP-S 第一轮考过这样的问题：

一张非连通简单无向图共有 36 条边，问这张图至少有多少个点？

题目里有三个关键词：简单无向图、非连通、共有 36 条边。

如果只看简单无向图，9 个点的完全图最多有 $\binom{9}{2}=36$ 条边。但 K_9 是连通的，不符合“非连通”。非连通图要求至少分成两个部分，不能所有点都在同一个连通块里。

要在点数尽量少的情况下放尽量多的边，可以让其中 $n-1$ 个点构成完全图，再让剩下 1 个点孤立。这样仍然非连通，边数最多为 $\binom{n-1}{2}$。

试 $n=9$，有 $\binom{8}{2}=28<36$，不够。试 $n=10$，有 $\binom{9}{2}=36$，可以让 9 个点构成完全图，再放一个孤立点。此时图非连通，边数正好为 36。

所以答案为 10。

这类题不要只套 $\binom{n}{2}$，还要把“非连通”这个限制放进去。

变式：边数改为 45

若一张非连通简单无向图共有 45 条边，至少需要多少个点？

同样让 $n-1$ 个点尽量连满，剩下一个点孤立。要求 $\binom{n-1}{2}\geq 45$。因为 $\binom{9}{2}=36$，$\binom{10}{2}=45$，所以 $n-1=10$，即 $n=11$。

至少需要 11 个点。

路径和环

图中的路径，是从一个点出发，沿着边依次到达若干点的过程。

在无向图中，若有边 $\{u,v\}$，就可以从 u 到 v，也可以从 v 到 u。在有向图中，若有边 (u,v)，只能从 u 到 v，不能反向走，除非图中还存在 (v,u)。

例如有向边为 $(1,2),(2,3),(3,4),(2,4)$。从 1 到 4 有路径 $1\to 2\to 4$，也有路径 $1\to 2\to 3\to 4$，但从 4 到 1 没有路径，因为没有反向边。

路径中经过的边数常称为路径长度。如果图是带权图，路径长度也可能指边权之和。做题时要看题目如何定义。

环是从某个点出发，沿边走若干步后回到原点的结构。例如无向图中的 $\{1,2\},\{2,3\},\{3,1\}$ 构成一个长度为 3 的环；有向图中的 $(1,2),(2,3),(3,1)$ 构成一个有向环。

若一张有向图没有有向环，称为有向无环图，常写作 DAG。DAG 和拓扑排序关系紧密，会在第 28 章讲。

例题 3：判断有向路径

有向图边集为 $(1,2),(2,5),(1,3),(3,4),(4,2)$。判断下面哪些路径存在。

A. $1\to 2\to 5$

B. $1\to 3\to 4\to 2\to 5$

C. $5\to 2\to 1$

D. $3\to 1\to 2$

A 存在，因为有 $(1,2)$ 和 $(2,5)$。

B 存在，因为有 $(1,3)$、$(3,4)$、$(4,2)$、$(2,5)$。

C 不存在。虽然有 $2\to 5$，但没有 $5\to 2$，方向不能反过来。

D 不存在。没有 $3\to 1$ 这条边。

有向图读路径时，方向比图形位置更重要。箭头方向错了，路径就不存在。

连通和连通块

在无向图中，如果任意两个点之间都有路径相连，这张图就是连通图。

如果不是所有点都能互相到达，图就会分成若干个连通块。每个连通块内部的点互相可达，不同连通块之间没有路径。

例如无向图边集为 $\{1,2\},\{2,3\},\{4,5\}$。点 $1,2,3$ 在同一个连通块中，点 $4,5$ 在另一个连通块中。如果还有点 6 没有任何边，那么点 6 单独构成一个连通块。

所以这张图有 3 个连通块：$\{1,2,3\}$、$\{4,5\}$、$\{6\}$。

有向图中也有“能不能到达”的问题，但要区分方向。若任意两个点 u,v 都满足 u 能到 v 且 v 能到 u，这张有向图才是强连通图。强连通会在下一章讲，本章只先记住：有向图的可达性不能按无向边处理。

树是一类特殊的无向连通图。含有 n 个点的树，恰好有 $n-1$ 条边，并且没有环。

这三个条件之间关系很紧：

- 无向图连通且有 $n-1$ 条边，则它是一棵树；
- 无向图无环且有 $n-1$ 条边，则它是一棵树；
- 树中任意两点之间恰好有一条简单路径。

这里的“简单路径”指路径中不重复经过点。后面讲最小生成树、LCA、树的重心时，都要用到这些性质。

例题 4：判断是否为树

一张无向图有 6 个点、5 条边，并且连通。它一定是一棵树吗？

是。

含 n 个点的连通无向图，如果边数为 $n-1$，就没有多余的环边，结构是一棵树。本题中 $n=6$，边数为 $5=n-1$，且图连通，所以它是一棵树。

如果只知道有 6 个点、5 条边，但不知道是否连通，就不能断定它是树。例如其中 3 个点可以构成一个三角形，另外 3 个点用两条边连成一条链，总边数也是 5，但整张图不连通，也不是树。

完全图中的环计数

完全图的计数题在第一轮中常出现。完全图 K_n 中，任意两个不同点之间都有边。

求长度为 4 的环时，要注意同一个环会被多次表示。例如在 4 个点 a,b,c,d 上，$a\to b\to c\to d\to a$、$b\to c\to d\to a\to b$、$a\to d\to c\to b\to a$ 表示的是同一个环。

起点可以换，方向也可以反过来。所以如果直接排列 4 个点，会把同一个无向环重复计算 8 次：4 个起点乘以 2 个方向。

在 K_n 中数长度为 4 的无向环，可以分两步。第一步选出参与环的 4 个点，有 $\binom{n}{4}$ 种选法。第二步，在这 4 个点上形成不同的无向环，数量为 $\frac{(4-1)!}{2}=3$。所以 K_n 中长度为 4 的无向环数量为 $3\binom{n}{4}$。

真题精讲：完全图中的 4 环

2024 年 CSP-S 第一轮考过类似问题：

一个 10 个顶点的完全图中，有多少个长度为 4 的环？

先选出环上的 4 个点，有 $\binom{10}{4}=210$ 种选法。每组选出的 4 个点可以构成 3 个不同的无向 4 环，所以总数为 $210\times 3=630$。

答案为 630。

这类题最容易错在重复计数。若直接算 $10\times 9\times 8\times 7$，那是在数有起点、有方向的走法，不是数不同的无向环。

变式：K_8 中的 4 环

K_8 中有多少个长度为 4 的无向环？

先选点，有 $\binom{8}{4} = 70$ 种选法。每 4 个点形成 3 个不同的无向环，所以答案为 $70 \times 3 = 210$。

图的存储

程序阅读题中，图常用两种方式存储：邻接矩阵和邻接表。

邻接矩阵用二维数组表示点与点之间是否有边。若有 n 个点，可以用：

```
int g[N][N];
```

若是无向图，加入边 $\{u, v\}$ 时：

```
g[u][v] = 1;
g[v][u] = 1;
```

若是有向图，加入边 (u, v) 时：

```
g[u][v] = 1;
```

不要再写 `g[v][u] = 1`，除非题目说这条边是双向边。

邻接矩阵的特点如下：

操作	代价
判断 u, v 是否有边	$O(1)$
枚举 u 的所有邻点	$O(n)$
空间	$O(n^2)$

当点数较小，或者题目经常判断两个点之间是否直接有边时，邻接矩阵很方便。

邻接表则为每个点保存它连出去的边。常见写法有 `vector` 写法，也有数组链式前向星写法。

`vector` 写法如下：

```
vector<int> e[N];

void add(int u, int v) {
    e[u].push_back(v);
}
```

若是无向图，加入边 $\{u, v\}$ 时要加两次：

```
add(u, v);
add(v, u);
```

若是有向图，只加一次：

```
add(u, v);
```

邻接表的特点如下：

操作	代价
枚举 u 的所有邻点	$O(\deg(u))$
判断 u, v 是否有边	通常要扫描 u 的邻接表
空间	$O(n+m)$

当边数远小于 n^2 时，邻接表更常用。最短路、拓扑排序、连通性搜索等程序里，经常使用邻接表。

链式前向星

第一轮程序阅读中，也可能出现下面这种数组存图：

```
int head[N], to[M], nxt[M], cnt;

void add(int u, int v) {
    cnt++;
    to[cnt] = v;
    nxt[cnt] = head[u];
    head[u] = cnt;
}
```

这里：

数组或变量	含义
head[u]	点 u 的第一条边编号
to[i]	第 i 条边指向的点
nxt[i]	与第 i 条边同起点的下一条边
cnt	当前已经加入的边数

枚举点 u 的所有出边：

```
for (int i = head[u]; i; i = nxt[i]) {
    int v = to[i];
}
```

这段循环的含义是：从 u 的第一条边开始，沿 nxt 找到下一条同样从 u 出发的边，直到编号为 0 表示结束。

如果题目是无向图，每条无向边要调用两次 add：

```
add(u, v);
add(v, u);
```

所以数组大小通常要能放下 $2m$ 条有向边。这里的 $2m$ 不是原图有 $2m$ 条边，而是程序把一条无向边拆成两个方向来存。

例题 5：读链式前向星

执行如下代码：

```
add(1, 2);
add(1, 3);
add(2, 4);
```

初始时 head 全为 0，cnt=0。求执行后点 1 的邻点按枚举顺序是什么。

第一次 add(1, 2) 后，cnt=1，to[1]=2，nxt[1]=head[1]=0，head[1]=1。

第二次 add(1, 3) 后，cnt=2，to[2]=3，nxt[2]=head[1]=1，head[1]=2。

第三次 `add(2, 4)` 不影响点 1 的边表。

枚举点 1 时，从 `i=head[1]=2` 开始，先得到 `to[2]=3`；再令 `i=nxt[2]=1`，得到 `to[1]=2`；再令 `i=nxt[1]=0`，循环结束。

所以点 1 的邻点枚举顺序为 $3, 2$。

它和加入顺序相反。链式前向星通常把新边插到表头，所以后加入的边会先被枚举到。大多数图算法不依赖邻点顺序，但程序阅读题可能会问具体输出，此时要按代码顺序追踪。

邻接矩阵和邻接表的选择

第一轮题中，存图方式常和复杂度一起考。

如果用邻接矩阵枚举所有边，常见代码是：

```
for (int u = 1; u <= n; u++) {
    for (int v = 1; v <= n; v++) {
        if (g[u][v]) {
            // 处理边 u -> v
        }
    }
}
```

这段循环无论实际有多少条边，都会检查 n^2 个位置，时间为 $O(n^2)$。

如果用邻接表枚举所有边，常见代码是：

```
for (int u = 1; u <= n; u++) {
    for (int i = head[u]; i; i = nxt[i]) {
        int v = to[i];
        // 处理边 u -> v
    }
}
```

这段代码每条存储的边会被枚举一次。若是有向图，复杂度为 $O(n+m)$；若是无向图按两个方向存储，枚举次数为 $2m$，量级仍为 $O(n+m)$。

存储方式	适合情况	注意点
邻接矩阵	点数较小，频繁判断两点是否有边	空间为 $O(n^2)$
邻接表	边数较少，经常枚举邻点	判断两点是否直接有边不如矩阵方便
链式前向星	边数较大，数组存储更紧凑	无向边通常要加两次

选择题中若问“哪种存储更适合稀疏图”，一般指邻接表。稀疏图是边数远少于 n^2 的图。若边数接近 n^2，图称为稠密图，邻接矩阵也可能合适。

这里不需要把“稀疏”和“稠密”背成固定数字，关键看边数和 n^2 的关系。

边权

有些图的边带有权值。例如 $(1, 2, 5)$ 可以表示从 1 到 2 有一条权值为 5 的边。

在无向带权图中，一条边 $\{u, v\}$ 的权值同时适用于两个方向。程序中常写：

```
add(u, v, w);
add(v, u, w);
```

在有向带权图中，只能按方向使用：

```
add(u, v, w);
```

边权可能表示距离、费用、时间、容量，也可能只是题目定义的某种代价。读题时不要把边权默认理解成距离，要看题目要求求什么。

带权图中，路径的总代价通常是路径上边权之和。例如路径 $1 \to 2 \to 4$ 的边权分别为 3 和 5，总代价为 8。

最短路、最小生成树会大量使用边权。第 29 章和第 30 章会展开，本章先把“边权属于边，不属于点”这件事分清楚。

例题 6：有向带权路径

有向带权边如下：$(1,2,4),(2,3,7),(1,3,15),(3,4,2)$。从 1 到 4 的路径 $1 \to 2 \to 3 \to 4$ 的总代价是多少？

路径上三条边权为 $4,7,2$，总代价为 $4+7+2=13$。

虽然有边 $(1,3,15)$，但这条边不在路径 $1 \to 2 \to 3 \to 4$ 中，不参与计算。

若选择路径 $1 \to 3 \to 4$，总代价为 $15+2=17$。不同路径可能连接同一对起点和终点，但总代价不同。

图题中的编号

图论程序常见两种编号方式：从 0 开始，或从 1 开始。

从 1 开始时，点编号为 $1,2,\ldots,n$。循环常写：

```
for (int i = 1; i <= n; i++) {
    // ...
}
```

从 0 开始时，点编号为 $0,1,\ldots,n-1$。循环常写：

```
for (int i = 0; i < n; i++) {
    // ...
}
```

程序阅读题里，编号方式会影响数组下标、循环边界和输出答案。尤其是题目输入从 1 开始，而程序内部转成 0 开始时，要留意：

```
u--;
v--;
```

或者输出时：

```
cout << u + 1 << endl;
```

例题 7：编号转换

题目输入一条边 `2 5`，表示点 2 到点 5。程序读入后执行：

```
u--;
v--;
add(u, v);
```

那么内部实际加入的是哪条边？

输入时 $u=2,v=5$。执行自减后，$u=1$，$v=4$，所以内部加入的是从 1 到 4 的边。

这不代表题目中的边变了，只是程序内部使用从 0 开始的编号。若最后要输出题目编号，需要再加回 1。

程序阅读中的图

图论程序阅读题通常不会只问一个概念，而是把存图、循环、状态数组放在一起考。读这类程序时，可以按下面顺序处理。

第一，看图是有向还是无向。若读入一条边后调用一次 `add`，通常是有向图；若调用两次 `add(u,v)` 和 `add(v,u)`，通常是无向图。

第二，看边是否带权。若 `add` 有三个参数 `u,v,w`，且结构体里有 `w`、`weight` 或 `cost`，这通常是带权图。

第三，看数组含义。常见数组如下：

数组	常见含义
`vis[u]`	点 u 是否被访问过
`deg[u]`	点 u 的度数、入度或出度
`dis[u]`	到点 u 的距离或代价
`pre[u]`	路径中点 u 的前驱
`head[u]`	点 u 的边表开头
`to[i]`	第 i 条边指向的点
`nxt[i]`	第 i 条边的下一条边

第四，看枚举顺序。`vector` 存边通常按加入顺序枚举；链式前向星通常按表头插入，枚举顺序和加入顺序相反。

第五，看输出。若输出的是 `dis[t]`，程序多半在求某种最短或最小代价；若输出计数变量，可能在统计可达点、连通块或路径数；若输出 `pre` 还原的点序列，则要追踪前驱更新。

本章不要求掌握所有图算法，但要能先把程序读成“这是一张什么图，边怎么存，循环在枚举什么”。

小结

图论的第一步不是算法，而是语言。点、边、方向、边权、度数、路径、环、连通块、存图方式，这些概念决定了后面题目该怎样读。

本章需要记住几件事：

- 无向边不区分方向，有向边必须按箭头走；
- 简单无向图最多有 $\binom{n}{2}$ 条边，简单有向图最多有 $n(n-1)$ 条边；
- 无向图度数和为 $2m$，有向图入度和、出度和都为 m；
- 非连通图的最大边数常通过“一个大完全图加孤立点”来分析；
- 完全图中数环要处理重复计数；
- 邻接矩阵适合判断两点是否有边，邻接表适合枚举邻点；
- 无向边在邻接表中常存成两个方向；
- 图论程序阅读先判断方向、边权、存储方式和编号方式。

下一章开始进入连通与强连通。那一章会把“哪些点能互相到达”讲清楚。

0x1A 连通与强连通

第 25 章讲了图的基本语言：点、边、方向、边权、度数、路径和存图方式。本章继续讲图中“能不能到达”的问题。

在无向图中，边没有方向，只要两点之间存在一条路径，就说它们连通。在有向图中，边有方向，从 u 能到 v，不代表从 v 也能回到 u。这一点是本章最重要的区别。

无向图的连通

在无向图中，如果两个点之间存在一条路径，就说这两个点连通。

例如无向边为 $\{1,2\},\{2,3\},\{4,5\}$。点 1 和点 3 连通，因为可以走 $1 \to 2 \to 3$；点 1 和点 4 不连通，因为从 1 出发无论怎么走，都到不了 4。

如果一张无向图中任意两个点都连通，这张图就是连通图。

边集	是否连通
$\{1,2\},\{2,3\},\{3,4\}$	连通
$\{1,2\},\{3,4\}$	不连通
$\{1,2\},\{2,3\},\{4,5\},\{5,6\}$	不连通

无向图的连通只看能不能通过边走到，不要求两个点之间直接有边。只要能经过若干条边到达，就算连通。

例题 1：判断连通

有 5 个点，边为 $\{1,2\},\{2,3\},\{3,1\},\{4,5\}$。这张图是否连通？

点 $1,2,3$ 互相连通，点 $4,5$ 互相连通。但从 $1,2,3$ 中任意一点出发，都不能到达 4 或 5。所以这张图不连通。

它不是因为边数少才不连通，而是因为点被分成了两个互相到不了的部分。

连通块

无向图中，彼此连通的一组点，如果不能再加入更多点，就构成一个连通块。

例如有 7 个点，边为 $\{1,2\},\{2,3\},\{4,5\},\{6,7\}$。这张图有 3 个连通块：$\{1,2,3\}$、$\{4,5\}$、$\{6,7\}$。

如果某个点没有任何边，它自己也是一个连通块。例如点 8 孤立时，$\{8\}$ 单独构成一个连通块。

连通块有两个特点：

- 同一个连通块内部，任意两个点之间都有路径；
- 不同连通块之间，没有任何路径相连。

这两个条件要同时满足。只说“里面的点互相能到达”还不够，还要保证这个集合已经不能继续扩大。

例题 2：数连通块

有 8 个点，边为 $\{1,2\},\{2,3\},\{4,5\},\{5,6\},\{6,4\}$。求连通块个数。

点 $1,2,3$ 在一个连通块中。点 $4,5,6$ 在一个连通块中。点 7 和点 8 没有边，各自单独成为一个连通块。

所以连通块共有 4 个：$\{1,2,3\}$、$\{4,5,6\}$、$\{7\}$、$\{8\}$。

这里点 $4,5,6$ 形成了一个环，但它仍然只是一个连通块。连通块关心的是“能不能到达”，不关心内部有没有环。

用 DFS 标记连通块

程序阅读题中，统计无向图连通块常用 DFS 或 BFS。核心做法是：从一个未访问过的点出发，把它所在的整个连通块都标记掉。

下面是一段常见代码：

```
vector<int> e[N];
bool vis[N];

void dfs(int u) {
    vis[u] = true;
    for (int v : e[u]) {
        if (!vis[v]) {
            dfs(v);
        }
    }
}

int main() {
    cin >> n >> m;
    for (int i = 1; i <= m; i++) {
        int u, v;
        cin >> u >> v;
        e[u].push_back(v);
        e[v].push_back(u);
    }

    int cnt = 0;
    for (int i = 1; i <= n; i++) {
        if (!vis[i]) {
            cnt++;
            dfs(i);
        }
    }

    cout << cnt << endl;
}
```

这里 `cnt` 统计的是连通块个数。每次遇到一个未访问过的点，说明发现了一个新的连通块，于是 `cnt++`，再用 `dfs(i)` 把这个连通块中的所有点标记为已访问。

注意读入无向边时写了两句：

```
e[u].push_back(v);
e[v].push_back(u);
```

这表示边可以双向走。如果这里只写第一句，程序就会按有向图处理，连通块结果可能完全不同。

例题 3：追踪 DFS 标记

无向图有 6 个点，边为 $\{1,2\},\{2,3\},\{4,5\}$。程序按点 1 到 6 扫描未访问点，问 `cnt` 最后是多少？

开始时所有点都未访问。

扫描到点 1，`vis[1]` 为假，`cnt` 变为 1。执行 `dfs(1)` 后，点 $1,2,3$ 都被标记。

扫描到点 $2,3$，它们已经访问过，不会增加 `cnt`。

扫描到点 4，还未访问，`cnt` 变为 2。执行 `dfs(4)` 后，点 $4,5$ 被标记。

扫描到点 5，已经访问过。

扫描到点 6，还未访问，`cnt` 变为 3。执行 `dfs(6)` 后，只标记点 6。

所以最终输出 3。孤立点也会被统计为一个连通块。

BFS 也能做连通性

BFS 同样可以标记一个连通块。与 DFS 不同，BFS 通常使用队列，从起点开始一层一层扩展。

```
void bfs(int s) {
    queue<int> q;
    vis[s] = true;
    q.push(s);

    while (!q.empty()) {
        int u = q.front();
        q.pop();

        for (int v : e[u]) {
            if (!vis[v]) {
                vis[v] = true;
                q.push(v);
            }
        }
    }
}
```

在统计连通块时，DFS 和 BFS 的作用相同：从一个点出发，访问它所在连通块里的所有点。

程序阅读题中，看到 `queue<int>`、`push`、`pop`、`front`，通常是在做 BFS。看到函数自己调用自己，通常是在做 DFS。两种写法都需要关注 `vis` 在什么时候被赋值。

易错点：什么时候标记访问

BFS 中一般在入队时标记：

```
vis[v] = true;
q.push(v);
```

如果等到出队时才标记，同一个点可能被多个相邻点反复加入队列。最后结果未必错，但过程会变复杂，计数类程序还可能出问题。

DFS 中一般进入函数后马上标记：

```
void dfs(int u) {
    vis[u] = true;
    ...
}
```

如果忘记标记访问，在有环的图中，程序会在相邻点之间反复调用，不能正常结束。

有向图的可达

有向图中，边只能按方向走。从 u 到 v 有路径，称为 v 从 u 可达。

例如有向边为 $(1,2),(2,3),(3,1),(3,4)$。从 1 出发可以到达 2、3、4；从 4 出发到不了其他点。

在有向图里，“u 能到 v” 和 “v 能到 u” 是两件不同的事。

有向边	可达关系
$(1,2)$	1 能到 2，2 不一定能到 1
$(1,2),(2,3)$	1 能到 3
$(1,2),(2,1)$	1 和 2 可以互相到达

有向图程序中，读入一条边 (u,v) 时通常只写：

```
e[u].push_back(v);
```

如果额外写 `e[v].push_back(u)`，就把有向边当成无向边了。

例题 4：有向可达性

有向图边为 $(1,2),(2,3),(3,1),(3,4)$。判断下面说法是否正确。

A. 从 1 可以到达 4。

B. 从 4 可以到达 1。

C. 点 $1,2,3$ 可以互相到达。

D. 点 $1,2,3,4$ 可以互相到达。

A 正确。路径是 $1 \to 2 \to 3 \to 4$。

B 错误。点 4 没有指出去的边。

C 正确。$1,2,3$ 构成有向环，任意两个点之间都能沿环到达。

D 错误。虽然 $1,2,3$ 都能到 4，但 4 回不到它们。

这道题的关键是不能把箭头反着走。

强连通

在有向图中，如果任意两个点 u,v 都满足 u 能到 v，且 v 能到 u，这张有向图就是强连通图。

强连通比无向图的连通要求更高。无向图中边可以双向走，只要有路就行；有向图中必须两个方向都可达。

例如有向边为 $(1,2),(2,3),(3,1)$。这张图强连通，因为：

- 1 可以到 2，也可以到 3；
- 2 可以到 3，也可以经 $3 \to 1$ 到 1；
- 3 可以到 1，也可以经 $1 \to 2$ 到 2。

如果把边 $(3,1)$ 删掉，只剩 $(1,2),(2,3)$，就不强连通。此时 1 能到 3，但 3 不能回到 1。

强连通不等于完全有向图

强连通只要求任意两点之间有路径，不要求任意两点之间有直接边。

例如 $(1,2),(2,3),(3,1)$ 是强连通图，但点 1 到点 3 没有直接边。它可以通过路径 $1 \to 2 \to 3$ 到达。

完全有向图要求任意两个不同点之间都有两个方向的直接边。强连通图不需要这么多边。

概念	要求
强连通	任意两点之间双向有路径
完全有向图	任意两点之间双向有直接边

这一区别非常容易出现在选择题里。看到“任意两个顶点之间都有边相连”时，要警惕它说的是直接有边，而不是存在路径。

真题精讲：强连通图的性质

2022 年 CSP-S 第一轮考过这样的问题：

强连通图的性质不包括哪一项？

A. 每个顶点的度数至少为 1

B. 任意两个顶点之间都有边相连

C. 任意两个顶点之间都有路径相连

D. 每个顶点至少都连有一条边

答案是 B。

强连通要求任意两个顶点之间都能通过有向路径互相到达，但不要求它们之间直接有边。一个有向环就可以强连通，例如 $1 \to 2 \to 3 \to 1$。在这个图中，点 1 和点 3 之间没有 $1 \to 3$ 的直接边，但 1 可以经 2 到 3，3 也可以直接到 1。

这道题的关键不是背选项，而是分清“有路径”和“有边”。强连通说的是路径，不是直接相邻。

变式：强连通判断

有向图边为 $(1,2),(2,3),(3,1),(3,4),(4,3)$。这张图是否强连通？

点 $1,2,3$ 之间可以互相到达。点 4 和点 3 之间也可以互相到达，因为有 $(3,4)$ 和 $(4,3)$。

从 4 到 1 可以走 $4 \to 3 \to 1$。从 1 到 4 可以走 $1 \to 2 \to 3 \to 4$。其他点之间也可以通过这些路径互相到达。

所以整张图强连通。

如果删去 $(4,3)$，那么点 $1,2,3$ 能到 4，但 4 回不到它们，整张图就不强连通。

强连通分量

有向图不一定整体强连通，但可以分成若干个强连通分量。强连通分量是一个极大的点集，里面任意两点都能互相到达，并且不能再加入更多点保持这个性质。

例如有向边为 $(1,2),(2,1),(2,3),(3,4),(4,3)$。点 $1,2$ 可以互相到达，构成一个强连通分量；点 $3,4$ 可以互相到达，构成另一个强连通分量。

虽然从 2 可以到 3，但从 3 回不到 2，所以 $\{1,2,3,4\}$ 不是一个强连通分量。

强连通分量要看双向可达，不是单向可达。

点集	是否为强连通分量
$\{1,2\}$	是
$\{3,4\}$	是
$\{1,2,3\}$	不是
$\{2,3\}$	不是

第 26 章只要求知道强连通分量是什么。Tarjan 算法、Kosaraju 算法不是本书这一章的重点。第一轮中更常见的是读懂“哪些点互相可达”，而不是手写求强连通分量的完整模板。

弱连通

有向图还有一个容易混淆的概念：弱连通。

把有向图的所有边都看成无向边后，如果整张图连通，那么这张有向图是弱连通的。

例如有向边为 $(1,2),(2,3)$。这张图不是强连通，因为 3 回不到 1。但如果忽略方向，它变成无向边 $\{1,2\},\{2,3\}$，整张图是连通的，所以它是弱连通的。

有向图	强连通	弱连通
$(1,2),(2,1),(2,3),(3,2)$	是	是
$(1,2),(2,3)$	否	是
$(1,2),(3,4)$	否	否

CSP-S 第一轮中不一定频繁使用“弱连通”这个词，但理解它有助于区分：有向图按方向看和忽略方向看，得到的连通性可能不同。

反图

有向图中，把每条边的方向反过来，得到的图叫反图。

如果原图中有边 (u,v)，反图中就有边 (v,u)。

例如原图边为 $(1,2),(2,3),(1,3)$，反图边为 $(2,1),(3,2),(3,1)$。

反图常用于判断强连通。原因是：如果从某个点 s 出发，在原图中能到达所有点，说明 s 可以走到别人；如果从 s 出发，在反图中也能到达所有点，说明在原图中所有点都能走到 s。两者合起来，就能说明所有点和 s 之间双向可达，从而整张图强连通。

这一思想在程序阅读中可能以两次 DFS 的形式出现。

例题 5：用原图和反图判断

有向图边为 $(1,2),(2,3),(3,1),(3,4),(4,3)$。从点 1 出发，原图 DFS 能访问所有点；反图 DFS 也能访问所有点。能否判断原图强连通？

可以。

原图中从 1 能到所有点，说明 1 能到每个点。反图中从 1 能到所有点，等价于原图中每个点都能到 1。

任取两个点 u,v。因为 u 能到 1，且 1 能到 v，所以 u 能到 v。反过来，v 也能到 u。因此整张图强连通。

这类判断题常把“反图可达”写得很隐蔽，要看清第二次搜索用的是不是反向边。

程序阅读：判断强连通

下面代码判断有向图是否强连通。点编号从 1 到 n。

```
vector<int> e[N], re[N];
bool vis[N];

void dfs(vector<int> g[], int u) {
    vis[u] = true;
    for (int v : g[u]) {
        if (!vis[v]) {
            dfs(g, v);
        }
    }
}
```

```
bool check() {
    memset(vis, 0, sizeof(vis));
    dfs(e, 1);
    for (int i = 1; i <= n; i++) {
        if (!vis[i]) {
            return false;
        }
    }

    memset(vis, 0, sizeof(vis));
    dfs(re, 1);
    for (int i = 1; i <= n; i++) {
        if (!vis[i]) {
            return false;
        }
    }

    return true;
}
```

读入边时：

```
cin >> u >> v;
e[u].push_back(v);
re[v].push_back(u);
```

数组 `e` 存原图，`re` 存反图。`check` 函数先在原图中从 1 出发 DFS，再在反图中从 1 出发 DFS。两次都能访问所有点，才返回 `true`。

注意第二次不是重新在原图中搜一遍，而是在反图中搜。如果两次都搜原图，就只能证明点 1 能到所有点，不能证明所有点都能回到 1。

例题 6：找出程序错误

下面程序想判断有向图是否强连通，但有错误：

```
bool check() {
    memset(vis, 0, sizeof(vis));
    dfs(e, 1);
    for (int i = 1; i <= n; i++) {
        if (!vis[i]) {
            return false;
        }
    }

    memset(vis, 0, sizeof(vis));
    dfs(e, 1);
    for (int i = 1; i <= n; i++) {
        if (!vis[i]) {
            return false;
        }
    }

    return true;
}
```

错误在哪里？

第二次 DFS 仍然使用了原图 `e`，没有使用反图 `re`。这样程序只是在重复检查“从 1 能否到达所有点”。

例如有向边为 $(1,2),(2,3)$。从 1 能到 $1,2,3$，所以第一次检查通过。第二次仍然从原图出发，也会通过。但这张图不是强连通，因为 3 回不到 1。

正确做法是第二次执行 `dfs(re, 1)`。

连通与树

无向图中，连通性和树有密切关系。

含有 n 个点的无向图，如果连通且恰好有 $n-1$ 条边，那么它是一棵树。反过来，树一定连通，并且恰好有 $n-1$ 条边。

这个结论常用于快速判断。

条件	能否推出是树
连通，边数为 $n-1$	能
无环，边数为 $n-1$	能
只有边数为 $n-1$	不能
只有连通	不能

只有边数为 $n-1$ 不够，因为图可能不连通。只有连通也不够，因为图中可能有环。

例题 7：边数和连通性

一张无向图有 7 个点、6 条边。它一定是一棵树吗？

不一定。

树需要连通且边数为 $n-1$。这里边数确实是 $7-1=6$，但题目没有说连通。如果这张图不连通，就不是树。

若题目补充“这张图连通”，那它就是树。

程序阅读中的常见陷阱

连通性程序看起来不长，但第一轮很爱考细节。

第一类陷阱是把有向图当无向图。读入边时，如果题目是有向图，却写了 `add(u,v)` 和 `add(v,u)`，可达性就被改变了。原本不能反向走的路径，会被程序误认为可以走。

第二类陷阱是漏掉孤立点。统计连通块时，外层循环必须扫所有点。孤立点没有边，但仍然是一个连通块。如果只从点 1 出发 DFS，就只能得到点 1 所在的连通块。

第三类陷阱是没有清空 `vis`。如果程序需要做多次 DFS，例如强连通判断中要在原图和反图各搜一次，中间必须重置访问数组。否则第二次搜索会继承第一次的访问结果。

第四类陷阱是误解路径和直接边。强连通要求任意两点之间有路径，不要求直接有边。选择题中出现“任意两点之间都有边”时，通常比“任意两点之间都有路径”更强。

第五类陷阱是编号范围。点编号从 0 开始还是从 1 开始，会影响循环边界。若点编号是 0 到 $n-1$，外层循环应写 `for (int i = 0; i < n; i++)`；若点编号是 1 到 n，则写 `for (int i = 1; i <= n; i++)`。

小结

本章的核心是区分“无向连通”和“有向强连通”。

无向图中，两个点之间有路径就连通；整张图任意两点连通，就是连通图。若图不连通，会分成若干连通块，孤立点也算一个连通块。

有向图中，边有方向。从 u 能到 v，不代表从 v 能到 u。强连通要求任意两点之间双向可达。它说的是“有路径”，不是“有直接边”。

程序阅读中，连通性题要重点看四件事：

- 读入边时加了一次还是两次；
- DFS/BFS 从哪里开始；
- `vis` 是否正确标记和清空；
- 有向图是否用到了反图。

下一章进入欧拉图与二分图。那一章会继续使用本章的连通性概念：欧拉图要关心点的度数和图是否连通，二分图要关心能否把相邻点染成不同颜色。

0x1B 欧拉图与二分图

第 26 章讲了连通与强连通。本章继续讲两个在 CSP-S 第一轮中常见、但容易混淆的图论概念：欧拉图和二分图。

欧拉图关心的是 “能不能一笔走完所有边”，重点在边；二分图关心的是 “能不能把点分成两类，使每条边都跨过两类”，重点在点的染色。两者都不需要复杂代码才能出题，选择题里常用定义、度数、连通性和奇偶性来考。

欧拉路和欧拉回路

欧拉路是一条经过图中每条边恰好一次的路径。可以重复经过点，但不能重复经过边。

欧拉回路是一条起点和终点相同的欧拉路。也就是说，它不仅经过每条边恰好一次，还要最后回到出发点。

注意这里强调的是 “边”。欧拉路可以重复经过某些点，只要每条边只走一次就可以。

例如无向图边为 $\{1,2\},\{2,3\},\{3,1\}$。从 1 出发，可以走 $1\to2\to3\to1$，每条边恰好经过一次，并且回到 1，所以这是欧拉回路。

若无向图边为 $\{1,2\},\{2,3\}$，可以走 $1\to2\to3$，每条边恰好经过一次，但起点和终点不同，所以它有欧拉路，没有欧拉回路。

名称	要求
欧拉路	经过每条边恰好一次
欧拉回路	经过每条边恰好一次，且起点等于终点
欧拉图	通常指存在欧拉回路的图

不同资料对 “欧拉图” 的表述有细微差别。CSP-S 第一轮中，通常按 “存在欧拉回路的图” 来理解。

例题 1：点能否重复

一条走法为 $1\to2\to3\to2\to4$，其中经过的边依次为 $\{1,2\}$、$\{2,3\}$、$\{2,3\}$、$\{2,4\}$。这是否可能是欧拉路？

不是。

虽然点 2 可以重复出现，但边 $\{2,3\}$ 被走了两次。欧拉路要求每条边恰好经过一次，边不能重复。

若走法为 $1\to2\to3\to4\to2$，边依次为 $\{1,2\}$、$\{2,3\}$、$\{3,4\}$、$\{2,4\}$，点 2 出现了两次，但边没有重复，这种走法可以是某张图中的欧拉路。

无向图的欧拉判定

对无向图来说，欧拉回路和点的度数关系很密切。

一张无向图若存在欧拉回路，那么每个有边相连的点度数都必须是偶数。原因是：走到一个点后，除了起点最后回到自身以外，通常要 “进来一次、出去一次”，每次进出会消耗两条与该点相连的边。欧拉回路中起点也一样，因为最后还要回到起点，所以它的相关边也成对出现。

反过来，如果一张无向图中所有非孤立点都在同一个连通块内，并且每个点的度数都是偶数，那么它存在欧拉回路。

这里要注意 “非孤立点”。如果某些点没有任何边，它们不影响是否能把所有边走完。第一轮选择题一般会简化为 “连通无向图”，这时只需要判断所有点度数是否为偶数。

无向图存在欧拉路但不一定存在欧拉回路时，度数条件稍微放宽：奇度点个数必须是 0 或 2。如果奇度点为 0，可以从任意有边的点出发并回到起点，是欧拉回路；如果奇度点为 2，欧拉路必须从其中一个奇度点出发，到另一个奇度点结束。

无向图情况	判定条件
存在欧拉回路	所有非孤立点连通，且所有点度数为偶数
存在欧拉路	所有非孤立点连通，且奇度点个数为 0 或 2
只有欧拉路，没有欧拉回路	所有非孤立点连通，且奇度点个数为 2

例题 2：判断欧拉回路

有一张无向图，边为 $\{1,2\},\{2,3\},\{3,4\},\{4,1\}$。它是否存在欧拉回路？

先看度数。点 $1,2,3,4$ 的度数都是 2。所有点在同一个连通块中，并且度数都是偶数，所以存在欧拉回路。

一条欧拉回路可以是 $1\to2\to3\to4\to1$。

例题 3：只有欧拉路

有一张无向图，边为 $\{1,2\},\{2,3\},\{3,4\}$。它是否存在欧拉路？是否存在欧拉回路？

点 1 和点 4 的度数为 1，点 2 和点 3 的度数为 2。奇度点有 2 个，因此存在欧拉路。

一条欧拉路是 $1\to2\to3\to4$。

但它不存在欧拉回路，因为有奇度点，不能从一个点出发走完所有边后回到原点。

连通性不能漏

判断欧拉图时，不能只看度数，还要看有边的部分是否连通。

例如无向图边为 $\{1,2\},\{2,1\}$ 和 $\{3,4\},\{4,3\}$ 这种说法在简单图中不规范，因为无向简单图没有重复边。换成多重图看，点 $1,2,3,4$ 的度数都可以是偶数，但图分成两块。无论从哪一块出发，都无法走到另一块，所以不可能用一条路走完所有边。

在简单无向图中也能看到类似问题。例如有两个互不相连的三角形：边为 $\{1,2\},\{2,3\},\{3,1\},\{4,5\},\{5,6\},\{6,4\}$。每个点度数都是 2，但图不连通。它不是一张整体上的欧拉图，因为无法用一条回路经过两块中的所有边。

第一轮题中，若题目已经说明“连通无向图”，就不需要再检查连通性；若没有说明连通，就不能只看度数。

例题 4：两个三角形

一张无向图由两个互不相连的三角形组成。它的每个点度数都是 2。这张图是否存在欧拉回路？

不存在。

每个连通块内部都有欧拉回路，但整张图没有一条走法能同时经过两个连通块里的所有边。欧拉回路是一条连续的走法，不能中途跳到另一个连通块。

真题精讲：欧拉图性质

2024 年 CSP-S 第一轮考过一道关于欧拉图的选择题，大意是：

假设有一个包含 n 个顶点的无向图，且该图是欧拉图。以下关于该图的描述中哪一项不一定正确？

A. 所有顶点的度数均为偶数

B. 该图连通

C. 该图存在一个欧拉回路

D. 该图的边数是奇数

按第一轮语境，欧拉图指存在欧拉回路的图。存在欧拉回路时，所有顶点度数为偶数，图的有边部分应连通，也确实存在欧拉回路。所以 A、B、C 都符合欧拉图的基本性质。

边数是否为奇数没有保证。一个三角形有 3 条边，是欧拉图；一个四边形有 4 条边，也是欧拉图。边数可以是奇数，也可以是偶数。

所以“不一定正确”的是 D。

这道题考的不是复杂算法，而是定义。欧拉图由“能否一笔走完并回到起点”和“度数、连通性”决定，不由边数奇偶性直接决定。

变式：边数一定为偶数吗

若一张连通无向图所有点度数都是偶数，那么它的边数一定是偶数吗？

不一定。

三角形中每个点度数都是 2，但边数是 3。所有点度数为偶数，只能说明度数和是偶数。由于度数和等于 $2m$，这个结论对任何无向图都成立，不能推出 m 本身的奇偶性。

有向图中的欧拉回路

有向图也可以讨论欧拉路和欧拉回路，只是判定条件要看入度和出度。

有向图存在欧拉回路时，每个点的入度必须等于出度。原因是每次进入一个点，都要有一次离开它；起点最后也会被回到，所以进出次数仍然相等。

此外，有边参与的点需要在方向意义上连成一个整体。CSP-S 第一轮通常不会深入考有向欧拉图的完整判定，但会考入度、出度是否平衡。

有向图情况	度数特征
欧拉回路	每个点入度等于出度
欧拉路但非回路	一个点出度比入度多 1，一个点入度比出度多 1，其他点入度等于出度

如果存在有向欧拉路但不是回路，那么起点会多出去一次，所以起点满足出度比入度多 1；终点会多进来一次，所以终点满足入度比出度多 1。

例题 5：有向欧拉回路

有向图边为 $(1,2),(2,3),(3,1)$。它是否存在欧拉回路？

点 1 的入度和出度都是 1，点 2 的入度和出度都是 1，点 3 的入度和出度都是 1。并且三点沿方向互相可达。

所以它存在欧拉回路，例如 $1 \to 2 \to 3 \to 1$。

例题 6：有向欧拉路

有向图边为 $(1,2),(2,3),(3,4)$。它是否存在欧拉路？是否存在欧拉回路？

点 1 的出度比入度多 1，点 4 的入度比出度多 1，点 $2,3$ 的入度等于出度。所以它存在从 1 到 4 的欧拉路，即 $1 \to 2 \to 3 \to 4$。

但它不存在欧拉回路，因为点 1 和点 4 的入度、出度不相等。

二分图

二分图是一类可以把点分成左右两部分的无向图。设这两部分为 X 和 Y，要求每条边的两个端点分别属于不同部分，不能出现一条边的两个端点都在 X 内，也不能都在 Y 内。

这两个部分不要求大小相同，也不要求每个点都有边。

例如边为 $\{1,4\},\{1,5\},\{2,4\},\{3,5\}$。可以把点分成 $X=\{1,2,3\}$，$Y=\{4,5\}$。每条边都跨过 X 和 Y，所以这是二分图。

树一定是二分图。因为可以从任意一个点开始，按距离奇偶分层：距离为偶数的点放一边，距离为奇数的点放另一边。树中没有环，不会出现染色冲突。

例题 7：判断点集划分

有图边为 $\{1,3\},\{1,4\},\{2,3\},\{2,5\}$。若分成 $X=\{1,2\}$，$Y=\{3,4,5\}$，这是否说明它是二分图？

是。

检查每条边：$\{1,3\}$ 跨过两边，$\{1,4\}$ 跨过两边，$\{2,3\}$ 跨过两边，$\{2,5\}$ 跨过两边。没有边的两个端点落在同一部分，所以这是一个合法二分划分。

二分图只要求存在一种这样的划分，不要求题目给出的某一种划分都正确。

二染色

判断二分图最常用的方法是二染色。把一个点染成颜色 0，它的所有邻点必须染成颜色 1；颜色 1 的邻点必须染成颜色 0。如果染色过程中遇到一条边连接了两个同色点，就不是二分图。

下面是常见 DFS 写法：

```
vector<int> e[N];
int col[N];

bool dfs(int u, int c) {
    col[u] = c;
    for (int v : e[u]) {
        if (col[v] == -1) {
            if (!dfs(v, c ^ 1)) {
                return false;
            }
        } else if (col[v] == col[u]) {
            return false;
        }
    }
    return true;
}
```

主程序需要先把 `col` 初始化为 -1，表示尚未染色。对于每个未染色的点，都启动一次 DFS：

```
bool ok = true;
memset(col, -1, sizeof(col));

for (int i = 1; i <= n; i++) {
    if (col[i] == -1) {
        if (!dfs(i, 0)) {
            ok = false;
        }
    }
```

```
}
```

为什么要对每个未染色点都启动 DFS？因为图可能不连通。只从点 1 开始，可能只能染到一个连通块，漏掉其他连通块中的冲突。

例题 8：二染色追踪

无向图边为 $\{1,2\},\{2,3\},\{3,4\},\{4,1\}$。从点 1 染颜色 0，能否完成二染色？

点 1 染 0。与点 1 相邻的点 2 和点 4 必须染 1。点 2 相邻的点 3 必须染 0。此时边 $\{3,4\}$ 连接颜色 0 和颜色 1，没有冲突。

所以可以完成二染色，这张图是二分图。一个划分是 $X=\{1,3\}$，$Y=\{2,4\}$。

例题 9：奇环导致冲突

无向图边为 $\{1,2\},\{2,3\},\{3,1\}$。从点 1 染颜色 0，会发生什么？

点 1 染 0。点 2 和点 3 都与点 1 相邻，所以它们都要染 1。但边 $\{2,3\}$ 连接了两个颜色 1 的点，发生冲突。

所以三角形不是二分图。

这说明奇数长度的环会破坏二分图染色。

二分图和奇环

二分图有一个重要判定：一张无向图是二分图，当且仅当它不含奇环。

这里的奇环指长度为奇数的环，例如长度 3、5、7 的环。偶数长度的环不影响二分图，例如四边形可以二染色。

为什么奇环不行？沿着环交替染两种颜色。若环长为奇数，最后会要求起点同时与一个同色点相邻，从而冲突。若环长为偶数，颜色交替一圈后能回到起点的另一侧，不冲突。

图	是否二分图
一条链	是
一棵树	是
四边形	是
三角形	否
五边形	否

真题精讲：完全三叉树与二分图

2023 年 CSP-S 第一轮考过一道选择题，问以下连通无向图中哪一种一定可以用不超过两种颜色染色。选项中有完全三叉树、平面图、边双连通图、欧拉图。

答案是完全三叉树。

完全三叉树仍然是一棵树。树没有环，所以更不可能有奇环，因此一定是二分图，也就可以用两种颜色染色。

平面图不一定能用两种颜色染色，例如三角形就是平面图，但不是二分图。边双连通图也不一定，例如一个三角形就是边双连通的。欧拉图也不一定，例如三角形每个点度数都是 2，是欧拉图，但它有奇环，不是二分图。

这道题把多个图论概念放在一起考。判断时不要被“平面图”“欧拉图”这些词带偏，题目问的是能否二染色，本质上要找一定没有奇环的结构。树一定没有环，所以完全三叉树一定可以。

变式：欧拉图一定是二分图吗

欧拉图一定是二分图吗？

不一定。

三角形是一张欧拉图，因为每个点度数都是 2，并且连通，存在欧拉回路 $1 \to 2 \to 3 \to 1$。但三角形有长度为 3 的奇环，不能二染色，所以不是二分图。

二分图和欧拉图关注的是两件不同的事。欧拉图看度数和连通性，二分图看能否二染色和是否存在奇环。

完全二分图

如果二分图两侧点集为 X 和 Y，并且 X 中每个点都与 Y 中每个点相连，这样的图叫完全二分图，记作 $K_{a,b}$，其中 $a = |X|$，$b = |Y|$。

完全二分图 $K_{a,b}$ 的边数为 ab。因为左边每个点都连向右边 b 个点，共有 a 个左侧点。

例如 $K_{2,3}$ 有 2 个左侧点、3 个右侧点，边数为 $2 \times 3 = 6$。

注意完全二分图不是完全图。完全图中任意两个不同点之间都有边；完全二分图中，同一侧内部没有边。

图	边的要求
完全图 K_n	任意两个不同点之间都有边
完全二分图 $K_{a,b}$	只要求两侧之间全部相连，同侧内部没有边

例题 10：完全二分图边数

完全二分图 $K_{4,5}$ 有多少条边？

左侧有 4 个点，右侧有 5 个点。每个左侧点都连向右侧 5 个点，所以边数为 $4 \times 5 = 20$。

若把它误当作 9 个点的完全图，会得到 $\binom{9}{2} = 36$，这是错误的。完全二分图同侧内部没有边。

二分图中的度数和边数

在二分图中，所有边都跨过左右两侧。因此从左侧统计度数，得到的总数等于边数；从右侧统计度数，也得到边数。

如果二分图两侧为 X 和 Y，边数为 m，那么 $\sum_{u \in X} \deg(u) = m$，$\sum_{v \in Y} \deg(v) = m$。

这个性质常用于快速检查计数是否合理。

例如一张二分图左侧点度数分别为 $2, 3, 1$，那么边数为 $2 + 3 + 1 = 6$。右侧所有点的度数和也必须是 6，否则给出的度数信息不可能来自同一张二分图。

例题 11：度数检查

一张二分图左侧有 3 个点，度数为 $2, 2, 1$；右侧有 2 个点，度数为 $3, 2$。这些度数是否可能对应同一张二分图？

左侧度数和为 $2 + 2 + 1 = 5$。右侧度数和为 $3 + 2 = 5$。两侧度数和相等，至少通过了边数检查。

这不等于一定能构造出来，但如果两侧度数和不同，就一定不可能。

程序阅读：二分图判定

下面代码判断无向图是否是二分图：

```
vector<int> e[N];
int col[N];

bool dfs(int u, int c) {
```

```
    col[u] = c;
    for (int v : e[u]) {
        if (col[v] == -1) {
            if (!dfs(v, c ^ 1)) {
                return false;
            }
        } else if (col[v] == col[u]) {
            return false;
        }
    }
    return true;
}

int main() {
    cin >> n >> m;
    for (int i = 1; i <= m; i++) {
        int u, v;
        cin >> u >> v;
        e[u].push_back(v);
        e[v].push_back(u);
    }

    memset(col, -1, sizeof(col));

    bool ok = true;
    for (int i = 1; i <= n; i++) {
        if (col[i] == -1) {
            if (!dfs(i, 0)) {
                ok = false;
            }
        }
    }

    cout << (ok ? "Yes" : "No") << endl;
}
```

阅读这段程序时，要抓住三个点。

第一，图按无向图存储，所以每条边加入了两个方向。

第二，`col[u]` 不是访问数组，而是颜色数组。`-1` 表示未染色，0 和 1 表示两种颜色。

第三，外层循环扫描了所有点。这样即使图不连通，也能检查每个连通块。

例题 12：删掉外层循环会怎样

若只执行 `dfs(1, 0)`，不再扫描其他未染色点，可能出现什么问题？

如果图不连通，程序只能检查点 1 所在的连通块，其他连通块不会被检查。

例如有两个连通块：第一个连通块是一条边 $\{1,2\}$，第二个连通块是三角形 $\{3,4\},\{4,5\},\{5,3\}$。从点 1 出发只能染到 $1,2$，不会发现 $3,4,5$ 中的奇环冲突。程序可能错误输出二分图。

所以判断整张图是否为二分图，必须处理所有连通块。

欧拉图和二分图的对比

欧拉图和二分图都与“奇偶性”有关，但奇偶性出现的位置不同。

欧拉图看的是每个点的度数奇偶。无向欧拉回路要求所有点度数为偶数。

二分图看的是环的长度奇偶。二分图不能含有奇环。

概念	关注对象	关键判定
欧拉图	边能否一次走完	连通性与度数
欧拉路	边能否一次走完但不一定回到起点	奇度点个数为 0 或 2
二分图	点能否分成两侧	能否二染色
非二分图	点染色出现冲突	含有奇环

三角形是一个很好的对比例子。它每个点度数都是 2，所以是欧拉图；但它有长度为 3 的奇环，所以不是二分图。

四边形每个点度数都是 2，且没有奇环，所以既是欧拉图，也是二分图。

一条长度为 3 的链有两个奇度点，所以有欧拉路但没有欧拉回路；它是一棵树，所以是二分图。

例题 13：综合判断

判断下面三个图形分别是否为欧拉图、是否为二分图。

A. 三角形

B. 四边形

C. 五边形

三角形中每个点度数为 2，连通，所以是欧拉图；它有奇环，所以不是二分图。

四边形中每个点度数为 2，连通，所以是欧拉图；它是偶环，可以二染色，所以是二分图。

五边形中每个点度数为 2，连通，所以是欧拉图；它是奇环，所以不是二分图。

图	欧拉图	二分图
三角形	是	否
四边形	是	是
五边形	是	否

这个表说明：欧拉图和二分图没有包含关系。一个图可以是欧拉图但不是二分图，也可以是二分图但不是欧拉图，还可以两者都是或两者都不是。

小结

欧拉图和二分图都属于 CSP-S 第一轮中适合用概念和手算考查的图论内容。

欧拉图要抓住 “边”。欧拉路要求每条边恰好走一次；欧拉回路还要求回到起点。无向连通图存在欧拉回路，当且仅当所有点度数都是偶数；存在欧拉路，当且仅当奇度点个数为 0 或 2。

二分图要抓住 “点”。它要求点能分成两侧，每条边都跨过两侧。判断二分图常用二染色；若染色过程中出现同色相邻，就不是二分图。二分图等价于不含奇环，树一定是二分图。

本章最容易混淆的是两点：

- 欧拉图看点的度数，不看环长是否为奇数；
- 二分图看能否二染色，不要求所有点度数为偶数。

下一章进入 DAG 与拓扑排序。那里会继续强调有向图中的方向限制，并用入度、出度和无环性质来判断程序流程。

0x1C DAG 与拓扑排序

第 27 章讲了欧拉图与二分图。本章继续讨论有向图中的一种重要结构：有向无环图，也就是 DAG。它的核心限制很简单：边有方向，并且沿着方向不能绕回原点。

拓扑排序是 DAG 最常见的考法。它不是普通排序，而是按照 “谁必须在谁之前” 排出一个合法顺序。CSP-S 第一轮中，拓扑排序常出现在选择题、程序阅读和完善程序里，考查点包括：是否存在拓扑序、拓扑序是否唯一、入度如何变化、队列中可选点如何影响答案。

DAG

DAG 是 Directed Acyclic Graph 的缩写，中文通常叫有向无环图。

这个名字包含两个条件：

- 它是有向图，边有方向；
- 它没有有向环。

例如有向边为 $(1,2),(1,3),(2,4),(3,4)$。沿着边的方向只能从前面的点走向后面的点，无法回到已经经过的点，所以这是一张 DAG。

如果边为 $(1,2),(2,3),(3,1)$，那么存在有向环 $1\to2\to3\to1$，它不是 DAG。

注意：DAG 禁止的是有向环。无向意义下看起来像环，不一定违反 DAG。

例如边为 $(1,2),(1,3),(2,4),(3,4)$。如果忽略方向，$1,2,4,3$ 形成一个四边形；但沿有向边走，不能从 4 回到 1，所以它仍然是 DAG。

例题 1：判断是否为 DAG

判断下面三张有向图是否为 DAG。

A. 边为 $(1,2),(2,3),(3,4)$。

B. 边为 $(1,2),(2,3),(3,1)$。

C. 边为 $(1,2),(1,3),(2,4),(3,4)$。

A 是 DAG。沿方向只能从 1 走到 4，没有回路。

B 不是 DAG。它有有向环 $1\to2\to3\to1$。

C 是 DAG。虽然忽略方向后有环的形状，但不存在沿箭头方向走一圈回到起点的路径。

判断 DAG 时，要始终按箭头方向走，不能把有向边当成无向边。

拓扑序

给定一张有向图，如果能把所有点排成一个序列，使得每条有向边 (u,v) 都满足 u 排在 v 前面，这个序列就叫拓扑序。

例如边为 $(1,3),(2,3),(3,4)$。序列 $1,2,3,4$ 是一个拓扑序，因为 1 在 3 前面，2 在 3 前面，3 在 4 前面。序列 $2,1,3,4$ 也是拓扑序。

但序列 $1,3,2,4$ 不是拓扑序，因为边 $(2,3)$ 要求 2 在 3 前面，而这个序列中 3 在 2 前面。

拓扑序的含义可以理解为“先做前置任务，再做后续任务”。如果有边 $u \to v$，就表示 u 必须排在 v 前面。

一张有向图存在拓扑序，当且仅当它是 DAG。只要有有向环，就无法排出拓扑序。因为环中每个点都要求排在下一个点之前，最后又要求某个点排在自己之前，产生矛盾。

例题 2：判断序列是否合法

有向图边为 $(1,3),(2,3),(3,5),(4,5)$。判断序列 $2,1,4,3,5$ 是否为拓扑序。

逐条检查边：

- $(1,3)$ 中，1 在 3 前面；
- $(2,3)$ 中，2 在 3 前面；
- $(3,5)$ 中，3 在 5 前面；
- $(4,5)$ 中，4 在 5 前面。

所有边都满足起点在终点前面，所以 $2,1,4,3,5$ 是拓扑序。

判断一个给定序列是否为拓扑序时，不需要重新排序，只要检查所有边的相对位置是否正确。

拓扑序不一定唯一

一张 DAG 可以有多个拓扑序。

例如边为 $(1,3),(2,3)$。点 1 和点 2 之间没有先后限制，所以 $1,2,3$ 和 $2,1,3$ 都是拓扑序。

若图中没有任何边，n 个点可以任意排列，拓扑序有 $n!$ 种。

若边为 $(1,2),(2,3),(3,4)$，那么只能按 $1,2,3,4$ 排，拓扑序唯一。

所以“有拓扑序”不等于“只有一种拓扑序”。第一轮选择题经常会把这两件事混在一起考。

真题式例题：拓扑排序有多少种

一个包含 n 个结点和 m 条边的 DAG，其拓扑排序的结果有多少种可能？

A. 只有 1 种

B. 最多 n 种

C. 等于 $n-m$ 种

D. 以上都不对

答案是 D。

DAG 的拓扑序可能只有 1 种，也可能有很多种。例如一条链 $1 \to 2 \to 3$ 只有一种拓扑序；若 n 个点之间没有边，则任意排列都是拓扑序，共有 $n!$ 种。

它也不等于 $n-m$。边数只能反映限制数量的一部分，不能直接决定拓扑序数量。

这类题要用反例排除。只要找到一种 DAG 有多种拓扑序，就能排除“只有 1 种”；找到空图有 $n!$ 种拓扑序，就能排除“最多 n 种”。

变式：什么时候唯一

若一张 DAG 的拓扑排序过程中，每一步可选的入度为 0 的点都只有一个，那么拓扑序是否唯一？

唯一。

因为每一步没有选择余地，当前点只能放在序列的当前位置。重复这个过程直到所有点被取出，整个序列被完全确定。

反过来，如果某一步有两个或更多入度为 0 的点可以选，通常就能得到不同拓扑序。把这两个点的选择顺序交换，不会破坏已经满足的前置关系。

入度为 0 的点

拓扑排序最常见的做法是不断取出入度为 0 的点。

入度为 0 的点没有任何尚未完成的前置点，所以可以放到当前拓扑序的下一个位置。取出这个点后，把它连出去的边删掉，相当于它对后续点的前置限制已经完成。某些点的入度会因此减少，若变为 0，就可以加入候选集合。

这个过程称为 Kahn 算法。CSP-S 第一轮中不一定直接叫这个名字，但代码结构很常见。

基本流程是：

1. 统计每个点的入度；
2. 把所有入度为 0 的点加入队列；
3. 每次取出一个点，加入拓扑序；
4. 枚举它的出边，让终点入度减 1；
5. 若某个终点入度变成 0，加入队列；
6. 若最后取出的点数为 n，说明存在拓扑序；否则图中有有向环。

例题 3：手算拓扑排序

有向图边为 $(1,3),(2,3),(3,4),(2,5)$。用入度为 0 的点做拓扑排序。

先统计入度：

点	入度
1	0
2	0
3	2
4	1
5	1

一开始可选点为 $1,2$。若先取 1，删除边 $(1,3)$ 后，点 3 入度从 2 变成 1，还不能选。

再取 2，删除边 $(2,3)$ 和 $(2,5)$，点 3 入度变成 0，点 5 入度变成 0。此时可选点为 $3,5$。

若取 3，删除边 $(3,4)$ 后，点 4 入度变成 0。之后可以取 5 和 4。

一种拓扑序是 $1,2,3,5,4$。另一种拓扑序是 $2,1,5,3,4$。只要所有边的起点都在终点前面，就是合法的。

队列代码

下面是拓扑排序的常见代码。图用邻接表存储，点编号从 1 到 n。

```
vector<int> e[N];
int deg[N];
int ans[N], cnt;

bool topo() {
    queue<int> q;
```

```
    for (int i = 1; i <= n; i++) {
        if (deg[i] == 0) {
            q.push(i);
        }
    }

    while (!q.empty()) {
        int u = q.front();
        q.pop();

        ans[++cnt] = u;

        for (int v : e[u]) {
            deg[v]--;
            if (deg[v] == 0) {
                q.push(v);
            }
        }
    }

    return cnt == n;
}
```

这里 `deg[v]` 表示当前还没有被删除的入边数量。每取出一个点 u，就枚举 u 指向的点 v，把 `deg[v]` 减 1。

最后 `cnt == n` 表示所有点都被取出，图中没有有向环；若 `cnt < n`，说明剩下的点没有入度为 0 的点可取，图中存在有向环。

例题 4：为什么用入度

拓扑排序中为什么选入度为 0 的点，而不是出度为 0 的点？

边 (u, v) 表示 u 必须排在 v 前面。入度为 0 的点没有任何点要求排在它前面，所以它可以放在当前最前面。

出度为 0 的点没有后继，它更适合从后往前构造拓扑序。如果从前往后生成拓扑序，应该优先选择入度为 0 的点。

例如边为 $(1, 2), (2, 3)$。入度为 0 的点是 1，它应该排在最前面；出度为 0 的点是 3，它应该排在最后面。

检测有向环

拓扑排序可以判断有向图中是否有环。

如果一张有向图是 DAG，那么总能找到至少一个入度为 0 的点。取出它并删除出边后，剩下的图仍然是 DAG，也继续能找到入度为 0 的点。最终可以取出所有点。

如果图中存在有向环，环上的每个点都至少有一条来自环内的入边。只靠删除环外的点，无法把环中所有点的入度降为 0。当队列为空但还有点没取出时，就说明存在有向环。

例题 5：环导致失败

有向图边为 $(1, 2), (2, 3), (3, 1), (3, 4)$。执行拓扑排序会怎样？

各点入度为：点 1 入度 1，点 2 入度 1，点 3 入度 1，点 4 入度 1。一开始没有入度为 0 的点，队列为空。

但图中还有 4 个点没有取出，所以拓扑排序失败。这张图存在有向环 $1 \to 2 \to 3 \to 1$。

注意点 4 虽然不在环上，但它的入边来自点 3。点 3 取不出来，点 4 的入度也降不下来。

删除边与拓扑序

有向图中如果没有拓扑序，原因一定是存在有向环。要让它有拓扑序，必须破坏所有有向环。删除一条边是否有效，要看它是否能打断导致无拓扑序的环。

如果一张图只有一个有向环，那么删除环上的任意一条边，都能破坏这个环；删除不在环上的边，则无法解决这个环。

如果有多个有向环，删除一条边必须同时破坏所有有向环，才能让整张图变成 DAG。

真题精讲：删除边后能拓扑排序

2023 年 CSP-S 第一轮出现过一类题：给出一张有向图，图中不存在拓扑序，问删除其中一条边后，使所有顶点能进行拓扑排序的候选边有多少条。

这类题的处理方法不是随便找“看起来多余”的边，而是先找有向环。没有拓扑序说明图中至少有一个有向环。删除边后要能拓扑排序，删除的边必须让图中不再存在任何有向环。

若图中只有一个环，答案通常就是这个环上的边数。若图中有多个环，就要看哪些边是所有环共同必须经过的边。只有删除这样的边，才能一次打断所有环。

纸面做这种题时，可以按下面步骤：

1. 先沿箭头找出所有有向环；
2. 标出每个环上的边；
3. 检查删除某条边后是否还剩其他环；
4. 能让所有环消失的边才是候选边。

如果图画得较复杂，不要只凭直觉删边。拓扑序存在与否只由有向环决定。

变式：两个环共享一条边

有向图中有两个有向环：$1 \to 2 \to 3 \to 1$ 和 $1 \to 2 \to 4 \to 1$。如果只能删除一条边，哪些边可能让图变成 DAG？

两个环共同包含边 $(1, 2)$。删除 $(1, 2)$ 后，两个环都会被打断。

若删除 $(2, 3)$，第一个环被打断，但第二个环 $1 \to 2 \to 4 \to 1$ 仍存在。若删除 $(2, 4)$，第二个环被打断，但第一个环仍存在。

所以在这部分结构中，只有删除 $(1, 2)$ 能同时破坏两个环。

DFS 与拓扑序

除了入度队列法，DFS 也能生成拓扑序。在一张 DAG 中，如果从一个点出发，先递归访问它的所有后继，再把这个点加入序列，那么最后把加入顺序反过来，得到拓扑序。

代码常写成：

```
vector<int> e[N];
bool vis[N];
vector<int> ord;

void dfs(int u) {
    vis[u] = true;
    for (int v : e[u]) {
        if (!vis[v]) {
            dfs(v);
```

```
        }
    }
    ord.push_back(u);
}
```

所有点都 DFS 后，`ord` 的逆序就是一个拓扑序。

为什么要逆序？因为 `u` 会在它的所有后继都处理完之后才加入 `ord`。也就是说，后继会更早进入 `ord`，前驱会更晚进入 `ord`。反过来以后，前驱就在后继前面了。

例题 6：DFS 加入顺序

有向边为 $(1,2),(1,3),(3,4)$。从点 1 开始 DFS，邻点按编号从小到大访问。求 `ord` 的加入顺序和拓扑序。

从 1 访问 2。点 2 没有后继，加入 `ord`，此时 `ord=[2]`。

回到 1，再访问 3。从 3 访问 4。点 4 没有后继，加入 `ord`，此时 `ord=[2,4]`。

回到 3，加入 3，此时 `ord=[2,4,3]`。

回到 1，加入 1，此时 `ord=[2,4,3,1]`。

把它反过来，得到 $1,3,4,2$，这是一个合法拓扑序。边 $(1,2)$、$(1,3)$、$(3,4)$ 都满足起点在终点前面。

注意 2 和 3 之间没有先后限制，所以 $1,2,3,4$ 也是拓扑序。DFS 得到的只是其中一种。

DFS 判环

DFS 也能判断有向图是否有环。常用三种颜色表示状态：

- 0：还没访问；
- 1：正在当前递归路径上；
- 2：已经处理完成。

如果从 u 走到一个颜色为 1 的点 v，说明从 v 出发经过若干边到达了 u，现在又有边回到 v，形成有向环。

代码如下：

```
vector<int> e[N];
int col[N];

bool dfs(int u) {
    col[u] = 1;

    for (int v : e[u]) {
        if (col[v] == 0) {
            if (!dfs(v)) {
                return false;
            }
        } else if (col[v] == 1) {
            return false;
        }
    }

    col[u] = 2;
    return true;
}
```

主程序中对每个未访问点启动 DFS。若某次返回 `false`，说明有有向环，不是 DAG。

这段代码中，颜色 1 和颜色 2 的区别很关键。颜色 1 表示点仍在当前递归链上，碰到它才说明有环；颜色 2 表示这个点已经处理完，从当前路径连向它不构成新的环。

例题 7：为什么不能只用 vis

若用一个 `vis[u]` 表示点是否访问过，在有向图中能否准确判断有环？

不能。

例如边为 $(1,2),(1,3),(2,4),(3,4)$。从 1 出发，先访问 2，再访问 4，此时 4 被标记为已访问。回到 1 后访问 3，从 3 又遇到 4。如果只看到 `vis[4]=true` 就认为有环，就会误判。

这里没有环，只是两条路径汇合到了同一个点。

因此有向图 DFS 判环要区分 “正在递归路径上” 和 “已经处理完成”。三色标记正是为了解决这个问题。

拓扑排序中的字典序

如果每一步有多个入度为 0 的点，选哪个都会得到合法拓扑序。若题目要求字典序最小的拓扑序，就不能用普通队列，而要用小根堆或有序集合，每次取编号最小的可选点。

普通队列取决于入队顺序，不保证字典序最小。

例如边为 $(1,3),(2,3),(4,3)$。初始入度为 0 的点有 $1,2,4$。若按编号从小到大入队，普通队列得到 $1,2,4,3$；若输入边顺序导致 4 先入队，可能得到 $4,1,2,3$。它仍然是拓扑序，但不是字典序最小。

使用小根堆：

```
priority_queue<int, vector<int>, greater<int>> q;
```

每次取出的都是当前可选点中编号最小的点，才能保证字典序最小。

例题 8：字典序最小拓扑序

有向边为 $(1,4),(2,4),(3,4)$。求字典序最小拓扑序。

初始可选点为 $1,2,3$。为了字典序最小，先取 1，再取 2，再取 3。此时点 4 的三个前驱都已取出，入度变为 0，最后取 4。

答案为 $1,2,3,4$。

若输出 $2,1,3,4$，它也是拓扑序，但不是字典序最小。

程序阅读中的拓扑排序

拓扑排序程序阅读题常考变量含义和顺序变化。读这类代码时，可以按下面几步处理。

第一，看 `deg` 表示入度还是出度。拓扑排序从前往后生成时，`deg` 通常表示入度。如果代码枚举出边后执行 `deg[v]--`，就能确认这一点。

第二，看初始队列里放了哪些点。若放入 `deg[i] == 0` 的点，就是从前往后生成拓扑序；若放入出度为 0 的点，可能是在反向处理。

第三，看容器类型。`queue` 表示按入队顺序处理，`priority_queue` 可能表示每次取最大或最小编号。若题目问具体输出，容器类型会影响答案。

第四，看是否统计 `cnt`。若最后判断 `cnt == n`，程序通常在检测是否存在有向环。

第五，看图是否被重复使用。拓扑排序会修改入度数组。如果后面还需要原始入度，必须事先备份。程序阅读题可能会故意在修改后的 `deg` 上继续判断，导致含义变化。

例题 9：追踪队列输出

有向边按顺序读入：$(1,3),(2,3),(3,4)$。使用普通队列进行拓扑排序，初始扫描点编号从 1 到 4。输出序列是什么？

初始入度：点 1 为 0，点 2 为 0，点 3 为 2，点 4 为 1。扫描编号从小到大，所以队列初始为 $1,2$。

取出 1，输出 1，点 3 入度从 2 变成 1。

取出 2，输出 2，点 3 入度从 1 变成 0，入队。

取出 3，输出 3，点 4 入度从 1 变成 0，入队。

取出 4，输出 4。

所以输出序列为 $1,2,3,4$。

如果初始扫描顺序不同，或者使用不同容器，输出可能变成另一种合法拓扑序。

完善程序中的空缺

拓扑排序也常出现在完善程序题里。空缺位置一般围绕入度、队列、删边和计数。

常见空缺有：

空缺位置	应关注的问题
读入边后	是否要 `deg[v]++`
初始化队列	是否加入 `deg[i] == 0` 的点
取出队首后	是否记录到答案数组
枚举出边时	是否执行 `--deg[v]`
入队条件	是否为 `deg[v] == 0`
最后判断	是否为 `cnt == n`

例如下面片段：

```
for (int v : e[u]) {
    deg[v]--;
    if (__________) {
        q.push(v);
    }
}
```

空缺处应填 `deg[v] == 0`。因为只有当前驱都已经被取出，点 v 才能加入拓扑序候选集合。

若填 `deg[v] <= 0`，在有重边或程序其他地方重复减入度时，可能把点多次加入队列。若填 `deg[v] == 1`，则会提前加入，还没有满足所有前置条件。

例题 10：补全入度更新

下面代码用于拓扑排序：

```
for (int i = 1; i <= m; i++) {
    int u, v;
    cin >> u >> v;
    e[u].push_back(v);
    ________;
}
```

空缺处应填什么？

应填 `deg[v]++`。

因为边 (u,v) 表示 u 指向 v，会给 v 增加一条入边。拓扑排序需要知道每个点还有多少前驱没有处理，所以读入边时要统计终点的入度。

如果误写成 `deg[u]++`，统计的是出边数量，不适合从前往后做拓扑排序。

DAG 上的简单 DP

DAG 没有有向环，所以可以按拓扑序做动态规划。虽然本章不展开 DP，但第一轮程序阅读中可能出现“先拓扑排序，再按顺序转移”的代码。

例如要求从起点 s 到每个点的路径条数。若图是 DAG，可以先求拓扑序，再按拓扑序处理：

```
dp[s] = 1;

for (int i = 1; i <= n; i++) {
    int u = ans[i];
    for (int v : e[u]) {
        dp[v] += dp[u];
    }
}
```

这里必须保证 u 在 v 前面被处理。拓扑序正好满足这个条件。

若图中有环，路径条数可能受到循环影响，这种简单顺序转移就不再适用。CSP-S 第一轮中看到 DAG 上的 DP，要先确认程序是不是利用了“前驱先处理，后继后处理”。

例题 11：路径条数

有向边为 $(1,2),(1,3),(2,4),(3,4)$。求从点 1 到点 4 的路径条数。

这是一张 DAG。拓扑序可以是 $1,2,3,4$。

设 `dp[u]` 表示从 1 到 u 的路径条数。开始 `dp[1]=1`。

处理点 1，它能到 2 和 3，所以 `dp[2]=1`，`dp[3]=1`。

处理点 2，它能到 4，所以 `dp[4]` 加上 `dp[2]`，变为 1。

处理点 3，它也能到 4，所以 `dp[4]` 再加上 `dp[3]`，变为 2。

所以从 1 到 4 有 2 条路径：$1\to2\to4$ 和 $1\to3\to4$。

小结

DAG 是有向无环图。判断它时，要沿着边的方向看是否能走出有向环，不能只看图形上是否有环的形状。

拓扑序是 DAG 中的合法线性顺序。对于每条边 (u,v)，u 必须排在 v 前面。一张有向图存在拓扑序，当且仅当它没有有向环。

拓扑排序最常见的程序是入度队列法：先把所有入度为 0 的点入队，每次取出一个点，删除它连出的边，并更新后继的入度。若最后取出的点数少于 n，说明图中有有向环。

本章需要重点记住：

- DAG 禁止的是有向环；
- 拓扑序不一定唯一；
- 入度为 0 的点可以放在当前最前面；

- 队列、优先队列会影响具体输出；
- `cnt == n` 常用于判断是否成功完成拓扑排序；
- DFS 判环要区分 “正在访问” 和 “已经完成”；
- DAG 上可以按拓扑序做顺序转移。

下一章进入最小生成树。那里会从无向带权图出发，研究如何在保持连通的前提下选出总代价最小的一组边。

0x1D 最小生成树

第 28 章讲的是有向图中的 DAG 与拓扑排序。本章回到无向图，讨论带权无向图中的最小生成树。

最小生成树的问题可以这样理解：有若干个点，点与点之间可以用不同代价的边连接。我们希望选出一些边，让所有点连成一个整体，并且总代价尽量小。选出的边不能形成环，因为环中至少有一条边对保持连通不是必需的。

CSP-S 第一轮不会要求学生写很长的最小生成树模板，但会考查几个关键点：生成树有多少条边，Kruskal 按什么顺序选边，Prim 每一步在选什么，哪些边一定不可能在最小生成树中，以及最小生成树和最短路的区别。

生成树

在一张无向连通图中，若选出一些边，使得所有点仍然连通，并且没有环，那么这些边构成一棵生成树。

“生成” 表示覆盖原图的所有点；“树” 表示连通且无环。

如果原图有 n 个点，那么任意一棵生成树都有 $n-1$ 条边。这个结论很常用：选少于 $n-1$ 条边，不可能把所有点连起来；选多于 $n-1$ 条边，在连通的情况下必然产生环。

例如一张图有 5 个点，那么任何生成树都必须有 4 条边。它可能有很多种生成树，但每一种都恰好选 4 条边。

例题 1：判断是否为生成树

一张无向图有 6 个点。某个选边方案选出了 5 条边，并且这 6 个点连通。这个方案一定是一棵生成树吗？

一定是。

含 6 个点的连通无向图，如果边数为 $6-1=5$，就没有多余的环边，所以是一棵树。又因为它覆盖了原图的所有 6 个点，所以是一棵生成树。

如果只知道选了 5 条边，但不知道是否连通，就不能判断它是生成树。边数正确只是必要条件，还要保证所有点连成一个整体。

最小生成树

如果原图的边带权，那么不同生成树的边权和可能不同。边权和最小的生成树，叫最小生成树，常写作 MST。

例如有 4 个点，边如下：

边	权值
$\{1,2\}$	1
$\{1,3\}$	4
$\{2,3\}$	2
$\{2,4\}$	5
$\{3,4\}$	3

一种生成树可以选 $\{1,2\}$、$\{2,3\}$、$\{3,4\}$，边权和为 $1+2+3=6$。另一种生成树可以选 $\{1,3\}$、$\{2,3\}$、$\{2,4\}$，边权和为 $4+2+5=11$。前一种更小。

最小生成树只要求把所有点连起来，并不要求任意两点之间的路径最短。它优化的是整棵树的总边权，不是某一对点之间的距离。

最小生成树和最短路

这是第一轮很容易混淆的一组概念。

最小生成树关注“选一批边，把所有点连起来，总权值最小”。最短路关注“从一个点到另一个点，路径权值和最小”。

同一张图中，最小生成树上的两点路径不一定是原图中的最短路。

例如有边 $\{1,2\}$ 权值 1，$\{2,3\}$ 权值 1，$\{1,3\}$ 权值 1.5。最小生成树会选 $\{1,2\}$ 和 $\{2,3\}$，总权值为 2。在这棵树中，从 1 到 3 的路径长度为 2。但原图中有直接边 $\{1,3\}$，长度为 1.5，它才是 1 到 3 的最短路。

所以最小生成树不能替代最短路。

问题	目标
最小生成树	连接所有点，总边权最小
最短路	从指定起点到指定终点或其他点，路径代价最小

Kruskal 算法

Kruskal 算法的思想很直接：把所有边按权值从小到大排序，然后依次考虑每条边。如果这条边连接了两个不同的连通块，就选它；如果它的两个端点已经在同一个连通块中，选它会形成环，就跳过。

选到 $n-1$ 条边时，生成树已经完成。

它的关键不是“选最小的 $n-1$ 条边”，而是“按权值从小到大尝试，不能形成环”。如果只选最小的 $n-1$ 条边，可能这些边在局部形成环，反而没有连通所有点。

Kruskal 常配合并查集判断两个点是否已经连通：

```
struct Edge {
    int u, v, w;
};

Edge e[M];
int fa[N];

int find(int x) {
    if (fa[x] == x) {
        return x;
    }
    return fa[x] = find(fa[x]);
}

bool cmp(Edge a, Edge b) {
    return a.w < b.w;
}

int kruskal() {
    sort(e + 1, e + m + 1, cmp);

    for (int i = 1; i <= n; i++) {
        fa[i] = i;
    }

    int cnt = 0, sum = 0;

    for (int i = 1; i <= m; i++) {
        int x = find(e[i].u);
        int y = find(e[i].v);
```

```
        if (x == y) {
            continue;
        }

        fa[x] = y;
        sum += e[i].w;
        cnt++;

        if (cnt == n - 1) {
            break;
        }
    }

    return sum;
}
```

这里 `cnt` 表示已经选了多少条边，`sum` 表示当前选中边的权值和。若图连通，最后应选出 $n-1$ 条边。

例题 2：手算 Kruskal

有 4 个点，边如下：

边	权值
$\{1,2\}$	1
$\{2,3\}$	2
$\{1,3\}$	3
$\{3,4\}$	4
$\{2,4\}$	5

按权值从小到大考虑。

先看 $\{1,2\}$，点 1 和点 2 不连通，选入。

再看 $\{2,3\}$，点 2 和点 3 不连通，选入。

再看 $\{1,3\}$。此时 $1,2,3$ 已经在同一个连通块中，若选 $\{1,3\}$ 会形成环 $1-2-3-1$，跳过。

再看 $\{3,4\}$，点 3 和点 4 不连通，选入。

现在已经选了 3 条边，而 $n=4$，生成树需要 $n-1=3$ 条边，所以停止。最小生成树总权值为 $1+2+4=7$。

易错：不是最小的几条边

仍看上面的图。权值最小的三条边是 $\{1,2\}$、$\{2,3\}$、$\{1,3\}$，权值和为 $1+2+3=6$。但这三条边只连接了点 $1,2,3$，还形成了环，没有连到点 4，所以不是生成树。

Kruskal 的“从小到大”不是机械地拿前 $n-1$ 条边，而是每条边都要检查是否形成环。

并查集在 Kruskal 中的作用

Kruskal 每次要判断一条边的两个端点是否已经连通。如果已经连通，加入这条边会形成环；如果不连通，就可以加入，并把两个连通块合并。

并查集正好用来维护这个关系。

`find(x)` 返回点 x 所在连通块的代表。若 `find(u) == find(v)`，说明 u 和 v 已经在同一个连通块中。若不同，说明这条边连接了两个不同的连通块，可以选入最小生成树。

在程序阅读中，要特别注意合并发生在什么时候。只有选中一条边时，才执行合并。被跳过的边不能合并，否则会改变后续判断。

例题 3：追踪并查集

边按顺序为 $\{1,2\}$、$\{2,3\}$、$\{1,3\}$、$\{3,4\}$，它们已经按权值升序排列。用 Kruskal 处理时，哪些边会被选中？

初始时，每个点单独一个集合。

处理 $\{1,2\}$，两个端点不在同一集合，选中，并合并 $1,2$。

处理 $\{2,3\}$，点 2 所在集合是 $\{1,2\}$，点 3 单独一个集合，选中，并合并成 $\{1,2,3\}$。

处理 $\{1,3\}$，点 1 和点 3 已经在同一集合中，跳过。

处理 $\{3,4\}$，点 3 在 $\{1,2,3\}$ 中，点 4 单独一个集合，选中。

选中的边为 $\{1,2\}$、$\{2,3\}$、$\{3,4\}$。

Prim 算法

Prim 算法从一个点出发，逐步扩大已经连通的点集。每一步选择一条权值最小的边，要求这条边一端在当前点集中，另一端在当前点集外。选入这条边后，把新的点加入点集。

Kruskal 是“按边从小到大合并连通块”，Prim 是“从一个连通块开始向外扩展”。

例如从点 1 开始，当前集合为 $\{1\}$。在所有从集合内连到集合外的边中，选最小的一条。假设选到 $\{1,2\}$，当前集合变为 $\{1,2\}$。然后继续在跨出当前集合的边中选最小边，直到所有点都加入。

Prim 的每一步都必须选“连接已选点集和未选点集”的边。两端都在已选点集中的边会形成环，不能选；两端都在未选点集中的边暂时不能把新点接入当前树，也不能选。

例题 4：手算 Prim

有 5 个点，边如下：

边	权值
$\{1,2\}$	2
$\{1,3\}$	6
$\{2,3\}$	3
$\{2,4\}$	5
$\{3,4\}$	1
$\{4,5\}$	4
$\{3,5\}$	7

从点 1 开始。

当前集合为 $\{1\}$。跨出去的边有 $\{1,2\}$ 和 $\{1,3\}$，选权值较小的 $\{1,2\}$，总权值为 2。

当前集合为 $\{1,2\}$。跨出去的边有 $\{1,3\}$、$\{2,3\}$、$\{2,4\}$，选 $\{2,3\}$，总权值为 5。

当前集合为 $\{1,2,3\}$。跨出去的边有 $\{2,4\}$、$\{3,4\}$、$\{3,5\}$，选 $\{3,4\}$，总权值为 6。

当前集合为 $\{1,2,3,4\}$。跨出去的边有 $\{4,5\}$、$\{3,5\}$，选 $\{4,5\}$，总权值为 10。

所有点都已加入，最小生成树权值为 10。

Kruskal 和 Prim 的区别

Kruskal 和 Prim 都能求最小生成树，但思路不同。

算法	选边方式	常见数据结构
Kruskal	按边权从小到大尝试选边，不成环才选	排序、并查集
Prim	从一个点集向外扩展，每次选跨出去的最小边	邻接矩阵或优先队列
共同点	都选出 $n-1$ 条边，使总权值最小	都用于无向带权连通图

第一轮中，Kruskal 更容易和并查集程序结合；Prim 更容易出现在矩阵或数组维护最小距离的代码中。

邻接矩阵版 Prim

点数不大时，Prim 常用邻接矩阵实现。数组 `dis[v]` 表示点 v 到当前已选点集的最小边权，`vis[v]` 表示点 v 是否已经加入生成树。

```
int g[N][N];
int dis[N];
bool vis[N];

int prim() {
    memset(dis, 0x3f, sizeof(dis));
    dis[1] = 0;

    int sum = 0;

    for (int i = 1; i <= n; i++) {
        int u = 0;

        for (int j = 1; j <= n; j++) {
            if (!vis[j] && (u == 0 || dis[j] < dis[u])) {
                u = j;
            }
        }

        vis[u] = true;
        sum += dis[u];

        for (int v = 1; v <= n; v++) {
            if (!vis[v] && g[u][v] < dis[v]) {
                dis[v] = g[u][v];
            }
        }
    }

    return sum;
}
```

这段代码每一轮选出一个还没加入生成树、且 `dis` 最小的点。加入点 u 后，用 u 连出去的边更新其他点到当前点集的最小代价。

`dis[1]=0` 表示从点 1 开始，加入第一个点不需要边。最终 `sum` 加上的就是每个点接入生成树时使用的边权。

例题 5：理解 dis

在 Prim 中，`dis[v]` 表示什么？

它不是从起点 1 到 v 的最短路长度，而是点 v 连接到当前已选点集的最小边权。

例如当前已选点集为 $\{1,2,3\}$。点 5 与点 $1,2,3$ 的边权分别为 $10,4,7$，那么 `dis[5]=4`。这表示如果现在把点 5 接入生成树，最便宜的接法是通过点 2，代价为 4。

把 `dis` 误解为最短路距离，是 Prim 程序阅读中很常见的错误。Prim 只看“接入当前树的一条边”，不看从起点走到该点的整条路径。

最小边一定在某棵最小生成树中吗

如果一条边是全图唯一的最小权边，那么它一定出现在所有最小生成树中。

原因可以从 Kruskal 理解：Kruskal 第一条会考虑它。它连接两个不同点，不可能在一开始形成环，所以必然被选入。

如果有多条权值相同的最小边，就不一定每一条都出现在同一棵最小生成树中，但通常每条都可能出现在某棵最小生成树中，具体要看图的结构。

例题 6：最小边是否必选

三角形中三条边权分别为 $1,2,3$。最小生成树一定包含权值为 1 的边吗？

一定包含。

三角形的生成树需要选 2 条边。若不选权值为 1 的边，只能选权值为 2 和 3 的边，总权值为 5。若选权值为 1 和 2 的边，总权值为 3，更小。

从 Kruskal 看，权值为 1 的边最先被考虑，且不会形成环，所以会被选中。

环中最大边

在一个环中，如果某条边的权值严格大于环上其他每一条边，那么它不可能出现在任何最小生成树中。

原因是：如果选了这条最大边，由于环上其他边可以绕过去连接同样的两部分，删掉这条大边并换成环上较小的边，可以得到更小的连通方案。

例如一个三角形三条边权为 $1,2,5$。权值为 5 的边不可能在最小生成树中。选 1 和 2 就能连通三个点，总权值为 3；任何包含 5 的生成树都更大。

例题 7：判断不可能选入的边

四个点构成一个环 $1-2-3-4-1$，四条边权分别为 $2,3,4,10$。边权为 10 的边是否可能出现在最小生成树中？

不可能。

这四条边形成一个环，权值 10 严格大于环上其他边。若某棵生成树包含这条边，可以删掉它，并改用环上其他较小的边来保持连通，总权值会变小。

所以权值为 10 的边不在任何最小生成树中。

权值相同与最小生成树唯一性

如果所有边权互不相同，那么最小生成树唯一。第一轮中常用这个结论做判断。

如果存在相同边权，最小生成树可能不唯一，但也可能仍然唯一。相同边权只是“可能产生多种选择”，不保证一定有多棵最小生成树。

例如一个三角形三条边权都为 1。任意选两条边都能连通三个点，总权值都是 2，所以最小生成树不唯一。

但如果一张图本身就是一棵树，即使所有边权都相同，它也只有一种生成树，因为所有边都必须选。

情况	结论
所有边权互不相同	最小生成树唯一
有相同边权	最小生成树可能唯一，也可能不唯一
原图本身是一棵树	最小生成树就是原图本身

例题 8：相同边权不一定多解

一张图有 4 个点，边为 $\{1,2\},\{2,3\},\{3,4\}$，三条边权都为 5。最小生成树是否唯一？

唯一。

这张图本身就是一棵树。要连通所有 4 个点，三条边都必须选，没有替代边。虽然边权相同，但没有选择空间。

因此“有相同边权”不能直接推出“最小生成树不唯一”。

真题精讲：完全图中的最小生成树

有一道 CSP-S 第一轮真题式题目：

一个包含 8 个顶点的完全图，顶点编号为 1 到 8，任意两点之间的边权等于两顶点编号差的绝对值。例如顶点 3 和 7 之间的边权为 $|7-3|=4$。该图的最小生成树总权重是多少？

这张图是完全图，任意两个点之间都有边。边权为编号差的绝对值，所以相邻编号之间的边权都是 1，例如 $\{1,2\}$、$\{2,3\}$、$\{3,4\}$ 等。

要连接 8 个点，生成树必须选 7 条边。每条边权至少为 1，所以总权值至少为 7。

现在选边 $\{1,2\}$、$\{2,3\}$、$\{3,4\}$、$\{4,5\}$、$\{5,6\}$、$\{6,7\}$、$\{7,8\}$，这 7 条边把所有点连成一条链，总权值正好是 7。

下界能达到，所以最小生成树总权重为 7。

这道题不用真的把完全图所有边都列出来。抓住两个信息就够了：生成树需要 $n-1$ 条边；每条边权至少为 1，且相邻编号边可以达到这个最小权值。

变式：编号为 1 到 n

若顶点编号为 1 到 n，任意两点边权为编号差的绝对值，这个完全图的最小生成树权值是多少？

生成树需要 $n-1$ 条边，每条边权至少为 1。选择所有相邻编号边 $\{1,2\},\{2,3\},\ldots,\{n-1,n\}$，可以用 $n-1$ 条权值为 1 的边连通所有点。

所以最小生成树权值为 $n-1$。

程序阅读中的 Kruskal

Kruskal 程序阅读题通常围绕排序、并查集和选边计数展开。

看下面这段代码：

```
sort(e + 1, e + m + 1, cmp);

for (int i = 1; i <= n; i++) {
    fa[i] = i;
}

int ans = 0, cnt = 0;
```

```
for (int i = 1; i <= m; i++) {
    int x = find(e[i].u);
    int y = find(e[i].v);

    if (x != y) {
        fa[x] = y;
        ans += e[i].w;
        cnt++;
    }
}

cout << ans << endl;
```

这段代码没有在 `cnt == n - 1` 时提前停止，但如果图是连通的，它仍然会选出 $n-1$ 条边。后面的边因为端点已经连通，会被跳过。

如果图不连通，`cnt` 最后会小于 $n-1$。这时 `ans` 只是最小生成森林的权值，不是整张图的最小生成树权值。程序若没有判断 `cnt == n-1`，就可能在不连通图上输出一个不符合题意的结果。

例题 9：不连通图会怎样

有 4 个点，边为 $\{1,2\}$ 权值 1，$\{3,4\}$ 权值 2。运行 Kruskal 后 `cnt` 是多少？

初始有 4 个连通块。

选 $\{1,2\}$ 后，合并成一个连通块，`cnt=1`。

选 $\{3,4\}$ 后，合并成另一个连通块，`cnt=2`。

所有边处理完，`cnt=2`。但 $n-1=3$，没有选够生成树所需的边数。原图不连通，不存在生成树。

因此完整程序应检查 `cnt == n - 1`。若不满足，说明无法构造最小生成树。

程序阅读中的 Prim

Prim 程序阅读题中，最容易误解的是 `dis` 数组。

常见代码如下：

```
for (int i = 1; i <= n; i++) {
    int u = 0;

    for (int j = 1; j <= n; j++) {
        if (!vis[j] && (u == 0 || dis[j] < dis[u])) {
            u = j;
        }
    }

    vis[u] = true;
    ans += dis[u];

    for (int v = 1; v <= n; v++) {
        if (!vis[v] && g[u][v] < dis[v]) {
            dis[v] = g[u][v];
        }
    }
}
```

外层循环每次加入一个新点。内层第一段找当前还没加入生成树、且接入代价最小的点。内层第二段用新加入的点更新其他点的接入代价。

如果图不连通，某一轮可能找不到合适的点，或者某些点的 `dis` 一直是很大的初值。严谨写法要检查这种情况。

例题 10：Prim 与最短路的差别

在 Prim 中，若 `dis[5]=3`，能不能说明从起点到点 5 的最短路长度是 3？

不能。

`dis[5]=3` 只能说明点 5 当前可以用一条权值为 3 的边接入已选点集。它不表示从起点沿树或沿原图走到点 5 的最短距离。

最短路中的 `dis` 是从起点到某点的路径总代价；Prim 中的 `dis` 是某点连到当前生成树的最小单边代价。名字相同，含义不同。

负权边与最小生成树

最小生成树允许边权为负数。Kruskal 仍然按权值从小到大排序，负权边会更早被考虑；只要不形成环，就可以选入。

这和某些最短路算法不同。最短路中负权边可能影响算法选择；最小生成树只需要比较边权、避免成环，负权边本身不会破坏 Kruskal 或 Prim 的基本思路。

例题 11：负权边

有 3 个点，边为 $\{1,2\}$ 权值 -2，$\{2,3\}$ 权值 3，$\{1,3\}$ 权值 4。最小生成树权值是多少？

生成树需要选 2 条边。按 Kruskal 顺序，先选 $\{1,2\}$，权值 -2；再选 $\{2,3\}$，权值 3。此时三个点已经连通。

总权值为 $-2+3=1$。

负权边可以出现在最小生成树中，不需要特殊排除。

完善程序中的常见空缺

最小生成树的完善程序题，通常不会让学生重新写完整算法，而是在关键位置留空。

Kruskal 常见空缺如下：

空缺位置	应关注的问题
排序比较函数	是否按边权从小到大排序
初始化并查集	是否令 `fa[i] = i`
判断能否选边	是否判断 `find(u) != find(v)`
合并集合	是否只在选边后合并
累加答案	是否加上当前边权
停止条件	是否选够 `n - 1` 条边

例如：

```
if (find(e[i].u) != find(e[i].v)) {
    ________;
    ans += e[i].w;
    cnt++;
}
```

空缺处应是合并两个集合的语句，例如 `fa[find(e[i].u)] = find(e[i].v)`。实际写法可能先保存根节点，避免重复调用。

Prim 常见空缺如下：

空缺位置	应关注的问题
初始化 `dis`	除起点外设为很大，起点设为 0
选择点 `u`	找未访问且 `dis` 最小的点
加入生成树	标记 `vis[u] = true`
累加答案	加上 `dis[u]`
更新其他点	用 `g[u][v]` 更新 `dis[v]`

例如：

```
if (!vis[v] && g[u][v] < dis[v]) {
    ________;
}
```

空缺处应填 `dis[v] = g[u][v]`。这里更新的是点 v 接入当前生成树的最小边权，不是 `dis[u] + g[u][v]`。如果写成 `dis[u] + g[u][v]`，那就变成了类似最短路的转移，含义错误。

常见错误

第一，认为最小生成树是最短路树。最小生成树保证总边权最小，不保证从某个起点到每个点的路径最短。

第二，认为选最小的 $n-1$ 条边就是答案。若这些边形成环或没有连通所有点，就不是生成树。

第三，忘记连通性。原图不连通时没有生成树，只能得到最小生成森林。程序中应检查是否选够 $n-1$ 条边。

第四，把 Prim 的 `dis[v]` 当成从起点到 v 的最短距离。Prim 的 `dis[v]` 是连接当前点集和点 v 的最小单边权。

第五，认为边权相同一定导致最小生成树不唯一。相同边权只是可能导致多种选择，是否不唯一还要看图的结构。

第六，认为负权边不能用于最小生成树。最小生成树可以有负权边，按权值排序和避免成环即可。

小结

最小生成树研究的是带权无向连通图中，如何选出一棵覆盖所有点、总边权最小的树。含 n 个点的生成树一定有 $n-1$ 条边。

Kruskal 按边权从小到大尝试选边，用并查集判断是否会形成环。若两个端点已经连通，就跳过；若不连通，就选入并合并。

Prim 从一个点出发，每次选择一条从已选点集连到未选点集的最小边，把新的点接入生成树。Prim 中的 `dis` 表示接入当前生成树的最小单边代价，不是最短路距离。

本章需要重点记住：

- 最小生成树只用于无向带权连通图；
- 生成树必须连通、无环，并且有 $n-1$ 条边；
- Kruskal 是按边权排序后避免成环；
- Prim 是从一个点集向外选最小跨边；
- 最小生成树不等于最短路；
- 不连通图没有生成树；
- 边权可以相同，也可以为负。

下一章进入最短路。那一章会研究从一个起点到其他点的最小路径代价，并重点区分 Dijkstra、Floyd 等常见程序结构。

0x1E 最短路

第 29 章讲的是最小生成树。最小生成树要在无向带权图中选出若干条边，使所有点连通且总边权最小。本章讲最短路，它关心的是从一个点到另一个点的路径代价最小。

这两个问题名字都带 “最小”，但目标不同。最小生成树选的是一棵覆盖所有点的树；最短路选的是一条从起点到终点的路径。最小生成树上的路径不一定是原图中的最短路。

CSP-S 第一轮中的最短路，常见考法不是让学生手写很长的模板，而是读懂程序在维护什么：`dis` 是什么含义，什么时候更新，队列或优先队列里放的是什么，Floyd 的三层循环为什么要按中转点在最外层，Dijkstra 为什么不能直接处理负权边。

路径与距离

在带权图中，一条路径的长度通常等于路径上所有边权之和。

例如有边 $(1,2,3)$、$(2,4,5)$、$(1,3,2)$、$(3,4,10)$。路径 $1\to2\to4$ 的长度为 $3+5=8$，路径 $1\to3\to4$ 的长度为 $2+10=12$。从 1 到 4 的最短路长度就是 8。

最短路不一定边数最少。边数少的路径，边权可能很大；边数多的路径，边权可能更小。

例如 $1\to3$ 的直接边权为 100，而 $1\to2\to3$ 两条边权分别为 1 和 1。从 1 到 3 的最短路不是直接边，而是经过点 2，长度为 2。

说法	是否一定正确
边数最少的路径一定最短	不一定
权值和最小的路径是最短路	正确
最短路一定唯一	不一定
有向图中从 s 到 t 最短，不代表从 t 到 s 也相同	正确

例题 1：手算最短路

有向边如下：

边	权值
$(1,2)$	4
$(1,3)$	2
$(3,2)$	1
$(2,4)$	3
$(3,4)$	8

求从 1 到 4 的最短路长度。

可以列出几条候选路径：

- $1\to2\to4$，长度为 $4+3=7$；
- $1\to3\to4$，长度为 $2+8=10$；
- $1\to3\to2\to4$，长度为 $2+1+3=6$。

所以从 1 到 4 的最短路长度为 6，对应路径为 $1 \to 3 \to 2 \to 4$。

注意它不是边数最少的路径。$1 \to 2 \to 4$ 只经过两条边，但长度为 7，比 $1 \to 3 \to 2 \to 4$ 更大。

单源、单点对、全源

最短路问题有几种常见问法。

单点对最短路：只问从 s 到 t 的最短距离。

单源最短路：给定一个起点 s，求 s 到所有点的最短距离。

全源最短路：求任意两个点之间的最短距离。

问法	目标
单点对最短路	求 s 到 t
单源最短路	求 s 到所有点
全源最短路	求所有 u 到所有 v

Dijkstra 和 Bellman-Ford 常用于单源最短路。Floyd 常用于全源最短路。无权图最短路可以用 BFS。

第一轮题中，看到 `dis[i]` 一维数组，常常是单源最短路；看到 `d[i][j]` 二维数组，常常是全源最短路。

无权图最短路：BFS

如果图中每条边的权值都相同，通常可以把每条边的长度看作 1。此时从起点到某个点的最短路，就是经过边数最少的路径。

这种问题可以用 BFS。BFS 从起点开始，一层一层向外扩展：先访问距离为 1 的点，再访问距离为 2 的点。第一次访问到某个点时，得到的就是最短距离。

```
vector<int> e[N];
int dis[N];

void bfs(int s) {
    memset(dis, -1, sizeof(dis));

    queue<int> q;
    dis[s] = 0;
    q.push(s);

    while (!q.empty()) {
        int u = q.front();
        q.pop();

        for (int v : e[u]) {
            if (dis[v] == -1) {
                dis[v] = dis[u] + 1;
                q.push(v);
            }
        }
    }
}
```

这里 `dis[v] == -1` 表示点 v 还没有被访问过。第一次给 `dis[v]` 赋值时，就是从起点到 v 的最短边数。

例题 2：BFS 距离

无向图边为 $\{1,2\},\{1,3\},\{2,4\},\{3,4\},\{4,5\}$。从点 1 出发，求各点最短距离。

从点 1 出发，`dis[1]=0`。

点 1 的邻点是 $2,3$，所以 `dis[2]=1`，`dis[3]=1`。

接着从点 2 和点 3 扩展，都可以到点 4，第一次访问点 4 时得到 `dis[4]=2`。

从点 4 扩展到点 5，得到 `dis[5]=3`。

所以距离为：点 1 为 0，点 $2,3$ 为 1，点 4 为 2，点 5 为 3。

BFS 的关键是“第一次访问即最短”。这是因为所有边长度相同，每多走一层，路径长度就增加 1。

带权图不能直接用 BFS

如果边权不全相同，BFS 不再适合直接求最短路。

例如从 1 到 3 有两条路径：直接边 $1 \to 3$ 权值为 100；另一条路径 $1 \to 2 \to 3$ 两条边权分别为 1 和 1。BFS 会认为直接到 3 只用一条边，比经过点 2 的两条边更近。但按权值和计算，$1 \to 2 \to 3$ 长度为 2，才是最短路。

所以 BFS 求的是“边数最少”，不是一般意义下的“权值和最小”。只有边权都相同，或者题目可以把每条边看成长度 1 时，BFS 才能直接用来求最短路。

松弛

最短路算法里经常出现一个操作：松弛。

若已经知道从起点到 u 的当前距离是 `dis[u]`，并且存在一条边 $u \to v$，权值为 w，那么从起点到 v 可以尝试走到 u，再经过这条边到 v。候选距离为 `dis[u] + w`。

如果 `dis[u] + w < dis[v]`，就更新：

```
dis[v] = dis[u] + w;
```

这就是松弛。

松弛不是最终确认答案，而是用一条边尝试改进某个点的当前距离。Dijkstra、Bellman-Ford、SPFA 等算法都有松弛操作。

例题 3：一次松弛

当前 `dis[2]=5`，`dis[4]=12`，有边 $2 \to 4$，边权为 3。经过这条边能否更新 `dis[4]`？

从起点到点 2 的距离为 5，再走边 $2 \to 4$，候选距离为 $5+3=8$。因为 $8<12$，所以可以更新 `dis[4]=8`。

若边权为 10，候选距离为 $5+10=15$，大于原来的 12，就不更新。

Dijkstra 算法

Dijkstra 算法用于边权非负的单源最短路。它维护一个 `dis` 数组，`dis[v]` 表示当前已知的从起点到 v 的最短候选距离。

算法每次选出一个尚未确定、且 `dis` 最小的点 u。在边权非负的条件下，这个点的最短距离已经不会再被其他路径改小，于是把它标记为确定。然后用 u 的所有出边去松弛其他点。

基本思想如下：

1. 起点距离为 0，其他点距离为无穷大；
2. 每次选出未确定点中 `dis` 最小的点；

3. 用这个点的出边进行松弛；
4. 重复直到所有可达点都被确定。

例题 4：手算 Dijkstra

有向边如下：

边	权值
$(1,2)$	4
$(1,3)$	2
$(3,2)$	1
$(2,4)$	3
$(3,4)$	8

从点 1 出发。

初始 `dis[1]=0`，其余为无穷大。确定点 1，用它的出边更新：`dis[2]=4`，`dis[3]=2`。

未确定点中，`dis` 最小的是点 3，确定点 3。用边 $(3,2)$ 更新点 2，得到候选距离 $2+1=3$，比原来的 4 小，所以 `dis[2]=3`。用边 $(3,4)$ 更新点 4，得到 `dis[4]=10`。

未确定点中，`dis` 最小的是点 2，确定点 2。用边 $(2,4)$ 更新点 4，得到候选距离 $3+3=6$，比原来的 10 小，所以 `dis[4]=6`。

最后确定点 4。从 1 到 4 的最短距离为 6。

这个过程和前面的手算结果一致。

朴素 Dijkstra

点数较小或用邻接矩阵存图时，常见朴素 Dijkstra 写法如下：

```
int g[N][N];
int dis[N];
bool vis[N];

void dijkstra(int s) {
    memset(dis, 0x3f, sizeof(dis));
    dis[s] = 0;

    for (int i = 1; i <= n; i++) {
        int u = 0;

        for (int j = 1; j <= n; j++) {
            if (!vis[j] && (u == 0 || dis[j] < dis[u])) {
                u = j;
            }
        }

        vis[u] = true;

        for (int v = 1; v <= n; v++) {
            if (g[u][v] < INF && dis[u] + g[u][v] < dis[v]) {
                dis[v] = dis[u] + g[u][v];
            }
        }
    }
}
```

这段代码每轮用一层循环找 `dis` 最小的未确定点，所以时间复杂度为 $O(n^2)$。它适合点数不大、边比较密的图。

读这段程序时，要看清 `vis[u]` 的含义：它表示点 u 的最短距离已经确定。被标记后，后面不会再用它作为候选点。

例题 5：`vis` 的作用

在 Dijkstra 中，为什么已经 `vis[u]=true` 的点不再被选？

因为在边权非负的条件下，每次选出的都是未确定点中 `dis` 最小的点。其他未确定点的距离不小于它，再加上非负边权，不可能绕一圈回来把它变得更小。

所以一旦点 u 被选中，它的最短距离就已经确定，可以放心用它去更新别人。

如果存在负权边，这个理由就可能失效。某条负边可能从较远的点绕回来，把已经确定的点距离改小。

堆优化 Dijkstra

当边数远少于 n^2 时，常用邻接表和优先队列写 Dijkstra。优先队列中存的是候选距离和点编号。

```
struct Edge {
    int to, w;
};

vector<Edge> e[N];
int dis[N];
bool vis[N];

void dijkstra(int s) {
    memset(dis, 0x3f, sizeof(dis));

    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> q;

    dis[s] = 0;
    q.push({0, s});

    while (!q.empty()) {
        int u = q.top().second;
        q.pop();

        if (vis[u]) {
            continue;
        }
        vis[u] = true;

        for (auto ed : e[u]) {
            int v = ed.to;
            int w = ed.w;

            if (dis[u] + w < dis[v]) {
                dis[v] = dis[u] + w;
                q.push({dis[v], v});
            }
        }
    }
}
```

这里可能会把同一个点多次加入优先队列。因为某个点的距离被更新后，会把新距离再次压入队列。旧距离对应的记录还留在队列中。用 `vis[u]` 可以保证每个点只在第一次弹出时被正式处理。

也可以不用 `vis`，改用下面的判断跳过旧记录：

```
int d = q.top().first;
int u = q.top().second;
q.pop();

if (d != dis[u]) {
    continue;
}
```

这表示如果弹出的距离已经不是当前最优距离，就跳过它。

例题 6：为什么会有旧记录

假设起点是 1。先通过边 $1 \to 2$ 得到 `dis[2]=10`，于是把 $(10, 2)$ 放入优先队列。后来又通过路径 $1 \to 3 \to 2$ 得到 `dis[2]=4`，于是又把 $(4, 2)$ 放入队列。

此时队列里同时有 $(10, 2)$ 和 $(4, 2)$。点 2 应该按距离 4 处理。等 $(10, 2)$ 被弹出时，它已经是旧记录，不能再用它松弛其他点。

所以堆优化 Dijkstra 中看到“同一个点多次入队”并不奇怪，关键是弹出时要判断是否还有效。

Dijkstra 不能直接处理负权边

Dijkstra 的核心前提是边权非负。如果有负权边，一个已经确定的点，后面仍可能通过负权边被更新得更小。

例如有边 $(1, 2)$ 权值 2，$(1, 3)$ 权值 5，$(3, 2)$ 权值 -10。从点 1 出发，Dijkstra 会先认为点 2 的距离 2 比点 3 的距离 5 小，于是确定点 2。但真正最短路是 $1 \to 3 \to 2$，长度为 $5 - 10 = -5$。

点 2 被过早确定，答案就错了。

因此，看到题目中有负权边时，不能直接使用普通 Dijkstra。第一轮题中，这经常作为判断题或选项陷阱出现。

Bellman-Ford 的基本思想

Bellman-Ford 可以处理负权边。它的做法是反复扫描所有边，进行松弛。

若图中没有从起点可达的负环，一条最短路最多经过 $n - 1$ 条边。因此扫描所有边 $n - 1$ 轮后，所有最短距离都能被更新出来。

伪代码可以写成：

```
memset(dis, 0x3f, sizeof(dis));
dis[s] = 0;

for (int i = 1; i <= n - 1; i++) {
    for (int j = 1; j <= m; j++) {
        int u = edge[j].u;
        int v = edge[j].v;
        int w = edge[j].w;

        if (dis[u] + w < dis[v]) {
            dis[v] = dis[u] + w;
        }
    }
}
```

如果第 n 轮仍然可以松弛，说明存在从起点可达的负环。因为不含环的路径最多只有 $n - 1$ 条边，继续变小只能来自某个总权值为负的环。

CSP-S 第一轮通常不会深入 Bellman-Ford 的证明，但会考“负权边能不能用 Dijkstra”“为什么扫描 $n-1$ 轮”“负环会导致什么”。

例题 7：负环

若从起点可以到达一个环，环上边权和为 -3，那么最短路会发生什么？

可以绕这个环一圈，路径长度减少 3；绕两圈，减少 6；绕更多圈，长度继续下降。此时不存在一个有限的最短距离。

所以有从起点可达的负环时，普通最短路问题没有正常答案。Bellman-Ford 可以用额外一轮松弛来检测这种情况。

Floyd 算法

Floyd 算法用于求所有点对之间的最短路。它使用二维数组 `d[i][j]`，表示从 i 到 j 的当前最短距离。

核心转移是：尝试让路径经过中转点 k。如果从 i 到 k，再从 k 到 j 比当前 i 到 j 更短，就更新：

```
d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
```

完整代码常写为：

```
for (int k = 1; k <= n; k++) {
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            if (d[i][k] + d[k][j] < d[i][j]) {
                d[i][j] = d[i][k] + d[k][j];
            }
        }
    }
}
```

最外层必须是 `k`。它表示当前允许使用编号不超过 k 的点作为中转点。若把三层循环随意交换，含义会被破坏，结果可能出错。

Floyd 的时间复杂度为 $O(n^3)$，适合点数较小、需要查询任意两点最短路的情况。

例题 8：一次 Floyd 更新

当前有 `d[1][2]=3`，`d[2][4]=5`，`d[1][4]=12`。当中转点 $k=2$ 时，`d[1][4]` 会不会更新？

经过点 2 的路径长度为 $3+5=8$。因为 $8<12$，所以更新 `d[1][4]=8`。

这表示从 1 到 4 的当前最短路可以通过 $1\to 2\to 4$ 改进。

Floyd 初始化

Floyd 的初始化很重要。

通常要先把所有 `d[i][j]` 设为无穷大，再令 `d[i][i]=0`。读入边 (u,v,w) 时，更新 `d[u][v]=min(d[u][v],w)`。如果是无向图，还要更新 `d[v][u]`。

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        d[i][j] = INF;
    }
```

```
    d[i][i] = 0;
}

for (int i = 1; i <= m; i++) {
    int u, v, w;
    cin >> u >> v >> w;
    d[u][v] = min(d[u][v], w);
    d[v][u] = min(d[v][u], w);
}
```

若题目是有向图，只能写 `d[u][v] = min(d[u][v], w)`，不能顺手加反向边。

若有重边，要取较小的边权。因为两点之间若有多条直接边，最短路只会使用其中权值最小的直接边。

例题 9：重边初始化

有向图中有两条边 $(1,2)$，权值分别为 7 和 3。初始化 Floyd 时，`d[1][2]` 应为多少？

应为 3。

因为从 1 到 2 的直接边有两条，较短的一条权值为 3。若直接赋值而不取最小值，读入顺序可能影响结果。

Floyd 与传递闭包

如果图没有边权，只想判断点与点之间是否可达，也可以用类似 Floyd 的思想做传递闭包。

设 `can[i][j]` 表示 i 是否能到 j。若 i 能到 k，且 k 能到 j，那么 i 能到 j。

```
for (int k = 1; k <= n; k++) {
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            if (can[i][k] && can[k][j]) {
                can[i][j] = true;
            }
        }
    }
}
```

这类代码看起来像 Floyd，但数组类型可能是 `bool`，含义不是距离，而是可达性。

程序阅读题中，要通过数组含义判断它在做最短路还是可达性传递。

最短路条数

有些题不仅求最短距离，还求最短路条数。第一轮中可能以程序阅读形式出现。

以 Dijkstra 为例，设 `cnt[v]` 表示从起点到 v 的最短路条数。当发现更短路径时，距离更新，条数也要替换；当发现同样长度的最短路径时，条数要累加。

```
if (dis[u] + w < dis[v]) {
    dis[v] = dis[u] + w;
    cnt[v] = cnt[u];
} else if (dis[u] + w == dis[v]) {
    cnt[v] += cnt[u];
}
```

这两种情况不能混淆。

如果找到更短路径，原来的最短路已经不再最短，所以 `cnt[v]` 要改成 `cnt[u]`。如果找到同样短的路径，才是在原有条数上增加。

例题 10：条数更新

从起点到 u 的最短路条数为 3，`dis[u]=5`。有边 $u \to v$，权值为 2。

若当前 `dis[v]=10`，则候选距离为 7，更短，所以更新 `dis[v]=7`，并令 `cnt[v]=3`。

若当前 `dis[v]=7`，则候选距离等于当前最短距离，说明发现了新的最短路，应令 `cnt[v] += 3`。

若当前 `dis[v]=6`，候选距离 7 更大，不更新距离，也不更新条数。

路径还原

最短路程序有时还要输出具体路径。这时会用 `pre[v]` 记录点 v 在最短路上的前驱点。

当用 u 更新 v 时，说明当前最优路径最后一步是 $u \to v$，于是写：

```
pre[v] = u;
```

最后从终点 t 开始，不断找前驱：$t, pre[t], pre[pre[t]], \ldots$，直到起点。得到的是反向路径，再倒过来输出。

例题 11：还原路径

若 `pre[5]=3`，`pre[3]=2`，`pre[2]=1`，起点为 1，终点为 5，最短路径是什么？

从终点倒着找：$5 \leftarrow 3 \leftarrow 2 \leftarrow 1$。反过来就是 $1 \to 2 \to 3 \to 5$。

所以路径为 $1, 2, 3, 5$。

若某个点没有前驱，且它不是起点，通常说明它从起点不可达，或者程序没有记录路径。

不可达

如果从起点到某个点没有路径，最短距离通常保持为无穷大。程序里会用一个很大的数表示无穷大，例如 `0x3f3f3f3f` 或 `1e18`。

输出时要根据题目要求处理不可达。有的题输出 `-1`，有的题输出某个固定字符串，有的题保证一定可达。

程序阅读题中，如果某点不可达，`dis[v]` 不会被更新。若程序没有判断不可达就直接输出 `dis[v]`，可能输出一个很大的数。

例题 12：不可达点

有向图边为 $(1,2), (2,3)$，从点 1 出发。点 4 没有任何入边或出边。执行最短路后，`dis[4]` 会怎样？

点 4 从起点不可达，所以 `dis[4]` 保持初始的无穷大。若题目要求输出不可达，应根据这个值判断并输出对应结果。

不能把不可达理解成距离为 0。距离为 0 只属于起点到自己，或者题目中特殊的零权路径。

无向图与有向图

最短路中的有向图和无向图差别很大。

如果是无向边 $\{u, v\}$，读入时通常要加两个方向：

```
add(u, v, w);
add(v, u, w);
```

如果是有向边 (u, v)，只能加一个方向：

```
add(u, v, w);
```

如果把有向图误存成无向图，很多原本不能走的反向路径会被加入，最短路可能变小。程序阅读题中，读边方式是判断图类型的重要线索。

例题 13：方向影响最短路

有向边为 $(1, 2, 1), (2, 3, 1)$。从 3 到 1 是否可达？

不可达。虽然从 1 可以到 3，但没有反向边。

若把这两条边误当成无向边，就会认为 $3 \to 2 \to 1$ 可达，距离为 2。这与原题有向图不符。

最短路与最小生成树再比较

最小生成树和最短路都出现在带权图中，但它们的 `dis`、选边和目标都不同。

项目	最小生成树	最短路
目标	连接所有点，总边权最小	从起点到目标点路径最短
常见算法	Kruskal、Prim	BFS、Dijkstra、Floyd、Bellman-Ford
是否一定保留原图最短路径	不一定	本身就是求最短路径
`dis` 含义	Prim 中表示接入生成树的最小边权	表示从起点到某点的路径长度
图类型	通常是无向带权图	有向、无向都可以

例题 14：区分目标

有一张图，要求“建设若干道路，使所有城市连通，且建设费用总和最小”。这是最短路问题还是最小生成树问题？

这是最小生成树问题。目标是选出若干边连接所有点，并使总费用最小。

若题目改成“从城市 1 到城市 n 的最小通行费用”，才是最短路问题。

读题时先看目标：是连接所有点，还是从一个起点到某个终点。

程序阅读中的常见陷阱

第一，误把 BFS 用在一般带权图上。BFS 只能直接处理边权相同的最短路；一般带权图要使用其他算法。

第二，误把 Dijkstra 用在负权图上。Dijkstra 的正确性依赖边权非负。

第三，混淆 Prim 和 Dijkstra。两者都可能有 `dis` 和 `vis`，但 Prim 的 `dis` 是接入生成树的最小边权，Dijkstra 的 `dis` 是从起点出发的路径长度。

第四，Floyd 三层循环顺序写错。最外层应是中转点 `k`，表示逐步放开可用中转点集合。

第五，无向图忘记加反向边。有向图误加反向边。这会直接改变可达关系和最短距离。

第六，更新最短路条数时，把“更短路径”和“同长路径”混在一起。更短时替换条数，同长时累加条数。

第七，没有处理不可达点。`dis` 保持无穷大时，不应当被当作普通距离理解。

完善程序中的常见空缺

最短路完善程序常围绕初始化、松弛、选点和判重展开。

Dijkstra 常见空缺如下：

空缺位置	应关注的问题
初始化	`dis[s] = 0`，其他为无穷大
选点	找未确定且 `dis` 最小的点
松弛条件	`dis[u] + w < dis[v]`
松弛操作	`dis[v] = dis[u] + w`
堆中元素	通常是 `{dis[v], v}`
跳过旧记录	判断弹出距离是否等于当前 `dis[u]`

Floyd 常见空缺如下：

空缺位置	应关注的问题
初始化	`d[i][i] = 0`，其他为无穷大
读入无向边	两个方向都要更新
读入重边	取较小权值
三层循环	最外层是 `k`
转移	`d[i][j] = min(d[i][j], d[i][k] + d[k][j])`

Bellman-Ford 常见空缺如下：

空缺位置	应关注的问题
循环轮数	通常扫描 $n-1$ 轮
扫描对象	每轮扫描所有边
松弛条件	`dis[u] + w < dis[v]`
负环判断	第 n 轮仍能松弛

例题 15：补全 Dijkstra 松弛

下面代码来自 Dijkstra：

```
for (auto ed : e[u]) {
    int v = ed.to;
    int w = ed.w;

    if (__________) {
        dis[v] = dis[u] + w;
        q.push({dis[v], v});
    }
}
```

空缺处应填 `dis[u] + w < dis[v]`。

这里是在判断经过点 u 再走边 $u \to v$，能否让从起点到 v 的距离更短。只有更短时才更新 `dis[v]`，并把新的候选距离放入优先队列。

例题 16：补全 Floyd 转移

下面代码来自 Floyd：

```
for (int k = 1; k <= n; k++) {
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            d[i][j] = min(__________);
        }
    }
}
```

空缺处应填 `d[i][j], d[i][k] + d[k][j]`。

完整语句为：

```
d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
```

含义是：从 i 到 j 的最短路，要么不经过当前中转点 k，保持原值；要么经过 k，长度为 `d[i][k] + d[k][j]`。

小结

最短路研究的是路径代价最小。无权图或等权图可以用 BFS；非负权单源最短路常用 Dijkstra；全源最短路常用 Floyd；有负权边时要考虑 Bellman-Ford 或其他适合负权的算法。

本章需要重点记住：

- 最短路的目标是路径权值和最小，不是边数最少；
- BFS 只能直接处理等权图的最短边数；
- 松弛是用一条边尝试改进距离；
- Dijkstra 要求边权非负；
- 堆优化 Dijkstra 中，同一个点可能多次入队，需要跳过旧记录；
- Floyd 的外层循环应是中转点 `k`；
- 不可达点的距离会保持为无穷大；
- 最短路和最小生成树目标不同，不能互相替代。

下一章进入状态最短路与次短路。那一章会在普通最短路基础上，处理 “多一维状态”“允许特殊操作”“严格大于最短路的第二短路径” 等第一轮完善程序中常见的模型。

0x1F 状态最短路与次短路

第 30 章讲了普通最短路：给定图、起点和边权，求从起点到某些点的最小路径代价。本章继续讲两类第一轮中常见的扩展：状态最短路和次短路。

状态最短路解决的是“只知道当前在哪个点还不够”的问题。例如一条路径中最多允许使用一次免费边，程序不仅要知道当前到达了哪个点，还要知道免费机会是否已经用过。次短路解决的是“除了最短，还要找严格大于最短的第二短路径”的问题。它不能只维护一个最短距离，而要同时维护最短和次短。

为什么需要状态

普通最短路中，`dis[u]` 表示从起点到点 u 的最短距离。这个表示默认了一件事：到达同一个点之后，未来的选择只由当前点决定，和之前怎么走来的无关。

但有些题不满足这个条件。

例如从 s 到 t 的路径中，最多可以选择一条边免费。到达点 u 时，如果免费机会还没用，后面仍可以免费走一条边；如果免费机会已经用掉，后面所有边都要正常付费。虽然当前都在点 u，但后续能力不同。

这时只写 `dis[u]` 不够。需要写成 `dis[u][0]` 和 `dis[u][1]`：

- `dis[u][0]` 表示到达点 u，且免费机会还没用的最小代价；
- `dis[u][1]` 表示到达点 u，且免费机会已经用过的最小代价。

这样，原图中的一个点被拆成了两个状态。

状态	含义
$(u,0)$	到达点 u，特殊机会未使用
$(u,1)$	到达点 u，特殊机会已使用

状态最短路的核心就是：把“点”扩展成“点 + 状态”，再在状态之间连边，最后跑普通最短路。

状态图

把状态看成新的点，原问题就变成了一张更大的图。原图有 n 个点，如果每个点有 2 种状态，状态图就有 $2n$ 个点。

仍以“最多一条边免费”为例。原图中有一条边 $u \to v$，边权为 w。

如果免费机会还没用，可以正常走这条边，状态从 $(u,0)$ 到 $(v,0)$，代价为 w；也可以把这条边作为免费边，状态从 $(u,0)$ 到 $(v,1)$，代价为 0。

如果免费机会已经用过，只能正常走，状态从 $(u,1)$ 到 $(v,1)$，代价为 w。

于是每条原图边会产生三类状态转移：

当前状态	操作	新状态	代价
$(u,0)$	不免费走边	$(v,0)$	w
$(u,0)$	免费走边	$(v,1)$	0
$(u,1)$	正常走边	$(v,1)$	w

不需要真的把状态图全部建出来。程序中常在枚举原图边时，直接写出这些转移。

例题 1：免费边状态转移

原图有边 $2 \to 5$，边权为 7。当前已知 `d[2][0]=10`，`d[2][1]=6`。这条边可以怎样更新点 5？

从 `d[2][0]` 出发，免费机会还没用。正常走边可以尝试更新 `d[5][0]`，候选值为 $10+7=17$；免费走这条边可以尝试更新 `d[5][1]`，候选值为 10。

从 `d[2][1]` 出发，免费机会已经用过，只能正常走边，可以尝试更新 `d[5][1]`，候选值为 $6+7=13$。

所以这条边最多产生三次松弛。

真题式完善程序：免费边最短路

近年 CSP-S 第一轮完善程序中出现过类似“特殊最短路”的框架：给定一张非负边权图，从 s 到 t 的路径中最多允许一条边免费，求最小费用。

这类题的关键不是重新发明算法，而是把题意对应到状态：

- `used = 0` 表示还没有使用免费边；
- `used = 1` 表示已经使用过免费边；
- `d[u][used]` 表示到达点 u 且处于状态 `used` 的最小代价。

初始化时，从起点出发，还没有使用免费边，所以 `d[s][0] = 0`，并把状态 `(s,0)` 放入优先队列。不能把 `d[s][1]` 也直接设成 0，因为“已经使用过免费机会但还在起点”并不是从题意中直接得到的初始状态。

枚举边 $u \to v$、边权 w 时，有两种更新。

第一种是正常走边：

```
if (d[u][used] + w < d[v][used]) {
    d[v][used] = d[u][used] + w;
    q.push({d[v][used], v, used});
}
```

第二种是如果还没有使用免费边，就免费走当前边：

```
if (used == 0 && d[u][0] < d[v][1]) {
    d[v][1] = d[u][0];
    q.push({d[v][1], v, 1});
}
```

最后答案是 `min(d[t][0], d[t][1])`。因为题目说“至多选择一条边免费”，可以不用免费机会，也可以用掉免费机会。如果题目改成“必须使用一次免费机会”，答案才是 `d[t][1]`。

例题 2：手算免费边

有向边如下：

边	权值
$(1,2)$	5
$(2,4)$	5
$(1,3)$	8
$(3,4)$	1

从 1 到 4，最多一条边免费。最小费用是多少？

如果不使用免费边，路径 $1 \to 3 \to 4$ 的费用为 $8+1=9$，路径 $1 \to 2 \to 4$ 的费用为 10。

如果走 $1 \to 3 \to 4$，把边 $(1,3)$ 免费，费用为 $0+1=1$。如果把边 $(3,4)$ 免费，费用为 $8+0=8$。

如果走 $1 \to 2 \to 4$，免费其中一条权值为 5 的边，费用为 5。

所以最小费用为 1。

状态写法中，这个答案对应 `d[4][1]=1`。同时 `d[4][0]=9`，表示完全不用免费机会的最短费用。最终取两者较小，得到 1。

状态必须够用

状态设计最重要的是：它必须包含所有会影响未来选择的信息。

例如“最多使用一次免费边”，只需要知道免费机会是否用过，所以状态 $0/1$ 足够。

如果题目改成“最多使用两次免费边”，状态就不能只有 $0/1$，而应记录已经用了几次：`d[u][0]`、`d[u][1]`、`d[u][2]`。

如果题目改成“路径中最后一条边的颜色会影响下一步能走什么边”，那么状态就要记录上一条边的颜色。

状态不是越多越好。状态太少，会丢失信息；状态太多，会让图变大，程序复杂度上升。第一轮中要能判断“这个维度为什么需要记录”。

例题 3：状态是否足够

题目规定一条路径中最多走一次红色边。若用 `d[u]` 表示到达 u 的最小费用，够不够？

不够。

到达同一个点 u 时，可能有两种情况：已经走过红色边，或者还没走过红色边。后者后面仍能走一条红色边，前者不能。这两种状态的后续可选边不同，必须区分。

应使用 `d[u][0]` 和 `d[u][1]`，分别表示还没走过红色边和已经走过红色边。

状态最短路的复杂度

如果原图有 n 个点、m 条边，每个点拆成 k 个状态，那么状态图大约有 kn 个点。每条原边可能产生若干条状态转移，通常也是原来的常数倍到 k 倍。

若用 Dijkstra，复杂度大致随状态数放大。第一轮中不需要把每个常数算得很细，但要知道：状态维度会直接增加点数和边数。

例如“免费边用过与否”只有 2 个状态，状态图有 $2n$ 个点。复杂度仍和普通 Dijkstra 同一类，只是规模乘了一个小常数。

如果状态维度达到 n 或更大，就不能再随便套这种写法，需要重新分析。

状态最短路和 DP 的关系

状态最短路看起来像 DP，因为它也有 `d[u][state]`。但它的转移不一定按某个简单顺序进行。只要边权非负，通常用 Dijkstra 按当前最小距离推进。

可以把它理解为：状态是点，转移是边，`d` 是状态图中的最短距离。

如果状态之间形成 DAG，也可以按拓扑序转移；如果所有边权都是 1，可以用 BFS；如果边权是非负的一般数值，常用 Dijkstra。选择哪种算法，取决于状态图的边权和结构。

次短路

普通最短路只关心最小距离。次短路关心的是严格大于最短路的最小距离。

设从 s 到 u 的最短距离为 `d1[u]`，严格次短距离为 `d2[u]`。这里 `d2[u]` 必须满足 `d2[u] > d1[u]`，并且在所有严格大于 `d1[u]` 的路径长度中最小。

这个“严格大于”很重要。如果从 s 到 t 有两条不同路径，长度都等于 5，那么它们都只是最短路。若题目要求严格次短路，答案不能仍然是 5，而要找大于 5 的最小路径长度。

路径长度集合	最短路	严格次短路
$5, 7, 9$	5	7
$5, 5, 8$	5	8
$3, 3, 3$	3	不存在
$2, 4, 4, 6$	2	4

例题 4：判断次短路

从 1 到 5 有四条路径，长度分别为 $6, 6, 8, 10$。严格次短路长度是多少？

最短路长度为 6。虽然有两条长度为 6 的路径，但它们都不是严格次短路。严格大于 6 的路径长度中，最小的是 8。

所以严格次短路长度为 8。

维护两个距离

求次短路时，每个点维护两个值：

- `d1[u]`：从起点到 u 的最短距离；
- `d2[u]`：从起点到 u 的严格次短距离。

每次从某个状态扩展一条边，得到候选距离 `nd`。需要按下面规则更新终点 v：

```
if (nd < d1[v]) {
    d2[v] = d1[v];
    d1[v] = nd;
} else if (d1[v] < nd && nd < d2[v]) {
    d2[v] = nd;
}
```

第一种情况：`nd` 比当前最短还短。原来的最短距离会退到次短位置，新的 `nd` 成为最短。

第二种情况：`nd` 严格大于当前最短，又小于当前次短。它更新次短。

如果 `nd == d1[v]`，在严格次短路问题中不更新 `d2[v]`。因为它只是另一条同长度最短路，不是严格更长的路径。

例题 5：更新两个距离

当前 `d1[4]=7`，`d2[4]=12`。现在得到一个到点 4 的候选距离 `nd=5`。如何更新？

因为 $5 < 7$，它成为新的最短距离。原来的最短 7 变成次短。更新后 `d1[4]=5`，`d2[4]=7`。

如果候选距离是 `nd=9`，则 $7 < 9 < 12$，更新次短为 9。

如果候选距离是 `nd=7`，等于最短距离，在严格次短路中不更新。

如果候选距离是 `nd=15`，比当前次短还大，也不更新。

次短路 Dijkstra 写法

在非负边权图中，可以把每个点的最短、次短看作两个状态。优先队列中存一个候选距离和点编号。每次弹出一个状态，用它去松弛邻边。

下面是一种常见写法：

```
struct Edge {
    int to, w;
};

vector<Edge> e[N];
long long d1[N], d2[N];

void solve(int s) {
    memset(d1, 0x3f, sizeof(d1));
    memset(d2, 0x3f, sizeof(d2));

    priority_queue<pair<long long, int>,
        vector<pair<long long, int>>,
        greater<pair<long long, int>>> q;

    d1[s] = 0;
    q.push({0, s});

    while (!q.empty()) {
        long long d = q.top().first;
        int u = q.top().second;
        q.pop();

        if (d > d2[u]) {
            continue;
        }

        for (auto ed : e[u]) {
            int v = ed.to;
            int w = ed.w;
            long long nd = d + w;

            if (nd < d1[v]) {
                d2[v] = d1[v];
                d1[v] = nd;
                q.push({d1[v], v});
                q.push({d2[v], v});
            } else if (d1[v] < nd && nd < d2[v]) {
                d2[v] = nd;
                q.push({d2[v], v});
            }
        }
    }
}
```

这段代码的思想是：一个点不仅可以用最短距离扩展，也可能需要用次短距离扩展。因为某个点的次短路径再接上一条边，可能成为终点的次短路径。

实际写代码时，可以在更新最短时只压入有效距离，避免把无穷大压进队列。第一轮程序阅读更关心规则本身：新距离小于最短时，最短下移为次短；新距离夹在最短和次短之间时，更新次短。

例题 6：手算次短路

有向边如下：

边	权值
$(1,2)$	1
$(2,4)$	2
$(1,3)$	2
$(3,4)$	2
$(1,4)$	5

从 1 到 4 的最短路和严格次短路分别是多少？

路径 $1 \to 2 \to 4$ 长度为 $1+2=3$。路径 $1 \to 3 \to 4$ 长度为 $2+2=4$。直接路径 $1 \to 4$ 长度为 5。

所以最短路长度为 3，严格次短路长度为 4。

注意次短路不是 “边数第二少” 的路径，也不是 “字典序第二条路径”，而是路径长度严格大于最短路的最小值。

等长最短路不是次短路

这是次短路题最容易错的地方。

有向边如下：

边	权值
$(1,2)$	1
$(2,4)$	2
$(1,3)$	1
$(3,4)$	2
$(1,4)$	4

从 1 到 4 有两条长度为 3 的路径：$1 \to 2 \to 4$ 和 $1 \to 3 \to 4$。还有一条长度为 4 的路径 $1 \to 4$。

如果题目要求严格次短路，答案是 4，不是 3。因为两条长度为 3 的路径都属于最短路。

若题目说 “第二条不同路径”，并允许长度相同，那才可能把第二条长度为 3 的路径作为答案。读题时要看清 “严格大于最短路” 这几个字。

真题式完善程序：次短路

2024 年 CSP-S 第一轮完善程序中出现过 “次短路” 框架：给定有向图和 s,t，求长度严格大于最短路的最短路径，并输出长度和方案。

这类程序往往把每个点拆成两层：

- 第一层表示到该点的最短距离；
- 第二层表示到该点的次短距离。

也可以用两个数组写成 `dis[u]` 和 `dis2[u]`。若程序中出现 `dis[n<<1]`，再配合 `dis2 = dis + n` 这种写法，意思通常是把长度为 $2n$ 的数组分成前后两段：前 n 个位置存最短，后 n 个位置存次短。

读这类程序时，要抓住三个更新动作。

第一，当发现一个比当前最短更短的距离时，原来的最短不能直接丢掉，它可能成为次短候选。所以要把旧的最短转移到次短层。

第二，当发现一个严格大于最短、又小于当前次短的距离时，更新次短。

第三，若候选距离等于最短，严格次短路中不能把它当成次短。

很多完善程序的空缺会围绕这三件事设置。

例题 7：补全次短路更新

下面是次短路更新的一部分：

```
long long nd = d + w;

if (nd < d1[v]) {
    d2[v] = d1[v];
    d1[v] = nd;
} else if (__________) {
    d2[v] = nd;
}
```

空缺处应填 `d1[v] < nd && nd < d2[v]`。

这里必须有 `d1[v] < nd`，而不是 `d1[v] <= nd`。因为题目要求严格次短路，等于最短的候选距离不能更新次短。

也必须有 `nd < d2[v]`，因为次短要尽量小。

次短路中的路径还原

若只求长度，维护 `d1` 和 `d2` 就够了。若还要输出方案，就需要记录前驱。

次短路路径还原比普通最短路多一个状态。因为到达点 u 的路径可能是最短状态，也可能是次短状态。前驱不能只写成 `pre[u]`，还要知道当前状态来自哪个点、哪个状态。

可以理解为记录：

- 当前到达的是点 u 的第几层；
- 前一步来自哪个点；
- 前一步来自最短层还是次短层。

若程序把点拆成 $2n$ 个编号，例如 u 表示最短层，$n+u$ 表示次短层，那么 `pre[x]` 可以直接记录前一个状态编号。还原路径时不断沿 `pre` 回溯，再把编号对 n 取模转回原图点号。

例题 8：为什么前驱也要分层

从点 1 到点 4 的最短路径是 $1 \to 2 \to 4$，次短路径是 $1 \to 3 \to 4$。如果只用 `pre[4]`，会有什么问题？

`pre[4]` 只能记录一个前驱。若它记录最短路前驱 2，就无法还原次短路径的前驱 3；若它记录 3，又会覆盖最短路径信息。

所以需要区分“点 4 的最短状态”和“点 4 的次短状态”。它们可以有不同前驱。

状态最短路和次短路的关系

状态最短路和次短路看起来不同，但程序结构有相似之处：都不是只给每个原图点维护一个距离。

状态最短路中，一个点会因为“免费机会是否用过”“已经使用几次特殊操作”“当前颜色或方向”等信息拆成多个状态。

次短路中，一个点会因为“这是到它的最短距离还是次短距离”拆成两个状态。

问题	状态含义
免费边最短路	免费机会是否已使用
最多使用 k 次特殊操作	已使用次数
红蓝边交替	上一条边颜色
次短路	到该点的是最短层还是次短层

它们的共同点是：到达同一个原图点，不同状态的未来意义不同，不能混成一个 `dis[u]`。

状态转移要对应题意

状态最短路最容易错在转移写多或写少。

以“最多一条边免费”为例，从 `(u,0)` 出发有两种选择：正常走，或者免费走。从 `(u,1)` 出发只有正常走。若从 `(u,1)` 还允许免费走，就违反了“最多一次”的限制。若从 `(u,0)` 忘了正常走，就表示免费机会没用时必须立刻使用，也不符合“至多一次”。

再看“必须恰好使用一次免费边”。状态转移仍然相同，但答案不同。至多一次的答案是 `min(d[t][0], d[t][1])`；恰好一次的答案是 `d[t][1]`。

同样的状态图，题目问法不同，输出位置也不同。

例题 9：至多一次和恰好一次

从 1 到 3 有直接边权 2，另有路径 $1 \to 2 \to 3$，两条边权分别为 100 和 100。若可以免费一条边，问从 1 到 3 的最小费用。

若题目说“至多免费一次”，答案是 0 吗？

直接边 $1 \to 3$ 可以免费，费用为 0，所以答案是 0。

若题目改成“至多免费一次，但免费边必须从某个指定集合中选择”，就要看直接边是否在指定集合里。状态转移必须按题目条件写，不能把所有边都当成可免费。

若题目说“可以不用免费机会”，输出时取两个状态的较小值；若题目说“必须用一次”，只能取已使用状态。

多状态的编号

程序里有两种常见写法表示状态。

第一种是二维数组：

```
long long d[N][2];
```

这种写法直观，`d[u][0]` 和 `d[u][1]` 含义清楚。

第二种是把状态压成一个编号：

```
int id = u + k * n;
```

这里 u 是点编号，k 是状态编号。若点编号从 0 开始，状态 k 的点 u 常写成 `k * n + u`。若点编号从 1 开始，写法要按程序习惯调整。

压成一维编号后，普通最短路代码可以直接套在状态图上。但程序阅读时要把编号还原出来，看清它对应哪个原图点和哪个状态。

例题 10：编号还原

点编号从 0 到 $n-1$，状态编号为 0 或 1。程序把状态点写成 `id = state * n + u`。若 $n = 10$，编号 17 表示什么？

因为 $17 = 1 \times 10 + 7$，所以它表示原图点 7、状态 1。

若编号为 6，则 $6 = 0 \times 10 + 6$，表示原图点 6、状态 0。

程序阅读中的优先队列状态

状态最短路中，优先队列里不能只放点编号，还要放状态。

例如免费边最短路中，可以写：

```
struct Node {
    long long dis;
    int u, used;

    bool operator>(const Node &o) const {
        return dis > o.dis;
    }
};
```

队列元素 `(dis,u,used)` 表示当前有一个候选状态：用代价 `dis` 到达点 `u`，免费机会状态为 `used`。

弹出队首后，通常要判断它是否已经过期：

```
if (dis != d[u][used]) {
    continue;
}
```

或者使用 `vis[u][used]` 标记状态是否已经确定。注意这里的访问标记也要带状态。`vis[u][0]` 和 `vis[u][1]` 是不同的，不能只写 `vis[u]`。

例题 11：为什么 vis 要带状态

在免费边最短路中，若点 u 已经以状态 `used=0` 被确定，能不能直接认为点 u 的状态 `used=1` 也不用处理？

不能。

状态 `0` 表示免费机会还没用，状态 `1` 表示已经用过。它们后续能做的事不同，距离也可能不同。

例如到达 u 且没用免费机会的代价是 10，到达 u 且已经用过免费机会的代价是 3。虽然原图点相同，但两个状态都可能对后续答案有贡献。只用 `vis[u]` 会把其中一个状态错误丢掉。

完善程序中的常见空缺

状态最短路的空缺通常围绕初始化、状态转移和答案位置。

空缺位置	应关注的问题
初始距离	起点对应的初始状态是什么
队列元素	是否包含点和状态
正常走边	状态是否保持不变
使用特殊操作	状态是否发生变化
使用次数限制	是否检查还可以使用
最终答案	是未使用、已使用，还是二者取较小

例如免费边问题中：

```
d[s][0] = 0;
q.push({0, s, 0});
```

这里不能写 `d[s][1] = 0` 作为唯一初值，因为一开始还没有使用免费边。

再看转移：

```
if (used == 0) {
    if (d[u][used] < d[v][1]) {
        d[v][1] = d[u][used];
        q.push({d[v][1], v, 1});
    }
}
```

这里进入的是状态 `1`，不是继续留在状态 `0`。因为免费机会已经被当前边用掉。

次短路的空缺则常围绕两个距离的更新：

情况	操作
新距离小于最短	原最短下移为次短，新距离成为最短
新距离等于最短	严格次短路中不更新
新距离在最短和次短之间	更新次短
新距离不小于次短	不更新

例题 12：补全免费边转移

下面代码用于“最多一条边免费”的状态最短路：

```
if (used == 0) {
    if (d[u][used] < d[v][1]) {
        d[v][1] = ________;
        q.push({d[v][1], v, 1});
    }
}
```

空缺处应填 `d[u][used]`。

因为当前边被免费使用，走这条边不增加费用。到达 v 后，免费机会已经使用，所以更新 `d[v][1]`。

若填 `d[u][used] + w`，就变成正常走边，没有体现免费边。若更新 `d[v][0]`，则状态没有改变，也不符合“已经使用免费机会”。

常见错误

第一，把状态压掉。到同一个点时，如果未来能力不同，就不能只用一个 `dis[u]`。

第二，状态变化写错。使用一次特殊操作后，状态必须改变；没有使用时，状态应保持。

第三，答案状态取错。“至多一次”和“恰好一次”的输出不同。前者通常取较小值，后者只能取已使用状态。

第四，`vis` 没有带状态。状态最短路中，`vis[u][0]` 和 `vis[u][1]` 是两件事。

第五，次短路把等长最短路当成次短。若题目要求严格大于最短路，`nd == d1[v]` 不能更新 `d2[v]`。

第六，更新更短距离时忘记下移旧最短。若 `nd < d1[v]`，原来的 `d1[v]` 可能成为次短，不能直接覆盖丢掉。

第七，路径还原时只记录一个前驱。次短路和状态最短路都需要记录状态层，否则无法还原对应路径。

小结

状态最短路的核心是：当“当前点”不足以描述后续选择时，把点扩展成状态。免费边是否用过、特殊操作用了几次、上一条边的类型，都可能成为状态的一部分。状态确定后，再把状态之间的转移看成边，运行 BFS 或 Dijkstra。

次短路的核心是：每个点不能只维护一个最短距离，还要维护严格次短距离。更新时要区分三种情况：新距离更短、夹在最短和次短之间、等于最短或更差。若题目要求严格次短，等长最短路不能算作次短。

本章需要重点记住：

- 状态必须包含影响未来选择的信息；
- 状态图上的点是 “原图点 + 状态”；
- 特殊操作会改变状态，普通操作通常保持状态；
- 状态最短路的 `vis`、`dis`、队列元素都要带状态；
- 次短路维护 `d1` 和 `d2`；
- 严格次短路要求 `d2 > d1`；
- 发现更短距离时，要把旧最短下移为次短；
- 需要输出方案时，前驱也要带状态。

下一章进入 LCA。那一章会回到树上问题，研究如何快速求两个结点的最近公共祖先，并为后续树形结构章节打基础。

0x20 LCA

第 31 章讲了状态最短路与次短路，仍然属于图上的路径问题。本章回到树，讲一个树上非常常见的概念：LCA，也就是最近公共祖先。

树有一个重要特点：任意两个点之间只有一条简单路径。正因为路径唯一，树上很多问题都可以转化为祖先关系、深度和最近公共祖先。CSP-S 第一轮中，LCA 常出现在选择题和程序阅读里，考法不一定要求完整实现算法，但会考：谁是谁的祖先，两个点的 LCA 是什么，路径长度怎么由深度和 LCA 计算，倍增数组 `fa[u][k]` 表示什么。

祖先

在一棵有根树中，根到某个点的路径上，经过的点都是这个点的祖先。

例如树以 1 为根，边为 $\{1,2\},\{1,3\},\{2,4\},\{2,5\},\{3,6\}$。点 5 的祖先有 $1,2,5$。这里通常把一个点也看作自己的祖先。

若点 u 在根到点 v 的路径上，就说 u 是 v 的祖先。若 $u \neq v$，也可以说 u 是 v 的真祖先。

结点	祖先
4	$1,2,4$
5	$1,2,5$
6	$1,3,6$

树没有指定根时，祖先关系没有意义。LCA 必须在有根树中讨论。CSP-S 第一轮题目如果说 “以结点 1 为根”，后面的祖先、深度和 LCA 都要按这个根来判断。

例题 1：判断祖先

树以 1 为根，父子关系为 1 是 $2,3$ 的父亲，2 是 $4,5$ 的父亲，3 是 6 的父亲。判断下列说法。

A. 2 是 5 的祖先。

B. 3 是 5 的祖先。

C. 1 是 6 的祖先。

D. 5 是 5 的祖先。

A 正确。根到 5 的路径是 $1 \to 2 \to 5$，包含 2。

B 错误。根到 5 的路径不经过 3。

C 正确。根是所有点的祖先。

D 通常按 LCA 约定，一个点是自己的祖先，所以正确。若题目特别说 “真祖先”，才要求两个点不同。

深度

深度表示一个点距离根有多远。常见约定有两种：

- 根的深度为 0；

- 根的深度为 1。

两种写法都可以，但同一道题中必须保持一致。程序阅读题里，要看代码如何初始化。

如果代码写：

```
dep[root] = 1;
```

那么根深度为 1。若代码写：

```
dep[root] = 0;
```

那么根深度为 0。

深度常用来比较两个点的位置高低。深度小的点更靠近根，深度大的点更靠近叶子。

例题 2：计算深度

树以 1 为根，边为 $\{1,2\},\{1,3\},\{2,4\},\{2,5\},\{5,7\}$。若根深度为 1，求各点深度。

点 1 深度为 1。点 $2,3$ 是根的儿子，深度为 2。点 $4,5$ 是点 2 的儿子，深度为 3。点 7 是点 5 的儿子，深度为 4。

若改成根深度为 0，所有深度都会少 1。LCA 本身不变，但路径长度公式中的深度值要和程序约定一致。

最近公共祖先

两个点 u,v 的公共祖先，是同时作为 u 的祖先和 v 的祖先的点。最近公共祖先，就是这些公共祖先中深度最大的那个。它常写作 $\mathrm{LCA}(u,v)$。

例如树以 1 为根：

```
      1
    /   \
   2     3
  / \   / \
 4   5 6   7
```

点 4 的祖先是 $1,2,4$，点 5 的祖先是 $1,2,5$。它们的公共祖先是 $1,2$，其中深度更大的是 2，所以 $\mathrm{LCA}(4,5)=2$。

点 4 的祖先是 $1,2,4$，点 6 的祖先是 $1,3,6$。公共祖先只有 1，所以 $\mathrm{LCA}(4,6)=1$。

点 2 和点 5 的 LCA 是 2，因为点 2 是自己的祖先，也是点 5 的祖先。

例题 3：手算 LCA

仍看上面的树，求下列 LCA。

A. $\mathrm{LCA}(6,7)$

B. $\mathrm{LCA}(3,7)$

C. $\mathrm{LCA}(4,2)$

D. $\mathrm{LCA}(4,7)$

A 中，点 6 和点 7 都在点 3 的子树中，最近公共祖先是 3。

B 中，点 3 是点 7 的祖先，也是自己的祖先，所以最近公共祖先是 3。

C 中，点 2 是点 4 的父亲，也是点 2 自己的祖先，所以最近公共祖先是 2。

D 中，点 4 在点 2 的子树中，点 7 在点 3 的子树中，直到根 1 才汇合，所以最近公共祖先是 1。

LCA 的位置关系

LCA 一定在两点到根路径的交汇处。它有几个常用性质。

第一，$\mathrm{LCA}(u,u)=u$。一个点和自己的最近公共祖先就是它自己。

第二，如果 u 是 v 的祖先，那么 $\mathrm{LCA}(u,v)=u$。这类题很容易错把答案写成 u 的父亲。最近公共祖先要求“最近”，既然 u 本身已经是公共祖先，就不需要再往上。

第三，若两个点在同一个儿子子树内，它们的 LCA 也在这个子树内；若它们分别在根的不同儿子子树中，它们的 LCA 是根。

第四，LCA 不一定是两个点之一。例如点 4 和点 5 的 LCA 是 2；也可能是其中一个点，例如点 2 和点 5 的 LCA 是 2。

真题式例题：不可能的 LCA 关系

在一棵以结点 1 为根的树中，已知结点 12 和结点 18 的最近公共祖先是结点 4。下面哪种情况不可能？

A. $\mathrm{LCA}(12,4)=4$

B. $\mathrm{LCA}(18,4)=4$

C. $\mathrm{LCA}(12,18,4)=4$

D. $\mathrm{LCA}(12,1)=4$

已知 $\mathrm{LCA}(12,18)=4$，说明点 4 是点 12 和点 18 的公共祖先，并且是它们所有公共祖先中最深的一个。

A 可能。因为 4 是 12 的祖先，所以 $\mathrm{LCA}(12,4)=4$。

B 可能。同理，4 是 18 的祖先，所以 $\mathrm{LCA}(18,4)=4$。

C 可能。三个点 $12,18,4$ 的最近公共祖先仍然是 4。

D 不可能。根 1 是所有点的祖先，并且是点 12 到根路径上的最高点。$\mathrm{LCA}(12,1)$ 一定是 1，不可能是 4。

这类题要抓住一个原则：根是所有点的祖先。如果一个点和根求 LCA，答案就是根。

路径长度公式

树上任意两点之间只有一条简单路径。若已知每个点深度和它们的 LCA，就可以计算路径上的边数。

设 $w=\mathrm{LCA}(u,v)$。从 u 到 v 的路径，可以拆成 u 向上走到 w，再从 w 向下走到 v。边数为 $\mathrm{dep}(u)+\mathrm{dep}(v)-2\,\mathrm{dep}(w)$。

如果根深度为 1 或 0，这个公式都成立，因为深度差才表示边数。

例如 $\mathrm{dep}(4)=3$，$\mathrm{dep}(7)=3$，$\mathrm{LCA}(4,7)=1$，且 $\mathrm{dep}(1)=1$。路径长度为 $3+3-2\times 1=4$，对应路径 $4\to 2\to 1\to 3\to 7$，确实有 4 条边。

如果树边带权，就不能只用深度计算边权和。需要记录从根到每个点的距离 `dist[u]`，则 u 到 v 的路径权值和为 `dist[u] + dist[v] - 2 * dist[lca]`。

例题 4：路径长度

树以 1 为根，边为 $\{1,2\},\{1,3\},\{2,4\},\{2,5\},\{3,6\},\{6,8\}$。根深度为 1。求点 5 到点 8 的路径边数。

点 5 深度为 3，点 8 深度为 4。它们的 LCA 是 1，深度为 1。

路径边数为 $3+4-2\times1=5$。实际路径是 $5\to2\to1\to3\to6\to8$，有 5 条边。

朴素求 LCA

如果只问一次 LCA，可以直接向上跳。

先让深度较大的点不断向父亲移动，直到两个点深度相同。然后两个点一起向上移动，直到它们相等。第一次相等的位置就是 LCA。

例如点 u 深度为 5，点 v 深度为 3，先让 u 向上跳 2 层，使它和 v 深度相同。若此时两个点相同，答案就是这个点；否则两点一起向父亲移动，直到相遇。

代码可以写成：

```
int lca(int u, int v) {
    while (dep[u] > dep[v]) {
        u = fa[u];
    }
    while (dep[v] > dep[u]) {
        v = fa[v];
    }
    while (u != v) {
        u = fa[u];
        v = fa[v];
    }
    return u;
}
```

这个做法简单，但如果树很高、询问很多，速度不够。第一轮程序阅读中，它仍然可能出现，因为逻辑清楚，适合考祖先关系和循环次数。

例题 5：朴素上跳

树中父亲关系为 `fa[4]=2`，`fa[5]=2`，`fa[2]=1`，`fa[1]=0`。求 $\mathrm{LCA}(4,5)$。

点 4 和点 5 深度相同，但不相等。一起向上跳一次，点 4 变为 2，点 5 也变为 2。此时相等，所以 LCA 是 2。

若求 $\mathrm{LCA}(4,2)$，点 4 比点 2 深一层。先把点 4 向上跳到 2，此时两个点相等，答案就是 2。

倍增祖先

朴素上跳一次只走一层。倍增的想法是预处理每个点向上跳 $1,2,4,8,\ldots$ 层分别到哪里。

设 `up[u][k]` 表示点 u 向上跳 2^k 层后到达的祖先。于是：

- `up[u][0]` 是 u 的父亲；
- `up[u][1]` 是 u 向上跳 2 层的祖先；
- `up[u][2]` 是 u 向上跳 4 层的祖先。

转移关系是：`up[u][k] = up[up[u][k-1]][k-1]`。意思是：先从 u 向上跳 2^{k-1} 层，再从那个点继续向上跳 2^{k-1} 层，加起来就是 2^k 层。

例如要向上跳 13 层，因为 $13=8+4+1$，可以依次用 2^3、2^2、2^0 这些跳跃完成。倍增就是利用二进制拆分，让上跳次数从一层一层变成按大步跳。

例题 6：理解 up

若 `up[9][0]=5`，`up[5][0]=2`，那么 `up[9][1]` 是多少？

`up[9][1]` 表示点 9 向上跳 2 层。点 9 向上一层到 5，再向上一层到 2，所以 `up[9][1]=2`。

也可以用转移式看：`up[9][1]=up[up[9][0]][0]=up[5][0]=2`。

DFS 预处理

倍增 LCA 需要先确定每个点的深度和父亲。通常从根开始 DFS。

```
vector<int> e[N];
int dep[N], up[N][20];

void dfs(int u, int f) {
    dep[u] = dep[f] + 1;
    up[u][0] = f;

    for (int k = 1; k < 20; k++) {
        up[u][k] = up[up[u][k - 1]][k - 1];
    }

    for (int v : e[u]) {
        if (v == f) {
            continue;
        }
        dfs(v, u);
    }
}
```

这里 `f` 是点 u 的父亲。根的父亲常设为 0，并令 `dep[0]=0`，这样根深度就是 1。

如果题目点数更大，`20` 需要换成足够大的值。因为 2^{20} 已经超过 10^6，所以点数在 10^5 级别时，20 通常够用。第一轮中，看到 `up[N][20]` 或 `fa[N][LOG]`，要能判断它是在存倍增祖先。

易错：不要走回父亲

DFS 遍历无向树时，每条边存了两个方向。若从 u 枚举到父亲 f 后又递归回去，就会在父子之间反复调用。

所以代码里必须有：

```
if (v == f) {
    continue;
}
```

有些程序用 `vis` 数组避免走回已经访问过的点，本质也是为了防止回到父亲。

倍增求 LCA

倍增求 LCA 分两步。

第一步，把较深的点向上跳到和另一个点同一深度。

第二步，如果两个点还不相同，就从大到小尝试同时上跳。要求跳完后两个点仍然不同。最后它们的父亲就是 LCA。

代码如下：

```
int lca(int u, int v) {
    if (dep[u] < dep[v]) {
        swap(u, v);
```

```
    }

    int d = dep[u] - dep[v];

    for (int k = 19; k >= 0; k--) {
        if (d >> k & 1) {
            u = up[u][k];
        }
    }

    if (u == v) {
        return u;
    }

    for (int k = 19; k >= 0; k--) {
        if (up[u][k] != up[v][k]) {
            u = up[u][k];
            v = up[v][k];
        }
    }

    return up[u][0];
}
```

第一段循环用深度差的二进制，把深的点上跳。第二段循环从大到小试跳，是为了尽量接近 LCA，但不能直接跳到 LCA 上方。

最后 u 和 v 停在 LCA 的两个不同儿子方向上，所以返回它们的父亲 up[u][0]。

例题 7：第一步为什么先调平深度

若一个点深度为 6，另一个点深度为 3，它们的 LCA 深度不可能大于 3。较深的点必须先向上跳到深度 3，才有可能和另一个点一起比较。

如果不调平深度，两个点同步向上跳时，一个点会一直比另一个点低，不能正确判断它们是否处在 LCA 的两侧。

例如点 u 是点 v 的后代。调平后，u 可能直接跳到 v，此时 LCA 就是 v。若不调平，就容易把答案错判为 v 的父亲或更高位置。

为什么第二步从大到小跳

第二步中，从最大的 k 开始尝试，如果 up[u][k] != up[v][k]，就让两个点同时跳到各自的 up[][k]。

这个条件表示：跳这么远以后，两点还没有跳到同一个祖先，说明还可以继续向上靠近 LCA，但不会越过 LCA。若 up[u][k] == up[v][k]，说明跳这么远会让两点到达同一个点，这个点可能就是 LCA，也可能在 LCA 上方。为了不跳过答案，此时不能跳。

从大到小尝试，可以用较少次数把两点推到 LCA 的正下方。

例题 8：同步上跳

设点 u 和点 v 深度相同，但不相等。若 up[u][2] != up[v][2]，可以同时跳 4 层吗？

可以。

因为跳 4 层后它们仍然不同，说明 LCA 还在更上面，或者至少没有被跳过到同一个祖先。这样跳能快速靠近答案。

若 up[u][2] == up[v][2]，就不能跳 4 层。否则两点会直接汇合到同一个祖先，可能错过 LCA 下方的定位过

程。此时应尝试更小的跳跃。

判断祖先关系

有时题目不直接问 LCA，而是问某个点是否为另一个点的祖先。

一种朴素方法是不断从点 v 向上跳父亲，看能否遇到 u。若询问很多，可以借助 DFS 序。

在 DFS 中记录每个点进入时间 `tin[u]` 和离开时间 `tout[u]`。若点 u 是点 v 的祖先，则 v 的整个子树访问区间包含在 u 的子树访问区间中，也就是 `tin[u] <= tin[v] && tout[v] <= tout[u]`。

第一轮中，如果程序出现 `tin`、`tout`，常用来判断祖先关系或子树范围。

例题 9：用 DFS 序判断祖先

若 `tin[2]=3`，`tout[2]=8`，`tin[5]=5`，`tout[5]=6`，点 2 是否可能是点 5 的祖先？

可能。因为 5 的访问区间 $[5,6]$ 被包含在 2 的访问区间 $[3,8]$ 中。

若另一个点 7 的 `tin[7]=9`，`tout[7]=10`，则点 2 不可能是点 7 的祖先，因为 $[9,10]$ 不在 $[3,8]$ 内。

LCA 与树上路径

树上路径常借助 LCA 拆开。

从 u 到 v 的路径会先从 u 向上到 $w=\mathrm{LCA}(u,v)$，再从 w 向下到 v。因此，很多树上路径信息都可以拆成“根到点”的信息相减。

若 `sum[u]` 表示从根到点 u 的边权和，则 u 到 v 的路径权值和为 `sum[u] + sum[v] - 2 * sum[w]`。

若 `dep[u]` 表示深度，则路径边数为 `dep[u] + dep[v] - 2 * dep[w]`。

若点权也要统计，要看题目是否包含 LCA 这个点。边权路径和不需要额外加点；点权路径和通常要加回 LCA 的点权，因为根到 u 和根到 v 的点权前缀中，LCA 被减了两次。

本章只要求掌握边数和边权路径和。点权路径和留到具体题目中再按题意处理。

例题 10：带权路径

树边权如下：$\{1,2\}$ 权值 3，$\{1,3\}$ 权值 4，$\{2,4\}$ 权值 5，$\{2,5\}$ 权值 6。以 1 为根，求点 4 到点 5 的路径权值和。

点 4 到根的距离为 $3+5=8$。点 5 到根的距离为 $3+6=9$。它们的 LCA 是 2，点 2 到根的距离为 3。

路径权值和为 $8+9-2\times 3=11$。实际路径是 $4\to 2\to 5$，权值为 $5+6=11$。

程序阅读：倍增数组

LCA 程序阅读题中，最常见的数组名有 `fa`、`up`、`dep`、`lg`。

数组	常见含义
`dep[u]`	点 u 的深度
`fa[u][0]`	点 u 的父亲
`fa[u][k]`	点 u 的 2^k 级祖先
`lg[i]`	与二进制长度或最大可跳层数有关
`dist[u]`	根到点 u 的距离

看到下面这种转移，就应想到倍增祖先：

```
fa[u][k] = fa[fa[u][k - 1]][k - 1];
```

它不是在做动态规划求最优值，而是在预处理“向上跳”的表。

看到下面这种循环，就应想到把深的点上跳：

```
for (int k = LOG - 1; k >= 0; k--) {
    if (dep[fa[u][k]] >= dep[v]) {
        u = fa[u][k];
    }
}
```

不同代码写法略有差别。有的用深度差的二进制，有的用 `dep[fa[u][k]] >= dep[v]` 判断能不能跳。它们的目的都是让两个点到达同一深度。

常见错误

第一，忘记一个点是自己的祖先。若 u 是 v 的祖先，则 $\mathrm{LCA}(u,v)=u$，不是 u 的父亲。

第二，把根和任意点的 LCA 判错。以 1 为根时，$\mathrm{LCA}(1,u)=1$。

第三，混淆深度约定。根深度为 0 或 1 都可以，但路径长度公式中的深度必须来自同一套约定。

第四，倍增预处理时 `fa[u][k]` 的含义写错。它表示 2^k 级祖先，不是第 k 级祖先。

第五，同步上跳时条件写反。若 `fa[u][k] != fa[v][k]`，才同时跳；若相等就跳，可能直接越过答案。

第六，无向树 DFS 时忘记跳过父亲。边存了两个方向，不跳过父亲会反复递归。

第七，路径权值和公式用深度代替距离。边权不全为 1 时，路径权值和要用根到点的距离 `dist`，不能用 `dep`。

完善程序中的常见空缺

LCA 完善程序常在预处理和查询函数中留空。

预处理部分常见空缺如下：

空缺位置	应关注的问题
设置深度	`dep[u] = dep[f] + 1`
设置父亲	`fa[u][0] = f`
倍增转移	`fa[u][k] = fa[fa[u][k-1]][k-1]`
DFS 枚举儿子	跳过父亲 `v == f`
边权距离	`dist[v] = dist[u] + w`

查询部分常见空缺如下：

空缺位置	应关注的问题
调整深度	把较深点上跳
祖先特判	若两点相等直接返回
同步上跳	从大到小判断祖先不同
返回答案	返回 `fa[u][0]`

例题 11：补全倍增转移

下面代码中，空缺处应填什么？

```
for (int k = 1; k < LOG; k++) {
    fa[u][k] = ________;
}
```

应填 `fa[fa[u][k - 1]][k - 1]`。

点 u 的 2^k 级祖先，可以先跳 2^{k-1} 层到 `fa[u][k-1]`，再从那里继续跳 2^{k-1} 层。

例题 12：补全查询返回

下面是 LCA 查询的最后部分：

```
for (int k = LOG - 1; k >= 0; k--) {
    if (fa[u][k] != fa[v][k]) {
        u = fa[u][k];
        v = fa[v][k];
    }
}

return ________;
```

空缺处应填 `fa[u][0]`，也可以填 `fa[v][0]`。

循环结束后，u 和 v 还不是同一个点，但它们的父亲已经相同。这个共同父亲就是 LCA。

若直接返回 `u` 或 `v`，会返回 LCA 下方的某个点，答案错误。

小结

LCA 是两个点公共祖先中深度最大的那个。若一个点是另一个点的祖先，那么它们的 LCA 就是这个祖先本身。若一个点和根求 LCA，答案一定是根。

树上两点路径可以用 LCA 拆开。若 $w = \mathrm{LCA}(u, v)$，路径边数为 `dep[u] + dep[v] - 2 * dep[w]`。若树边带权，把 `dep` 换成根到点的距离 `dist`，即可求路径权值和。

倍增 LCA 的核心是预处理 `fa[u][k]`，表示点 u 的 2^k 级祖先。查询时先把较深点上跳到同一深度，再从大到小同步上跳，最后返回共同父亲。

本章需要重点记住：

- LCA 必须在有根树中讨论；
- 一个点通常视为自己的祖先；
- 最近公共祖先是公共祖先中深度最大的点；
- 倍增数组 `fa[u][k]` 表示 2^k 级祖先；
- 路径边数公式依赖深度和 LCA；
- 带权路径要用根到点距离 `dist`；
- DFS 预处理无向树时要跳过父亲；
- LCA 查询最后返回的是共同父亲，不是同步上跳后的两个点本身。

下一章进入树的重心与树形结构。那一章会继续研究树的整体形态，特别是“以哪个点为中心时，最大子树尽量小”这类第一轮常见判断。

0x21 树的重心与树形结构

第 32 章讲了 LCA，用最近公共祖先处理树上两点路径。本章继续看树的整体形态，重点讲树的重心、子树大小、换根视角和一些第一轮常见的树形判断。

树是一类特殊的图：连通、无环，任意两点之间有且只有一条简单路径。正因为结构稳定，树上很多题可以不靠复杂算法，而靠“删掉一个点后树会分成几块”“以哪个点为根时子树有多大”“某个点是否能让树尽量平衡”来判断。

删掉一个点

在一棵树中删掉一个点，同时删去与它相连的边，原树会被分成若干个连通块。这个数量等于这个点在树中的度数。若删掉的是叶子，剩下的树仍然是一整块；若删掉的是一个度数较大的点，树会被分成多块。

例如有一棵树，边为 $\{1,2\},\{1,3\},\{2,4\},\{2,5\},\{3,6\}$。若删掉点 1，剩下两块：$\{2,4,5\}$ 和 $\{3,6\}$。若删掉点 2，剩下三块：$\{4\}$、$\{5\}$ 和 $\{1,3,6\}$。

删点以后形成的块大小，正是树的重心要关心的内容。

例题 1：删点后的连通块

树有 7 个点，边为 $\{1,2\},\{1,3\},\{1,4\},\{3,5\},\{3,6\},\{6,7\}$。删掉点 3 后，剩下哪些连通块？

点 3 与点 $1,5,6$ 相连。删掉点 3 后：

- 沿点 1 的方向，可以到达 $\{1,2,4\}$；
- 点 5 单独成为一块；
- 沿点 6 的方向，可以到达 $\{6,7\}$。

所以剩下三个连通块，大小分别为 $3,1,2$。

树的重心

树的重心，是指删掉这个点以后，剩余各连通块大小的最大值尽量小的点。

如果一棵树有 n 个点，点 u 是重心，那么删掉 u 后，每个剩余连通块的大小都不超过 $n/2$。反过来，如果某个点删掉后所有连通块大小都不超过 $n/2$，它就是重心。

这个条件很适合手算和程序判断。对每个点，计算删掉它以后最大的那一块有多大；让这个最大值尽量小的点，就是重心。

例如一条链 $1-2-3-4-5$。删掉点 3 后，左右两边大小都是 2，最大块大小为 2。删掉点 2 后，剩下大小为 1 和 3 的两块，最大块大小为 3。所以点 3 比点 2 更适合作为重心。

点	删除后最大连通块大小
1	4
2	3
3	2
4	3
5	4

因此这条 5 个点的链的重心是点 3。

例题 2：六点链的重心

一条链为 $1-2-3-4-5-6$。它的重心是谁？

逐个看中间位置。删掉点 3 后，左边为 $\{1,2\}$，右边为 $\{4,5,6\}$，最大块大小为 3。删掉点 4 后，左边为 $\{1,2,3\}$，右边为 $\{5,6\}$，最大块大小也是 3。

删掉点 2 后，会出现大小为 4 的块；删掉点 5 后，也会出现大小为 4 的块。其他点更差。

所以这条 6 个点的链有两个重心：点 3 和点 4。

这说明树的重心不一定只有一个。

一个或两个重心

一棵树的重心只有两种可能：一个或两个。如果有两个，它们一定相邻。

直观理解是：如果删掉某个点后存在一块大小超过 $n/2$，那么重心一定在那一大块方向上；沿着这个方向移动，会让最大块变小。这个过程不会来回绕，因为树中没有环。最后会停在一个点，或者停在一条边的两端。

当树有两个重心时，删掉连接它们的那条边，会把树分成两个大小相等的部分。因此两个重心只可能出现在 n 为偶数时。若 n 为奇数，不可能分成两个大小都是 $n/2$ 的整数块，所以重心一定只有一个。

但 n 为偶数时，不一定有两个重心，也可能只有一个。星形树有 6 个点，一个中心连着 5 个叶子，它的重心只有中心点一个。

点数情况	重心个数
n 为奇数	一定只有 1 个
n 为偶数	可能 1 个，也可能 2 个
有 2 个重心	两个重心一定相邻

真题精讲：哪种树一定只有一个重心

2023 年 CSP-S 第一轮考过树的重心：

在图论中，树的重心是树上的一个结点，以该结点为根时，使得其所有的子树中结点数最多的子树的结点数最少。一棵树可能有多个重心。请问下面哪种树一定只有一个重心？

A. 4 个结点的树

B. 6 个结点的树

C. 7 个结点的树

D. 8 个结点的树

答案是 C。

树的重心可能有一个或两个。如果有两个重心，它们相邻，并且删去它们之间的边后，整棵树会被分成两个大小相同的部分。所以有两个重心时，结点总数必须是偶数。

7 是奇数，不可能分成两个相等的整数部分，所以 7 个结点的树一定只有一个重心。

而 4、6、8 都是偶数，它们有些树只有一个重心，有些树有两个重心。例如一条 4 个点的链有两个重心，中间两个点都是重心；一个 6 个点的星形树只有一个重心。

这道题不是让你画出所有树，而是要抓住“两个重心只能出现在偶数点树中”这个性质。

变式：偶数点一定有两个重心吗

一棵有 8 个结点的树一定有两个重心吗？

不一定。

例如一个中心点连接另外 7 个叶子，这是一棵 8 个点的星形树。删掉中心点后，剩下 7 个大小为 1 的连通块，最大块大小为 1。删掉任意叶子后，剩下一个大小为 7 的连通块。中心点明显是唯一重心。

所以偶数点树只是 “可能有两个重心”，不是 “一定有两个重心”。

子树大小

求树的重心，通常先任选一个点作为根，计算每个点的子树大小。

设 `sz[u]` 表示以 u 为根的子树大小。DFS 时，先让 `sz[u]=1`，表示包含自己；然后枚举所有儿子 v，递归计算 `sz[v]`，再把它加到 `sz[u]` 上。

```
vector<int> e[N];
int sz[N];

void dfs(int u, int f) {
    sz[u] = 1;

    for (int v : e[u]) {
        if (v == f) {
            continue;
        }

        dfs(v, u);
        sz[u] += sz[v];
    }
}
```

无向树的边存了两个方向，所以 DFS 中必须跳过父亲 `f`。否则会从儿子走回父亲，再从父亲走回儿子，无法正常计算。

例题 3：计算子树大小

树以 1 为根，边为 $\{1,2\},\{1,3\},\{2,4\},\{2,5\},\{3,6\}$。求各点子树大小。

点 $4,5,6$ 是叶子，子树大小都是 1。

点 2 的子树包含 $2,4,5$，所以 `sz[2]=3`。点 3 的子树包含 $3,6$，所以 `sz[3]=2`。点 1 的子树包含全部 6 个点，所以 `sz[1]=6`。

点	子树大小
1	6
2	3
3	2
4	1
5	1
6	1

用子树大小找重心

任选一个点作为根后，删掉点 u，剩下的连通块分为两类。

第一类是 u 的每个儿子子树。对于儿子 v，这一块大小为 `sz[v]`。

第二类是 u 的父亲方向，也就是整棵树中不在 u 子树里的部分。大小为 `n - sz[u]`。如果 u 是根，这一部分大小为 0。

因此，删掉点 u 后最大连通块大小为：所有儿子子树大小和 `n - sz[u]` 中的最大值。

程序常写成：

```
int best = n;
int cen = 0;

void dfs(int u, int f) {
    sz[u] = 1;
    int mx = 0;

    for (int v : e[u]) {
        if (v == f) {
            continue;
        }

        dfs(v, u);
        sz[u] += sz[v];
        mx = max(mx, sz[v]);
    }

    mx = max(mx, n - sz[u]);

    if (mx < best) {
        best = mx;
        cen = u;
    }
}
```

这里 `mx` 就是删掉点 u 后最大连通块的大小。让 `mx` 最小的点就是重心。

若题目要求输出所有重心，就不能只记录一个 `cen`。当 `mx == best` 时，也要把当前点加入答案。

例题 4：手算重心

树有 6 个点，边为 $\{1,2\},\{1,3\},\{2,4\},\{2,5\},\{3,6\}$。以 1 为根，求重心。

前面已经算出 `sz[1]=6`，`sz[2]=3`，`sz[3]=2`，`sz[4]=sz[5]=sz[6]=1`。

删掉点 1 后，剩下点 2 子树大小 3，点 3 子树大小 2，最大块为 3。

删掉点 2 后，儿子方向有点 4、点 5 两块，各为 1；父亲方向大小为 `6 - sz[2] = 3`，即 $\{1,3,6\}$。最大块为 3。

删掉点 3 后，儿子方向有点 6 一块，大小 1；父亲方向大小为 `6 - sz[3] = 4`。最大块为 4。

删掉叶子点时，父亲方向大小为 5，最大块为 5。

最小的最大块大小为 3，点 1 和点 2 都可以达到。所以这棵树有两个重心：1 和 2。

根的选择不影响重心

求子树大小时，我们会任选一个点作为根。这只是为了方便计算，并不会改变原树的重心。

树的重心定义是“删掉某点后，剩下连通块最大值尽量小”。这个定义和根无关。把树临时以 1 为根，只是为了把删点后的块分成“儿子子树”和“父亲方向”两类。

所以程序中以哪个点为根进行 DFS，并不影响最后重心是谁。只要计算 `n - sz[u]` 这一块时没有漏掉父亲方向，结果就是正确的。

例题 5：为什么要算父亲方向

仍以 1 为根。若判断点 2 是否为重心，只看它的儿子子树大小，会看到点 4 和点 5 的子树都只有 1。如果漏掉父亲方向，就会误以为删掉点 2 后最大块为 1。

但删掉点 2 后，点 $1, 3, 6$ 仍然连在一起，形成大小为 3 的连通块。这一块就是 `n - sz[2]`。

所以求重心时，必须同时考虑所有儿子子树和父亲方向。

树的直径

树的直径是树上最长简单路径的长度。它也属于常见树形结构。

如果边没有权值，直径长度通常指路径上的边数。如果边有权值，直径长度通常指路径边权和。

例如一条链 $1-2-3-4-5$ 的直径是从 1 到 5，长度为 4 条边。星形树中，任意两个叶子之间的路径经过中心，直径长度为 2 条边。

求直径常用两次搜索：

1. 从任意点出发，找到距离它最远的点 a；
2. 从 a 出发，再找到距离 a 最远的点 b；
3. a 到 b 的距离就是树的直径。

这个结论在复赛中常用。第一轮中更可能考手算或读程序：看到两次 DFS/BFS 找最远点，很可能是在求树的直径。

例题 6：手算直径

树边为 $\{1,2\}, \{2,3\}, \{3,4\}, \{3,5\}, \{5,6\}$。求这棵树的直径长度。

最长路径可以从点 1 到点 6，路径为 $1 \to 2 \to 3 \to 5 \to 6$，长度为 4 条边。也可以从点 4 到点 6，路径为 $4 \to 3 \to 5 \to 6$，长度为 3 条边。

所以直径长度为 4。

直径可能不唯一，但直径长度是确定的。

重心和直径不是一回事

重心和直径都描述树的 “中心感”，但它们不是同一个概念。

重心关心删掉某点后，最大连通块尽量小。它看的是子树规模平衡。

直径关心树上最长路径。直径中点关心的是距离平衡。

这两个点可能相同，也可能不同。

例如一条链中，重心和直径中点位置一致。长度为 5 个点的链，重心是中间点，直径中点也是中间点。长度为 6 个点的链，重心有两个，中间两点也对应直径中间位置。

但在一些带有大分支的树中，重心更偏向结点数量多的一侧，而直径中点更看距离。CSP-S 第一轮中如果题目明确说 “删去该点后最大子树最小”，就是重心；如果说 “树上最长路径”，就是直径。

概念	关注内容
树的重心	删点后各连通块大小
树的直径	两点间最长路径
LCA	两点到根路径的最近交汇点

例题 7：区分概念

题目说：“选择一个点作为根，使得它所有子树中结点数最多的子树尽量小。”这是在求什么？

这是在求树的重心。

如果题目说“求树上距离最远的两点之间的距离”，才是在求树的直径。如果题目说“求两个点的最近公共祖先”，才是 LCA。

读题时先看目标：是平衡子树大小，还是最大路径长度，还是祖先关系。

换根视角

很多树形题会说“以某点为根时”。换根并不会改变树的边，只会改变父子关系。

例如无根树中有边 $\{1,2\},\{2,3\},\{2,4\}$。如果以 1 为根，点 2 是 1 的儿子，点 $3,4$ 是 2 的儿子。如果以 3 为根，点 2 是 3 的儿子，点 $1,4$ 是 2 的儿子。

同一棵树，根不同，子树大小会变化。重心定义中的“以该点为根时，使得其所有子树中结点数最多的子树最小”，其实等价于“删掉该点后最大连通块最小”。因为以 u 为根时，u 的每个儿子子树，正好对应删掉 u 后的一块。

例题 8：换根后的子树

无根树边为 $\{1,2\},\{2,3\},\{2,4\}$。若以 1 为根，点 2 的子树包含哪些点？若以 3 为根，点 2 的子树包含哪些点？

以 1 为根时，树方向是 $1\to2\to3$ 和 $2\to4$，点 2 的子树包含 $2,3,4$。

以 3 为根时，树方向是 $3\to2\to1$ 和 $2\to4$，点 2 的子树包含 $2,1,4$。

所以“子树”依赖根。题目没有指定根时，要先看它是否在讨论无根树结构，还是默认以某个点为根。

树形 DP 的影子

后面第七编会讲 DP。本章只提前说明树上程序中常见的一类结构：先处理儿子，再合并到父亲。

例如计算子树大小：

```
sz[u] = 1;

for (int v : e[u]) {
    if (v == f) {
        continue;
    }

    dfs(v, u);
    sz[u] += sz[v];
}
```

这个过程就是“先算儿子，再算父亲”。很多树形程序都有类似结构：递归进入儿子，回到父亲后把儿子的结果合并过来。

第一轮程序阅读中，只要看到 DFS 里先递归 `dfs(v,u)`，后面再用 `sz[v]`、`dp[v]` 更新 `sz[u]` 或 `dp[u]`，就要意识到它在做自底向上的树形统计。

本章不展开树形 DP 的状态设计，只要求能读懂“子树信息如何汇总到父亲”。

程序阅读：找重心

下面是一段求树的重心的代码。点编号从 1 到 n。

```
vector<int> e[N];
int sz[N], best, cen;

void dfs(int u, int f) {
    sz[u] = 1;
    int mx = 0;

    for (int v : e[u]) {
        if (v == f) {
            continue;
        }

        dfs(v, u);
        sz[u] += sz[v];
        mx = max(mx, sz[v]);
    }

    mx = max(mx, n - sz[u]);

    if (mx < best) {
        best = mx;
        cen = u;
    }
}

int main() {
    cin >> n;

    for (int i = 1; i < n; i++) {
        int u, v;
        cin >> u >> v;
        e[u].push_back(v);
        e[v].push_back(u);
    }

    best = n;
    dfs(1, 0);

    cout << cen << endl;
}
```

阅读时要抓住三个变量：

- `sz[u]` 是以当前 DFS 根为准的子树大小；
- `mx` 是删掉点 u 后最大连通块大小；
- `best` 是目前见过的最小 `mx`。

这段程序只输出一个重心。如果树有两个重心，它会输出 DFS 遍历过程中较早更新到的那个，具体是哪一个与遍历顺序有关。若题目要求输出所有重心，代码还要处理 `mx == best` 的情况。

例题 9：程序输出哪个重心

一条链为 $1-2-3-4$，按边输入顺序建立邻接表：$\{1,2\},\{2,3\},\{3,4\}$。上面的程序以 1 为 DFS 起点，并只在 `mx < best` 时更新答案。它可能输出哪个重心？

这条链有两个重心：点 2 和点 3。

DFS 从 1 出发，会先递归到 4，再回到 3、2、1。计算完成点 3 时，它的 `mx` 为 2，会更新答案为 3。计算点 2 时，它的 `mx` 也是 2，但条件是 `mx < best`，不是 `mx <= best`，所以不会覆盖点 3。

因此程序输出点 3。

如果改成 `mx <= best`，或者邻接表遍历顺序变化，输出的重心可能不同。若题目只要求任意一个重心，这样没问题；若要求编号最小重心，就必须额外处理。

程序阅读：树的直径

下面代码用两次 DFS 求无权树直径。

```
vector<int> e[N];
int dis[N];

void dfs(int u, int f) {
    for (int v : e[u]) {
        if (v == f) {
            continue;
        }

        dis[v] = dis[u] + 1;
        dfs(v, u);
    }
}

int farthest(int s) {
    memset(dis, 0, sizeof(dis));
    dfs(s, 0);

    int p = s;
    for (int i = 1; i <= n; i++) {
        if (dis[i] > dis[p]) {
            p = i;
        }
    }

    return p;
}
```

若主程序写：

```
int a = farthest(1);
int b = farthest(a);
cout << dis[b] << endl;
```

它输出的是树的直径长度。第一次从任意点 1 找到一个直径端点 a，第二次从 a 找到另一个端点 b，`dis[b]` 就是直径长度。

程序阅读时，要注意第二次 `farthest(a)` 会重新计算 `dis`。最后输出的 `dis[b]` 是从点 a 到点 b 的距离，不是从点 1 到点 b 的距离。

例题 10：追踪直径程序

一条链为 $1-2-3-4-5$。执行上面的程序，`a` 和 `b` 可能是什么？

从点 1 出发，最远点是 5，所以 `a=5`。

再从点 5 出发，最远点是 1，所以 `b=1`。此时 `dis[1]=4`，输出直径长度 4。

若树有多个最远点，程序会根据遍历顺序或扫描顺序选其中一个。直径端点可能不唯一，但输出的直径长度应相同。

树的重心与删除后的最大块

选择题中有时不会直接问“重心”，而是问某个点删掉后最大连通块大小。

做这类题，可以按点的每个相邻方向分块。对于某个点 u，它的每条相邻边都通向删点后的一块。若这条边通向儿子，块大小是儿子子树大小；若通向父亲，块大小是 $n-\mathrm{sz}[u]$。

不要把“度数最大”当成“重心”。度数大只是说明删掉它后块数多，不代表最大块小。星形树中中心点确实是重心，但在其他树中，高度数点可能连接着一个很大的分支，未必是重心。

例题 11：度数最大不一定是重心

有一棵树，点 1 连接点 $2,3,4$，同时点 4 连接一条长链 $5-6-7-8$。点 1 的度数为 3，点 4 的度数为 2。谁更可能接近重心？

删掉点 1 后，会出现点 2、点 3 两个单点块，以及包含 $4,5,6,7,8$ 的大块，最大块大小为 5。

删掉点 4 后，会出现包含 $1,2,3$ 的块，以及包含 $5,6,7,8$ 的块，最大块大小为 4。

所以点 4 比点 1 更接近重心。度数大不等于重心。

完善程序中的常见空缺

树的重心和树形结构程序，常见空缺集中在 DFS 汇总和最大块计算。

求子树大小时：

空缺位置	应关注的问题
初始化	`sz[u] = 1`
跳过父亲	`if (v == f) continue`
递归儿子	`dfs(v, u)`
汇总大小	`sz[u] += sz[v]`

求重心时：

空缺位置	应关注的问题
儿子方向最大块	`mx = max(mx, sz[v])`
父亲方向最大块	`mx = max(mx, n - sz[u])`
更新答案	比较 `mx` 和当前最优值
多个重心	处理 `mx == best`

求直径时：

空缺位置	应关注的问题
距离转移	`dis[v] = dis[u] + 1` 或加边权
第一次搜索	从任意点找最远点
第二次搜索	从第一次最远点再找最远点
输出	第二次搜索得到的最大距离

例题 12：补全重心代码

下面代码用于计算删掉点 u 后最大连通块大小：

```
sz[u] = 1;
int mx = 0;

for (int v : e[u]) {
    if (v == f) {
        continue;
    }

    dfs(v, u);
    sz[u] += sz[v];
    mx = max(mx, ________);
}

mx = max(mx, ________);
```

两个空缺分别应填 `sz[v]` 和 `n - sz[u]`。

`sz[v]` 表示删掉 u 后，儿子 v 所在那一块的大小。`n - sz[u]` 表示删掉 u 后，父亲方向那一块的大小。

若漏掉第二个空缺，就只统计了向下的子树，根以外的点会被错误判断。

常见错误

第一，把重心理解成度数最大的点。重心看的是删点后最大连通块大小，不是度数。

第二，认为偶数点树一定有两个重心。偶数点树可能有一个重心，也可能有两个。

第三，认为奇数点树可能有两个重心。两个重心会把树分成大小相等的两部分，因此奇数点树一定只有一个重心。

第四，求重心时漏掉父亲方向的连通块。以某点为根计算子树时，删掉非根结点后，父亲方向大小是 `n - sz[u]`。

第五，把重心和直径中点混为一谈。重心平衡的是结点数量，直径中点平衡的是最长路径距离。

第六，树形 DFS 忘记跳过父亲。无向树边存了两个方向，必须避免走回父亲。

第七，程序只记录一个重心，却题目要求所有重心。遇到 `mx == best` 时是否处理，要看题目要求。

小结

树的重心关注删掉一个点后，剩余各连通块的最大大小。一个点是重心，当且仅当删掉它后每个连通块大小都不超过 $n/2$。一棵树有一个或两个重心；若有两个，它们一定相邻，并且只可能出现在偶数点树中。奇数点树一定只有一个重心。

求重心时，常任选一个点作为根，先 DFS 求子树大小。对于点 u，删掉它后的最大块来自两类位置：儿子子树 `sz[v]`，以及父亲方向 `n - sz[u]`。让这个最大值尽量小的点就是重心。

本章还整理了树的直径、换根视角和树形 DFS。直径关心最长路径，重心关心删点后的规模平衡，二者不是同一个概念。

本章需要重点记住：

- 树的重心看删点后的最大连通块大小；
- 重心有一个或两个，两个重心一定相邻；
- 奇数点树一定只有一个重心；
- 求重心时不能漏掉父亲方向的块；
- 子树大小依赖所选根，但重心本身与临时选根无关；
- 直径是最长路径，不是重心；
- 树形 DFS 常见结构是先递归儿子，再汇总到父亲；
- 无向树 DFS 必须跳过父亲。

到这里，第五编 “树与图” 的主线结束。下一编进入 “字符串与序列”，从字符串匹配开始，继续讲 CSP-S 第一轮中常见的字符串程序阅读、KMP、字符串哈希、LCS 和逆序思想。

第六编 字符串与序列

0x22 字符串匹配

第五编讲完了树与图。本章开始进入第六编“字符串与序列”。这一编的内容看起来比图论轻一些，但 CSP-S 第一轮很喜欢把字符串写进程序阅读题中：下标从 0 开始还是从 1 开始，循环边界有没有越界，匹配成功后返回什么位置，失败时指针怎样移动，都会影响答案。

本章讲字符串匹配的基本问题：给定一个文本串和一个模式串，判断模式串是否在文本串中出现，或者找出出现位置。第 35 章会单独讲 KMP，本章不提前展开失配数组。

文本串和模式串

字符串匹配中，通常有两个字符串。

较长的那个叫文本串，也可以叫主串。要寻找的那个叫模式串。

例如文本串为 `"ababcabc"`，模式串为 `"abc"`。从文本串下标 2 开始，有连续三个字符 `"abc"`，所以模式串在文本串中出现了一次。

若使用 C++ 的 `string`，下标从 0 开始。字符串 `"ababcabc"` 的下标如下：

下标	0	1	2	3	4	5	6	7
字符	a	b	a	b	c	a	b	c

模式串 `"abc"` 从下标 2 开始匹配，也从下标 5 开始匹配。

如果题目问“第一次出现的位置”，要看它要求输出从 0 开始的位置，还是从 1 开始的位置。C++ 标准库 `find` 返回的是从 0 开始的下标。

例题 1：找出现位置

文本串为 `"banana"`，模式串为 `"ana"`。模式串在哪些位置出现？

按 0 开始下标看，文本串为：

下标	0	1	2	3	4	5
字符	b	a	n	a	n	a

从下标 1 开始，连续三个字符是 `"ana"`。从下标 3 开始，连续三个字符也是 `"ana"`。

所以模式串出现位置为 1 和 3。

注意这两个出现是重叠的。字符串匹配中，如果题目没有特别说明“不允许重叠”，通常要把重叠出现也算进去。

子串和子序列

字符串中有两个容易混淆的词：子串和子序列。

子串要求连续。比如 `"abcde"` 的子串可以是 `"abc"`、`"bcd"`、`"de"`，但 `"ace"` 不是子串。

子序列不要求连续，但要求相对顺序不变。比如 `"abcde"` 的子序列可以是 `"ace"`、`"bd"`、`"abe"`。

概念	是否要求连续	示例
子串	要求连续	`"bcd"` 是 `"abcde"` 的子串
子序列	不要求连续	`"ace"` 是 `"abcde"` 的子序列

字符串匹配通常找的是子串，而不是子序列。若题目说“模式串在文本串中出现”，一般指连续出现。

例题 2：区分子串和子序列

判断 `"ace"` 是否为 `"abcde"` 的子串，是否为它的子序列。

`"ace"` 不是子串，因为在 `"abcde"` 中，`a`、`c`、`e` 不连续。

`"ace"` 是子序列，因为可以从 `"abcde"` 中依次保留 `a`、`c`、`e`，删除其他字符。

本章讨论字符串匹配，默认关注连续匹配，也就是子串。

朴素匹配

朴素匹配的想法很直接：枚举文本串中的每个可能起点，再逐字符比较模式串。

设文本串为 `s`，长度为 n；模式串为 `t`，长度为 m。若从 `s[i]` 开始匹配，就要比较 `s[i+j]` 和 `t[j]`，其中 $0 \le j < m$。

代码如下：

```
int match(string s, string t) {
    int n = s.size();
    int m = t.size();

    for (int i = 0; i + m <= n; i++) {
        bool ok = true;

        for (int j = 0; j < m; j++) {
            if (s[i + j] != t[j]) {
                ok = false;
                break;
            }
        }

        if (ok) {
            return i;
        }
    }

    return -1;
}
```

外层循环的条件是 `i + m <= n`，因为从下标 `i` 开始要连续取 m 个字符，最后访问到 `s[i + m - 1]`。必须保证 `i + m - 1 < n`，也就是 `i + m <= n`。

若写成 `i < n`，当 `i` 接近末尾时，内层可能访问越界。

例题 3：朴素匹配过程

文本串为 `"ababa"`，模式串为 `"aba"`。用朴素匹配找第一次出现位置。

从下标 0 开始比较：`s[0..2]` 是 `"aba"`，与模式串相同，所以第一次出现位置为 0。

如果要求统计所有出现位置，不能在第一次匹配成功后直接返回，而要继续枚举后面的起点。从下标 2 开始，`s[2..4]` 也是 `"aba"`。所以所有出现位置为 0 和 2。

变式：统计出现次数

若要统计模式串在文本串中出现了多少次，允许重叠，可以这样写：

```
int count_match(string s, string t) {
    int n = s.size();
    int m = t.size();
    int ans = 0;

    for (int i = 0; i + m <= n; i++) {
        bool ok = true;

        for (int j = 0; j < m; j++) {
            if (s[i + j] != t[j]) {
                ok = false;
                break;
            }
        }

        if (ok) {
            ans++;
        }
    }

    return ans;
}
```

这段代码每次外层循环只让 `i` 增加 1，所以会统计重叠出现。

若匹配成功后直接让 `i += m`，则会跳过重叠位置，适用于“不允许重叠统计”的题目。题目没说清楚时，不能随意改变移动方式。

时间复杂度

朴素匹配最坏情况下要比较很多次。

外层最多有 $n-m+1$ 个起点，内层每次最多比较 m 个字符，所以最坏时间复杂度为 $O(nm)$。

例如文本串为 `"aaaaaaaaab"`，模式串为 `"aaaab"`。从很多起点开始，前几个字符都能匹配，到最后才失败，重复比较很多次。

如果 n 和 m 都不大，朴素匹配完全够用。CSP-S 第一轮程序阅读题中，也经常用朴素匹配考下标和边界。第 35 章会讲如何用 KMP 把匹配过程优化到线性级别。

例题 4：比较次数

文本串为 `"aaaaa"`，模式串为 `"aaa"`。若朴素匹配统计所有出现位置，内层字符比较一共执行多少次？

可能起点为 $0, 1, 2$。

从 0 开始，比较 3 次，匹配成功。
从 1 开始，比较 3 次，匹配成功。
从 2 开始，比较 3 次，匹配成功。

总比较次数为 9。

若程序在第一次匹配成功后立即返回，则只比较 3 次。读程序时要看它是“找第一次出现”，还是“统计所有出现”。

C++ 中的 find

C++ 的 string 提供了 find 函数，用来查找子串第一次出现的位置。

```
string s = "banana";
int p = s.find("ana");
```

这里 p 的值为 1，因为 "ana" 第一次从下标 1 开始出现。

如果没有找到，find 返回 string::npos。不要直接把 npos 当作普通整数理解，它是一个很大的无符号值。判断是否找到时，应写：

```
if (s.find(t) != string::npos) {
    cout << "found" << endl;
}
```

若要继续查找下一次出现，可以指定起始位置：

```
int p = s.find(t, start);
```

这表示从下标 start 开始找。

例题 5：find 的返回值

执行下面代码：

```
string s = "ababcabc";
cout << s.find("abc") << endl;
```

输出是什么？

"abc" 第一次从下标 2 开始出现，所以输出 2。

若执行：

```
cout << s.find("zzz") << endl;
```

由于没有找到，返回的是 string::npos。不同环境中直接输出它可能显示成一个很大的数。程序中应使用 != string::npos 判断是否找到。

下标边界

字符串程序阅读中，边界是最常见错误来源。

设文本串长度为 n，模式串长度为 m。从下标 i 开始匹配时，会访问到 s[i + m - 1]。因此合法起点满足 i + m - 1 < n，也就是 i <= n - m。

所以外层循环常写成：

```
for (int i = 0; i <= n - m; i++)
```

或者：

```
for (int i = 0; i + m <= n; i++)
```

第二种写法在 $n < m$ 时更安全。因为若 `n` 和 `m` 是无符号类型，`n - m` 可能出现问题。CSP-S 第一轮代码中若使用 `int n = s.length()`，通常不会在这个细节上展开，但读程序时要知道两种写法的含义。

例题 6：最后一个起点

文本串长度为 10，模式串长度为 4。朴素匹配中，最后一个可能起点是多少？

最后起点 i 要满足 `i + 4 <= 10`，所以 $i \le 6$。最后一个可能起点是 6。

从下标 6 开始，会访问下标 $6, 7, 8, 9$，刚好到文本串末尾。若从下标 7 开始，要访问到下标 10，越界。

重叠匹配

字符串匹配中，重叠出现很常见。

例如文本串 `"aaaa"`，模式串 `"aa"`。出现位置为 $0, 1, 2$，共 3 次。位置 0 的匹配使用字符 $0, 1$；位置 1 的匹配使用字符 $1, 2$；它们共享了字符 1。

统计出现次数时，要看题目是否允许重叠。若允许，匹配成功后起点仍然只加 1；若不允许，匹配成功后可以跳过整个模式串长度。

例题 7：允许重叠与不允许重叠

文本串为 `"aaaaa"`，模式串为 `"aa"`。

若允许重叠，出现位置为 $0, 1, 2, 3$，答案为 4。

若不允许重叠，可以选择位置 0 和 2，答案为 2。如果从左到右贪心匹配，匹配 0..1 后，下一个起点从 2 开始。

CSP-S 第一轮若没有说明“不重叠”，通常按普通子串出现理解，即允许重叠。程序阅读题则要按代码移动方式判断。

双指针式子序列判断

虽然本章主要讲子串匹配，但程序阅读中也会出现“判断一个字符串是否为另一个字符串的子序列”的代码。它不是字符串匹配，却常被放在字符串题里混考。

例如：

```
bool sub(string a, string b) {
    int j = 0;

    for (int i = 0; i < b.size(); i++) {
        if (j < a.size() && b[i] == a[j]) {
            j++;
        }
    }

    return j == a.size();
}
```

这段代码判断 `a` 是否为 `b` 的子序列。指针 `i` 扫描长串 `b`，指针 `j` 指向短串 `a` 中等待匹配的位置。每当字符相等，`j` 向后移动。

它不要求连续，只要求顺序一致。所以它不能用于判断子串。

例题 8：子序列判断

`a="ace"`，`b="abcde"`。上面的程序返回什么？

扫描 b：

- 看到 a，匹配 a[0]，j 变为 1；
- 看到 c，匹配 a[1]，j 变为 2；
- 看到 e，匹配 a[2]，j 变为 3。

最后 j == a.size()，返回 true。所以 "ace" 是 "abcde" 的子序列。

但 "ace" 不是 "abcde" 的子串，因为它不连续。

真题精讲：带跳跃的字符串匹配

2022 年 CSP-S 第一轮阅读程序中出现过一个字符串匹配程序。它不是朴素匹配，也不是本书下一章要系统讲的 KMP，而是一种根据模式串字符预处理跳跃长度的匹配方法。

程序核心如下：

```
int f(const string &s, const string &t)
{
    int n = s.length(), m = t.length();

    vector<int> shift(128, m + 1);

    int i, j;

    for (j = 0; j < m; j++)
        shift[t[j]] = m - j;

    for (i = 0; i <= n - m; i += shift[s[i + m]]) {
        j = 0;
        while (j < m && s[i + j] == t[j]) j++;
        if (j == m) return i;
    }

    return -1;
}
```

这段程序的目标仍然是寻找模式串 t 在文本串 s 中第一次出现的位置。它返回匹配起点，若找不到则返回 −1。

先看预处理：

```
vector<int> shift(128, m + 1);
for (j = 0; j < m; j++)
    shift[t[j]] = m - j;
```

shift[c] 表示当某次从起点 i 尝试匹配失败后，根据 s[i+m] 这个字符决定下一次起点向后移动多少。初始为 m+1，表示如果这个字符没有在模式串中出现，就可以跳过更长距离。若字符出现在模式串中，则按它在模式串中的位置设置移动量。

再看匹配：

```
while (j < m && s[i + j] == t[j]) j++;
if (j == m) return i;
```

这仍然是从当前起点 i 开始逐字符比较。若 j 最终达到 m，说明 t[0..m-1] 全部匹配，返回当前起点 i。

最后看外层循环：

```
for (i = 0; i <= n - m; i += shift[s[i + m]])
```

它不是每次 `i++`，而是根据当前窗口后面一个字符 `s[i+m]` 来跳跃。这里要求访问 `s[i+m]`，所以题目保证输入字符串范围不会让程序访问非法位置，或者测试数据避开相关越界风险。阅读这类程序时，先按题目说明处理，不要只凭常规模板改代码。

例题 9：预处理 shift

模式串 `t="abab"`，长度 $m = 4$。预处理后，`shift['a']` 和 `shift['b']` 分别是多少？

初始所有字符的 `shift` 都是 $m + 1 = 5$。

循环处理模式串：

- `j=0`，`t[0]='a'`，令 `shift['a']=4`；
- `j=1`，`t[1]='b'`，令 `shift['b']=3`；
- `j=2`，`t[2]='a'`，令 `shift['a']=2`，覆盖之前的 4；
- `j=3`，`t[3]='b'`，令 `shift['b']=1`，覆盖之前的 3。

最终 `shift['a']=2`，`shift['b']=1`。

同一个字符多次出现时，后面的位置会覆盖前面的位置。最终保留的是这个字符在模式串中最靠右位置对应的移动量。

例题 10：真题式追踪

输入为：

```
abcde fg
```

文本串 `s="abcde"`，模式串 `t="fg"`。程序输出什么？

模式串长度 $m = 2$。从起点 0 开始比较：

- `s[0]='a'`，`t[0]='f'`，第一位就不匹配。
- 下一次起点移动量由 `s[i+m]=s[2]='c'` 决定。字符 `c` 不在模式串 `"fg"` 中，所以 `shift['c']=3`，起点从 0 跳到 3。

此时 `i=3`，但循环条件是 `i <= n - m`，即 `i <= 3`，还会尝试一次。

从起点 3 开始，`s[3]='d'`，`t[0]='f'`，不匹配。下一次移动后越过范围，循环结束。

没有找到模式串，所以返回 -1。

这道题的关键是跟着程序的 `i += shift[s[i + m]]` 走，而不是按朴素匹配的 `i++` 走。

读字符串匹配程序的顺序

字符串匹配程序通常不长，但变量很多。读程序时可以按下面顺序。

第一，看两个字符串谁是文本串，谁是模式串。通常长串叫 `s`，短串叫 `t`，但不一定。要看循环中哪个字符串被枚举起点，哪个字符串被逐字符比较。

第二，看下标从哪里开始。C++ `string` 通常从 0 开始。若题目输出位置，有时会再加 1。

第三，看外层循环的边界。`i + m <= n` 表示从每个合法起点尝试匹配；`i <= n - m` 是等价写法。若外层不是 `i++`，要特别注意移动规则。

第四，看匹配成功的判断。`j == m` 表示模式串全部匹配。若程序返回 `i`，说明返回起点；若返回 `i + 1`，说明转换成从 1 开始的位置。

第五，看失败后如何移动。朴素匹配通常外层加 1；KMP 会移动模式串指针；跳跃式匹配可能根据某个字符查表移动。

第六，看是否统计多个匹配。若匹配成功后立即 `return`，它只找第一次出现；若用 `ans++` 后继续循环，才是在统计出现次数。

完善程序中的常见空缺

字符串匹配完善程序常在边界、比较、返回值和指针移动处留空。

朴素匹配中常见空缺如下：

空缺位置	应关注的问题
外层循环条件	`i + m <= n`
内层循环条件	`j < m`
字符比较	`s[i + j] == t[j]`
匹配成功	`j == m`
返回位置	返回 `i` 或 `i + 1`
匹配失败	跳出内层，尝试下一个起点

子序列判断中常见空缺如下：

空缺位置	应关注的问题
扫描长串	外层遍历长串
当前目标字符	`a[j]`
匹配后移动	`j++`
成功条件	`j == a.size()`

跳跃式匹配中常见空缺如下：

空缺位置	应关注的问题
初始跳跃值	通常和模式串长度有关
预处理字符	根据模式串字符设置
当前窗口比较	仍然逐字符比较
下一次起点	根据某个查表值移动

例题 11：补全朴素匹配

下面代码找模式串 `t` 在文本串 `s` 中第一次出现的位置：

```
for (int i = 0; ________; i++) {
    int j = 0;

    while (j < m && ________) {
        j++;
    }

    if (j == m) {
        return i;
    }
}
```

第一个空应填 `i + m <= n`。第二个空应填 `s[i + j] == t[j]`。

完整含义是：只枚举合法起点；从当前起点开始，逐字符比较文本串和模式串。

若第一个空写成 `i < n`，后面可能访问越界。若第二个空写成 `s[j] == t[j]`，就只会一直从文本串开头比较，忽略当前起点 `i`。

例题 12：补全子序列判断

下面代码判断 `a` 是否为 `b` 的子序列：

```
int j = 0;

for (int i = 0; i < b.size(); i++) {
    if (j < a.size() && b[i] == a[j]) {
        ________;
    }
}

return ________;
```

第一个空应填 `j++`。第二个空应填 `j == a.size()`。

每匹配到短串的一个字符，就把 `j` 向后移动。最后若 `j` 到达短串末尾，说明短串所有字符都按顺序匹配到了。

常见错误

第一，把子串和子序列混淆。字符串匹配通常要求连续；子序列只要求顺序。

第二，外层循环边界写错。长度为 n 的文本串中，长度为 m 的模式串最后起点是 $n-m$。

第三，忽略重叠匹配。统计出现次数时，匹配成功后是否跳过，要看题目要求和代码移动方式。

第四，把 `find` 返回的 `string::npos` 当普通小整数处理。判断是否找到时应和 `string::npos` 比较。

第五，忘记 C++ 字符串下标从 0 开始。题目若要求从 1 开始输出，要额外加 1。

第六，没有分清程序是在找第一次出现，还是统计所有出现。看到 `return i` 时，通常只找第一次。

第七，看到跳跃式匹配程序时，仍按朴素匹配逐个起点追踪。外层 `i` 如何变化，要按代码执行。

第八，误以为字符串越长就一定是文本串。变量名和读入顺序可能误导，要看程序具体怎样比较。

小结

字符串匹配的基本任务，是在文本串中寻找模式串的连续出现位置。朴素匹配枚举每个合法起点，再逐字符比较，最坏时间复杂度为 $O(nm)$。它适合讲清下标、边界和匹配过程，也常出现在第一轮程序阅读中。

本章需要重点记住：

- 子串要求连续，子序列不要求连续；
- C++ `string` 下标从 0 开始；
- 长度为 m 的模式串在长度为 n 的文本串中，合法起点满足 `i + m <= n`；
- 朴素匹配中比较的是 `s[i + j]` 和 `t[j]`；
- `j == m` 表示模式串完全匹配；
- 统计出现次数时，要注意是否允许重叠；
- `string::find` 找不到时返回 `string::npos`；
- 程序阅读中要看失败后起点或指针怎样移动。

下一章进入 KMP。KMP 仍然解决字符串匹配问题，但它会利用模式串自身的前后缀信息，在失配后避免从头重复比较。

0x23 KMP

第 34 章讲了普通字符串匹配：枚举文本串中的每个起点，再逐字符比较模式串。朴素做法很好理解，但遇到大量重复字符时，会反复比较已经比较过的部分。

KMP 也是做字符串匹配的算法。它的关键不是“比较得更快”，而是失配后不把模式串完全退回开头。它利用模式串自身的前后缀信息，判断下一次应该从哪里继续比较。

CSP-S 第一轮中，KMP 通常不会要求写一份很长的模板，但会考三个地方：前缀函数或 `next` 数组怎么算，失配时指针怎样跳，匹配成功后位置怎样还原。本章只讲第一轮需要读懂和手算的部分，不把 KMP 扩展到自动机、最小循环节等更深内容。

问题

给定文本串 `s` 和模式串 `p`，要求找出 `p` 在 `s` 中出现的位置。

例如 `s="ababcabab"`，`p="abab"`。模式串从文本串下标 0 开始出现一次，也从下标 5 开始出现一次。

朴素匹配从每个起点开始比较。若从某个起点已经比较了很多字符，最后一个字符失配，朴素做法会把起点后移 1，再从模式串开头重新比较。KMP 的改进就在这里：既然前面已经知道一段字符匹配成功，就应该利用这些信息，不必把所有内容重新比较一遍。

前缀、后缀和 border

讲 KMP 前，要先分清前缀和后缀。

一个字符串的前缀，是从开头开始的一段。字符串 `"ababa"` 的前缀有 `"a"`、`"ab"`、`"aba"`、`"abab"`、`"ababa"`。

一个字符串的后缀，是到结尾结束的一段。字符串 `"ababa"` 的后缀有 `"a"`、`"ba"`、`"aba"`、`"baba"`、`"ababa"`。

在 KMP 中，通常关心“真前缀”和“真后缀”。真前缀不能等于整个字符串，真后缀也不能等于整个字符串。

如果一个字符串的某个真前缀等于某个真后缀，这段相同的字符串常叫 border。

例如 `"ababa"` 的真前缀有 `"a"`、`"ab"`、`"aba"`、`"abab"`；真后缀有 `"a"`、`"ba"`、`"aba"`、`"baba"`。相同的有 `"a"` 和 `"aba"`，其中最长的是 `"aba"`，长度为 3。

例题 1：找最长相等真前后缀

求字符串 `"abcab"` 的最长相等真前后缀长度。

它的真前缀有 `"a"`、`"ab"`、`"abc"`、`"abca"`。真后缀有 `"b"`、`"ab"`、`"cab"`、`"bcab"`。

相同的是 `"ab"`，长度为 2。所以答案是 2。

求这类长度时，不要把整个字符串本身算进去。整个字符串既是前缀也是后缀，但不是真前后缀。

前缀函数

前缀函数常记作 `pi`。对于字符串 `p`，`pi[i]` 表示子串 `p[0..i]` 的最长相等真前后缀长度。

注意 `pi[i]` 只看从 0 到 i 这一段前缀，不看后面的字符。

例如 p="ababaca"。逐项计算：

i	p[0..i]	最长相等真前后缀	pi[i]
0	a	空	0
1	ab	空	0
2	aba	a	1
3	abab	ab	2
4	ababa	aba	3
5	ababac	空	0
6	ababaca	a	1

所以 pi 数组为 0 0 1 2 3 0 1。

例题 2：手算 pi

求 p="aaaa" 的前缀函数。

p[0..0]="a"，没有非空真前后缀，pi[0]=0。

p[0..1]="aa"，最长相等真前后缀是 "a"，pi[1]=1。

p[0..2]="aaa"，最长相等真前后缀是 "aa"，pi[2]=2。

p[0..3]="aaaa"，最长相等真前后缀是 "aaa"，pi[3]=3。

所以 pi 为 0 1 2 3。

这类全相同字符串很适合检查前缀函数是否理解正确。

前缀函数怎么递推

假设已经算出 pi[i-1]，现在要计算 pi[i]。设 j=pi[i-1]，表示当前已经知道 p[0..i-1] 的最长相等真前后缀长度为 j。

若 p[i] == p[j]，说明原来的相等前后缀可以再向后扩展一个字符，于是 pi[i]=j+1。

若 p[i] != p[j]，说明长度为 j 的方案不能继续扩展。此时不需要从头试所有长度，而是把 j 退到 pi[j-1]。因为长度为 j 的前缀本身，也有自己的最长相等真前后缀。继续尝试这个更短的长度。

代码如下：

```
vector<int> getpi(string p) {
    int m = p.size();
    vector<int> pi(m);

    for (int i = 1; i < m; i++) {
        int j = pi[i - 1];

        while (j > 0 && p[i] != p[j]) {
            j = pi[j - 1];
        }

        if (p[i] == p[j]) {
            j++;
        }

        pi[i] = j;
    }
```

```
    return pi;
}
```

这里最关键的是失配时的语句 `j = pi[j - 1]`。它不是把 `j` 直接变成 0，而是退到下一个可能继续匹配的位置。

例题 3：递推一项

已知 `p="ababac"`，前面已经算到 `pi[4]=3`，现在计算 `pi[5]`。

当前 `i=5`，字符 `p[5]='c'`。先令 `j=pi[4]=3`。比较 `p[5]` 和 `p[3]`，即比较 `c` 和 `b`，失配。

于是令 `j=pi[2]=1`。再比较 `p[5]` 和 `p[1]`，即比较 `c` 和 `b`，仍然失配。

于是令 `j=pi[0]=0`。此时 `j=0`，再比较 `p[5]` 和 `p[0]`，即比较 `c` 和 `a`，仍然不相等，所以不能扩展。

因此 `pi[5]=0`。

这一步说明，KMP 的退回不是随便退，而是沿着前缀函数一层一层退。

next 数组

有些教材和程序不用 `pi`，而用 `next` 或 `nxt`。它们的含义和下标写法不完全统一。

本书在正文中优先使用 `pi`，因为它的定义清楚：`pi[i]` 表示 `p[0..i]` 的最长相等真前后缀长度。

但在程序阅读题中，经常看到下面这种从 1 开始的写法：

```
for (int i = 2, j = 0; i <= m; i++) {
    while (j && p[i] != p[j + 1]) {
        j = nxt[j];
    }

    if (p[i] == p[j + 1]) {
        j++;
    }

    nxt[i] = j;
}
```

这里字符串从下标 1 开始存。`nxt[i]` 表示 `p[1..i]` 的最长相等真前后缀长度。它和 0 开始写法中的 `pi[i-1]` 对应。

写法	字符串下标	含义
`pi[i]`	从 0 开始	`p[0..i]` 的最长相等真前后缀长度
`nxt[i]`	从 1 开始	`p[1..i]` 的最长相等真前后缀长度

读程序时不要只看数组名，要看它的下标从哪里开始，以及失配时跳到哪里。

例题 4：pi 与 nxt 对应

若模式串为 `"ababaca"`，0 开始写法的 `pi` 为 `0 0 1 2 3 0 1`。若改成 1 开始存储，`nxt[1..7]` 是多少？

它们一一对应：`nxt[1]=pi[0]`，`nxt[2]=pi[1]`，一直到 `nxt[7]=pi[6]`。

所以 `nxt[1..7]` 为 `0 0 1 2 3 0 1`。

数值相同，不代表所有代码都能直接替换。因为 1 开始写法比较字符时常用 `p[j+1]`，0 开始写法常用 `p[j]`。

KMP 匹配过程

有了 pi 后，就可以匹配文本串 s 和模式串 p。

设 j 表示当前已经匹配了模式串的前 j 个字符。扫描文本串的每个字符 s[i]。

若 s[i] == p[j]，说明可以继续匹配，令 j++。

若不相等，就让 j 按 pi 回退：j = pi[j - 1]，直到能匹配，或者退到 0。

当 j == m 时，说明模式串完整匹配一次，出现位置是 i - m + 1。

代码如下：

```
vector<int> kmp(string s, string p) {
    vector<int> pi = getpi(p);
    vector<int> pos;

    int n = s.size();
    int m = p.size();
    int j = 0;

    for (int i = 0; i < n; i++) {
        while (j > 0 && s[i] != p[j]) {
            j = pi[j - 1];
        }

        if (s[i] == p[j]) {
            j++;
        }

        if (j == m) {
            pos.push_back(i - m + 1);
            j = pi[j - 1];
        }
    }

    return pos;
}
```

匹配成功后，令 j = pi[j - 1]，是为了继续寻找下一个可能出现位置，并允许重叠匹配。

例题 5：匹配位置

文本串 s="ababcabab"，模式串 p="abab"。用 KMP 找出现位置。

模式串 "abab" 的 pi 为 0 0 1 2。

扫描文本串时，前四个字符 a b a b 完整匹配，第一次出现位置为 0。匹配成功后，j 回到 pi[3]=2，表示后缀 "ab" 仍可以作为下一次匹配的开头。

继续扫描后面的字符，最终在下标 5 到 8 再次匹配 "abab"。所以出现位置为 0 和 5。

如果匹配成功后直接把 j 设为 0，有时会漏掉重叠匹配。KMP 中成功后的回退也要按 pi 来做。

失配时为什么不回退 i

朴素匹配失配后，会移动起点，文本串下标也可能回到较早位置重新比较。KMP 中，文本串指针 i 始终向右扫描，不会回退。回退的是模式串指针 j。

原因是：`s[i-j..i-1]` 已经和 `p[0..j-1]` 匹配。失配发生在 `s[i]` 和 `p[j]`。这时已经匹配的那段文本串内容等于模式串的前缀。下一次可能匹配的位置，必须让模式串的某个前缀对齐到这段文本的后缀上。

前缀函数正好告诉我们：当前已匹配长度 `j` 时，最长可以保留多少长度继续尝试。于是只需要把 `j` 改成 `pi[j-1]`，文本串当前字符 `s[i]` 还可以继续和新的 `p[j]` 比较。

这就是 KMP 避免重复比较的地方。

例题 6：失配回退

文本串已经匹配到某一刻，模式串 `p="ababaca"`，当前已经匹配了前 5 个字符 `"ababa"`，下一位失配。此时 `j=5`，应该退到哪里？

看 `pi[4]`。前面算过，`p[0..4]="ababa"` 的最长相等真前后缀是 `"aba"`，长度为 3，所以 `pi[4]=3`。

失配后，`j` 从 5 退到 3。这表示已经匹配的文本末尾 `"aba"` 可以继续作为模式串开头 `"aba"` 使用，不需要重新比较它们。

复杂度

KMP 的预处理用 $O(m)$ 时间计算模式串的 `pi` 数组。匹配时，文本串指针 `i` 从左到右扫描，每个字符只作为当前文本字符处理一次；模式串指针 `j` 虽然会前进和回退，但总的前进、回退次数受字符串长度限制。因此匹配时间为 $O(n)$。

总时间复杂度为 $O(n+m)$，额外空间为 $O(m)$。

第一轮中，常见选项会把 KMP 和朴素匹配混在一起。朴素匹配最坏为 $O(nm)$，KMP 为 $O(n+m)$。若题目只问“查找一个模式串在文本串中的出现位置”，且使用了前缀函数或 `next` 数组，通常应判断为 KMP 结构。

1 开始写法

很多竞赛代码喜欢把字符串从 1 开始存。下面是 1 开始写法的 KMP。

```
char s[N], p[N];
int nxt[N];

void getnxt(int m) {
    for (int i = 2, j = 0; i <= m; i++) {
        while (j && p[i] != p[j + 1]) {
            j = nxt[j];
        }

        if (p[i] == p[j + 1]) {
            j++;
        }

        nxt[i] = j;
    }
}

int match(int n, int m) {
    getnxt(m);

    for (int i = 1, j = 0; i <= n; i++) {
        while (j && s[i] != p[j + 1]) {
            j = nxt[j];
        }
```

```
        if (s[i] == p[j + 1]) {
            j++;
        }

        if (j == m) {
            return i - m + 1;
        }
    }

    return -1;
}
```

这段代码返回第一次出现位置，且位置从 1 开始。因为当前匹配到文本串第 `i` 个字符，模式串长度为 `m`，起点为 `i - m + 1`。

若要继续统计所有出现次数，匹配成功后不能直接 `return`，而要计数，并写 `j = nxt[j]` 继续寻找下一次出现。

例题 7：返回位置

文本串从 1 开始存，模式串长度为 4。若匹配成功时 `i=9`，出现位置是多少？

出现位置是 `i - m + 1 = 9 - 4 + 1 = 6`。

也就是说，模式串对应文本串下标 $6, 7, 8, 9$。

如果是 0 开始写法，匹配成功时当前位置为 `i`，起点同样写作 `i - m + 1`，但数值代表 0 开始下标。

匹配成功后的处理

KMP 中，匹配成功后如何处理，取决于题目要求。

如果只找第一次出现，匹配成功后可以直接返回起点。

如果统计所有出现次数，匹配成功后应先记录答案，再令 `j = pi[j - 1]` 或 `j = nxt[j]`，继续匹配。

以模式串 `"aa"`、文本串 `"aaaa"` 为例。出现位置为 $0, 1, 2$。若匹配成功后把 `j` 直接清零，仍可能找到某些位置，但在更多模式串中会漏掉重叠出现。按前缀函数回退才是 KMP 的标准处理方式。

例题 8：重叠出现

文本串 `s="aaaa"`，模式串 `p="aa"`。统计所有出现次数。

模式串 `p` 的 `pi` 为 `0 1`。

扫描前两个字符时，第一次匹配成功，出现位置为 0。此时 `j=2`，回退到 `pi[1]=1`，保留一个已经匹配的 `a`。

继续读下一个 `a`，又能形成一次完整匹配，位置为 1。再次回退到 1。

继续读最后一个 `a`，得到第三次匹配，位置为 2。

所以总次数为 3。

拼接字符串写法

还有一种常见写法：把模式串、分隔符和文本串拼接起来，计算整个串的前缀函数。

例如构造 `p + "#" + s`。其中 `#` 是一个不会出现在原字符串中的分隔符。若某个位置的前缀函数值等于 `p.size()`，说明模式串在文本串中出现了一次。

代码示意如下：

```cpp
vector<int> findpos(string s, string p) {
    string t = p + "#" + s;
    vector<int> pi = getpi(t);
    vector<int> ans;
    int m = p.size();

    for (int i = m + 1; i < t.size(); i++) {
        if (pi[i] == m) {
            ans.push_back(i - 2 * m);
        }
    }

    return ans;
}
```

这里位置换算容易出错。拼接串中，文本串从下标 `m + 1` 开始。如果在拼接串位置 `i` 处匹配结束，则文本串中的结束下标是 `i - (m + 1)`，起点是 `i - (m + 1) - m + 1`，化简为 `i - 2 * m`。

这种写法适合统一使用前缀函数，但程序阅读中要特别小心分隔符和下标换算。

例题 9：拼接串中的位置

模式串 `p="aba"`，文本串 `s="ababa"`，拼接串为 `"aba#ababa"`。若在拼接串下标 6 处有 `pi[6]=3`，对应文本串中的出现起点是多少？

模式串长度 $m = 3$。按公式，文本串中的起点为 `i - 2 * m = 6 - 6 = 0`。

这对应文本串 `s[0..2]="aba"`。

若在拼接串下标 8 处有 `pi[8]=3`，起点为 `8 - 6 = 2`，对应 `s[2..4]="aba"`。

next 的不同版本

KMP 的 `next` 数组有多种写法。有些代码让 `next[0] = -1`，有些代码让 `nxt[1] = 0`。它们不一定能混用。

例如某些教材写法如下：

```cpp
int i = 0, j = -1;
nxt[0] = -1;

while (i < m) {
    if (j == -1 || p[i] == p[j]) {
        i++;
        j++;
        nxt[i] = j;
    } else {
        j = nxt[j];
    }
}
```

这种写法中，`j=-1` 是一个特殊状态，表示已经退无可退。它和本章前面讲的 `pi` 写法不同。

第一轮程序阅读中，遇到 KMP 不要只背某一份模板。要先看三件事：

- 字符串下标从 0 还是 1 开始；
- 失配时写的是 `j = pi[j - 1]`、`j = nxt[j]`，还是 `j = nxt[j - 1]`；
- 比较字符时写的是 `p[j]` 还是 `p[j + 1]`。

只要这三点看清楚，数组含义就能确定。

真题式程序阅读：数组含义

下面代码计算模式串的 `nxt` 数组，字符串从 1 开始：

```
for (int i = 2, j = 0; i <= m; i++) {
    while (j && p[i] != p[j + 1]) {
        j = nxt[j];
    }

    if (p[i] == p[j + 1]) {
        j++;
    }

    nxt[i] = j;
}
```

读这段代码时，要注意 `j` 表示当前已匹配的前缀长度，而不是某个固定下标。比较 `p[i]` 和 `p[j+1]`，是因为若已有长度为 `j` 的相等前后缀，现在尝试把它扩展到长度 `j+1`。

失配时，`j = nxt[j]`，表示把当前候选长度退到更短的 border 长度。循环结束后，如果 `p[i] == p[j+1]`，就能扩展一位。

例题 10：追踪 nxt

模式串从 1 开始存，`p="abab"`。求 `nxt[1..4]`。

`nxt[1]=0`。

计算 `i=2`，比较 `p[2]='b'` 和 `p[1]='a'`，不相等，所以 `nxt[2]=0`。

计算 `i=3`，比较 `p[3]='a'` 和 `p[1]='a'`，相等，`j` 变成 1，所以 `nxt[3]=1`。

计算 `i=4`，当前 `j=1`，比较 `p[4]='b'` 和 `p[2]='b'`，相等，`j` 变成 2，所以 `nxt[4]=2`。

最终 `nxt` 为 `0 0 1 2`。

KMP 与周期

KMP 的前缀函数还可以帮助判断字符串周期。虽然这不是本章主线，但第一轮选择题可能出现简单结论。

设字符串长度为 n，最长相等真前后缀长度为 `pi[n-1]`。令 `k = n - pi[n-1]`。如果 n 能被 k 整除，那么这个字符串可以由长度为 k 的循环节重复得到。

例如 `"ababab"` 长度为 6，最长相等真前后缀是 `"abab"`，长度为 4，所以 $k = 6 - 4 = 2$。因为 6 能被 2 整除，循环节长度为 2，即 `"ab"`。

如果不能整除，就不能由这个长度的循环节完整重复得到。

例题 11：判断循环节

字符串 `"abcabcabc"` 的长度为 9。它的最长相等真前后缀是 `"abcabc"`，长度为 6。最短循环节长度是多少？

计算 $k = 9 - 6 = 3$。因为 9 能被 3 整除，所以最短循环节长度为 3，循环节是 `"abc"`。

字符串 `"abababa"` 长度为 7，最长相等真前后缀长度为 5，得到 $k = 2$。但 7 不能被 2 整除，所以不能说它由长度为 2 的循环节完整重复得到。

KMP 程序阅读顺序

KMP 程序阅读题可以按下面顺序处理。

第一，看字符串下标。`string` 多为 0 开始，字符数组可能从 1 开始。下标决定比较的是 `p[j]` 还是 `p[j+1]`。

第二，看 `pi` 或 `nxt` 的含义。若 `pi[i]` 对应 `p[0..i]`，这是 0 开始前缀函数；若 `nxt[i]` 对应 `p[1..i]`，这是 1 开始写法。

第三，看失配时跳到哪里。`j = pi[j-1]` 或 `j = nxt[j]` 都表示退到更短的 border，但不能随便替换。

第四，看匹配成功条件。0 开始写法常见 `j == m`；1 开始写法也常见 `j == m`，但当前位置换算要看 `i` 的含义。

第五，看匹配成功后的动作。直接返回表示只找第一次出现；计数后回退表示统计所有出现。

第六，看输出位置。0 开始返回 `i - m + 1`，1 开始也可能返回 `i - m + 1`，但含义分别是 0 开始位置和 1 开始位置。

完善程序中的常见空缺

KMP 完善程序常在前缀函数、失配跳转、匹配成功判断和答案位置处留空。

前缀函数部分常见空缺如下：

空缺位置	应关注的问题
初始 `j`	通常为前一项 `pi[i-1]` 或当前匹配长度
失配回退	`j = pi[j - 1]` 或 `j = nxt[j]`
扩展条件	当前字符是否等于候选前缀下一字符
记录结果	`pi[i] = j` 或 `nxt[i] = j`

匹配部分常见空缺如下：

空缺位置	应关注的问题
文本串失配	模式串指针按 `pi` 或 `nxt` 回退
匹配成功	`j++`
完整匹配	`j == m`
出现位置	`i - m + 1`
继续匹配	成功后 `j = pi[j - 1]` 或 `j = nxt[j]`

例题 12：补全前缀函数

下面是 0 开始写法的前缀函数：

```
for (int i = 1; i < m; i++) {
    int j = pi[i - 1];

    while (j > 0 && p[i] != p[j]) {
        j = ________;
    }

    if (p[i] == p[j]) {
        j++;
    }

    pi[i] = j;
}
```

空缺处应填 `pi[j - 1]`。

当前候选长度为 j。若无法扩展，就退到长度 j 这段前缀自身的最长相等真前后缀长度，也就是 pi[j-1]。

例题 13：补全匹配

下面代码用 KMP 匹配文本串 s 和模式串 p：

```
for (int i = 0; i < n; i++) {
    while (j > 0 && s[i] != p[j]) {
        j = pi[j - 1];
    }

    if (s[i] == p[j]) {
        ________;
    }

    if (j == m) {
        ans++;
        ________;
    }
}
```

第一个空应填 j++。第二个空应填 j = pi[j - 1]。

匹配一个字符后，模式串指针向后移动。完整匹配后，如果还要继续统计出现次数，就按前缀函数回退，保留可能参与下一次重叠匹配的部分。

常见错误

第一，把 pi[i] 理解成整个模式串的前后缀长度。pi[i] 只对应 p[0..i]。

第二，失配时直接把 j 设为 0。这样仍可能得到正确答案，但会失去 KMP 的效率，也可能破坏某些统计逻辑。

第三，混用不同版本的 next。next[0] = -1 和 nxt[1] = 0 是不同写法，不能只改数组名。

第四，匹配成功后忘记回退。若要统计所有出现，成功后要让 j 回到合适的 border 长度，否则可能漏掉重叠匹配。

第五，返回位置差 1。要看字符串下标从 0 还是从 1 开始，也要看题目要求输出哪种编号。

第六，把最长相等真前后缀和最长回文前后缀混淆。KMP 中前后缀要求字符顺序完全相同，不是反过来相同。

第七，拼接串写法中忘记分隔符。若直接拼 p+s，可能让模式串尾部和文本串开头产生错误的前后缀关系。

第八，分隔符出现在原字符串中。分隔符必须选一个不会出现在 p 和 s 中的字符。

小结

KMP 解决的是字符串匹配问题。它和朴素匹配一样逐字符比较，但失配后不会把文本串指针退回去，而是根据模式串的前后缀信息移动模式串指针。

前缀函数 pi[i] 表示 p[0..i] 的最长相等真前后缀长度。计算 pi 时，若当前候选长度不能扩展，就沿 pi[j-1] 回退。匹配时，若 s[i] 和 p[j] 失配，也沿 pi[j-1] 回退。

本章需要重点记住：

- 前缀从开头开始，后缀到结尾结束；
- pi[i] 只看 p[0..i]；
- 失配时回退的是模式串指针 j，不是文本串指针 i；
- 匹配成功位置为 i - m + 1；

- 统计所有出现时，完整匹配后要继续按 `pi` 回退；
- `pi`、`nxt` 有不同下标版本，读程序时先看定义；
- KMP 总复杂度为 $O(n+m)$；
- KMP 也能辅助判断简单循环节，但本章重点仍是字符串匹配。

下一章进入字符串哈希。它会换一种思路：不再逐字符比较两个子串，而是把子串映射成数值，用数值快速判断两个子串是否相同。

0x24 字符串哈希

第 35 章讲了 KMP。KMP 的思路是利用模式串的前后缀信息，在失配后减少重复比较。本章换一种方法：给字符串计算一个数值，用这个数值来快速判断两个字符串是否相同。

字符串哈希并不是把字符串变成一个绝对可靠的编号。不同字符串可能得到相同哈希值，这叫冲突。CSP-S 第一轮中，更常见的考法不是追问冲突概率，而是考查：哈希值如何递推，子串哈希如何由前缀哈希得到，为什么比较哈希比逐字符比较更快，程序中下标和幂次是否对应。

哈希的想法

把字符串看成一个数，是字符串哈希最常见的想法。

例如把字符映射成数字：`a` 映射为 1，`b` 映射为 2，`c` 映射为 3。字符串 `"abc"` 可以看成按某个底数 B 展开的数，它的哈希可以写成 $1 \cdot B^2 + 2 \cdot B + 3$。

为了防止数字过大，程序通常会对一个大模数取模。于是哈希值变成：

```
h = (h * B + value) % mod;
```

从左到右扫描字符串，每加入一个字符，就把原来的哈希乘以底数，再加上当前字符值。

例如底数 $B = 10$，字符值仍为 `a=1,b=2,c=3`。扫描 `"abc"` 时，扫到 `a` 后哈希为 1；扫到 `b` 后哈希为 $1 \times 10 + 2 = 12$；扫到 `c` 后哈希为 $12 \times 10 + 3 = 123$。

实际程序中，底数不会取 10 这么小，这里只是为了说明递推过程。

例题 1：计算简单哈希

设底数 $B = 10$，字符 `a,b,c,d` 分别映射为 $1, 2, 3, 4$。不取模时，字符串 `"bad"` 的哈希是多少？

从左到右递推。扫到 `b` 后，哈希为 2；扫到 `a` 后，哈希为 $2 \times 10 + 1 = 21$；扫到 `d` 后，哈希为 $21 \times 10 + 4 = 214$。

所以 `"bad"` 的哈希为 214。

这个例子里，哈希值和十进制数字很像。正式程序中底数和模数会更大，但递推形式一样。

前缀哈希

如果每次比较两个子串都重新计算哈希，仍然会重复扫描字符。为了快速取子串哈希，需要预处理前缀哈希。

设字符串从 1 开始编号，`h[i]` 表示前 i 个字符的哈希值。递推式为 `h[i] = h[i - 1] * B + val(s[i])`。同时预处理 $pw[i] = B^i$。

这里的 `pw[i]` 是幂数组，表示 B^i。例如 $pw[0] = 1$，$pw[1] = B$，$pw[2] = B^2$。

若字符串为 `s[1..n]`，前缀哈希满足：`h[0]=0`，`h[1]` 是 `s[1]` 的哈希，`h[2]` 是 `s[1..2]` 的哈希，`h[i]` 是 `s[1..i]` 的哈希。

这样，任意子串 `s[l..r]` 的哈希都可以由前缀哈希快速得到。

子串哈希公式

设使用从左到右的哈希方式。`h[r]` 表示 `s[1..r]` 的哈希，`h[l-1]` 表示 `s[1..l-1]` 的哈希。

要取出 `s[l..r]` 的哈希，需要从 `h[r]` 中去掉前面 `s[1..l-1]` 的贡献。由于 `s[1..l-1]` 在 `h[r]` 中被整体向左推了 $r-l+1$ 位，所以要乘上 B^{r-l+1} 再相减。

子串哈希公式为 $hash(l,r)=h[r]-h[l-1]\times B^{r-l+1}$。如果使用取模，还要处理减法后的负数。

代码常写成：

```
long long get(int l, int r) {
    return (h[r] - h[l - 1] * pw[r - l + 1] % mod + mod) % mod;
}
```

其中 `pw[r - l + 1]` 表示 B^{r-l+1}，`+ mod` 是为了防止减法后出现负数。

例题 2：理解子串公式

设底数 $B=10$，不取模，字符串 `"abcd"` 的字符值为 $1,2,3,4$。

前缀	哈希
`"a"`	1
`"ab"`	12
`"abc"`	123
`"abcd"`	1234

现在要取子串 `"bc"`，也就是 `s[2..3]`。按公式，`h[3]=123`，`h[1]=1`，子串长度为 2，所以子串哈希为 $123-1\times10^2=23$。

这正好对应 `"bc"` 的数字表示 23。

比较两个子串

有了 `get(l,r)`，就可以快速比较两个长度相同的子串。

若 `get(l1,r1) == get(l2,r2)`，通常认为两个子串相同；若哈希不同，两个子串一定不同。

注意这句话有方向：

- 两个字符串相同，它们的哈希一定相同；
- 两个字符串哈希相同，它们不一定相同，因为可能发生冲突。

在竞赛程序中，通常通过选择合适的底数和模数，或者使用双哈希，降低冲突影响。CSP-S 第一轮更关注程序逻辑，一般不会要求学生构造冲突。

例题 3：比较子串

字符串 `s="abcab"`。判断子串 `s[1..2]` 和 `s[4..5]` 是否相同。这里下标从 1 开始。

`s[1..2]="ab"`，`s[4..5]="ab"`。两个子串相同，所以它们的哈希应相同。

若比较 `s[2..3]` 和 `s[4..5]`，分别是 `"bc"` 和 `"ab"`，哈希不同，子串不同。

从 0 开始的写法

C++ `string` 的下标从 0 开始。为了让子串公式更方便，常把前缀哈希数组 `h` 设计成长度 $n+1$，其中 `h[i]` 表示前 i 个字符的哈希，也就是 `s[0..i-1]` 的哈希。

代码如下：

```
const long long B = 131;
const long long mod = 1000000007;

long long h[N], pw[N];

void build(string s) {
    int n = s.size();

    h[0] = 0;
    pw[0] = 1;

    for (int i = 0; i < n; i++) {
        h[i + 1] = (h[i] * B + s[i]) % mod;
        pw[i + 1] = pw[i] * B % mod;
    }
}

long long get(int l, int r) {
    return (h[r + 1] - h[l] * pw[r - l + 1] % mod + mod) % mod;
}
```

这里 `get(l,r)` 取的是 `s[l..r]`，下标从 0 开始。因为 `h[r+1]` 表示前 $r+1$ 个字符的哈希，正好包含到 `s[r]`。

例题 4：0 开始子串

字符串 `s="abcdef"`，下标从 0 开始。要取子串 `"cde"`，应调用 `get(l,r)` 中的哪两个参数？

`"cde"` 对应下标 $2,3,4$，所以应调用 `get(2,4)`。

公式中会使用 `h[5]` 和 `h[2]`，因为 `h[5]` 表示前 5 个字符 `"abcde"` 的哈希，`h[2]` 表示前 2 个字符 `"ab"` 的哈希。相减后得到中间 `"cde"` 的哈希。

1 开始的写法

有些竞赛代码喜欢把字符串从 1 开始存。此时 `h[i]` 表示 `s[1..i]` 的哈希，子串 `s[l..r]` 的哈希直接写成：

```
long long get(int l, int r) {
    return (h[r] - h[l - 1] * pw[r - l + 1] % mod + mod) % mod;
}
```

构建时：

```
void build(char s[], int n) {
    h[0] = 0;
    pw[0] = 1;

    for (int i = 1; i <= n; i++) {
        h[i] = (h[i - 1] * B + s[i]) % mod;
        pw[i] = pw[i - 1] * B % mod;
    }
}
```

这两种写法本质相同，但下标差一位。程序阅读时，先判断字符串是从 0 开始还是从 1 开始，再判断 `get` 函数的边界。

写法	h[i] 含义	取 s[l..r]
0 开始 string	前 i 个字符，即 s[0..i-1]	h[r+1] - h[l] * pw[r-l+1]
1 开始字符数组	s[1..i]	h[r] - h[l-1] * pw[r-l+1]

表中代码表达式保留程序写法；它们分别对应数学意义上的 $h[r+1]-h[l]\times B^{r-l+1}$ 和 $h[r]-h[l-1]\times B^{r-l+1}$。

模数和负数

使用取模时，子串哈希的减法可能得到负数。

例如 h[r] - h[l - 1] * pw[len] % mod 可能小于 0。为了让结果落在 0 到 $mod-1$ 之间，常写：

```
(h[r] - h[l - 1] * pw[len] % mod + mod) % mod
```

如果一次 + mod 仍可能不够，说明前面的乘法或减法写法不规范。常见做法是在每一步都先 % mod，再加 mod，最后再 % mod。

C++ 中负数取模仍可能是负数。例如 -1 % 5 的结果是 -1，不是 4。所以字符串哈希中看到 + mod，通常就是为了修正负数。

例题 5：为什么要加 mod

设 $mod=7$，某次计算得到 h[r]=2，h[l-1]*pw[len]%mod=5。直接相减为 $2-5=-3$。

正确的模意义下，-3 等价于 4，因为 $-3+7=4$。代码写成 (-3 + 7) % 7 得到 4。

所以子串哈希中要写 + mod。

unsigned long long 写法

有些程序不用显式模数，而使用 unsigned long long。这种写法依赖无符号整数溢出后的截断效果，相当于自动对 2^{64} 取模。

代码可能写成：

```
using ull = unsigned long long;

const ull B = 131;
ull h[N], pw[N];

void build(string s) {
    int n = s.size();

    h[0] = 0;
    pw[0] = 1;

    for (int i = 0; i < n; i++) {
        h[i + 1] = h[i] * B + s[i];
        pw[i + 1] = pw[i] * B;
    }
}

ull get(int l, int r) {
    return h[r + 1] - h[l] * pw[r - l + 1];
}
```

这种写法没有 % mod，也不用处理负数。它代码短，程序阅读中常出现。

但要知道，它仍然可能发生哈希冲突。不要把 `unsigned long long` 哈希理解成完全不会错的字符串编号。

双哈希

为了降低冲突风险，可以同时使用两个不同模数，或者一个模数加一个 `unsigned long long`。比较两个子串时，两个哈希值都相同才认为子串相同。

例如使用两个模数：

```
const long long B = 131;
const long long mod1 = 1000000007;
const long long mod2 = 1000000009;

pair<long long, long long> get(int l, int r) {
    long long x = (h1[r] - h1[l - 1] * pw1[r - l + 1] % mod1 + mod1) % mod1;
    long long y = (h2[r] - h2[l - 1] * pw2[r - l + 1] % mod2 + mod2) % mod2;
    return {x, y};
}
```

双哈希不是数学上的绝对保证，但比单哈希更稳。第一轮中，如果程序同时维护 `h1,h2` 或返回 `pair`，通常就是双哈希。

例题 6：双哈希比较

两个子串的第一个哈希值相同，第二个哈希值不同。它们能否认为相同？

不能。

双哈希比较时，两个哈希值都相同，才把它们当作相同。只要其中一个不同，就说明两个子串不同。

如果两个哈希值都相同，仍然只能说“高度可能相同”。在第一轮题目语境里，通常按相同处理。

用哈希做字符串匹配

字符串哈希可以用来做模式串匹配。

设文本串为 `s`，模式串为 `p`，长度分别为 n 和 m。先计算模式串的哈希值 `hp`。再枚举文本串中所有长度为 m 的子串，比较它们的哈希是否等于 `hp`。

代码如下：

```
int count_match(string s, string p) {
    int n = s.size();
    int m = p.size();

    build(s);
    long long hp = 0;

    for (int i = 0; i < m; i++) {
        hp = (hp * B + p[i]) % mod;
    }

    int ans = 0;

    for (int i = 0; i + m <= n; i++) {
        if (get(i, i + m - 1) == hp) {
            ans++;
        }
    }
```

```
    return ans;
}
```

这里 `get(i, i + m - 1)` 表示文本串中从 `i` 开始、长度为 m 的子串哈希。若等于模式串哈希，就计数。

若担心冲突，可以在哈希相等后再逐字符核对，或者使用双哈希。

例题 7：哈希匹配位置

文本串 `s="ababcabc"`，模式串 `p="abc"`。枚举长度为 3 的子串，哪些位置哈希应等于模式串哈希？

长度为 3 的子串依次为：

起点	子串
0	`"aba"`
1	`"bab"`
2	`"abc"`
3	`"bca"`
4	`"cab"`
5	`"abc"`

所以起点 2 和 5 的子串哈希应等于模式串 `"abc"` 的哈希。

比较任意两个子串

字符串哈希更常见的用途，是快速回答“两个子串是否相同”。

例如有一个字符串 `s`，多次询问：`s[l1..r1]` 是否等于 `s[l2..r2]`。如果每次逐字符比较，最坏要花子串长度的时间。预处理前缀哈希后，每次只要计算两个哈希，复杂度为 $O(1)$。

判断时先看长度。如果长度不同，子串一定不同；长度相同，再比较哈希。

```
bool same(int l1, int r1, int l2, int r2) {
    if (r1 - l1 != r2 - l2) {
        return false;
    }

    return get(l1, r1) == get(l2, r2);
}
```

这里用的是 0 开始下标。长度相同的判断写成 `r1 - l1 == r2 - l2`，因为两边实际长度分别是 $r1 - l1 + 1$ 和 $r2 - l2 + 1$，同时加 1 不影响比较。

例题 8：先判长度

字符串 `s="abcdef"`。比较 `s[1..3]` 和 `s[2..5]` 是否相同。下标从 0 开始。

`s[1..3]` 长度为 3，内容是 `"bcd"`。`s[2..5]` 长度为 4，内容是 `"cdef"`。长度不同，不需要计算哈希，直接判断不同。

若比较 `s[1..3]` 和 `s[2..4]`，两者长度相同，才需要继续比较哈希。

最长公共前缀

哈希也可以配合二分求两个后缀的最长公共前缀。

给定两个起点 i, j，想知道 `s[i..]` 和 `s[j..]` 从开头起最多有多少个字符相同。设答案长度为 L。若长度为 x 的前缀相同，那么长度小于 x 的前缀也相同；若长度为 x 的前缀不同，那么长度大于 x 的前缀也不可能相同。

这个单调性适合二分。每次用哈希判断 `s[i..i+mid-1]` 和 `s[j..j+mid-1]` 是否相同。

例题 9：手算最长公共前缀

字符串 `s="ababc"`。比较后缀 `s[0..]` 和 `s[2..]` 的最长公共前缀长度。

`s[0..]="ababc"`，`s[2..]="abc"`。从开头比较：第一个字符都是 `a`，第二个字符都是 `b`，第三个字符分别是 `a` 和 `c`，不同。

所以最长公共前缀长度为 2。

若用哈希和二分，就会判断长度 2 的前缀相同，长度 3 的前缀不同，最后得到答案 2。

回文判断

字符串哈希还可以判断一个子串是否为回文。

做法是同时计算原串的前缀哈希和反串的前缀哈希。一个子串若和它在反串中对应位置的哈希相同，就可以认为它是回文。

例如原串 `s="abac"`，反串为 `"caba"`。原串中子串 `s[0..2]="aba"`，在反串中对应的也是 `"aba"`。哈希相同，因此它是回文。

下标换算要小心。若原串长度为 n，原串的子串 $[l, r]$ 对应到反串中的区间是 $[n-1-r, n-1-l]$。

例题 10：反串区间

字符串长度为 8。原串下标从 0 开始，子串区间为 $[2, 5]$。它在反串中对应哪个区间？

对应区间为 $[n-1-r, n-1-l]$，也就是 $[8-1-5, 8-1-2]$，得到 $[2, 5]$。

如果原区间是 $[1, 3]$，对应区间是 $[4, 6]$。

回文判断中，下标换算比哈希公式更容易错。程序阅读题若出现反串，要先把对应区间写清楚。

哈希冲突

哈希冲突指两个不同字符串得到相同哈希值。

例如某个哈希函数把很多字符串都映射到有限范围内，而字符串可能有非常多种，冲突无法完全避免。模数越小，冲突越容易出现。模数较大、底数选择合适、使用双哈希，可以降低冲突风险。

第一轮题目中，如果问“哈希相同是否一定字符串相同”，答案通常是否定的。只有字符串相同能推出哈希相同，反过来不一定成立。

判断	是否一定成立
字符串相同 ⇒ 哈希相同	一定
哈希不同 ⇒ 字符串不同	一定
哈希相同 ⇒ 字符串相同	不一定

程序阅读中，如果代码只比较哈希值，它通常是在概率意义或题目默认语境下判断相同。选择题如果讨论理论保证，要记住冲突可能存在。

例题 11：哈希相同能说明什么

两个字符串的哈希值相同，能否断定它们完全相同？

不能断定。

哈希函数可能发生冲突，不同字符串可能得到相同哈希值。若题目要求绝对正确，可以在哈希相同后再逐字符比较，或者使用其他确定性算法。

但如果两个字符串的哈希值不同，就可以断定它们不同。因为相同字符串按同一个哈希函数计算，结果必然相同。

程序阅读：前缀哈希

下面是一段常见的字符串哈希代码，下标从 1 开始：

```
const long long B = 131;
const long long mod = 1000000007;

long long h[N], pw[N];

void build(string s) {
    int n = s.size();

    pw[0] = 1;
    h[0] = 0;

    for (int i = 1; i <= n; i++) {
        h[i] = (h[i - 1] * B + s[i - 1]) % mod;
        pw[i] = pw[i - 1] * B % mod;
    }
}

long long get(int l, int r) {
    return (h[r] - h[l - 1] * pw[r - l + 1] % mod + mod) % mod;
}
```

虽然参数 `l,r` 是从 1 开始的，但 `string s` 本身下标从 0 开始。因此构建时使用 `s[i-1]`。

读这种代码时，要区分 “字符串对象的下标” 和 “哈希数组的下标”。它们不一定一致。

例题 12：下标对应

上面代码中，若要取原字符串 `s` 的前三个字符，应该调用 `get(1,3)` 还是 `get(0,2)`？

应调用 `get(1,3)`。

因为 `get` 函数按 1 开始区间设计。虽然 `s` 本身是 `string`，但构建时已经把 `s[0]` 放进了 `h[1]`，把 `s[1]` 放进了 `h[2]`。

如果调用 `get(0,2)`，会访问 `h[-1]`，这是错误的。

程序阅读：滚动窗口哈希

有些代码不使用前缀哈希，而是在固定长度窗口上滚动更新。

例如要在文本串中找长度为 m 的模式串，可以先算第一个窗口 `s[0..m-1]` 的哈希，然后每次移除最左字符，加入新的右端字符。

示意代码如下：

```
long long cur = 0;

for (int i = 0; i < m; i++) {
    cur = (cur * B + s[i]) % mod;
}

for (int i = m; i < n; i++) {
    cur = (cur - s[i - m] * pw[m - 1] % mod + mod) % mod;
    cur = (cur * B + s[i]) % mod;
}
```

第一句减去离开窗口的字符贡献，第二句整体乘以底数并加入新字符。

这种写法和前缀哈希都能取固定长度子串哈希。前缀哈希更适合任意区间查询，滚动窗口更适合按顺序扫描所有等长窗口。

例题 13：窗口移动

设窗口长度为 3，当前窗口是 `"abc"`，下一个窗口是 `"bcd"`。滚动哈希做了什么？

从 `"abc"` 变成 `"bcd"`，需要移除最左边的 `a`，保留 `"bc"`，再加入右边的 `d`。在多项式哈希里，`a` 原来在最高位，所以要先减去它的贡献；剩下部分整体左移一位，也就是乘以底数；最后加上 `d` 的字符值。

这和手动维护一个长度固定的数字窗口很像。

完善程序中的常见空缺

字符串哈希完善程序常在构建、取子串和比较位置留空。

前缀哈希构建常见空缺如下：

空缺位置	应关注的问题
`pw[0]`	应初始化为 1
`pw[i]`	应由 `pw[i-1] * B` 得到
`h[0]`	通常初始化为 0
`h[i]`	由 `h[i-1] * B +` 当前字符值 得到

子串哈希常见空缺如下：

空缺位置	应关注的问题
子串长度	$r-l+1$
前缀贡献	$h[l-1]\times B^{r-l+1}$
减法修正	加上 `mod` 后再取模
0 开始写法	注意使用 `h[r+1]` 和 `h[l]`

比较类程序常见空缺如下：

空缺位置	应关注的问题
子串长度是否相同	长度不同直接不同
哈希比较	比较 `get(l1,r1)` 和 `get(l2,r2)`
双哈希	两个哈希都要相同
回文判断	原串区间要换到反串对应区间

例题 14：补全子串哈希

下面代码按 1 开始下标取子串哈希：

```
long long get(int l, int r) {
    int len = ________;
    return (h[r] - h[l - 1] * pw[len] % mod + mod) % mod;
}
```

空缺处应填 `r - l + 1`。

子串 `s[l..r]` 一共有 $r-l+1$ 个字符。前面被删掉的前缀 `s[1..l-1]` 在 `h[r]` 中被整体乘上了 B^{r-l+1}，所以要乘 `pw[len]` 后再相减。

例题 15：补全构建代码

下面代码构建前缀哈希：

```
pw[0] = 1;
h[0] = 0;

for (int i = 1; i <= n; i++) {
    pw[i] = ________;
    h[i] = ________;
}
```

若字符串从 1 开始存，空缺应填 `pw[i - 1] * B % mod` 和 `(h[i - 1] * B + s[i]) % mod`。

若字符串使用 C++ `string`，但哈希数组从 1 开始，第二个空通常写成 `(h[i - 1] * B + s[i - 1]) % mod`。差别只在字符下标。

常见错误

第一，把哈希相同当成绝对相同。哈希可能冲突，理论上不能反推字符串一定相同。

第二，子串长度写错。`s[l..r]` 的长度是 $r-l+1$，少写 1 会导致幂次错误。

第三，0 开始和 1 开始混用。`string` 下标、哈希数组下标、题目位置编号可能是三套不同约定。

第四，取模减法忘记加 `mod`。C++ 中负数取模仍可能为负，比较时会出错。

第五，忘记预处理 `pw[0]=1`。如果幂数组初值错，后面所有子串哈希都会错。

第六，双哈希只比较其中一个值。双哈希要两个值都相等才认为相同。

第七，回文判断时反串区间换算错误。原串 $[l,r]$ 对应反串 $[n-1-r,n-1-l]$。

第八，滚动窗口中删除左端字符时，幂次用错。长度为 m 的窗口，最左字符贡献通常乘的是 B^{m-1}。

小结

字符串哈希把字符串映射成数值，使子串比较可以通过哈希值完成。预处理前缀哈希和幂数组后，任意子串哈希可以在 $O(1)$ 时间取出。

本章需要重点记住：

- 前缀哈希常用递推 `h[i] = h[i-1] * B + val`；
- 子串 `s[l..r]` 的长度是 $r-l+1$；
- 1 开始写法中，子串哈希对应 $h[r]-h[l-1]\times B^{r-l+1}$；
- 0 开始写法中，子串哈希对应 $h[r+1]-h[l]\times B^{r-l+1}$；
- 取模减法要加 `mod` 修正；

- 哈希不同一定字符串不同，哈希相同不保证字符串相同；
- 双哈希可以降低冲突影响；
- 哈希可用于模式匹配、子串比较、最长公共前缀和回文判断；
- 程序阅读中先判断下标体系，再看 `get` 函数含义。

下一章进入 LCS 与子序列。那一章会回到“子序列不要求连续”这一概念，讲最长公共子序列的状态表示和常见程序阅读方法。

0x25 LCS 与子序列

第 34 章讲过，子串要求连续，子序列不要求连续。第 35 章和第 36 章讨论的 KMP、字符串哈希，主要服务于“连续片段”的比较。本章换到另一类问题：在两个字符串中，按原有顺序选出若干字符，使它们相同，并尽量长。

这个问题叫最长公共子序列，英文是 Longest Common Subsequence，常写作 LCS。

CSP-S 第一轮中，LCS 常见于动态规划程序阅读和完善程序。题目不一定直接写“LCS”，但代码里如果出现二维数组 `dp[i][j]`，并且围绕两个字符串的前 i 个字符、前 j 个字符转移，就要警惕它可能是在求最长公共子序列。

子序列

子序列是从原字符串中删去若干字符后，剩下的字符按原相对顺序组成的序列。可以一个字符都不删，也可以删掉很多字符，但不能改变剩余字符的相对顺序。

例如字符串 `"abcde"` 的子序列包括 `"ace"`、`"bd"`、`"abcde"`、`"a"` 和空串。`"aec"` 不是它的子序列，因为在原串中 `e` 在 `c` 后面，顺序被改变了。

子序列和子串的区别非常重要：

概念	是否要求连续	示例
子串	要求连续	`"bcd"` 是 `"abcde"` 的子串
子序列	不要求连续	`"ace"` 是 `"abcde"` 的子序列

字符串 `"abcde"` 中，`"bcd"` 既是子串，也是子序列；`"ace"` 是子序列，但不是子串。

例题 1：判断子序列

判断 `"bdf"` 是否为 `"abcdef"` 的子序列。

从 `"abcdef"` 中依次选择 `b`、`d`、`f`，它们的相对顺序没有改变，所以 `"bdf"` 是 `"abcdef"` 的子序列。

判断 `"daf"` 是否为 `"abcdef"` 的子序列。

虽然三个字符都出现在原串中，但在原串里 `a` 在 `d` 前面，而 `"daf"` 要求 `d` 在 `a` 前面。顺序不一致，所以 `"daf"` 不是子序列。

公共子序列

两个字符串的公共子序列，是同时作为两个字符串子序列的序列。

例如 `A="abcde"`，`B="ace"`。字符串 `"ace"` 是 `A` 的子序列，也是 `B` 自己的子序列，所以 `"ace"` 是它们的公共子序列。

再看 `A="abcbdab"`，`B="bdcaba"`。序列 `"bdab"` 是 `A` 的子序列，也是 `B` 的子序列，因此它是公共子序列。

公共子序列不要求在两个字符串中位置相同，只要求各自内部顺序一致。

例题 2：找公共子序列

设 `A="abac"`，`B="cab"`。下面哪些是公共子序列？

A. `"ab"`

B. `"ac"`

C. `"ba"`

D. `"c"`

`"ab"` 是 `A` 的子序列，也是 `B` 的子序列，因为 `B="cab"` 中可以选择 `a,b`。

`"ac"` 是 `A` 的子序列，但不是 `B` 的子序列。在 `B` 中，`c` 在 `a` 前面，不能得到 `"ac"`。

`"ba"` 是 `A` 的子序列，因为可以选择下标较早的 `b` 和后面的 `a`；但 `B` 中 `b` 在最后，后面没有 `a`，所以不是公共子序列。

`"c"` 同时出现在两个字符串中，是公共子序列。

所以 A 和 D 正确。

LCS

LCS 是最长公共子序列。它要求在所有公共子序列中，长度尽量大。

例如 `A="abcde"`，`B="ace"`。它们的一个最长公共子序列是 `"ace"`，长度为 3。

例如 `A="abcbdab"`，`B="bdcaba"`。它们的最长公共子序列长度为 4，`"bcba"` 和 `"bdab"` 都可以作为答案。LCS 不一定唯一，但长度是确定的。

字符串 A	字符串 B	一个 LCS	LCS 长度
`"abcde"`	`"ace"`	`"ace"`	3
`"abc"`	`"def"`	空串	0
`"aaaa"`	`"aa"`	`"aa"`	2

本章多数时候只求 LCS 长度。若要输出具体方案，还需要额外记录转移路径，后面会简单说明。

状态表示

求两个字符串 LCS 的常见 DP 状态是：

`dp[i][j]` 表示字符串 `A` 的前 i 个字符和字符串 `B` 的前 j 个字符的 LCS 长度。

这里“前 i 个字符”指 `A[0..i-1]`，不是到下标 i。这是因为 C++ 字符串下标从 0 开始，而 DP 数组通常多开一行一列，用 `dp[0][j]` 和 `dp[i][0]` 表示有一个字符串为空的情况。

如果 $i = 0$ 或 $j = 0$，至少有一个字符串为空，LCS 长度为 0。所以初始化为：

```
dp[0][j] = 0;
dp[i][0] = 0;
```

在全局数组中，这些值默认就是 0。

例题 3：理解 dp[i][j]

设 `A="abc"`，`B="adc"`。`dp[2][2]` 表示什么？

`A` 的前 2 个字符是 `"ab"`，`B` 的前 2 个字符是 `"ad"`。所以 `dp[2][2]` 表示 `"ab"` 和 `"ad"` 的 LCS 长度。

它们的最长公共子序列是 `"a"`，长度为 1，所以 `dp[2][2]=1`。

注意 `dp[2][2]` 不表示比较 `A[2]` 和 `B[2]`，而是比较两个前缀。

转移

计算 dp[i][j] 时，要看 A 的第 i 个字符和 B 的第 j 个字符。若使用 C++ 下标，它们分别是 A[i-1] 和 B[j-1]。

如果 A[i-1] == B[j-1]，这两个字符可以一起放进公共子序列的末尾。此时有：

$dp[i][j] = dp[i-1][j-1] + 1$。

如果 A[i-1] != B[j-1]，这两个末尾字符不能同时作为同一个位置加入 LCS。此时可以丢掉 A 的末尾字符，或者丢掉 B 的末尾字符，取较大的结果：

$dp[i][j] = \max(dp[i-1][j], dp[i][j-1])$。

代码如下：

```
int lcs(string a, string b) {
    int n = a.size();
    int m = b.size();

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            if (a[i - 1] == b[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1] + 1;
            } else {
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
            }
        }
    }

    return dp[n][m];
}
```

这里的两层循环从 1 开始，是为了让 dp[i-1][j]、dp[i][j-1] 和 dp[i-1][j-1] 都已经算好。

例题 4：一次转移

设 A="abc"，B="adc"。计算 dp[3][3] 时，比较的是哪两个字符？如何转移？

dp[3][3] 比较 A[2] 和 B[2]，也就是 c 和 c。它们相等，所以 dp[3][3]=dp[2][2]+1。

前面已经知道 dp[2][2]=1，所以 dp[3][3]=2。对应的 LCS 可以是 "ac"。

手算 DP 表

用一个小例子完整看一遍。

设 A="abc"，B="ac"。行表示 A 的前缀长度，列表示 B 的前缀长度。

dp[i][j]	$j=0$	$j=1$ (a)	$j=2$ (ac)
$i=0$	0	0	0
$i=1$ (a)	0	1	1
$i=2$ (ab)	0	1	1
$i=3$ (abc)	0	1	2

解释几个格子：

dp[1][1] 比较 a 和 a，相等，所以为 1。

dp[2][2] 比较 b 和 c，不相等，所以取 dp[1][2] 和 dp[2][1] 的较大值，得到 1。

dp[3][2] 比较 c 和 c，相等，所以 dp[3][2]=dp[2][1]+1=2。

最终 LCS 长度为 dp[3][2]=2，对应公共子序列 "ac"。

例题 5：手算 LCS

求 A="abcbd"，B="abd" 的 LCS 长度。

可以看出 "abd" 是 A 的子序列：选择 A 中的 a、第一个 b、最后的 d。它也是 B 自己。

所以 LCS 长度至少为 3。由于 B 长度就是 3，LCS 不可能超过 3。因此 LCS 长度为 3。

这类小题可以直接观察；程序阅读题中则要按 dp 表转移。

为什么不相等时取两边最大

当 A[i-1] != B[j-1] 时，两个末尾字符不能作为同一个匹配字符同时加入公共子序列。一个 LCS 要么不使用 A[i-1]，落在 A[0..i-2] 和 B[0..j-1] 中；要么不使用 B[j-1]，落在 A[0..i-1] 和 B[0..j-2] 中。

这两种选择对应 dp[i-1][j] 和 dp[i][j-1]。取较大值，就是当前两个前缀的 LCS 长度。

不要写成 dp[i-1][j-1]。当末尾字符不相等时，同时丢掉两个末尾字符可能错过答案。

例题 6：不能只看左上角

设 A="ab"，B="acb"。计算 dp[2][3] 时，比较 A[1]='b' 和 B[2]='b'，相等，所以 dp[2][3]=dp[1][2]+1=2。

如果看 dp[2][2]，比较的是 "ab" 和 "ac"，LCS 长度为 1。

这个例子说明，DP 表中不同方向的格子含义不同。相等时取左上角加 1；不相等时取上方和左方较大值。

LCS 不要求连续

这是本章最容易和前几章混淆的地方。

A="abcde"，B="ace"。LCS 是 "ace"，长度为 3。但 "ace" 不是 A 的子串，因为它在 A 中不连续。

所以 LCS 和最长公共子串不是同一个问题。

最长公共子串要求连续；LCS 只要求顺序一致。状态转移也不同。最长公共子串在字符不相等时，通常把当前连续长度清零；LCS 在字符不相等时，会取上方和左方较大值。

问题	是否要求连续	不相等时常见处理
LCS	不要求连续	取上方、左方较大值
最长公共子串	要求连续	当前连续长度变为 0

例题 7：LCS 与最长公共子串

设 A="abcde"，B="ace"。

它们的 LCS 长度为 3，一个 LCS 是 "ace"。

它们的最长公共子串长度为 1，因为两个字符串中没有长度为 2 的连续公共片段。"a"、"c"、"e" 都可以作为长度为 1 的公共子串。

所以看到“子序列”三个字，不能按“连续子串”来做。

输出一个 LCS

如果只求长度，返回 dp[n][m] 就够了。若要输出一个具体的 LCS，可以从 dp[n][m] 反向回溯。

从位置 (i,j) 开始：

- 若 a[i-1] == b[j-1]，说明这个字符可以放入答案末尾，然后走到 (i-1,j-1)；
- 若不相等，比较 dp[i-1][j] 和 dp[i][j-1]，向较大的方向走；
- 直到 $i = 0$ 或 $j = 0$ 结束。

由于是从后往前找，得到的字符要反转。

代码如下：

```
string build_lcs(string a, string b) {
    int i = a.size();
    int j = b.size();
    string ans;

    while (i > 0 && j > 0) {
        if (a[i - 1] == b[j - 1]) {
            ans += a[i - 1];
            i--;
            j--;
        } else if (dp[i - 1][j] >= dp[i][j - 1]) {
            i--;
        } else {
            j--;
        }
    }

    reverse(ans.begin(), ans.end());
    return ans;
}
```

当 dp[i-1][j] 和 dp[i][j-1] 相等时，向上或向左都可能得到一个合法 LCS。不同选择可能输出不同方案，但长度相同。

例题 8：回溯方案不唯一

设 A="abc"，B="bac"。它们的 LCS 长度为 2。

一个 LCS 是 "ac"，另一个 LCS 是 "bc"。程序回溯时，如果遇到相等情况选择不同方向，可能输出不同答案。

因此题目若只要求输出长度，就不要纠结具体是哪一个 LCS。若题目要求输出字典序最小 LCS，则需要额外设计，本章不展开。

空串和边界

LCS 的边界条件很简单：任何字符串和空串的 LCS 长度都是 0。

这就是为什么 DP 表要多开一行一列：

- dp[0][j]=0；
- dp[i][0]=0。

如果数组是局部变量，需要手动初始化。若是全局数组，C++ 会默认初始化为 0。

例题 9：边界值

A="", B="abc"。它们的 LCS 长度是多少？

空串没有字符，无法和 B 形成非空公共子序列。所以 LCS 长度为 0。

在 DP 表中，这对应 `dp[0][3]=0`。

复杂度

若两个字符串长度分别为 n 和 m，LCS 的普通 DP 有 $n \times m$ 个状态。每个状态只做常数次比较和取最大值，所以时间复杂度为 $O(nm)$。

空间上，如果保留整个 `dp` 表，需要 $O(nm)$ 空间。如果只求长度，可以用滚动数组把空间降到 $O(m)$，因为当前行只依赖上一行和当前行左边的值。

CSP-S 第一轮中，程序阅读更常见的是完整二维数组；完善程序中也可能出现滚动数组，考查更新顺序。

滚动数组

LCS 的转移只依赖上一行和当前行左边。可以用两行数组：

```
for (int i = 1; i <= n; i++) {
    int cur = i & 1;
    int pre = cur ^ 1;

    for (int j = 1; j <= m; j++) {
        if (a[i - 1] == b[j - 1]) {
            dp[cur][j] = dp[pre][j - 1] + 1;
        } else {
            dp[cur][j] = max(dp[pre][j], dp[cur][j - 1]);
        }
    }
}
```

这里 `cur` 表示当前行，`pre` 表示上一行。

如果使用一维数组，也要额外保存左上角旧值。因为 `dp[j]` 在更新前表示上一行的 `dp[i-1][j]`，更新后表示当前行的 `dp[i][j]`。`dp[j-1]` 更新后表示当前行的 `dp[i][j-1]`，但左上角 `dp[i-1][j-1]` 会被覆盖，需要提前保存。

一维写法比二维更容易出错，第一轮若考程序阅读，重点看变量是否保存了左上角。

例题 10：为什么要保存左上角

LCS 转移中，若 `a[i-1] == b[j-1]`，需要用 `dp[i-1][j-1]+1`。在一维数组中，`dp[j-1]` 可能已经被更新成当前行的 `dp[i][j-1]`，不再是上一行的 `dp[i-1][j-1]`。

所以一维优化时，通常用变量 `last` 保存更新前的左上角值。否则相等时会用错状态。

子序列判断的双指针

判断一个字符串是否为另一个字符串的子序列，可以不用 LCS，直接双指针。

```
bool is_sub(string a, string b) {
    int j = 0;

    for (int i = 0; i < b.size(); i++) {
        if (j < a.size() && a[j] == b[i]) {
            j++;
        }
    }

    return j == a.size();
```

```
}
```

这里判断的是 a 是否为 b 的子序列。指针 i 扫描长串 b，指针 j 指向短串 a 中等待匹配的位置。

如果只是判断能不能匹配出一个给定子序列，双指针比 LCS 更简单。LCS 是在两个字符串中自己寻找最长公共子序列。

例题 11：双指针判断

判断 "ace" 是否为 "abcde" 的子序列。

扫描 "abcde"：遇到 a，匹配第一个字符；继续扫，遇到 c，匹配第二个字符；继续扫，遇到 e，匹配第三个字符。此时短串已经全部匹配，所以是子序列。

判断 "aec" 是否为 "abcde" 的子序列。

先匹配 a，再匹配 e。但 e 后面已经没有 c，所以不是子序列。

LCS 与最少删除

LCS 还可以解释一些“删除字符使两个字符串相同”的问题。

若两个字符串长度分别为 n 和 m，它们的 LCS 长度为 L。保留这段公共子序列，删除其他字符，就能让两个字符串都变成同一个序列。需要删除的字符总数为 $(n-L)+(m-L)=n+m-2L$。

例如 A="sea"，B="eat"。它们的 LCS 是 "ea"，长度为 2。若要通过删除字符使两个字符串相同，A 删除 s，B 删除 t，共删除 2 个字符。公式给出 $3+3-2\times 2=2$。

例题 12：最少删除次数

A="abcde"，B="ace"。若只能删除字符，最少删除多少个字符能让两个字符串相同？

它们的 LCS 是 "ace"，长度为 3。A 长度为 5，B 长度为 3。

最少删除次数为 $5+3-2\times 3=2$。也就是从 A 中删除 b 和 d，两个字符串都变成 "ace"。

LCS 与最长回文子序列

最长回文子序列可以通过 LCS 理解。

一个字符串的最长回文子序列长度，等于它和反串的 LCS 长度。因为回文串正着读和反着读相同，它在原串中作为子序列出现，也会在反串中对应出现。

例如 s="bbbab"，反串为 "babbb"。它们的 LCS 长度为 4，对应最长回文子序列可以是 "bbbb"。

这个结论在第一轮中可能作为选择题或程序阅读背景出现。若程序先构造反串，再对原串和反串做 LCS，很可能是在求最长回文子序列长度。

例题 13：反串 LCS

字符串 s="abcda"。它的反串是 "adcba"。若程序求 s 和反串的 LCS，它在求什么？

它在求 s 的最长回文子序列长度。

例如 "aca" 是 s 的一个回文子序列，长度为 3。程序通过 LCS 的方式寻找这样的最长长度。

注意这是最长回文子序列，不是最长回文子串。回文子串要求连续，回文子序列不要求连续。

程序阅读：LCS 长度

下面是一段典型 LCS 程序：

```
int dp[N][N];

int lcs(string a, string b) {
    int n = a.size();
    int m = b.size();

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            if (a[i - 1] == b[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1] + 1;
            } else {
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
            }
        }
    }

    return dp[n][m];
}
```

读这段程序时，要抓住四个点：

第一，`i` 和 `j` 是前缀长度，不是字符串下标。真正访问字符时使用 `a[i-1]` 和 `b[j-1]`。

第二，字符相等时，从左上角转移，并加 1。

第三，字符不相等时，从上方和左方取较大值。

第四，答案是 `dp[n][m]`，表示两个完整字符串的 LCS 长度。

例题 14：判断程序功能

若把上面代码中不相等时的转移改成：

```
dp[i][j] = 0;
```

它还在求 LCS 吗？

不是。

不相等时清零，是最长公共子串常见写法的一部分，因为连续匹配被打断了。LCS 不要求连续，即使末尾字符不相等，前面的公共子序列仍然可以保留，所以应取 `dp[i-1][j]` 和 `dp[i][j-1]` 的较大值。

程序阅读：输出方案

下面代码在已经求出 `dp` 表后，输出一个 LCS：

```
string ans;
int i = n, j = m;

while (i > 0 && j > 0) {
    if (a[i - 1] == b[j - 1]) {
        ans += a[i - 1];
        i--;
        j--;
    } else if (dp[i - 1][j] >= dp[i][j - 1]) {
```

```
        i--;
    } else {
        j--;
    }
}

reverse(ans.begin(), ans.end());
cout << ans << endl;
```

这段代码从右下角往左上角回溯。遇到相同字符时，把它加入答案。由于加入顺序是从后往前，所以最后要反转。

如果去掉 `reverse`，输出会变成反向的 LCS。

例题 15：为什么要反转

设回溯时依次加入的字符是 `c`、`b`、`a`。这说明从后往前找到了 LCS 的字符。真正的 LCS 应该是 `"abc"`，所以需要反转。

如果直接输出 `"cba"`，顺序就错了。子序列必须保持原字符串中的相对顺序。

完善程序中的常见空缺

LCS 完善程序常在状态定义、相等转移、不相等转移和回溯位置留空。

求长度时常见空缺如下：

空缺位置	应关注的问题
访问字符	`a[i - 1]` 与 `b[j - 1]`
相等转移	`dp[i - 1][j - 1] + 1`
不相等转移	`max(dp[i - 1][j], dp[i][j - 1])`
初始边界	`dp[0][j]=0, dp[i][0]=0`
最终答案	`dp[n][m]`

输出方案时常见空缺如下：

空缺位置	应关注的问题
字符相等	加入 `a[i-1]` 或 `b[j-1]`
相等后移动	`i--, j--`
不相等时	向 `dp` 值较大的方向走
输出前	反转答案

滚动数组时常见空缺如下：

空缺位置	应关注的问题
当前行	`cur = i & 1`
上一行	`pre = cur ^ 1`
左上角	来自上一行上一列
当前行左边	已经更新过的 `dp[cur][j-1]`

例题 16：补全 LCS 转移

下面代码用于求 LCS 长度：

```
if (a[i - 1] == b[j - 1]) {
    dp[i][j] = ________;
} else {
    dp[i][j] = ________;
}
```

第一个空应填 `dp[i - 1][j - 1] + 1`。

第二个空应填 `max(dp[i - 1][j], dp[i][j - 1])`。

字符相等时，可以把这个字符接在两个前缀的 LCS 后面；字符不相等时，只能在删去其中一个末尾字符的两种情况中取较大值。

例题 17：补全回溯

下面代码用于输出一个 LCS：

```
while (i > 0 && j > 0) {
    if (a[i - 1] == b[j - 1]) {
        ans += a[i - 1];
        ________;
    } else if (dp[i - 1][j] >= dp[i][j - 1]) {
        i--;
    } else {
        j--;
    }
}
```

空缺处应填 `i--, j--`，实际代码中写成两条语句：

```
i--;
j--;
```

因为当前两个字符相同，并且已经被加入答案，接下来要继续回溯两个字符串更短的前缀。

常见错误

第一，把子序列当成子串。LCS 不要求连续，最长公共子串才要求连续。

第二，误解 `dp[i][j]` 的含义。它表示前 i 个字符和前 j 个字符的 LCS 长度，不是下标 i, j 两个字符之间的关系。

第三，访问字符时忘记减 1。循环变量从 1 到 n，字符串下标从 0 开始，所以访问 `a[i-1]`。

第四，字符不相等时写成 `dp[i-1][j-1]`。这样会同时丢掉两个末尾字符，可能错过答案。

第五，把 LCS 和最长公共子串的转移混用。不相等时，LCS 取最大值；最长公共子串通常清零。

第六，输出方案时忘记反转。回溯从末尾开始，加入的字符顺序是反的。

第七，认为 LCS 一定唯一。LCS 长度唯一，但具体方案可能有多个。

第八，滚动数组时覆盖了左上角状态。一维优化需要额外变量保存旧的 `dp[i-1][j-1]`。

小结

LCS 是最长公共子序列。子序列不要求连续，只要求相对顺序不变。两个字符串的 LCS 可以用二维动态规划求解。

本章需要重点记住：

- 子串要求连续，子序列不要求连续；
- `dp[i][j]` 表示两个前缀的 LCS 长度；
- 字符相等时，$dp[i][j] = dp[i-1][j-1] + 1$；
- 字符不相等时，$dp[i][j] = \max(dp[i-1][j], dp[i][j-1])$；
- LCS 普通 DP 时间复杂度为 $O(nm)$；
- LCS 长度唯一，但具体方案不一定唯一；
- 输出方案时要从右下角回溯，并在最后反转；
- LCS 可用于理解最少删除问题和最长回文子序列；
- 程序阅读中要分清 LCS 和最长公共子串。

下一章进入编码类程序阅读。那一章会讲字符到数字、数字到字符、进制式编码、压缩与还原等常见程序结构，重点放在如何追踪字符串变量的变化。

0x26 编码类程序阅读

第 37 章讲了 LCS 与子序列。本章继续留在字符串与序列这一编，但重点不再是匹配或动态规划，而是编码类程序阅读。

编码类程序常把字符转成数字，把数字转成字符，或者按照某种规则把一个字符串改写成另一个字符串。它们看起来不像一个固定算法，却很适合出程序阅读题：变量多、下标多、循环边界细，稍不注意就会把字符值、数字值、位置编号和输出格式混在一起。

字符和数字

C++ 中，字符可以参与整数运算。常见写法是把数字字符转成数字：

```
int x = s[i] - '0';
```

如果 `s[i]` 是字符 `'7'`，那么 `s[i] - '0'` 的结果是整数 7。

反过来，如果有一个 0 到 9 的整数 `x`，可以转成对应数字字符：

```
char c = x + '0';
```

若 `x=7`，则 `c` 是字符 `'7'`。

这两句在程序阅读中非常常见。要注意字符 `'7'` 和整数 7 不是同一个东西。字符 `'7'` 是字符，整数 7 是数值；它们能相互转换，是因为数字字符在常见字符编码中按顺序排列。

表达式	含义
`'5' - '0'`	整数 5
`'9' - '0'`	整数 9
`3 + '0'`	字符 `'3'`
`s[i] - 'a'`	小写字母相对 `a` 的编号

例题 1：字符转数字

执行下面代码：

```
string s = "508";
int x = 0;

for (int i = 0; i < s.size(); i++) {
    x = x * 10 + (s[i] - '0');
}

cout << x << endl;
```

输出是多少？

初始 `x=0`。

读到字符 `'5'`，`s[i]-'0'=5`，所以 `x=0*10+5=5`。

读到字符 `'0'`，`s[i]-'0'=0`，所以 `x=5*10+0=50`。

读到字符 `'8'`，`s[i]-'0'=8`，所以 `x=50*10+8=508`。

最后输出 508。

这段代码把十进制数字字符串转成了整数。

数字字符串转整数

把数字字符串转整数的核心是“原来的数乘以 10，再加上新数字”。

例如字符串 `"1234"`：

- 读到 `1`，当前数为 1；
- 读到 `2`，当前数为 $1 \times 10 + 2 = 12$；
- 读到 `3`，当前数为 $12 \times 10 + 3 = 123$；
- 读到 `4`，当前数为 $123 \times 10 + 4 = 1234$。

代码如下：

```
int toint(string s) {
    int x = 0;

    for (int i = 0; i < s.size(); i++) {
        x = x * 10 + (s[i] - '0');
    }

    return x;
}
```

程序阅读时，要看清它读的是一位一位的字符，而不是一次读完整数。若字符串中含有非数字字符，这段代码就不再表示普通十进制转整数。

例题 2：前导零

执行：

```
string s = "0072";
int x = 0;

for (int i = 0; i < s.size(); i++) {
    x = x * 10 + (s[i] - '0');
}

cout << x << endl;
```

输出是多少？

计算过程为：读到第一个 `0`，`x=0`；读到第二个 `0`，`x=0`；读到 `7`，`x=7`；读到 `2`，`x=72`。

所以输出 72。

前导零在转成整数后会消失。如果题目要求保留原字符串格式，不能随便转成整数再输出。

整数转数字字符串

把整数转成字符串，常见做法是反复取个位。

```
string tostring(int x) {
    if (x == 0) {
        return "0";
    }

    string s;

    while (x > 0) {
        int d = x % 10;
        s += char(d + '0');
        x /= 10;
    }

    reverse(s.begin(), s.end());
    return s;
}
```

为什么要反转？因为每次 `x % 10` 取到的是当前个位。对于 1234，依次取到 $4, 3, 2, 1$，加入字符串后得到 `"4321"`，需要反转成 `"1234"`。

例题 3：整数转字符串

执行 `tostring(305)` 的过程中，字符串如何变化？

初始 `s` 为空。

$305 \bmod 10 = 5$，加入 `'5'`，`s="5"`，然后 `x=30`。

$30 \bmod 10 = 0$，加入 `'0'`，`s="50"`，然后 `x=3`。

$3 \bmod 10 = 3$，加入 `'3'`，`s="503"`，然后 `x=0`。

循环结束后反转，得到 `"305"`。

若忘记反转，输出会变成 `"503"`。

字母编号

小写字母也常被转成编号。若 `c` 是小写字母，则 `c - 'a'` 得到 0 到 25 的编号，`c - 'a' + 1` 得到 1 到 26 的编号。

```
int id = c - 'a';
```

例如 `a` 的编号为 0，`b` 的编号为 1，`z` 的编号为 25。

若要从编号转回字符：

```
char c = id + 'a';
```

其中 `id` 应在 0 到 25 之间。

大写字母同理，可以用 `c - 'A'` 得到 0 到 25 的编号。

例题 4：字母编号

执行下面代码：

```
string s = "cab";
int sum = 0;
```

```
for (int i = 0; i < s.size(); i++) {
    sum += s[i] - 'a' + 1;
}

cout << sum << endl;
```

输出是多少？

c 对应 3，a 对应 1，b 对应 2。所以总和为 $3+1+2=6$。

输出 6。

这里使用的是 1 到 26 编号。如果写成 `s[i] - 'a'`，则对应 0 到 25 编号，结果会变成 $2+0+1=3$。

大小写转换

大小写转换也常出现在编码类程序中。

若 c 是小写字母，转成大写可以写：

```
c = c - 'a' + 'A';
```

若 c 是大写字母，转成小写可以写：

```
c = c - 'A' + 'a';
```

也可以利用两个字符之间的差值：

```
c = c - 32;
```

但在教材正文中，更建议看成 `c - 'a' + 'A'`，因为它直接表达了“先求小写字母编号，再转成对应大写字母”的过程。

例题 5：大小写转换

执行：

```
char c = 'd';
c = c - 'a' + 'A';
cout << c << endl;
```

输出是什么？

`'d' - 'a' = 3`，再加上 `'A'`，得到大写字母中编号为 3 的字符，也就是 `'D'`。

输出 D。

若原字符不是小写字母，这种转换就没有普通大小写含义。程序阅读题中要看题目是否保证输入范围。

循环移位编码

循环移位是一类很常见的编码规则。比如把每个小写字母向后移动 k 位，超过 z 后回到 a。

若小写字母 c 的编号为 `c - 'a'`，移动 k 位后的编号为 `(c - 'a' + k) % 26`，再转回字符：

```
char t = (c - 'a' + k) % 26 + 'a';
```

例如 z 向后移动 2 位：z 的编号是 25，$(25+2) \bmod 26 = 1$，对应 b。

例题 6：循环移位

字符串 s="xyz"，每个字符向后移动 3 位，得到什么？

x 的编号是 23，移动后为 $(23+3) \bmod 26 = 0$，得到 a。

y 的编号是 24，移动后为 $(24+3) \bmod 26 = 1$，得到 b。

z 的编号是 25，移动后为 $(25+3) \bmod 26 = 2$，得到 c。

所以结果是 "abc"。

向前移动与负数

若把字母向前移动 k 位，常见写法是：

```
char t = (c - 'a' - k + 26) % 26 + 'a';
```

这里加 26 是为了避免出现负数。

例如 a 向前移动 1 位，应得到 z。若直接算 (0 - 1) % 26，在 C++ 中结果可能是 -1，再加 'a' 就不是 z。写成 (0 - 1 + 26) % 26，结果为 25，对应 z。

如果 k 可能很大，可以先写 k %= 26，或者加上足够大的 26 的倍数。

例题 7：向前移动

字符 c='b'，向前移动 3 位，结果是什么？

b 的编号是 1。计算 $(1-3+26) \bmod 26 = 24$。编号 24 对应 y。

所以结果是 y。

若忘记加 26，就会遇到负数取模问题。

数位编码

有些程序把一个整数的每一位拆开，再按规则编码。

例如把每一位数字都加 3 后对 10 取模：

```
string encode(string s) {
    for (int i = 0; i < s.size(); i++) {
        int x = s[i] - '0';
        x = (x + 3) % 10;
        s[i] = x + '0';
    }

    return s;
}
```

输入 "890" 时，8 变成 1，9 变成 2，0 变成 3，结果为 "123"。

这种程序本质上还是字符到数字、数字到字符。读题时要看它是按“整个数”处理，还是按“每一位字符”处理。

例题 8：逐位处理

执行：

```cpp
string s = "109";
for (int i = 0; i < s.size(); i++) {
    int x = s[i] - '0';
    x = (x * 2) % 10;
    s[i] = x + '0';
}
cout << s << endl;
```

输出是什么？

`1` 变成 2，`0` 变成 0，`9` 变成 $18 \bmod 10 = 8$。

所以输出 `"208"`。

注意这不是把整数 109 乘以 2。如果按整数乘法，结果应为 218；本程序是逐位处理。

进制字符转换

十六进制中，数字 0 到 9 用字符 `0` 到 `9` 表示，数值 10 到 15 常用 `A` 到 `F` 表示。

字符转数值可以写成：

```cpp
int val(char c) {
    if ('0' <= c && c <= '9') {
        return c - '0';
    }

    return c - 'A' + 10;
}
```

数值转字符可以写成：

```cpp
char dig(int x) {
    if (x < 10) {
        return x + '0';
    }

    return x - 10 + 'A';
}
```

这类函数在程序阅读中经常出现。它们不是普通的大小写转换，而是在处理不同进制下的数字表示。

例题 9：十六进制字符

求 `val('C')` 和 `dig(15)` 的结果。

`'C' - 'A' = 2`，所以 `val('C') = 2 + 10 = 12`。

15 大于等于 10，所以 `dig(15) = 15 - 10 + 'A'`，得到 `'F'`。

编码与解码

编码类程序常成对出现：一个函数负责把原字符串变成编码串，另一个函数负责还原。

例如一个简单规则：每个小写字母向后移动 k 位。编码时加 k，解码时减 k。

```
char enc(char c, int k) {
    return (c - 'a' + k) % 26 + 'a';
}

char dec(char c, int k) {
    return (c - 'a' - k + 26) % 26 + 'a';
}
```

若编码规则是可逆的，解码应能还原原串。程序阅读时，可以用一个短字符串手算一遍，检查两个函数是否互为逆过程。

例题 10：还原编码

某编码规则把小写字母向后移动 2 位。原串 `"az"` 编码后是什么？再解码能否还原？

`a` 向后移动 2 位得到 `c`，`z` 向后移动 2 位得到 `b`。编码结果为 `"cb"`。

解码时每个字符向前移动 2 位。`c` 变回 `a`，`b` 变回 `z`，可以还原 `"az"`。

游程编码

游程编码是一类简单压缩方法。它把连续相同字符合并成“字符 + 次数”。

例如 `"aaabbc"` 可以编码成 `"a3b2c1"`。连续三个 `a` 写成 `a3`，连续两个 `b` 写成 `b2`，连续一个 `c` 写成 `c1`。

代码如下：

```
string encode(string s) {
    string t;

    for (int i = 0; i < s.size(); ) {
        int j = i;

        while (j < s.size() && s[j] == s[i]) {
            j++;
        }

        t += s[i];
        t += char(j - i + '0');

        i = j;
    }

    return t;
}
```

这段代码只适合连续次数不超过 9 的情况。若连续次数可能是 12，`char(12 + '0')` 不是字符串 `"12"`，而是一个别的字符。此时必须把次数转成十进制字符串。

例题 11：游程编码

输入 `s="abbccc"`，上面的程序输出什么？

第一段是一个 `a`，输出 `a1`。

第二段是两个 `b`，输出 `b2`。

第三段是三个 `c`，输出 `c3`。

所以结果为 `"a1b2c3"`。

多位次数

若连续次数可能超过 9，编码时应使用整数转字符串。

```
string encode(string s) {
    string t;

    for (int i = 0; i < s.size(); ) {
        int j = i;

        while (j < s.size() && s[j] == s[i]) {
            j++;
        }

        t += s[i];
        t += to_string(j - i);

        i = j;
    }

    return t;
}
```

例如 `"aaaaaaaaaaaa"` 有 12 个 `a`，应编码成 `"a12"`，而不是把 12 当成一个字符。

解码时也要注意，次数可能有多位，不能只读一个数字字符。

例题 12：多位次数解码

编码串为 `"a12b3"`，表示什么？

先读到字符 `a`，后面数字部分是 `12`，表示输出 12 个 `a`。

再读到字符 `b`，后面数字部分是 `3`，表示输出 3 个 `b`。

所以解码结果是 12 个 `a` 后接 3 个 `b`。

读解码程序时，要看它是否用循环读取连续数字。若只写 `cnt = s[i] - '0'`，就只能处理一位次数。

解码程序

下面是一段支持多位次数的解码程序。假设编码串格式为“字母 + 正整数次数”反复出现。

```
string decode(string s) {
    string t;

    for (int i = 0; i < s.size(); ) {
        char c = s[i];
        i++;

        int cnt = 0;

        while (i < s.size() && '0' <= s[i] && s[i] <= '9') {
            cnt = cnt * 10 + (s[i] - '0');
            i++;
        }

        for (int j = 1; j <= cnt; j++) {
            t += c;
        }
```

```
    }

    return t;
}
```

这段程序的关键有两个。

第一，读完字符后，后面可能有多位数字，所以要用 `while` 把连续数字全部读完。

第二，读数字时使用 `cnt = cnt * 10 + (s[i] - '0')`，和数字字符串转整数是同一个过程。

例题 13：追踪解码

执行 `decode("x2y10")`，输出是什么？

先读字符 `x`，后面数字为 `2`，输出两个 `x`。

再读字符 `y`，后面数字为 `10`，输出十个 `y`。

所以结果是 `"xxyyyyyyyyyy"`，其中有 2 个 `x` 和 10 个 `y`。

注意 `10` 是一个整体次数，不是分别输出 1 个和 0 个。

分隔符编码

有些编码规则用分隔符把不同部分隔开。例如 `"abc#12#xy"` 中，`#` 用来分隔字段。

处理这类字符串时，常见方法是扫描字符，把当前字段暂存在 `cur` 中；遇到分隔符时，把 `cur` 作为一个完整字段处理，然后清空。

```
vector<string> split(string s) {
    vector<string> res;
    string cur;

    for (int i = 0; i < s.size(); i++) {
        if (s[i] == '#') {
            res.push_back(cur);
            cur.clear();
        } else {
            cur += s[i];
        }
    }

    res.push_back(cur);
    return res;
}
```

最后循环结束后还要把 `cur` 加入答案，因为最后一个字段后面不一定有分隔符。

例题 14：分隔符扫描

执行 `split("ab#cde#f")`，得到哪些字段？

扫描到第一个 `#` 前，`cur="ab"`，加入答案。

扫描到第二个 `#` 前，`cur="cde"`，加入答案。

循环结束后，`cur="f"`，还要加入答案。

所以字段为 `"ab"`、`"cde"`、`"f"`。

若忘记最后的 `res.push_back(cur)`，就会漏掉最后一个字段。

编码类程序阅读顺序

编码类程序没有固定模板，但可以按一个稳定顺序读。

第一，看输入是什么。它是普通字符串、数字字符串、带分隔符的字符串，还是已经编码过的字符串。

第二，看字符怎样分类。程序是否判断了数字字符、小写字母、大写字母、分隔符或其他字符。

第三，看字符和数字怎样转换。出现 `c - '0'`、`c - 'a'`、`x + '0'`、`x + 'A'` 时，要立刻写出它们的含义。

第四，看循环指针怎样移动。编码类程序常用 `i` 和 `j` 扫描一段连续字符。若内层 `while` 已经移动了 `j`，外层通常要令 `i = j`。

第五，看输出字符串怎样构造。`t += c` 是追加一个字符；`t += s` 是追加一个字符串；`t = c` 或 `t = s` 则是覆盖原内容。

第六，看是否有反转。整数转字符串、某些栈式编码、从后往前构造的程序，最后常需要 `reverse`。

程序阅读：混合大小写转换

看下面代码：

```
string f(string s) {
    for (int i = 0; i < s.size(); i++) {
        if ('a' <= s[i] && s[i] <= 'z') {
            s[i] = s[i] - 'a' + 'A';
        } else if ('A' <= s[i] && s[i] <= 'Z') {
            s[i] = s[i] - 'A' + 'a';
        }
    }

    return s;
}
```

这段程序会把小写字母转成大写字母，把大写字母转成小写字母，其他字符不变。

例题 15：追踪大小写程序

输入 `s="aB3z"`，函数 `f(s)` 返回什么？

`a` 是小写字母，转成 `A`。

`B` 是大写字母，转成 `b`。

`3` 不是字母，不变。

`z` 是小写字母，转成 `Z`。

所以返回 `"Ab3Z"`。

程序阅读：分组编码

看下面代码：

```
string f(string s) {
    string t;
```

```
    for (int i = 0; i < s.size(); ) {
        int j = i;

        while (j < s.size() && s[j] == s[i]) {
            j++;
        }

        t += s[i];
        t += to_string(j - i);

        i = j;
    }

    return t;
}
```

这段程序按连续相同字符分组，并输出每组字符和组长。它是支持多位次数的游程编码。

例题 16：追踪分组编码

输入 `s="aaabbcccc"`，函数返回什么？

第一组是三个 `a`，输出 `a3`。

第二组是两个 `b`，输出 `b2`。

第三组是四个 `c`，输出 `c4`。

所以返回 `"a3b2c4"`。

若输入 `"aaaaaaaaaa"`，有 10 个 `a`，返回 `"a10"`，因为程序使用了 `to_string`。

完善程序中的常见空缺

编码类完善程序常在字符分类、数值转换、循环移动和结果追加处留空。

字符转数字常见空缺如下：

空缺位置	应关注的问题
数字字符转整数	`s[i] - '0'`
整数转数字字符	`x + '0'`
小写字母编号	`s[i] - 'a'`
大写字母编号	`s[i] - 'A'`
编号转小写字母	`id + 'a'`
编号转大写字母	`id + 'A'`

循环移位常见空缺如下：

空缺位置	应关注的问题
向后移位	`(c - 'a' + k) % 26 + 'a'`
向前移位	`(c - 'a' - k + 26) % 26 + 'a'`
大写字母	把 `'a'` 换成 `'A'`
k 很大	先令 `k %= 26`

分组编码常见空缺如下：

空缺位置	应关注的问题
找连续段右端	`while (j < n && s[j] == s[i]) j++`
组长	`j - i`
追加字符	`t += s[i]`
追加多位次数	`t += to_string(j - i)`
外层移动	`i = j`

解码常见空缺如下：

空缺位置	应关注的问题
读取次数	连续数字要整体读入
数字累积	`cnt = cnt * 10 + (s[i] - '0')`
输出重复字符	循环追加 `cnt` 次
指针移动	读字符后、读数字后都要移动

例题 17：补全循环移位

下面代码把小写字母向后移动 k 位：

```
char move(char c, int k) {
    return ________;
}
```

空缺处应填：

```
(c - 'a' + k) % 26 + 'a'
```

先用 `c - 'a'` 得到 0 到 25 的编号，再加上 k，对 26 取模后转回小写字母。

例题 18：补全多位次数读取

下面代码从位置 `i` 开始读取连续数字：

```
int cnt = 0;

while (i < s.size() && '0' <= s[i] && s[i] <= '9') {
    cnt = ________;
    i++;
}
```

空缺处应填：

```
cnt * 10 + (s[i] - '0')
```

每读入一位新数字，原来的数要乘以 10，再加上当前数字值。

常见错误

第一，把字符数字和整数数字混淆。`'7'` 是字符，7 是整数；两者之间要通过 `'0'` 转换。

第二，逐位处理和整体处理混淆。对字符串 `"109"` 逐位变换，不等于对整数 109 做同样运算。

第三，忘记处理循环移位中的负数。向前移动时要加 26 或先处理 k。

第四，把大小写转换写反。小写转大写用 `c - 'a' + 'A'`，大写转小写用 `c - 'A' + 'a'`。

第五，游程编码中只支持一位次数。若连续次数可能超过 9，必须用 `to_string`，解码时也要读多位数字。

第六，分隔符扫描后忘记加入最后一段。最后一个字段后面通常没有分隔符。

第七，外层指针没有跳到内层扫描结束位置。分组编码中，内层 `j` 已经找到下一段开头，外层应令 `i = j`。

第八，追加和覆盖混淆。`t += c` 是追加，`t = c` 会覆盖已有内容。

第九，没有看输入保证。若代码只处理小写字母，题目却可能输入大写或数字，结果含义会变化。

小结

编码类程序阅读的核心是追踪“字符、数字、位置、结果串”四类信息。字符可以通过 `'0'`、`'a'`、`'A'` 转成对应数字编号，数字也可以转回字符。循环移位、逐位编码、进制字符转换、游程编码和分隔符扫描，都是这些基本操作的组合。

本章需要重点记住：

- `s[i] - '0'` 把数字字符转成整数；
- `x + '0'` 把 0 到 9 的整数转成数字字符；
- `c - 'a'` 和 `c - 'A'` 得到字母编号；
- 循环移位要对 26 取模，向前移动要处理负数；
- 数字字符串转整数时，递推是 `x = x * 10 +` 当前数字；
- 整数转字符串常需要反转；
- 游程编码要注意连续段长度是否可能超过 9；
- 解码多位次数时，要用循环读取连续数字；
- 分隔符扫描后，不要漏掉最后一个字段；
- 程序阅读时先判断它在处理字符、数字、段，还是字段。

下一章进入相邻交换与逆序思想。那一章会从序列中两个元素的相对顺序出发，讲为什么“逆序对”可以衡量相邻交换次数，并分析相关程序阅读题。

0x27 相邻交换与逆序思想

第 38 章讲的是编码类程序阅读。本章进入第六编“字符串与序列”的最后一章，讨论序列中一个很常见的思想：逆序。

逆序思想常和相邻交换放在一起考。一个序列如果要变成升序，哪些数对的相对顺序是反的？每次相邻交换会改变多少个逆序对？为什么冒泡排序的交换次数等于逆序对数量？这些问题看起来像排序，实际考的是“相对顺序”与“局部交换”的关系。

CSP-S 第一轮中，这类内容常出现在选择题、程序阅读和完善程序里。题目可能不会直接说“逆序对”，而是给一段冒泡排序、归并排序、插入排序或相邻交换程序，让学生判断交换次数、输出序列或时间复杂度。

逆序对

给定一个序列 $a_1, a_2, \ldots, a_n$，若存在一对下标 $i < j$，但 $a_i > a_j$，那么 (i, j) 就是一对逆序对。

“逆序”的意思是：下标顺序是 i 在前、j 在后，但数值大小却是前面的数更大。它们和升序排列的要求相反。

例如序列 $3, 1, 2$ 中：

- 3 和 1 构成逆序对；
- 3 和 2 构成逆序对；
- 1 和 2 不构成逆序对。

所以逆序对数量为 2。

如果一个序列已经升序排列，就没有逆序对。如果一个长度为 n 的序列严格降序排列，则任意两个位置都构成逆序对，逆序对数量为 $\frac{n(n-1)}{2}$。

例题 1：数逆序对

求序列 $4, 1, 3, 2$ 的逆序对数量。

逐个看：

- 4 比后面的 $1, 3, 2$ 都大，形成 3 对；
- 1 后面没有比它小的数，形成 0 对；
- 3 比后面的 2 大，形成 1 对。

总数为 $3 + 0 + 1 = 4$。

逆序对分别是 $(4, 1)$、$(4, 3)$、$(4, 2)$、$(3, 2)$。这里为了说明方便写成数值对；严格说，逆序对是下标对。

逆序对看相对顺序

逆序对不是只看相邻元素。

例如序列 $5, 1, 2, 3, 4$ 中，5 和 $1, 2, 3, 4$ 都构成逆序对。虽然 5 只和 1 相邻，但它和后面每个比它小的数相对顺序都反了。

这点很重要。相邻交换每次只交换相邻的两个元素，但它最终要解决的，是所有元素对之间的相对顺序。

例题 2：不相邻也能逆序

序列 $2,3,4,1$ 中有多少个逆序对？

2 和 1 构成逆序对，3 和 1 构成逆序对，4 和 1 构成逆序对。

其他数对都没有逆序。总数为 3。

这些逆序对中，只有 4 和 1 在初始序列中相邻。2 和 1、3 和 1 虽然不相邻，也同样是逆序对。

相邻交换

相邻交换指每次只能交换两个相邻位置的元素。

例如序列 $3,1,2$，交换前两个元素，得到 $1,3,2$；再交换后两个元素，得到 $1,2,3$。

相邻交换和任意交换不同。若允许任意两个位置交换，$3,1,2$ 可以直接交换 3 和 2 得到 $2,1,3$，这不一定沿着排序方向前进。相邻交换更像让元素一步一步穿过旁边的元素。

一个重要结论是：对一个排列进行相邻交换，每交换一对相邻元素，逆序对数量只会改变 1。

如果相邻两个元素原来是逆序的，交换后这一对逆序被消除，逆序对数量减少 1。如果相邻两个元素原来不是逆序的，交换后这一对变成逆序，逆序对数量增加 1。

为什么只改变 1？因为这次交换的两个元素只改变了它们彼此之间的相对顺序。它们和其他元素的相对顺序没有改变。

例题 3：一次相邻交换

序列 $2,1,3$ 有 1 个逆序对。交换前两个元素后变成 $1,2,3$，逆序对数量变成 0。

这次交换的是逆序相邻对 $2,1$，所以逆序对数量减少 1。

再看序列 $1,2,3$。若交换前两个元素，变成 $2,1,3$，逆序对数量从 0 变成 1。这次交换的是非逆序相邻对 $1,2$，所以逆序对数量增加 1。

最少相邻交换次数

若要把一个排列变成升序，每次只能交换相邻元素，那么最少交换次数等于初始逆序对数量。

原因可以分两步看。

第一，升序排列的逆序对数量为 0。如果初始逆序对数量为 K，每次相邻交换最多只能让逆序对数量减少 1，所以至少需要 K 次交换。

第二，可以用冒泡排序或插入排序，每次只交换相邻逆序对。这样每次交换都会让逆序对数量减少 1，直到逆序对变为 0。所以 K 次一定够。

上下界相同，最少相邻交换次数就是逆序对数量。

例题 4：最少交换次数

把序列 $4,1,3,2$ 变成升序，最少需要多少次相邻交换？

前面算过，它有 4 个逆序对。所以最少需要 4 次相邻交换。

可以验证一组操作：

$4,1,3,2 \to 1,4,3,2 \to 1,3,4,2 \to 1,3,2,4 \to 1,2,3,4$。

共 4 次。

每次都交换一对相邻逆序元素，所以每次都减少一个逆序对。

冒泡排序与逆序对

冒泡排序不断比较相邻元素，若前一个比后一个大，就交换它们。每次交换都消除一个相邻逆序对。

典型代码如下：

```
int cnt = 0;

for (int i = 1; i <= n; i++) {
    for (int j = 1; j < n; j++) {
        if (a[j] > a[j + 1]) {
            swap(a[j], a[j + 1]);
            cnt++;
        }
    }
}
```

如果程序完整执行到序列有序，那么 cnt 就等于初始序列的逆序对数量。

注意这里说的是交换次数，不是比较次数。冒泡排序的比较次数通常由循环边界决定，而交换次数取决于数据本身。

例题 5：冒泡排序交换次数

序列 $3,1,2$ 使用冒泡排序升序排列，交换次数是多少？

初始逆序对有两个：$(3,1)$ 和 $(3,2)$。所以交换次数应为 2。

运行过程：

第一轮，比较 3 和 1，交换，得到 $1,3,2$，交换次数 1。再比较 3 和 2，交换，得到 $1,2,3$，交换次数 2。

之后序列已经有序，不再发生交换。交换次数为 2。

比较次数不等于交换次数

冒泡排序中，比较次数和交换次数不同。

例如下面代码：

```
int cmp = 0, swp = 0;

for (int i = 1; i <= n - 1; i++) {
    for (int j = 1; j <= n - i; j++) {
        cmp++;

        if (a[j] > a[j + 1]) {
            swap(a[j], a[j + 1]);
            swp++;
        }
    }
}
```

比较次数固定为 $\frac{n(n-1)}{2}$，和初始序列内容无关。交换次数等于逆序对数量，和初始序列内容有关。

若序列已经升序，比较次数仍然是 $\frac{n(n-1)}{2}$，交换次数是 0。若序列严格降序，比较次数仍然是 $\frac{n(n-1)}{2}$，交换次数也是 $\frac{n(n-1)}{2}$。

例题 6：比较和交换

序列 $1,2,3,4$ 使用上面的冒泡排序代码，比较次数和交换次数分别是多少？

$n=4$，比较次数为 $3+2+1=6$。

因为序列已经升序，没有任何逆序对，所以不会发生交换，交换次数为 0。

不要看到冒泡排序就把比较次数和交换次数混在一起。

插入排序与逆序对

插入排序也和逆序对有关。它从左到右处理元素，把当前元素向左移动到合适位置。

常见代码如下：

```
int cnt = 0;

for (int i = 2; i <= n; i++) {
    int j = i;

    while (j > 1 && a[j - 1] > a[j]) {
        swap(a[j - 1], a[j]);
        cnt++;
        j--;
    }
}
```

每次 `while` 中的交换，也是在交换一对相邻逆序元素。所以完整排序后，`cnt` 同样等于初始逆序对数量。

插入排序也可以不显式交换，而是用移动赋值实现。此时移动次数和逆序对数量之间的关系要看代码怎样统计，不能直接套结论。

例题 7：插入排序追踪

序列 $2,4,1,3$ 用上面的相邻交换版插入排序，交换次数是多少？

先数逆序对：

- 2 和 1 构成逆序对；
- 4 和 1 构成逆序对；
- 4 和 3 构成逆序对。

总共 3 个逆序对，所以交换次数为 3。

运行时，处理 1 时，它会依次越过 4 和 2，发生 2 次交换；处理 3 时，它会越过 4，发生 1 次交换。总共 3 次。

归并计数

逆序对也可以用归并排序思想计数。第一轮中常见考法是阅读一段归并程序，判断 `ans += mid - i + 1` 的含义。

在归并两个有序段时，设左段为 `a[l..mid]`，右段为 `a[mid+1..r]`，两个段内部都已经升序。若当前左段指针为 `i`，右段指针为 `j`。

如果 `a[i] <= a[j]`，说明 `a[i]` 不会和当前右段元素形成逆序，取左段元素。

如果 `a[i] > a[j]`，由于左段已经有序，`a[i],a[i+1],...,a[mid]` 都大于 `a[j]`。这些元素都在原序列中位于 `a[j]` 左边，因此它们分别和 `a[j]` 构成逆序对，数量为 $mid-i+1$。

所以此时要写：

```
ans += mid - i + 1;
```

例题 8：归并时计数

左段为 $2,5,8$，右段为 $1,6$。归并时，第一次比较 2 和 1，因为 $2>1$，会增加多少个逆序对？

左段当前从 2 开始，后面还有 $5,8$。由于左段有序，$2,5,8$ 都大于 1。

因此 1 和左段剩余 3 个元素分别形成逆序对，增加 3 个。

这对应公式 $mid-i+1$。

归并排序计数代码

下面是常见的归并计数代码：

```
long long ans;
int a[N], tmp[N];

void solve(int l, int r) {
    if (l == r) {
        return;
    }

    int mid = (l + r) / 2;

    solve(l, mid);
    solve(mid + 1, r);

    int i = l, j = mid + 1, k = l;

    while (i <= mid && j <= r) {
        if (a[i] <= a[j]) {
            tmp[k++] = a[i++];
        } else {
            tmp[k++] = a[j++];
            ans += mid - i + 1;
        }
    }

    while (i <= mid) {
        tmp[k++] = a[i++];
    }

    while (j <= r) {
        tmp[k++] = a[j++];
    }

    for (int p = l; p <= r; p++) {
        a[p] = tmp[p];
    }
}
```

这段程序做了两件事：一边把区间排成升序，一边统计逆序对数量。最后 ans 就是整个序列的逆序对数量。

注意比较条件使用 a[i] <= a[j]。如果两个元素相等，不应算作逆序对，因为逆序对要求 $a_i > a_j$。若写成 a[i] < a[j]，相等元素可能被错误计数。

例题 9：相等元素不算逆序

序列 $2, 2, 1$ 有多少个逆序对？

两个 2 之间不构成逆序对，因为它们相等，不满足前面的数大于后面的数。

第一个 2 和 1 构成逆序对，第二个 2 和 1 也构成逆序对。

所以共有 2 个逆序对。

归并计数时，若左段当前元素等于右段当前元素，应优先取左段元素，不增加逆序对。

从一个排列变到另一个排列

有些题不是问把序列变成升序，而是问通过相邻交换把一个排列变成另一个排列，最少需要多少次。

如果两个序列包含相同且互不重复的元素，可以把目标序列中的每个元素映射成它的位置，然后把初始序列转成位置序列。问题就变成：把这个位置序列变成升序，最少需要多少次相邻交换。答案仍然是逆序对数量。

例如初始序列为 $A = [b, a, c]$，目标序列为 $B = [a, b, c]$。在目标序列中，a 的位置是 1，b 的位置是 2，c 的位置是 3。把 A 转成位置序列，得到 $[2, 1, 3]$。它有 1 个逆序对，所以最少需要 1 次相邻交换。

例题 10：排列变换

初始序列为 $A = [3, 1, 2]$，目标序列为 $B = [1, 2, 3]$。最少需要多少次相邻交换？

目标序列中，1 的位置是 1，2 的位置是 2，3 的位置是 3。把初始序列 $[3, 1, 2]$ 转成位置序列 $[3, 1, 2]$。

这个位置序列的逆序对有 $(3, 1)$ 和 $(3, 2)$，共 2 个。

所以最少需要 2 次相邻交换。

如果目标序列不是升序，也可以用同样方法。关键是把 “目标顺序” 转成位置编号。

有重复元素时要小心

若序列中有重复元素，“把一个序列变成另一个序列” 的相邻交换次数会更复杂。因为相同值之间可以互换身份，不同匹配方式可能导致不同的逆序对数量。

例如初始序列和目标序列都有两个相同的 `a`，这两个 `a` 本身无法区分。通常要按出现顺序给相同元素编号：第一个 `a`、第二个 `a`，这样才能把问题转成排列。

CSP-S 第一轮中，若题目涉及重复元素，一般会在程序中体现如何处理。不要直接把相同值当成同一个位置，否则会丢失信息。

例题 11：重复元素编号

初始序列为 `a b a`，目标序列为 `a a b`。如何把它转成位置序列？

目标序列中，第一个 `a` 在位置 1，第二个 `a` 在位置 2，`b` 在位置 3。

初始序列从左到右：

- 第一个 `a` 对应目标中的第一个 `a`，位置 1；
- `b` 对应位置 3；
- 第二个 `a` 对应目标中的第二个 `a`，位置 2。

所以位置序列为 $1, 3, 2$，逆序对数量为 1。

这表示最少需要一次相邻交换：`a b a` 交换后两个字符，变成 `a a b`。

奇偶性

每次相邻交换都会改变逆序对数量的奇偶性。因为逆序对数量每次变化 1，奇偶性必然翻转。

因此，如果一个排列的逆序对数量是偶数，那么经过偶数次相邻交换可以变成升序；如果逆序对数量是奇数，那么经过奇数次相邻交换可以变成升序。

这类奇偶性常出现在判断题中。对于排列来说，任意交换两个元素也会改变排列奇偶性，但本章只讨论相邻交换视角。

例题 12：交换次数奇偶

序列 $3,2,1$ 的逆序对数量是多少？把它变成升序的相邻交换次数奇偶性如何？

$3,2,1$ 是长度为 3 的降序排列，逆序对有 $(3,2)$、$(3,1)$、$(2,1)$，共 3 个。

所以最少相邻交换次数为 3，是奇数。

冒泡排序程序阅读

看下面代码：

```
int cnt = 0;

for (int i = 1; i <= n - 1; i++) {
    for (int j = 1; j <= n - i; j++) {
        if (a[j] > a[j + 1]) {
            swap(a[j], a[j + 1]);
            cnt++;
        }
    }
}
```

这段代码最终会把数组排成升序。`cnt` 统计的是交换次数，也就是初始逆序对数量。

若题目问比较次数，则要看两个循环边界。内层执行次数为 $(n-1)+(n-2)+\cdots+1=\frac{n(n-1)}{2}$。

若题目问最终数组，则输出升序数组。若题目问中间某一轮后的数组，需要逐轮模拟，不要直接跳到最终状态。

例题 13：模拟一轮冒泡

序列 $4,1,3,2$，执行冒泡排序第一轮内层循环后，数组变成什么？

第一轮 `j` 从 1 到 3。

比较 4 和 1，交换，得到 $1,4,3,2$。

比较 4 和 3，交换，得到 $1,3,4,2$。

比较 4 和 2，交换，得到 $1,3,2,4$。

第一轮后，最大值 4 被放到最后，数组为 $1,3,2,4$。

归并计数程序阅读

看下面关键片段：

```
if (a[i] <= a[j]) {
    tmp[k++] = a[i++];
```

```
} else {
    tmp[k++] = a[j++];
    ans += mid - i + 1;
}
```

读这段代码时，要先知道左右两段已经分别有序。

当 `a[i] > a[j]` 时，左段从 `i` 到 `mid` 的所有元素都大于 `a[j]`。这些元素都在原序列中位于 `a[j]` 左边，因此它们和 `a[j]` 形成逆序对。

所以增加的是 `mid - i + 1`，不是 1。

例题 14：补全归并计数

下面代码中空缺处应填什么？

```
if (a[i] <= a[j]) {
    tmp[k++] = a[i++];
} else {
    tmp[k++] = a[j++];
    ans += ________;
}
```

空缺处应填 `mid - i + 1`。

因为左段是有序的，若当前 `a[i]` 已经大于 `a[j]`，则 `a[i]` 到 `a[mid]` 全部大于 `a[j]`，数量为 $mid - i + 1$。

相邻交换与稳定性

如果排序时只在 `a[j] > a[j+1]` 时交换，那么相等元素不会交换。这种排序是稳定的：相等元素的相对顺序保持不变。

如果条件写成 `a[j] >= a[j+1]`，相等元素也会交换，排序结果仍然可能有序，但相等元素的原相对顺序会改变，交换次数也会增加。

逆序对定义要求 $a_i > a_j$，相等元素不算逆序。因此冒泡排序若用 `>=` 交换，交换次数就不再等于逆序对数量。

例题 15：相等元素交换

序列 $2, 2, 1$。若冒泡排序条件是 `a[j] > a[j+1]`，交换次数是多少？

逆序对数量为 2，所以交换次数为 2。

若条件改成 `a[j] >= a[j+1]`，两个相等的 2 也可能发生交换，交换次数会增加。此时不能用逆序对数量直接判断交换次数。

完善程序中的常见空缺

相邻交换与逆序思想的完善程序，常在比较条件、计数位置、归并统计和循环边界处留空。

冒泡排序常见空缺如下：

空缺位置	应关注的问题
比较条件	升序时通常是 `a[j] > a[j+1]`
交换操作	`swap(a[j], a[j+1])`
交换计数	只在发生交换时 `cnt++`
内层边界	常见为 `j <= n - i`

逆序对朴素统计常见空缺如下：

空缺位置	应关注的问题
下标关系	`i < j`
数值关系	`a[i] > a[j]`
计数变量	满足条件时加 1

归并计数常见空缺如下：

空缺位置	应关注的问题
左右区间	左段 `l..mid`，右段 `mid+1..r`
比较条件	`a[i] <= a[j]` 时取左段
计数公式	`ans += mid - i + 1`
复制回原数组	合并后要把 `tmp[l..r]` 复制回 `a[l..r]`

例题 16：补全朴素逆序对统计

下面代码统计逆序对数量：

```
int ans = 0;

for (int i = 1; i <= n; i++) {
    for (int j = i + 1; j <= n; j++) {
        if (__________) {
            ans++;
        }
    }
}
```

空缺处应填 `a[i] > a[j]`。

外层和内层已经保证 $i < j$，只需要判断前面的数是否大于后面的数。

例题 17：补全冒泡计数

下面代码统计冒泡排序交换次数：

```
if (a[j] > a[j + 1]) {
    ________;
    cnt++;
}
```

空缺处应填 `swap(a[j], a[j + 1])`。

只有真的交换相邻元素，`cnt++` 才表示交换次数。如果只计数不交换，后面的比较过程就不再是冒泡排序过程。

常见错误

第一，把逆序对只理解成相邻逆序。逆序对只要求下标 $i < j$，不要求相邻。

第二，把比较次数当成交换次数。冒泡排序比较次数由循环边界决定，交换次数由逆序对数量决定。

第三，认为任意交换次数也等于逆序对数量。逆序对数量对应的是最少相邻交换次数，不是任意交换次数。

第四，相等元素算成逆序。逆序对要求 $a_i > a_j$，相等不算。

第五，归并计数时写成 `ans++`。当 `a[i]>a[j]` 时，左段剩余所有元素都和 `a[j]` 构成逆序，增加的是 $mid - i + 1$。

第六，归并后忘记复制回原数组。若不复制，上一层归并拿到的不是有序段，后续计数会错。

第七，把一个排列变到另一个排列时，没有先映射目标位置。目标序列不是升序时，不能直接数原序列的大小逆序。

第八，有重复元素时没有区分相同值的出现次序。相同元素若要映射位置，应按出现顺序匹配。

第九，冒泡排序用 `>=` 交换后仍把交换次数当成逆序对数量。相等元素被交换时，交换次数会多出来。

小结

逆序对描述的是序列中相对顺序反了的元素对。若 $i < j$ 且 $a_i > a_j$，就形成一对逆序对。逆序对不要求两个元素相邻。

相邻交换每次只改变被交换的两个元素之间的相对顺序，因此逆序对数量只改变 1。把一个排列变成升序所需的最少相邻交换次数，等于初始逆序对数量。冒泡排序和相邻交换版插入排序中，交换次数都与逆序对数量直接相关。

本章需要重点记住：

- 逆序对要求 $i < j$ 且 $a_i > a_j$；
- 相等元素不构成逆序对；
- 逆序对可以是不相邻的元素对；
- 最少相邻交换次数等于逆序对数量；
- 冒泡排序交换次数等于逆序对数量，比较次数不等于交换次数；
- 归并计数中，若 `a[i] > a[j]`，应增加 $mid - i + 1$；
- 一个排列变到另一个排列，要先映射目标位置；
- 有重复元素时，要按出现顺序区分相同值；
- `>` 和 `>=` 的差别会影响交换次数和稳定性。

到这里，第六编 “字符串与序列” 结束。下一编进入 “DP、贪心与搜索”，从 DP 的状态与转移开始，系统讲第一轮中最常见的动态规划程序阅读和完善程序结构。

第七编 DP、贪心与搜索

0x28 DP 的状态与转移

第 39 章结束了第六编 “字符串与序列”。从本章开始进入第七编 “DP、贪心与搜索”。

CSP-S 第一轮中的 DP，很多时候不要求写出很长的代码。它更常考：

- 一个递推式算几项；
- 一个 DP 数组表示什么；
- 某个初始化是否正确；
- 某个循环顺序能否保证转移来源已经算好；
- 程序最终输出的是哪个状态；
- 把朴素递归改成 DP 后，复杂度为什么下降；
- 完善程序中某个空缺处应填哪个状态或转移。

如果只记几个模型名，遇到程序阅读题还是会卡住。第一轮要重点训练的是：把代码中的数组、下标和循环读成 “状态与转移”。

状态

DP 程序里，最重要的数组通常叫 `f`、`dp`、`g` 或类似名字。状态就是这个数组中一个元素的含义。

例如：

```
f[i]
```

可能表示前 i 个数的答案，也可能表示走到第 i 个位置的方案数，还可能表示长度为 i 的某种序列数量。只看到 `f[i]` 还不够，必须结合转移和初始化判断。

再看：

```
dp[i][j]
```

它通常表示两个维度共同决定的状态。常见含义有：

状态	常见含义
`dp[i][j]`	前 i 个物品、容量为 j 的最优值
`dp[i][j]`	第一个串前 i 个字符、第二个串前 j 个字符的答案
`dp[i][j]`	走到格子 (i, j) 的方案数或最优值
`dp[i][j]`	处理到第 i 步，某个附加状态为 j 的答案

状态不是数组长什么样，而是数组下标代表了什么信息。

一个好的状态必须把 “以后还会用到的信息” 记住。若信息记少了，不同情况被混在同一个数组元素里，后续转移会错。若信息记多了，状态数量变大，程序也会更复杂。

例题 1：看懂状态含义

有如下代码片段：

```
f[1] = 1;
f[2] = 2;

for (int i = 3; i <= n; i++) {
    f[i] = f[i - 1] + f[i - 2];
}
```

问：`f[i]` 更可能表示什么？

这段代码中，`f[i]` 只依赖 `f[i-1]` 和 `f[i-2]`，并且从小到大计算。它适合表示“规模为 i 的问题答案”。

例如，走楼梯时每次可以走 1 级或 2 级，`f[i]` 可以表示走到第 i 级的方法数。最后一步若走 1 级，来自第 $i-1$ 级；最后一步若走 2 级，来自第 $i-2$ 级，所以有 `f[i] = f[i - 1] + f[i - 2]`。

注意，不能只看变量名判断含义。真正决定含义的是初始化、转移和题意之间的对应关系。

转移

转移描述的是一个状态怎样由其他状态推出。

例如：

```
f[i] = f[i - 1] + f[i - 2];
```

它表示 `f[i]` 由前两个规模的答案合并得到。

再看：

```
dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
```

它表示当前位置的答案来自上方或左方的较大值。

转移通常回答三个问题：

问题	例子
从哪些状态来	`f[i-1]`、`f[i-2]`
怎样合并	相加、取最大、取最小、取或
为什么覆盖所有情况	最后一步、最后一个元素、最后一次选择

在第一轮中，看转移时不要只背公式。要问自己：这个转移是在分类，还是在取最优？如果是分类，分类之间能不能重复？如果是取最优，候选状态是否都合法？

例题 2：递推式手算

已知 $f(1)=1$，当 $n \geq 2$ 时，$f(n)=f(n-1)+f(\lfloor n/2 \rfloor)$。求 $f(4)$。

按顺序计算：

n	计算	结果
1	已知	1
2	$f(1)+f(1)$	2
3	$f(2)+f(1)$	3
4	$f(3)+f(2)$	5

所以 $f(4)=5$。

这类题不难在公式本身，而在下标。$\lfloor n/2 \rfloor$ 要先取整，再代入已有结果。

状态要记够

看一个简单问题：

给定一个整数序列，选若干个数，要求不能选相邻位置，求最大和。

若用 `f[i]` 表示前 i 个数中满足条件的最大和，可以这样转移：

- 不选第 i 个数，答案是 `f[i-1]`；
- 选第 i 个数，则第 $i-1$ 个数不能选，答案是 `f[i-2] + a[i]`。

所以：

```
f[i] = max(f[i - 1], f[i - 2] + a[i]);
```

这里 `f[i]` 足够用，因为处理前 i 个数时，后续只需要知道前缀的最优结果，不需要知道具体选了哪些位置。

但如果题目再加一个限制："必须恰好选 k 个数"，`f[i]` 就不够了。因为同样是前 i 个数的最大和，选了 1 个数和选了 3 个数，对后续影响不同。此时需要增加一维：

```
dp[i][j]
```

表示前 i 个数中恰好选 j 个数的最大和。

这就是状态设计中的关键：题目中哪些信息会影响后续选择，就要放进状态里。

例题 3：状态是否足够

有 n 个任务，每个任务可以选或不选。要求选出的任务总耗时不超过 m，求最大收益。若用 `f[i]` 表示前 i 个任务的最大收益，够不够？

不够。

只知道前 i 个任务的最大收益，无法知道已经用了多少时间。后面还能不能选新任务，取决于剩余时间。因此需要把 "已用时间" 或 "容量" 放进状态，例如 `dp[i][j]` 表示前 i 个任务、耗时不超过 j 的最大收益。

这类状态会在下一章背包中系统讲。本章先记住判断方法：若某个量会限制后面的选择，就不能丢掉它。

初始化

初始化告诉程序哪些最小状态已经知道。

例如走楼梯：

```
f[1] = 1;
f[2] = 2;
```

这两个值是后续递推的起点。若它们错了，后面的值都会错。

再看计数类 DP：

```
dp[0] = 1;
```

这类初始化经常表示 "什么都不选，也是一种方案"。学生容易把它误写成 0。如果 `dp[0]` 作为后续方案数的来源，写成 0 会让所有状态都无法累加出答案。

最优化类 DP 的初始化常分两类：

- 求最大值时，不合法状态常设为很小的数；
- 求最小值时，不合法状态常设为很大的数。

例如：

```
const int INF = 1e9;

for (int i = 0; i <= n; i++) {
    f[i] = INF;
}

f[0] = 0;
```

这里 `f[0]=0` 是合法起点，其他状态暂时不可达。后面用转移逐步更新。

如果所有状态初值都设成 0，在求最小值时就可能把不可达状态误当作合法答案。

例题 4：为什么 dp[0]=1

有一个程序统计若干种方案数：

```
dp[0] = 1;

for (int i = 1; i <= n; i++) {
    for (int j = s; j >= a[i]; j--) {
        dp[j] += dp[j - a[i]];
    }
}
```

问：`dp[0]=1` 的含义是什么？

`dp[j]` 表示凑出和为 j 的方案数。凑出和为 0 的方案是“什么都不选”，这是一种空方案。它不是最终一定要输出的方案，但它是后续转移的起点。

若 `dp[0]=0`，则选择第一个数时，`dp[a[i]]` 无法从 `dp[0]` 得到贡献，整个计数会少掉所有从空方案开始扩展出来的情况。

循环顺序

DP 的循环顺序必须保证：计算一个状态时，它依赖的状态已经算好。

对于 `f[i] = f[i - 1] + f[i - 2]`，要从小到大计算 i。如果从大到小算，`f[i-1]` 和 `f[i-2]` 可能还没有正确结果。

对于二维 DP，循环顺序要看转移来源。

例如：

```
dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
```

计算 `dp[i][j]` 时，需要上方 `dp[i-1][j]` 和左方 `dp[i][j-1]`。只要按 `i` 从小到大、`j` 从小到大，就能保证这两个状态已经算好。

代码如下：

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= m; j++) {
        dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
```

```
    }
}
```

如果转移用到的是右方或下方，循环顺序就要相应调整。程序阅读题中，循环方向经常是判断状态含义的重要线索。

例题 5：判断循环方向

下面代码中，`f[i]` 的计算顺序是否合理？

```
f[0] = 0;
f[1] = 1;

for (int i = n; i >= 2; i--) {
    f[i] = f[i - 1] + f[i - 2];
}
```

不合理。

当从 n 往 2 计算时，`f[i-1]` 和 `f[i-2]` 对于较大的 i 还没有被正确计算出来。只有 `f[0]` 和 `f[1]` 已知，应该从 2 到 n 递推。

正确循环应为：

```
for (int i = 2; i <= n; i++) {
    f[i] = f[i - 1] + f[i - 2];
}
```

答案位置

DP 程序中，最终答案不一定就是数组最后一个元素。要看状态含义。

若 `f[i]` 表示前 i 个元素的答案，最终常是 `f[n]`。

若 `dp[i][j]` 表示走到格子 (i, j) 的答案，最终常是 `dp[n][m]`。

若 `dp[i][j]` 表示处理到第 i 个位置、最后状态为 j 的答案，最终可能要在所有 `j` 中取最大或求和：

```
int ans = 0;

for (int j = 0; j <= m; j++) {
    ans = max(ans, dp[n][j]);
}
```

或者：

```
long long ans = 0;

for (int j = 0; j <= m; j++) {
    ans += dp[n][j];
}
```

不能看到 `n` 就直接输出 `dp[n][m]`。第二维如果表示“恰好为 m”，答案才可能是 `dp[n][m]`；如果表示“当前状态编号”，就可能要枚举所有状态。

例题 6：答案在哪里

有如下状态：`dp[i][j]` 表示处理完前 i 个字符后，当前自动机状态为 j 的方案数。若题目要求所有合法字符串数量，最后应该输出什么？

如果所有状态 j 都是合法结束状态，就应输出所有 `dp[n][j]` 的和。

```
long long ans = 0;

for (int j = 0; j < m; j++) {
    ans += dp[n][j];
}
```

若题目只允许结束在某个特定状态，比如状态 0，答案才是 `dp[n][0]`。

所以答案位置取决于状态定义和题目要求，不是数组下标越靠后越像答案。

复杂度

DP 的时间复杂度通常可以用一句话判断：状态个数乘以每个状态的转移次数。

例如一维 DP：

```
for (int i = 1; i <= n; i++) {
    f[i] = f[i - 1] + f[i - 2];
}
```

状态数为 n，每个状态常数次转移，时间复杂度为 $O(n)$。

二维 DP：

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= m; j++) {
        dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
    }
}
```

状态数为 nm，每个状态常数次转移，时间复杂度为 $O(nm)$。

若每个状态还要枚举一个决策：

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= m; j++) {
        for (int k = 1; k <= j; k++) {
            dp[i][j] = max(dp[i][j], dp[i - 1][j - k] + val[k]);
        }
    }
}
```

状态数为 nm，每个状态最多枚举 m 个决策，时间复杂度为 $O(nm^2)$。

第一轮中，复杂度选项常故意混淆“状态数”和“循环层数”。看复杂度时，先数循环范围，再看循环中是否有隐藏的查找、排序或枚举。

例题 7：复杂度判断

下面代码的时间复杂度是多少？

```
for (int i = 1; i <= n; i++) {
    for (int j = 0; j <= m; j++) {
        dp[i][j] = dp[i - 1][j];

        if (j >= w[i]) {
            dp[i][j] = max(dp[i][j], dp[i - 1][j - w[i]] + v[i]);
        }
    }
}
```

外层循环执行 n 次，内层循环执行 $m+1$ 次，每个状态只做常数次操作。时间复杂度为 $O(nm)$。

空间上使用了 `dp[n][m]` 级别的数组，空间复杂度为 $O(nm)$。

这段代码属于背包类 DP，下一章会详细讲。这里先关注复杂度判断方法。

递归与 DP

第 6 章讲过递归。很多 DP 程序可以看成“把重复递归算过的结果存下来”。

例如朴素递归计算斐波那契数：

```
int fib(int n) {
    if (n <= 1) {
        return n;
    }

    return fib(n - 1) + fib(n - 2);
}
```

计算 `fib(5)` 时，会多次计算 `fib(3)`、`fib(2)`、`fib(1)`。这些重复计算让程序变慢。

如果使用数组从小到大计算：

```
f[0] = 0;
f[1] = 1;

for (int i = 2; i <= n; i++) {
    f[i] = f[i - 1] + f[i - 2];
}
```

每个 `f[i]` 只算一次，复杂度变为 $O(n)$。

这也是第一轮常见判断点：朴素递归慢，不是因为函数调用本身多几个语句，而是因为同一个子问题被反复求解。DP 用数组保存子问题答案，避免重复计算。

例题 8：判断原因

用朴素递归计算斐波那契数列的时间复杂度很高，而用动态规划可以线性计算。根本原因是什么？

根本原因是朴素递归中存在大量重复子问题。

例如计算 `fib(6)` 时，会先算 `fib(5)` 和 `fib(4)`；而计算 `fib(5)` 时又会算 `fib(4)`。同一个 `fib(4)` 被重复计算。规模越大，重复越多。

动态规划把 `fib(0)`, `fib(1)`, ..., `fib(n)` 依次保存下来，每个状态只计算一次，所以复杂度下降。

程序阅读：一维 DP

看下面代码：

```
int solve(int n) {
    f[0] = 1;
    f[1] = 1;

    for (int i = 2; i <= n; i++) {
        f[i] = f[i - 1] + f[i - 2];
    }

    return f[n];
}
```

这段程序的状态是 `f[i]`，表示规模为 i 的方案数。初始化给出 `f[0]` 和 `f[1]`，循环从 2 开始，每次依赖前两个状态。

如果题目给 `n=4`，依次有：

i	f[i]
0	1
1	1
2	2
3	3
4	5

所以返回 5。

读一维 DP 时，最常用的追踪表只需要记录下标和值。若状态含义较复杂，再额外记录来源。

例题 9：修改初始化的影响

把上面代码中的 `f[0]=1` 改成 `f[0]=0`，当 `n=4` 时返回多少？

此时：

i	f[i]
0	0
1	1
2	1
3	2
4	3

所以返回 3。

这说明初始化不是装饰。递推式相同，初值不同，整个数列都会改变。

程序阅读：二维 DP

看下面代码：

```
for (int i = 1; i <= n; i++) {
    dp[i][1] = 1;
}

for (int j = 1; j <= m; j++) {
```

```
    dp[1][j] = 1;
}

for (int i = 2; i <= n; i++) {
    for (int j = 2; j <= m; j++) {
        dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
    }
}

cout << dp[n][m] << endl;
```

这段程序很像在统计从左上角走到 (n, m) 的路径数。因为：

- 第一列都设为 1，表示只能一直向下走；
- 第一行都设为 1，表示只能一直向右走；
- 中间格子来自上方和左方。

若 `n=3,m=3`，表格为：

dp[i][j]	$j=1$	$j=2$	$j=3$
$i=1$	1	1	1
$i=2$	1	2	3
$i=3$	1	3	6

最终输出 `dp[3][3]=6`。

例题 10：二维表追踪

上面程序中，当 `n=2,m=4` 时输出多少？

第一行全是 1，第一列全是 1。

第二行依次计算：

- `dp[2][2]=dp[1][2]+dp[2][1]=1+1=2`；
- `dp[2][3]=dp[1][3]+dp[2][2]=1+2=3`；
- `dp[2][4]=dp[1][4]+dp[2][3]=1+3=4`。

所以输出 4。

程序阅读：取最大值

有些 DP 不是统计方案数，而是求最大或最小。

看下面代码：

```
f[1] = a[1];

for (int i = 2; i <= n; i++) {
    f[i] = max(f[i - 1] + a[i], a[i]);
}

int ans = f[1];

for (int i = 2; i <= n; i++) {
    ans = max(ans, f[i]);
}
```

这里 f[i] 不是前 i 个数的最大答案，而是“必须以第 i 个数结尾”的最大连续段和。

为什么最后还要再取一遍最大值？因为最大连续段不一定以第 n 个数结尾。答案可能在任何位置结束，所以要枚举所有 f[i]。

这类“以某位置结尾”的状态非常常见。第 42 章会专门讲最大子段和，本章先强调一点：看到最后还有一轮取最大值，就要想到 f[i] 可能只是某个局部状态，不是前缀总答案。

例题 11：答案不是 f[n]

设数组为 $2, -5, 3, 4, -1$。按上面代码计算，最终答案是多少？

逐项计算：

i	$a[i]$	f[i]
1	2	2
2	-5	-3
3	3	3
4	4	7
5	-1	6

f[5]=6，但最大值出现在 f[4]=7。所以最终答案是 7，不是 f[n]。

完善程序中的常见空缺

DP 完善程序常在五类位置留空：状态初始化、转移来源、比较或累加方式、循环边界、最终答案。

初始化空缺

计数类 DP 常见：

```
dp[0] = ________;
```

如果 dp[0] 表示空方案，通常填 1。

最短路或最小代价类 DP 常见：

```
for (int i = 0; i <= n; i++) {
    f[i] = INF;
}

f[0] = ________;
```

若从状态 0 开始，常填 0。

最大值类 DP 若允许负数，要特别小心。不能随便把所有状态设为 0，否则可能把“不选任何数”误当成答案。

转移来源空缺

例如：

```
f[i] = max(f[i - 1], ________ + a[i]);
```

如果表示不能选相邻元素，那么选第 i 个时，上一项不能选，应来自 f[i-2]。空缺处应填 f[i - 2]。

再如：

```
dp[i][j] = dp[i - 1][j] + _______;
```

如果表示从左上角走到 (i, j)，只能从上方或左方来，空缺处应填 `dp[i][j - 1]`。

合并方式空缺

计数类通常用加法：

```
dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
```

求最大值用 `max`：

```
dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
```

求最小值用 `min`：

```
dp[i][j] = min(dp[i - 1][j], dp[i][j - 1]);
```

判断是否可达常用逻辑或：

```
ok[i][j] = ok[i - 1][j] || ok[i][j - 1];
```

完善程序时，不能只看两个状态长得像，还要看题目问的是数量、最大值、最小值，还是能否做到。

循环边界空缺

如果转移用到 `i-1`，循环通常从 1 或 2 开始，并要保证 `i-1` 合法。

例如：

```
for (int i = 2; i <= n; i++) {
    f[i] = f[i - 1] + f[i - 2];
}
```

若从 1 开始，就会访问 `f[-1]` 或含义错误的状态。

二维 DP 中，若第一行和第一列已经初始化，中间转移常从 2 开始：

```
for (int i = 2; i <= n; i++) {
    for (int j = 2; j <= m; j++) {
        dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
    }
}
```

若使用 `0` 开始的下标，则边界要重新对应。第一轮题中常把下标从 0 或 1 开始混在不同程序里，不能直接套模板。

答案空缺

若状态本身就是完整答案，可能输出：

```
cout << f[n] << endl;
```

若答案要在多个结束状态里取最大值，可能是：

```cpp
int ans = 0;

for (int j = 0; j <= m; j++) {
    ans = max(ans, dp[n][j]);
}
```

若答案要统计所有结束状态，可能是：

```cpp
long long ans = 0;

for (int j = 0; j <= m; j++) {
    ans += dp[n][j];
}
```

看答案位置时，要回到状态定义：第二维究竟是限制条件、当前状态，还是必须达到的目标。

真题式完善程序：不能选相邻

补全下面程序。给定 n 个整数 $a_1, a_2, \ldots, a_n$，选若干个数，不能选择相邻位置，求最大和。允许一个数都不选。

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

const int N = 1005;

int n;
int a[N], f[N];

int main() {
    cin >> n;

    for (int i = 1; i <= n; i++) {
        cin >> a[i];
    }

    f[0] = 0;
    f[1] = max(0, a[1]);

    for (int i = 2; i <= n; i++) {
        f[i] = max(__________, __________);
    }

    cout << f[n] << endl;
    return 0;
}
```

空缺处应填：

```cpp
f[i - 1], f[i - 2] + a[i]
```

完整转移为：

```cpp
f[i] = max(f[i - 1], f[i - 2] + a[i]);
```

含义是：

- 不选第 i 个数，答案继承 `f[i-1]`；

- 选第 i 个数，则第 $i-1$ 个不能选，来自 `f[i-2]+a[i]`。

因为题目允许一个数都不选，所以 `f[1]` 写成 `max(0, a[1])`。如果题目要求必须至少选一个数，初始化就不能这样写。

变式：必须至少选一个数

若题目改成“必须至少选一个数”，上面的初始化应怎样改？

此时不能让答案为空方案 0。如果 $a[1]$ 是负数，`f[1]` 也应为 `a[1]`，表示必须选一个数时，前 1 个数的最大和只能是 $a[1]$。

可以写成：

```
f[1] = a[1];
```

后续转移也要小心处理 `f[0]` 的含义。很多 DP 错误就出在“是否允许空方案”没有和初始化对应起来。

真题式程序阅读：状态压缩的影子

看下面代码：

```
int a = 0, b = 1;

for (int i = 2; i <= n; i++) {
    int c = a + b;
    a = b;
    b = c;
}

cout << b << endl;
```

这段代码没有数组 `f`，但它仍然在做递推。`a` 保存上一轮的 `f[i-2]`，`b` 保存上一轮的 `f[i-1]`，`c` 是当前的 `f[i]`。

如果 `n=5`，过程如下：

循环后	a	b
初始	0	1
$i=2$	1	1
$i=3$	1	2
$i=4$	2	3
$i=5$	3	5

最终输出 5。

这种写法叫滚动变量或状态压缩。第一轮中出现时，不要因为没有 `f[i]` 就认为不是 DP。要看变量是否在保存前几个状态。

本章只要求能读懂这种写法。背包中的一维数组优化会在下一章讲。

常见错误

第一，只看数组名，不看状态含义。`f[i]`、`dp[i][j]` 只是名字，含义要从题意、初始化和转移中判断。

第二，状态少记了信息。例如题目有限制容量，却只写前 i 个元素的最优值，就会丢掉剩余容量的信息。

第三，把计数、最优值和可达性混在一起。计数常用加法，最优值常用 `max` 或 `min`，可达性常用逻辑或。

第四，初始化写错。`dp[0]=1` 常表示空方案；最小值 DP 中，不合法状态不能随便设为 0。

第五，循环顺序不满足依赖关系。计算当前状态时，它依赖的状态必须已经计算完成。

第六，答案位置看错。有的状态表示前缀答案，可以输出 `f[n]`；有的状态表示以某位置结尾，需要再枚举取最大或求和。

第七，把滚动变量当成普通变量更新。状态压缩中，更新顺序很重要。若先覆盖了旧值，后面的转移就可能拿不到原来的状态。

第八，忽略“是否允许空方案”。允许不选和必须至少选一个，初始化和答案都可能不同。

第九，复杂度只数循环层数，不看循环范围。两层循环不一定是 $O(n^2)$，如果范围是 n 和 m，就是 $O(nm)$；若内层还枚举决策，复杂度还要再乘一项。

小结

DP 程序的核心是状态与转移。状态说明数组元素代表什么，转移说明一个状态怎样由已经知道的状态推出。读 DP 程序时，应按“状态、初始化、转移、循环顺序、答案位置、复杂度”的顺序检查。

本章需要重点记住：

- 状态不是数组名，而是数组下标所代表的信息；
- 会影响后续选择的信息，必须放进状态；
- 转移要看来源状态、合并方式和分类是否完整；
- 计数用加法，最大值用 `max`，最小值用 `min`，可达性用逻辑或；
- 初始化决定递推起点，空方案常对应 1 种方案；
- 不合法状态在最小值 DP 中常设为大值，在最大值 DP 中常设为小值；
- 循环顺序必须保证依赖状态已经算好；
- 答案不一定是 `f[n]` 或 `dp[n][m]`，要看状态定义；
- 时间复杂度通常是状态数乘以每个状态的转移次数；
- 朴素递归变慢的关键原因是重复子问题，DP 通过保存状态避免重复计算。

下一章进入背包。背包是 CSP-S 第一轮中非常适合考程序阅读和完善程序的 DP 模型，重点在状态定义、二维转移、一维优化和循环方向。

0x29 背包

背包题在 CSP-S 第一轮中常以选择题、程序阅读和完善程序出现。它不一定要求学生完整写出程序，但很喜欢考：

- `dp[i][j]` 的含义；
- 选或不选某个物品时从哪里转移；
- 容量循环为什么有时倒序、有时正序；
- `dp[0]`、`dp[0][0]` 这类初值是什么意思；
- “恰好装满” 和 “不超过容量” 的区别；
- 程序最终输出 `dp[n][m]`、`max(dp[n][j])` 还是方案数。

背包看起来像一个模型，实际考的是 DP 的基本功。

问题

最常见的是 0-1 背包。

有 n 个物品，第 i 个物品有重量 w_i 和价值 v_i。背包容量为 m。每个物品最多选一次，要求选出的物品总重量不超过 m，并使总价值最大。

“每个物品最多选一次” 是 0-1 背包的关键。对每个物品，只有两种选择：选或不选。

例如：

物品	重量	价值
1	3	5
2	4	6
3	5	10

容量为 8。可以选物品 1 和 3，总重量 8，总价值 15；也可以选物品 1 和 2，总重量 7，总价值 11。最优方案是选物品 1 和 3。

第一轮中，题目通常不会让你枚举所有方案，而是让你判断 DP 数组或某段程序的含义。

二维状态

先用二维数组来写，最容易看清楚。

设 `dp[i][j]` 表示：只考虑前 i 个物品，背包容量不超过 j 时，能够得到的最大价值。

这里有两个下标：

- i 表示已经考虑到第几个物品；
- j 表示当前允许使用的容量。

如果不考虑第 i 个物品，那么答案就是前 $i-1$ 个物品在容量 j 下的答案：

```
dp[i - 1][j]
```

如果考虑选第 i 个物品，需要先保证 $j \geq w_i$。选了它以后，前面 $i-1$ 个物品最多只能占用 $j-w_i$ 的容量，于是价值为：

```
dp[i - 1][j - w[i]] + v[i]
```

所以转移为：

```
dp[i][j] = dp[i - 1][j];

if (j >= w[i]) {
    dp[i][j] = max(dp[i][j], dp[i - 1][j - w[i]] + v[i]);
}
```

也可以写成一行：

```
dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - w[i]] + v[i]);
```

但这一行只在 $j \geq w_i$ 时合法。若 $j < w_i$，不能访问 `dp[i - 1][j - w[i]]`。

手算一张表

有 3 个物品：

物品	重量	价值
1	2	3
2	3	4
3	4	5

背包容量为 5。求最大价值。

状态 `dp[i][j]` 表示前 i 个物品、容量不超过 j 的最大价值。初始化时，不考虑任何物品，所有容量下最大价值都是 0。

考虑物品 1 后：

容量 j	0	1	2	3	4	5
`dp[1][j]`	0	0	3	3	3	3

考虑物品 2 后：

容量 j	0	1	2	3	4	5
`dp[2][j]`	0	0	3	4	4	7

容量为 5 时，可以选物品 1 和 2，价值为 $3+4=7$。

考虑物品 3 后：

容量 j	0	1	2	3	4	5
`dp[3][j]`	0	0	3	4	5	7

最终答案是 `dp[3][5]=7`。

这张表体现了背包的两个选择：不选当前物品，继承上一行；选当前物品，从上一行的较小容量转移过来。

真题式例题：容量为 20 的背包

有一个 0-1 背包，容量为 20。现有 5 个物品，它们的重量分别为 $7, 5, 4, 3, 6$，价值分别为 $15, 12, 9, 7, 13$。能获得的最大总价值是多少？

物品	重量	价值
1	7	15
2	5	12
3	4	9
4	3	7
5	6	13

若直接观察，所有物品总重量为 25，超过容量 20，所以至少要舍弃一些物品。

可以按 DP 思路算，也可以先看几种接近容量上限的组合：

选择物品	总重量	总价值
$1, 2, 3, 4$	19	43
$1, 3, 4, 5$	20	44
$2, 3, 4, 5$	18	41
$1, 2, 4, 5$	21	不合法
$1, 2, 3, 5$	22	不合法

合法方案中，$1, 3, 4, 5$ 的总重量为 $7 + 4 + 3 + 6 = 20$，总价值为 $15 + 9 + 7 + 13 = 44$。

答案是 44。

这类选择题可以用 DP，也可以用“总重量接近容量”的组合检查。但在程序阅读和完善程序中，还是要回到状态和转移。

初始化

对“总重量不超过容量，最大价值”的 0-1 背包，常用初始化很简单：

```
for (int j = 0; j <= m; j++) {
    dp[0][j] = 0;
}
```

前 0 个物品什么都不能选，最大价值为 0。

如果使用全局数组，`dp` 初始值本来就是 0，代码中常常省略这一步。读程序时要知道它的含义，不要以为没有初始化。

但如果题目改成“必须恰好装满容量 m”，初始化就不一样。

设 `dp[i][j]` 表示前 i 个物品中，恰好装出重量 j 的最大价值。此时很多状态一开始是不合法的。应该把不合法状态设成很小的数，只让 `dp[0][0]=0` 合法。

```
const int NEG = -1e9;

for (int i = 0; i <= n; i++) {
    for (int j = 0; j <= m; j++) {
        dp[i][j] = NEG;
    }
}

dp[0][0] = 0;
```

如果把所有状态都初始化为 0，程序会把“无法恰好装出重量 j”误当成“可以装出重量 j 且价值为 0”。这种错误在完善程序题中很常见。

不超过容量和恰好容量

两种题目的状态看起来很像，但答案含义不同。

题目要求	常见状态含义	初始化
总重量不超过 m，价值最大	容量上限为 j 时的最大价值	初值可为 0
总重量恰好为 m，价值最大	恰好装出重量 j 的最大价值	只有 `dp[0]=0` 合法
问能否恰好装满	是否能装出重量 j	`ok[0]=true`
问恰好装满的方案数	装出重量 j 的方案数	`cnt[0]=1`

同样是背包，题目问法不同，初始化和答案位置都会变化。

例题：初始化判断

下面代码用于判断能否从若干物品中选出一些，使总重量恰好为 m。

```
ok[0] = true;

for (int i = 1; i <= n; i++) {
    for (int j = m; j >= w[i]; j--) {
        ok[j] = ok[j] || ok[j - w[i]];
    }
}
```

这里 `ok[0]=true` 表示什么？

它表示“不选任何物品，可以凑出重量 0”。这是所有后续状态的起点。

如果没有 `ok[0]=true`，那么第一次选物品时，`ok[w[i]]` 就无法由 `ok[0]` 转移得到，所有状态都会保持不可达。

一维优化

二维 DP 中，`dp[i][j]` 只依赖上一行 `dp[i-1][...]`。因此可以把第一维省掉，用一维数组 `dp[j]` 表示当前已经处理过的物品中，容量不超过 j 的最大价值。

0-1 背包的一维写法是：

```
for (int i = 1; i <= n; i++) {
    for (int j = m; j >= w[i]; j--) {
        dp[j] = max(dp[j], dp[j - w[i]] + v[i]);
    }
}
```

内层容量必须从大到小循环。

原因是：第 i 个物品最多选一次。更新 `dp[j]` 时，右边的 `dp[j-w[i]]` 应该来自“还没有选第 i 个物品之前”的状态。倒序循环可以保证 `dp[j-w[i]]` 还没有被本轮物品更新过。

为什么不能正序

假设只有一个物品，重量为 2，价值为 3，容量为 4。如果用正序循环：

```
for (int j = w[i]; j <= m; j++) {
    dp[j] = max(dp[j], dp[j - w[i]] + v[i]);
}
```

处理这个物品时：

- 当 $j=2$，dp[2] 会变成 3；
- 当 $j=4$，程序会用已经更新过的 dp[2]，得到 dp[4]=6。

这相当于把同一个物品选了两次。但 0-1 背包不允许这样。

倒序循环时，先算 $j=4$，此时 dp[2] 还是处理本物品之前的值，不会重复选择第 i 个物品。

这就是 0-1 背包一维优化中最重要的循环方向。

例题：一维数组追踪

有两个物品：

物品	重量	价值
1	2	3
2	3	4

容量为 5。用一维 0-1 背包计算。

初始：

j	0	1	2	3	4	5
dp[j]	0	0	0	0	0	0

处理物品 1，倒序更新 $j=5,4,3,2$：

j	0	1	2	3	4	5
dp[j]	0	0	3	3	3	3

处理物品 2，倒序更新 $j=5,4,3$：

- dp[5]=max(dp[5],dp[2]+4)=max(3,7)=7；
- dp[4]=max(dp[4],dp[1]+4)=max(3,4)=4；
- dp[3]=max(dp[3],dp[0]+4)=max(3,4)=4。

最终：

j	0	1	2	3	4	5
dp[j]	0	0	3	4	4	7

答案为 dp[5]=7。

这里 dp[4]=4，表示容量不超过 4 时，只选物品 2 比只选物品 1 更好。它不是“恰好重量为 4”的意思。

完全背包

完全背包和 0-1 背包很像，但每个物品可以选任意多次。

二维写法可以从状态含义推出来。若 dp[i][j] 表示前 i 种物品、容量不超过 j 的最大价值，那么：

- 不选第 i 种物品：`dp[i - 1][j]`；
- 至少选一个第 i 种物品：选掉一个后，仍然可以继续选第 i 种物品，所以来自 `dp[i][j - w[i]] + v[i]`。

转移为：

```
dp[i][j] = dp[i - 1][j];

if (j >= w[i]) {
    dp[i][j] = max(dp[i][j], dp[i][j - w[i]] + v[i]);
}
```

和 0-1 背包相比，第二项从 `dp[i - 1][...]` 变成了 `dp[i][...]`。这正是“可以重复选当前物品”的体现。

一维写法中，完全背包容量要从小到大循环：

```
for (int i = 1; i <= n; i++) {
    for (int j = w[i]; j <= m; j++) {
        dp[j] = max(dp[j], dp[j - w[i]] + v[i]);
    }
}
```

正序循环会让 `dp[j-w[i]]` 使用本轮已经更新过的状态，正好表示当前物品可以选多次。

0-1 背包与完全背包对比

类型	每个物品可选次数	二维转移中选当前物品来自	一维容量循环
0-1 背包	最多 1 次	`dp[i - 1][j - w[i]]`	从大到小
完全背包	任意多次	`dp[i][j - w[i]]`	从小到大

第一轮中，最常考的是这张表背后的含义。不要机械记“0-1 倒序，完全正序”，要知道它们分别是在防止重复选、允许重复选。

例题：正序和倒序的区别

只有一种物品，重量 2，价值 3，容量 6。

如果是 0-1 背包，最多选一次，答案是 3。

如果是完全背包，可以选 3 次，总重量 6，总价值 9。

对应代码差别是：

```
// 0-1 背包
for (int j = 6; j >= 2; j--) {
    dp[j] = max(dp[j], dp[j - 2] + 3);
}
```

和：

```
// 完全背包
for (int j = 2; j <= 6; j++) {
    dp[j] = max(dp[j], dp[j - 2] + 3);
}
```

如果把 0-1 背包写成正序，就会误算成可以选多次。

方案数背包

有些背包不求最大价值，而是求方案数。

例如：给定若干个数，每个数最多使用一次，问有多少种选法使总和恰好为 m。

可以设 `cnt[j]` 表示凑出和为 j 的方案数。

初始化：

```
cnt[0] = 1;
```

转移：

```
for (int i = 1; i <= n; i++) {
    for (int j = m; j >= a[i]; j--) {
        cnt[j] += cnt[j - a[i]];
    }
}
```

这里仍然倒序，因为每个数最多使用一次。

若每个数可以使用任意多次，则是完全背包计数，容量应正序：

```
for (int i = 1; i <= n; i++) {
    for (int j = a[i]; j <= m; j++) {
        cnt[j] += cnt[j - a[i]];
    }
}
```

例题：计数不是取最大

有数 $2,3,5$，每个数最多使用一次，问凑出和 5 有多少种选法。

合法选法有：

- 选 5；
- 选 2 和 3。

共有 2 种。

用计数 DP：

```
cnt[0] = 1;

for (int i = 1; i <= n; i++) {
    for (int j = m; j >= a[i]; j--) {
        cnt[j] += cnt[j - a[i]];
    }
}
```

处理完后 `cnt[5]=2`。

若把转移写成 `max`，就不是在统计方案数；若把初值 `cnt[0]` 写成 0，所有方案都无法产生。

可达性背包

还有一类题只问能不能凑出某个重量，不关心价值，也不关心方案数。

可以设 ok[j] 表示是否能凑出重量 j。

0-1 写法：

```
ok[0] = true;

for (int i = 1; i <= n; i++) {
    for (int j = m; j >= w[i]; j--) {
        ok[j] = ok[j] || ok[j - w[i]];
    }
}
```

如果最后 ok[m] 为真，就说明能恰好凑出 m。

这里的合并方式是逻辑或。因为只要存在一种来源能凑出，就说明当前状态可达。

例题：判断可达

有重量 $3, 5, 7$，每个最多使用一次，能否凑出 10？

可以选 3 和 7，所以能凑出。

用 DP 看：

- 初始 ok[0]=true；
- 处理 3 后，ok[3]=true；
- 处理 5 后，ok[5]=true，ok[8]=true；
- 处理 7 后，ok[7]=true，ok[10]=true，ok[12]=true，ok[15]=true。

所以 ok[10] 为真。

程序阅读中的背包

看下面代码：

```
for (int i = 1; i <= n; i++) {
    for (int j = m; j >= w[i]; j--) {
        f[j] = max(f[j], f[j - w[i]] + v[i]);
    }
}
```

读这段代码时，先抓三个点：

第一，外层枚举物品，内层枚举容量。

第二，容量从大到小，所以每个物品最多被使用一次。

第三，合并方式是 max，所以求的是最大价值，而不是方案数或可达性。

由此可以判断：这是一段 0-1 背包代码，f[j] 表示处理过若干物品后，容量不超过 j 能得到的最大价值。

再看下面代码：

```
for (int i = 1; i <= n; i++) {
    for (int j = w[i]; j <= m; j++) {
        f[j] = max(f[j], f[j - w[i]] + v[i]);
    }
}
```

容量从小到大，所以当前物品可以反复使用。这是完全背包最大价值。

再看：

```
c[0] = 1;

for (int i = 1; i <= n; i++) {
    for (int j = m; j >= a[i]; j--) {
        c[j] += c[j - a[i]];
    }
}
```

这里倒序说明每个数最多用一次；加法说明统计方案数；`c[0]=1` 是空方案。它不是最大价值背包。

完善程序常见空缺

空缺在循环方向

若题目是 0-1 背包，且使用一维数组：

```
for (int i = 1; i <= n; i++) {
    for (int j = ________; j >= ________; j--) {
        dp[j] = max(dp[j], dp[j - w[i]] + v[i]);
    }
}
```

通常应填：

```
m
w[i]
```

完整代码：

```
for (int j = m; j >= w[i]; j--) {
    dp[j] = max(dp[j], dp[j - w[i]] + v[i]);
}
```

若把 `j++` 或从小到大填进去，就会变成完全背包的效果。

空缺在转移来源

二维 0-1 背包：

```
dp[i][j] = dp[i - 1][j];

if (j >= w[i]) {
    dp[i][j] = max(dp[i][j], ________ + v[i]);
}
```

空缺处应填：

```
dp[i - 1][j - w[i]]
```

因为选第 i 个物品后，前面只能从前 $i-1$ 个物品里选。

如果填成 `dp[i][j - w[i]]`，就允许第 i 个物品被重复使用，变成完全背包含义。

空缺在初始化

恰好装满求最大价值：

```
for (int j = 0; j <= m; j++) {
    dp[j] = NEG;
}

dp[________] = ________;
```

应填：

```
0
0
```

即：

```
dp[0] = 0;
```

这表示恰好装出重量 0 的最大价值为 0。其他重量一开始不可达。

方案数背包：

```
cnt[________] = ________;
```

常填：

```
0
1
```

即：

```
cnt[0] = 1;
```

表示空方案有 1 种。

空缺在答案

若 `dp[j]` 表示容量不超过 j 的最大价值，通常输出：

```
cout << dp[m] << endl;
```

若 `dp[j]` 表示恰好装出重量 j 的最大价值，也通常输出 `dp[m]`，但要判断它是否仍为非法初值。

若题目要求“不超过容量 m，但状态写成恰好重量 j”，答案应在所有 $0 \le j \le m$ 中取最大：

```
int ans = 0;

for (int j = 0; j <= m; j++) {
    ans = max(ans, dp[j]);
}
```

这也是容易混淆的地方。状态若表示“容量上限”，答案是 `dp[m]`；状态若表示“恰好重量”，答案可能要枚举。

真题式完善程序：0-1 背包

补全下面程序。给定 n 个物品，背包容量为 m，第 i 个物品重量为 `w[i]`，价值为 `v[i]`。每个物品最多选一次，求总重量不超过 m 的最大价值。

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

const int N = 1005;

int n, m;
int w[N], v[N], dp[N];

int main() {
    cin >> n >> m;

    for (int i = 1; i <= n; i++) {
        cin >> w[i] >> v[i];
    }

    for (int i = 1; i <= n; i++) {
        for (int j = ________; j >= ________; j--) {
            dp[j] = max(dp[j], ________ + v[i]);
        }
    }

    cout << dp[m] << endl;
    return 0;
}
```

三个空依次应填：

```
m
w[i]
dp[j - w[i]]
```

完整核心代码为：

```
for (int i = 1; i <= n; i++) {
    for (int j = m; j >= w[i]; j--) {
        dp[j] = max(dp[j], dp[j - w[i]] + v[i]);
    }
}
```

解释如下：

- `j` 从 `m` 到 `w[i]`，保证容量足够放下第 i 个物品；
- 倒序循环，保证第 i 个物品最多被使用一次；
- `dp[j - w[i]] + v[i]` 表示选第 i 个物品；
- `max` 表示在“不选”和“选”之间取较优结果。

变式：完全背包

如果题目改成每个物品可以选任意多次，内层循环应改成：

```
for (int j = w[i]; j <= m; j++) {
    dp[j] = max(dp[j], dp[j - w[i]] + v[i]);
}
```

其他部分不一定变。只改变循环方向，就改变了“每个物品能选几次”的含义。

这类变式在完善程序中很常见。看到一维背包，先判断物品能否重复选，再判断容量循环方向。

真题式程序阅读：一维背包

阅读下面程序片段：

```
int n = 3, m = 5;
int w[4] = {0, 2, 3, 4};
int v[4] = {0, 3, 4, 5};
int dp[6];

for (int i = 1; i <= n; i++) {
    for (int j = m; j >= w[i]; j--) {
        dp[j] = max(dp[j], dp[j - w[i]] + v[i]);
    }
}

cout << dp[m] << endl;
```

问输出多少？

这是 0-1 背包。物品如下：

物品	重量	价值
1	2	3
2	3	4
3	4	5

容量为 5。最优选择是物品 1 和 2，总重量 5，总价值 7。输出 7。

如果把内层循环改成：

```
for (int j = w[i]; j <= m; j++)
```

就变成完全背包。此时物品 1 可以选两次，重量 4，价值 6；物品 1 加物品 2 仍为价值 7。本例答案还是 7，但含义已经变了。换一组数据时结果可能不同。

例如只有一个物品，重量 2，价值 3，容量 4。0-1 背包答案为 3，完全背包答案为 6。

易错点

第一，混淆 0-1 背包和完全背包。0-1 背包每个物品最多选一次，一维循环要倒序；完全背包可以重复选，一维循环要正序。

第二，把 `dp[j]` 的含义看错。它可能表示容量不超过 j 的最大价值，也可能表示恰好装出重量 j 的最大价值，还可能表示方案数或可达性。

第三，初始化不看题意。不超过容量求最大值时，初值为 0 通常没问题；恰好装满时，不合法状态不能直接设成 0。

第四，把计数 DP 写成最大值 DP。求方案数要用加法，求最大价值才用 `max`。

第五，把可达性 DP 写成计数 DP。判断能否做到时常用逻辑或，不需要统计有多少种。

第六，一维优化时覆盖了还要使用的状态。0-1 背包若正序循环，会在同一轮中重复使用当前物品。

第七，答案位置不匹配状态。如果状态表示 “恰好重量”，而题目要求 “不超过容量”，可能要枚举所有 `dp[j]`；如果状态表示 “容量上限”，输出 `dp[m]` 即可。

第八，忽略物品编号从 1 还是从 0 开始。读程序时要看输入循环和数组访问，不要把自己的习惯带进去。

第九，看到背包就只背模板。第一轮更常考代码含义、循环方向和初始化，而不是默写模板。

小结

背包是 DP 中最适合考程序阅读和完善程序的模型之一。掌握背包，不只是记住一段代码，更要能解释每个下标和每个循环方向的意义。

本章需要重点记住：

- 0-1 背包中，每个物品最多选一次；
- 二维状态 `dp[i][j]` 常表示前 i 个物品、容量不超过 j 的最大价值；
- 0-1 背包选第 i 个物品时，应从 `dp[i-1][j-w[i]]` 转移；
- 一维 0-1 背包容量要倒序，避免同一物品被重复使用；
- 完全背包允许物品重复使用，一维容量通常正序；
- 求最大价值用 `max`，求方案数用加法，求可达性用逻辑或；
- `cnt[0]=1` 表示空方案，`ok[0]=true` 表示重量 0 可达；
- 恰好装满和不超过容量的初始化不同；
- 答案位置必须与状态含义一致。

下一章讲最大子段和。它也是一类常见 DP，但状态不再是 “容量”，而是 “以当前位置结尾” 的最优值。

0x2A 最大子段和

背包的状态围绕“前几个物品”和“容量”展开；最大子段和的状态围绕“连续”和“结尾位置”展开。它的代码很短，但很适合考程序阅读，因为学生经常把“前缀最优”和“以当前位置结尾”混在一起。

本章只讲连续子段的最大和，不展开到子矩阵、带修改维护等更复杂内容。

问题

给定一个整数序列 $a_1, a_2, \dots, a_n$，从中选出一段连续的非空子段，使这段数的和最大。

连续子段指的是形如 $a_l, a_{l+1}, \dots, a_r$ 的一段，其中 $1 \le l \le r \le n$。它不能跳着选。

例如序列：

位置	1	2	3	4	5
数值	2	-5	3	4	-1

所有连续子段中，$3, 4$ 的和为 7，是最大值。虽然 $2, 3, 4$ 的和为 9 看起来更大，但它们在原序列中不是连续的，中间隔着 -5，不能这样选。

最大子段和常考两件事：

- 状态 `f[i]` 到底表示什么；
- 最后为什么还要枚举所有 `f[i]` 取最大。

朴素做法

可以枚举子段的左右端点。

先预处理前缀和：

```
s[0] = 0;

for (int i = 1; i <= n; i++) {
    s[i] = s[i - 1] + a[i];
}
```

子段 $a_l, \dots, a_r$ 的和是 $s[r] - s[l-1]$。

于是可以写：

```
int ans = a[1];

for (int l = 1; l <= n; l++) {
    for (int r = l; r <= n; r++) {
        ans = max(ans, s[r] - s[l - 1]);
    }
}
```

这段程序枚举了所有 $O(n^2)$ 个连续子段，每个子段用前缀和 $O(1)$ 算出和，所以时间复杂度为 $O(n^2)$。

第一轮一般不会要求写这个程序，但它有助于理解题意：最大子段和不是选若干个数，而是在所有连续区间中取最大和。

状态

更常用的 DP 状态是：

`f[i]` 表示以第 i 个数结尾的非空连续子段的最大和。

这句话要读准确。`f[i]` 不是前 i 个数中的最大子段和，而是“必须以 i 结尾”。

为什么要这样定义？因为若一段连续子段以 i 结尾，它只有两种情况：

- 只包含 a_i；
- 在某个以 $i-1$ 结尾的连续子段后面接上 a_i。

所以：

```
f[i] = max(a[i], f[i - 1] + a[i]);
```

初始化为：

```
f[1] = a[1];
```

最终答案是所有 `f[i]` 的最大值：

```
int ans = f[1];

for (int i = 2; i <= n; i++) {
    ans = max(ans, f[i]);
}
```

因为最大子段可能在任意位置结束，不一定以第 n 个数结尾。

手算

仍然看序列 $2, -5, 3, 4, -1$。

i	a_i	`f[i]` 的计算	`f[i]`
1	2	初始	2
2	-5	`max(-5, 2-5)`	-3
3	3	`max(3, -3+3)`	3
4	4	`max(4, 3+4)`	7
5	-1	`max(-1, 7-1)`	6

`f[5]=6`，但答案是所有 `f[i]` 中的最大值 7。

这张表说明了一个重要区别：

- `f[i]` 是以 i 结尾的最优子段和；
- `ans` 才是目前见过的全局最大子段和。

如果直接输出 `f[n]`，本例会输出 6，而正确答案是 7。

为什么状态不是前缀最优

有些学生会把 `f[i]` 定义成前 i 个数中的最大子段和。这个状态本身也可以用，但转移会多一个辅助信息。

若 `f[i]` 表示前 i 个数中的最大子段和，那么前 i 个数的最优子段可能：

- 完全在前 $i-1$ 个数中；
- 以第 i 个数结尾。

第二种情况还需要知道“以第 i 个数结尾的最大子段和”。所以只用一个 `f[i]` 不够方便。

如果把两个含义混在一起，常会写出这样的错误转移：

```
f[i] = max(f[i - 1], f[i - 1] + a[i]);
```

这段转移的问题是：如果 `f[i-1]` 表示前缀最优，它对应的子段可能不以 $i-1$ 结尾，后面不能直接接上 a_i，否则连续性会被破坏。

例如序列 $5, -100, 4$。前 2 个数的最大子段和是 5，但这个子段以位置 1 结尾，不能直接接上位置 3 的 4。若强行用 `f[2]+a[3]`，就会把不连续的 5 和 4 接在一起。

因此，最大子段和的关键状态是“以当前位置结尾”，不是“前缀最大”。

代码

非空子段版本可以写成：

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

const int N = 100005;

int n;
int a[N], f[N];

int main() {
    cin >> n;

    for (int i = 1; i <= n; i++) {
        cin >> a[i];
    }

    f[1] = a[1];
    int ans = f[1];

    for (int i = 2; i <= n; i++) {
        f[i] = max(a[i], f[i - 1] + a[i]);
        ans = max(ans, f[i]);
    }

    cout << ans << endl;
    return 0;
}
```

这段代码适用于必须选择非空子段的题目。若所有数都是负数，答案就是其中最大的那个负数。

例如 $-5, -2, -7$，最大非空连续子段是只选 -2，答案为 -2。

允许空子段

有些程序会把空子段也算作合法，空子段的和为 0。这时答案至少是 0。

对应代码常写成：

```
f[0] = 0;
int ans = 0;

for (int i = 1; i <= n; i++) {
    f[i] = max(0, f[i - 1] + a[i]);
    ans = max(ans, f[i]);
}
```

这段代码中，`f[i]` 表示以第 i 个数结尾，或者在第 i 个位置选择空段后的最大和。更通俗地说，一旦前面的累计和变成负数，就丢掉它，从后面重新开始。

对于 $-5, -2, -7$，这段代码输出 0，因为它允许什么都不选。

所以程序阅读时一定要看初始化和转移：

写法	是否允许空子段	全负数时答案
`f[1]=a[1]`，`f[i]=max(a[i],f[i-1]+a[i])`	不允许	最大的负数
`f[0]=0`，`f[i]=max(0,f[i-1]+a[i])`	允许	0

第一轮题中，判断题很喜欢利用这个差别。

滚动变量

由于 `f[i]` 只依赖 `f[i-1]`，可以不用数组，只保留一个变量。

```
int cur = a[1];
int ans = a[1];

for (int i = 2; i <= n; i++) {
    cur = max(a[i], cur + a[i]);
    ans = max(ans, cur);
}
```

这里 `cur` 相当于上一节中的 `f[i]`。

若允许空子段，则常写成：

```
int cur = 0;
int ans = 0;

for (int i = 1; i <= n; i++) {
    cur = max(0, cur + a[i]);
    ans = max(ans, cur);
}
```

读滚动变量程序时，不要因为没有 `f[i]` 就看不出 DP。变量 `cur` 保存的就是“以当前位置结尾”的状态。

程序阅读：表格追踪

看下面程序：

```cpp
int a[6] = {0, -2, 3, -1, 4, -5};

int cur = a[1];
int ans = a[1];

for (int i = 2; i <= 5; i++) {
    cur = max(a[i], cur + a[i]);
    ans = max(ans, cur);
}

cout << ans << endl;
```

逐步追踪：

i	a_i	cur 更新	ans
1	-2	-2	-2
2	3	max(3, -2+3)=3	3
3	-1	max(-1, 3-1)=2	3
4	4	max(4, 2+4)=6	6
5	-5	max(-5, 6-5)=1	6

最终输出 6。对应子段是 $3, -1, 4$。

这类题不要只心算答案，要跟着变量走。尤其当题目问 “某一行执行后变量是多少” 时，追踪表比直接猜更稳。

分治做法

最大子段和也可以用分治来做。把区间分成左右两半，最大子段有三类：

- 完全在左半边；
- 完全在右半边；
- 跨过中点。

前两类递归求解。第三类必须由 “左半边的最大后缀和” 和 “右半边的最大前缀和” 相加得到。

例如区间：

位置	1	2	3	4	5	6
数值	-1	3	-2	4	-5	2

若中点在 3 和 4 之间，跨过中点的子段必须包含左半边的某个后缀和右半边的某个前缀。

左半边 $-1, 3, -2$ 的最大后缀和为 1，来自 $3, -2$。

右半边 $4, -5, 2$ 的最大前缀和为 4，来自 4。

所以跨中点的最大子段和为 $1 + 4 = 5$。

分治算法的核心不是 “递归拆开”，而是会把跨中点的情况算出来。

区间合并的四个量

在程序阅读中，最大子段和常用结构体维护四个量：

名称	含义
`sum`	整个区间的和
`pre`	区间最大前缀和
`suf`	区间最大后缀和
`best`	区间最大子段和

若有两个相邻区间 L 和 R，合并后的四个量为：

合并后的量	计算方式
`sum`	`L.sum + R.sum`
`pre`	`max(L.pre, L.sum + R.pre)`
`suf`	`max(R.suf, R.sum + L.suf)`
`best`	`max(max(L.best, R.best), L.suf + R.pre)`

其中 `best` 的三种来源刚好对应：

- 最大子段完全在左区间；
- 最大子段完全在右区间；
- 最大子段跨过左右边界。

这四个量在分治程序、线段树程序、程序阅读题中都很常见。本章不讲线段树维护，只要求能读懂合并逻辑。

合并例子

左区间为 $2, -5, 3$，右区间为 $4, -1$。

左区间：

- `sum = 0`；
- `pre = 2`；
- `suf = 3`；
- `best = 3`。

右区间：

- `sum = 3`；
- `pre = 4`；
- `suf = 3`；
- `best = 4`。

合并后：

- `sum = 0+3=3`；
- `pre = max(2, 0+4)=4`；
- `suf = max(3, 3+3)=6`；
- `best = max(3, 4, 3+4)=7`。

最大子段为 $3, 4$，答案 7。

真题式程序阅读：结构体合并

阅读下面代码片段：

```
struct Node {
    int pre, best, suf, sum;
};

Node merge(Node a, Node b) {
    Node c;
    c.pre = max(a.pre, a.sum + b.pre);
    c.best = max(max(a.best, b.best), a.suf + b.pre);
    c.suf = max(a.suf + b.sum, b.suf);
    c.sum = a.sum + b.sum;
    return c;
}
```

问：best 表示什么？为什么要比较三项？

best 表示当前区间的最大子段和。

把两个相邻区间合并时，最大子段只有三种位置：

- 完全在左区间，值为 a.best；
- 完全在右区间，值为 b.best；
- 横跨左右区间，值为左区间最大后缀和 a.suf 加右区间最大前缀和 b.pre。

所以：

```
c.best = max(max(a.best, b.best), a.suf + b.pre);
```

这段代码是最大子段和的区间合并。即使变量名不是 pre、suf、best，只要看到“前缀、后缀、区间总和、内部最优”这四个量，就应想到这个结构。

变式：允许空段

如果叶子结点这样初始化：

```
Node x;
x.sum = a[i];
x.pre = max(a[i], 0);
x.suf = max(a[i], 0);
x.best = max(a[i], 0);
```

这表示允许空子段。因为当 $a_i < 0$ 时，最大前缀、后缀和子段都可以取空段，值为 0。

如果初始化为：

```
Node x;
x.sum = a[i];
x.pre = a[i];
x.suf = a[i];
x.best = a[i];
```

这表示不允许空子段。

程序阅读题中，只看合并公式还不够，还要看叶子或边界初始化。

分治程序复杂度

如果用普通分治求最大子段和，每一层递归都要在线性时间内扫描左右两边求跨中点答案。递归式为 $T(n)=2T(n/2)+O(n)$，时间复杂度为 $O(n\log n)$。

如果用前面四个量进行区间合并，每个区间只在合并时做常数次计算，整棵递归过程总共处理 $O(n)$ 个结点，时间复杂度为 $O(n)$。

两种程序都可能出现在阅读题中。判断时要看合并时做了什么：

做法	合并时的工作	时间复杂度
普通分治	扫描中点两侧	$O(n\log n)$
四个量合并	常数次计算	$O(n)$
线性 DP	每个位置更新一次	$O(n)$

不要只看到递归就认为一定是 $O(n\log n)$。递归中的合并代价不同，复杂度也不同。

例题：判断复杂度

看下面函数：

```
int solve(int l, int r) {
    if (l == r) {
        return max(a[l], 0);
    }

    int mid = (l + r) >> 1;
    int left = solve(l, mid);
    int right = solve(mid + 1, r);

    int s = 0, bestl = 0;
    for (int i = mid; i >= l; i--) {
        s += a[i];
        bestl = max(bestl, s);
    }

    s = 0;
    int bestr = 0;
    for (int i = mid + 1; i <= r; i++) {
        s += a[i];
        bestr = max(bestr, s);
    }

    return max(max(left, right), bestl + bestr);
}
```

这段代码允许空子段，因为单点返回 `max(a[l],0)`，并且 `bestl`、`bestr` 初值为 0。

每次递归把区间分成两半，然后扫描整个区间的一部分来求跨中点答案。每一层扫描总长度为 $O(n)$，共有 $O(\log n)$ 层，所以时间复杂度为 $O(n\log n)$。

如果题目问“当所有数都是负数时输出多少”，这段程序会输出 0，不是最大的负数。

例题：改动一行的影响

下面代码求非空最大子段和：

```
f[1] = a[1];
ans = f[1];

for (int i = 2; i <= n; i++) {
    f[i] = max(a[i], f[i - 1] + a[i]);
    ans = max(ans, f[i]);
}
```

若把第一行改成：

```
f[1] = max(a[1], 0);
ans = f[1];
```

会有什么影响？

它会改变全负数时的答案。

例如序列为 $-3, -5$。原程序中：

- f[1]=-3，ans=-3；
- f[2]=max(-5,-8)=-5；
- 最终答案为 -3。

修改后：

- f[1]=0，ans=0；
- f[2]=max(-5,-5)=-5；
- 最终答案仍为 0。

这等价于把空子段作为合法答案加入了程序。若题目要求非空子段，修改后的程序错误。

例题：答案位置

下面代码是否能正确输出最大子段和？

```
f[1] = a[1];

for (int i = 2; i <= n; i++) {
    f[i] = max(a[i], f[i - 1] + a[i]);
}

cout << f[n] << endl;
```

不一定。

f[n] 表示以第 n 个数结尾的最大子段和，而最大子段不一定以第 n 个数结尾。

例如序列 $3, 4, -10$：

- f[1]=3；
- f[2]=7；
- f[3]=max(-10,7-10)=-3。

程序输出 -3，但最大子段和是 7。

正确写法需要维护全局答案：

```cpp
int ans = f[1];

for (int i = 2; i <= n; i++) {
    f[i] = max(a[i], f[i - 1] + a[i]);
    ans = max(ans, f[i]);
}

cout << ans << endl;
```

真题式选择题

给定数组 $-1, 2, -3, 4, 5, -2$。使用如下程序：

```cpp
int cur = 0, ans = 0;

for (int i = 1; i <= n; i++) {
    cur = max(0, cur + a[i]);
    ans = max(ans, cur);
}

cout << ans << endl;
```

输出是多少？

逐步计算：

i	a_i	cur	ans
1	-1	0	0
2	2	2	2
3	-3	0	2
4	4	4	4
5	5	9	9
6	-2	7	9

输出 9。对应子段是 $4, 5$。

注意，这段程序允许空子段。若所有数都是负数，它会输出 0。

真题式完善程序

补全下面程序。给定 n 个整数，求非空连续子段的最大和。

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

const int N = 100005;

int n;
int a[N], f[N];

int main() {
    cin >> n;

    for (int i = 1; i <= n; i++) {
        cin >> a[i];
    }
```

```
    f[1] = a[1];
    int ans = ________;

    for (int i = 2; i <= n; i++) {
        f[i] = max(__________, __________);
        ans = max(ans, f[i]);
    }

    cout << ans << endl;
    return 0;
}
```

三个空依次应填：

```
f[1]
a[i]
f[i - 1] + a[i]
```

完整转移为：

```
f[i] = max(a[i], f[i - 1] + a[i]);
```

解释：

- `f[i]` 必须以第 i 个数结尾；
- 如果从 i 重新开始，子段和为 `a[i]`；
- 如果接在前一个以 $i-1$ 结尾的最优子段后面，子段和为 `f[i-1]+a[i]`；
- 答案可能在任意位置结束，所以用 `ans` 维护所有 `f[i]` 的最大值。

变式：允许空子段

若题目允许一个数都不选，程序可以改成：

```
int cur = 0;
int ans = 0;

for (int i = 1; i <= n; i++) {
    cur = max(0, cur + a[i]);
    ans = max(ans, cur);
}
```

这时全负数答案为 0。这不是“优化写法”的问题，而是题目定义改变了。

常见错误

第一，把 `f[i]` 当成前 i 个数的最大子段和。最大子段和中最常用的 `f[i]` 是以第 i 个数结尾的最大子段和。

第二，最后直接输出 `f[n]`。最大子段不一定以最后一个元素结尾，通常要维护 `ans=max(ans,f[i])`。

第三，忽略非空子段和空子段的区别。`max(0,...)` 往往表示允许空子段；若题目要求非空，全负数时会出错。

第四，把不连续的元素接到一起。只有以 $i-1$ 结尾的子段才能接上 a_i，前缀最优不能随便接。

第五，分治时漏掉跨中点的情况。最大子段可能一部分在左区间，一部分在右区间。

第六，看不懂四个量的合并。`sum` 是总和，`pre` 是最大前缀和，`suf` 是最大后缀和，`best` 是最大子段和。`best` 合并时必须比较左、右、跨边界三种情况。

第七，复杂度只看递归形式。普通分治合并时扫描区间，是 $O(n \log n)$；四个量合并每次常数计算，是 $O(n)$。

第八，变量名换了就认不出模型。有的代码用 `h,j,m,w`，有的代码用 `pre,best,suf,sum`，要看含义和合并公式。

小结

最大子段和的代码短，但它很好地考查了 DP 状态的准确含义。

本章需要重点记住：

- 最大子段必须连续，不能跳选；
- 常用状态 `f[i]` 表示以第 i 个数结尾的最大非空子段和；
- 转移是 `f[i]=max(a[i],f[i-1]+a[i])`；
- 最终答案是所有 `f[i]` 的最大值，不一定是 `f[n]`；
- `max(0,...)` 通常表示允许空子段；
- 全负数时，非空版本输出最大负数，允许空段版本输出 0；
- 分治做法要比较左半、右半、跨中点三种情况；
- 区间合并常维护 `sum`、`pre`、`suf`、`best` 四个量；
- 结构体合并公式本质上还是在维护最大子段和。

下一章讲子序列 DP。它与最大子段和的区别很大：子段要求连续，子序列可以删除若干元素但保持相对顺序。读题时先分清 “连续子段” 和 “子序列”，后面的状态设计才不会偏。

0x2B 子序列 DP

子序列和子段最大的区别在于：子段必须连续，子序列可以删除若干元素，但剩余元素的相对顺序不能改变。第 37 章已经讲过 LCS 与子序列的基本概念，本章不重复铺开字符串部分，而是把重点放在第一轮更常考的程序阅读和状态判断上。

子序列 DP 常见考法有三类：

- 判断 `dp[i][j]` 表示两个前缀之间的关系；
- 判断 `f[i]` 是否表示“以第 i 个元素结尾”；
- 根据循环和转移判断程序求的是长度、方案数，还是最优值。

子序列

给定序列 $a_1, a_2, \dots, a_n$，从中删除若干个元素，保留剩下元素的原有顺序，就得到一个子序列。

例如序列 $3, 1, 4, 2, 5$ 中：

- $3, 4, 5$ 是子序列；
- $1, 2, 5$ 是子序列；
- $4, 1$ 不是子序列，因为原序列中 4 在 1 后面；
- $3, 4, 2$ 是子序列，但不是连续子段。

子序列 DP 最常见的状态有两种：

状态形式	常见含义
`f[i]`	以第 i 个元素结尾的某类子序列最优值
`dp[i][j]`	第一个序列前 i 项、第二个序列前 j 项之间的答案

这两类状态的共同点是：都围绕“前缀”和“结尾”组织。

最长上升子序列

最长上升子序列，常写作 LIS。给定一个序列，要求选出尽量长的子序列，使选出的数严格递增。

例如序列 $3, 1, 4, 2, 5$，一个最长上升子序列是 $1, 2, 5$，长度为 3；另一个是 $3, 4, 5$，长度也是 3。

最常见的 $O(n^2)$ DP 状态是：

`f[i]` 表示以第 i 个数结尾的最长上升子序列长度。

为什么要强调“以第 i 个数结尾”？因为如果一个上升子序列以 a_i 结尾，那么它前面的最后一个元素一定来自某个位置 $j < i$，并且要满足 $a_j < a_i$。于是可以从 `f[j]` 转移到 `f[i]`。

转移为：

```
f[i] = 1;

for (int j = 1; j < i; j++) {
    if (a[j] < a[i]) {
        f[i] = max(f[i], f[j] + 1);
```

```
    }
}
```

其中 `f[i]=1` 表示只选 a_i 本身，也能形成长度为 1 的上升子序列。

最终答案是所有 `f[i]` 的最大值：

```
int ans = 0;

for (int i = 1; i <= n; i++) {
    ans = max(ans, f[i]);
}
```

不能直接输出 `f[n]`，因为最长上升子序列不一定以最后一个元素结尾。

手算 LIS

看序列 $3, 1, 4, 2, 5$。

i	a_i	可接在谁后面	f[i]
1	3	无	1
2	1	无	1
3	4	3、1	2
4	2	1	2
5	5	3、1、4、2	3

最终答案为 3。

其中 `f[5]=3`，但这只是因为本例中最优子序列恰好可以以 5 结尾。若序列改为 $3, 1, 4, 2, 0$，最长上升子序列长度仍为 2，但 `f[5]=1`。

所以和最大子段和一样，`f[i]` 是“以 i 结尾”的局部状态，答案要在所有位置中取最大。

上升还是不下降

题目中如果写“严格递增”，转移条件是：

```
a[j] < a[i]
```

如果写“非降”或“不下降”，转移条件应改为：

```
a[j] <= a[i]
```

这个差别会影响有相同元素的情况。

例如序列 $2, 2, 2$：

要求	最长长度
严格递增	1
不下降	3

程序阅读时，看到 `<` 和 `<=` 要特别留意。它往往不是小细节，而是题目条件的一部分。

真题式程序阅读：LIS

阅读下面程序：

```
int ans = 0;

for (int i = 1; i <= n; i++) {
    f[i] = 1;

    for (int j = 1; j < i; j++) {
        if (a[j] < a[i]) {
            f[i] = max(f[i], f[j] + 1);
        }
    }

    ans = max(ans, f[i]);
}

cout << ans << endl;
```

问：这段程序求什么？

外层枚举结尾位置 i，内层枚举前面的位置 j。当 $a_j < a_i$ 时，可以把以 j 结尾的上升子序列接上 a_i。`f[i]=1` 表示只选 a_i。

因此它求的是最长严格上升子序列长度。

若把条件改成：

```
if (a[j] <= a[i])
```

程序求的就是最长不下降子序列长度。

变式：答案位置

若把最后的输出改成：

```
cout << f[n] << endl;
```

程序是否仍然正确？

不一定。`f[n]` 只表示以第 n 个数结尾的最长上升子序列长度。最长上升子序列可能在前面已经结束。

例如序列 $1, 2, 3, 0$：

i	a_i	f[i]
1	1	1
2	2	2
3	3	3
4	0	1

正确答案是 3，但 `f[n]=1`。

最长公共子序列

最长公共子序列，常写作 LCS。给定两个序列，要求找出一个最长序列，使它同时是两个序列的子序列。

例如：

- 第一个串为 ABCBDAB；
- 第二个串为 BDCABA。

它们的一个公共子序列是 BCBA，长度为 4。公共子序列不要求在原串中连续，但顺序必须保持。

第 37 章已经讲过 LCS 的概念。这里重点看程序。

常用状态：

dp[i][j] 表示第一个串前 i 个字符和第二个串前 j 个字符的 LCS 长度。

若当前字符相同，即 s[i] == t[j]，可以把这个字符接到前面的公共子序列后面：

```
dp[i][j] = dp[i - 1][j - 1] + 1;
```

若当前字符不同，就不能同时选这两个字符，只能丢掉其中一个前缀末尾：

```
dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
```

完整代码常写成：

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= m; j++) {
        if (s[i] == t[j]) {
            dp[i][j] = dp[i - 1][j - 1] + 1;
        } else {
            dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
        }
    }
}
```

最终答案是 dp[n][m]。

LCS 表格

设 s = "ABC"，t = "AC"。为了方便下标从 1 开始，记 $s_1 = A, s_2 = B, s_3 = C$，$t_1 = A, t_2 = C$。

初始化时，空串和任何串的 LCS 长度都是 0，所以第 0 行、第 0 列都是 0。

计算表格：

dp[i][j]	$j = 0$	$j = 1(A)$	$j = 2(C)$
$i = 0$	0	0	0
$i = 1(A)$	0	1	1
$i = 2(B)$	0	1	1
$i = 3(C)$	0	1	2

最终答案 dp[3][2]=2，公共子序列可以是 AC。

真题式例题：LCS 长度

求字符串 ABCAAAABA 和 ABABCBABA 的最长公共子序列长度。

直接手工列完整 9×9 表比较费时。第一轮选择题中，可以结合状态表和选项判断。

一种公共子序列是 ABCABA，长度为 6。它在第一个串中的位置可以取：

A(1), B(2), C(3), A(4), B(8), A(9)

在第二个串中的位置可以取：

`A(1), B(2), C(5), A(7), B(8), A(9)`

所以答案至少为 6。

若要得到长度 7，两个串需要有更多字符按相同相对顺序匹配。用 LCS 转移计算，最后 `dp[9][9]=6`，所以最长公共子序列长度为 6。

这类题如果在考场上手算，不建议枚举所有子序列。更稳的做法是按 LCS 表格计算，或者在选项较少时先构造一个较长公共子序列，再判断能否继续延长。

子序列与子串

LCS 容易和最长公共子串混淆。

公共子序列不要求连续，公共子串要求连续。

例如：

- `ABCDEF`；
- `ACE`。

`ACE` 是第一个串的子序列，但不是子串。它和 `ACE` 的 LCS 长度是 3，但最长公共子串长度只有 1。

最长公共子串的状态常写成：

`dp[i][j]` 表示以 `s[i]` 和 `t[j]` 同时结尾的最长公共后缀长度。

转移为：

```
if (s[i] == t[j]) {
    dp[i][j] = dp[i - 1][j - 1] + 1;
} else {
    dp[i][j] = 0;
}
```

最终答案是所有 `dp[i][j]` 的最大值，而不是 `dp[n][m]`。

对比 LCS：

问题	是否要求连续	不相等时
最长公共子序列	不要求连续	`max(dp[i-1][j], dp[i][j-1])`
最长公共子串	要求连续	置为 0

看到不相等时是否置零，是区分这两类程序的重要线索。

计数型子序列 DP

子序列 DP 不一定求最长长度，也可能统计方案数。

例如：给定字符串 `s` 和 `t`，问 `s` 中有多少个子序列等于 `t`。

设 `dp[i][j]` 表示：在 `s` 的前 i 个字符中，选出一个子序列等于 `t` 的前 j 个字符的方案数。

初始化：

```
dp[i][0] = 1;
```

因为从任何前缀中选出空串，都只有一种方法：什么都不选。

转移：

如果不选 `s[i]`，方案数来自 `dp[i-1][j]`。

如果 `s[i] == t[j]`，还可以选 `s[i]` 作为匹配 `t[j]` 的字符，方案数来自 `dp[i-1][j-1]`。

所以：

```
dp[i][j] = dp[i - 1][j];

if (s[i] == t[j]) {
    dp[i][j] += dp[i - 1][j - 1];
}
```

最终答案是 `dp[n][m]`。

这里不是取最大值，而是累加方案数。读程序时看到 `+=`，要想到它可能是在统计数量。

例题：统计子序列

`s = "babgbag"`，`t = "bag"`。问 `s` 中有多少个子序列等于 `t`。

可以用上面的 DP。这里列出几个选择：

- 选位置 $1, 2, 3$：`bag`；
- 选位置 $1, 2, 7$：`bag`；
- 选位置 $1, 6, 7$：`bag`；
- 选位置 $4, 6, 7$：`bag`；
- 选位置 $5, 6, 7$：`bag`。

共有 5 种。

程序中 `dp[i][0]=1` 很关键，因为匹配空串是后续所有方案的起点。

选择与不选择

许多子序列 DP 都可以从 “最后一个元素选不选” 理解。

例如统计 `s` 中等于 `t` 的子序列个数时，处理到 `s[i]`：

- 不选 `s[i]`，方案数不变，来自 `dp[i-1][j]`；
- 若 `s[i]==t[j]`，选 `s[i]` 去匹配 `t[j]`，来自 `dp[i-1][j-1]`。

这和背包中的 “选或不选物品” 很像，只是背包的限制是容量，子序列的限制是相对顺序和目标匹配。

所以读子序列 DP 时，可以先问：

- 当前元素是否一定要选？
- 如果不选，状态从哪里来？
- 如果选，需要满足什么条件？
- 选了以后，下标怎样变化？

这四个问题能帮助学生看懂多数子序列转移。

程序阅读：判断状态

看下面代码：

```
for (int i = 0; i <= n; i++) {
    dp[i][0] = 1;
}

for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= m; j++) {
        dp[i][j] = dp[i - 1][j];

        if (s[i] == t[j]) {
            dp[i][j] += dp[i - 1][j - 1];
        }
    }
}

cout << dp[n][m] << endl;
```

这段程序不是 LCS。原因有三点：

第一，初始化 `dp[i][0]=1`，说明空目标串有一种匹配方法。

第二，不相等时没有取最大值，只继承 `dp[i-1][j]`，表示不选 `s[i]`。

第三，相等时用加法，说明在统计方案数。

所以这段程序统计的是：字符串 `s` 中有多少个子序列等于字符串 `t`。

如果把 `+=` 改成 `max`，含义就完全变了。

程序阅读：一维优化

统计子序列个数也可以用一维数组。

```
cnt[0] = 1;

for (int i = 1; i <= n; i++) {
    for (int j = m; j >= 1; j--) {
        if (s[i] == t[j]) {
            cnt[j] += cnt[j - 1];
        }
    }
}
```

这里 `cnt[j]` 表示当前已经扫描过的 `s` 前缀中，匹配 `t` 前 j 个字符的方案数。

内层 `j` 必须倒序。因为每个 `s[i]` 只能用于匹配 `t` 中的一个位置。如果正序更新，当前字符可能在同一轮中被重复使用。

例如 `s="a"`，`t="aa"`。正确答案为 0，因为一个字符不能匹配两个位置。

若正序：

- 先用 `s[1]` 更新 `cnt[1]`；
- 接着又可能用刚更新的 `cnt[1]` 更新 `cnt[2]`。

这就把同一个字符用了两次。

倒序可以避免这个问题。它和 0-1 背包倒序循环的原因很像：防止当前元素在同一轮被重复使用。

子序列 DP 与背包的相似处

子序列计数的一维优化和 0-1 背包很像：

模型	当前元素	倒序原因
0-1 背包	当前物品	防止同一物品被选多次
子序列计数	当前字符	防止同一字符匹配多个位置

这类联系有助于理解程序。第一轮不一定直接问 “这是哪个模型”，但会问修改循环方向后结果是否变化。

只要当前元素只能使用一次，一维 DP 中就要警惕正序循环是否会重复使用它。

真题式完善程序：统计目标子序列

补全下面程序。给定两个字符串 `s` 和 `t`，求 `s` 中有多少个子序列等于 `t`。字符串下标从 1 开始。

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

const int N = 1005;

int n, m;
string s, t;
long long dp[N][N];

int main() {
    cin >> s >> t;

    n = s.size();
    m = t.size();

    s = " " + s;
    t = " " + t;

    for (int i = 0; i <= n; i++) {
        dp[i][0] = ________;
    }

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            dp[i][j] = __________;

            if (s[i] == t[j]) {
                dp[i][j] += __________;
            }
        }
    }

    cout << dp[n][m] << endl;
    return 0;
}
```

三个空依次应填：

```
1
dp[i - 1][j]
dp[i - 1][j - 1]
```

解释：

- `dp[i][0]=1`，表示匹配空串有一种方法；
- `dp[i-1][j]` 表示不选 `s[i]`；
- 当 `s[i]==t[j]` 时，可以选 `s[i]` 去匹配 `t[j]`，方案数来自 `dp[i-1][j-1]`。

注意这里用的是加法，不是 `max`。

变式：一维写法

若改成一维数组，核心代码应为：

```
cnt[0] = 1;

for (int i = 1; i <= n; i++) {
    for (int j = m; j >= 1; j--) {
        if (s[i] == t[j]) {
            cnt[j] += cnt[j - 1];
        }
    }
}
```

这里 `j` 要倒序。若正序，会使同一个 `s[i]` 在同一轮中贡献到多个位置。

真题式完善程序：LCS

补全下面程序，求两个字符串的最长公共子序列长度。

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

const int N = 1005;

string s, t;
int dp[N][N];

int main() {
    cin >> s >> t;

    int n = s.size();
    int m = t.size();

    s = " " + s;
    t = " " + t;

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            if (s[i] == t[j]) {
                dp[i][j] = __________;
            } else {
                dp[i][j] = __________;
            }
        }
    }

    cout << dp[n][m] << endl;
    return 0;
}
```

两个空依次应填：

```
dp[i - 1][j - 1] + 1
max(dp[i - 1][j], dp[i][j - 1])
```

当两个末尾字符相同时，可以把它们作为公共子序列的最后一个字符。当前缀末尾字符不同时，最优解只能来自删去第一个串末尾或删去第二个串末尾这两种情况。

这段程序求的是长度，不负责输出具体序列。

子序列 DP 的答案位置

不同子序列 DP 的答案位置不同。

问题	状态含义	答案
LIS	`f[i]` 表示以 i 结尾的长度	`max(f[i])`
LCS	`dp[i][j]` 表示两个前缀的答案	`dp[n][m]`
统计 `s` 中等于 `t` 的子序列个数	`dp[i][j]` 表示前缀匹配方案数	`dp[n][m]`
最长公共子串	`dp[i][j]` 表示以两位置结尾的公共后缀长度	`max(dp[i][j])`

判断答案位置时，先看状态是否已经表示“整个前缀的答案”。如果是，通常输出右下角；如果状态带有“以某个位置结尾”的限制，就要枚举结束位置。

常见错误

第一，把子序列和子段混淆。子段要求连续，子序列可以删除元素但必须保持相对顺序。

第二，LIS 中直接输出 `f[n]`。`f[i]` 表示以 i 结尾，答案要在所有 `f[i]` 中取最大。

第三，严格递增和不下降不分。条件 `<` 和 `<=` 会影响相同元素能否同时选入。

第四，把 LCS 和最长公共子串混淆。LCS 不相等时取 `max(dp[i-1][j],dp[i][j-1])`；最长公共子串不相等时通常置 0。

第五，计数型子序列 DP 用错合并方式。统计方案数用加法，求长度用 `max`。

第六，忘记空串初始化。统计匹配方案数时，`dp[i][0]=1` 或 `cnt[0]=1` 常常是转移起点。

第七，一维优化时循环方向错误。当前字符只能用一次时，目标串下标通常要倒序。

第八，只看变量名判断模型。`dp[i][j]` 可能是 LCS，也可能是子序列计数，还可能是最长公共子串。要看初始化、相等时转移、不相等时转移和答案位置。

小结

子序列 DP 的难点不在公式长，而在状态含义要读准。子序列可以跳过元素，但不能改变顺序；因此许多转移都围绕“当前元素选不选”“当前元素能否接在前面状态后面”展开。

本章需要重点记住：

- 子序列不要求连续，但必须保持相对顺序；
- LIS 常用 `f[i]` 表示以第 i 个元素结尾的最长长度；
- LIS 的答案是所有 `f[i]` 的最大值，不一定是 `f[n]`；
- 严格递增用 `<`，不下降用 `<=`；
- LCS 常用 `dp[i][j]` 表示两个前缀的最长公共子序列长度；

- LCS 相等时来自 `dp[i-1][j-1]+1`，不相等时取 `max(dp[i-1][j],dp[i][j-1])`；
- 最长公共子串要求连续，不相等时通常置 0；
- 统计子序列方案数时，常用 “不选当前字符” 和 “选当前字符” 两类转移；
- 计数用加法，求长度用 `max`；
- 一维子序列计数中，目标串下标常要倒序，避免同一个字符被重复使用。

下一章进入贪心。贪心和 DP 都会做选择，但 DP 会保存多个状态，贪心则希望每一步的局部选择能够导向整体最优。它们在第一轮中经常被放在选择题和程序阅读中对比考查。

0x2C 贪心

贪心和 DP 都是在做选择。区别在于：DP 会把多个可能状态保存下来，贪心则希望每一步只保留一个看起来最合适的选择，并且这个选择不会破坏最终最优答案。

CSP-S 第一轮中的贪心常考：

- 某个排序规则是否正确；
- 某段程序是在按什么规则贪心；
- 贪心选择后，后续变量怎样变化；
- 某种看起来合理的规则为什么会错；
- 贪心和 DP 在同一个问题中的区别；
- 完善程序中比较条件、排序关键字、更新位置应填什么。

本章重点不在背很多模型，而在训练判断：这个局部选择能不能放心固定下来。

问题

贪心算法的基本形式是：每一步根据当前信息做一个局部选择，并且不回头修改之前的选择。

例如，有若干个活动，每个活动有开始时间和结束时间，要求选择尽量多的互不重叠活动。

一种可行的贪心规则是：每次选择结束时间最早、且与已选活动不冲突的活动。

为什么选择结束时间最早？因为它尽量少占用后面的时间，给剩余活动留下更多空间。

但这只是一个想法，还不能直接当成正确算法。贪心最容易出错的地方就在这里：局部看起来合适，不一定能得到整体最优。

贪心和 DP

先看一个背包问题。

有背包容量 10，物品如下：

物品	重量	价值	单位重量价值
1	6	12	2
2	5	10	2
3	5	10	2
4	10	19	1.9

如果按单位重量价值从高到低贪心，可能先选物品 1，剩余容量为 4，无法再选物品 2 或 3，总价值为 12。

但最优方案是选物品 2 和 3，总重量 10，总价值 20。

这说明 0-1 背包不能简单用单位重量价值贪心解决。第 41 章用 DP，是因为每个物品选不选会影响后续容量，需要保存多个状态。

再看活动选择。每次选结束时间最早的活动后，后续只需要考虑从这个结束时间之后开始的活动。这个选择不会让问题结构变复杂，所以适合贪心。

可以粗略区分：

问题特征	更常见方法
当前选择会影响后续剩余资源，且多种剩余状态都可能有用	DP
每一步有一个可以固定的局部选择，后续问题仍是同类问题	贪心
需要尝试多种路径，不能确定局部选择	搜索

这不是严格分类，但对第一轮判断题很有帮助。

活动选择

有若干活动，每个活动有开始时间 s_i 和结束时间 t_i。若两个活动时间重叠，就不能同时选择。目标是选出最多的活动。

贪心规则：

先按结束时间从小到大排序；依次扫描活动，如果当前活动的开始时间不早于上一个已选活动的结束时间，就选择它。

代码框架如下：

```
struct Act {
    int s, t;
};

bool cmp(Act a, Act b) {
    return a.t < b.t;
}

sort(a + 1, a + n + 1, cmp);

int ans = 0;
int last = -1;

for (int i = 1; i <= n; i++) {
    if (a[i].s >= last) {
        ans++;
        last = a[i].t;
    }
}
```

这里 `last` 表示最近一次选择的活动结束时间。每次选择一个活动后，后面只能选开始时间不早于 `last` 的活动。

手算

活动如下：

活动	开始	结束
1	1	4
2	3	5
3	0	6
4	5	7
5	8	9
6	5	9

按结束时间排序后为：

活动	开始	结束
1	1	4
2	3	5
3	0	6
4	5	7
5	8	9
6	5	9

扫描：

- 选活动 1，`last=4`；
- 活动 2 开始时间 $3 < 4$，不能选；
- 活动 3 开始时间 $0 < 4$，不能选；
- 活动 4 开始时间 $5 \geq 4$，选，`last=7`；
- 活动 5 开始时间 $8 \geq 7$，选，`last=9`；
- 活动 6 开始时间 $5 < 9$，不能选。

最终选了 3 个活动。

为什么按结束时间

按结束时间最早选择，有一个很重要的交换思路。

设当前所有可选活动中，结束时间最早的是 x。假设存在一个最优方案，它第一项选择的不是 x，而是另一个活动 y。

因为 x 是结束时间最早的活动，所以 x 的结束时间不晚于 y 的结束时间。把这个最优方案中的第一个活动 y 换成 x 后，后面原本能接在 y 后面的活动，也仍然能接在 x 后面。活动数量不会减少。

这说明：至少存在一个最优方案以 x 开头。于是我们可以放心选择 x，再处理剩下的活动。

这种证明方法常叫交换法。第一轮中不要求写得很形式化，但要能说清楚：为什么当前选择不会让最优答案变差。

错误的贪心规则

活动选择中，有些规则看起来也合理，但会失败。

按开始时间最早

活动如下：

活动	开始	结束
A	1	10
B	2	3
C	3	4
D	4	5

如果按开始时间最早，先选 A，后面都不能选，只能选 1 个活动。

但最优方案是选 B, C, D，可以选 3 个活动。

按持续时间最短

活动如下：

活动	开始	结束	持续时间
A	1	4	3
B	4	7	3
C	3	5	2

如果先选持续时间最短的 C，则 A 和 B 都与它冲突，只能选 1 个。

但选 A 和 B 可以选 2 个。

这些反例说明：贪心规则不是“看起来省”就可以，要和题目目标对齐。

排序贪心

很多贪心程序的第一步是排序。第一轮中，排序关键字经常是考点。

常见写法：

```
sort(a + 1, a + n + 1, cmp);
```

比较函数决定了贪心顺序。

例如活动选择：

```
bool cmp(Act a, Act b) {
    return a.t < b.t;
}
```

按结束时间从小到大排序。

若题目要让小任务优先完成，可能按处理时间从小到大：

```
bool cmp(Job a, Job b) {
    return a.len < b.len;
}
```

若题目要优先保留惩罚大的任务，可能按惩罚从大到小：

```
bool cmp(Job a, Job b) {
    return a.pen > b.pen;
}
```

程序阅读时，看到 `cmp` 不要跳过。排序规则往往就是整段贪心程序的核心。

同一关键字时的次序

有些程序会在主关键字相同时再比较一个次关键字：

```
bool cmp(Act a, Act b) {
    if (a.t != b.t) {
        return a.t < b.t;
    }

    return a.s < b.s;
}
```

这表示先按结束时间升序，结束时间相同再按开始时间升序。

第一轮中，如果题目只问最大活动数，活动选择里结束时间相同的活动选哪一个通常不影响数量。但如果题目还要求输出方案、字典序或某种编号，次关键字就可能影响答案。

真题式例题：任务顺序

有 5 个不可抢占任务 A_1, A_2, A_3, A_4, A_5，从时间 0 开始在一台机器上执行。它们的处理时长分别为 $3, 4, 2, 5, 1$，截止时刻分别为 $5, 10, 3, 15, 11$。如果某个任务超时，惩罚等于其处理时长。为了尽量减少总惩罚，优先执行哪个任务更合适？

先把信息列出来：

任务	处理时长	截止时刻
A_1	3	5
A_2	4	10
A_3	2	3
A_4	5	15
A_5	1	11

按截止时刻从早到晚排序为：

顺序	任务	完成时刻	是否超时
1	A_3	2	否
2	A_1	5	否
3	A_2	9	否
4	A_5	10	否
5	A_4	15	否

这样所有任务都能按时完成，总惩罚为 0。因此优先执行截止时间最早的 A_3 是合适的。

这里不要被“处理时间最短”迷惑。虽然 A_5 只需要 1 个单位时间，但它的截止时刻较晚；如果先执行它，可能让截止时间更早的任务变得紧张。

这类题的关键是：排序规则要服务题目目标。当前最短、当前最大、当前最早，哪一个有效，要结合约束判断。

区间覆盖

另一类常见贪心是区间覆盖。

问题：给定若干区间，要求用尽量少的区间覆盖目标区间 $[L, R]$。

常用方法：

先按左端点从小到大排序。维护当前已经覆盖到的位置 `cur`。在所有左端点不超过 `cur` 的区间中，选择右端点最大的那个，把覆盖范围尽量向右推进。

代码框架：

```
sort(a + 1, a + n + 1, cmp);

int cur = L;
int ans = 0;
int i = 1;

while (cur < R) {
    int best = cur;

    while (i <= n && a[i].l <= cur) {
```

```
        best = max(best, a[i].r);
        i++;
    }

    if (best == cur) {
        break;
    }

    ans++;
    cur = best;
}
```

这里每一步不是选择最短区间，也不是选择左端点最小的区间，而是在当前能接上的区间里选择右端点最远的区间。

手算

目标覆盖 $[0, 10]$，区间如下：

区间	左端点	右端点
1	0	3
2	0	5
3	4	8
4	5	10
5	6	9

初始 `cur=0`。左端点不超过 0 的区间有 $1, 2$，选择右端点最远的区间 2，覆盖到 5。

现在 `cur=5`。左端点不超过 5 且尚未处理的区间有 $3, 4$，选择右端点最远的区间 4，覆盖到 10。

共选 2 个区间。

哈夫曼合并

有一类贪心是每次合并代价最小的两个元素。典型例子是合并果子、哈夫曼树。

给定若干堆果子，每次可以合并两堆，代价为两堆大小之和。合并后形成新的一堆。求总合并代价最小。

贪心规则：每次取当前最小的两堆合并。

常用最小堆维护：

```
priority_queue<int, vector<int>, greater<int> > q;

for (int i = 1; i <= n; i++) {
    q.push(a[i]);
}

int ans = 0;

while (q.size() > 1) {
    int x = q.top();
    q.pop();

    int y = q.top();
    q.pop();

    ans += x + y;
```

```
    q.push(x + y);
}
```

手算

有四堆果子：$1, 2, 3, 9$。

每次合并最小两堆：

操作	合并	代价	新堆
1	$1+2$	3	$3, 3, 9$
2	$3+3$	6	$6, 9$
3	$6+9$	15	15

总代价 $3+6+15=24$。

如果先合并 9 和 1，代价会变大，因为大的数会在后续合并中继续被重复计入。让小的数先合并，可以减少它们后续参与总代价的层数影响。

第一轮中，看到 `priority_queue` 加 `greater<int>`，并且每次弹出两个最小值再放回它们的和，要想到哈夫曼合并类贪心。

贪心与反例

贪心题常见的错误做法，是把某个局部指标当作唯一标准。

反例一：换硬币

硬币面值为 $1, 3, 4$，要凑出 6。若每次选择不超过剩余金额的最大面值：

- 先选 4，剩余 2；
- 再选 1，剩余 1；
- 再选 1，剩余 0。

共用 3 枚硬币。

但最优方案是 $3+3$，只用 2 枚。

所以“每次选最大面值”并不总能得到最少硬币数。

反例二：0-1 背包按性价比

第 41 章已经讲过，0-1 背包按单位价值排序可能失败。一个物品看起来更划算，但选了它会占掉容量，使后面无法组成更好的组合。

这类反例说明：贪心需要题目结构支持。不能看到“最小”“最大”“排序”就直接套某个规则。

程序阅读：排序后扫描

看下面代码：

```
sort(a + 1, a + n + 1, cmp);

int ans = 0;
int last = -1;
```

```
for (int i = 1; i <= n; i++) {
    if (a[i].l >= last) {
        ans++;
        last = a[i].r;
    }
}
```

若 cmp 为：

```
bool cmp(Node x, Node y) {
    return x.r < y.r;
}
```

这就是按右端点从小到大扫描区间，选择最多互不重叠区间。

若 cmp 改为：

```
bool cmp(Node x, Node y) {
    return x.l < y.l;
}
```

程序含义就变成按左端点从小到大尝试选择区间，这个规则不一定能选出最多区间。

程序阅读题常把 cmp 放在代码前面，后面扫描代码看似一样，但排序规则一变，算法含义就变了。

程序阅读：堆中取最小

看下面程序片段：

```
priority_queue<int, vector<int>, greater<int> > q;

for (int i = 1; i <= n; i++) {
    q.push(a[i]);
}

long long ans = 0;

while (q.size() > 1) {
    int x = q.top();
    q.pop();

    int y = q.top();
    q.pop();

    ans += x + y;
    q.push(x + y);
}
```

这段程序每次取当前最小的两个数合并，累计合并代价。它求的是哈夫曼合并的最小总代价。

如果把 greater<int> 去掉，默认优先队列会变成大根堆，每次取最大的两个数。那样总代价通常会变大。

例如 $1, 2, 3, 9$：

- 小根堆贪心总代价为 24；
- 大根堆先合并 $9 + 3 = 12$，再合并 $12 + 2 = 14$，再合并 $14 + 1 = 15$，总代价 41。

所以堆的方向也是重要信息。

完善程序中的常见空缺

排序关键字

活动选择：

```
bool cmp(Act a, Act b) {
    return ________;
}
```

应填：

```
a.t < b.t
```

因为要按结束时间从早到晚选择。

区间覆盖常见：

```
bool cmp(Seg a, Seg b) {
    if (a.l != b.l) {
        return a.l < b.l;
    }

    return a.r > b.r;
}
```

先按左端点升序，左端点相同时右端点大的排前面，便于扫描时把覆盖范围推进更远。

扫描条件

活动选择：

```
if (a[i].s >= last) {
    ans++;
    last = a[i].t;
}
```

若区间端点定义为闭区间，题目规定两个活动不能有重叠但允许前一个结束时后一个开始，条件就是 `>=`。如果题目不允许端点相接，则可能要用 `>`。第一轮题中要看题目对“相邻时间”的定义。

更新位置

区间覆盖中：

```
while (i <= n && a[i].l <= cur) {
    best = max(best, a[i].r);
    i++;
}
```

`best` 要在所有能接上的区间中取最大右端点，不能看到第一个能接上的区间就立刻选择。否则会错过能覆盖更远的区间。

真题式完善程序：活动选择

补全下面程序。给定 n 个活动，每个活动有开始时间 `s` 和结束时间 `t`。若一个活动的开始时间不早于上一个已选活动的结束时间，则它们可以同时选择。求最多能选择多少个活动。

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

const int N = 100005;

struct Act {
    int s, t;
} a[N];

int n;

bool cmp(Act x, Act y) {
    return __________;
}

int main() {
    cin >> n;

    for (int i = 1; i <= n; i++) {
        cin >> a[i].s >> a[i].t;
    }

    sort(a + 1, a + n + 1, cmp);

    int ans = 0;
    int last = -1;

    for (int i = 1; i <= n; i++) {
        if (__________) {
            ans++;
            __________;
        }
    }

    cout << ans << endl;
    return 0;
}
```

三个空依次应填：

```
x.t < y.t
a[i].s >= last
last = a[i].t
```

解释：

- 按结束时间从早到晚排序；
- 当前活动开始时间不早于 `last` 时，可以选择；
- 选择后，更新最后结束时间。

如果第一个空填成 `x.s < y.s`，就变成按开始时间排序，不能保证得到最多活动数。

真题式完善程序：合并代价

补全下面程序。给定 n 个正整数，每次取出两个数 x, y，付出代价 $x + y$，再把 $x + y$ 放回。直到只剩一个数，求最小总代价。

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

int n;
priority_queue<int, vector<int>, __________> q;

int main() {
    cin >> n;

    for (int i = 1; i <= n; i++) {
        int x;
        cin >> x;
        q.push(x);
    }

    long long ans = 0;

    while (q.size() > 1) {
        int x = q.top();
        q.pop();

        int y = q.top();
        q.pop();

        ans += __________;
        q.push(__________);
    }

    cout << ans << endl;
    return 0;
}
```

三个空依次应填：

```
greater<int>
x + y
x + y
```

解释：

- 需要小根堆，每次取当前最小的两个数；
- 本次合并代价为 x+y；
- 合并后形成的新数也为 x+y，要放回堆中。

证明思路

第一轮中，贪心证明一般不用写得很长，但要能理解几种常见说法。

交换法

交换法常用于排序贪心。

基本思路是：如果某个最优方案没有按照贪心规则安排，那么找到第一处不符合贪心规则的位置，把它换成贪心选择，答案不会变差。不断交换后，可以得到一个符合贪心规则的最优方案。

活动选择中，把最优方案第一个活动换成结束时间最早的活动，就是交换法。

反证法

假设贪心选择不是最优的一部分，然后推出可以把某个最优方案改成包含贪心选择，且答案不变差，和假设冲突。

它和交换法经常一起出现。

反例

如果不能证明，就试着构造反例。第一轮选择题中，有些选项会给出看似合理的贪心规则。判断它是否正确时，可以尝试用很小的数据反驳。

例如：

- 按开始时间选活动，会被一个很长的早开始活动卡住；
- 0-1 背包按单位价值选，会被容量组合卡住；
- 换硬币每次选最大面值，会被 $1, 3, 4$ 凑 6 卡住。

小反例比空想更可靠。

贪心代码的复杂度

贪心程序的复杂度通常来自排序或堆。

活动选择：

```
sort(a + 1, a + n + 1, cmp);
```

排序为 $O(n \log n)$，后面扫描为 $O(n)$，总复杂度为 $O(n \log n)$。

哈夫曼合并：

每次从堆中弹出两个元素，再放入一个元素。共进行 $n-1$ 次合并，每次涉及堆操作，复杂度为 $O(\log n)$，总复杂度为 $O(n \log n)$。

区间覆盖：

排序为 $O(n \log n)$，扫描时每个区间最多进入一次内层循环，总复杂度为 $O(n \log n)$。

读复杂度时，不要只看扫描部分。很多贪心程序真正的主要开销在排序和堆操作上。

易错点

第一，把 “看起来最优” 当成 “整体最优”。贪心规则必须和问题结构匹配，否则会被反例击破。

第二，排序关键字看错。活动选择按结束时间，区间覆盖常按左端点扫描并取最大右端点，任务调度可能按截止时间或惩罚排序，不能混用。

第三，忽略相等情况。`<`、`<=`、`>`、`>=` 会影响活动是否能首尾相接、区间是否覆盖到端点。

第四，只选第一个可行项。区间覆盖中，要在所有当前能接上的区间里选右端点最远的，不是遇到一个可行区间就立刻定下来。

第五，把小根堆和大根堆弄反。哈夫曼合并要每次取最小的两个数。

第六，贪心和 DP 混淆。0-1 背包、很多子序列问题不能只靠一个局部选择解决，要保存状态。

第七，不会构造反例。判断错误贪心时，可以从 3 到 5 个元素的小数据开始试。

第八，复杂度漏掉排序。排序贪心一般不是 $O(n)$，而是 $O(n \log n)$。

小结

贪心的核心是局部选择。它的难点不在代码长，而在判断这个局部选择能否固定下来。

本章需要重点记住：

- 贪心每一步做当前选择，并且不回头修改；
- 贪心规则要能解释为什么不会破坏最优答案；
- 活动选择按结束时间从小到大排序；
- 区间覆盖在当前能接上的区间中选右端点最远的；
- 哈夫曼合并每次取最小的两个元素；
- 0-1 背包按单位价值贪心会失败，通常应使用 DP；
- 换硬币 “每次选最大面值” 不一定正确；
- 排序关键字、堆方向、扫描条件是程序阅读重点；
- 证明贪心常用交换法，也可以通过反例否定错误规则；
- 贪心复杂度常由排序或堆操作决定。

下一章进入 DFS 与回溯。和贪心不同，DFS 不急着把某一步固定为唯一选择，而是系统尝试多种选择，并在不合适时退回。

0x2D DFS 与回溯

贪心每一步试图固定一个选择；DFS 不急着固定唯一选择，而是把可能的选择逐一尝试。若走到某一步发现不合适，就返回上一步，撤销刚才的选择，再尝试其他分支。这就是回溯。

CSP-S 第一轮中的 DFS 与回溯常常考：

- `dfs(k)` 中参数 k 表示什么；
- 搜索什么时候停止；
- `flag`、`vis` 数组在记录什么；
- 为什么递归后要撤销标记；
- 一段程序总共会搜索多少种情况；
- 修改某一行后，答案会怎样变化；
- 剪枝条件是否会漏掉合法方案；
- 程序的时间复杂度大致是多少。

读 DFS 程序时，最重要的不是背模板，而是看清“当前处在搜索树的哪一层”。

搜索树

DFS 可以理解为在一棵搜索树上向下走。

例如，要从数字 $1,2,3$ 中选出一个排列。第一层选择第一个位置，第二层选择第二个位置，第三层选择第三个位置。

搜索树大致如下：

```
空
├── 1
│   ├── 1 2
│   │   └── 1 2 3
│   └── 1 3
│       └── 1 3 2
├── 2
│   ├── 2 1
│   │   └── 2 1 3
│   └── 2 3
│       └── 2 3 1
└── 3
    ├── 3 1
    │   └── 3 1 2
    └── 3 2
        └── 3 2 1
```

从根到叶子的一条路径，就是一个完整排列。DFS 的过程就是先沿一条路径走到底，再退回去尝试别的分支。

如果用 `dfs(k)` 表示正在填写第 k 个位置，那么：

- `k=1` 时，准备填第 1 个位置；
- `k=2` 时，前 1 个位置已经填好；
- `k=n+1` 时，前 n 个位置已经填好，得到一个完整方案。

这类参数含义在程序阅读题里非常关键。

基本结构

排列搜索的常见写法如下：

```
void dfs(int k) {
    if (k == n + 1) {
        ans++;
        return;
    }

    for (int i = 1; i <= n; i++) {
        if (vis[i]) {
            continue;
        }

        p[k] = i;
        vis[i] = true;

        dfs(k + 1);

        vis[i] = false;
    }
}
```

这段代码里：

- `k` 表示当前要填的位置；
- `i` 表示当前尝试填入的数字；
- `vis[i]` 表示数字 i 是否已经用过；
- `p[k]=i` 表示第 k 个位置填数字 i；
- `dfs(k+1)` 表示继续填下一个位置；
- `vis[i]=false` 是回溯，撤销当前选择。

如果少了最后一句撤销标记，后面的分支会误以为数字 i 仍然被占用，搜索会出错。

回溯

回溯的核心是：进入下一层前做出选择，下一层返回后撤销选择。

在排列搜索中，选择是：

```
p[k] = i;
vis[i] = true;
```

撤销是：

```
vis[i] = false;
```

`p[k]` 通常不用专门清空。因为下一次填写第 k 个位置时，会重新赋值。真正会影响后续判断的是 `vis[i]`，所以必须撤销。

这类代码常成对出现：

```
vis[i] = true;
dfs(k + 1);
vis[i] = false;
```

程序阅读时，看到递归调用前后有一对相反操作，就要意识到这是回溯。

例题：少撤销一步

下面程序用于统计 1 到 n 的排列数。

```
void dfs(int k) {
    if (k == n + 1) {
        ans++;
        return;
    }

    for (int i = 1; i <= n; i++) {
        if (vis[i]) {
            continue;
        }

        vis[i] = true;
        dfs(k + 1);
    }
}
```

这段程序少了什么？会有什么影响？

它少了递归返回后的：

```
vis[i] = false;
```

这样一来，第一次尝试过的数字会一直保持已使用状态。程序不能正确枚举其他分支，答案会偏小，甚至只得到极少数方案。

以 $n = 3$ 为例，程序会沿着 $1, 2, 3$ 走到一个完整方案，然后所有数字都被标记为已用。返回后无法再尝试其他排列，答案为 1，而正确排列数应为 6。

停止条件

DFS 一定要有停止条件。停止条件表示已经得到完整方案，或已经无法继续。

排列搜索中：

```
if (k == n + 1) {
    ans++;
    return;
}
```

表示前 n 个位置都填好了。

组合搜索中，如果要选出 m 个数，可能写成：

```
if (cnt == m) {
    ans++;
    return;
}
```

走迷宫时，如果已经到达终点，可能写成：

```
if (x == tx && y == ty) {
    ans++;
    return;
}
```

停止条件之后通常要 `return`。否则程序还会继续向下执行，可能造成重复统计或访问不该访问的位置。

停止条件的位置

停止条件一般放在函数开头。因为进入某一层时，先判断当前状态是否已经完整，再决定是否继续扩展。

例如：

```
void dfs(int k) {
    if (k == n + 1) {
        ans++;
        return;
    }

    ...
}
```

若把停止条件放在循环后面，程序可能会先访问 `p[n+1]` 或继续枚举没有意义的选择。

标记数组

DFS 中常见标记数组有几种：

标记	常见含义
`vis[i]`	点、数字、位置是否已经访问或使用
`flag[i]`	某个候选是否被占用
`col[i]`	第 i 列是否已有棋子
`row[i]`	第 i 行是否已有棋子
`ban[i]`	某个状态是否被禁止
`ok[i]`	某个状态是否可行

标记数组本身不难，关键是看它在递归前后怎样变化。

若标记在进入下一层前设为 `true`，在返回后设回 `false`，它通常表示当前路径中使用过的东西。

若标记设为 `true` 后不撤销，它可能表示全局已经访问过的点。例如图的 DFS 遍历中，一个点访问过后通常不需要在返回时重新设成未访问。

这两种情况要分清：

场景	返回后是否撤销
枚举排列、组合、棋盘放置	通常要撤销
图遍历中记录是否已经访问过	通常不撤销
统计所有路径，允许不同路径重复使用点	要按题意判断
搜索当前路径，禁止路径内重复点	通常要撤销

不要机械认为所有 DFS 都要撤销标记。要看标记表示“当前路径占用”，还是“全局已经处理”。

排列搜索

排列搜索是最典型的回溯。

问题：输出 1 到 n 的所有排列。

```
void dfs(int k) {
    if (k == n + 1) {
        for (int i = 1; i <= n; i++) {
            cout << p[i] << ' ';
        }

        cout << endl;
        return;
    }

    for (int i = 1; i <= n; i++) {
        if (vis[i]) {
            continue;
        }

        p[k] = i;
        vis[i] = true;

        dfs(k + 1);

        vis[i] = false;
    }
}
```

时间复杂度大致是 $O(n \cdot n!)$，因为共有 $n!$ 个排列，每个完整排列输出时需要 $O(n)$。如果只统计数量，不输出每个排列，递归结点数量仍然和 $n!$ 同阶。

第一轮中，排列 DFS 经常用来考：

- 某个条件删去后，会不会允许重复数字；
- 某个回溯语句删去后，答案怎样变化；
- k 的取值范围；
- 输入较小时输出多少。

例题：统计特殊排列

下面程序统计 1 到 n 的排列数，但要求相邻两个数不能满足“后一个数等于前一个数加 1”。

```
bool flag[30];
int n, p[30], ans;

void dfs(int k) {
    if (k == n + 1) {
        ans++;
        return;
    }

    for (int i = 1; i <= n; i++) {
        if (flag[i]) {
            continue;
        }

        if (k > 1 && i == p[k - 1] + 1) {
```

```
            continue;
        }

        p[k] = i;
        flag[i] = true;

        dfs(k + 1);

        flag[i] = false;
    }
}
```

读这段程序时，先看两条限制：

```
if (flag[i]) continue;
```

表示每个数字最多使用一次。

```
if (k > 1 && i == p[k - 1] + 1) continue;
```

表示从第二个位置开始，若当前数字恰好是前一个数字加 1，就不能放。

当 $n = 3$ 时，所有排列为：

排列	是否合法
$1, 2, 3$	不合法，$1, 2$ 相邻且 $2 = 1 + 1$
$1, 3, 2$	合法
$2, 1, 3$	合法
$2, 3, 1$	不合法，$2, 3$ 相邻且 $3 = 2 + 1$
$3, 1, 2$	不合法，$1, 2$ 相邻且 $2 = 1 + 1$
$3, 2, 1$	合法

所以输出为 3。

当 $n = 4$ 时，也可以逐层枚举，答案为 11。手算时建议不要直接列 24 个排列，可以按首位分组统计。

组合搜索

组合搜索常用于“从 n 个数中选 m 个”。

一种写法是用参数 pos 表示当前考虑到哪个数，用 cnt 表示已经选了几个。

```
void dfs(int pos, int cnt) {
    if (cnt == m) {
        ans++;
        return;
    }

    if (pos == n + 1) {
        return;
    }

    dfs(pos + 1, cnt + 1);
    dfs(pos + 1, cnt);
}
```

这里每个数有两种选择：

- 选当前数；
- 不选当前数。

这和 DP 中的“选或不选”很像，只是 DFS 会把所有可能路径都展开。

还有一种写法是直接枚举下一个选哪个数：

```
void dfs(int last, int cnt) {
    if (cnt == m) {
        ans++;
        return;
    }

    for (int i = last + 1; i <= n; i++) {
        choose[cnt + 1] = i;
        dfs(i, cnt + 1);
    }
}
```

这里 `last` 表示上一个选的数，下一次只能选更大的数，从而避免重复组合。

例题：从 5 个数中选 3 个

用第二种写法：

```
dfs(0, 0);
```

第一层可以选 $1,2,3,4,5$。若第一层选了 2，下一层只能选 $3,4,5$，不能回头选 1。这样每个组合只会被统计一次。

例如组合 $1,3,5$ 只会按 $1\to3\to5$ 这条路径出现，不会出现 $3\to1\to5$ 或 $5\to1\to3$。

组合搜索中，常用“下一个选择必须比上一个大”来去掉排列顺序造成的重复。

子集搜索

子集搜索可以看成组合搜索的变形。每个元素选或不选，共有 2^n 个子集。

代码如下：

```
void dfs(int pos) {
    if (pos == n + 1) {
        ans++;
        return;
    }

    use[pos] = true;
    dfs(pos + 1);

    use[pos] = false;
    dfs(pos + 1);
}
```

这里 `use[pos]` 表示第 `pos` 个元素是否被选入当前子集。

如果题目只统计子集数量，那么答案是 2^n。但如果有额外限制，比如和不超过 S，就可以在搜索过程中加条件或剪枝。

加入当前和

例如统计有多少个子集的元素和等于 S：

```
void dfs(int pos, int sum) {
    if (pos == n + 1) {
        if (sum == S) {
            ans++;
        }

        return;
    }

    dfs(pos + 1, sum + a[pos]);
    dfs(pos + 1, sum);
}
```

这里 sum 是当前已经选出的元素和。它作为参数传入下一层，不需要回溯恢复，因为每次递归调用都有自己的参数值。

如果使用全局变量 sum，则要手动加上和撤销：

```
sum += a[pos];
dfs(pos + 1);
sum -= a[pos];

dfs(pos + 1);
```

程序阅读时要分清：有些状态通过参数传递，不需要撤销；有些状态通过全局变量或数组记录，返回后要恢复。

棋盘搜索

棋盘类 DFS 常见于放棋子、走迷宫、八皇后等问题。它们通常有坐标、方向数组和标记数组。

以走迷宫为例：

```
int dx[4] = {1, -1, 0, 0};
int dy[4] = {0, 0, 1, -1};

void dfs(int x, int y) {
    if (x == tx && y == ty) {
        ans++;
        return;
    }

    for (int d = 0; d < 4; d++) {
        int nx = x + dx[d];
        int ny = y + dy[d];

        if (nx < 1 || nx > n || ny < 1 || ny > m) {
            continue;
        }

        if (a[nx][ny] == '#') {
            continue;
        }

        if (vis[nx][ny]) {
            continue;
```

```
        }

        vis[nx][ny] = true;
        dfs(nx, ny);
        vis[nx][ny] = false;
    }
}
```

这里：

- `dx,dy` 表示四个方向；
- 越界、障碍、已访问都不能走；
- `vis` 记录当前路径经过过的格子；
- 返回后撤销 `vis`，表示换一条路径时可以重新经过这个格子。

若题目是“求从起点能到达哪些格子”，则访问过的格子通常不撤销。若题目是“统计不重复经过格子的路径数”，就需要撤销。

八皇后式标记

在棋盘上每行放一个棋子，要求列和斜线不能冲突。常见标记如下：

```
bool col[N];
bool d1[N], d2[N];
```

若当前尝试在第 `r` 行、第 `c` 列放棋子，则：

- `col[c]` 表示第 c 列是否已有棋子；
- `d1[r+c]` 表示一类斜线是否已有棋子；
- `d2[r-c+n]` 表示另一类斜线是否已有棋子。

代码框架：

```
void dfs(int r) {
    if (r == n + 1) {
        ans++;
        return;
    }

    for (int c = 1; c <= n; c++) {
        if (col[c] || d1[r + c] || d2[r - c + n]) {
            continue;
        }

        col[c] = d1[r + c] = d2[r - c + n] = true;

        dfs(r + 1);

        col[c] = d1[r + c] = d2[r - c + n] = false;
    }
}
```

这一类程序常考标记下标。`r+c` 和 `r-c+n` 用来把同一条斜线上的格子映射到同一个编号。

学生不必在第一轮中展开八皇后的所有优化，但要能读懂：每行只放一个棋子，DFS 的层数就是行号。

剪枝

剪枝是指提前停止某些不可能得到答案或不可能更优的分支。

例如统计子集和是否能达到 S。如果所有数都是正数，当前和已经超过 S，继续往后加只会更大，可以停止：

```
if (sum > S) {
    return;
}
```

再如组合搜索中，要从剩余元素中再选若干个。如果剩下的数量不够，也可以停止：

```
if (cnt + (n - pos + 1) < m) {
    return;
}
```

这里 `cnt` 是已经选了多少个，`n-pos+1` 是从 `pos` 到 `n` 还剩多少个。如果全部选上也不够 m 个，就没有必要继续。

剪枝不能随便加。剪枝条件必须保证不会删掉合法答案。

错误剪枝

假设数组中可能有负数，想找子集和等于 S。如果写：

```
if (sum > S) {
    return;
}
```

就可能错误。

例如 $S=5$，当前和已经是 8，后面还有一个 -3。继续选择 -3 后可以回到 5。如果提前停止，就会漏掉答案。

所以 `sum>S` 这个剪枝只适用于后面不会让和变小的情况，比如所有剩余数都是非负数。

第一轮中，判断剪枝是否正确，必须看题目条件。

复杂度

DFS 的复杂度通常看搜索树有多少结点。

排列搜索：

```
dfs(1)
```

第 1 层有 n 种选择，第 2 层有 $n-1$ 种选择，直到第 n 层。所以完整方案数为 $n!$，递归结点数量也与 $n!$ 同阶。

子集搜索：

每个元素选或不选，共 2^n 种方案。

组合搜索：

从 n 个元素中选 m 个，共有 $\binom{n}{m}$ 个完整方案。

走迷宫：

如果不允许重复经过格子，最坏情况下路径数量可能增长很快。第一轮一般只要求能判断它不是简单的 $O(nm)$ 遍历，除非程序把每个格子访问一次后不撤销标记。

图遍历 DFS：

如果只是访问每个点、每条边一次，复杂度通常是 $O(n+m)$。这是图遍历，不是枚举所有路径。

类型	大致规模
排列搜索	$n!$
子集搜索	2^n
组合搜索	$\binom{n}{m}$
图 DFS 遍历	$O(n+m)$
网格 DFS 遍历	通常 $O(nm)$
枚举网格路径	可能远大于 $O(nm)$

看复杂度时，先判断程序是在“遍历每个点一次”，还是在“枚举很多方案”。这两类 DFS 差别很大。

真题式程序阅读：排列回溯

阅读下面程序：

```
bool flag[30];
int n, p[30], ans;

void dfs(int k) {
    if (k == n + 1) {
        ans++;
        return;
    }

    for (int i = 1; i <= n; i++) {
        if (flag[i]) {
            continue;
        }

        if (k > 1 && i == p[k - 1] + 1) {
            continue;
        }

        p[k] = i;
        flag[i] = true;
        dfs(k + 1);
        flag[i] = false;
    }
}
```

k 的取值

从 `dfs(1)` 开始，每次递归调用 `dfs(k+1)`。当 `k==n+1` 时立即返回。

所以在函数执行过程中，`k` 的取值会从 1 到 $n+1$。不会继续进入 `dfs(n+2)`，因为 `dfs(n+1)` 一开始就 `return`。

删除撤销语句

若删去：

```
flag[i] = false;
```

程序会出错。

因为某个数字在一条路径中使用后，返回上一层时没有释放。后面的分支会误以为这个数字仍然被使用，导致合法排列无法枚举。

p 的初值是否影响

在这段程序里，判断 `p[k-1]` 只有在 `k>1` 时才执行。进入 `dfs(k)` 时，`p[1]` 到 `p[k-1]` 已经由前面的选择赋值。`p[k]` 及后面的旧值不会被读到。

因此，调用 `dfs(1)` 前，`p` 数组的初值通常不会影响答案。

删去使用检查

若删去：

```
if (flag[i]) {
    continue;
}
```

程序就不再限制数字只能使用一次。此时每个位置都可以选择 1 到 n 中任意数字，只剩下相邻限制 `i != p[k-1]+1`。

以 $n=3$ 为例，可以逐层统计：

第一位有 3 种。

若上一位为 1，下一位不能为 2，有 2 种选择：$1,3$。

若上一位为 2，下一位不能为 3，有 2 种选择：$1,2$。

若上一位为 3，下一位没有数字等于 4，有 3 种选择：$1,2,3$。

可以继续追踪第 3 位，最终答案为 16。

这类修改题不要只看删掉的是哪一行，要重新判断搜索树分支数量。

真题式完善程序：排列统计

补全下面程序。统计 1 到 n 的所有排列数量。

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

const int N = 15;

int n, ans;
int p[N];
bool vis[N];

void dfs(int k) {
    if (__________) {
        ans++;
        return;
    }

    for (int i = 1; i <= n; i++) {
        if (__________) {
            continue;
        }

        p[k] = i;
        vis[i] = true;

        dfs(__________);
```

```
        _________;
    }
}

int main() {
    cin >> n;
    dfs(1);
    cout << ans << endl;
    return 0;
}
```

四个空依次应填：

```
k == n + 1
vis[i]
k + 1
vis[i] = false
```

解释：

- k==n+1 表示已经填完 n 个位置；
- vis[i] 表示数字 i 已经用过，不能再选；
- 下一层填写第 $k+1$ 个位置；
- 递归返回后撤销使用标记，方便尝试其他分支。

真题式完善程序：组合统计

补全下面程序。统计从 1 到 n 中选出 m 个数的方案数。

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

int n, m, ans;

void dfs(int last, int cnt) {
    if (__________) {
        ans++;
        return;
    }

    for (int i = ________; i <= n; i++) {
        dfs(__________, __________);
    }
}

int main() {
    cin >> n >> m;
    dfs(0, 0);
    cout << ans << endl;
    return 0;
}
```

四个空依次应填：

```
cnt == m
last + 1
i
```

```
cnt + 1
```

解释：

- `cnt==m` 表示已经选够 m 个数；
- 下一次只能选比 `last` 更大的数，避免同一个组合以不同顺序重复出现；
- 选择了 `i` 后，新的最后一个数是 `i`；
- 已选数量增加 1。

真题式完善程序：迷宫路径数

补全下面程序。给定一个 $n \times m$ 的网格，`.` 表示可走，`#` 表示障碍。从 `(sx,sy)` 出发，到 `(tx,ty)`，每条路径中不能重复经过格子，统计路径数。

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

const int N = 25;

int n, m;
int sx, sy, tx, ty;
char a[N][N];
bool vis[N][N];
int ans;

int dx[4] = {1, -1, 0, 0};
int dy[4] = {0, 0, 1, -1};

void dfs(int x, int y) {
    if (__________) {
        ans++;
        return;
    }

    for (int d = 0; d < 4; d++) {
        int nx = x + dx[d];
        int ny = y + dy[d];

        if (nx < 1 || nx > n || ny < 1 || ny > m) {
            continue;
        }

        if (__________) {
            continue;
        }

        if (__________) {
            continue;
        }

        vis[nx][ny] = true;
        dfs(nx, ny);
        __________;
    }
}

int main() {
    cin >> n >> m;
```

```
    for (int i = 1; i <= n; i++) {
        cin >> (a[i] + 1);
    }

    cin >> sx >> sy >> tx >> ty;

    vis[sx][sy] = true;
    dfs(sx, sy);

    cout << ans << endl;
    return 0;
}
```

四个空依次应填：

```
x == tx && y == ty
a[nx][ny] == '#'
vis[nx][ny]
vis[nx][ny] = false
```

解释：

- 到达终点时统计一条路径；
- 障碍不能走；
- 当前路径中已经走过的格子不能再走；
- 返回后撤销标记，允许其他路径使用这个格子。

如果题目只是判断终点能否到达，可以不撤销标记。但本题统计所有不重复路径，所以需要撤销。

常见错误

第一，看不清 `dfs(k)` 中 `k` 的含义。它可能表示位置、行号、当前点、当前考虑的元素，也可能表示已选数量。

第二，停止条件写错。排列搜索通常是 `k==n+1`，不是 `k==n`；若在填第 n 个位置前就统计，会少填一个位置。

第三，递归后忘记撤销标记。排列、组合、棋盘放置、路径枚举中，当前路径标记通常要回溯。

第四，所有 DFS 都机械撤销标记。图遍历中，若 `vis` 表示全局已经访问过，通常不撤销。

第五，把组合枚举写成排列枚举。从 n 个数中选 m 个时，若不限制下一个数大于上一个数，同一个组合会按不同顺序重复出现。

第六，剪枝条件不安全。只有确认被剪掉的分支不可能产生合法或更优答案，才能剪枝。

第七，复杂度判断混乱。图 DFS 遍历每点每边一次，可能是 $O(n+m)$；排列枚举是 $n!$ 级别，子集枚举是 2^n 级别。

第八，方向数组下标写错。网格 DFS 中，`dx` 和 `dy` 要成对使用，边界判断也要对应行列。

第九，参数传递和全局变量回溯混淆。通过参数传递的 `sum`、`cnt` 通常不需要手动恢复；全局数组和全局变量若被修改，就要检查是否需要恢复。

小结

DFS 与回溯的核心是搜索树。每一层表示一次选择，递归向下表示继续扩展，返回后撤销当前选择表示换另一条分支。

本章需要重点记住：

- `dfs` 参数要先读懂，它决定当前处在搜索树的哪一层；
- 停止条件表示已经得到完整方案或已经无法继续；
- 回溯通常由“做选择、递归、撤销选择”三步构成；
- 排列搜索中 `vis` 标记数字是否已用，返回后要撤销；
- 组合搜索常用递增选择避免重复；
- 子集搜索每个元素有选或不选两条分支；
- 迷宫路径计数中，当前路径的访问标记要撤销；
- 图遍历中的全局访问标记通常不撤销；
- 剪枝必须保证不漏掉正确答案；
- 排列、子集、组合、图遍历的复杂度级别不同，不能混为一谈。

下一章讲折半搜索。它仍然属于搜索，但不再直接把所有选择一路搜到底，而是把选择拆成两半分别处理，再用排序、查找或计数把两边合起来。

0x2E 折半搜索

折半搜索仍然属于搜索。它和普通 DFS 的区别在于：普通搜索可能把所有选择一路枚举到底；折半搜索把选择拆成前后两半，分别枚举，再把两边的结果合并。这样可以把原本约为 2^n 的枚举，降到约 $2^{n/2}$ 级别。

CSP-S 第一轮中，折半搜索可能以程序阅读形式出现：

- 程序把数组分成两段；
- 分别 DFS 出两个答案数组；
- 对其中一个数组排序；
- 用二分、双指针或计数合并答案；
- 复杂度选项中出现 $2^{n/2}$ 或 $m^{n/2}$。

读到这种结构，就要想到折半搜索。

问题

给定 n 个整数 $a_1, a_2, \dots, a_n$，问是否存在一个子集，使子集元素和等于 S。

如果直接枚举每个元素选或不选，一共有 2^n 种情况。

当 $n = 40$ 时，2^{40} 已经很大。若把 40 个数分成两半，每半 20 个数，每一半分别枚举子集和，就各有 2^{20} 种。然后再把两边的子集和配对，判断是否能拼出 S。

这就是折半搜索的基本想法。

它不是把问题改成两个互不相关的小问题，而是先把所有“一半的选择”列出来，再在合并时恢复完整选择。

直接枚举

先看普通搜索。

```
void dfs(int pos, long long sum) {
    if (pos == n + 1) {
        if (sum == S) {
            ok = true;
        }

        return;
    }

    dfs(pos + 1, sum);
    dfs(pos + 1, sum + a[pos]);
}
```

每个数都有选和不选两种分支，所以完整方案数是 2^n。

如果 n 较小，这样写很直接。但如果 n 在 40 左右，直接枚举会很慢。

折半搜索的目标就是避免一次性枚举 2^n 个完整方案。

拆成两半

把数组分成两段：

- 左半段：$a_1, \dots, a_{mid}$；
- 右半段：$a_{mid+1}, \dots, a_n$。

分别枚举两半的所有子集和。

```
void dfs1(int pos, int end, long long sum) {
    if (pos == end + 1) {
        L.push_back(sum);
        return;
    }

    dfs1(pos + 1, end, sum);
    dfs1(pos + 1, end, sum + a[pos]);
}

void dfs2(int pos, int end, long long sum) {
    if (pos == end + 1) {
        R.push_back(sum);
        return;
    }

    dfs2(pos + 1, end, sum);
    dfs2(pos + 1, end, sum + a[pos]);
}
```

如果左右两半长度接近，`L` 和 `R` 的大小都约为 $2^{n/2}$。

枚举完后，一个完整子集的和可以写成：

左半某个子集和 + 右半某个子集和。

所以问题变成：是否存在 $x \in L$ 和 $y \in R$，使 $x + y = S$。

排序与二分

最常见的合并方式是：把右半部分的子集和排序，然后枚举左半部分的每个和 x，在右半部分中查找 $S - x$。

```
sort(R.begin(), R.end());

for (long long x : L) {
    if (binary_search(R.begin(), R.end(), S - x)) {
        ok = true;
        break;
    }
}
```

复杂度分成三部分：

步骤	复杂度
枚举左半子集和	$O(2^{n/2})$
枚举右半子集和	$O(2^{n/2})$
排序并二分查找	$O(2^{n/2} \log 2^{n/2})$

总体复杂度通常写作 $O(2^{n/2} \log 2^{n/2})$。

在第一轮中，复杂度选项常写成 $O(2^{n/2}\log 2^{n/2})$，也可能写成 $O(n2^{n/2})$。因为 $\log 2^{n/2}=n/2$，两种写法在量级上可以对应。

手算例子

给定数组 $2,4,7,9$，目标和 $S=11$。

分成两半：

- 左半：$2,4$；
- 右半：$7,9$。

左半所有子集和：

选择	和
不选	0
选 2	2
选 4	4
选 $2,4$	6

所以 $L=\{0,2,4,6\}$。

右半所有子集和：

选择	和
不选	0
选 7	7
选 9	9
选 $7,9$	16

所以 $R=\{0,7,9,16\}$。

枚举左半：

- 若 $x=0$，需要右半有 11，没有；
- 若 $x=2$，需要右半有 9，有；
- 找到方案 $2+9=11$。

所以存在合法子集。

计数问题

折半搜索不只能判断是否存在，也能统计方案数。

问题：给定 n 个数，统计有多少个子集和等于 S。

仍然枚举左右两半的子集和。把右半子集和排序。对于左半每个 x，需要统计右半中有多少个 $S-x$。

可以用 `lower_bound` 和 `upper_bound`：

```
sort(R.begin(), R.end());

long long ans = 0;

for (long long x : L) {
    long long need = S - x;
```

```
    auto l = lower_bound(R.begin(), R.end(), need);
    auto r = upper_bound(R.begin(), R.end(), need);

    ans += r - l;
}
```

这里 r-l 表示右半子集和中等于 need 的个数。

重复值不能去掉

如果是统计方案数，不能把子集和数组简单去重。

例如右半有多个不同子集的和都等于 5，它们与同一个左半子集和配对，会形成多个不同的完整方案。

判断是否存在时，去重不影响真假；统计方案数时，重复次数就是答案的一部分。

这是程序阅读中很容易被忽略的点。

双指针合并

如果要统计满足 $x + y \le S$ 的方案数，可以把左右两边都排序，用双指针。

```
sort(L.begin(), L.end());
sort(R.begin(), R.end());

long long ans = 0;
int j = (int)R.size() - 1;

for (int i = 0; i < (int)L.size(); i++) {
    while (j >= 0 && L[i] + R[j] > S) {
        j--;
    }

    ans += j + 1;
}
```

解释：

- L 从小到大枚举；
- j 表示右半中最大的可配位置；
- 若 L[i]+R[j]>S，右边太大，需要让 j 左移；
- 当条件满足时，R[0] 到 R[j] 都可以与 L[i] 配对。

因为 L[i] 逐渐变大，j 只会往左移动，不会反复回退，所以合并过程是线性的。

整体复杂度为排序的 $O(2^{n/2} \log 2^{n/2})$ 加上线性扫描。

真题式程序阅读：拆半枚举

阅读下面代码片段：

```
vector<long long> A, B;

void dfs(vector<long long>& v, int l, int r, long long sum) {
    if (l > r) {
        v.push_back(sum);
        return;
```

```
    }

    dfs(v, l + 1, r, sum);
    dfs(v, l + 1, r, sum + a[l]);
}

int main() {
    int mid = n >> 1;

    dfs(A, 1, mid, 0);
    dfs(B, mid + 1, n, 0);

    sort(B.begin(), B.end());

    long long ans = 0;

    for (long long x : A) {
        long long need = S - x;
        ans += upper_bound(B.begin(), B.end(), need)
             - lower_bound(B.begin(), B.end(), need);
    }

    cout << ans << endl;
}
```

这段程序求什么？

`dfs(A,1,mid,0)` 枚举左半所有子集和，`dfs(B,mid+1,n,0)` 枚举右半所有子集和。

后面对每个左半子集和 `x`，在右半中统计等于 `S-x` 的子集和数量。每一对左半选择和右半选择可以合成一个完整子集。

所以程序统计的是：原数组中子集和等于 S 的方案数。

注意，程序没有对 `B` 去重，因为重复的子集和对应不同子集，统计方案数时需要保留。

多值选择

有些题不是每个元素选或不选，而是每个变量可以取多个值。

例如有方程：

$k_1x_1^{p_1} + k_2x_2^{p_2} + \cdots + k_nx_n^{p_n} = 0$，

其中每个 x_i 可以取 1 到 m 的整数。若直接枚举所有变量，有 m^n 种情况。

折半搜索可以把变量分成两半：

- 枚举左半所有表达式值；
- 枚举右半所有表达式值；
- 统计两边相加为 0 的组合数。

如果两半变量数量接近，每边枚举规模约为 $m^{n/2}$。

这种形式在程序阅读题中经常出现。代码里可能有一个 DFS，参数不是“选或不选”，而是枚举当前变量取 1 到 m 的所有值：

```
void dfs(vector<int>& res, int l, int r, int val) {
    if (l > r) {
        res.push_back(val);
```

```
        return;
    }

    for (int x = 1; x <= m; x++) {
        dfs(res, l + 1, r, val + k[l] * power(x, p[l]));
    }
}
```

外层看起来是搜索，合并时则会排序和统计相反数。

真题式例题：方程计数

有如下形式的程序：

```
dfs(ans1, 1, n / 2, 0);
dfs(ans2, n / 2 + 1, n, 0);

sort(ans1.begin(), ans1.end());

long long ans = 0;

for (int i = 0; i < (int)ans2.size(); i++) {
    int x = -ans2[i];

    ans += upper_bound(ans1.begin(), ans1.end(), x)
         - lower_bound(ans1.begin(), ans1.end(), x);
}
```

若 `dfs` 枚举的是若干变量在 1 到 m 中取值时产生的表达式值，那么这段代码统计的是什么？

它统计的是左半表达式值加右半表达式值等于 0 的方案数。也就是整个方程取值为 0 的解的数量。

这里排序的是 `ans1`，枚举的是 `ans2`。当然也可以反过来，排序 `ans2`，枚举 `ans1`。两种写法的本质相同。

去重与计数数组

若一个数组中有很多重复值，为了加快合并，程序可能先排序并压缩。

例如：

```
sort(A.begin(), A.end());

vector<long long> val;
vector<int> cnt;

for (int i = 0; i < (int)A.size(); i++) {
    if (val.empty() || val.back() != A[i]) {
        val.push_back(A[i]);
        cnt.push_back(1);
    } else {
        cnt.back()++;
    }
}
```

压缩后：

- `val[k]` 表示某个子集和值；
- `cnt[k]` 表示这个和值出现了多少次。

统计时，若右半某个值需要配对 `val[k]`，贡献不是 1，而是 `cnt[k]`。

例如左半子集和中，值 5 出现了 3 次；右半当前值为 7，目标 $S = 12$。那么这一个右半选择可以和左半 3 个不同选择配对，贡献 3。

程序阅读题中，若看到排序后把相同值合并，再额外维护次数数组，就要判断答案是否在累加次数。

复杂度

折半搜索的复杂度一般由三部分组成：

- 分别枚举两半；
- 排序其中一半或两半；
- 用二分或双指针合并。

以子集和为例，两半大小约为 $2^{n/2}$。

枚举复杂度为 $O(2^{n/2})$，排序复杂度为 $O(2^{n/2}\log 2^{n/2})$，每次二分又会带来一个对数。

所以常见总复杂度为：

$O(2^{n/2}\log 2^{n/2})$。

如果每个位置有 m 种选择，而不是选或不选，则两半规模约为 $m^{n/2}$，复杂度常写成：

$O(m^{n/2}\log m^{n/2})$。

在第一轮中，若选项里同时有 $O(m^n)$ 和 $O(m^{n/2}\log m^{n/2})$，并且程序确实把变量拆成两半枚举，再排序合并，应选后者。

例题：复杂度判断

下面程序把 n 个变量分成两半，每个变量枚举 1 到 m，计算所有表达式值。

```
dfs(A, 1, n / 2, 0);
dfs(B, n / 2 + 1, n, 0);
sort(A.begin(), A.end());

for (int x : B) {
    ...
}
```

若暂不考虑表达式中快速幂的代价，`A` 的规模约为 $m^{n/2}$，`B` 的规模约为 $m^{n/2}$。

排序 `A` 的复杂度为 $O(m^{n/2}\log m^{n/2})$。枚举 `B` 并合并，若每次二分，也是 $O(m^{n/2}\log m^{n/2})$。

所以整体复杂度约为 $O(m^{n/2}\log m^{n/2})$。

如果计算每个表达式项还需要额外代价，例如快速幂需要 $O(\log P)$，复杂度中还要乘上或加上相应因素。第一轮题中，选项常会把这一点也放进去。

和二分的区别

折半搜索中常用二分，但它不是第 8 章讲的二分答案。

二分答案是对答案范围二分，每次检查某个候选答案是否可行。

折半搜索中的二分，是在已经枚举出的数组中查找配对值。

例如：

```
binary_search(R.begin(), R.end(), S - x)
```

这里二分的是数组 R 中是否存在某个值，不是在猜答案。

所以看到 lower_bound、upper_bound 时，要结合上下文判断：它是在查找配对，还是在二分答案。

和分治的区别

折半搜索和分治都把问题分成两半，但目的不同。

分治通常把一个问题拆成若干个同类子问题，分别求解后合并。例如归并排序、分治求最大子段和。

折半搜索通常不是求 “左半答案” 和 “右半答案” 后直接合并成同类答案，而是枚举两边所有可能的状态，再配对。

可以这样区分：

方法	拆分后做什么
分治	求左右子问题的答案，再合并答案
折半搜索	枚举左右两边所有状态，再查找或计数配对
普通 DFS	直接枚举完整方案
DP	保存状态并复用子问题结果

折半搜索的关键是 “状态集合配对”。

和普通 DFS 的区别

普通 DFS 枚举完整方案：

```
第 1 个选不选
第 2 个选不选
...
第 n 个选不选
```

折半搜索枚举半个方案：

```
左半所有选择 -> 得到 L
右半所有选择 -> 得到 R
在 L 和 R 中找能拼成答案的组合
```

它没有减少所有半边状态的枚举，但避免了直接产生 2^n 个完整组合。

当 n 在 40 左右时，2^{20} 级别的数组还可以处理，而 2^{40} 级别的完整枚举很难接受。这正是折半搜索的使用场景。

空集

折半搜索枚举子集和时，空集一定要包含进去。

例如左半什么都不选，子集和为 0；右半如果正好能凑出 S，这也是合法方案。

因此 DFS 初始调用一般是：

```
dfs(1, mid, 0);
```

并且当到达边界时，会把当前 sum 放入数组：

```cpp
if (l > r) {
    v.push_back(sum);
    return;
}
```

若漏掉空集，会少统计只从另一半选元素的方案。

例题：只在右半取数

数组 $2, 5, 8, 10$，目标 $S = 18$。分成左半 $2, 5$，右半 $8, 10$。

合法子集可以只选右半的 8 和 10，左半什么都不选。

若左半枚举时不包含空集和 0，就无法统计这个方案。

所以折半枚举中，空选择是必须保留的。

负数

如果数组中有负数，折半搜索仍然可以使用。

例如子集和中，`L` 和 `R` 可以包含负数。排序和二分依然有效。

但一些基于单调性的剪枝可能失效。例如普通 DFS 中，如果所有数都是非负数，当前和已经超过 S 可以停止；有负数时，后面还可能把和减回来，不能这样剪枝。

折半搜索对负数比较友好，因为它不依赖“和只会增加”的性质，而是把所有半边可能值枚举出来再配对。

真题式完善程序：是否存在子集和

补全下面程序。给定 n 个整数和目标 S，判断是否存在一个子集使元素和等于 S。

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

const int N = 45;

int n;
long long S;
long long a[N];
vector<long long> L, R;

void dfs(vector<long long>& v, int l, int r, long long sum) {
    if (__________) {
        v.push_back(sum);
        return;
    }

    dfs(v, l + 1, r, sum);
    dfs(v, l + 1, r, __________);
}

int main() {
    cin >> n >> S;

    for (int i = 1; i <= n; i++) {
        cin >> a[i];
    }
```

```
    int mid = n >> 1;

    dfs(L, 1, mid, 0);
    dfs(R, mid + 1, n, 0);

    sort(R.begin(), R.end());

    bool ok = false;

    for (long long x : L) {
        if (__________) {
            ok = true;
            break;
        }
    }

    cout << (ok ? "Yes" : "No") << endl;
    return 0;
}
```

三个空依次应填：

```
l > r
sum + a[l]
binary_search(R.begin(), R.end(), S - x)
```

解释：

- 当 `l>r` 时，当前半边已经处理完，把 `sum` 放入数组；
- 选当前数时，新的和为 `sum+a[l]`；
- 对每个左半子集和 `x`，在右半中找 `S-x`。

真题式完善程序：统计方案数

补全下面程序。给定 n 个整数和目标 S，统计子集和等于 S 的方案数。

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

const int N = 45;

int n;
long long S;
long long a[N];
vector<long long> L, R;

void dfs(vector<long long>& v, int l, int r, long long sum) {
    if (l > r) {
        v.push_back(sum);
        return;
    }

    dfs(v, l + 1, r, sum);
    dfs(v, l + 1, r, sum + a[l]);
}

int main() {
    cin >> n >> S;
```

```
    for (int i = 1; i <= n; i++) {
        cin >> a[i];
    }

    int mid = n >> 1;

    dfs(L, 1, mid, 0);
    dfs(R, mid + 1, n, 0);

    sort(R.begin(), R.end());

    long long ans = 0;

    for (long long x : L) {
        long long need = __________;

        ans += __________ - __________;
    }

    cout << ans << endl;
    return 0;
}
```

三个空依次应填：

```
S - x
upper_bound(R.begin(), R.end(), need)
lower_bound(R.begin(), R.end(), need)
```

解释：

- 右半需要的和为 S-x；
- upper_bound-lower_bound 得到等于 need 的元素个数；
- 重复值不能去掉，因为每个重复值对应不同选择。

真题式程序阅读：计数压缩

阅读下面代码：

```
sort(A.begin(), A.end());

vector<long long> val;
vector<int> cnt;

for (int i = 0; i < (int)A.size(); i++) {
    if (val.empty() || val.back() != A[i]) {
        val.push_back(A[i]);
        cnt.push_back(1);
    } else {
        cnt.back()++;
    }
}

long long ans = 0;

for (long long x : B) {
    long long need = S - x;
```

```
    int p = lower_bound(val.begin(), val.end(), need) - val.begin();

    if (p < (int)val.size() && val[p] == need) {
        ans += cnt[p];
    }
}
```

这段程序先把 A 中相同的值压缩到一起，并记录出现次数。后面枚举 B 中的每个值 x，查找 A 中是否有 S-x。

若有，贡献是 cnt[p]，不是 1。这说明它统计的是方案数，而不只是判断是否存在。

如果题目只问是否存在，压缩后不需要 cnt，只要找到 need 即可。

易错点

第一，把折半搜索当成分治。折半搜索的核心是枚举两边状态后配对，不是分别求左右答案。

第二，漏掉空集。子集和中，什么都不选的和 0 必须放入数组，否则会漏掉只从另一半选择的方案。

第三，统计方案数时去重。重复的子集和可能来自不同子集，不能简单删除。

第四，判断存在和统计方案数混淆。存在性可以用 binary_search，计数要用 lower_bound 和 upper_bound 或次数数组。

第五，复杂度写成 $O(2^n)$。折半搜索仍然是指数级，但通常是 $O(2^{n/2}\log 2^{n/2})$ 级别。

第六，变量有多种取值时仍写成 $2^{n/2}$。若每个变量有 m 种取值，规模应是 $m^{n/2}$。

第七，二分的对象看错。折半搜索中的二分是在半边结果数组中找配对值，不是在二分答案。

第八，忽略计算表达式的代价。如果枚举时每一步还要计算快速幂，复杂度中可能还要带上 $\log P$。

第九，只排序一边却在另一边二分。二分查找的数组必须有序。若代码对 A 排序，就应在 A 中二分；若对 B 排序，就应在 B 中二分。

小结

折半搜索适合处理直接枚举过大、但拆成两半后还能接受的问题。它常见于子集和、方程计数、多值选择等场景。

本章需要重点记住：

- 折半搜索把选择拆成两半，分别枚举，再合并；
- 子集问题直接枚举是 2^n，折半后每边约为 $2^{n/2}$；
- 多值选择若每个变量有 m 种取值，每边约为 $m^{n/2}$；
- 合并时常对一边排序，再用二分查找配对值；
- 判断存在可用 binary_search；
- 统计方案数常用 upper_bound-lower_bound；
- 重复的半边结果不能随便删除，除非只判断存在性；
- 空集对应的和 0 必须保留；
- 折半搜索中的二分不是答案二分；
- 折半搜索和分治的区别在于：前者配对状态集合，后者合并子问题答案。

第七编到这里结束。下一章开始进入第八编“程序阅读专项”。前面各章已经讲过复杂度、递归、二分、数据结构、图论、字符串、DP、贪心和搜索；后续章节会把这些知识放回程序阅读题中，训练如何追踪变量、判断循环、读递归、看数组和字符串变化。

第八编 程序阅读专项

0x2F 变量追踪

从本章开始进入第八编“程序阅读专项”。前面章节已经讲过复杂度、递归、分治、二分、数据结构、图论、字符串、DP、贪心和搜索；这一编不再把这些知识点重讲一遍，而是把重点放在“怎样在试卷上读懂程序”。

变量追踪是程序阅读的第一项基本功。很多程序看起来很长，但真正决定答案的变量并不多。读程序时，如果把所有变量都追一遍，纸面很快就乱；如果只凭感觉跳过变量变化，又很容易错在边界、更新顺序和递归恢复上。

变量追踪要解决的问题是：哪些变量值得追，追到什么程度，怎样用较少的记录得到可靠结论。

变量分层

程序中的变量可以分成三类。

第一类是控制变量。它们决定程序走到哪里，例如循环变量 `i,j`，二分边界 `l,r,mid`，递归参数 `k,pos,l,r`。

第二类是状态变量。它们记录程序当前处在什么状态，例如 `vis[i]`、`p[k]`、`cnt`、`sum`、`dp[i]`、`q` 中的元素。

第三类是答案变量。它们直接或间接决定输出，例如 `ans`、`res`、`ret`，或者最后被输出的数组、字符串、结构体成员。

追踪时，不是变量越多越好。应先找答案变量，再沿着答案变量往前看：谁修改了答案，修改答案时依赖哪些变量，这些变量又由谁决定。

例如：

```
int ans = 0;

for (int i = 1; i <= n; i++) {
    if (a[i] > a[i - 1]) {
        ans++;
    }
}
```

这里真正要追的是 `i`、`a[i] > a[i - 1]` 的真假和 `ans`。若题目只问输出，不需要记录所有 `a[i]` 被访问过几次；若题目问比较次数，就要记录条件判断执行了多少次。

变量追踪的第一步不是画表，而是判断表里应该有哪些列。

变量表

读一段较长程序时，可以先写一个变量表。变量表不用大，三列通常够用：变量、含义、改变位置。

变量	含义	改变位置
`i`	当前循环位置	`for` 每轮更新
`j`	当前窗口左端	`while` 中增加
`s`	已统计的数量	每轮执行 `s += i - j`
`a[i]`	排序后的第 i 个数	排序后不再改变

变量表的作用不是装饰，而是防止把变量含义读错。

看下面的程序片段：

```cpp
int s = 0;

for (int i = 0, j = 0; i < n; i++) {
    while (a[i] - a[j] > m) {
        j++;
    }

    s += i - j;
}
```

如果只看 `for`，会以为内层 `while` 每轮都可能执行很多次；但写出变量表后，`j` 的含义很清楚：它是当前满足条件的最小左端点，只会增加，不会回到前面。于是可以判断，这段程序的总推进次数与 n 同阶，而不是每轮重新从头找。

变量表还可以帮助处理判断题。若题目问“将 `j++` 改为 `i++` 是否不影响结果”，就能立刻回到变量表中看 `j` 的含义，而不是只凭代码外形猜。

小表追踪

遇到具体输入时，常用一张小表记录每轮关键变量。

例如：

```cpp
int x = 1, y = 0;

for (int i = 1; i <= 4; i++) {
    y += x;
    x += i;
}
cout << x << ' ' << y << endl;
```

追踪表可以写成：

轮次	执行前 `x`	执行前 `y`	执行 `y += x` 后	执行 `x += i` 后
$i=1$	1	0	1	2
$i=2$	2	1	3	4
$i=3$	4	3	7	7
$i=4$	7	7	14	11

最终输出 `11 14`。

表格的列要按语句执行顺序来写。本例中先改 `y`，后改 `x`。如果把两句顺序看反，答案就会错。

程序阅读题常考“更新顺序”。例如 `x += y; y += x;` 和 `y += x; x += y;` 的结果不同。追踪表应该直接体现这种顺序差异。

不要追太满

有些程序里数组很长，不能把每个元素都写进表格。数组追踪要抓住“被修改的位置”和“被查询的位置”。

看下面的代码：

```cpp
for (int i = 1; i <= n; i++) {
    b[a[i]]++;
}
```

```
int ans = 0;

for (int i = 1; i <= n; i++) {
    if (b[i] > 0) {
        ans++;
    }
}
```

如果输入为：

```
n = 6
a = {2, 4, 2, 5, 4, 4}
```

不需要写出完整的 `b[1..n]` 每一轮变化，只记录被修改过的位置：

读入的数	改变
2	`b[2]` 从 0 到 1
4	`b[4]` 从 0 到 1
2	`b[2]` 从 1 到 2
5	`b[5]` 从 0 到 1
4	`b[4]` 从 1 到 2
4	`b[4]` 从 2 到 3

最后 `b[i] > 0` 的位置是 $2, 4, 5$，所以 `ans=3`。

数组追踪时要注意两件事：

- 数组下标从 0 开始还是从 1 开始；
- 数组存的是“值本身”，还是“值出现的位置、次数、状态”。

例如 `b[a[i]]++` 不是修改 `b[i]`，而是修改下标为 `a[i]` 的位置。这类间接下标在程序阅读中很常见。

真题式例题：回溯变量

阅读下面程序：

```
bool flag[10];
int n, p[10];
int ans = 0;

void dfs(int k) {
    if (k == n + 1) {
        ans++;
        return;
    }

    for (int i = 1; i <= n; i++) {
        if (flag[i]) {
            continue;
        }

        if (k > 1 && i == p[k - 1] + 1) {
            continue;
        }
```

```
            p[k] = i;
            flag[i] = true;
            dfs(k + 1);
            flag[i] = false;
        }
    }

    int main() {
        cin >> n;
        dfs(1);
        cout << ans << endl;
        return 0;
    }
```

这个程序中的关键变量表如下：

变量	含义
k	当前要填写的位置
i	当前尝试放入第 k 个位置的数字
p[k]	第 k 个位置填入的数字
flag[i]	数字 i 是否已经在当前排列中使用
ans	合法排列数量

限制有两条：

- flag[i] 为真时不能再使用数字 i；
- 从第二个位置开始，不能让当前数字等于前一个数字加 1。

当 $n = 3$ 时，可以直接列排列：

排列	是否计数	原因
$1, 2, 3$	否	$2 = 1 + 1$
$1, 3, 2$	是	满足限制
$2, 1, 3$	是	满足限制
$2, 3, 1$	否	$3 = 2 + 1$
$3, 1, 2$	否	$2 = 1 + 1$
$3, 2, 1$	是	满足限制

所以输出为 3。

这道题不必把完整递归树画出来。因为 $n = 3$ 很小，列出所有排列更省纸面，也更不容易漏。

如果题目问删掉 flag[i] = false 会怎样，就不能只看最终排列。要看 flag 的含义：它表示“当前路径中已经使用的数字”。递归返回后，当前数字应当从当前路径中撤销。若不撤销，前一个分支用过的数字会影响后一个分支，答案会改变。

如果题目问 k 的取值范围，则看入口和停止条件：

- 主函数调用 dfs(1)；
- 每次递归调用 dfs(k + 1)；
- 当 k == n + 1 时返回，不再继续向下。

所以实际进入 dfs 时，k 可能取 $1, 2, \ldots, n + 1$。

递归追踪不一定画树

递归程序有时适合画树，有时适合写状态表。

若递归是排列、组合、DFS 搜索，画出少量分支可以帮助判断回溯。

若递归是分治，例如：

```
int solve(int l, int r) {
    if (l == r) {
        return a[l];
    }

    int mid = (l + r) / 2;
    return solve(l, mid) + solve(mid + 1, r);
}
```

更适合追踪区间。

当 $n = 4$，初始调用 `solve(1,4)`，递归区间为：

调用	分裂成
`solve(1,4)`	`solve(1,2)` 和 `solve(3,4)`
`solve(1,2)`	`solve(1,1)` 和 `solve(2,2)`
`solve(3,4)`	`solve(3,3)` 和 `solve(4,4)`

返回值是各个叶子 `a[1]` 到 `a[4]` 的和。

如果题目问时间复杂度，就看每层有多少区间、每层合并做多少事。若合并只做常数次加法，总体是 $O(n)$；若每层还要扫描当前区间，总体可能是 $O(n \log n)$。

不要所有递归都画成完整树。分治程序优先追区间，搜索程序优先追选择，树形程序优先追父子关系。

函数返回值

程序阅读中常有多个函数。读函数时，要先弄清它是“返回一个值”，还是“修改外部状态”。

看下面的代码：

```
int add(int x) {
    x++;
    return x;
}

int main() {
    int a = 5;
    add(a);
    cout << a << endl;
}
```

`add(a)` 的返回值没有被接住，`a` 不会改变，输出仍是 5。

再看：

```
int add(int x) {
    x++;
    return x;
}
```

```cpp
int main() {
    int a = 5;
    a = add(a);
    cout << a << endl;
}
```

这次输出 6。

再看引用参数：

```cpp
void add(int& x) {
    x++;
}

int main() {
    int a = 5;
    add(a);
    cout << a << endl;
}
```

这里 x 是引用，函数内部修改 x 会改变 a，输出 6。

函数追踪时可以先问三个问题：

问题	作用
返回值有没有被使用	判断函数结果是否影响后续
参数是否带 &	判断函数内部修改是否保留
函数是否修改全局变量	判断没有返回值时是否仍有影响

很多判断题会把这三点混在一起考。

更新位置

变量什么时候更新，往往比变量叫什么更重要。

看下面的二分片段：

```cpp
while (l < r) {
    int mid = (l + r) / 2;

    if (check(mid)) {
        r = mid;
    } else {
        l = mid + 1;
    }
}
```

追踪时要记录的是 `l,r,mid` 和 `check(mid)` 的结果。

例如初始 `l=1,r=8`，并设 `check(x)` 在 $x \geq 5$ 时为真：

轮次	l	r	mid	check(mid)	更新后
1	1	8	4	假	l=5,r=8
2	5	8	6	真	l=5,r=6
3	5	6	5	真	l=5,r=5

最后答案是 5。

如果把 `r = mid` 改成 `r = mid - 1`，这套边界就变了；如果把 `l = mid + 1` 改成 `l = mid`，还可能导致循环无法推进。二分程序阅读必须追边界，不能只看“像不像二分”。

本章只要求掌握变量追踪的写法，二分细节会在完善程序专项中继续出现。

输出前最后一次变化

有些程序中，变量在输出前还会被改变一次。读题时要特别看输出语句前面的几行。

例如：

```
int ans = 0;

for (int i = 1; i <= n; i++) {
    ans += i;
}

ans /= 2;

cout << ans << endl;
```

如果只追到循环结束，会得到 $ans = 1 + 2 + \cdots + n$。但真正输出的是除以 2 后的值。

再如：

```
cout << ans++ << endl;
```

输出的是自增前的 `ans`。

而：

```
cout << ++ans << endl;
```

输出的是自增后的 `ans`。

这类语法不难，但考场上很容易因为急着计算主流程而忽略。变量追踪表最好在最后加一行“输出时”，确认输出语句使用的是哪个时刻的值。

选项反推

选择题不一定要完整求出程序所有行为。有时可以用变量范围排除选项。

例如程序中有：

```
int ans = 0;

for (int i = 1; i <= n; i++) {
    if (a[i] % 2 == 0) {
        ans++;
    }
}
```

若题目问 `ans` 的可能值，先由变量含义得到 `ans` 是偶数个数，因此 $0 \le ans \le n$。所有小于 0 或大于 n 的选项都可以排除。

再如：

```
int x = 1;

for (int i = 1; i <= n; i++) {
    x *= 2;
}
```

循环结束后 x 是 2^n。若选项里有奇数，可以直接排除。

反推不是猜答案，而是利用变量含义和范围缩小判断量。程序阅读题的选项常常设置一些很离谱的干扰项，先排除它们，可以减少后面的计算压力。

变式题：更新顺序

阅读下面程序，判断输出。

```
int a = 2, b = 3;

for (int i = 1; i <= 3; i++) {
    a += b;
    b += a;
}

cout << a << ' ' << b << endl;
```

追踪表如下：

轮次	执行前 a	执行前 b	a += b 后	b += a 后
$i=1$	2	3	5	8
$i=2$	5	8	13	21
$i=3$	13	21	34	55

输出为：

```
34 55
```

若把循环体改成：

```
b += a;
a += b;
```

结果会变成：

轮次	执行前 a	执行前 b	b += a 后	a += b 后
$i=1$	2	3	5	7
$i=2$	7	5	12	19
$i=3$	19	12	31	50

输出为：

```
50 31
```

同样是两句加法，顺序不同，变量值不同。追踪表必须按照真实执行顺序填写。

变式题：数组间接下标

阅读下面程序：

```
int n = 5;
int a[6] = {0, 2, 1, 2, 3, 1};
int b[6] = {0};

for (int i = 1; i <= n; i++) {
    b[a[i]] += i;
}

for (int i = 1; i <= 3; i++) {
    cout << b[i] << ' ';
}
```

逐项追踪：

i	a[i]	修改
1	2	b[2] += 1
2	1	b[1] += 2
3	2	b[2] += 3
4	3	b[3] += 4
5	1	b[1] += 5

最后：

- b[1]=2+5=7；
- b[2]=1+3=4；
- b[3]=4。

输出为：

```
7 4 4
```

这里最容易错的地方是把 b[a[i]] 看成 b[i]。间接下标要先算出下标，再修改对应位置。

变式题：引用参数

阅读下面程序：

```
void work(int x, int& y) {
    x += 2;
    y += x;
}

int main() {
    int a = 1, b = 3;
    work(a, b);
    cout << a << ' ' << b << endl;
}
```

x 是按值传递，y 是引用。

调用 work(a,b) 时，函数内部一开始 x=1,y 对应主函数中的 b。

执行：

- `x += 2` 后，函数内部 `x=3`，主函数中的 `a` 不变；
- `y += x` 后，主函数中的 `b` 增加 3，变为 6。

输出为：

```
1 6
```

如果题目把函数头改成：

```
void work(int& x, int& y)
```

那么 `x += 2` 会改变主函数中的 `a`，输出就会变成 `3 6`。

函数参数中有没有 `&`，是程序阅读中必须看的细节。

小结

做变量追踪题时，可以按下面的顺序检查：

检查点	要问的问题
输出	最后输出的是哪个变量或表达式
修改	这个变量在哪里被修改
控制	循环和递归由哪些变量控制
状态	哪些数组或标记会影响分支
顺序	同一轮中变量修改的先后是否影响结果
下标	数组下标从 0 还是 1 开始，是否有间接下标
函数	返回值是否使用，参数是否引用，是否修改全局变量
最后一刻	输出前变量是否又被改变

变量追踪不是把程序抄成表格，而是把程序中真正会影响答案的变化记录下来。题目越复杂，越要先缩小追踪范围；输入越小，越可以直接列状态。能把变量含义、改变位置和输出时刻说清楚，大多数程序阅读小问都会变得稳定。

0x30 循环程序

循环程序在程序阅读中很常见。它的难点通常不在语法，而在“到底执行了几次”“哪一次变量被改掉”“break 或 continue 会跳过哪些语句”。很多学生读循环时只看循环头，却忽略循环体中对变量的修改；也有人把 for 都当成简单的 i++，遇到二分、双指针、步长变化就容易算错。

读循环程序要先看骨架，再看变量。

循环骨架

一个循环至少有三部分：

- 初始值；
- 继续条件；
- 每轮更新。

以 for 循环为例：

```
for (int i = 1; i <= n; i++) {
    ...
}
```

它的骨架是：

部分	内容	含义
初始值	`int i = 1`	第一次进入循环前，`i=1`
继续条件	`i <= n`	条件为真才进入循环体
每轮更新	`i++`	每次循环体执行完后，`i` 增加 1

当 $n = 5$ 时，`i` 的取值是 $1, 2, 3, 4, 5$，循环体执行 5 次。

但是，程序阅读题中的循环不一定这么整齐。

```
for (int i = 1; i <= n; i *= 2) {
    ...
}
```

若 $n = 20$，`i` 的取值是 $1, 2, 4, 8, 16$，循环体执行 5 次。

再看：

```
for (int i = n; i > 0; i -= 3) {
    ...
}
```

若 $n = 10$，`i` 的取值是 $10, 7, 4, 1$，循环体执行 4 次。

循环骨架要逐项读，不要只凭代码外形判断次数。

进入与退出

`while` 循环尤其要注意“先判断，再执行”。

```
int x = 5;

while (x < 5) {
    x++;
}

cout << x << endl;
```

一开始 `x < 5` 为假，循环体一次也不执行，输出 5。

再看：

```
int x = 5;

do {
    x++;
} while (x < 5);

cout << x << endl;
```

`do-while` 是先执行一次，再判断条件。这里 `x` 先变成 6，然后判断 `x < 5` 为假，输出 6。

CSP-S 第一轮中更常见的是 `while`，但无论是哪种循环，都要分清第一次进入循环前条件是否已经满足。

退出循环时也要看清变量停在哪个值上。

```
int i = 1;

while (i <= 5) {
    i++;
}

cout << i << endl;
```

循环体执行时 `i` 依次为 $1, 2, 3, 4, 5$。执行完 `i=5` 这一轮后，`i++` 使 `i=6`，下一次判断失败，所以输出 6，不是 5。

循环输出题常把“最后一次进入循环的值”和“退出循环后的值”混在选项里。读完循环后，应多写一行“退出时”。

循环体中的修改

循环变量不一定只在循环头更新，也可能在循环体里被修改。

```
int cnt = 0;

for (int i = 1; i <= 10; i++) {
    if (i % 3 == 0) {
        i++;
    }
    cnt++;
}

cout << cnt << endl;
```

逐轮追踪：

进入循环时 i	是否执行 i++	循环体结束后 i	循环头更新后 i	cnt
1	否	1	2	1
2	否	2	3	2
3	是	4	5	3
5	否	5	6	4
6	是	7	8	5
8	否	8	9	6
9	是	10	11	7

退出时 cnt=7。

这种程序不能只看 for (int i=1;i<=10;i++) 就说执行 10 次。循环体里的 i++ 改变了下一轮进入时的值。

另一类常见错误是忽略数组下标的变化：

```
for (int i = 1; i <= n; i++) {
    a[i + 1] += a[i];
}
```

这里每轮都会使用上一轮可能刚修改过的 a[i]。如果输入为 a[1]=1,a[2]=2,a[3]=3,a[4]=4，且 $n = 3$，过程是：

轮次	修改
$i = 1$	a[2] = 2 + 1 = 3
$i = 2$	a[3] = 3 + 3 = 6
$i = 3$	a[4] = 4 + 6 = 10

最终 a[4]=10。这里的 a[3] 用的是已经更新后的值，不是初始的 3。

break 与 continue

break 会直接结束当前循环。

```
int s = 0;

for (int i = 1; i <= 10; i++) {
    if (i == 6) {
        break;
    }
    s += i;
}

cout << s << endl;
```

当 i=6 时执行 break，s += i 不再执行。实际加的是 $1 + 2 + 3 + 4 + 5$，输出 15。

continue 会跳过本轮后面的语句，进入下一轮更新和判断。

```
int s = 0;

for (int i = 1; i <= 5; i++) {
    if (i == 3) {
        continue;
    }
    s += i;
}
```

```
cout << s << endl;
```

当 `i=3` 时跳过 `s += i`，实际加的是 $1+2+4+5$，输出 12。

`continue` 在 `for` 循环中不是回到循环体开头，而是先执行循环头里的更新语句，再判断条件。

看下面的程序：

```
int cnt = 0;

for (int i = 1; i <= 5; i++) {
    if (i % 2 == 0) {
        continue;
    }
    cnt += i;
}

cout << cnt << endl;
```

`i=2` 和 `i=4` 时跳过加法，但仍会执行循环头中的 `i++`。所以只加 $1, 3, 5$，输出 9。

嵌套循环中，`break` 和 `continue` 只影响它所在的那一层循环。

```
int cnt = 0;

for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 3; j++) {
        if (j == 2) {
            break;
        }
        cnt++;
    }
}

cout << cnt << endl;
```

内层循环每次只在 `j=1` 时执行 `cnt++`，遇到 `j=2` 就跳出内层。外层仍会继续，所以输出 3。

如果把 `break` 改成 `continue`：

```
int cnt = 0;

for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 3; j++) {
        if (j == 2) {
            continue;
        }
        cnt++;
    }
}

cout << cnt << endl;
```

内层每轮跳过 `j=2`，但 `j=1` 和 `j=3` 都会计数。外层 3 轮，共计 6 次。

嵌套循环

嵌套循环不一定是“外层次数乘以内层次数”。只有内层次数与外层变量无关时，才可以直接相乘。

```cpp
int cnt = 0;

for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 4; j++) {
        cnt++;
    }
}

cout << cnt << endl;
```

外层 3 次，内层每次 4 次，共输出 12。

但下面这段不能直接看成 5×5：

```cpp
int cnt = 0;

for (int i = 1; i <= 5; i++) {
    for (int j = 1; j <= i; j++) {
        cnt++;
    }
}

cout << cnt << endl;
```

每个 `i` 对应的内层次数是：

i	j 的取值	次数
1	1	1
2	1, 2	2
3	1, 2, 3	3
4	1, 2, 3, 4	4
5	1, 2, 3, 4, 5	5

总次数是 $1+2+3+4+5=15$。

再看：

```cpp
int cnt = 0;

for (int i = 1; i <= 5; i++) {
    for (int j = i + 1; j <= 5; j++) {
        cnt++;
    }
}

cout << cnt << endl;
```

这里统计的是满足 $1 \le i < j \le 5$ 的有序下标对，次数为 $4+3+2+1+0=10$。

程序阅读题中，嵌套循环常常与数组、字符串、计数搭配。要先写清楚内层循环对每个外层变量执行多少次，再看循环体做什么。

循环条件中的越界感

程序阅读题通常保证输入不超过数组或字符串定义范围，但读循环时仍要看清访问位置。

```
for (int i = 0; i + 1 < n; i++) {
    if (a[i] < a[i + 1]) {
        cnt++;
    }
}
```

这里 `i` 的最大值是 $n-2$，最后一次访问 `a[n-2]` 和 `a[n-1]`。如果把条件看成 `i < n`，就会误以为访问到 `a[n]`。

字符串程序也常这样写：

```
for (int i = 0; i + 3 <= s.size(); i += 3) {
    ...
}
```

这里每次处理从 `i` 开始的连续 3 个字符，最后一次进入循环时必须满足 `i+2 < s.size()`。条件写成 `i + 3 <= s.size()`，正好保证这一点。

读这类循环时，建议把条件改写成“最后一次允许的下标”。

例如 `i+3<=len` 等价于 `i<=len-3`。若 `len=10`，且 `i` 每次加 3，则 `i` 取 $0, 3, 6$。

双指针循环

双指针循环是 CSP-S 程序阅读中很容易设问的一类。它的外形常是一个 `for` 配一个 `while`。

```
int cnt = 0;

for (int i = 0, j = 0; i < n; i++) {
    while (a[i] - a[j] > m) {
        j++;
    }
    cnt += i - j;
}
```

这段程序要求 `a` 已经按非降序排列。变量含义是：

变量	含义
`i`	当前右端点
`j`	最小的可行左端点
`i-j`	以 `i` 为右端点，满足距离不超过 m 的左端点数量
`cnt`	已统计的数对数量

关键点是 `j` 只会增加，不会减少。

举例：`a={1,3,4,8,10}`，$m=4$。下标从 0 开始。

`i`	`a[i]`	调整前 `j`	调整过程	调整后 `j`	加上 `i-j`
0	1	0	无	0	0
1	3	0	无	0	1
2	4	0	无	0	2
3	8	0	$8-1>4$，`j=1`； $8-3>4$，`j=2`	2	1
4	10	2	$10-4>4$，`j=3`	3	1

最终 `cnt=5`。

这类程序如果用双重循环硬算，会很慢；但程序阅读题给的小输入通常可以列表。列表时只需要记录 `i`、`j` 和累计值，不需要写出每个数对。

双指针循环常考两个问题：

第一，时间复杂度不是 $O(n^2)$。虽然有 `for` 和 `while`，但 `j` 总共最多增加 n 次。

第二，条件中的 `>` 和 `>=` 差一位。

如果把 `a[i] - a[j] > m` 改成 `a[i] - a[j] >= m`，那么距离恰好为 m 的左端点也会被移走，统计结果会变小或不变。

二分循环

程序阅读中的二分循环，重点不是背模板，而是看区间如何缩小。

常见写法之一：

```
while (l < r) {
    int mid = (l + r) / 2;

    if (check(mid)) {
        r = mid;
    } else {
        l = mid + 1;
    }
}
```

这段程序寻找第一个使 `check(x)` 为真的位置。

设 `check(x)` 在 $x \geq 7$ 时为真，初始 `l=1,r=10`：

轮次	l	r	mid	check(mid)	更新
1	1	10	5	假	`l=6`
2	6	10	8	真	`r=8`
3	6	8	7	真	`r=7`
4	6	7	6	假	`l=7`

最后 `l=r=7`。

另一种写法寻找最后一个满足条件的位置：

```
while (l < r) {
    int mid = (l + r + 1) / 2;

    if (check(mid)) {
        l = mid;
    } else {
        r = mid - 1;
    }
}
```

这里 `mid` 取上中位数，防止当 `l+1=r` 且 `check(l)` 为真时，`l` 无法推进。

二分题不要只看 `while (l < r)`。要看：

- `mid` 是下中位数还是上中位数；
- 条件为真时更新左端还是右端；

- 更新是否保留 `mid`；
- 退出时输出 `l`、`r`、`mid` 还是别的变量。

真题式例题：距离统计

阅读下面程序：

```
bool check(int a[], int n, int m, int k) {
    int s = 0;

    for (int i = 0, j = 0; i < n; i++) {
        while (a[i] - a[j] > m) {
            j++;
        }
        s += i - j;
    }

    return s >= k;
}

int solve(int a[], int n, int k) {
    sort(a, a + n);

    int l = 0, r = a[n - 1] - a[0];

    while (l < r) {
        int mid = (l + r) / 2;

        if (check(a, n, mid, k)) {
            r = mid;
        } else {
            l = mid + 1;
        }
    }

    return l;
}
```

这段程序由两层意思组成。

`check(a,n,m,k)` 统计有多少对下标 `j<i` 满足 `a[i]-a[j]<=m`。若这样的数对至少有 k 对，就返回真。

`solve` 对答案二分，寻找最小的 m，使满足距离不超过 m 的数对数量至少为 k。

例如输入数组为：

```
a = {2, -4, 5, 1, -3}
k = 7
```

排序后：

```
-4, -3, 1, 2, 5
```

先看不同距离的数对数量。所有 10 对距离如下：

数对	距离
$(-4,-3)$	1
$(-4,1)$	5
$(-4,2)$	6
$(-4,5)$	9
$(-3,1)$	4
$(-3,2)$	5
$(-3,5)$	8
$(1,2)$	1
$(1,5)$	4
$(2,5)$	3

按距离从小到大排，第 7 小距离是 5。所以 `solve` 返回 5。

再用程序过程验证 `check(m=4)` 和 `check(m=5)`。

当 $m=4$ 时：

i	a[i]	调整后 j	加上 i-j
0	−4	0	0
1	−3	0	1
2	1	1	1
3	2	1	2
4	5	2	2

此时 `s=6`，不够 7，`check` 返回假。

当 $m=5$ 时：

i	a[i]	调整后 j	加上 i-j
0	−4	0	0
1	−3	0	1
2	1	0	2
3	2	1	2
4	5	2	2

此时 `s=7`，`check` 返回真。二分会把答案压到 5。

这类题的阅读顺序是：

1. 先读 `check`，弄清真假含义；
2. 再读 `solve`，判断二分找的是最小可行值还是最大可行值；
3. 最后代入具体输入追踪边界。

如果直接从二分开始算，而不知道 `check` 的含义，很容易把真和假对应的更新方向看反。

真题式例题：特殊步长

阅读下面程序：

```
int f(string s, string t) {
    int n = s.length();
    int m = t.length();
```

```
    int shift[128];

    for (int i = 0; i < 128; i++) {
        shift[i] = m + 1;
    }

    for (int j = 0; j < m; j++) {
        shift[t[j]] = m - j;
    }

    for (int i = 0; i <= n - m; i += shift[s[i + m]]) {
        int j = 0;

        while (j < m && s[i + j] == t[j]) {
            j++;
        }

        if (j == m) {
            return i;
        }
    }

    return -1;
}
```

这段程序看起来像字符串匹配。程序阅读时不需要完整掌握某个算法名称，先看循环变量。

第一个循环把所有 `shift` 初始化为 `m+1`。

第二个循环根据模式串 `t` 更新部分字符的 `shift`。若字符 `t[j]` 出现，`shift[t[j]]=m-j`。若同一个字符多次出现，后面的赋值会覆盖前面的赋值。

第三个循环最关键：

```
for (int i = 0; i <= n - m; i += shift[s[i + m]])
```

它不是 `i++`，而是根据 `s[i+m]` 的字符决定跳多少。

例如 `s="abbababbbab"`，`t="abab"`。

这里 $n = 11, m = 4$。先算 `shift`：

字符	最后一次赋值来源	shift
'a'	j=2	2
'b'	j=3	1
其他字符	初始化	5

追踪外层循环：

i	比较的 s[i..i+3]	while 中成功比较次数	是否返回	下一步
0	abba	2	否	看 s[4]='b'，i+=1
1	bbab	0	否	看 s[5]='a'，i+=2
3	abab	4	是	返回 3

所以函数返回 3。

这类循环的重点有两个：

第一，`i` 的变化由循环头最后一部分决定，不是每次加 1。

第二，`s[i+m]` 访问的是当前匹配窗口后面的一个字符，而不是窗口内部的最后一个字符。当前窗口是 `s[i]` 到 `s[i+m-1]`，`s[i+m]` 在窗口右边。

如果题目问某行 `j++` 执行多少次，要逐次统计 `while` 成功比较的次数，而不是统计外层循环次数。

变式题：continue 跳过更新吗

阅读下面程序：

```
int s = 0;

for (int i = 1; i <= 8; i++) {
    if (i % 3 == 0) {
        continue;
    }

    s += i;
}

cout << s << endl;
```

`continue` 跳过的是本轮循环体中它后面的语句，即 `s += i`。循环头里的 `i++` 仍会执行。

所以被加上的数是 $1, 2, 4, 5, 7, 8$，输出 27。

若把程序改为：

```
int s = 0, i = 1;

while (i <= 8) {
    if (i % 3 == 0) {
        i++;
        continue;
    }

    s += i;
    i++;
}

cout << s << endl;
```

结果仍是 27。这里在 `continue` 前手动写了 `i++`，避免 `i=3` 时反复进入同一轮。

如果写成：

```
int s = 0, i = 1;

while (i <= 8) {
    if (i % 3 == 0) {
        continue;
    }

    s += i;
    i++;
}
```

当 `i=3` 时会一直执行 `continue`，`i` 不再改变，循环无法结束。读 `while` 时要特别检查每条分支是否会推进循环变量。

变式题：内层变量的初值

阅读下面程序：

```
int cnt = 0;

for (int i = 1; i <= 3; i++) {
    int j = i;

    while (j <= 5) {
        cnt++;
        j += 2;
    }
}

cout << cnt << endl;
```

逐轮计算：

i	j 的取值	次数
1	$1, 3, 5$	3
2	$2, 4$	2
3	$3, 5$	2

输出 7。

这里 j 在每次外层循环开始时重新定义并赋值为 i。如果误以为 j 会保留上一轮的值，就会算错。

再看改写：

```
int cnt = 0;
int j = 1;

for (int i = 1; i <= 3; i++) {
    while (j <= 5) {
        cnt++;
        j += 2;
    }
}

cout << cnt << endl;
```

这次 j 在外层循环外定义。第一轮 i=1 时，j 取 $1, 3, 5$，随后变成 7。后两轮内层 while 都不再执行。输出 3。

同样的变量名，定义位置不同，生命周期和初值就不同。程序阅读题常用这一点设置干扰。

变式题：循环后的变量

阅读下面程序：

```
int i;

for (i = 1; i * i <= 30; i++) {
}

cout << i << endl;
```

循环体为空，但循环头仍会更新 i。

逐项看：

- `i=1`，$1 \cdot 1 \le 30$，进入，随后 `i++`；
- `i=2`，$2 \cdot 2 \le 30$，进入；
- `i=3`，$3 \cdot 3 \le 30$，进入；
- `i=4`，$4 \cdot 4 \le 30$，进入；
- `i=5`，$5 \cdot 5 \le 30$，进入；
- `i=6`，$6 \cdot 6 \le 30$ 为假，退出。

输出 6。

循环结束后，`i` 是第一个不满足条件的值，而不是最后一个满足条件的值。最后一个满足条件的是 5。

这类空循环常出现在求某个边界的程序里。不要因为循环体为空就忽略它。

小结

读循环程序时，可以按下面的顺序检查：

检查点	要问的问题
初始值	第一次判断前变量是什么
继续条件	什么情况下还能进入循环
更新语句	每轮结束后变量怎么变
循环体修改	循环体里是否又改了循环变量
`break`	是否提前结束当前循环
`continue`	是否跳过了后面的关键语句
嵌套层数	`break` 或 `continue` 影响哪一层
内层次数	内层循环次数是否依赖外层变量
下标边界	最后一次访问的是哪个下标
退出状态	退出后变量停在哪个值
输出位置	输出发生在循环中还是循环后

循环程序的读法可以概括为一句话：不要只看循环头，要看每一条能改变变量和流程的语句。先确定变量怎样推进，再统计执行次数；先确定循环何时退出，再判断输出值。这样处理，循环程序阅读就不会只靠感觉。

0x31 递归程序

递归程序的难点不在“函数调用自己”这句话，而在调用过程中变量会分成很多层：每一层有自己的参数和局部变量，有些全局变量又会被所有层共同修改。读递归时，如果只盯着当前一行，很容易忘记函数返回到哪里；如果一上来就画完整递归树，又会把纸面画得很乱。

读递归程序要抓住四件事：停止条件、参数变化、返回值、外部状态。

先找停止条件

每个递归函数都要先找停止条件。停止条件通常写在函数开头：

```
int f(int n) {
    if (n <= 1) {
        return 1;
    }

    return f(n - 1) + f(n - 2);
}
```

这里停止条件是 `n <= 1`，返回值是 1。

如果题目问 `f(4)`，可以先展开到停止条件：

调用	返回
`f(0)`	1
`f(1)`	1
`f(2)`	`f(1)+f(0)=2`
`f(3)`	`f(2)+f(1)=3`
`f(4)`	`f(3)+f(2)=5`

所以 `f(4)=5`。

有些递归有多个停止条件：

```
int g(int n) {
    if (n == 0) {
        return 0;
    }

    if (n % 2 == 0) {
        return g(n / 2) + 1;
    } else {
        return g(n - 1) + 1;
    }
}
```

读这种程序时，要按当前参数判断进入哪一支。

例如 `g(7)`：

当前调用	进入分支	下一步
g(7)	奇数	g(6)+1
g(6)	偶数	g(3)+1
g(3)	奇数	g(2)+1
g(2)	偶数	g(1)+1
g(1)	奇数	g(0)+1
g(0)	停止	0

向回代入得到 g(1)=1，g(2)=2，g(3)=3，g(6)=4，g(7)=5。

停止条件不仅决定最终结果，也决定递归会不会继续向下。读递归程序时，第一眼应该看 if 中哪些分支会直接 return。

参数怎样变

递归函数的参数变化决定调用链。

看下面的程序：

```
void print(int n) {
    if (n == 0) {
        return;
    }

    cout << n << ' ';
    print(n - 1);
}
```

调用 print(4) 时：

调用	先输出	下一次调用
print(4)	4	print(3)
print(3)	3	print(2)
print(2)	2	print(1)
print(1)	1	print(0)
print(0)	不输出	返回

输出为：

```
4 3 2 1
```

如果把输出放到递归调用后面：

```
void print(int n) {
    if (n == 0) {
        return;
    }

    print(n - 1);
    cout << n << ' ';
}
```

调用顺序仍然是 4,3,2,1,0，但输出发生在返回过程中：

返回的调用	输出
`print(1)`	1
`print(2)`	2
`print(3)`	3
`print(4)`	4

输出为：

```
1 2 3 4
```

递归程序中，语句在递归调用前还是递归调用后，差别很大。读代码时要标出“先做什么、递归什么、回来后做什么”。

返回值怎么代入

有返回值的递归，要把每个调用看成一个表达式。

```
int f(int n) {
    if (n == 1) {
        return 2;
    }

    return 3 * f(n - 1) + 1;
}
```

计算 `f(4)`：

调用	返回值
`f(1)`	2
`f(2)`	$3 \times 2 + 1 = 7$
`f(3)`	$3 \times 7 + 1 = 22$
`f(4)`	$3 \times 22 + 1 = 67$

这种单分支递归可以直接从小到大填表，不必画树。

若一个函数里有两次递归调用，就要看是否会重复计算同一个值：

```
int f(int n) {
    if (n <= 2) {
        return 1;
    }

    return f(n - 1) + f(n - 2);
}
```

计算 `f(5)` 时，调用关系可以写成：

调用	展开
`f(5)`	`f(4)+f(3)`
`f(4)`	`f(3)+f(2)`
`f(3)`	`f(2)+f(1)`

若题目问返回值，可以按递推表算：

n	f(n)
1	1
2	1
3	2
4	3
5	5

若题目问调用次数，就不能只填返回值表，因为同一个 `f(3)` 可能被调用多次。

调用次数

递归题常问某个函数一共被调用多少次。此时要追“调用次数”，不是只追返回值。

看下面的程序：

```
int cnt = 0;

int f(int n) {
    cnt++;

    if (n <= 1) {
        return 1;
    }

    return f(n - 1) + f(n - 2);
}
```

调用 `f(4)` 时，调用树为：

```
f(4)
  f(3)
    f(2)
      f(1)
      f(0)
    f(1)
  f(2)
    f(1)
    f(0)
```

一共有 9 次调用，所以 `cnt=9`。

可以用表格从小到大算调用次数。设 `c(n)` 表示调用 `f(n)` 时总共进入 `f` 的次数：

n	c(n)
0	1
1	1
2	$1+c(1)+c(0)=3$
3	$1+c(2)+c(1)=5$
4	$1+c(3)+c(2)=9$

这里多出来的 1 是当前这次调用本身。

如果题目问“第几行执行多少次”，要看这行是在停止条件前，还是在停止条件后。

例如：

```
int f(int n) {
    cnt1++;

    if (n <= 1) {
        return 1;
    }

    cnt2++;
    return f(n - 1) + f(n - 2);
}
```

`cnt1` 每次进入函数都会加；`cnt2` 只在 `n>1` 时加。调用 `f(4)` 时，总调用次数是 9，其中非停止调用为 `f(4),f(3),f(2),f(2)`，所以 `cnt2=4`。

读调用次数题时，要先确定被统计的语句在哪个位置。

全局变量

递归程序中，全局变量会被所有递归层共同修改。

```
int ans = 0;

void dfs(int n) {
    if (n == 0) {
        ans++;
        return;
    }

    dfs(n - 1);
    dfs(n - 1);
}
```

调用 `dfs(3)`，每次到 `n==0` 时 `ans++`。每一层都分成两支，所以到达停止条件的次数为 $2^3 = 8$，最后 `ans=8`。

如果程序改成：

```
int ans = 0;

void dfs(int n) {
    ans++;

    if (n == 0) {
        return;
    }

    dfs(n - 1);
    dfs(n - 1);
}
```

这次 `ans` 统计的是所有调用次数，不只统计叶子。调用 `dfs(3)` 时，总调用次数是 $1 + 2 + 4 + 8 = 15$，所以 `ans=15`。

全局变量要看修改位置。修改在停止条件内、停止条件前、递归调用前、递归调用后，含义都不同。

回溯恢复

搜索类递归常使用 “选择—递归—恢复” 的结构。

```cpp
bool vis[10];
int p[10];
int n;

void dfs(int k) {
    if (k == n + 1) {
        for (int i = 1; i <= n; i++) {
            cout << p[i] << ' ';
        }
        cout << endl;
        return;
    }

    for (int i = 1; i <= n; i++) {
        if (vis[i]) {
            continue;
        }

        p[k] = i;
        vis[i] = true;
        dfs(k + 1);
        vis[i] = false;
    }
}
```

关键变量表：

变量	含义
k	当前要填写的位置
i	当前尝试的数字
p[k]	第 k 个位置填的数
vis[i]	数字 i 是否已被当前路径使用

`vis[i] = true` 是做选择，`vis[i] = false` 是恢复现场。恢复后，后面的分支才能重新使用数字 i。

以 $n = 3$ 为例，第一条路径是：

```
p[1]=1
p[2]=2
p[3]=3
```

输出 `1 2 3` 后返回到 `k=3` 的循环。此时需要执行 `vis[3]=false`，否则数字 3 在后续分支中仍被认为已经使用。

若删去恢复语句：

```cpp
vis[i] = false;
```

前面分支使用过的数字会影响后面分支，输出数量通常会变少。判断这类改动是否影响结果时，要看标记是否表示“当前路径状态”。只要它表示当前路径，一般递归返回后就需要恢复。

但不是所有递归修改都要恢复。例如：

```cpp
void dfs(int u) {
    vis[u] = true;

    for (int v = 1; v <= n; v++) {
        if (g[u][v] && !vis[v]) {
            dfs(v);
```

```
        }
    }
}
```

这里 `vis[u]` 表示“这个点是否已经访问过”，不是当前路径临时选择。它通常不需要在返回后改回 `false`。是否恢复，要看变量含义，而不是看到递归就自动恢复。

调用顺序

有些递归程序的输出由调用顺序决定。

```
void dfs(int n) {
    if (n == 0) {
        return;
    }

    cout << n << ' ';
    dfs(n - 1);
    cout << n << ' ';
}
```

调用 `dfs(3)`：

步骤	输出
进入 `dfs(3)`	3
进入 `dfs(2)`	2
进入 `dfs(1)`	1
`dfs(1)` 返回前	1
`dfs(2)` 返回前	2
`dfs(3)` 返回前	3

输出为：

```
3 2 1 1 2 3
```

再看多分支：

```
void dfs(int n) {
    if (n == 0) {
        cout << "A";
        return;
    }

    cout << "B";
    dfs(n - 1);
    cout << "C";
    dfs(n - 1);
}
```

调用 `dfs(2)`：

调用位置	输出
dfs(2) 开头	B
第一次 dfs(1) 开头	B
第一次 dfs(0)	A
回到第一次 dfs(1) 中间	C
第二次 dfs(0)	A
回到 dfs(2) 中间	C
第二次 dfs(1) 开头	B
第一次 dfs(0)	A
回到第二次 dfs(1) 中间	C
第二次 dfs(0)	A

输出为：

```
BBACACBACA
```

这类题不要只数有多少个 A、B、C，还要按语句顺序写。可以先把函数体分成：

```
输出 B
左递归
输出 C
右递归
```

再逐层展开。

区间递归

分治类递归常以区间作为参数。

```
int solve(int l, int r) {
    if (l == r) {
        return a[l];
    }

    int mid = (l + r) / 2;
    return solve(l, mid) + solve(mid + 1, r);
}
```

读区间递归时，重点是区间如何分裂。

当调用 solve(1,5)：

调用	分裂
solve(1,5)	solve(1,3) 和 solve(4,5)
solve(1,3)	solve(1,2) 和 solve(3,3)
solve(1,2)	solve(1,1) 和 solve(2,2)
solve(4,5)	solve(4,4) 和 solve(5,5)

所有长度为 1 的区间都会返回对应的 a[i]，所以整体返回的是区间内所有数的和。

若程序改成：

```
return max(solve(l, mid), solve(mid + 1, r));
```

整体返回的就是区间最大值。

区间递归题常问时间复杂度。要看每次分裂后，除了递归调用以外还做了多少工作：

```
int solve(int l, int r) {
    if (l == r) {
        return a[l];
    }

    int mid = (l + r) / 2;
    int s = 0;

    for (int i = l; i <= r; i++) {
        s += a[i];
    }

    return s + solve(l, mid) + solve(mid + 1, r);
}
```

这里每一层递归的区间扫描总长度约为 n，层数约为 $\log n$，所以总时间是 $O(n \log n)$。

如果没有中间的 `for`，每个结点只做常数次操作，总调用结点数与区间个数同阶，时间是 $O(n)$。

真题式例题：递归求最大子段和

阅读下面程序：

```
struct Node {
    int h, j, m, w;

    Node(int _h, int _j, int _m, int _w) {
        h = _h;
        j = _j;
        m = _m;
        w = _w;
    }

    Node operator + (const Node& o) const {
        return Node(
            max(h, w + o.h),
            max(max(j, o.j), m + o.h),
            max(m + o.w, o.m),
            w + o.w
        );
    }
};

int a[1005];

Node solve(int l, int r) {
    if (l == r) {
        return Node(max(a[l], 0), max(a[l], 0), max(a[l], 0), a[l]);
    }

    int mid = (l + r) / 2;
    return solve(l, mid) + solve(mid + 1, r);
}
```

这段程序使用递归处理区间。先看 `Node` 的四个成员含义：

成员	含义
h	区间从左端开始的最大非负前缀和
j	区间内部的最大非负连续子段和
m	区间到右端结束的最大非负后缀和
w	整个区间的总和

叶子区间只有一个数 a[l]：

```
Node(max(a[l], 0), max(a[l], 0), max(a[l], 0), a[l])
```

如果 a[l] 是正数，则前缀、最大子段、后缀都是它本身；如果 a[l] 是负数，则允许取空段，前三项为 0，总和仍是 a[l]。

合并两个相邻区间时：

- 新前缀最大值，要么只在左区间，要么是左区间整体加右区间前缀，即 max(h, w + o.h)；
- 新最大子段，要么在左区间，要么在右区间，要么跨过中点，即 max(max(j,o.j),m+o.h)；
- 新后缀最大值，要么只在右区间，要么是左区间后缀加右区间整体，即 max(m+o.w,o.m)；
- 新总和是 w+o.w。

例如数组为：

```
-3 2 10 0 -8 9 -4 -5 9 4
```

若只问最终第一行输出，可以理解为求最大非负连续子段和。直接手算连续子段：

位置	数值	以当前位置结尾的最大非负和	当前最大值
1	−3	0	0
2	2	2	2
3	10	12	12
4	0	12	12
5	−8	4	12
6	9	13	13
7	−4	9	13
8	−5	4	13
9	9	13	13
10	4	17	17

所以 solve(1,n).j 为 17。

读这类递归程序时，不一定要把每个 Node 都算出来。若能识别成员含义，可以用更短的方式验证输出；若题目问某个成员的具体变化，再局部计算对应区间。

真题式例题：递归与取模

阅读下面程序：

```
int solve(int t, int n) {
    if (t == 1) {
        return 1;
    } else {
        return 5 * solve(t - 1, n) % n;
```

```
    }
}
```

求 `solve(23,23)`。

这个递归每次把 `t` 减 1，直到 `t==1`。返回时每层乘 5 再对 n 取模。也就是说：

调用	值
`solve(1,23)`	1
`solve(2,23)`	5
`solve(3,23)`	$25 \bmod 23 = 2$
`solve(4,23)`	10
`solve(5,23)`	$50 \bmod 23 = 4$
`solve(6,23)`	20
`solve(7,23)`	$100 \bmod 23 = 8$
`solve(8,23)`	$40 \bmod 23 = 17$
`solve(9,23)`	$85 \bmod 23 = 16$
`solve(10,23)`	$80 \bmod 23 = 11$
`solve(11,23)`	$55 \bmod 23 = 9$
`solve(12,23)`	$45 \bmod 23 = 22$
`solve(13,23)`	$110 \bmod 23 = 18$
`solve(14,23)`	$90 \bmod 23 = 21$
`solve(15,23)`	$105 \bmod 23 = 13$
`solve(16,23)`	$65 \bmod 23 = 19$
`solve(17,23)`	$95 \bmod 23 = 3$
`solve(18,23)`	15
`solve(19,23)`	$75 \bmod 23 = 6$
`solve(20,23)`	$30 \bmod 23 = 7$
`solve(21,23)`	$35 \bmod 23 = 12$
`solve(22,23)`	$60 \bmod 23 = 14$
`solve(23,23)`	$70 \bmod 23 = 1$

所以结果为 1。

这道题如果完整写 5^{22} 再取模，会很麻烦。按递归返回过程逐项取模，更适合纸面计算。

递归程序里出现 `%` 时，要随算随取模，避免数字膨胀，也减少手算错误。

互相递归

有些程序不是函数直接调用自己，而是两个函数互相调用。

```
int f(int n);
int g(int n);

int f(int n) {
    if (n == 0) {
        return 1;
    }

    return g(n - 1) + 1;
```

```
}

int g(int n) {
    if (n == 0) {
        return 2;
    }

    return f(n - 1) + 2;
}
```

计算 f(4)：

调用	展开
f(4)	g(3)+1
g(3)	f(2)+2
f(2)	g(1)+1
g(1)	f(0)+2
f(0)	1

向回代入：

- g(1)=1+2=3；
- f(2)=3+1=4；
- g(3)=4+2=6；
- f(4)=6+1=7。

互相递归只要把函数名和参数写清楚，处理方式和普通递归一样。不要因为函数名变了就断开调用链。

递归中的局部变量

局部变量属于某一次函数调用。不同调用中的同名局部变量互不相同。

```
void f(int n) {
    int x = n;

    if (n == 0) {
        return;
    }

    f(n - 1);
    cout << x << ' ';
}
```

调用 f(3) 时，每一层都有自己的 x：

调用	本层 x
f(3)	3
f(2)	2
f(1)	1
f(0)	0

输出发生在递归调用返回后，所以依次输出 $1, 2, 3$。

这里不是一个 x 被不断覆盖，而是每次调用都有自己的 x。读递归程序时，局部变量要跟着调用层走。

与之相反，全局变量只有一份：

```
int x = 0;

void f(int n) {
    if (n == 0) {
        return;
    }

    x += n;
    f(n - 1);
    cout << x << ' ';
}
```

调用 `f(3)` 时，`x` 先后变成 $3, 5, 6$。返回输出时，所有层看到的是同一个全局 `x=6`，所以输出：

```
6 6 6
```

局部变量和全局变量的区别，是递归阅读中的常见考点。

变式题：输出顺序

阅读下面程序，写出输出。

```
void f(int n) {
    if (n == 0) {
        cout << "X";
        return;
    }

    cout << n;
    f(n - 1);
    cout << n;
}

int main() {
    f(3);
    return 0;
}
```

按顺序展开：

步骤	输出
进入 `f(3)`	`3`
进入 `f(2)`	`2`
进入 `f(1)`	`1`
进入 `f(0)`	`X`
回到 `f(1)`	`1`
回到 `f(2)`	`2`
回到 `f(3)`	`3`

输出为：

```
321X123
```

若把递归调用前后的两个输出交换位置，答案也会跟着改变。递归输出题一定要按执行顺序写，不能只看出现了哪些字符。

变式题：回溯计数

阅读下面程序：

```
int n = 3;
int a[5];
int ans = 0;

void dfs(int k) {
    if (k == n + 1) {
        if (a[1] + a[2] + a[3] == 4) {
            ans++;
        }
        return;
    }

    for (int i = 0; i <= 2; i++) {
        a[k] = i;
        dfs(k + 1);
    }
}
```

这里 `dfs` 枚举 $a[1], a[2], a[3]$，每个位置可以取 $0, 1, 2$。最后统计和为 4 的方案数。

列出满足条件的三元组：

三元组	和
$(0, 2, 2)$	4
$(1, 1, 2)$	4
$(1, 2, 1)$	4
$(2, 0, 2)$	4
$(2, 1, 1)$	4
$(2, 2, 0)$	4

共有 6 种，所以 `ans=6`。

这段程序没有 `vis`，也不需要恢复 `a[k]`。因为下一次循环会直接覆盖 `a[k]`，并且停止条件只读取当前路径上已经赋值的位置。

不是所有搜索递归都需要写恢复语句。是否恢复，取决于修改是否会影响后续分支。

变式题：返回值与全局变量同时出现

阅读下面程序：

```
int cnt = 0;

int f(int n) {
    cnt++;

    if (n == 1) {
        return 1;
    }

    int x = f(n - 1);
    return x + n;
}
```

```cpp
int main() {
    cout << f(4) << ' ' << cnt << endl;
    return 0;
}
```

先算返回值：

调用	返回值
f(1)	1
f(2)	$1+2=3$
f(3)	$3+3=6$
f(4)	$6+4=10$

再算 cnt。每进入一次 f，cnt++，调用了 f(4),f(3),f(2),f(1)，共 4 次，所以 cnt=4。

输出为：

```
10 4
```

有些题会同时问返回值和全局变量。不要把二者混在一起。返回值按表达式代入，全局变量按语句执行次数统计。

小结

读递归程序时，可以按下面顺序检查：

检查点	要问的问题
停止条件	哪些情况直接返回
参数变化	每次递归参数怎样变
递归次数	一个调用中会递归几次
调用顺序	递归前、递归后分别做什么
返回值	当前调用返回的表达式是什么
全局变量	哪些变量会被所有调用共同修改
局部变量	同名局部变量是否属于不同调用
回溯恢复	修改的状态是否只属于当前路径
输出语句	输出发生在进入时还是返回时
复杂度	每层调用数量和额外工作量是多少

递归程序阅读不要急着画完整大树。单分支递归优先列调用链；分治递归优先列区间；搜索递归优先列选择和恢复；有返回值的递归优先做返回表；问调用次数时再画小规模调用树。只要先抓住停止条件和参数变化，递归程序就能拆成一层一层的普通语句。

0x32 数组与字符串程序

数组和字符串程序的难点往往不在“会不会用数组”，而在下标、边界和修改顺序。程序阅读中，很多错误不是算法没看懂，而是把 `a[i]` 看成 `a[i+1]`，把 `s[i+m]` 看成窗口最后一个字符，或者忽略了某个数组已经被前一轮修改过。

读数组与字符串程序时，要先确定三件事：下标从哪里开始，循环访问到哪里，数组或字符串是否在过程中被修改。

下标先行

数组程序先看下标范围。

```
for (int i = 1; i <= n; i++) {
    cin >> a[i];
}
```

这段代码使用 1 到 n 号位置。

```
for (int i = 0; i < n; i++) {
    cin >> a[i];
}
```

这段代码使用 0 到 $n-1$ 号位置。

两种写法都很常见。读题时不能默认一种下标方式，要根据读入循环确定。

看下面的程序：

```
int ans = 0;

for (int i = 1; i < n; i++) {
    if (a[i] < a[i + 1]) {
        ans++;
    }
}
```

这里下标从 1 开始，最后一次循环是 `i=n-1`，访问 `a[n]`。它统计相邻两项中后一项更大的次数。

若输入为：

```
n = 5
a[1..5] = 3 1 4 4 7
```

追踪如下：

i	比较	是否计数
1	$3 < 1$	否
2	$1 < 4$	是
3	$4 < 4$	否
4	$4 < 7$	是

所以 ans=2。

若把循环条件误读成 i<=n，就会以为访问 a[n+1]；若把数组误认为从 0 开始，就会漏掉第一对或多算一对。

字符串也是一样：

```
for (int i = 0; i < s.size(); i++) {
    cout << s[i];
}
```

合法下标是 0 到 s.size()-1。字符串长度为 m 时，最后一个字符是 s[m-1]，不是 s[m]。

窗口与边界

字符串程序常常处理一段连续窗口。

```
for (int i = 0; i + m <= n; i++) {
    bool ok = true;

    for (int j = 0; j < m; j++) {
        if (s[i + j] != t[j]) {
            ok = false;
        }
    }

    if (ok) {
        ans++;
    }
}
```

这里外层 i 表示窗口左端，内层 j 表示模式串中的位置。窗口覆盖的是：

```
s[i], s[i + 1], ..., s[i + m - 1]
```

外层条件 i + m <= n 保证 i+m-1 不超过 n-1。

例如 s="ababa"，t="aba"，则 $n = 5, m = 3$。外层 i 取 $0, 1, 2$。

i	窗口	是否等于 t
0	aba	是
1	bab	否
2	aba	是

所以 ans=2。

读窗口程序时，要把窗口左端、右端写清楚。i+m<=n 对应右端 i+m-1，这一步很容易漏。

再看一个容易混淆的写法：

```
for (int i = 0; i + 3 <= s.size(); i += 3) {
    cout << s[i] << s[i + 1] << s[i + 2] << endl;
}
```

每次输出从 i 开始的 3 个字符。若 s.size() 为 10，i 取 $0, 3, 6$，输出的最后一组使用 s[6],s[7],s[8]，s[9] 不会被输出。

条件中的 i+3<=s.size() 不是说访问 s[i+3]，而是说从 i 开始还能取够 3 个字符。

修改顺序

数组在循环中被修改时，要分清本轮使用的是原值还是新值。

```
for (int i = 1; i < n; i++) {
    a[i + 1] += a[i];
}
```

这段代码会把前面的修改传给后面。

若初始数组为：

```
a[1..5] = 1 2 3 4 5
```

过程如下：

i	修改	修改后关键值
1	a[2] += a[1]	a[2]=3
2	a[3] += a[2]	a[3]=6
3	a[4] += a[3]	a[4]=10
4	a[5] += a[4]	a[5]=15

最终数组为：

```
1 3 6 10 15
```

如果程序改成从右往左：

```
for (int i = n - 1; i >= 1; i--) {
    a[i + 1] += a[i];
}
```

初始仍为 `1 2 3 4 5`，过程是：

i	修改	修改后关键值
4	a[5] += a[4]	a[5]=9
3	a[4] += a[3]	a[4]=7
2	a[3] += a[2]	a[3]=5
1	a[2] += a[1]	a[2]=3

最终数组为：

```
1 3 5 7 9
```

同一句赋值，循环方向不同，结果不同。程序阅读题中，数组原地更新时一定要看方向。

类似地，字符串也可能被原地修改：

```
for (int i = 0; i + 1 < s.size(); i++) {
    if (s[i] == 'a') {
        s[i + 1] = 'b';
    }
}
```

若 `s="aaaa"`，过程为：

i	修改前字符串	判断	修改后
0	aaaa	s[0]=='a'	abaa
1	abaa	s[1]=='b'	abaa
2	abaa	s[2]=='a'	abab

最终是 abab。第二轮看到的是已经被改过的 s[1]，不是初始字符 'a'。

计数数组

数组不一定存原序列，有时用来记录次数、位置或状态。

```
int cnt[10] = {0};

for (int i = 0; i < s.size(); i++) {
    cnt[s[i] - '0']++;
}
```

若 s="102201"，则：

字符	修改
'1'	cnt[1]++
'0'	cnt[0]++
'2'	cnt[2]++
'2'	cnt[2]++
'0'	cnt[0]++
'1'	cnt[1]++

最终 cnt[0]=2,cnt[1]=2,cnt[2]=2，其他为 0。

计数数组常见的下标转换有：

写法	含义
cnt[x]++	值 x 出现次数增加
cnt[s[i]-'0']++	数字字符出现次数增加
cnt[s[i]-'a']++	小写字母出现次数增加
pos[x]=i	值 x 最后一次出现位置为 i
vis[x]=true	值 x 已经出现或被使用

读这类程序时，要先判断数组下标代表什么。cnt[i] 不一定对应原数组第 i 项，它可能对应值 i。

看下面的程序：

```
int ans = 0;

for (int i = 1; i <= n; i++) {
    cnt[a[i]]++;
}

for (int i = 1; i <= n; i++) {
    if (cnt[i] == 1) {
        ans++;
    }
}
```

第一段统计每个值出现次数，第二段统计值 1 到 n 中恰好出现一次的数的个数。它不是统计原数组中有多少位置只出现一次，而是统计“哪些值出现一次”。

若 n=5，a[1..5]=2,1,2,5,5，则 cnt[1]=1,cnt[2]=2,cnt[5]=2，所以 ans=1。

排序后的数组

很多程序会先排序，再做后续处理。读这种程序时，原来的相对位置可能已经失效。

```
sort(a + 1, a + n + 1);

int ans = 0;

for (int i = 2; i <= n; i++) {
    if (a[i] == a[i - 1]) {
        ans++;
    }
}
```

它统计排序后相邻相等的次数，也就是重复出现造成的相邻重复数。

若原数组为：

```
4 1 4 2 2 2
```

排序后：

```
1 2 2 2 4 4
```

追踪：

i	比较	是否计数
2	$2=1$	否
3	$2=2$	是
4	$2=2$	是
5	$4=2$	否
6	$4=4$	是

所以 ans=3。

若题目问不同数的个数，常见写法是：

```
sort(a + 1, a + n + 1);

int ans = 0;

for (int i = 1; i <= n; i++) {
    if (i == 1 || a[i] != a[i - 1]) {
        ans++;
    }
}
```

这段程序统计排序后每段相等值的第一项。上面的数组不同数为 $1,2,4$，所以 ans=3。

读排序程序时要注意：排序后 a[i] 不再是原来的第 i 个输入。若还需要原位置，程序通常会使用结构体或下标数组。

unique 的读法

unique 常和排序一起出现。

```
sort(a, a + n);
int m = unique(a, a + n) - a;
```

这两句执行后，a[0..m-1] 是去重后的序列，m 是不同元素个数。

例如：

```
a = {4, 1, 4, 2, 2, 2}
```

排序后为：

```
1 2 2 2 4 4
```

执行 unique 后，前面有效部分为：

```
1 2 4
```

所以 m=3。

要注意，unique 并不会真正删除数组后面的旧数据，只是把不重复的部分移动到前面，并返回有效区间的结束位置。程序若继续访问 a[m] 之后的内容，不能把它们当作已经清空。

字符串也可以用类似方式：

```
sort(s.begin(), s.end());
s.erase(unique(s.begin(), s.end()), s.end());
```

这会把字符串中的不同字符保留下来，每种字符只留一个。

例如 s="banana"，排序后为 aaabnn，去重后为 abn。

程序阅读中若出现 unique，先问两个问题：

- 前面是否已经排序；
- 后续使用的是有效长度，还是仍使用原长度。

若没有排序，unique 只会去掉相邻重复，不能去掉所有重复。

例如 1 2 1 2 使用 unique 后仍是 1 2 1 2 的有效部分，因为没有相邻相等。

字符与整数

字符串程序常把字符转成编号。

```
int x = s[i] - '0';
```

若 s[i]='7'，则 x=7。

```
int x = s[i] - 'a';
```

若 s[i]='c'，则 x=2。

```
char ch = 'A' + x;
```

若 x=3，则 ch='D'。

读字符程序时，不要把字符 '3' 和整数 3 混在一起。'3' 是字符，3 是整数；表达式 '3'-'0' 才得到整数 3。

看下面的程序：

```
string s = "314";
int x = 0;

for (int i = 0; i < s.size(); i++) {
    x = x * 10 + (s[i] - '0');
}

cout << x << endl;
```

追踪：

i	字符	转成数字	x
0	'3'	3	3
1	'1'	1	31
2	'4'	4	314

输出 314。

如果写成：

```
x = x * 10 + s[i];
```

就不是把字符转成数字，而是加上字符编码对应的整数值。程序阅读题中一般会给出代码，不能自动替程序“改成想要的意思”。

字符串追加

字符串经常通过 += 追加字符或片段。

```
string t;

for (int i = 0; i < s.size(); i++) {
    if (s[i] != 'a') {
        t += s[i];
    }
}
```

若 s="banana"，则 t 依次变为：

读到字符	操作后 t
b	b
a	b
n	bn
a	bn
n	bnn
a	bnn

最终输出 bnn。

再看：

```
string t;

for (int i = s.size() - 1; i >= 0; i--) {
    t += s[i];
}
```

这段程序把字符串反向复制。若 `s="abcd"`，则 `t="dcba"`。

注意 `s.size()` 的类型在实际 C++ 中是无符号整数，但 CSP-S 程序阅读中若代码写成 `int i = s.size() - 1`，通常按给定合法输入追踪即可。读题时重点看循环方向和停止条件。

真题式例题：字符串匹配跳步

阅读下面程序：

```
int f(const string& s, const string& t) {
    int n = s.length();
    int m = t.length();

    vector<int> shift(128, m + 1);

    for (int j = 0; j < m; j++) {
        shift[t[j]] = m - j;
    }

    for (int i = 0; i <= n - m; i += shift[s[i + m]]) {
        int j = 0;

        while (j < m && s[i + j] == t[j]) {
            j++;
        }

        if (j == m) {
            return i;
        }
    }

    return -1;
}
```

这段程序看起来长，但关键变量只有四个：

变量	含义
`i`	当前匹配窗口左端
`j`	当前已经匹配到模式串的第几个字符
`shift[c]`	字符 `c` 对应的跳步长度
`m`	模式串长度

设：

```
s = "abbababbbab"
t = "abab"
```

先算 `shift`。模式串长度 $m = 4$，初始全是 5。

j	t[j]	赋值
0	a	shift['a']=4
1	b	shift['b']=3
2	a	shift['a']=2
3	b	shift['b']=1

最终 shift['a']=2，shift['b']=1。

再追踪外层循环：

i	窗口 s[i..i+3]	成功比较次数 j	是否返回	下一次 i
0	abba	2	否	i += shift[s[4]] = shift['b'] = 1
1	bbab	0	否	i += shift[s[5]] = shift['a'] = 2
3	abab	4	是	返回 3

所以函数返回 3。

这段程序最容易错在 s[i+m]。当前窗口是 s[i] 到 s[i+m-1]，s[i+m] 是窗口后面的字符，用来决定下一次跳多少。

若题目问第 20 行 j++ 执行多少次，就统计每个窗口中成功比较的字符数。上表中分别为 $2, 0, 4$，总次数为 6。

真题式例题：编码表

阅读下面程序片段：

```
char base[64];
char table[256];

void init() {
    for (int i = 0; i < 26; i++) {
        base[i] = 'A' + i;
    }

    for (int i = 0; i < 26; i++) {
        base[26 + i] = 'a' + i;
    }

    for (int i = 0; i < 10; i++) {
        base[52 + i] = '0' + i;
    }

    base[62] = '+';
    base[63] = '/';

    for (int i = 0; i < 256; i++) {
        table[i] = -1;
    }

    for (int i = 0; i < 64; i++) {
        table[base[i]] = i;
    }
}
```

这段程序先建立编号到字符的表 base，再建立字符到编号的表 table。

base 的含义如下：

编号范围	字符
0 到 25	A 到 Z
26 到 51	a 到 z
52 到 61	0 到 9
62	+
63	/

table[base[i]]=i 的意思是反向记录。例如：

- base[0]='A'，所以 table['A']=0；
- base[25]='Z'，所以 table['Z']=25；
- base[26]='a'，所以 table['a']=26；
- base[52]='0'，所以 table['0']=52；
- base[63]='/'，所以 table['/']=63。

读编码类程序时，先不要急着算复杂表达式。应先确定表的方向：

- base[x] 是“编号转字符”；
- table[ch] 是“字符转编号”。

如果把方向读反，后面所有计算都会错。

再看一个简化编码函数：

```
string encode(string s) {
    string ret;

    for (int i = 0; i < s.size(); i++) {
        int x = s[i] - 'a';
        ret += base[x];
    }

    return ret;
}
```

若 s="abc"，则 x 依次为 $0,1,2$，ret 依次追加 base[0],base[1],base[2]，结果为 ABC。

编码程序往往只是“把值拆开，再查表输出”。纸面上先写清楚表的含义，比直接代入表达式更稳。

真题式例题：哈希结构比较

阅读下面程序片段：

```
struct H {
    int h1, h2, l;

    bool operator < (const H& h) const {
        if (l != h.l) {
            return l < h.l;
        } else if (h1 != h.h1) {
            return h1 < h.h1;
        } else {
            return h2 < h.h2;
        }
    }
```

```
};

H a[100];

sort(a + 1, a + n + 1);
int m = unique(a + 1, a + n + 1) - (a + 1);
```

这段程序先定义结构体 H 的排序规则，然后排序去重。

排序规则的优先级是：

1. 先按 l 从小到大；
2. l 相同，再按 h1 从小到大；
3. l 和 h1 都相同，再按 h2 从小到大。

若有四个元素：

编号	l	h1	h2
A	2	5	1
B	1	9	3
C	2	4	8
D	2	5	0

排序后顺序为：

```
B, C, D, A
```

原因是：

- B 的 l=1 最小；
- 其余三个 l=2；
- 在 l=2 中，C 的 h1=4 小于 5；
- D 和 A 的 l,h1 相同，比较 h2，$0 < 1$，所以 D 在 A 前。

如果结构体还定义了 operator==，unique 会根据相等规则去重。若没有自定义相等规则，具体能否编译要看完整程序；在程序阅读中，题目通常会给出所有相关运算符。

读结构体数组程序时，不要只看 sort，还要看比较函数。排序后相邻关系由比较函数决定。

变式题：前缀和数组

阅读下面程序：

```
int n = 5;
int a[6] = {0, 3, -1, 4, 2, -5};
int s[6] = {0};

for (int i = 1; i <= n; i++) {
    s[i] = s[i - 1] + a[i];
}

cout << s[4] - s[1] << endl;
```

逐项计算：

i	s[i]
1	3
2	2
3	6
4	8
5	3

输出 `s[4]-s[1]=8-3=5`。它对应的是 `a[2]+a[3]+a[4]`，即 $-1+4+2=5$。

前缀和程序中，`s[r]-s[l-1]` 对应区间 $[l,r]$；`s[r]-s[l]` 对应区间 $[l+1,r]$。程序阅读题经常在这里差一位。

变式题：字符串删除字符

阅读下面程序：

```
string s = "abacaba";
string t;

for (int i = 0; i < s.size(); i++) {
    if (i == 0 || s[i] != s[i - 1]) {
        t += s[i];
    }
}

cout << t << endl;
```

这段程序保留每一段连续相同字符的第一个字符。

对于 `s="abacaba"`，相邻字符都不相等，所以 `t="abacaba"`。

若输入改成 `s="aaabbcaaa"`，追踪如下：

位置	字符	是否追加	t
0	a	是	a
1	a	否	a
2	a	否	a
3	b	是	ab
4	b	否	ab
5	c	是	abc
6	a	是	abca
7	a	否	abca
8	a	否	abca

输出 `abca`。

它不是删除所有重复字符，而是压缩连续重复段。若要删除所有重复字符，通常会配合计数数组、布尔数组或排序去重。

变式题：二维数组展开

虽然本章重点是线性数组和字符串，但程序阅读中有时会把二维数组按行扫描。

```
int a[3][4] = {
    {0, 0, 0, 0},
```

```
    {0, 1, 2, 3},
    {0, 4, 5, 6}
};

int s = 0;

for (int i = 1; i <= 2; i++) {
    for (int j = 1; j <= 3; j++) {
        if ((i + j) % 2 == 0) {
            s += a[i][j];
        }
    }
}

cout << s << endl;
```

只统计 `i+j` 为偶数的位置：

位置	值	是否计入
$(1,1)$	1	是
$(1,2)$	2	否
$(1,3)$	3	是
$(2,1)$	4	否
$(2,2)$	5	是
$(2,3)$	6	否

输出 $1+3+5=9$。

二维数组题先写行列范围，再看条件筛选。不要把 `a[i][j]` 和 `a[j][i]` 混掉。

小结

读数组与字符串程序时，可以按下面顺序检查：

检查点	要问的问题
下标起点	数组从 0 还是 1 开始使用
访问终点	最后一次访问的是哪个位置
窗口范围	`s[i..i+m-1]` 还是其他范围
修改顺序	本轮用的是原值还是已修改后的值
循环方向	从左到右和从右到左是否影响结果
间接下标	`cnt[a[i]]` 的下标代表什么
字符转换	字符是否通过 `-'0'` 或 `-'a'` 转成编号
排序影响	排序后原位置是否还有效
去重长度	`unique` 后是否使用了新的有效长度
字符串追加	`+=` 追加的是字符、字符串还是转换后的值
输出内容	输出的是整个数组、有效前缀还是某个位置

数组与字符串程序阅读要先把“位置”读准。下标、边界、方向和修改顺序一旦确定，后面的计算通常只是耐心追踪。遇到较长程序时，不要把全部数组都抄下来，只记录被访问、被修改、被输出的关键位置。

0x33 树图程序

树图程序的代码通常更长，因为它要先存边，再遍历，再根据遍历结果更新答案。读这类程序时，不要一行一行硬扫。应先判断图怎么存、从哪里开始走、每个数组表示什么，再看核心循环或递归怎样改变状态。

树图程序阅读可以按四步来做：读存图，读入口，读访问规则，读答案更新。

存图方式

树图程序首先要看边是怎么存的。

最常见的是邻接表：

```
vector<int> g[N];

for (int i = 1; i <= m; i++) {
    int u, v;
    cin >> u >> v;
    g[u].push_back(v);
}
```

这表示有一条从 u 到 v 的有向边。

如果写成：

```
g[u].push_back(v);
g[v].push_back(u);
```

表示无向边。读图程序时，第一步就是看每条输入边会加入几次。

例如输入：

```
3 2
1 2
2 3
```

若代码只有 g[u].push_back(v)，则边为：

```
1 -> 2
2 -> 3
```

若代码同时加入 g[v].push_back(u)，则边为：

```
1 - 2 - 3
```

二者的遍历结果可能完全不同。

有时程序使用链式前向星：

```
int head[N], to[M], nxt[M], tot;

void add(int u, int v) {
    ++tot;
    to[tot] = v;
    nxt[tot] = head[u];
    head[u] = tot;
}
```

这段代码的含义是：从 u 出发的一条边编号为 tot，它指向 v，并接到 u 的边表开头。

遍历时通常写成：

```
for (int e = head[u]; e; e = nxt[e]) {
    int v = to[e];
    ...
}
```

读链式前向星时，只要记住：

数组	含义
head[u]	点 u 的第一条边编号
to[e]	边 e 指向的点
nxt[e]	点 u 的下一条边编号
tot	当前边的编号

链式前向星的边遍历顺序与输入顺序可能相反。因为每次新边都插到表头。若题目问具体访问顺序，要特别注意这一点。

例如依次执行：

```
add(1, 2);
add(1, 3);
add(1, 4);
```

从 1 出发遍历时，访问顺序是 4,3,2，不是 2,3,4。

若程序使用 vector 并按输入顺序 push_back，则访问顺序与输入顺序一致。

有向与无向

图程序中，有向边和无向边会影响可达性。

```
void dfs(int u) {
    vis[u] = true;

    for (int v : g[u]) {
        if (!vis[v]) {
            dfs(v);
        }
    }
}
```

如果图是有向图，dfs(u) 只能沿边的方向走。如果图是无向图，代码必须在存边时加入两个方向，才可以来回走。

看下面输入：

```
4 3
1 2
3 2
3 4
```

若按有向边存：

```
1 -> 2
3 -> 2
3 -> 4
```

从 1 出发只能到 1,2。

若按无向边存：

```
1 - 2 - 3 - 4
```

从 1 出发能到所有点。

程序阅读题常给出一段 DFS，然后问哪些点会被访问。做这类题时，不要只看图画得像连通，要看代码是否加了反向边。

父亲与访问标记

无向图或树的 DFS 中，常用 fa 或 vis 避免走回去。

```
void dfs(int u, int fa) {
    for (int v : g[u]) {
        if (v == fa) {
            continue;
        }

        dfs(v, u);
    }
}
```

这里 fa 表示 u 的父亲。无向边会让 u 看到自己的父亲，如果不跳过父亲，就会在父子之间反复递归。

另一种写法是使用 vis：

```
void dfs(int u) {
    vis[u] = true;

    for (int v : g[u]) {
        if (!vis[v]) {
            dfs(v);
        }
    }
}
```

两种写法的区别：

写法	常见使用场景	含义
v == fa	树	只避免走回父亲
vis[v]	一般图	避免访问已经访问过的点

在树上，用 fa 通常够用；在有环图中，仅靠 fa 不一定能避免重复访问。

例如无向三角形：

```
1 - 2
2 - 3
3 - 1
```

从 1 调用 dfs(1,0)，若只跳过父亲：

- 1 可以到 2；
- 2 可以到 3；
- 3 的父亲是 2，它不会跳过 1；
- 于是 3 又回到 1。

所以一般图需要 vis。

读代码时，要看程序认为输入一定是树，还是可能是一般图。若题目保证是树，fa 写法是合理的；若输入可能有环，只靠父亲参数就要小心。

树的 DFS 数组

树上 DFS 常同时计算深度、父亲、子树大小。

```
void dfs(int u, int fa) {
    dep[u] = dep[fa] + 1;
    siz[u] = 1;

    for (int v : g[u]) {
        if (v == fa) {
            continue;
        }

        dfs(v, u);
        siz[u] += siz[v];
    }
}
```

变量含义：

变量	含义
fa	当前点 u 的父亲
dep[u]	当前点深度
siz[u]	以 u 为根的子树大小

例如树：

```
1-2
1-3
2-4
2-5
```

并从 1 开始 DFS。

若 dep[0]=0，则：

点	父亲	深度	子树大小
1	0	1	5
2	1	2	3
3	1	2	1
4	2	3	1
5	2	3	1

`siz[u]` 的计算发生在递归返回后。也就是说，只有子树已经算完，父亲才能把 `siz[v]` 加进来。

如果题目问 `siz[2]`，看的是点 2 的整个子树，包括 2 自己。这里是 $2, 4, 5$，所以为 3。

先序与后序

树上 DFS 中，语句放在递归前还是递归后，会形成不同顺序。

```
void dfs(int u, int fa) {
    cout << u << ' ';

    for (int v : g[u]) {
        if (v != fa) {
            dfs(v, u);
        }
    }
}
```

这是先访问当前点，再访问孩子。

若邻接表顺序为：

```
g[1]: 2, 3
g[2]: 1, 4, 5
g[3]: 1
g[4]: 2
g[5]: 2
```

从 1 出发输出：

```
1 2 4 5 3
```

如果把输出放到递归后：

```
void dfs(int u, int fa) {
    for (int v : g[u]) {
        if (v != fa) {
            dfs(v, u);
        }
    }

    cout << u << ' ';
}
```

输出为：

```
4 5 2 3 1
```

树图程序中，很多数组是在“进入时”赋值，很多答案是在“返回时”更新。读代码时可以在函数体上做三段标记：

```
进入当前点
遍历孩子
离开当前点
```

这样可以避免把先序和后序混掉。

BFS 队列

BFS 程序常用队列。

```cpp
queue<int> q;

vis[s] = true;
dis[s] = 0;
q.push(s);

while (!q.empty()) {
    int u = q.front();
    q.pop();

    for (int v : g[u]) {
        if (!vis[v]) {
            vis[v] = true;
            dis[v] = dis[u] + 1;
            q.push(v);
        }
    }
}
```

变量含义：

变量	含义
q	等待扩展的点
vis[v]	点 v 是否已经入队或访问
dis[v]	从起点到 v 的边数

注意 vis[v]=true 通常在入队时设置，而不是出队时设置。这样可以避免同一个点被重复入队。

看图：

```
1 - 2
1 - 3
2 - 4
3 - 4
```

从 1 开始 BFS，邻接表顺序为：

```
g[1]: 2, 3
g[2]: 1, 4
g[3]: 1, 4
g[4]: 2, 3
```

过程：

步骤	出队 u	新入队	队列变化	距离更新
初始	-	1	1	dis[1]=0

步骤	出队 u	新入队	队列变化	距离更新
1	1	2,3	2,3	dis[2]=1,dis[3]=1
2	2	4	3,4	dis[4]=2
3	3	无	4	4 已访问
4	4	无	空	-

最终 dis[4]=2。

若 vis[v] 放到出队时才设为真，点 4 可能会从 2 和 3 各入队一次。读 BFS 程序时，要看标记是在入队时还是出队时设置。

拓扑排序

拓扑排序程序常见结构如下：

```
queue<int> q;

for (int i = 1; i <= n; i++) {
    if (deg[i] == 0) {
        q.push(i);
    }
}

while (!q.empty()) {
    int u = q.front();
    q.pop();
    ans[++cnt] = u;

    for (int v : g[u]) {
        deg[v]--;

        if (deg[v] == 0) {
            q.push(v);
        }
    }
}
```

变量含义：

变量	含义
deg[v]	点 v 当前剩余入度
q	当前入度为 0、可以输出的点
ans	拓扑序
cnt	已输出点数

例如有向边：

```
1 -> 3
2 -> 3
3 -> 4
```

初始入度：

点	入度
1	0
2	0
3	2
4	1

初始队列为 `1,2`。过程：

出队	影响	新入队
1	`deg[3]` 从 2 到 1	无
2	`deg[3]` 从 1 到 0	3
3	`deg[4]` 从 1 到 0	4
4	无	无

一种拓扑序为：

```
1 2 3 4
```

如果初始队列中 2 在 1 前，或者使用优先队列，拓扑序可能不同。拓扑排序不一定唯一，程序输出哪一种，要看容器和邻接表顺序。

读拓扑排序程序时，最容易错的地方是入度减少与入队判断的先后。

正确写法通常是：

```
deg[v]--;
if (deg[v] == 0) {
    q.push(v);
}
```

如果代码写成：

```
if (deg[v] == 1) {
    q.push(v);
}
deg[v]--;
```

效果在这一句上可能相同，因为入度为 1 时减少后变为 0。但如果判断条件、减少位置或其他语句不同，结果就可能变化。程序阅读题会利用这种顺序差异设问。

路径计数

DAG 上常见路径计数程序：

```
for (int u : order) {
    f[u] = 1;

    for (int v : g[u]) {
        f[u] += f[v];
    }
}
```

这类程序通常要求 `order` 是反向拓扑序，即所有后继 `v` 的 `f[v]` 已经算好。

若 `f[u]` 初始化为 1，一般表示“只包含点 `u` 本身的路径也算一条”。若初始化为 0，含义会不同。

看一个 DAG：

```
1 -> 2
1 -> 3
2 -> 4
3 -> 4
```

从每个点出发的路径数量，若单点路径也算：

点	路径数量
4	1
2	$1 + f[4] = 2$
3	$1 + f[4] = 2$
1	$1 + f[2] + f[3] = 5$

从 1 出发的路径有：

```
1
1 2
1 2 4
1 3
1 3 4
```

共 5 条。

读路径计数程序时，要看 `f[u]` 的初值。初值为 1 通常说明空扩展或单点路径被计入；初值为 0 则往往只统计到达某些终点的路径。

真题式例题：DAG 第 k 小路径

阅读下面程序片段：

```
long long f[N], k;
vector<int> g[N];

int nxt(vector<int> cand, long long& k) {
    sort(cand.begin(), cand.end());

    for (int u : cand) {
        if (k <= f[u]) {
            return u;
        }

        k -= f[u];
    }

    return -1;
}
```

这个函数在候选点 `cand` 中选择一个点。候选点会先按编号从小到大排序。每个点 `u` 代表一段数量为 `f[u]` 的路径集合。

变量含义：

变量	含义
cand	当前可以选择的下一批点
f[u]	以 u 开始的路径数量
k	要找的第 k 条路径
返回值	第 k 条路径当前应选择的点

例如：

```
cand = {3, 1, 2}
f[1] = 4
f[2] = 2
f[3] = 5
k = 6
```

排序后候选为 1,2,3。

候选点	f[u]	当前 k	判断
1	4	6	$6 > 4$，跳过，k=2
2	2	2	$2 <= 2$，选择 2

函数返回 2，同时 k 变为 2，表示在以 2 开始的那一组路径中找第 2 条。

继续看主流程中的片段：

```
int u = nxt(all, k);
cout << u << endl;

while (k > 1) {
    k--;
    u = nxt(g[u], k);
    cout << u << endl;
}
```

这里输出路径上的点。为什么循环前要判断 k > 1？因为在以当前点 u 开始的路径中，第一条通常是只包含 u 的单点路径。如果 k==1，路径到此结束；如果 k>1，先用 k-- 跳过单点路径，再从 u 的后继中选择下一点。

读这类程序时，不要把 k 当作固定常量。它会随着跳过一整组路径而减少，表示“在剩余组内的编号”。

最短路程序

最短路程序通常有三类关键变量：

变量	含义
dis[u]	当前已知到 u 的最短距离
vis[u]	点 u 是否已经确定
优先队列元素	当前待扩展的距离和点

简化的 Dijkstra 代码：

```
priority_queue<pair<int, int>> q;

memset(dis, 0x3f, sizeof(dis));
dis[s] = 0;
q.push({0, s});

while (!q.empty()) {
    int u = q.top().second;
```

```
    int d = -q.top().first;
    q.pop();

    if (vis[u]) {
        continue;
    }

    vis[u] = true;

    for (auto e : g[u]) {
        int v = e.to;
        int w = e.w;

        if (dis[v] > d + w) {
            dis[v] = d + w;
            q.push({-dis[v], v});
        }
    }
}
```

C++ 默认优先队列是大根堆，所以常把距离取负数放进去。队列中 `{ -dis[v], v }` 的第一项越大，表示原距离越小。

例如从 1 出发：

```
1 -> 2, 权 3
1 -> 3, 权 1
3 -> 2, 权 1
```

过程：

步骤	取出点	当前距离	更新
1	1	0	`dis[2]=3, dis[3]=1`
2	3	1	`dis[2]=2`
3	2	2	无
4	2	3	已访问，跳过

最终 `dis[2]=2`。

读最短路程序时，要注意队列里可能有同一个点的旧记录。`vis[u]` 或 `if (d != dis[u]) continue;` 用来跳过旧记录。若题目问队列弹出次数，旧记录也算弹出；若问确定最短路的次数，则只看未访问时的弹出。

状态图

有些图程序不是直接在原图上跑，而是把“点 + 状态”当作新点。

例如“最多使用一次免费边”的最短路：

```
long long d[N][2];

d[s][0] = 0;
pq.push({0, s, 0});

while (!pq.empty()) {
    auto cur = pq.top();
    pq.pop();

    int u = cur.u;
```

```
    int used = cur.used;

    for (auto e : g[u]) {
        int v = e.to;
        int w = e.w;

        if (d[v][used] > d[u][used] + w) {
            d[v][used] = d[u][used] + w;
            pq.push({d[v][used], v, used});
        }

        if (used == 0 && d[v][1] > d[u][0]) {
            d[v][1] = d[u][0];
            pq.push({d[v][1], v, 1});
        }
    }
}
```

这里每个原图点拆成两个状态：

状态	含义
d[u][0]	到达 u 时还没有使用免费边
d[u][1]	到达 u 时已经使用过免费边

走一条普通边时，used 不变：

```
d[v][used] = d[u][used] + w
```

使用免费边时，只能从 used=0 转到 used=1，费用不增加：

```
d[v][1] = d[u][0]
```

最后答案通常是：

```
min(d[t][0], d[t][1])
```

因为允许“至多使用一次”，所以不用免费边也合法。

读状态图程序时，关键是解释第二维。不要把 d[u][1] 简单理解成“点 u 的另一个距离”，它表示带着某种状态到达 u。

次短路程序

次短路程序常维护两层距离。简化理解是：

数组	含义
dis[u]	到 u 的最短路
dis2[u]	到 u 的严格次短路

有些程序会把点编号拆成两层：

```
u       表示最短路层的点 u
n + u   表示次短路层的点 u
```

更新时常见两种情况：

1. 找到更短的路，原最短路下放成次短路；

2. 找到介于最短路和次短路之间的路，更新次短路。

读这类程序时，不要急着跟完整路径。先看编号是否用 `% n` 或 `+ n` 做层间转换。

例如：

```
int a = aa % n;
```

如果 aa 是状态点编号，则 a 是原图中的点编号。

```
out(n + t);
```

表示从次短路层的终点开始输出路径。

次短路程序的代码通常较长，但阅读小问经常只考某个数组或某个编号含义。先把“两层点”的含义写清楚，后面就不会乱。

树的遍历序列

二叉树程序常涉及先序、中序、后序。程序阅读题不一定让你写树，但常让你根据遍历规则判断输出。

三种遍历：

遍历	顺序
先序	根、左、右
中序	左、根、右
后序	左、右、根

代码对应：

```
void pre(int u) {
    if (!u) {
        return;
    }

    cout << u << ' ';
    pre(lc[u]);
    pre(rc[u]);
}
```

这是先序。

```
void mid(int u) {
    if (!u) {
        return;
    }

    mid(lc[u]);
    cout << u << ' ';
    mid(rc[u]);
}
```

这是中序。

```
void post(int u) {
    if (!u) {
        return;
```

```
    }

    post(lc[u]);
    post(rc[u]);
    cout << u << ' ';
}
```

这是后序。

若二叉树为：

```
    1
   / \
  2   3
 / \
4   5
```

则：

遍历	序列
先序	1 2 4 5 3
中序	4 2 5 1 3
后序	4 5 2 3 1

读二叉树程序时，看输出语句与两次递归调用的位置，就能判断遍历类型。

二叉搜索树

二叉搜索树程序常利用大小关系。

```
void insert(int& u, int x) {
    if (u == 0) {
        u = ++tot;
        val[u] = x;
        return;
    }

    if (x < val[u]) {
        insert(lc[u], x);
    } else {
        insert(rc[u], x);
    }
}
```

这里 u 是引用参数，表示当前子树根编号。若当前为空，就新建结点；否则根据 x 与 val[u] 的大小进入左子树或右子树。

读插入过程时，可以画出树。

例如插入序列：

```
6 4 10 2 5 8 12
```

得到：

```
     6
    / \
   4   10
  / \  / \
 2  5 8 12
```

先序为：

```
6 4 2 5 10 8 12
```

后序为：

```
2 5 4 8 12 10 6
```

如果题目给二叉搜索树的后序序列，也可以反推根。后序最后一个元素是根；小于根的部分在左子树，大于根的部分在右子树。

例如后序：

```
2 5 4 8 12 10 6
```

最后的 6 是根。前面小于 6 的连续部分 `2 5 4` 是左子树，大于 6 的连续部分 `8 12 10` 是右子树。继续分解：

- 左子树后序 `2 5 4`，根为 4，左为 2，右为 5；
- 右子树后序 `8 12 10`，根为 10，左为 8，右为 12。

所以先序为：

```
6 4 2 5 10 8 12
```

这类题不需要写代码，按遍历规则拆即可。

真题式例题：堆式二叉树哈希

阅读下面程序片段：

```
struct H {
    int h1, h2, l;

    H(bool b = false) {
        h1 = b;
        h2 = b + 13;
        l = 1;
    }

    H operator + (const H& h) const {
        H res;
        res.l = l + h.l;
        res.h1 = h1 * p1[h.l] + h.h1;
        res.h2 = h2 * p2[h.l] + h.h2;
        return res;
    }
};

H h[N];
bool p[N];
```

```
for (int i = n; i; i--) {
    h[i] = H(p[i]);

    if (2 * i + 1 <= n) {
        h[i] = h[2 * i] + h[i] + h[2 * i + 1];
    } else if (2 * i <= n) {
        h[i] = h[2 * i] + h[i];
    }
}
```

这段程序把编号 1 到 n 的点看成一棵按堆编号存储的二叉树：

点 i	左儿子	右儿子
i	2*i	2*i+1

循环从 n 到 1，保证处理 i 时，孩子的 h 已经算好。

合并顺序是：

```
左子树 + 当前点 + 右子树
```

这对应二叉树的中序结构编码。

若只有左儿子，没有右儿子，则合并为：

```
左子树 + 当前点
```

读这类程序时，先确定树的编号规则，再看合并顺序。若合并写成 h[i] + h[2*i] + h[2*i+1]，含义就会变成“当前点、左子树、右子树”的顺序。

程序最后常会有：

```
sort(h + 1, h + n + 1);
int m = unique(h + 1, h + n + 1) - (h + 1);
```

这表示统计不同子树结构编码的数量。若题目问第二行输出，就要比较各个 h[i] 是否相同；若题目问合并顺序，就看 operator + 在循环中的排列。

变式题：DFS 访问顺序

阅读下面程序：

```
vector<int> g[6];
bool vis[6];

void dfs(int u) {
    vis[u] = true;
    cout << u << ' ';

    for (int v : g[u]) {
        if (!vis[v]) {
            dfs(v);
        }
    }
}

int main() {
```

```
    g[1].push_back(2);
    g[1].push_back(3);
    g[2].push_back(4);
    g[3].push_back(4);
    g[4].push_back(5);

    dfs(1);
}
```

这是有向图。边为：

```
1 -> 2
1 -> 3
2 -> 4
3 -> 4
4 -> 5
```

从 1 出发：

当前点	输出	后续
1	1	先到 2
2	2	到 4
4	4	到 5
5	5	无后继
回到 1	-	再看 3
3	3	4 已访问

输出：

```
1 2 4 5 3
```

若把 `g[1].push_back(2)` 和 `g[1].push_back(3)` 的顺序交换，则输出会变成：

```
1 3 4 5 2
```

DFS 访问顺序依赖邻接表顺序。

变式题：子树大小

阅读下面程序：

```
vector<int> g[6];
int siz[6];

void dfs(int u, int fa) {
    siz[u] = 1;

    for (int v : g[u]) {
        if (v == fa) {
            continue;
        }

        dfs(v, u);
        siz[u] += siz[v];
    }
```

```
}
```

树边为：

```
1-2
1-3
2-4
2-5
```

从 `1` 调用 `dfs(1,0)`。

叶子点 $3,4,5$ 的 `siz` 都是 1。点 2 的子树包括 $2,4,5$，所以 `siz[2]=3`。点 1 的子树包括所有点，所以 `siz[1]=5`。

最终：

点	siz
1	5
2	3
3	1
4	1
5	1

注意 `siz[u] += siz[v]` 在递归之后。如果放在递归之前，`siz[v]` 还没有算好，含义就变了。

变式题：BFS 距离

阅读下面程序：

```
queue<int> q;
int dis[6];
bool vis[6];

vis[1] = true;
q.push(1);

while (!q.empty()) {
    int u = q.front();
    q.pop();

    for (int v : g[u]) {
        if (!vis[v]) {
            vis[v] = true;
            dis[v] = dis[u] + 1;
            q.push(v);
        }
    }
}
```

无向图边为：

```
1-2
1-3
2-4
3-4
4-5
```

从 1 开始：

点	最短边数
1	0
2	1
3	1
4	2
5	3

所以 `dis[5]=3`。

若题目只问距离，不必写完整队列；但若题目问出队顺序，就要根据邻接表顺序追队列。

变式题：入度变化

阅读下面程序：

```
queue<int> q;

for (int i = 1; i <= n; i++) {
    if (deg[i] == 0) {
        q.push(i);
    }
}

while (!q.empty()) {
    int u = q.front();
    q.pop();
    cout << u << ' ';

    for (int v : g[u]) {
        deg[v]--;
        if (deg[v] == 0) {
            q.push(v);
        }
    }
}
```

有向边为：

```
1 -> 4
2 -> 4
3 -> 4
4 -> 5
```

初始入度：

点	入度
1	0
2	0
3	0
4	3
5	1

若初始按编号入队，则队列为 `1,2,3`。

过程：

出队	影响
1	`deg[4]` 从 3 到 2
2	`deg[4]` 从 2 到 1
3	`deg[4]` 从 1 到 0，入队 4
4	`deg[5]` 从 1 到 0，入队 5
5	无

输出：

```
1 2 3 4 5
```

如果初始容器换成优先队列，且还有更多入度为 0 的点，输出顺序可能改变。拓扑排序的核心是“谁先变成入度 0，谁才有资格入队”。

小结

读树图程序时，可以按下面顺序检查：

检查点	要问的问题
存图	每条边加入一次还是两次
方向	代码沿有向边走，还是无向边来回走
邻接顺序	`vector` 顺序还是链式前向星逆序
入口	从哪个点开始 DFS/BFS
标记	`vis` 是入队时标记，还是访问时标记
父亲	`fa` 只用于树，还是程序可能遇到环
深度	`dep[u]` 是怎样由父亲推来的
子树	`siz[u]` 是进入时赋值，还是返回时累计
队列	BFS 或拓扑排序中的入队、出队顺序
入度	`deg[v]--` 与判断入队的先后
状态	是否把一个点拆成多个状态
输出	输出发生在进入点、离开点，还是弹出队列时

树图程序不适合从第一行读到最后一行硬算。先把图的存储方式和变量含义整理出来，再追核心遍历。DFS 看递归入口和访问顺序，BFS 看队列变化，拓扑排序看入度，最短路看距离更新，状态图看第二维含义。只要这些线索清楚，长代码也能拆成几段可控的过程。

0x34 综合程序阅读

综合程序阅读的难点不是某一行特别难，而是信息分散：输入在 `main` 中，核心逻辑在函数里，答案变量可能是全局的，判断题又常问“删掉某行是否影响结果”“某个函数的时间复杂度是多少”“输入某组数据输出什么”。这类题不能一开始就逐行模拟。先拆结构，再追细节，才能减少错漏。

综合程序阅读建议按五步进行：

1. 先看输入输出；
2. 再看全局变量和函数表；
3. 找核心流程；
4. 只追会影响题目的变量；
5. 用选项反查细节。

先读主函数

长程序先读 `main`，因为 `main` 决定程序真正执行哪些函数、执行顺序是什么。

看下面的程序：

```
int n, a[1005];

int calc1() {
    int s = 0;

    for (int i = 1; i <= n; i++) {
        s += a[i];
    }

    return s;
}

int calc2() {
    int s = 0;

    for (int i = 1; i <= n; i++) {
        if (a[i] > 0) {
            s += a[i];
        }
    }

    return s;
}

int main() {
    cin >> n;

    for (int i = 1; i <= n; i++) {
        cin >> a[i];
    }

    cout << calc2() << endl;
```

```
    return 0;
}
```

这里虽然有 `calc1` 和 `calc2` 两个函数，但 `main` 只调用了 `calc2`。如果题目问输出，就不需要计算 `calc1`。

读主函数时要记下：

内容	要看什么
输入	输入几个变量，数组下标从哪里开始
调用	调用了哪些函数，顺序是什么
分支	哪些函数可能不执行
输出	输出的是变量、函数返回值，还是表达式

有些程序会根据输入选择不同算法：

```
if (n <= 20) {
    cout << solve1() << endl;
}

cout << solve2() << endl;
```

这种程序在 `n<=20` 时会输出两行，在 `n>20` 时只输出一行。读题时要先看输出次数，否则后面的判断题很容易错。

建函数表

综合程序常有多个函数。可以先写一个函数表，不用立刻进细节。

```
int f(int x) {
    return x * x;
}

void work() {
    for (int i = 1; i <= n; i++) {
        b[i] = f(a[i]);
    }
}

int query() {
    int s = 0;

    for (int i = 1; i <= n; i++) {
        s += b[i];
    }

    return s;
}
```

函数表：

函数	作用	是否返回值	是否修改外部变量
`f`	返回 x^2	是	否
`work`	把 $a[i]^2$ 写入 `b[i]`	否	修改 `b`
`query`	求 `b` 的和	是	否

这张表的作用是让程序结构清楚。之后如果题目问 `query()` 输出什么，就知道必须先看 `work()` 是否执行；如果题目问删掉 `work()` 是否影响结果，就知道它修改了 `b`。

函数表尤其适合处理“有返回值但返回值没被用”的情况：

```
check(x);
```

与：

```
if (check(x)) {
    ...
}
```

前者只有在 check 内部修改了全局变量或输出内容时才会影响后续；后者会使用返回值影响分支。

全局变量要单独列

综合程序中，全局变量往往跨函数使用。读代码前可以先列出来。

```
int n, m;
int a[1005], b[1005];
int ans;
bool vis[1005];
```

可以整理为：

变量	可能含义	在哪里改
n,m	规模	main 输入
a	原数组	main 输入
b	辅助数组	某个函数更新
ans	答案	初始化、统计函数
vis	标记	搜索或遍历函数

全局变量要特别看初始化。全局数组默认初值为 0，但程序中也可能手动清空：

```
memset(vis, 0, sizeof(vis));
ans = 0;
```

如果一个函数被多次调用，而全局变量没有重新初始化，前一次调用的结果可能影响后一次。

例如：

```
int cnt = 0;

void work(int n) {
    for (int i = 1; i <= n; i++) {
        cnt += i;
    }
}

int main() {
    work(3);
    work(2);
    cout << cnt << endl;
}
```

第一次 work(3) 后 cnt=6，第二次 work(2) 后 cnt=9，输出 9。这里 cnt 没有在 work 内清零。

如果程序改成：

```
void work(int n) {
    cnt = 0;

    for (int i = 1; i <= n; i++) {
        cnt += i;
    }
}
```

输出就变成 3。全局变量的初始化位置会改变程序含义。

分清“准备工作”和“核心计算”

长程序通常由几段组成：

```
读入
预处理
主计算
输出
```

读程序时要先把段落分出来。

例如：

```
for (int i = 0; i < 26; i++) {
    mp['a' + i] = i;
}

for (int i = 0; i < s.size(); i++) {
    a[i] = mp[s[i]];
}

for (int i = 0; i < s.size(); i++) {
    ans += a[i];
}

cout << ans << endl;
```

第一段是建立字符编号表，第二段是把字符串转成数字数组，第三段才是求和。

若题目问时间复杂度，要把三段都算进去；若题目问某个输入的输出，可以先简化为：每个字符 `'a'+i` 转成 i，最后求和。

例如 `s="abca"`：

字符	编号
a	0
b	1
c	2
a	0

输出 3。

预处理代码有时看起来很长，但后面只用它查表。读这种程序时，先把表的含义写成一句话，再看主计算。

不要平均用力

综合程序阅读最忌每一行都同等用力。题目问什么，就追什么。

若题目问输出，重点是影响输出的变量。

若题目问时间复杂度，重点是循环层数、递归规模、排序和二分次数。

若题目问某行删除是否影响结果，重点是这行是否修改了后续会使用的状态。

例如：

```
int f(int n) {
    int s = 0;

    for (int i = 1; i <= n; i++) {
        s += i;
    }

    return s;
}

int main() {
    int n;
    cin >> n;

    f(n);

    cout << n << endl;
}
```

`f(n)` 虽然执行了，但返回值没有使用，也没有修改全局变量，输出只与 `n` 有关。若题目问输出，不需要算 `f(n)` 的返回值；若题目问时间复杂度，`f(n)` 仍然要计入。

同一段代码，对不同问题的重要性不同。

真题式例题：双算法输出

阅读下面程序：

```
int n;
int a[1005];

int solve1(int l, int r) {
    if (l > r) {
        return 0;
    }

    if (l == r) {
        return max(a[l], 0);
    }

    int mid = (l + r) / 2;

    int left = solve1(l, mid);
    int right = solve1(mid + 1, r);

    int s = 0, bestl = 0;

    for (int i = mid; i >= l; i--) {
        s += a[i];
        bestl = max(bestl, s);
    }
```

```
    s = 0;
    int bestr = 0;

    for (int i = mid + 1; i <= r; i++) {
        s += a[i];
        bestr = max(bestr, s);
    }

    return max(max(left, right), bestl + bestr);
}

int solve2() {
    int cur = 0, ans = 0;

    for (int i = 1; i <= n; i++) {
        cur = max(0, cur + a[i]);
        ans = max(ans, cur);
    }

    return ans;
}

int main() {
    cin >> n;

    for (int i = 1; i <= n; i++) {
        cin >> a[i];
    }

    cout << solve1(1, n) << endl;
    cout << solve2() << endl;
    return 0;
}
```

先读 `main`：

- 读入 `n` 和数组 `a[1..n]`；
- 输出 `solve1(1,n)`；
- 输出 `solve2()`。

程序一定输出两行。

再建函数表：

函数	作用
`solve1(l,r)`	用分治求区间最大非负连续子段和
`solve2()`	从左到右递推求最大非负连续子段和

两者求的是同一个量，但过程不同。

若输入为：

```
6
-2 3 -1 4 -5 2
```

用 `solve2` 追踪更快：

i	a[i]	cur=max(0,cur+a[i])	ans
1	-2	0	0
2	3	3	3
3	-1	2	3
4	4	6	6
5	-5	1	6
6	2	3	6

所以第二行输出 6。`solve1` 求同一问题，第一行也输出 6。

若题目问 `solve1` 的时间复杂度，就不能用 `solve2` 的线性过程代替。`solve1` 每层分治会扫描当前区间，层数约为 $\log n$，总时间为 $O(n \log n)$。`solve2` 是 $O(n)$。

这类双算法程序常用来考：

- 两个函数输出是否相同；
- 两个函数复杂度是否相同；
- 哪个函数更快；
- 某个函数的局部变量含义。

读的时候先确定“功能是否相同”，再分别分析复杂度。

真题式例题：选择执行路径

阅读下面程序：

```
int n, m;
string s;
int dp[1 << 10];

int solve1() {
    int ans = 0;

    for (int mask = 0; mask < (1 << n); mask++) {
        int cnt = 0;
        int num = 0;

        for (int i = 0; i < n; i++) {
            if (mask & (1 << i)) {
                num = num * 2 + (s[i] - '0');
                cnt++;
            }
        }

        if (cnt <= m) {
            ans += num;
        }
    }

    return ans;
}

int solve2() {
    dp[0] = 1;

    for (int i = 0; i < n; i++) {
        for (int j = (1 << (m - 1)) - 1; j >= 0; j--) {
```

```
            int k = (j << 1) | (s[i] - '0');

            if (j != 0 || s[i] == '1') {
                dp[k] += dp[j];
            }
        }
    }

    int ans = 0;

    for (int i = 0; i < (1 << m); i++) {
        ans += i * dp[i];
    }

    return ans;
}

int main() {
    cin >> n >> m;
    cin >> s;

    if (n <= 20) {
        cout << solve1() << endl;
    }

    cout << solve2() << endl;
    return 0;
}
```

这段程序较长。先不急着算 dp，先看主函数：

条件	输出
n <= 20	先输出 solve1()，再输出 solve2()
n > 20	只输出 solve2()

所以如果题目给 n=8，程序输出两行；如果给 n=30，程序输出一行。

再看 solve1。它枚举所有 mask，从字符串中选出若干位置，按原顺序组成一个二进制数 num，若选出位置数 cnt<=m，就把 num 加入答案。

例如 n=3,m=2,s="101"。所有可选位置不超过 2 个：

选择的位置	形成的二进制	数值
空	0	0
1	1	1
2	0	0
3	1	1
1, 2	10	2
1, 3	11	3
2, 3	01	1

总和为 8。所以 solve1() 返回 8。

solve2 是用动态规划统计同类结果，程序阅读题不一定要求手算所有 dp。若题目问 solve1 的含义，用枚举解释更清楚；若题目问 solve2 的复杂度，则看两层循环：

- 外层 i 执行 n 次；
- 内层 j 从 (1<<(m-1))-1 到 0，数量约为 2^{m-1}；

- 最后求和约为 2^m。

所以 `solve2` 的时间复杂度是 $O(n2^m + 2^m)$，可写成 $O(n2^m)$。

综合程序中，常有一个好理解但慢的函数和一个更快但难读的函数。读题时可以用慢函数理解问题含义，用快函数分析实际复杂度。

真题式例题：生成、递归和排序

阅读下面程序：

```
int c[1005];

int logic(int x, int y) {
    return (x & y) ^ ((x ^ y) | (~x & y));
}

void generate(int a, int b) {
    for (int i = 0; i < b; i++) {
        c[i] = logic(a, i) % (b + 1);
    }
}

void rec(int dep, int* arr, int siz) {
    if (dep <= 0 || siz <= 1) {
        return;
    }

    int pivot = arr[0];
    int i = 0, j = siz - 1;

    while (i <= j) {
        while (arr[i] < pivot) {
            i++;
        }

        while (arr[j] > pivot) {
            j--;
        }

        if (i <= j) {
            int t = arr[i];
            arr[i] = arr[j];
            arr[j] = t;
            i++;
            j--;
        }
    }

    rec(dep - 1, arr, j + 1);
    rec(dep - 1, arr + i, siz - i);
}

int main() {
    int a, b, d;
    cin >> a >> b >> d;

    generate(a, b);
    rec(d, c, b);
```

```
    for (int i = 0; i < b; i++) {
        cout << c[i] << ' ';
    }

    cout << endl;
    return 0;
}
```

这段程序分成三块：

代码块	作用
`logic` 和 `generate`	生成数组 `c[0..b-1]`
`rec`	对数组做有限深度的划分交换
`main` 输出	输出数组前 `b` 个数

读综合程序时，先拆块，再决定每块追到什么程度。

若题目问 `logic` 的功能，可以按二进制位逐位化简。设当前这一位的 `x,y` 分别为 0 或 1。

x	y	$x\&y$	$x\oplus y$	$\sim x\&y$	$(x\oplus y)\mid(\sim x\&y)$	完整表达式结果
0	0	0	0	0	0	0
0	1	0	1	1	1	1
1	0	0	1	0	1	1
1	1	1	0	0	0	1

最后一列分别为 $0,1,1,1$，与按位或的结果相同，所以 `logic(x,y)` 的功能等价于 C++ 表达式 `x | y`。

若题目给 `a=5,b=5`，先生成数组：

`i`	`logic(5,i)`	`%6` 后
0	5	5
1	5	5
2	7	1
3	7	1
4	5	5

生成后：

```
5 5 1 1 5
```

若 `d=1`，只做一层划分。`pivot=5`，双指针交换过程：

操作	数组	`i`	`j`
初始	`5 5 1 1 5`	0	4
交换 `c[0],c[4]`	`5 5 1 1 5`	1	3
交换 `c[1],c[3]`	`5 1 1 5 5`	2	2
`c[2]<5, i++`	`5 1 1 5 5`	3	2

退出后，因为 `dep` 减到 0，递归不再继续。输出：

```
5 1 1 5 5
```

这类程序看起来包含位运算、数组生成、递归划分。处理方法仍然是拆块：先求生成数组，再追一次 `rec`，最后输出。

如果题目问时间复杂度，就不要花时间手算数组，而是看 `rec` 的结构。每层划分会扫描当前段，递归深度最多为 `d`，若划分较均匀，接近每层总扫描 b；但最坏情况下仍可能出现不均衡划分。具体选项要结合题目给出的范围和说法判断。

真题式例题：表、编码与解码

阅读下面程序片段：

```
char base[64];
char table[256];

void init() {
    for (int i = 0; i < 26; i++) {
        base[i] = 'A' + i;
    }

    for (int i = 0; i < 26; i++) {
        base[26 + i] = 'a' + i;
    }

    for (int i = 0; i < 10; i++) {
        base[52 + i] = '0' + i;
    }

    base[62] = '+';
    base[63] = '/';

    for (int i = 0; i < 256; i++) {
        table[i] = -1;
    }

    for (int i = 0; i < 64; i++) {
        table[base[i]] = i;
    }

    table['='] = 0;
}
```

综合程序中，初始化表通常占不少篇幅。读这段不需要死记每个字符，只要整理出方向：

表	作用
`base[x]`	编号 x 转字符
`table[ch]`	字符 `ch` 转编号

继续看编码函数的核心：

```
ret += base[str[i] >> 2];
ret += base[(str[i] & 0x03) << 4 | str[i + 1] >> 4];
ret += base[(str[i + 1] & 0x0f) << 2 | str[i + 2] >> 6];
ret += base[str[i + 2] & 0x3f];
```

这段每次处理 3 个字符，生成 4 个字符。题目如果问字符串长度变化，就不必计算每一位，只要看分组：

原字符串剩余长度	编码结果
每满 3 个字符	变成 4 个字符
剩 1 个字符	生成 2 个有效字符，再补 ==
剩 2 个字符	生成 3 个有效字符，再补 =

若题目问 encode 的时间复杂度，循环按字符串长度线性推进，答案是 $O(n)$。

若题目问某个具体字符的 table 值，就查初始化表：

- table['A']=0;
- table['Z']=25;
- table['a']=26;
- table['0']=52;
- table['/']=63;
- table['=']=0。

编码类综合程序通常分成 “建表” 和 “按组处理”。建表解决映射方向，按组处理解决长度和输出。

真题式例题：折半枚举

阅读下面程序：

```
int n, m;
int k[20], p[20];
vector<int> a, b;

int mpow(int x, int y) {
    int res = 1;

    while (y) {
        if (y & 1) {
            res *= x;
        }

        x *= x;
        y >>= 1;
    }

    return res;
}

void dfs(vector<int>& vec, int l, int r, int val) {
    if (l > r) {
        vec.push_back(val);
        return;
    }

    for (int x = 1; x <= m; x++) {
        dfs(vec, l + 1, r, val + k[l] * mpow(x, p[l]));
    }
}

int main() {
    cin >> n >> m;

    for (int i = 1; i <= n; i++) {
        cin >> k[i] >> p[i];
    }
```

```
    dfs(a, 1, n / 2, 0);
    dfs(b, n / 2 + 1, n, 0);

    sort(a.begin(), a.end());

    long long ans = 0;

    for (int x : b) {
        ans += upper_bound(a.begin(), a.end(), -x)
             - lower_bound(a.begin(), a.end(), -x);
    }

    cout << ans << endl;
    return 0;
}
```

先读 `main`，程序分成：

1. 读入 n, m 和每一项的系数、指数；
2. 枚举前半部分所有取值，放入 `a`；
3. 枚举后半部分所有取值，放入 `b`；
4. 对 `a` 排序；
5. 对每个 `b` 中的值，统计 `a` 中有多少个数等于它的相反数；
6. 输出总数。

核心是求满足下式的方案数，其中每个 x_i 都在 1 到 m 之间：$\sum_{i=1}^{n} k_i x_i^{p_i} = 0$。

`dfs` 的变量含义：

变量	含义
`vec`	存放枚举结果的数组
`l,r`	当前枚举的项范围
`val`	当前已经累加出的值
`x`	当前项选择的变量值

如果前半部分有 q 项，每项有 m 种选择，则 `a` 的长度为 m^q。后半部分类似。因此这是折半枚举。

例如：

```
n=2, m=3
k[1]=1, p[1]=2
k[2]=-1, p[2]=2
```

前半部分枚举 x_1^2：

x_1	值
1	1
2	4
3	9

后半部分枚举 $-x_2^2$：

x_2	值
1	-1
2	-4
3	-9

每个后半值 x 都要找 a 中等于 -x 的数量，所以有 $(1,1),(2,2),(3,3)$ 三种方案，输出 3。

读这类综合程序时，递归 dfs 不必画完整树。先确定它枚举什么，再看排序和二分统计什么。

判断题的处理

综合程序阅读中的判断题常见有几类。

第一类：某个函数是否一定输出两行、是否一定返回正确结果。

处理方法：看 main 的分支和输出语句。

```
if (n <= 20) {
    cout << solve1() << endl;
}
cout << solve2() << endl;
```

它不是总输出两行，而是在 n<=20 时输出两行。

第二类：删掉某行是否影响结果。

处理方法：判断该行是否影响后续会使用的状态。

```
sort(a.begin(), a.end());

for (int x : b) {
    ans += upper_bound(a.begin(), a.end(), x)
         - lower_bound(a.begin(), a.end(), x);
}
```

这里后面使用二分函数查找，要求 a 有序。删掉排序会影响结果。

但如果排序的是一个后面不再使用的数组，删掉可能不影响输出。不要看到 sort 就自动判断有影响，要看后续有没有依赖有序性。

第三类：替换某个表达式是否不影响结果。

处理方法：看类型、运算顺序和边界。

例如：

```
int mid = (l + r) / 2;
```

替换成：

```
int mid = l + (r - l) / 2;
```

在不溢出的前提下结果相同。若题目没有保证不溢出，就要考虑 l+r 是否可能超出范围。

又如：

```
a / b / c
```

与：

```
a / (b / c)
```

不是一回事。整数除法尤其要按代码顺序计算。

第四类：复杂度判断。

处理方法：只看规模变化，不代入具体数据。

```
for (int i = 1; i <= n; i++) {
    for (int j = i; j <= n; j += i) {
        ...
    }
}
```

总次数是 $\sum_{i=1}^{n} \lfloor n/i \rfloor$，量级为 $O(n \log n)$。

```
for (int i = 1; i <= n; i *= 2) {
    for (int j = 1; j <= n; j++) {
        ...
    }
}
```

外层约 $\log n$ 次，内层 n 次，总时间 $O(n \log n)$。

综合程序里可能同时有排序、递归、双指针、二分。复杂度取各部分之和，再保留主导项。

选择题的处理

选择题可以从选项反推，不必完整算完所有内容。

例如题目问输出，选项为：

```
A. 0
B. 1
C. 7
D. 100
```

若程序中 `ans` 是统计一个长度为 5 的数组中满足条件的元素个数，那么答案范围一定在 0 到 5，可以先排除 7 和 100。

若题目问某个布尔表达式结果，选项有 `true,false,1,0`，要看输出语句：

```
cout << res << endl;
```

默认输出 `bool` 时通常输出 `0` 或 `1`，不是单词 `true` 或 `false`。如果前面写了：

```
cout << boolalpha;
```

才会输出 `true` 或 `false`。

若题目问函数功能，选项可能写成“求最大值”“求最小值”“统计不同值”“判断是否有序”。可以先看函数返回的变量是如何更新的：

```
ans = max(ans, x);
```

多半与最大值有关；

```
if (!vis[x]) {
    vis[x] = true;
    ans++;
}
```

多半与不同元素个数有关；

```
return l;
```

若前面是二分，就看 check 的真假含义，判断返回的是最小可行值还是最大可行值。

选择题不要求展示过程，但做题时过程要可靠。先用选项缩小范围，再对关键变量追踪，可以节省时间。

综合追踪表

面对长程序，可以使用“综合追踪表”，只记录关键阶段。

例如：

```
读入数组
排序
去重
统计
输出
```

可以写成：

阶段	结果
读入	4 1 4 2 2
排序	1 2 2 4 4
去重有效部分	1 2 4
有效长度	3
输出	3

这种表比逐行记录每个变量更适合综合程序。

再如图论程序：

阶段	结果
建图	有向边为 $(1,2),(1,3),(2,4),(3,4)$
初始入度	deg[1]=0, deg[2]=1, deg[3]=1, deg[4]=2
初始队列	1
输出顺序	1 2 3 4

综合追踪表不要求列出所有细节，只要能支撑题目要问的结论。

变式题：综合模拟

阅读下面程序，写出输出。

```
int n;
int a[10], b[10];

void work() {
    for (int i = 1; i <= n; i++) {
        b[i] = a[i];
    }

    sort(b + 1, b + n + 1);

    int m = unique(b + 1, b + n + 1) - (b + 1);
```

```
    cout << m << endl;

    int ans = 0;

    for (int i = 1; i <= n; i++) {
        int pos = lower_bound(b + 1, b + m + 1, a[i]) - b;
        ans += pos * i;
    }

    cout << ans << endl;
}

int main() {
    cin >> n;

    for (int i = 1; i <= n; i++) {
        cin >> a[i];
    }

    work();
    return 0;
}
```

输入：

```
5
4 1 4 2 2
```

先看程序结构：

1. 把 a 复制到 b；
2. 排序 b；
3. 去重，得到不同元素个数 m；
4. 对每个 a[i] 找它在去重数组中的位置 pos；
5. 累加 pos*i。

追踪表：

阶段	结果
初始 a	4 1 4 2 2
排序后 b	1 2 2 4 4
去重后有效 b[1..m]	1 2 4
m	3

第一行输出 3。

继续算第二行：

i	a[i]	pos	pos*i
1	4	3	3
2	1	1	2
3	4	3	9
4	2	2	8
5	2	2	10

ans=3+2+9+8+10=32。

输出：

```
3
32
```

这个程序混合了数组复制、排序、去重和二分查找。用阶段表比逐行模拟更清楚。

变式题：删行判断

阅读下面程序：

```
int n;
int a[1005];

int count_same() {
    sort(a + 1, a + n + 1);

    int ans = 0;

    for (int i = 2; i <= n; i++) {
        if (a[i] == a[i - 1]) {
            ans++;
        }
    }

    return ans;
}
```

判断：删去 `sort(a + 1, a + n + 1);` 后，函数返回值一定不变。

这个判断是错误的。

反例：

```
n = 4
a = 1 2 1 2
```

有排序时，数组变成：

```
1 1 2 2
```

相邻相等出现 2 次，返回 2。

无排序时，原数组为：

```
1 2 1 2
```

没有相邻相等，返回 0。

所以排序影响结果。

删行判断题的基本方法是找反例。只要能找到一组输入使结果改变，就说明“不影响结果”是错的。

变式题：两次调用

阅读下面程序：

```cpp
int cnt = 0;

int work(int n) {
    for (int i = 1; i <= n; i++) {
        cnt += i;
    }

    return cnt;
}

int main() {
    cout << work(3) << endl;
    cout << work(2) << endl;
    return 0;
}
```

第一次调用 `work(3)`：

```
cnt = 1 + 2 + 3 = 6
```

输出 6。

第二次调用 `work(2)` 时，`cnt` 没有清零，从 6 开始继续加：

```
cnt = 6 + 1 + 2 = 9
```

输出 9。

最终输出：

```
6
9
```

若 `cnt` 是 `work` 内的局部变量，第二次调用就不会继承第一次的结果。综合程序中，多次调用同一个函数时，要看函数内部使用的是局部变量还是全局变量。

变式题：输出行数

阅读下面程序：

```cpp
int solve(int n) {
    int s = 0;

    for (int i = 1; i <= n; i++) {
        s += i;
    }

    return s;
}

int main() {
    int n;
    cin >> n;

    if (n % 2 == 0) {
        cout << solve(n) << endl;
    }
```

```
    if (n % 3 == 0) {
        cout << solve(n / 3) << endl;
    }

    cout << n << endl;
    return 0;
}
```

若输入 6：

- 6 是偶数，输出 solve(6)=21；
- 6 能被 3 整除，输出 solve(2)=3；
- 最后输出 6。

输出三行：

```
21
3
6
```

若输入 5：

- 不是偶数；
- 不能被 3 整除；
- 只输出 5。

输出一行：

```
5
```

综合程序阅读中，题目不一定只问输出内容，也可能问输出行数。读 main 的分支比计算函数更优先。

考场顺序

综合程序阅读可以采用下面的纸面顺序：

步骤	操作
第一步	圈出 main 中的输入、函数调用和输出
第二步	给每个函数写一句话功能
第三步	标出全局变量和会被修改的数组
第四步	根据题目小问决定追哪一段
第五步	用表格追关键变量，不追无关变量
第六步	用选项检查范围、行数、复杂度
第七步	回看边界：下标、循环退出、初始化、修改顺序

小问之间也可以互相帮助。前面一问常常会提示函数功能，后面一问会要求具体输入输出或复杂度。不要把每个小问割裂开做。

例如第一问问“该函数是否总能返回一个非负数”，如果判断出函数是在求最大非负子段和，后面问某个输入输出时就可以用这个理解简化计算。

常见失误

综合程序阅读中，常见失误有下面几类。

失误	后果
不看 `main`，直接读第一个函数	可能计算了根本没有执行的函数
忽略输出条件	输出行数判断错误
忽略全局变量未清零	多次调用结果错误
把排序前位置当成排序后位置	数组含义错
忽略 `unique` 的有效长度	把旧数据算入答案
把递归返回前后的语句顺序看反	输出顺序错
把 `break` 当成跳出所有循环	嵌套循环结果错
把队列、优先队列、栈的弹出顺序混淆	遍历顺序错
只看算法名称，不看具体代码	边界和变式题易错
只算返回值，不算函数调用次数	调用次数题易错

避免这些错误，靠的不是把程序从头到尾抄一遍，而是先建立结构图。

小结

做综合程序阅读题时，可以按下面清单检查：

检查点	要问的问题
`main`	程序从哪里开始，调用哪些函数
输入	数组下标从 0 还是 1 开始
输出	输出几行，输出的变量何时被修改
函数	返回值是否使用，是否修改全局变量
全局变量	是否在多次调用之间保留值
预处理	建的表或数组后面怎样使用
排序去重	后续是否依赖有序性和有效长度
循环	次数、退出状态、`break`、`continue`
递归	停止条件、调用次数、返回顺序
图结构	存边方向、遍历顺序、标记时机
复杂度	各段相加后主导项是什么
选项	是否可用范围、奇偶、行数先排除

综合程序阅读不是把所有知识点重新学一遍，而是把前面学过的方法按顺序用起来。长程序先拆块，短输入再追值；问输出就追输出，问复杂度就追规模，问删改就追状态依赖。做到这一点，第八编的方法就能连成完整的一套。

第九编 完善程序专项

0x35 初始化与边界

本章进入第九编“完善程序专项”。前八编已经讲过主要知识点和程序阅读方法，从本章开始，重点转向“代码已经给出大部分，空缺处应该怎样补”。

完善程序题和程序阅读题不同。程序阅读题常问程序做了什么；完善程序题还要判断：空缺处处在什么流程中，它影响的是初始化、循环边界、状态转移、条件判断，还是最终输出。本章先讲最常见、也最容易丢分的一类空缺：初始化与边界。

初始化和边界看起来不难，但在完善程序题中经常藏得很深。一个 `0`、`1`、`n`、`n-1`、`INF`、`d[s][0]`，填错以后程序整体思路可能仍像对的，结果却会在最小数据或特殊数据上出错。

先判断空缺位置

看到空缺处，第一步不是代选项，而是判断它所在的位置。

空缺大致可以分成几类：

位置	常见作用
变量第一次赋值	初始化
循环条件	控制范围
数组下标	确定边界
`if` 条件	决定分支
递推初值	确定起点
队列或堆初始插入	确定搜索入口
返回语句	确定最终答案
特殊情况判断	处理空区间、叶子、起点、终点

本章只处理初始化与边界，因此空缺多半会出现在这些地方：

```
f[0] = ①;
for (int i = ②; i <= n; i++) ...
while (l < ③) ...
d[s][④] = 0;
return ⑤;
```

如果空缺处在变量第一次赋值处，就优先问“这个变量的含义是什么”。如果空缺处在循环条件里，就优先问“这个循环应该覆盖哪些位置”。如果空缺处在返回语句里，就优先问“题目要求的答案到底对应哪个状态”。

初始化不是随便给值

初始化要和变量含义一致。

看下面的程序：

```
int n;
int a[1005], s[1005];

cin >> n;
```

```
for (int i = 1; i <= n; i++) {
    cin >> a[i];
}

s[0] = ①;

for (int i = 1; i <= n; i++) {
    s[i] = s[i - 1] + a[i];
}
```

这里 `s[i]` 表示前 i 个数的和，所以 `s[0]` 表示前 0 个数的和，应为 0。① 应填 `0`。

如果把 `s[0]` 填成 `1`，后面所有前缀和都会多 1。程序结构看起来仍然完整，但含义已经错了。

再看一个最大值初始化：

```
int mx = ①;

for (int i = 1; i <= n; i++) {
    mx = max(mx, a[i]);
}
```

若题目保证 `a[i]` 都是非负整数，① 可以填 `0`。但若 `a[i]` 可能为负数，填 `0` 就可能错误。例如数组全是负数时，最大值不应该是 0。

更稳的写法通常是：

```
int mx = a[1];

for (int i = 2; i <= n; i++) {
    mx = max(mx, a[i]);
}
```

所以完善程序中，遇到最大值、最小值初始化，要看数据范围和循环起点：

目标	常见初始化
求和	`0`
求积	`1`
最大值，且可能有负数	第一个元素或很小的数
最小值	第一个元素或很大的数
计数	`0`
判断是否存在	`false`
判断是否全部满足	`true`

初始化不是看哪个选项顺眼，而是看它对应的数学含义。

下标从哪里开始

边界题首先看数组下标。

```
for (int i = 1; i <= n; i++) {
    cin >> a[i];
}
```

说明有效位置是 `a[1]` 到 `a[n]`。

如果后面要处理相邻元素：

```
int cnt = 0;

for (int i = ①; i <= n; i++) {
    if (a[i] > a[i - 1]) {
        cnt++;
    }
}
```

因为访问了 a[i-1]，最小的 i 应该是 2，所以① 应填 2。若从 1 开始，会访问 a[0]，它不属于输入序列。

如果数组从 0 开始读入：

```
for (int i = 0; i < n; i++) {
    cin >> a[i];
}
```

处理相邻元素时可以写：

```
for (int i = 1; i < n; i++) {
    if (a[i] > a[i - 1]) {
        cnt++;
    }
}
```

此时最后一个合法位置是 a[n-1]，循环条件应是 i < n，不是 i <= n。

完善程序题中，同样是“从第二个元素开始”，若下标从 1 开始，循环变量从 2 开始；若下标从 0 开始，循环变量从 1 开始。

区间左右端点

区间程序的边界要先判断区间写法。

常见有两种：

写法	含义
[l,r]	左右端都包含
[l,r)	包含左端，不包含右端

看下面的递归：

```
void solve(int l, int r) {
    if (l == r) {
        ans += a[l];
        return;
    }

    int mid = (l + r) / 2;
    solve(l, mid);
    solve(mid + 1, r);
}
```

这里使用闭区间 [l,r]。当 l==r 时，区间内只有一个元素。左右分裂为 [l,mid] 和 [mid+1,r]，不会重叠，也不会漏掉。

如果使用左闭右开区间：

```
void solve(int l, int r) {
    if (l + 1 == r) {
        ans += a[l];
        return;
    }

    int mid = (l + r) / 2;
    solve(l, mid);
    solve(mid, r);
}
```

这里区间 `[l,r)` 中只有一个元素时，满足 `l+1==r`。左右分裂为 `[l,mid)` 和 `[mid,r)`。

两种写法不能混用。若递归调用写成 `solve(l,mid)` 和 `solve(mid+1,r)`，那停止条件通常对应闭区间；若递归调用写成 `solve(l,mid)` 和 `solve(mid,r)`，那通常对应左闭右开区间。

完善程序中遇到 `l,r,mid`，不要只看选项。先判断整个程序使用哪种区间。

循环端点

循环边界要和目标数量对应。

例如求长度为 m 的所有连续子串：

```
for (int i = 0; ①; i++) {
    string t = s.substr(i, m);
    ...
}
```

若 `s` 长度为 `n`，从位置 `i` 开始取长度 m，最后一个字符下标是 `i+m-1`。必须满足 `i+m-1<n`，也就是 `i+m<=n`。

所以① 可写为：

```
i + m <= n
```

若选项中有 `i + m < n`，它会漏掉最后一个合法窗口；若有 `i <= n`，则会越过窗口边界。

再看前缀和查询：

```
for (int i = 1; i <= n; i++) {
    s[i] = s[i - 1] + a[i];
}

cout << s[r] - s[①] << endl;
```

若要求区间 `[l,r]` 的和，① 应填 `l - 1`。因为 `s[r]` 是前 r 项和，减去前 $l-1$ 项，剩下 l 到 r。

若要求区间 `[l,r)` 的和，答案应是 `s[r-1]-s[l-1]` 或在 0 下标前缀和中写成 `s[r]-s[l]`。这要看前缀和数组的定义。

边界题没有统一模板，只有统一的检查方法：把第一个合法位置和最后一个合法位置代进去，看是否刚好覆盖。

二分边界

二分填空常考 `l`、`r`、`mid` 和更新语句。

看下面的程序：

```cpp
int l = 1, r = n;

while (l < r) {
    int mid = (l + r) / 2;

    if (check(mid)) {
        r = mid;
    } else {
        l = mid + 1;
    }
}

cout << l << endl;
```

这段程序寻找第一个使 `check(x)` 为真的位置。这里 `r=mid` 保留了 `mid`，说明当 `mid` 可行时，答案可能就是 `mid`；`l=mid+1` 排除了不可行的 `mid`。

如果空缺是：

```cpp
if (check(mid)) {
    ①;
} else {
    ②;
}
```

那么① 应填 `r = mid`，② 应填 `l = mid + 1`。

再看另一种：

```cpp
int l = 1, r = n;

while (l < r) {
    int mid = (l + r + 1) / 2;

    if (check(mid)) {
        l = mid;
    } else {
        r = mid - 1;
    }
}

cout << l << endl;
```

这段程序寻找最后一个使 `check(x)` 为真的位置。`mid` 使用上中位数，是为了让 `l` 能推进。

判断二分空缺时，可以看三点：

位置	需要判断
`mid`	下中位数还是上中位数
条件为真	保留左半还是右半
更新语句	是否排除 `mid`

不要只背“真就右边界，假就左边界”。要看 `check(mid)` 的含义是“够了”还是“不够”，答案找最小值还是最大值。

DP 初值

DP 程序的初始化通常对应最小规模。

看一个简单递推：

```
f[1] = ①;
f[2] = ②;

for (int i = 3; i <= n; i++) {
    f[i] = f[i - 1] + f[i - 2];
}
```

若题目定义 `f[1]=1,f[2]=1`，那么①、② 都应填 `1`。

如果题目定义：

```
f[0]=0, f[1]=1
```

循环就可能写成：

```
f[0] = ①;
f[1] = ②;

for (int i = 2; i <= n; i++) {
    f[i] = f[i - 1] + f[i - 2];
}
```

此时① 应填 `0`，② 应填 `1`。

DP 初值不能只看递推式。递推式只说明后面的状态怎样来，初值来自题目定义或最小情况。

再看计数 DP：

```
dp[0] = ①;

for (int i = 1; i <= n; i++) {
    for (int j = i; j <= n; j++) {
        dp[j] += dp[j - i];
    }
}
```

这里 `dp[j]` 常表示凑出和为 j 的方案数。`dp[0]` 表示凑出和为 0 的方案数。什么都不选是一种方案，所以① 应填 `1`。

若 `dp[0]` 填成 `0`，所有状态都会从 0 推不出来。

这种 “空方案” 为 1 的初始化在计数题中很常见。它不是因为 0 本身有一个数值，而是因为递推需要一个起点。

最短路初始化

最短路程序的初始化通常是：

```
for (int i = 1; i <= n; i++) {
    d[i] = INF;
}

d[s] = 0;
```

含义是：起点到自己的距离为 0，其他距离暂时未知，用一个很大的数表示。

如果是带状态的最短路，例如每个点有两个状态：

```
for (int i = 1; i <= n; i++) {
    d[i][0] = d[i][1] = INF;
}

d[s][0] = 0;
```

这里 `0` 状态可能表示“还没有使用某种特殊操作”，`1` 状态表示“已经使用过”。若题目是“至多使用一次免费边”，起点时还没有使用免费边，所以应初始化 `d[s][0]=0`，而不是 `d[s][1]=0`。

如果空缺在：

```
d[s][①] = 0;
pq.push({0, s, ①});
```

就要根据状态含义填。若 `0` 表示未使用，① 应填 `0`。

最终答案也要看题目是“恰好使用”还是“至多使用”。

```
cout << min(d[t][0], d[t][1]) << endl;
```

适合“至多使用一次”。因为不用特殊操作也合法。

```
cout << d[t][1] << endl;
```

适合“必须使用一次”。因为只有状态 1 表示已经使用。

完善程序题中，起点初始化和终点返回经常成对出现。只填对起点还不够，还要看最终取哪个状态。

队列初始状态

BFS、拓扑排序、搜索程序常在初始化阶段把入口放进队列。

BFS：

```
queue<int> q;
vis[s] = true;
dis[s] = 0;
q.push(s);
```

起点是 `s`，距离为 0，并且入队前或入队时设为已访问。

若空缺是：

```
vis[①] = true;
dis[①] = 0;
q.push(①);
```

通常应填 `s`。

拓扑排序：

```
for (int i = 1; i <= n; i++) {
    if (deg[i] == 0) {
        q.push(i);
    }
}
```

初始队列中应放入所有入度为 0 的点，而不是只放编号为 1 的点，也不是放出度为 0 的点。

如果空缺是：

```
if (①) {
    q.push(i);
}
```

通常应填 `deg[i] == 0`。

搜索程序的入口也要看题意。例如迷宫从 `(sx,sy)` 出发：

```
vis[sx][sy] = true;
q.push({sx, sy});
```

如果题目给终点 `(tx,ty)`，不能把终点当成起点，除非程序明确是从终点反向搜索。

递归边界

递归填空首先看停止条件。

```
int f(int n) {
    if (①) {
        return 1;
    }

    return n * f(n - 1);
}
```

若函数求 $n!$，通常停止条件是 `n == 0` 或 `n == 1`，对应返回 1。如果题目保证 $n \geq 1$，写 `n == 1` 即可；如果允许 $n = 0$，就要支持 `n == 0`。

区间递归：

```
void solve(int l, int r) {
    if (①) {
        ans += a[l];
        return;
    }

    int mid = (l + r) / 2;
    solve(l, mid);
    solve(mid + 1, r);
}
```

这里使用闭区间 `[l,r]`，单元素边界是 `l == r`。

若代码写成：

```
solve(l, mid);
solve(mid, r);
```

则很可能使用左闭右开区间，单元素边界应是 `l + 1 == r`。

递归边界要和递归分裂方式一起看。只填停止条件，不看下面怎么分裂，很容易选错。

特殊值 INF

很多程序用 `INF` 表示“不可能”“还没算出”“很大的初值”。

```
const int INF = 1e9;
int f[1005];

for (int i = 0; i <= n; i++) {
    f[i] = INF;
}

f[0] = 0;

for (int i = 1; i <= n; i++) {
    for (int j = w[i]; j <= m; j++) {
        f[j] = min(f[j], f[j - w[i]] + c[i]);
    }
}
```

若 `f[j]` 表示达到状态 j 的最小代价，那么初始时除了起点 `f[0]=0`，其他状态都未知，应设为 `INF`。

最大化问题则常用很小的初值：

```
for (int i = 0; i <= n; i++) {
    f[i] = -INF;
}

f[0] = 0;
```

如果求最大价值，不能把所有状态都初始化为 0。否则原本不可达的状态也会被当成价值为 0 的可达状态。

判断用 `INF` 还是 `0`，要看状态是否一定可达：

含义	常见初值
最小代价，未知状态	`INF`
最大价值，未知状态	`-INF`
方案数，未知但可累加	`0`
起点代价	`0`
空方案数量	`1`

真题式例题：免费边最短路

给定一个带权无向图，从 `s` 到 `t`，允许至多选择一条边免费通过一次。下面程序使用 Dijkstra 处理两个状态：

```
const long long INF = 1e18;

vector<pair<int, int>> g[N];
long long d[N][2];

void solve(int n, int s, int t) {
    for (int i = 1; i <= n; i++) {
        d[i][0] = d[i][1] = INF;
    }

    priority_queue<State, vector<State>, greater<State>> pq;

    d[s][①] = 0;
```

```
    pq.push({0, s, ①});

    while (!pq.empty()) {
        State cur = pq.top();
        pq.pop();

        int u = cur.u;
        int used = cur.used;

        if (cur.dist != d[u][used]) {
            continue;
        }

        for (auto e : g[u]) {
            int v = e.first;
            int w = e.second;

            if (d[v][used] > d[u][used] + w) {
                d[v][used] = d[u][used] + w;
                pq.push({d[v][used], v, used});
            }

            if (used == 0 && d[v][1] > d[u][used]) {
                d[v][1] = d[u][used];
                pq.push({d[v][1], v, 1});
            }
        }
    }

    cout << ② << endl;
}
```

先解释状态：

状态	含义
d[u][0]	到达 u 时还没有使用免费边
d[u][1]	到达 u 时已经使用过免费边

起点时还没有走任何边，更没有使用免费边，因此① 应填 `0`。

题目说“至多选择一条边免费”，不用免费边也合法。因此答案应在 `d[t][0]` 和 `d[t][1]` 中取较小值，② 应填：

```
min(d[t][0], d[t][1])
```

如果题目改成“必须使用一条免费边”，② 就应填 `d[t][1]`。同一份代码，题意中“至多”和“必须”会改变最终边界。

这类题的空缺虽然出现在图论程序中，但本质是状态初始化和答案边界。

真题式例题：归并第 k 小

有两个长度均为 n 的递增数组 `a1` 和 `a2`，要求合并后的第 k 小值。下面程序使用二分式缩小范围：

```
int solve(int* a1, int* a2, int n, int k) {
    int l1 = 0, r1 = n - 1;
    int l2 = 0, r2 = n - 1;

    while (l1 <= r1 && l2 <= r2) {
```

```
        int m1 = (l1 + r1) >> 1;
        int m2 = (l2 + r2) >> 1;

        int cnt = ①;

        if (a1[m1] <= a2[m2]) {
            if (cnt < k) {
                l1 = m1 + 1;
            } else {
                r2 = m2 - 1;
            }
        } else {
            if (cnt < k) {
                l2 = m2 + 1;
            } else {
                r1 = m1 - 1;
            }
        }
    }

    if (l1 > r1) {
        if (l1 == 0) {
            return a2[k - 1];
        } else {
            int x = a1[l1 - 1];
            int y = ②;
            return max(x, y);
        }
    } else {
        if (l2 == 0) {
            return a1[k - 1];
        } else {
            int x = a2[l2 - 1];
            int y = ③;
            return max(x, y);
        }
    }
}
```

这段程序较难，但本章只看边界。

m1 前面有 m1+1 个数，m2 前面有 m2+1 个数。因此：

```
cnt = (m1 + 1) + (m2 + 1)
```

所以① 应填：

```
(m1 + 1) + (m2 + 1)
```

后面的边界表示某一个数组的搜索区间已经空了。

若 l1 > r1，说明 a1 的边界已经确定在 l1 附近。此时要从 a2 中找与 a1[l1-1] 配合的边界位置。已经从 a1 中确定了 l1 个元素在答案左侧或附近，所以 a2 中对应下标是 k-l1-1，② 应填：

```
a2[k - l1 - 1]
```

同理，③ 应填：

```
a1[k - l2 - 1]
```

这类题不用急着完全理解整个二分过程。先抓住数组下标从 0 开始、前缀数量是“下标加 1”，再处理最后的 `k-1` 边界。

变式题：窗口最后位置

补全下面程序，统计字符串 `s` 中长度为 `m` 且首尾字符相同的子串个数。

```
int count(string s, int m) {
    int n = s.size();
    int ans = 0;

    for (int i = 0; ①; i++) {
        if (s[i] == s[i + m - 1]) {
            ans++;
        }
    }

    return ans;
}
```

窗口从 `i` 到 `i+m-1`，最后一个字符必须满足 `i+m-1<n`，所以 `i+m<=n`。

① 应填：

```
i + m <= n
```

若填 `i + m < n`，会漏掉最后一个窗口；若填 `i < n`，会访问越界位置。

变式题：空方案初值

补全下面程序，计算用若干个正整数凑出和为 `n` 的方案数，其中每个数可以使用多次。

```
int dp[1005];

dp[①] = ②;

for (int x = 1; x <= n; x++) {
    for (int j = x; j <= n; j++) {
        dp[j] += dp[j - x];
    }
}

cout << dp[n] << endl;
```

`dp[j]` 表示凑出和为 j 的方案数。递推从 `dp[j-x]` 转移而来，要让第一个数能被选出来，必须有 `dp[0]=1`，表示凑出和为 0 的空方案有一种。

所以：

```
dp[0] = 1;
```

① 应填 `0`，② 应填 `1`。

如果 `dp[0]=0`，所有方案数都会保持 0。

变式题：最小代价初始化

补全下面程序。给定若干物品，每个物品有体积 `w[i]` 和代价 `c[i]`，求恰好凑出体积 `m` 的最小代价。

```
const int INF = 1e9;
int f[1005];

for (int j = 0; j <= m; j++) {
    f[j] = ①;
}

f[0] = ②;

for (int i = 1; i <= n; i++) {
    for (int j = m; j >= w[i]; j--) {
        f[j] = min(f[j], f[j - w[i]] + c[i]);
    }
}
```

`f[j]` 表示凑出体积 j 的最小代价。最开始只有体积 0 可达，代价为 0；其他体积还不可达，应设为 `INF`。

所以① 应填 `INF`，② 应填 `0`。

若所有 `f[j]` 都初始化为 0，程序会把所有体积都当成已经可以免费凑出，结果会偏小。

变式题：递归停止条件

补全下面程序，计算数组区间 `[l,r]` 中所有元素之和。

```
int sum(int l, int r) {
    if (①) {
        return a[l];
    }

    int mid = (l + r) / 2;
    return sum(l, mid) + sum(mid + 1, r);
}
```

递归调用使用 `[l,mid]` 和 `[mid+1,r]`，这是闭区间写法。区间只剩一个元素时，满足 `l==r`，返回 `a[l]`。

① 应填：

```
l == r
```

如果填 `l + 1 == r`，长度为 1 的区间不能及时停止，会继续分裂出错误区间。

小结

遇到初始化与边界空缺，可以按下面顺序检查：

检查点	要问的问题
变量含义	这个变量表示和、积、最大值、最小值、方案数还是距离
初始状态	最小规模或起点是什么
数组下标	有效范围从 0 开始还是从 1 开始
区间类型	使用 `[l,r]` 还是 `[l,r)`
循环端点	第一个合法位置和最后一个合法位置是否覆盖
二分目标	找最小可行值还是最大可行值
状态维度	`0`、`1` 等状态分别表示什么
终点答案	取单一状态还是多个状态的最优值
不可达状态	应初始化为 `INF`、`-INF` 还是 `0`
空方案	计数 DP 中 `dp[0]` 是否应为 1
递归边界	停止条件是否和分裂方式一致

完善程序题中的初始化与边界，靠猜很容易错。正确做法是先说清变量含义，再代入最小样例检查。把 $n=1$、空区间、起点、终点、最后一个窗口这些边界想清楚，很多空缺就只剩一个合理选项。

0x36 二分与查找

二分与查找在完善程序题中很常见。题面往往已经给出大部分代码，空缺处可能是循环条件、mid 的计算、左右边界更新、lower_bound 或 upper_bound 的返回位置，也可能是“统计不超过某个值的数量”的判断式。只要有一个边界填错，程序可能在大多数样例上看起来正常，却在恰好相等、重复元素、最左端、最右端这些位置出错。

做二分与查找填空，不要先背模板。要先判断：这个程序要找的是一个位置、一个值，还是一个答案；找的是第一个满足条件的位置，还是最后一个满足条件的位置；比较中相等时应该归入左边还是右边。

查找什么

二分与查找题先判断目标。常见目标有四类：

目标	常见返回
找某个值是否存在	true/false 或位置
找第一个不小于 x 的位置	lower_bound
找第一个大于 x 的位置	upper_bound
找最小可行答案	答案二分
找最大可行答案	答案二分

同样是二分，目标不同，边界更新就不同。

看下面代码：

```
int l = 1, r = n;

while (l < r) {
    int mid = (l + r) / 2;

    if (a[mid] >= x) {
        r = mid;
    } else {
        l = mid + 1;
    }
}

cout << l << endl;
```

这里数组有序，条件 a[mid] >= x 表示位置 mid 已经“够靠右的值”。当条件成立时，答案可能在 mid，也可能在更左边，所以保留 mid，令 r=mid。它找的是第一个满足 a[pos]>=x 的位置。

再看：

```
int l = 1, r = n;

while (l < r) {
    int mid = (l + r + 1) / 2;

    if (a[mid] <= x) {
        l = mid;
    } else {
```

```
        r = mid - 1;
    }
}

cout << l << endl;
```

这里条件 a[mid] <= x 表示位置 mid 仍然不超过 x。当条件成立时，答案可能在 mid，也可能在更右边，所以保留 mid，令 l=mid。它找的是最后一个满足 a[pos]<=x 的位置。

补二分空缺时，先把目标写成一句话，再填代码。

左闭右闭

左闭右闭区间 [l,r] 中，l 和 r 都是候选答案。

找第一个满足条件的位置：

```
while (l < r) {
    int mid = (l + r) / 2;

    if (check(mid)) {
        r = mid;
    } else {
        l = mid + 1;
    }
}
```

这里 check(mid) 成立时，mid 不能丢，因为它可能就是第一个满足条件的位置。

找最后一个满足条件的位置：

```
while (l < r) {
    int mid = (l + r + 1) / 2;

    if (check(mid)) {
        l = mid;
    } else {
        r = mid - 1;
    }
}
```

这里用上中位数。若仍写 (l+r)/2，在 l+1==r 且 check(l) 为真、check(r) 为假时，mid=l，执行 l=mid 后边界不变，循环无法继续推进。

可以用一个最小区间检查。

设 l=5,r=6。

若找最后一个满足条件的位置，mid 应该取 6：

```
mid = (5 + 6 + 1) / 2 = 6
```

这样无论 check(6) 真还是假，区间都会缩小。

左闭右闭写法中，填空常见位置如下：

空缺	判断方法
`mid = ?`	找第一个满足条件用下中位数；找最后一个满足条件用上中位数
`check(mid)` 为真后更新	看 `mid` 是否仍可能是答案
`check(mid)` 为假后更新	通常排除 `mid`
最终输出	循环结束时 `l==r`，输出 `l` 或 `r` 都可

左闭右开

左闭右开区间 `[l,r)` 中，`l` 是候选位置，`r` 不是候选位置。许多库函数的思路接近这种写法。

找第一个满足 `a[pos]>=x` 的位置：

```
int l = 0, r = n;

while (l < r) {
    int mid = (l + r) / 2;

    if (a[mid] >= x) {
        r = mid;
    } else {
        l = mid + 1;
    }
}

return l;
```

这里返回值可能是 n，表示所有元素都小于 `x`。

例如 `a={1,3,3,7}`，下标从 0 开始：

x	返回位置	含义
0	0	第一个元素已经不小于 0
3	1	第一个 3 的位置
5	3	第一个不小于 5 的元素是 7
9	4	不存在，返回末尾位置

找第一个满足 `a[pos]>x` 的位置：

```
int l = 0, r = n;

while (l < r) {
    int mid = (l + r) / 2;

    if (a[mid] > x) {
        r = mid;
    } else {
        l = mid + 1;
    }
}

return l;
```

同样，返回值可能是 n。

左闭右开写法的好处是返回位置可以直接表示“插入位置”。补空时要记住：`r=n` 合法，因为它不被访问；访问的永远是 `mid`，而 `mid<r`，所以 `mid` 最大为 $n-1$。

lower_bound 与 upper_bound

完善程序中经常手写类似 lower_bound 或 upper_bound 的函数。

lower_bound(a,a+n,x) 返回第一个不小于 x 的位置。

判断条件是：

```
a[mid] >= x
```

手写为：

```
int lower(int a[], int n, int x) {
    int l = 0, r = n;

    while (l < r) {
        int mid = (l + r) / 2;

        if (a[mid] >= x) {
            r = mid;
        } else {
            l = mid + 1;
        }
    }

    return l;
}
```

upper_bound(a,a+n,x) 返回第一个大于 x 的位置。

判断条件是：

```
a[mid] > x
```

手写为：

```
int upper(int a[], int n, int x) {
    int l = 0, r = n;

    while (l < r) {
        int mid = (l + r) / 2;

        if (a[mid] > x) {
            r = mid;
        } else {
            l = mid + 1;
        }
    }

    return l;
}
```

二者差别只在相等时归到哪边：

函数	条件成立时缩右边界	相等时
lower_bound	a[mid] >= x	归入右边界，继续向左找
upper_bound	a[mid] > x	不归入右边界，向右跳过相等元素

若数组为 1 2 2 2 5：

查找	返回下标
lower_bound(2)	1
upper_bound(2)	4
upper_bound(2)-lower_bound(2)	3

所以统计某个值出现次数时，常写：

```
upper_bound(a, a + n, x) - lower_bound(a, a + n, x)
```

若手写填空，看到“统计等于 x 的数量”，通常要找到第一个 >=x 的位置和第一个 >x 的位置。

指针返回

有些完善程序手写查找函数，返回的是指针。

```
int* upper_bound(int* a, int* an, int x) {
    int l = 0, r = an - a;

    while (l < r) {
        int mid = (l + r) >> 1;

        if (①) {
            r = mid;
        } else {
            l = mid + 1;
        }
    }

    return ②;
}
```

这里 a 指向数组开头，an 指向数组末尾的后一位。长度是 an-a。

如果函数名是 upper_bound，它要返回第一个大于 x 的位置，所以① 应填：

```
a[mid] > x
```

返回第 l 个位置的指针，应写：

```
a + l
```

所以② 应填 a + l。

不要把 an-a-1 当作右边界。因为这里使用的是左闭右开写法，r 应该等于长度，而不是最后一个下标。只要循环中不直接访问 a[r]，r=an-a 是安全的。

统计不超过某值的数量

很多二分答案题需要一个函数，统计有多少个数不超过 x。

例如两个有序数组 a、b，要统计所有 a[i]+b[j] 中不超过 sum 的数量。常见写法：

```
long long get_rank(int sum) {
    long long cnt = 0;

    for (int i = 0; i < n; i++) {
```

```
        cnt += upper_bound(b, b + n, sum - a[i]) - b;
    }

    return cnt;
}
```

对于固定的 a[i]，要求：

```
a[i] + b[j] <= sum
```

等价于：

```
b[j] <= sum - a[i]
```

由于 b 有序，upper_bound(b,b+n,sum-a[i]) 返回第一个大于 sum-a[i] 的位置。这个位置减去 b，就是不超过 sum-a[i] 的元素个数。

例如：

```
a = {1, 3}
b = {2, 4, 6}
sum = 7
```

对 a[0]=1，要求 b[j]<=6，有 3 个。

对 a[1]=3，要求 b[j]<=4，有 2 个。

总共 5 对。

这种统计函数常用于求第 k 小的和。

如果题目要求“严格小于 sum 的数量”，条件就会变成：

```
b[j] < sum - a[i]
```

此时应使用 lower_bound(b,b+n,sum-a[i])，因为它返回第一个不小于该值的位置，前面的元素都严格小于该值。

相等是否计入，是这类填空最常见的分界点。

第 k 小答案

二分第 k 小值时，常用“排名函数”判断当前值是否足够大。

假设 rank(x) 表示小于等于 x 的数有多少个。要找第 k 小值，就找最小的 x，使：

```
rank(x) >= k
```

代码常写成：

```
int l = minv, r = maxv;

while (l < r) {
    int mid = ((long long)l + r) >> 1;

    if (rank(mid) >= k) {
        r = mid;
    } else {
```

```
        l = mid + 1;
    }
}

return l;
```

若空缺为：

```
if (①) {
    l = mid + 1;
} else {
    r = mid;
}
```

则① 应填：

```
rank(mid) < k
```

因为如果不超过 `mid` 的数量还不到 k，说明第 k 小值比 `mid` 大，应把左边界移到 `mid+1`。

如果写成 `rank(mid) <= k`，当 `mid` 恰好已经覆盖第 k 个数时，会把可能答案跳过。重复元素较多时尤其容易出错。

例如所有数为：

```
2 2 2 5
```

第 2 小是 2。当 `mid=2` 时，`rank(2)=3`，已经满足 `rank(mid)>=2`，应保留 `mid`，不能右移。

二分第 k 小，本质是找最小可行值。条件 “数量是否至少为 k” 决定了更新方向。

真题式例题：序列合并第 k 小和

有两个长度为 `n` 的单调不降序列 `a` 和 `b`，求所有 `a[i]+b[j]` 中第 `k` 小的和。下面程序需要补全：

```
int* upper(int* a, int* an, int x) {
    int l = 0, r = ①;

    while (l < r) {
        int mid = (l + r) >> 1;

        if (②) {
            r = mid;
        } else {
            l = mid + 1;
        }
    }

    return ③;
}

long long get_rank(int sum) {
    long long cnt = 0;

    for (int i = 0; i < n; i++) {
        cnt += upper(b, b + n, sum - a[i]) - b;
    }
```

```
    return cnt;
}

int solve() {
    int l = 0, r = ④;

    while (l < r) {
        int mid = ((long long)l + r) >> 1;

        if (⑤) {
            l = mid + 1;
        } else {
            r = mid;
        }
    }

    return l;
}
```

逐空分析。

upper 是手写 upper_bound，区间为 [a,an)，长度是 an-a，所以① 应填：

```
an - a
```

upper_bound 找第一个大于 x 的位置，所以② 应填：

```
a[mid] > x
```

返回第 l 个位置的指针，所以③ 应填：

```
a + l
```

solve 中答案的最大可能值是两个序列最大值之和。由于序列单调不降，最大值为 a[n-1] 和 b[n-1]，所以④ 应填：

```
a[n - 1] + b[n - 1]
```

get_rank(mid) 表示不超过 mid 的和有多少个。若这个数量小于 k，说明 mid 太小，应右移左边界。所以⑤ 应填：

```
get_rank(mid) < k
```

这道题的关键不在写二分模板，而在理解 get_rank(sum)：它数的是不超过 sum 的数对数量。二分找的是最小的 sum，使数量达到 k。

真题式例题：归并数组第 k 小

有两个长度均为 n 的递增数组 a1、a2，要求合并后的第 k 小。下面程序采用二分缩小两个数组的候选范围：

```
int solve(int* a1, int* a2, int n, int k) {
    int left1 = 0, right1 = n - 1;
    int left2 = 0, right2 = n - 1;

    while (left1 <= right1 && left2 <= right2) {
        int m1 = (left1 + right1) >> 1;
```

```
        int m2 = (left2 + right2) >> 1;

        int cnt = ①;

        if (②) {
            if (cnt < k) {
                left1 = m1 + 1;
            } else {
                right2 = m2 - 1;
            }
        } else {
            if (cnt < k) {
                left2 = m2 + 1;
            } else {
                right1 = m1 - 1;
            }
        }
    }

    if (③) {
        if (left1 == 0) {
            return a2[k - 1];
        } else {
            int x = a1[left1 - 1], y = ④;
            return max(x, y);
        }
    } else {
        if (left2 == 0) {
            return a1[k - 1];
        } else {
            int x = a2[left2 - 1], y = ⑤;
            return max(x, y);
        }
    }
}
```

先处理①。数组下标从 0 开始，`a1[0..m1]` 有 `m1+1` 个数，`a2[0..m2]` 有 `m2+1` 个数。因此：

```
cnt = (m1 + 1) + (m2 + 1)
```

② 要判断当前两个中点谁更小。若 `a1[m1] <= a2[m2]`，并且 `cnt < k`，说明两数组前面这些数还不够 k 个，需要在较小的 `a1` 中继续向右排除一部分，所以进入 `left1=m1+1` 分支。因此② 应填：

```
a1[m1] <= a2[m2]
```

③ 判断第一个数组的候选区间是否已经空。候选区间为空对应：

```
left1 > right1
```

所以③ 应填 `left1 > right1`。

④、⑤ 是最后边界。若 `left1 > right1`，说明 `a1` 的分割位置在 `left1`，已经有 `left1` 个元素位于分割左侧。第 k 小附近还要从 `a2` 中取下标 `k-left1-1` 的元素作比较。因此④ 应填：

```
a2[k - left1 - 1]
```

同理，⑤ 应填：

```
a1[k - left2 - 1]
```

这道题代码不短，但空缺都围绕“下标从 0 开始时，前缀长度等于下标加 1”。若把 `m1+m2` 当成数量，就会少算两个元素。

真题式例题：第 k 小距离

给定数组 `a`，要求所有数对距离 `|a[i]-a[j]|` 中第 `k` 小的值。下面程序先排序，再二分答案。

```
bool check(vector<int>& a, int m, int k) {
    int s = 0;

    for (int i = 0, j = 0; i < a.size(); i++) {
        while (a[i] - a[j] > m) {
            j++;
        }

        s += i - j;
    }

    return s >= k;
}

int solve(vector<int>& a, int k) {
    sort(a.begin(), a.end());

    int l = 0;
    int r = ①;

    while (l < r) {
        int mid = ②;

        if (check(a, mid, k)) {
            ③;
        } else {
            ④;
        }
    }

    return l;
}
```

先看 `check`。数组排序后，`a[i]-a[j]` 是距离。`while` 把 `j` 推到第一个满足 `a[i]-a[j]<=m` 的位置。此时以 `i` 为右端的合法左端有 `i-j` 个。所以 `check(a,m,k)` 判断距离不超过 `m` 的数对是否至少有 k 对。

要找第 k 小距离，就是找最小的 `m`，使 `check` 为真。

答案最小为 0，最大为最大值减最小值，所以① 应填：

```
a.back() - a[0]
```

找最小可行值，使用下中位数即可，② 应填：

```
(l + r) / 2
```

如果 `check(mid)` 为真，说明 `mid` 已经够大，答案可能是 `mid` 或更小，所以③ 应填：

```
r = mid
```

否则 mid 太小，④ 应填：

```
l = mid + 1
```

这类题中，check 的真假含义决定二分更新方向。只要确定它是“数量至少为 k”，后面的更新就清楚了。

查找表与映射

查找不一定都是二分。完善程序中也会出现用数组或表进行查找。

```
int pos[1005];

for (int i = 1; i <= n; i++) {
    pos[a[i]] = i;
}
```

这里 pos[x] 表示值 x 出现的位置。如果 a[i] 不重复，之后可以用 pos[x] 快速找到位置。

若值可能重复：

```
pos[a[i]] = i;
```

会保留最后一次出现的位置。

如果要保留第一次出现的位置，常写：

```
if (pos[a[i]] == 0) {
    pos[a[i]] = i;
}
```

这里 0 作为“还没有出现”的标记，前提是合法位置从 1 开始。若合法位置从 0 开始，就不能用 0 表示未出现，通常要初始化为 -1。

填空时若看到：

```
memset(pos, -1, sizeof(pos));

for (int i = 0; i < n; i++) {
    if (pos[a[i]] == ①) {
        pos[a[i]] = i;
    }
}
```

① 应填 -1，因为数组位置从 0 开始，0 是合法位置。

查找表题的核心是看“未出现”用什么值表示，以及合法下标是否会和这个值冲突。

坐标压缩中的查找

坐标压缩常见流程：

```
for (int i = 1; i <= n; i++) {
    b[i] = a[i];
```

```
}

sort(b + 1, b + n + 1);
int m = unique(b + 1, b + n + 1) - (b + 1);

for (int i = 1; i <= n; i++) {
    a[i] = lower_bound(b + 1, b + m + 1, a[i]) - b;
}
```

这里 `b[1..m]` 是去重后的有序值。`lower_bound` 找到原值在去重数组中的位置。因为 `b` 从下标 1 开始有效，返回位置减去 `b` 后，压缩值也是 1 到 m。

例如：

```
a = 40 10 40 20
```

复制排序去重后：

```
b[1..3] = 10 20 40
```

压缩结果：

原值	压缩值
40	3
10	1
40	3
20	2

得到：

```
3 1 3 2
```

若代码写成：

```
lower_bound(b, b + m, a[i]) - b
```

则常对应 `b[0..m-1]` 有效，压缩值为 0 到 $m-1$。填空时要根据有效区间起点判断结果是否需要加 1。

坐标压缩题常在 `lower_bound` 的范围和返回值上设空。先看去重后的有效区间，再看程序希望压缩值从 0 开始还是从 1 开始。

二分中的相等情况

很多二分填空错在相等情况。

看查找第一个不小于 `x`：

```
if (a[mid] >= x) {
    r = mid;
} else {
    l = mid + 1;
}
```

相等时应保留 `mid` 并向左找。

查找第一个大于 `x`：

```
if (a[mid] > x) {
    r = mid;
} else {
    l = mid + 1;
}
```

相等时要跳过 `mid`，因为它还不大于 `x`。

查找最后一个不大于 `x`：

```
if (a[mid] <= x) {
    l = mid;
} else {
    r = mid - 1;
}
```

相等时应保留 `mid` 并向右找。

查找最后一个小于 `x`：

```
if (a[mid] < x) {
    l = mid;
} else {
    r = mid - 1;
}
```

相等时要排除 `mid`。

可以整理为：

目标	成立条件	相等时
第一个 >=x	`a[mid] >= x`	保留，向左
第一个 >x	`a[mid] > x`	跳过，向右
最后一个 <=x	`a[mid] <= x`	保留，向右
最后一个 <x	`a[mid] < x`	排除，向左

二分题的相等情况要根据“第一个”还是“最后一个”、“严格”还是“不严格”来判断。

变式题：手写 lower_bound

补全函数，返回有序数组 `a[0..n-1]` 中第一个不小于 `x` 的位置。

```
int lower(int a[], int n, int x) {
    int l = 0, r = ①;

    while (l < r) {
        int mid = (l + r) / 2;

        if (②) {
            r = mid;
        } else {
            l = ③;
        }
    }

    return l;
}
```

使用左闭右开区间，右端初值是 n，所以① 填：

```
n
```

找第一个不小于 x，条件是 a[mid] >= x，所以② 填：

```
a[mid] >= x
```

条件不成立时，说明 a[mid] < x，mid 不可能是答案，应从 mid+1 继续，所以③ 填：

```
mid + 1
```

完整为：

```
int lower(int a[], int n, int x) {
    int l = 0, r = n;

    while (l < r) {
        int mid = (l + r) / 2;

        if (a[mid] >= x) {
            r = mid;
        } else {
            l = mid + 1;
        }
    }

    return l;
}
```

变式题：统计等于 x 的个数

补全函数，返回有序数组中等于 x 的元素个数。

```
int count_x(int a[], int n, int x) {
    int L = lower_bound(a, a + n, x) - a;
    int R = ①;
    return ②;
}
```

等于 x 的区间为：

```
[lower_bound(x), upper_bound(x))
```

所以① 应填：

```
upper_bound(a, a + n, x) - a
```

数量是右端减左端，所以② 应填：

```
R - L
```

若用 lower_bound(a,a+n,x+1) 代替 upper_bound(a,a+n,x)，在整数数组中也常可行；但如果类型不是整数，或者 x+1 会溢出，就不合适。完善程序中优先按题目选项和数据类型判断。

变式题：最小可行答案

补全代码。check(x) 表示当答案为 x 时是否可行，且若 x 可行，则更大的值也可行。求最小可行值。

```
int l = 0, r = 1000000;

while (l < r) {
    int mid = (l + r) / 2;

    if (check(mid)) {
        ①;
    } else {
        ②;
    }
}

cout << l << endl;
```

因为要找最小可行值，check(mid) 为真时，mid 可能是答案，也可能答案更小，所以保留 mid，① 填：

```
r = mid
```

若 check(mid) 为假，答案必须更大，② 填：

```
l = mid + 1
```

变式题：最大可行答案

补全代码。check(x) 表示当答案为 x 时是否可行，且若 x 可行，则更小的值也可行。求最大可行值。

```
int l = 0, r = 1000000;

while (l < r) {
    int mid = ①;

    if (check(mid)) {
        ②;
    } else {
        ③;
    }
}

cout << l << endl;
```

要求最大可行值，check(mid) 为真时，要保留 mid 并向右找，所以② 填：

```
l = mid
```

为了保证 l+1==r 时能推进，mid 要取上中位数，① 填：

```
(l + r + 1) / 2
```

若 check(mid) 为假，说明 mid 太大，③ 填：

```
r = mid - 1
```

变式题：查找插入位置

补全代码，将 x 插入有序数组 a[0..n-1] 后仍保持有序，并要求插入到所有相等元素之前。

```
int pos = ①;

for (int i = n; i > pos; i--) {
    a[i] = a[i - 1];
}

a[pos] = x;
n++;
```

要求插入到所有相等元素之前，就是找第一个不小于 x 的位置，所以① 应填：

```
lower_bound(a, a + n, x) - a
```

若要求插入到所有相等元素之后，则应使用：

```
upper_bound(a, a + n, x) - a
```

“相等元素之前” 对应 lower_bound，“相等元素之后” 对应 upper_bound。

变式题：压缩值从 1 开始

补全代码，将数组值压缩到 1 到 m。

```
for (int i = 1; i <= n; i++) {
    b[i] = a[i];
}

sort(b + 1, b + n + 1);
int m = unique(b + 1, b + n + 1) - ①;

for (int i = 1; i <= n; i++) {
    a[i] = lower_bound(b + 1, b + m + 1, a[i]) - ②;
}
```

unique(b+1,b+n+1) 返回有效区间的后一位指针。若有效区间从 b+1 开始，长度应为：

```
unique(...) - (b + 1)
```

所以① 填：

```
(b + 1)
```

这样得到的 m 是不同值数量。

后面希望压缩值从 1 开始。lower_bound 返回 b+1 时，应得到 1。因此应减去 b，不是 b+1。② 填：

```
b
```

若减去 b+1，压缩值会从 0 开始。

小结

做二分与查找填空时，可以按下面顺序检查：

检查点	要问的问题
查找目标	找值、找位置，还是找答案
单调性	条件真假是否有分界点
第一个还是最后一个	找最左满足还是最右满足
严格还是不严格	相等时归入哪一边
区间写法	`[l,r]` 还是 `[l,r)`
`mid`	下中位数还是上中位数
更新语句	条件成立时是否保留 `mid`
返回值	返回下标、指针，还是答案值
指针范围	`an-a` 是长度，`a+l` 是返回位置
统计数量	`upper-lower`，还是 `lower` 本身
第 k 小	判断“不超过 mid 的数量是否至少为 k”
压缩	压缩值从 0 还是从 1 开始

二分与查找填空的关键，是把“比较条件”和“答案目标”对齐。相等时是否保留、`mid` 是否还能作为答案、返回位置是否可能等于 `n`，这些都要在填空前想清楚。只要目标和区间写法一致，代码选项就不会乱。

0x37 排序与归并

排序与归并类完善程序有一个共同特点：代码框架通常已经给出，空缺处看似只是一个比较符号、一个下标、一个循环边界或一次复制，但它决定了程序到底按什么顺序处理元素，是否漏掉末尾，是否把相等元素放到正确一边。

做这类题时，不能只看“这里像排序”。要问清楚：排序范围是什么，比较方向是什么，合并时两个指针各自代表什么，临时数组什么时候写回原数组。

排序范围

C++ 中 `sort` 的范围是左闭右开区间 `[first,last)`。也就是说：

```
sort(a + l, a + r);
```

排序的是 `a[l]` 到 `a[r-1]`，不包括 `a[r]`。

若数组从 1 到 n 有效，要排序整个数组，应写：

```
sort(a + 1, a + n + 1);
```

若数组从 0 到 $n-1$ 有效，应写：

```
sort(a, a + n);
```

完善程序题常在这里设空：

```
sort(a + 1, a + ①);
```

若要排序 `a[1..n]`，① 应填 `n + 1`。

再看结构体数组：

```
sort(p + 1, p + n + 1, cmp);
```

同样排序 `p[1]` 到 `p[n]`。

填排序范围时，先看读入位置：

```
for (int i = 1; i <= n; i++) {
    cin >> a[i];
}
```

读入从 1 开始，排序范围通常也从 `a+1` 开始。若读入是：

```
for (int i = 0; i < n; i++) {
    cin >> a[i];
}
```

排序范围通常是 `a` 到 `a+n`。

排序范围错一位，后续去重、二分和合并都会跟着错。

比较函数

排序方向由比较函数决定。

```
bool cmp(int x, int y) {
    return x < y;
}
```

表示升序。

```
bool cmp(int x, int y) {
    return x > y;
}
```

表示降序。

结构体排序常见写法：

```
struct Node {
    int x, y;
};

bool cmp(Node a, Node b) {
    if (a.x != b.x) {
        return a.x < b.x;
    }

    return a.y > b.y;
}
```

含义是：先按 x 升序；若 x 相同，再按 y 降序。

例如有四个元素：

元素	x	y
A	2	5
B	1	7
C	2	9
D	1	3

排序后顺序为：

```
B, D, C, A
```

原因是：

- x=1 的元素在 x=2 前面；
- 在 x=1 中，y=7 的 B 在 y=3 的 D 前面；
- 在 x=2 中，y=9 的 C 在 y=5 的 A 前面。

完善程序中，如果空缺在比较函数：

```
if (a.x != b.x) {
    return ①;
```

```
}
return ②;
```

就要回到题意：第一关键字是从小到大还是从大到小？第二关键字在相等时怎样处理？

不要只根据函数名 `cmp` 判断，函数名并不保证排序方向。

相等时放哪边

排序和归并中，相等元素的处理很重要。

看归并两个有序段：

```
while (i <= mid && j <= r) {
    if (a[i] <= a[j]) {
        tmp[++cnt] = a[i++];
    } else {
        tmp[++cnt] = a[j++];
    }
}
```

这里当 `a[i] == a[j]` 时，先取左段元素。这样可以保持原来相等元素的相对顺序。

如果写成：

```
if (a[i] < a[j])
```

相等时会先取右段元素。排序结果仍然是非降序，但相等元素的原相对顺序可能改变。

在只要求排序结果有序时，`<` 和 `<=` 的差别可能不影响最终数值序列；但在计算逆序对、稳定排序、带编号结构体排序时，这个差别会影响答案。

例如归并排序统计逆序对时：

```
if (a[i] <= a[j]) {
    tmp[++cnt] = a[i++];
} else {
    ans += mid - i + 1;
    tmp[++cnt] = a[j++];
}
```

当 `a[i] == a[j]` 时，不应计入逆序对，因为逆序对要求前面的数严格大于后面的数。所以条件必须把相等归到左边，即使用 `<=`。

若改成 `a[i] < a[j]`，相等时进入 `else`，会把相等元素错误计为逆序对。

相等情况要根据题意判断：

任务	相等时常见处理
普通升序排序	两边都可能，结果数值有序即可
稳定归并	先取左段
统计逆序对	相等不算逆序，先取左段
统计严格小于	相等不能计入
统计小于等于	相等要计入

完善程序中看到 `<`、`<=`、`>`、`>=` 的选项时，要重点判断相等情况。

归并两个有序数组

归并程序的基本结构如下：

```
int i = 1, j = 1, k = 0;

while (i <= n && j <= m) {
    if (a[i] <= b[j]) {
        c[++k] = a[i++];
    } else {
        c[++k] = b[j++];
    }
}

while (i <= n) {
    c[++k] = a[i++];
}

while (j <= m) {
    c[++k] = b[j++];
}
```

变量含义：

变量	含义
i	当前还未合并的 a 中位置
j	当前还未合并的 b 中位置
k	已写入 c 的元素个数
c	合并后的数组

例如：

```
a = 1 4 6
b = 2 3 5 7
```

过程：

比较	写入
1 与 2	1
4 与 2	2
4 与 3	3
4 与 5	4
6 与 5	5
6 与 7	6
a 用完	继续写入 7

最终：

```
1 2 3 4 5 6 7
```

归并程序填空常见位置：

```
while (i <= n && j <= m)
```

表示两个数组都还有元素时才比较。

后面必须有两个收尾循环，把剩下的元素复制过去。若漏掉其中一个数组的剩余部分，结果会缺元素。

例如：

```
while (i <= n) c[++k] = a[i++];
while (j <= m) c[++k] = b[j++];
```

这两段不能省略。

归并排序的边界

归并排序一般写成闭区间 `[l,r]`：

```
void merge_sort(int l, int r) {
    if (l == r) {
        return;
    }

    int mid = (l + r) / 2;

    merge_sort(l, mid);
    merge_sort(mid + 1, r);

    merge(l, mid, r);
}
```

边界关系：

部分	区间
左半段	`[l,mid]`
右半段	`[mid+1,r]`
停止条件	`l==r`

若使用左闭右开区间 `[l,r)`：

```
void merge_sort(int l, int r) {
    if (l + 1 == r) {
        return;
    }

    int mid = (l + r) / 2;

    merge_sort(l, mid);
    merge_sort(mid, r);

    merge(l, mid, r);
}
```

边界关系：

部分	区间
左半段	`[l,mid)`
右半段	`[mid,r)`
停止条件	`l+1==r`

完善程序中，递归边界要和合并函数匹配。

若合并函数写成：

```
int i = l, j = mid + 1;

while (i <= mid && j <= r) {
    ...
}
```

说明左段是 [l,mid]，右段是 [mid+1,r]，整个区间是 [l,r]。

若合并函数写成：

```
int i = l, j = mid;

while (i < mid && j < r) {
    ...
}
```

说明左段是 [l,mid)，右段是 [mid,r)，整个区间是 [l,r)。

不能只看递归函数名，要看合并阶段如何使用 mid。

临时数组写回

归并排序合并后，要把临时数组写回原数组。

闭区间写法常见如下：

```
int cnt = 0;

while (i <= mid && j <= r) {
    ...
}

while (i <= mid) {
    tmp[++cnt] = a[i++];
}

while (j <= r) {
    tmp[++cnt] = a[j++];
}

for (int i = 1; i <= cnt; i++) {
    a[l + i - 1] = tmp[i];
}
```

为什么写回位置是 a[l+i-1]？

因为 tmp[1] 对应原区间的第一个位置 a[l]，tmp[2] 对应 a[l+1]。所以：

tmp 位置	写回位置
1	a[l]
2	a[l+1]
i	a[l+i-1]

如果写成 a[i]=tmp[i]，只有当 l=1 时才正确。递归处理中，很多区间的左端不是 1，会写错位置。

也可以写成：

```
for (int i = l; i <= r; i++) {
    a[i] = tmp[i];
}
```

但这种写法要求 tmp 的下标也按原数组位置存储。比如合并时写：

```
tmp[k++] = ...
```

还是：

```
tmp[pos++] = ...
```

要保持一致。

填写回语句时，要看临时数组的下标从哪里开始。如果 tmp 从 1 开始存，写回常用 a[l+i-1]；如果 tmp 直接从 l 开始存，写回常用 a[i]=tmp[i]。

逆序对

归并排序常用于统计逆序对。逆序对是满足 $i < j$ 且 $a_i > a_j$ 的二元组。

归并时，左段和右段都已经有序。若当前：

```
a[i] > a[j]
```

则左段中从 i 到 mid 的所有元素都大于 a[j]，因为左段有序。因此新增逆序对数量为：

```
mid - i + 1
```

典型代码：

```
long long ans = 0;

void merge_sort(int l, int r) {
    if (l == r) {
        return;
    }

    int mid = (l + r) / 2;

    merge_sort(l, mid);
    merge_sort(mid + 1, r);

    int i = l, j = mid + 1, cnt = 0;

    while (i <= mid && j <= r) {
        if (a[i] <= a[j]) {
            tmp[++cnt] = a[i++];
        } else {
            ans += mid - i + 1;
            tmp[++cnt] = a[j++];
        }
    }

    while (i <= mid) {
        tmp[++cnt] = a[i++];
    }
```

```
    while (j <= r) {
        tmp[++cnt] = a[j++];
    }

    for (int k = 1; k <= cnt; k++) {
        a[l + k - 1] = tmp[k];
    }
}
```

空缺可能出现在：

```
ans += ①;
```

应填：

```
mid - i + 1
```

不是 `j-mid`，也不是 `r-j+1`。新增逆序对来自左段剩下的元素与当前右段元素 `a[j]`。

举例：

左段：

```
2 5 8
```

右段：

```
1 6
```

当比较 2 和 1 时，`2,5,8` 都大于 1，新增 3 个逆序对，即 `mid-i+1`。

当比较 8 和 6 时，左段剩下只有 8，新增 1 个逆序对。

注意使用 `a[i] <= a[j]`。相等不算逆序，不能进入统计分支。

第 k 小归并

两个有序数组归并后求第 k 小，可以直接模拟归并到第 k 个：

```
int kth(int a[], int n, int b[], int m, int k) {
    int i = 1, j = 1, cnt = 0;

    while (i <= n && j <= m) {
        int x;

        if (a[i] <= b[j]) {
            x = a[i++];
        } else {
            x = b[j++];
        }

        cnt++;

        if (cnt == k) {
            return x;
        }
```

```
    }

    while (i <= n) {
        cnt++;

        if (cnt == k) {
            return a[i];
        }

        i++;
    }

    while (j <= m) {
        cnt++;

        if (cnt == k) {
            return b[j];
        }

        j++;
    }

    return -1;
}
```

空缺常见在：

```
if (cnt == ①) return x;
```

这里应填 k，因为第 k 次取出的元素就是合并后第 k 小值。

若程序采用 0 下标计数，有时会写：

```
if (cnt == k - 1)
```

所以要看 cnt 初值和每次增加的位置。

直接归并求第 k 小的时间是 $O(k)$。若 k 可能很大，题目可能使用上一章讲过的二分或更复杂方法。本章关注的是归并流程中的边界和取数顺序。

真题式例题：归并第 k 小

下面程序给出两个长度均为 n 的递增数组 a1 和 a2，求合并后的第 k 小。代码使用边界缩小的方法。

```
int solve(int* a1, int* a2, int n, int k) {
    int left1 = 0, right1 = n - 1;
    int left2 = 0, right2 = n - 1;

    while (left1 <= right1 && left2 <= right2) {
        int m1 = (left1 + right1) >> 1;
        int m2 = (left2 + right2) >> 1;

        int cnt = ①;

        if (②) {
            if (cnt < k) {
                left1 = m1 + 1;
            } else {
```

```
                right2 = m2 - 1;
            }
        } else {
            if (cnt < k) {
                left2 = m2 + 1;
            } else {
                right1 = m1 - 1;
            }
        }
    }

    if (③) {
        if (left1 == 0) {
            return a2[k - 1];
        } else {
            int x = a1[left1 - 1], y = ④;
            return max(x, y);
        }
    } else {
        if (left2 == 0) {
            return a1[k - 1];
        } else {
            int x = a2[left2 - 1], y = ⑤;
            return max(x, y);
        }
    }
}
```

这道题已经在二分与查找中出现过，这里从“归并边界”的角度再看一次。

数组下标从 0 开始。`a1[0..m1]` 中有 `m1+1` 个元素，`a2[0..m2]` 中有 `m2+1` 个元素，所以① 应填：

```
(m1 + 1) + (m2 + 1)
```

若 `a1[m1] <= a2[m2]`，说明 `a1[m1]` 是两个中点中较小的那个。若此时 `cnt < k`，前面这些元素还不够，需要排除 `a1` 的前半部分，执行 `left1=m1+1`。所以② 应填：

```
a1[m1] <= a2[m2]
```

循环退出时，可能是第一个数组候选区间空了，也可能是第二个数组候选区间空了。③ 用来判断第一个数组是否空，应填：

```
left1 > right1
```

若第一个数组的候选区间空了，边界落在 `left1`。已经有 `left1` 个来自 `a1` 的元素在答案左侧附近，所以需要从 `a2` 中找下标 `k-left1-1` 的元素参与比较。④ 应填：

```
a2[k - left1 - 1]
```

同理，⑤ 应填：

```
a1[k - left2 - 1]
```

这道题最容易错在①，把数量写成 `m1+m2`。下标从 0 开始时，前缀长度是下标加 1，这是归并类边界题的核心。

真题式例题：最大值之和的分治合并

给定序列，要求所有非空连续子序列的最大值之和。下面程序使用分治。这里不需要重新推导完整算法，只看合并部分的空缺。

```
void solve(int l, int r) {
    if (l + 1 == r) {
        ans += a[l];
        return;
    }

    int mid = (l + r) >> 1;

    vector<int> pre(a + mid, a + r);

    for (int i = 1; i < r - mid; i++) {
        ①;
    }

    vector<long long> sum(r - mid + 1);

    for (int i = 0; i < r - mid; i++) {
        sum[i + 1] = sum[i] + pre[i];
    }

    for (int i = mid - 1, j = mid, mx = 0; i >= l; i--) {
        while (j < r && ②) {
            j++;
        }

        mx = max(mx, a[i]);

        ans += ③;
        ans += ④;
    }

    solve(l, mid);
    solve(mid, r);
}
```

先看区间。递归调用是：

```
solve(l, mid);
solve(mid, r);
```

停止条件是 `l+1==r`，说明这里使用左闭右开区间 `[l,r)`。

`pre` 存的是右半段从 `mid` 开始的前缀最大值。初始：

```
pre[0] = a[mid]
pre[1] = a[mid+1]
...
```

经过处理后，应使：

```
pre[i] = max(a[mid], a[mid+1], ..., a[mid+i])
```

所以① 应填：

```
pre[i] = max(pre[i - 1], pre[i])
```

也可以理解为每个位置保存右半段前缀最大值。

在循环中，i 从左半段往左走，mx 表示左侧部分的最大值。注意程序里 while 在 mx=max(mx,a[i]) 之前，因此条件② 使用的是上一轮已经维护好的 mx。j 向右移动，直到右侧前缀最大值不再小于左侧最大值。也就是说，当：

```
pre[j - mid] < mx
```

时，跨中点区间的最大值由左侧 mx 决定，应把 j 继续右移。因此② 应填：

```
pre[j - mid] < mx
```

对于右侧前缀最大值小于 mx 的那些右端点，贡献是左侧最大值 mx。数量是 j-mid，所以③ 应填：

```
(long long)(j - mid) * mx
```

对于剩下的右端点，贡献由右侧前缀最大值决定，已经用 sum 做了前缀和。右侧剩余部分贡献为：

```
sum[r - mid] - sum[j - mid]
```

所以④ 应填：

```
sum[r - mid] - sum[j - mid]
```

这类题的合并部分较复杂，但空缺依然围绕三件事：

- 右半段前缀最大值怎样维护；
- 指针 j 把右半段分成哪两类；
- 两类贡献分别怎样计算。

完善程序中遇到分治合并，不要急着重写算法。先给每个辅助数组和指针写含义，再填空。

分治合并的指针

很多分治合并代码都会用两个指针或多个指针。填空时要看指针移动后的区间含义。

例如：

```
int j = mid;

for (int i = mid - 1; i >= l; i--) {
    while (j < r && pre[j - mid] < mx) {
        j++;
    }

    ans += (long long)(j - mid) * mx;
    ans += sum[r - mid] - sum[j - mid];
}
```

这里右半段 [mid,r) 被分成两段：

右端点范围	数量	最大值来源
[mid,j)	j-mid	左侧 mx
[j,r)	r-j	右侧 pre

sum 的下标也要跟着这个划分。因为 pre[0] 对应右端点 mid，所以：

- [mid,j) 对应 pre[0..j-mid-1]；
- [j,r) 对应 pre[j-mid..r-mid-1]。

因此剩余贡献是 sum[r-mid]-sum[j-mid]。

指针移动题的检查方法是：写出移动后左右两段分别是什么，不要只看表达式是否长得像。

排序后去重

排序后去重常见代码：

```
sort(a + 1, a + n + 1);

int m = 0;

for (int i = 1; i <= n; i++) {
    if (i == 1 || a[i] != a[i - 1]) {
        b[++m] = a[i];
    }
}
```

或者：

```
sort(a + 1, a + n + 1);
int m = unique(a + 1, a + n + 1) - (a + 1);
```

若空缺在 unique 的减法部分：

```
int m = unique(a + 1, a + n + 1) - ①;
```

如果希望 m 是不同元素个数，应填：

```
(a + 1)
```

因为有效区间从 a+1 开始。

如果后续要使用去重后的数组 a[1..m]，要注意 unique 后有效值仍从 a[1] 开始。

例如：

```
a = 4 1 4 2 2
```

排序后：

```
1 2 2 4 4
```

去重后有效部分：

```
1 2 4
```

m=3。

若后续写：

```
for (int i = 1; i <= m; i++) {
    ...
}
```

是正确的。

若误把 m 算成 unique(...) - a，则 m 会多 1，后续可能访问无效位置。

稳定排序相关空缺

虽然 C++ 的 sort 不保证稳定，但手写归并排序可以保持稳定。稳定的意思是：若两个元素关键字相等，排序后它们的相对顺序不变。

例如学生信息：

编号	分数
1	90
2	80
3	90

若按分数升序稳定排序，结果为：

```
2, 1, 3
```

两个 90 分的编号 1 和 3 仍保持原顺序。

归并时要做到稳定，相等时取左段元素：

```
if (a[i].score <= a[j].score) {
    tmp[++cnt] = a[i++];
} else {
    tmp[++cnt] = a[j++];
}
```

如果空缺是比较条件：

```
if (①) {
    tmp[++cnt] = a[i++];
} else {
    tmp[++cnt] = a[j++];
}
```

并且题目要求稳定升序排序，① 应填：

```
a[i].score <= a[j].score
```

若题目只要求按分数升序，且不关心相等元素的相对顺序，< 也能得到有序结果。但完善程序一般有唯一正确选项，需要结合后续是否用到稳定性。

桶与计数排序中的边界

排序类填空不只包括比较排序，也可能包括计数排序。

```
int cnt[1005];

for (int i = 1; i <= n; i++) {
    cnt[a[i]]++;
}

for (int x = 0; x <= 1000; x++) {
    while (cnt[x] > 0) {
        cout << x << ' ';
        cnt[x]--;
    }
}
```

这段程序按值从小到大输出。

如果题目中 `a[i]` 的范围是 1 到 1000，输出循环可以从 1 开始；如果范围包括 0，必须从 0 开始。

计数排序的空缺常在：

```
for (int x = ①; x <= ②; x++)
```

应根据值域填写，而不是根据元素个数 `n` 填写。

例如 `n` 是数的个数，`a[i]` 的值在 0 到 100，那么循环应遍历值 0 到 100，不是 1 到 n。

若要降序输出，循环方向应变成：

```
for (int x = maxv; x >= minv; x--)
```

计数排序题要分清“下标表示值”，不是表示第几个元素。

变式题：归并排序补空

补全下面归并排序程序，使数组 `a[1..r]` 按非降序排列。

```
void sort_arr(int l, int r) {
    if (①) {
        return;
    }

    int mid = (l + r) / 2;

    sort_arr(l, mid);
    sort_arr(②, r);

    int i = l, j = mid + 1, cnt = 0;

    while (i <= mid && j <= r) {
        if (a[i] <= a[j]) {
            tmp[++cnt] = a[i++];
        } else {
            tmp[++cnt] = a[j++];
        }
    }

    while (i <= mid) {
        tmp[++cnt] = a[i++];
    }
```

```
    while (j <= r) {
        tmp[++cnt] = a[j++];
    }

    for (int k = 1; k <= cnt; k++) {
        a[③] = tmp[k];
    }
}
```

这是闭区间 `[l,r]`。单元素时停止，所以① 应填：

```
l == r
```

右半段从 `mid+1` 开始，所以② 应填：

```
mid + 1
```

`tmp[1]` 对应 `a[l]`，所以③ 应填：

```
l + k - 1
```

变式题：逆序对补空

补全下面代码，统计逆序对数量。

```
while (i <= mid && j <= r) {
    if (a[i] <= a[j]) {
        tmp[++cnt] = a[i++];
    } else {
        ans += ①;
        tmp[++cnt] = a[j++];
    }
}
```

当 `a[i] > a[j]` 时，左段中从 `i` 到 `mid` 的元素都大于 `a[j]`，新增逆序对数量为：

```
mid - i + 1
```

所以① 应填：

```
mid - i + 1
```

若填 `j-mid`，那表示右段已经取出的数量，与当前右段元素能构成多少逆序对无关。

变式题：合并两个有序数组

补全代码，将递增数组 `a[1..n]` 和 `b[1..m]` 合并到 `c` 中。

```
int i = 1, j = 1, k = 0;

while (①) {
    if (a[i] <= b[j]) {
        c[++k] = a[i++];
    } else {
```

```
        c[++k] = b[j++];
    }
}

while (i <= n) {
    c[++k] = a[i++];
}

while (②) {
    c[++k] = b[j++];
}
```

主循环要在两个数组都还有元素时执行比较，所以① 应填：

```
i <= n && j <= m
```

第二个收尾循环复制 b 中剩下的元素，所以② 应填：

```
j <= m
```

变式题：去重后的长度

补全代码，求数组中不同元素个数。数组使用 1 到 n 号位置。

```
sort(a + 1, a + n + 1);
int m = unique(a + 1, a + n + 1) - ①;
cout << m << endl;
```

unique 返回去重后有效区间的后一位。有效区间从 a+1 开始，所以不同元素个数是：

```
unique(...) - (a + 1)
```

① 应填：

```
(a + 1)
```

若填 a，会多算 1。

变式题：结构体排序

补全比较函数。要求按分数 score 从高到低排序，分数相同按编号 id 从小到大排序。

```
struct Node {
    int id, score;
};

bool cmp(Node a, Node b) {
    if (a.score != b.score) {
        return ①;
    }

    return ②;
}
```

分数从高到低，① 应填：

```
a.score > b.score
```

编号从小到大，② 应填：

```
a.id < b.id
```

不要把第二关键字也写成降序。多关键字排序要逐项对照题意。

变式题：计数排序输出

补全代码，输出值域为 0 到 100 的数组，要求从大到小输出。

```
for (int i = 1; i <= n; i++) {
    cnt[a[i]]++;
}

for (int x = ①; x >= ②; x--) {
    while (cnt[x] > 0) {
        cout << x << ' ';
        cnt[x]--;
    }
}
```

值域最大为 100，最小为 0，从大到小输出，所以① 填 `100`，② 填 `0`。

这里循环的是值 `x`，不是原数组下标。若填 `n`，当 `n` 不等于最大值域时就会错。

小结

做排序与归并填空时，可以按下面顺序检查：

检查点	要问的问题
排序范围	`sort` 的右端是否为后一位
下标起点	数组有效范围从 0 还是 1 开始
比较方向	升序还是降序
第二关键字	第一关键字相等时怎么排
相等情况	相等时放左边还是右边
归并区间	左段和右段分别是什么
主循环条件	是否两个段都还有元素
收尾循环	剩余元素是否全部复制
写回位置	`tmp` 的下标和原数组下标怎样对应
逆序对	新增数量是否为左段剩余长度
去重长度	`unique` 减去的是有效区间起点
计数排序	循环变量表示值域还是元素下标

排序与归并类完善程序的空缺，多数不是难在算法名，而是难在边界一致性。只要把排序范围、归并区间、相等情况和写回位置对齐，代码就能稳住。遇到复杂分治合并时，先给每个指针写清楚它划分出的区间，再填表达式。

0x38 DP 程序填空

DP 程序填空的难点通常不在“知道这是 DP”，而在代码中的某个空缺到底影响了状态含义、初值、转移来源、循环顺序，还是答案位置。很多 DP 程序看起来只有几行，但一个循环方向填错，就会把 $0/1$ 选择写成可重复选择；一个初值填错，就会把不可达状态当成可达状态；一个答案下标填错，就会输出“处理到第 $n-1$ 项”的结果。

做 DP 填空，要先把状态含义写清楚，再补转移。不要先盯着选项猜公式。

先看状态

DP 程序中，最重要的是状态数组的含义。

例如：

```
int f[1005];

f[0] = 1;

for (int i = 1; i <= n; i++) {
    for (int j = i; j <= n; j++) {
        f[j] += f[j - i];
    }
}
```

这里 `f[j]` 很可能表示“凑出和为 j 的方案数”。`f[0]=1` 表示空方案有一种。内层转移 `f[j] += f[j-i]` 表示使用一个数值为 `i` 的元素后，从和 `j-i` 转移到和 `j`。

再看：

```
int f[1005];

for (int j = 1; j <= m; j++) {
    f[j] = INF;
}

f[0] = 0;

for (int i = 1; i <= n; i++) {
    for (int j = m; j >= w[i]; j--) {
        f[j] = min(f[j], f[j - w[i]] + c[i]);
    }
}
```

这里 `f[j]` 更可能表示“凑出体积 j 的最小代价”。`f[0]=0` 是起点，其他状态初始为 `INF` 表示不可达。转移中的 `min` 说明这是最小值型 DP。

判断状态时可以先看三处：

位置	提示
初值	`0`、`1`、`INF`、`-INF` 分别暗示不同含义
转移	`+=` 常用于计数，`max` 常用于最大值，`min` 常用于最小值
答案	输出 `f[n]`、`max f[i]`、`sum f[i]`，表示目标不同

完善程序中的空缺，往往只有在状态含义明确后才能判断。

初值

DP 初值不是为了让数组“有个值”，而是给转移一个起点。

计数型 DP：

```
dp[0] = 1;
```

通常表示空方案有一种。

最小值型 DP：

```
for (int i = 0; i <= n; i++) {
    dp[i] = INF;
}

dp[0] = 0;
```

通常表示只有起点可达，其他状态暂时不可达。

最大值型 DP：

```
for (int i = 0; i <= n; i++) {
    dp[i] = -INF;
}

dp[0] = 0;
```

通常表示只有起点可达，其他状态不应参与最大值比较。

看下面的程序：

```
const int INF = 1e9;
int f[1005];

for (int j = 0; j <= m; j++) {
    f[j] = ①;
}

f[0] = ②;

for (int i = 1; i <= n; i++) {
    for (int j = m; j >= w[i]; j--) {
        f[j] = min(f[j], f[j - w[i]] + c[i]);
    }
}
```

若题目要求“恰好凑出容量 m 的最小代价”，则 `f[j]` 表示恰好凑出 j 的最小代价。开始时只有 0 可达，其他容量不可达。所以① 填 `INF`，② 填 `0`。

如果把所有 `f[j]` 都初始化为 0，程序会把每个容量都当成已经可以免费凑出，答案会偏小。

再看计数：

```
int dp[1005];

dp[①] = ②;

for (int i = 1; i <= n; i++) {
    for (int j = i; j <= n; j++) {
        dp[j] += dp[j - i];
    }
}
```

若 `dp[j]` 表示凑出和为 j 的方案数，起点应是 `dp[0]=1`。所以① 填 `0`，② 填 `1`。

计数 DP 中的 `1` 不是说数值 0 有一个元素，而是说 “不选任何东西” 是一种起始方案。

循环方向

一维背包 DP 中，循环方向决定物品是否可以重复使用。

0/1 背包：每个物品最多使用一次。

```
for (int i = 1; i <= n; i++) {
    for (int j = m; j >= w[i]; j--) {
        f[j] = max(f[j], f[j - w[i]] + v[i]);
    }
}
```

内层 `j` 从大到小，保证 `f[j-w[i]]` 仍是上一轮物品处理后的结果，不会在同一轮重复使用第 `i` 个物品。

完全背包：每个物品可以使用多次。

```
for (int i = 1; i <= n; i++) {
    for (int j = w[i]; j <= m; j++) {
        f[j] = max(f[j], f[j - w[i]] + v[i]);
    }
}
```

内层 `j` 从小到大，`f[j-w[i]]` 可能已经在本轮被当前物品更新过，因此允许继续使用同一个物品。

同一个转移式：

```
f[j] = max(f[j], f[j - w[i]] + v[i]);
```

循环方向不同，含义不同。

看一个小例子：容量 4，只有一个物品，体积 2，价值 3。

若是 0/1 背包，只能选一次，最大价值为 3。

若是完全背包，可以选两次，最大价值为 6。

所以完善程序中，若空缺是：

```
for (int j = ①; ②; ③)
```

要先看题目是否允许重复使用。不能只根据转移式判断。

常见填法：

类型	内层循环
0/1 背包	`for (int j = m; j >= w[i]; j--)`
完全背包	`for (int j = w[i]; j <= m; j++)`
二维 DP	通常按状态依赖顺序从小到大或分层循环

循环方向是 DP 程序填空中的高频考点。

取或不取

很多 DP 转移来自“取或不取”。

0/1 背包二维写法：

```
for (int i = 1; i <= n; i++) {
    for (int j = 0; j <= m; j++) {
        f[i][j] = f[i - 1][j];

        if (j >= w[i]) {
            f[i][j] = max(f[i][j], f[i - 1][j - w[i]] + v[i]);
        }
    }
}
```

状态含义：

状态	含义
`f[i][j]`	只考虑前 i 个物品，容量不超过 j 的最大价值
`f[i-1][j]`	不选第 i 个物品
`f[i-1][j-w[i]]+v[i]`	选第 i 个物品

如果空缺在：

```
f[i][j] = max(f[i - 1][j], ①);
```

则① 应填：

```
f[i - 1][j - w[i]] + v[i]
```

不能填 `f[i][j-w[i]]+v[i]`，因为那会允许第 i 个物品重复选，变成完全背包。

若题目是完全背包，二维转移可以写成：

```
f[i][j] = f[i - 1][j];

if (j >= w[i]) {
    f[i][j] = max(f[i][j], f[i][j - w[i]] + v[i]);
}
```

这里选第 `i` 个物品后，仍然可以继续使用第 `i` 个物品，所以来源是 `f[i][j-w[i]]`。

判断 `i-1` 还是 `i`，关键看当前物品能不能重复使用。

计数与最值

同样的状态，转移方式可能不同。

计数型：

```
dp[j] += dp[j - w[i]];
```

最小值型：

```
dp[j] = min(dp[j], dp[j - w[i]] + c[i]);
```

最大值型：

```
dp[j] = max(dp[j], dp[j - w[i]] + v[i]);
```

判断空缺时，要看题目要求“多少种方案”“最少代价”还是“最大价值”。

例如：

```
for (int i = 1; i <= n; i++) {
    for (int j = m; j >= w[i]; j--) {
        dp[j] = ①;
    }
}
```

若题目要求最大价值，① 应填：

```
max(dp[j], dp[j - w[i]] + v[i])
```

若题目要求最小代价，① 应填：

```
min(dp[j], dp[j - w[i]] + c[i])
```

若题目要求方案数，① 应填：

```
dp[j] + dp[j - w[i]]
```

或者在代码中写成：

```
dp[j] += dp[j - w[i]];
```

有时计数还需要取模：

```
dp[j] = (dp[j] + dp[j - w[i]]) % MOD;
```

如果题目给定模数，填空时要检查转移中是否需要 `% MOD`。方案数题若不取模，可能超出整数范围；但如果题目没有模数，不能凭习惯加模。

二维状态

二维 DP 程序先看两个维度分别表示什么。

例如最长公共子序列：

```
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;
        } else {
            f[i][j] = max(f[i - 1][j], f[i][j - 1]);
        }
    }
}
```

状态含义：

状态	含义
f[i][j]	a[1..i] 和 b[1..j] 的最长公共子序列长度

若最后两个字符相同，就可以把它们配对，来源是 f[i-1][j-1]+1。

若最后两个字符不同，就不能同时取这两个末尾字符，只能删掉其中一个末尾，取较大值：

```
max(f[i - 1][j], f[i][j - 1])
```

若空缺是：

```
if (a[i] == b[j]) {
    f[i][j] = ①;
} else {
    f[i][j] = ②;
}
```

则① 填 f[i - 1][j - 1] + 1，② 填 max(f[i - 1][j], f[i][j - 1])。

这里的边界通常是 f[0][j]=0、f[i][0]=0，因为空串与任何串的 LCS 长度都是 0。如果数组是全局变量，默认初值为 0，代码中可能不显式初始化。

二维 DP 填空时，要问：

- 第一维表示处理到哪里；
- 第二维表示什么限制或另一段序列；
- 当前状态依赖上方、左方、左上方，还是上一层。

子序列 DP

子序列 DP 常用 “选或不选当前字符” 的思路。

看下面程序，统计字符串 s 的所有非空子序列中，按二进制解释后的数值之和，只保留长度不超过 m 的低位状态。这里不要求掌握完整背景，只看状态转移。

```
const int P = 998244353;
int n, m;
string s;
int dp[1 << M];

int solve() {
    dp[0] = 1;

    for (int i = 0; i < n; i++) {
        for (int j = (1 << (m - 1)) - 1; j >= 0; j--) {
            int k = ①;

            if (j != 0 || s[i] == '1') {
                dp[k] = ②;
```

```
            }
        }
    }

    int ans = 0;

    for (int i = 0; i < (1 << m); i++) {
        ans = (ans + 1ll * i * dp[i]) % P;
    }

    return ans;
}
```

内层状态 `j` 表示当前已经形成的二进制数的低位状态。读到新字符 `s[i]` 后，把原状态左移一位，再加上当前字符：

```
k = (j << 1) | (s[i] - '0')
```

所以① 应填：

```
(j << 1) | (s[i] - '0')
```

更新是把原状态 `j` 的方案数加到新状态 `k`：

```
dp[k] = (dp[k] + dp[j]) % P
```

所以② 应填：

```
(dp[k] + dp[j]) % P
```

为什么内层 `j` 从大到小？因为每个字符只能被本轮选一次。若从小到大更新，当前字符刚产生的新状态可能在同一轮继续被用，含义就会改变。

这类 DP 程序看起来像位运算，但填空点仍然是“旧状态怎样变成新状态”“方案数怎样累加”。

区间 DP

区间 DP 的状态通常形如 `f[l][r]`，表示区间 `[l,r]` 的答案。

例如合并石子：

```
for (int len = 2; len <= n; len++) {
    for (int l = 1; l + len - 1 <= n; l++) {
        int r = l + len - 1;
        f[l][r] = INF;

        for (int k = l; k < r; k++) {
            f[l][r] = min(f[l][r], f[l][k] + f[k + 1][r] + sum[r] - sum[l - 1]);
        }
    }
}
```

状态含义：

状态	含义
f[l][r]	合并区间 [l,r] 的最小代价
k	最后一次合并时的分界点
sum[r]-sum[l-1]	区间 [l,r] 的总重量

区间 DP 的循环顺序通常按长度从小到大。因为计算 f[l][r] 时，需要较短区间 f[l][k] 和 f[k+1][r] 已经算好。

若空缺是：

```
for (int len = ①; len <= n; len++)
```

因为长度为 1 的区间代价通常为 0，真正需要转移的是长度至少为 2 的区间，所以① 应填 2。

若空缺是：

```
for (int k = l; ②; k++)
```

分裂为 [l,k] 和 [k+1,r]，要求 k<r，所以② 应填 k < r。

若空缺是区间和：

```
f[l][r] = min(f[l][r], f[l][k] + f[k + 1][r] + ③);
```

则③ 应填：

```
sum[r] - sum[l - 1]
```

区间 DP 填空要抓住三个边界：长度从几开始，左端枚举到哪里，分界点能取到哪里。

滚动数组

滚动数组会把二维 DP 压成一维或两行，填空时要分清当前层和上一层。

二维 0/1 背包：

```
f[i][j] = max(f[i - 1][j], f[i - 1][j - w[i]] + v[i]);
```

压成一维后：

```
for (int i = 1; i <= n; i++) {
    for (int j = m; j >= w[i]; j--) {
        f[j] = max(f[j], f[j - w[i]] + v[i]);
    }
}
```

从大到小枚举 j，是为了让 f[j-w[i]] 仍对应上一层。

二维 LCS 若用滚动数组：

```
for (int i = 1; i <= n; i++) {
    int cur = i & 1;
    int pre = cur ^ 1;

    for (int j = 1; j <= m; j++) {
        if (a[i] == b[j]) {
            f[cur][j] = f[pre][j - 1] + 1;
```

```
        } else {
            f[cur][j] = max(f[pre][j], f[cur][j - 1]);
        }
    }
}
```

这里：

数组	含义
f[pre][j]	上一行同列
f[cur][j-1]	当前行左边
f[pre][j-1]	上一行左上

如果空缺是：

```
int pre = ①;
```

常填：

```
cur ^ 1
```

因为两行交替使用。

滚动数组填空还要注意清空。若当前层某些位置不会被完整覆盖，就需要在每一层开始时清零或设为初值。若每个 f[cur][j] 都会被赋值覆盖，则不一定需要清空。

答案位置

DP 程序最后输出哪个状态，需要根据题意判断。

最长公共子序列：

```
cout << f[n][m] << endl;
```

因为状态表示两个前缀的答案。

背包最大价值：

```
cout << f[m] << endl;
```

若 f[j] 表示容量不超过 j 的最大价值，答案是 f[m]。

若 f[j] 表示恰好装满 j 的最大价值，但题目要求容量不超过 m，答案可能是：

```
max(f[0], f[1], ..., f[m])
```

这要看状态定义。

最长上升子序列的一个写法：

```
for (int i = 1; i <= n; i++) {
    f[i] = 1;

    for (int j = 1; j < i; j++) {
        if (a[j] < a[i]) {
            f[i] = max(f[i], f[j] + 1);
```

```
        }
    }
}
```

这里 `f[i]` 表示“以第 i 个元素结尾的 LIS 长度”。答案不是 `f[n]`，而是所有 `f[i]` 的最大值：

```
int ans = 0;

for (int i = 1; i <= n; i++) {
    ans = max(ans, f[i]);
}
```

如果题目空缺在答案处：

```
cout << ① << endl;
```

要先判断状态是否“必须以某位置结尾”。若必须以 `i` 结尾，最终答案通常要在所有结尾中取最优；若状态已经表示前缀整体答案，才直接输出最后一个状态。

真题式例题：子序列状态填空

阅读下面程序，补全空缺。

```
const int P = 998244353;
int n, m;
string s;
int dp[1 << 20];

int solve() {
    dp[0] = 1;

    for (int i = 0; i < n; i++) {
        for (int j = (1 << (m - 1)) - 1; j >= 0; j--) {
            int k = ①;

            if (j != 0 || s[i] == '1') {
                dp[k] = ②;
            }
        }
    }

    int ans = 0;

    for (int i = 0; i < (1 << m); i++) {
        ans = (ans + ③) % P;
    }

    return ans;
}
```

先读状态更新。

旧状态是 `j`，新读入一个字符 `s[i]`，二进制数要左移一位，再把当前位放到最低位：

```
k = (j << 1) | (s[i] - '0')
```

所以① 填：

```
(j << 1) | (s[i] - '0')
```

若当前状态合法，就把 dp[j] 的方案数转移到 dp[k]：

```
dp[k] = (dp[k] + dp[j]) % P
```

所以② 填：

```
(dp[k] + dp[j]) % P
```

最后 dp[i] 表示形成状态值 i 的方案数，总贡献应为 i * dp[i]，注意转成较大整数再取模：

```
1ll * i * dp[i]
```

所以③ 填：

```
1ll * i * dp[i]
```

这道题的空缺分别对应“状态变化”“方案累加”“贡献统计”。做题时不要被位运算吓住，先把 j 到 k 的状态关系说清楚。

真题式例题：最大子段和 DP

补全下面程序，求最大非负连续子段和。

```
int n, a[1005];

int solve() {
    int cur = 0;
    int ans = 0;

    for (int i = 1; i <= n; i++) {
        cur = ①;
        ans = ②;
    }

    return ans;
}
```

这里 cur 表示“以当前位置结尾的最大非负连续子段和”。读入 a[i] 后，若接上前面能变得更好，就接上；若接上后变成负数，就从空段重新开始。所以：

```
cur = max(0, cur + a[i])
```

① 应填：

```
max(0, cur + a[i])
```

ans 表示到目前为止出现过的最大值，所以② 应填：

```
max(ans, cur)
```

若题目要求非空连续子段最大和，且数组可能全为负数，就不能把 cur 和 ans 初始化为 0。那时通常需要用 a[1] 初始化，或把初值设为很小的数。本题若写成“最大非负连续子段和”，则允许空段，初始 0 合理。

完善程序中要留意题目是否允许空方案、空段、空序列。它会影响初值和转移。

真题式例题：魔法数字

有一个数字 4，允许用若干个 4 进行加法、减法和整除运算，求得到给定正整数 n 至少需要多少个 4。程序中 F[x] 表示得到数字 x 至少需要几个 4，Vis[x] 表示该数字是否已经作为当前最优状态处理过。

```
const int M = 10000;
bool Vis[M + 1];
int F[M + 1];

void update(int& x, int y) {
    if (y < x) {
        x = y;
    }
}

int main() {
    int n;
    cin >> n;

    for (int i = 0; i <= M; i++) {
        F[i] = INT_MAX;
    }

    ①;

    int r = 0;

    while (②) {
        r++;

        int x = 0;

        for (int i = 1; i <= M; i++) {
            if (③) {
                x = i;
            }
        }

        Vis[x] = true;

        for (int i = 1; i <= M; i++) {
            if (④) {
                int t = F[i] + F[x];

                if (i + x <= M) {
                    update(F[i + x], t);
                }

                if (i != x) {
                    update(F[abs(i - x)], t);
                }

                if (i % x == 0) {
                    update(F[i / x], t);
                }

                if (x % i == 0) {
                    update(F[x / i], t);
```

```
                }
            }
        }
    }

    cout << F[n] << endl;
}
```

先看状态含义。`F[x]` 是得到 `x` 所需的最少 4 的个数。数字 4 本身可以用一个 4 得到，所以① 应填：

```
F[4] = 1
```

循环应持续到目标 `n` 被处理出来。`Vis[n]` 表示 `n` 的最优值已经确定，所以② 应填：

```
!Vis[n]
```

接着选择一个还没有处理、且当前 `F` 最小的数字 `x`。这里 `r` 每轮递增，程序要找当前 `F[i]==r` 且未访问的 `i`：

```
!Vis[i] && F[i] == r
```

所以③ 应填：

```
!Vis[i] && F[i] == r
```

最后枚举已经处理过的数字 `i`，与新确定的 `x` 做运算。只有 `Vis[i]` 为真时，`F[i]` 才已经确定，可以拿来合并：

```
Vis[i]
```

所以④ 应填：

```
Vis[i]
```

这道题不像常见表格 DP，但状态含义仍然是“最少使用数量”。空缺分别对应初始状态、终止条件、取出当前层状态、枚举已确定状态。完善程序题中，只要变量含义能讲清，形式特殊一些也能拆开判断。

变式题：0/1 背包填空

补全程序，求容量为 `m` 时的最大价值，每个物品最多选一次。

```
int f[1005];

for (int i = 1; i <= n; i++) {
    for (int j = ①; ②; ③) {
        f[j] = max(f[j], ④);
    }
}

cout << f[m] << endl;
```

每个物品最多选一次，所以内层容量要从大到小枚举。起点是 `m`，终点是 `w[i]`。

① 填：

```
m
```

② 填：

```
j >= w[i]
```

③ 填：

```
j--
```

选第 `i` 个物品时，来源为上一层含义的 `f[j-w[i]]`，一维倒序保证这一点。④ 填：

```
f[j - w[i]] + v[i]
```

完整转移：

```
for (int j = m; j >= w[i]; j--) {
    f[j] = max(f[j], f[j - w[i]] + v[i]);
}
```

变式题：完全背包填空

补全程序，求容量为 `m` 时的最大价值，每个物品可以选任意多次。

```
int f[1005];

for (int i = 1; i <= n; i++) {
    for (int j = ①; ②; ③) {
        f[j] = max(f[j], f[j - w[i]] + v[i]);
    }
}
```

每个物品可以重复选，所以内层容量从小到大枚举。当前物品本轮更新出的状态可以继续被使用。

① 填：

```
w[i]
```

② 填：

```
j <= m
```

③ 填：

```
j++
```

完整为：

```
for (int j = w[i]; j <= m; j++) {
    f[j] = max(f[j], f[j - w[i]] + v[i]);
}
```

这题和上一题转移式一样，但循环方向相反。填空时先看“最多一次”还是“任意多次”。

变式题：LCS 填空

补全程序，求两个字符串 `a[1..n]` 和 `b[1..m]` 的最长公共子序列长度。

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= m; j++) {
        if (a[i] == b[j]) {
            f[i][j] = ①;
        } else {
            f[i][j] = ②;
        }
    }
}

cout << ③ << endl;
```

若末尾字符相同，可以匹配这两个字符，来源于两个前缀各少一个字符：

```
f[i - 1][j - 1] + 1
```

所以① 填：

```
f[i - 1][j - 1] + 1
```

若末尾字符不同，只能丢掉其中一个末尾，取较优者：

```
max(f[i - 1][j], f[i][j - 1])
```

所以② 填：

```
max(f[i - 1][j], f[i][j - 1])
```

答案是两个完整字符串的 LCS：

```
f[n][m]
```

所以③ 填：

```
f[n][m]
```

变式题：LIS 填空

补全程序，求最长严格上升子序列长度。

```
for (int i = 1; i <= n; i++) {
    f[i] = ①;

    for (int j = 1; j < i; j++) {
        if (②) {
            f[i] = max(f[i], ③);
        }
    }
}

int ans = 0;
```

```
for (int i = 1; i <= n; i++) {
    ans = max(ans, f[i]);
}

cout << ans << endl;
```

`f[i]` 表示以 `a[i]` 结尾的最长严格上升子序列长度。只选 `a[i]` 自己时，长度为 1，所以① 填：

```
1
```

若要把 `a[i]` 接到以 `a[j]` 结尾的序列后面，需要满足严格上升：

```
a[j] < a[i]
```

所以② 填：

```
a[j] < a[i]
```

接上后长度为 `f[j]+1`，所以③ 填：

```
f[j] + 1
```

答案不能直接输出 `f[n]`，因为最长上升子序列不一定以最后一个元素结尾，需要取所有 `f[i]` 的最大值。

变式题：区间 DP 填空

补全程序，求合并区间 `[1,n]` 的最小代价。`sum[i]` 是前缀和。

```
for (int len = ①; len <= n; len++) {
    for (int l = 1; ②; l++) {
        int r = l + len - 1;

        f[l][r] = INF;

        for (int k = l; ③; k++) {
            f[l][r] = min(f[l][r], ④);
        }
    }
}
```

长度为 1 的区间不需要合并，代价为 0。需要转移的最短长度为 2，所以① 填：

```
2
```

右端为 `r=l+len-1`，要求 `r<=n`，所以② 填：

```
l + len - 1 <= n
```

分界点 `k` 把区间分成 `[l,k]` 和 `[k+1,r]`，所以 `k<r`，③ 填：

```
k < r
```

最后一次合并代价是左右区间代价加上当前区间总和：

```
f[l][k] + f[k + 1][r] + sum[r] - sum[l - 1]
```

所以④ 填：

```
f[l][k] + f[k + 1][r] + sum[r] - sum[l - 1]
```

区间 DP 填空要特别注意 `len`、`l`、`r`、`k` 四个变量之间的关系。

变式题：方案数取模

补全程序，计算方案数并对 `MOD` 取模。

```
dp[0] = 1;

for (int i = 1; i <= n; i++) {
    for (int j = m; j >= w[i]; j--) {
        dp[j] = ①;
    }
}
```

若每个物品最多选一次，且 `dp[j]` 表示凑出容量 `j` 的方案数，那么转移是：

```
dp[j] = (dp[j] + dp[j - w[i]]) % MOD
```

所以① 填：

```
(dp[j] + dp[j - w[i]]) % MOD
```

若题目没有要求取模，不能加 `% MOD`；若题目要求取模，转移中要保持每次更新后都在模数范围内。

小结

做 DP 程序填空时，可以按下面顺序检查：

检查点	要问的问题
状态含义	`f[i]`、`f[i][j]` 分别表示什么
初值	最小规模、空方案、起点是否正确
不可达	应设为 `INF`、`-INF` 还是 `0`
转移类型	计数、最大值还是最小值
取或不取	来源是上一层还是当前层
循环方向	物品能否重复使用
二维依赖	当前状态依赖上方、左方、左上还是上一层
区间 DP	长度、左端、右端、分界点是否匹配
滚动数组	当前层和上一层是否分清
取模	方案数是否要求对模数取余
答案位置	输出最后一个状态，还是所有状态的最优值
空方案	是否允许空段、空序列、空选择

DP 填空的核心不是把公式背下来，而是把状态说清楚。状态含义决定初值，初值决定转移能否开始，转移来源决定循环顺序，答案位置决定最后输出什么。只要按这条线检查，复杂一些的 DP 程序也能拆成几个明确的空缺。

0x39 图论程序填空

图论完善程序的代码通常较长，但空缺位置往往集中在几类地方：边如何加入，入度怎样变化，起点状态如何初始化，松弛条件怎样写，优先队列中放什么，路径计数如何累加，答案从哪个状态输出。

读图论填空，先不要急着补选项。应先确定三件事：图是有向还是无向，数组含义是什么，更新发生在“进入点”“遍历边”还是“弹出队列”时。

先看建图

图论程序首先看建图。

有向图：

```
g[u].push_back(v);
deg[v]++;
```

这表示存在一条从 u 到 v 的边，v 的入度增加 1。

无向图：

```
g[u].push_back(v);
g[v].push_back(u);
```

如果还要记录度数：

```
deg[u]++;
deg[v]++;
```

链式前向星写法：

```
void add(int u, int v, int w) {
    ++tot;
    to[tot] = v;
    val[tot] = w;
    nxt[tot] = head[u];
    head[u] = tot;
}
```

若题目给无向边，需要调用两次：

```
add(u, v, w);
add(v, u, w);
```

如果只调用一次，程序处理的是有向边。完善程序中，空缺可能出现在：

```
add(u, v, w);
①;
```

若题目说无向图，① 应填：

```
add(v, u, w)
```

若题目说有向图，就不应该补反向边。

建图填空还常考入度：

```
cin >> u >> v;
g[u].push_back(v);
①;
```

对于拓扑排序，边是 u -> v，被指向的点 v 入度增加，所以① 应填：

```
deg[v]++
```

不能填 deg[u]++。出边起点和入度增加点不要混掉。

入度与拓扑排序

拓扑排序的核心是：入度为 0 的点可以先处理；处理点 u 后，所有从 u 出发的边 u -> v 被删除，v 的入度减少。

常见代码：

```
queue<int> q;

for (int i = 1; i <= n; i++) {
    if (deg[i] == 0) {
        q.push(i);
    }
}

while (!q.empty()) {
    int u = q.front();
    q.pop();

    ans[++cnt] = u;

    for (int v : g[u]) {
        deg[v]--;

        if (deg[v] == 0) {
            q.push(v);
        }
    }
}
```

若空缺在：

```
for (int i = 1; i <= n; i++) {
    if (①) {
        q.push(i);
    }
}
```

① 应填：

```
deg[i] == 0
```

若空缺在：

```
for (int v : g[u]) {
    ②;

    if (deg[v] == 0) {
        q.push(v);
    }
}
```

② 应填：

```
deg[v]--
```

或者写成：

```
--deg[v]
```

但要注意判断位置。如果代码写成：

```
if (①) {
    q.push(v);
}
deg[v]--;
```

那么① 应该判断的是减少前的入度。若减少前为 1，减少后才变为 0，所以① 应填：

```
deg[v] == 1
```

这和先减少再判断的写法不同。

比较两种写法：

```
deg[v]--;
if (deg[v] == 0) q.push(v);
```

等价于：

```
if (deg[v] == 1) q.push(v);
deg[v]--;
```

完善程序题常把这两种顺序混在选项里。要看代码中 `deg[v]--` 在判断前还是判断后。

DAG 上的路径数量

有向无环图上，经常先拓扑排序，再按反向拓扑序计算路径数量。

例如：

```
for (int i = n; i >= 1; i--) {
    int u = topo[i];
    f[u] = 1;

    for (int v : g[u]) {
        f[u] += f[v];
    }
```

```
}
```

这里 f[u] 表示从点 u 出发的路径数量，并且单点路径也算一条。因此 f[u] 初始为 1。

若题目不把单点路径计入，初值就不应是 1，而要根据终点或题意初始化。

例如图中有边：

```
1 -> 2
1 -> 3
2 -> 4
3 -> 4
```

若单点路径也算，从每个点出发的路径数为：

点	路径数
4	1
2	$1+f[4]=2$
3	$1+f[4]=2$
1	$1+f[2]+f[3]=5$

若空缺在：

```
f[u] = ①;

for (int v : g[u]) {
    f[u] += f[v];
}
```

并且题目说“所有至少包含一个点的路径”，① 应填 1。

如果路径数可能很大，程序常会写上限：

```
const long long LIM = 1000000000000000000LL;

f[u] = 1;

for (int v : g[u]) {
    f[u] = min(f[u] + f[v], LIM);
}
```

若空缺在：

```
f[u] = ①;
```

应填：

```
min(f[u] + f[v], LIM)
```

注意不是 f[u] + f[v] + 1。因为 f[u] 在进入循环前已经包含了单点路径那 1 条，每个后继 v 贡献的是以 v 开始的路径数量。

真题式例题：DAG 第 k 小路径

下面程序求一张有向无环图中，字典序第 k 小的路径。路径至少包含一个点。程序先计算从每个点出发的路径数量，再按 k 的值逐步确定路径。

```
const long long LIM = 1000000000000000000LL;

int n, m, deg[N];
vector<int> g[N];
long long k, f[N];

int nxt(vector<int> cand, long long& k) {
    sort(cand.begin(), cand.end());

    for (int u : cand) {
        if (①) {
            return u;
        }

        k -= f[u];
    }

    return -1;
}

int main() {
    cin >> n >> m >> k;

    for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
        g[u].push_back(v);
        deg[v]++;
    }

    vector<int> q;

    for (int i = 0; i < n; i++) {
        if (!deg[i]) {
            q.push_back(i);
        }
    }

    for (int i = 0; i < n; i++) {
        int u = q[i];

        for (int v : g[u]) {
            if (②) {
                q.push_back(v);
            }

            deg[v]--;
        }
    }

    reverse(q.begin(), q.end());

    for (int u : q) {
        f[u] = 1;

        for (int v : g[u]) {
            f[u] = ③;
        }
    }

    int u = nxt(q, k);
```

```
    cout << u << endl;

    while (④) {
        ⑤;
        u = nxt(g[u], k);
        cout << u << endl;
    }
}
```

逐空分析。

`nxt` 在候选点中按编号从小到大选择。若当前 `k` 落在以 `u` 开头的路径组内，就返回 `u`。这一组大小是 `f[u]`，所以① 应填：

```
k <= f[u]
```

若 `k > f[u]`，说明要跳过这一整组，于是执行 `k -= f[u]`。

拓扑排序部分中，代码先判断，再执行 `deg[v]--`。如果减少后要变为 0，减少前应为 1，所以② 应填：

```
deg[v] == 1
```

如果代码顺序是先 `--deg[v]` 再判断，就应填 `deg[v] == 0`。本题这里判断在前。

计算路径数量时，`f[u]` 初始为 1，表示单点路径。每个后继 `v` 贡献 `f[v]` 条路径，并限制上界，所以③ 应填：

```
min(f[u] + f[v], LIM)
```

输出路径时，当前点 `u` 已经确定。以 `u` 开始的路径中，第一条是只包含 `u` 的单点路径。如果 `k==1`，路径到此结束；若 `k>1`，需要跳过单点路径，再到后继中选下一点。因此④ 应填：

```
k > 1
```

⑤ 应填：

```
--k
```

这道题的空缺分布在三个位置：选择组、拓扑入度、路径计数。做题时应先把 `f[u]` 的含义写清楚，再看 `k` 怎样跳过一整组路径。

BFS 距离填空

无权图最短路常用 BFS。

```
queue<int> q;

for (int i = 1; i <= n; i++) {
    dis[i] = -1;
}

dis[s] = 0;
q.push(s);

while (!q.empty()) {
    int u = q.front();
```

```
    q.pop();

    for (int v : g[u]) {
        if (dis[v] == -1) {
            dis[v] = dis[u] + 1;
            q.push(v);
        }
    }
}
```

这里 `dis[v] == -1` 表示还没有访问过。也可以使用 `vis[v]` 标记。

若空缺在：

```
if (①) {
    dis[v] = ②;
    q.push(v);
}
```

① 应填：

```
dis[v] == -1
```

② 应填：

```
dis[u] + 1
```

如果使用 `vis`：

```
vis[s] = true;
q.push(s);

while (!q.empty()) {
    int u = q.front();
    q.pop();

    for (int v : g[u]) {
        if (!vis[v]) {
            vis[v] = true;
            dis[v] = dis[u] + 1;
            q.push(v);
        }
    }
}
```

标记一般在入队时完成，而不是出队时完成。这样可以避免同一个点被多次入队。

若题目问最短边数，起点距离为 0；若题目问经过点数，起点可能计为 1。初始化要按题意来。

Dijkstra 初始化与松弛

带非负边权的单源最短路常用 Dijkstra。

```
const long long INF = 4e18;

struct Node {
    long long d;
```

```
    int u;

    bool operator < (const Node& other) const {
        return d > other.d;
    }
};

priority_queue<Node> pq;

for (int i = 1; i <= n; i++) {
    dis[i] = INF;
}

dis[s] = 0;
pq.push({0, s});

while (!pq.empty()) {
    Node cur = pq.top();
    pq.pop();

    int u = cur.u;

    if (cur.d != dis[u]) {
        continue;
    }

    for (auto e : g[u]) {
        int v = e.to;
        int w = e.w;

        if (dis[v] > dis[u] + w) {
            dis[v] = dis[u] + w;
            pq.push({dis[v], v});
        }
    }
}
```

常见空缺：

```
dis[①] = 0;
pq.push({0, ①});
```

① 是起点 s。

松弛条件：

```
if (②) {
    dis[v] = dis[u] + w;
}
```

② 应填：

```
dis[v] > dis[u] + w
```

如果优先队列使用 pair<int,int>，由于默认是大根堆，常见写法是放入负距离：

```
priority_queue<pair<int, int>> pq;

pq.push({0, s});
```

```
...
pq.push({-dis[v], v});
```

弹出时：

```
int u = pq.top().second;
int d = -pq.top().first;
```

若空缺在：

```
pq.push(①);
```

而程序用负距离模拟小根堆，① 应填：

```
make_pair(-dis[v], v)
```

若程序已经定义了 `greater<pair<int,int>>` 或自定义比较函数，就不一定要取负。填空前要看优先队列的定义。

memset 初始化距离

一些程序用 `memset` 初始化整型数组为较大的数：

```
memset(dis, 0x3f, sizeof(dis));
```

这样每个字节都填 `0x3f`，`int` 值会变成一个较大的正数，常用作 `INF`。

若空缺是：

```
memset(dis, ①, sizeof(dis));
```

要把距离初始化为大数，常填：

```
0x3f
```

若要初始化为 `-1`，则填：

```
-1
```

若要清零，填：

```
0
```

不要把 `memset(dis, INF, sizeof(dis))` 当成通用写法。`memset` 按字节填充，不是按整数赋值。若 `INF=1e9`，用 `memset` 填 `INF` 不会得到每个元素都是 `1e9`。

完善程序题中，如果选项有 `0x3f`、`0x1f`、`0xff`，要看目标是大正数、较小正数还是 −1。最短路距离初始通常用 `0x3f`。

最短路方案记录

有些最短路程序不仅求距离，还要记录路径。

```
if (dis[v] > dis[u] + w) {
    dis[v] = dis[u] + w;
    pre[v] = u;
}
```

pre[v] 表示最短路中 v 的前一个点是 u。

输出路径时常用递归：

```
void print(int u) {
    if (u != s) {
        print(pre[u]);
    }

    cout << u << ' ';
}
```

若空缺在：

```
pre[①] = ②;
```

当松弛边 u -> v 成功时，应填：

```
pre[v] = u
```

如果程序中把点拆成状态，如 n+u 表示另一层状态，则 pre 存的可能是状态点编号，而不是原图点编号。此时要结合输出函数看。

例如：

```
void out(int a) {
    if (a != s) {
        out(pre[a]);
    }

    cout << a % n + 1 << ' ';
}
```

这里 a 可能是状态编号，a%n 才是原图点编号。

次短路的两层状态

次短路题常把每个点拆成两层：

状态编号	含义
u	到点 u 的最短路状态
n+u	到点 u 的次短路状态

也可以用两个数组：

数组	含义
dis[u]	最短路长度
dis2[u]	严格大于最短路的最短长度

当找到更短的路径时，原来的最短路可能变成次短路；当找到介于最短路和次短路之间的路径时，更新次短路。

用状态点写时，边的含义通常是：

- 从最短路层走普通边，可能更新最短路层；
- 若不能更新最短路层，可能更新次短路层；
- 从次短路层走普通边，只能继续留在次短路层。

读这类程序时，先把 u 和 n+u 的含义写在旁边。看到：

```
int a = aa % n;
```

说明 aa 是状态点编号，a 是原图点。

看到：

```
if (aa < n) {
    ...
} else {
    ...
}
```

说明程序在区分最短路层和次短路层。

真题式例题：次短路填空

下面程序求有向图从 s 到 t 的严格次短路。如果存在，输出次短路长度和一条路径。

```
const int INF = 522133279;

int n, m, s, t;
int head[N], nxt[M], to[M], w[M], tot = 1;

int dis[N << 1], *dis2;
int pre[N << 1], *pre2;
bool vis[N << 1];

void add(int a, int b, int c) {
    ++tot;
    nxt[tot] = head[a];
    to[tot] = b;
    w[tot] = c;
    head[a] = tot;
}

bool upd(int a, int b, int d, priority_queue<pair<int, int>>& q) {
    if (d >= dis[b]) {
        return false;
    }

    if (b < n) {
        ①;
    }

    q.push(②);
    dis[b] = d;
    pre[b] = a;
    return true;
}

void solve() {
    priority_queue<pair<int, int>> q;
```

```
    q.push(make_pair(0, s));

    memset(dis, ③, sizeof(dis));
    memset(pre, -1, sizeof(pre));

    dis2 = dis + n;
    pre2 = pre + n;

    dis[s] = 0;

    while (!q.empty()) {
        int aa = q.top().second;
        q.pop();

        if (vis[aa]) {
            continue;
        }

        vis[aa] = true;

        int a = aa % n;

        for (int e = head[a]; e; e = nxt[e]) {
            int b = to[e];
            int c = w[e];

            if (aa < n) {
                if (!upd(a, b, dis[a] + c, q)) {
                    ④;
                }
            } else {
                upd(n + a, n + b, dis2[a] + c, q);
            }
        }
    }
}

void out(int a) {
    if (a != s) {
        if (a < n) {
            out(pre[a]);
        } else {
            out(⑤);
        }
    }

    printf("%d%c", a % n + 1, " \n"[a == n + t]);
}
```

先看 upd(a,b,d,q)。它尝试用前驱状态 a、目标状态 b、新距离 d 更新 dis[b]。

若 b<n，说明正在更新某个点的最短路层。原来的最短路 dis[b] 若存在，应该被下放到次短路层 n+b。它的前驱也要保留为原来的 pre[b]。因此① 应填：

```
upd(pre[b], n + b, dis[b], q)
```

这里传入的前驱是 pre[b]，目标是 n+b，距离是原最短路 dis[b]。

优先队列是默认大根堆，程序用负距离实现小根堆，所以② 应填：

```
make_pair(-d, b)
```

距离数组要初始化为大正数，③ 应填：

```
0x3f
```

在最短路层扩展边 a -> b 时，若新路径不能更新 b 的最短路，就可能成为 b 的次短路。因此④ 应填：

```
upd(a, n + b, dis[a] + c, q)
```

注意前驱状态仍是最短路层的 a，目标状态是次短路层的 n+b。

输出路径时，如果当前状态 a 在次短路层，要回到它记录的前驱。pre2[x] 是 pre[n+x] 的别名。当前 a 的原图点编号是 a%n，因此⑤ 应填：

```
pre2[a % n]
```

这道题的关键是两层状态：

- b 表示点 b 的最短路层；
- n+b 表示点 b 的次短路层；
- 最短路被更短路径替代时，旧最短路要下放为次短路候选；
- 从次短路层继续走边，仍在次短路层。

如果不先写出这两层含义，① 和④ 很容易混淆。

并查集

并查集常用于连通性和最小生成树。

基本函数：

```
int fa[N];

int find(int x) {
    if (fa[x] == x) {
        return x;
    }

    return fa[x] = find(fa[x]);
}
```

初始化：

```
for (int i = 1; i <= n; i++) {
    fa[i] = i;
}
```

合并：

```
int fx = find(x);
int fy = find(y);

if (fx != fy) {
    fa[fx] = fy;
```

```
}
```

若空缺在初始化：

```
for (int i = 1; i <= n; i++) {
    fa[i] = ①;
}
```

应填：

```
i
```

因为一开始每个点各自成一个集合，自己的父亲是自己。

若空缺在查找：

```
return fa[x] = ①;
```

应填：

```
find(fa[x])
```

这一步是路径压缩，把 x 直接连到所在集合代表元。

若空缺在合并：

```
if (fx != fy) {
    ②;
}
```

常见填法是：

```
fa[fx] = fy
```

也可以按秩合并或按大小合并，具体看题目给出的数组。

并查集填空要分清原点 x 和代表元 fx。合并时通常连代表元，不应写成 fa[x]=y，否则可能破坏集合结构。

Kruskal 最小生成树

Kruskal 常见代码：

```
struct Edge {
    int u, v, w;
};

bool cmp(Edge a, Edge b) {
    return a.w < b.w;
}

sort(e + 1, e + m + 1, cmp);

for (int i = 1; i <= n; i++) {
    fa[i] = i;
}
```

```
long long ans = 0;
int cnt = 0;

for (int i = 1; i <= m; i++) {
    int fu = find(e[i].u);
    int fv = find(e[i].v);

    if (fu != fv) {
        fa[fu] = fv;
        ans += e[i].w;
        cnt++;
    }
}
```

最小生成树要按边权从小到大排序，所以比较函数应为：

```
return a.w < b.w;
```

若要求最大生成树，则改成：

```
return a.w > b.w;
```

若空缺在：

```
if (①) {
    fa[fu] = fv;
    ans += e[i].w;
    cnt++;
}
```

应填：

```
fu != fv
```

因为只有两个端点在不同连通块中时，加入这条边才不会形成环。

如果题目要求判断是否连通，最后还要看：

```
cnt == n - 1
```

若 `cnt<n-1`，说明无法形成生成树。

Kruskal 填空的核心是：排序方向、并查集初始化、是否同集合、边数是否达到 $n-1$。

Floyd 的中转点

Floyd 常见写法：

```
for (int k = 1; k <= n; k++) {
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            if (d[i][j] > d[i][k] + d[k][j]) {
                d[i][j] = d[i][k] + d[k][j];
            }
        }
    }
```

```
}
```

三层循环中，k 是当前允许使用的中转点。它必须在最外层。

若写成：

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        for (int k = 1; k <= n; k++) {
            ...
        }
    }
}
```

这不是标准 Floyd 的状态顺序，可能无法正确表达“只使用前 k 个点作为中转点”的递推。

空缺常见在松弛式：

```
d[i][j] = min(d[i][j], ①);
```

① 应填：

```
d[i][k] + d[k][j]
```

初始化也很重要：

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        if (i == j) {
            d[i][j] = 0;
        } else {
            d[i][j] = INF;
        }
    }
}
```

读入边 u -> v 权值 w：

```
d[u][v] = min(d[u][v], w);
```

如果是无向图，还要：

```
d[v][u] = min(d[v][u], w);
```

Floyd 填空要看图的方向、初始距离和三层循环顺序。

图上 DP 的拓扑顺序

DAG 上 DP 常依赖拓扑序。若状态从前驱转移到后继：

```
f[s] = 1;

for (int u : topo) {
    for (int v : g[u]) {
        f[v] += f[u];
    }
```

```
}
```

这里必须按拓扑序从前到后处理。因为 `f[u]` 要在更新后继前已经确定。

若状态从后继汇总到当前点：

```
for (int u : rev_topo) {
    f[u] = 1;

    for (int v : g[u]) {
        f[u] += f[v];
    }
}
```

这里需要反向拓扑序。因为 `f[v]` 要先算好。

空缺常见在：

```
reverse(topo.begin(), topo.end());
```

是否需要反转，要看转移方向。

如果代码后面写：

```
for (int u : topo) {
    for (int v : g[u]) {
        f[u] += f[v];
    }
}
```

它从后继 `v` 汇总到当前 `u`，就需要 `v` 已经算好，因此应使用反向拓扑序。

如果代码后面写：

```
for (int u : topo) {
    for (int v : g[u]) {
        f[v] += f[u];
    }
}
```

它从当前 `u` 推到后继 `v`，就使用正向拓扑序。

图上 DP 填空中，拓扑序方向与转移方向必须一致。

变式题：拓扑排序补空

补全下面程序。

```
queue<int> q;

for (int i = 1; i <= n; i++) {
    if (①) {
        q.push(i);
    }
}

while (!q.empty()) {
    int u = q.front();
```

```
    q.pop();

    topo[++cnt] = u;

    for (int v : g[u]) {
        ②;

        if (③) {
            q.push(v);
        }
    }
}
```

拓扑排序初始应放入入度为 0 的点，所以① 填：

```
deg[i] == 0
```

处理边 u -> v 后，v 的入度减少，所以② 填：

```
deg[v]--
```

减少后如果入度为 0，加入队列，所以③ 填：

```
deg[v] == 0
```

变式题：无向图建图

补全下面无向带权图建图代码。

```
void add(int u, int v, int w) {
    ++tot;
    to[tot] = v;
    val[tot] = w;
    nxt[tot] = head[u];
    head[u] = tot;
}

for (int i = 1; i <= m; i++) {
    int u, v, w;
    cin >> u >> v >> w;

    add(u, v, w);
    ①;
}
```

无向边要加入两个方向，所以① 填：

```
add(v, u, w)
```

如果题目说有向图，则不填反向边。建图前一定要看题面。

变式题：Dijkstra 松弛

补全下面代码。

```
for (auto e : g[u]) {
    int v = e.to;
    int w = e.w;

    if (①) {
        dis[v] = ②;
        pre[v] = ③;
        pq.push({dis[v], v});
    }
}
```

松弛条件是通过 u 到达 v 更短：

```
dis[v] > dis[u] + w
```

所以① 填：

```
dis[v] > dis[u] + w
```

新距离为：

```
dis[u] + w
```

所以② 填：

```
dis[u] + w
```

路径前驱为 u，所以③ 填：

```
u
```

变式题：并查集补空

补全并查集。

```
int find(int x) {
    if (fa[x] == x) {
        return x;
    }

    return fa[x] = ①;
}

void unite(int x, int y) {
    int fx = find(x);
    int fy = find(y);

    if (②) {
        fa[fx] = fy;
    }
}
```

路径压缩时，x 的父亲应改成所在集合的代表元：

```
find(fa[x])
```

所以① 填：

```
find(fa[x])
```

两个点在不同集合时才合并，所以② 填：

```
fx != fy
```

变式题：Kruskal 补空

补全最小生成树程序。

```
bool cmp(Edge a, Edge b) {
    return ①;
}

sort(e + 1, e + m + 1, cmp);

for (int i = 1; i <= n; i++) {
    fa[i] = i;
}

for (int i = 1; i <= m; i++) {
    int fu = find(e[i].u);
    int fv = find(e[i].v);

    if (fu != fv) {
        ②;
        ans += e[i].w;
        cnt++;
    }
}
```

最小生成树按边权升序处理，所以① 填：

```
a.w < b.w
```

合并两个集合，可以令一个代表元的父亲指向另一个代表元，所以② 填：

```
fa[fu] = fv
```

如果题目使用按大小合并，② 可能会更复杂；但在普通 Kruskal 中这样即可。

变式题：Floyd 补空

补全 Floyd 程序。

```
for (int k = 1; k <= n; k++) {
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            if (①) {
                d[i][j] = ②;
            }
        }
    }
}
```

若经过中转点 k 可以让 i 到 j 更短，则更新：

```
d[i][j] > d[i][k] + d[k][j]
```

所以① 填：

```
d[i][j] > d[i][k] + d[k][j]
```

② 填：

```
d[i][k] + d[k][j]
```

如果有 INF，实际代码中还可能先判断 d[i][k] 和 d[k][j] 是否不是 INF，避免无意义相加。完善程序要按题目给出的代码框架判断是否需要这一层判断。

变式题：DAG 路径计数

补全代码。topo 是正向拓扑序，要求统计从起点 s 到每个点的路径条数。

```
f[s] = 1;

for (int i = 1; i <= n; i++) {
    int u = topo[i];

    for (int v : g[u]) {
        ①;
    }
}
```

从 u 到 v 有一条边，所有到达 u 的路径都可以接上这条边到达 v。因此：

```
f[v] += f[u]
```

① 应填：

```
f[v] += f[u]
```

若题目要求对模数取余，则应写：

```
f[v] = (f[v] + f[u]) % MOD
```

小结

做图论程序填空时，可以按下面顺序检查：

检查点	要问的问题
建图方向	有向边加入一次，无向边加入两次
入度	边 `u -> v` 使 `deg[v]` 增加
拓扑初始	入度为 0 的点入队
入度更新	判断发生在 `deg[v]--` 前还是后
路径计数	状态是从前驱推后继，还是从后继汇总
BFS	起点距离是 0 还是 1，标记在入队还是出队
最短路	初始距离是否为 `INF`，起点是否为 0
松弛	条件是否为“新距离更短”
优先队列	是否用负距离模拟小根堆
路径记录	`pre[v]` 是否应记录前驱点或前驱状态
状态图	`u` 与 `n+u` 等状态编号各表示什么
次短路	最短路被替换时，旧最短路是否下放
并查集	合并的是代表元，不是原点
Kruskal	排序方向与最小、最大生成树是否一致
Floyd	中转点 `k` 是否在最外层

图论程序填空要先把“图怎么存”和“状态怎么表示”写清楚。边的方向、入度更新、距离松弛、状态编号，这些都是图论代码的骨架。骨架判断对了，空缺处通常只是在正确位置填入对应的更新语句。

0x3A 搜索与递归填空

本章是第九编“完善程序专项”的最后一章。前面已经讲过初始化与边界、二分与查找、排序与归并、DP 和图论程序填空。本章处理搜索与递归填空。

搜索与递归填空的代码往往看起来很熟悉：`dfs(pos)`、`vis[i]`、`return`、`for` 枚举、递归前标记、递归后恢复。但完善程序题不会只考“这是不是 DFS”。它更常考：停止条件填什么，递归下一层传什么参数，什么时候更新答案，标记是否需要撤销，剪枝条件该写在递归前还是递归后。

搜索与递归填空要抓住四个位置：入口、停止、选择、恢复。

先看框架

搜索程序一般可以拆成：

```
void dfs(当前状态) {
    if (到达结束条件) {
        处理答案;
        return;
    }

    for (枚举下一步选择) {
        if (选择合法) {
            做选择;
            dfs(下一状态);
            撤销选择;
        }
    }
}
```

完善程序的空缺常落在这些位置：

空缺位置	常见作用
`if (...) return`	停止条件
`for (...)`	枚举选择范围
`if (...) continue`	合法性判断或剪枝
`vis[x] = ...`	标记选择
`dfs(...)`	下一层状态
`vis[x] = ...`	恢复现场
`ans++`、`ans=max(...)`	更新答案
`return true/false`	判断是否找到解

读搜索填空时，不要先看选项。先给递归参数写含义，再判断这层递归要完成什么事。

例如：

```
void dfs(int k) {
    if (k == n + 1) {
        ans++;
        return;
```

```
    }

    for (int i = 1; i <= n; i++) {
        if (!vis[i]) {
            vis[i] = true;
            dfs(k + 1);
            vis[i] = false;
        }
    }
}
```

这里 k 表示当前填写第几个位置，i 表示尝试放哪个数。停止条件是所有位置都填完，即 k==n+1。递归下一层是 dfs(k+1)，因为填完当前位置后进入下一个位置。

只要参数含义清楚，空缺就不会乱。

停止条件

递归填空首先看停止条件。停止条件通常对应“已经完成一个方案”“已经走出范围”“已经到达目标”。

排列搜索：

```
void dfs(int k) {
    if (①) {
        ans++;
        return;
    }

    for (int i = 1; i <= n; i++) {
        ...
    }
}
```

若 k 表示当前要填的位置，并且位置范围是 1 到 n，当 k==n+1 时，说明 n 个位置都填完了。① 应填：

```
k == n + 1
```

组合搜索：

```
void dfs(int x, int cnt) {
    if (cnt == m) {
        ans++;
        return;
    }

    if (x == n + 1) {
        return;
    }

    dfs(x + 1, cnt + 1);
    dfs(x + 1, cnt);
}
```

这里 x 表示当前考虑到哪个数，cnt 表示已经选了多少个。cnt==m 时已经选够，形成一个方案；x==n+1 时已经没有数可选。

迷宫搜索：

```
void dfs(int x, int y) {
    if (x == tx && y == ty) {
        ans++;
        return;
    }

    ...
}
```

这里停止条件是到达终点。

停止条件要和参数含义一致。若参数 k 表示“已经填了几个位置”，停止条件可能是 k==n；若参数 k 表示“当前要填第几个位置”，停止条件通常是 k==n+1。

这两个含义不能混用。

选择范围

搜索中 for 循环枚举下一步选择。空缺常出现在起点、终点和步长。

排列：

```
for (int i = ①; i <= ②; i++) {
    if (!vis[i]) {
        ...
    }
}
```

若要在 1 到 n 中选数，① 填 1，② 填 n。

组合：

```
void dfs(int last, int cnt) {
    if (cnt == m) {
        ans++;
        return;
    }

    for (int i = ①; i <= n; i++) {
        dfs(i, cnt + 1);
    }
}
```

若 last 表示上一次选的数，并要求选出的数严格递增，则下一次最小应从 last+1 开始。① 应填：

```
last + 1
```

这样可以避免同一组选法按不同顺序重复出现。

例如从 $1,2,3,4$ 中选 2 个数：

- 若每层都从 1 开始，会出现 $(1,2)$ 和 $(2,1)$ 两种顺序；
- 若下一层从 last+1 开始，只会得到 $(1,2)$ 这一种组合表示。

子集搜索也可以不用循环，而是用二选一：

```
void dfs(int x) {
    if (x == n + 1) {
        ans++;
        return;
    }

    choose[x] = true;
    dfs(x + 1);

    choose[x] = false;
    dfs(x + 1);
}
```

这里每个元素有选和不选两种选择。递归下一层都是 `x+1`，因为处理完当前元素后考虑下一个元素。

完善程序中，选择范围的边界通常来自题目中的可选对象范围，不要把“第几层”误当成“可选值”。

标记与恢复

回溯程序中，标记和恢复通常成对出现。

```
vis[i] = true;
dfs(k + 1);
vis[i] = false;
```

`vis[i]` 表示数字 `i` 已经在当前路径中使用。递归返回后，当前路径结束，必须把 `vis[i]` 恢复为 `false`，让其他分支可以使用 `i`。

若空缺为：

```
vis[i] = true;
dfs(k + 1);
①;
```

① 应填：

```
vis[i] = false
```

数组记录也常需要恢复：

```
path[k] = i;
vis[i] = true;
dfs(k + 1);
vis[i] = false;
```

`path[k]` 不一定需要显式恢复，因为下一次给 `path[k]` 赋值会覆盖旧值。但 `vis[i]` 会影响后续合法性判断，必须恢复。

并不是所有标记都要恢复。看图遍历：

```
void dfs(int u) {
    vis[u] = true;

    for (int v : g[u]) {
        if (!vis[v]) {
            dfs(v);
        }
```

```
    }
}
```

这里 `vis[u]` 表示点 `u` 是否已经访问过，访问后就不应再访问，不需要恢复。

判断是否恢复，要看标记含义：

标记含义	递归返回后是否恢复
当前路径中使用过	通常恢复
整个搜索中已经访问过	通常不恢复
当前排列或组合选择状态	通常恢复
已经确定的最短路、连通块访问	通常不恢复

完善程序题中，不能看到 `dfs` 就自动补恢复语句，要先看标记服务于“当前路径”还是“全局访问”。

答案更新

搜索程序中的答案更新位置很重要。

计数方案：

```
if (k == n + 1) {
    ans++;
    return;
}
```

找到一个完整方案后，答案增加 1。

求最大值：

```
if (k == n + 1) {
    ans = max(ans, sum);
    return;
}
```

找到一个完整方案后，用当前值更新最大答案。

判断是否存在解：

```
bool dfs(int k) {
    if (k == n + 1) {
        return true;
    }

    for (int i = 1; i <= n; i++) {
        if (合法) {
            if (dfs(k + 1)) {
                return true;
            }
        }
    }

    return false;
}
```

只要某一条分支找到解，就一路返回 `true`。

若空缺在：

```
if (dfs(k + 1)) {
    ①;
}
```

并且函数返回 bool，用于判断是否存在可行方案，① 通常应填：

```
return true
```

循环结束后没有找到任何方案，再返回 false。

如果题目要求统计所有方案，就不能在找到一个方案后直接 return true，而应继续枚举。

答案更新要看题目要求：

题目要求	常见写法
统计方案数	ans++ 或 ans += ...
求最大值	ans = max(ans, cur)
求最小值	ans = min(ans, cur)
判断是否存在	找到后 return true
输出所有方案	到达终点时输出路径

同一个搜索框架，答案目标不同，停止处的代码也不同。

剪枝

剪枝是在不影响正确性的前提下提前停止无用分支。

例如从 n 个数中选 m 个：

```
void dfs(int x, int cnt) {
    if (cnt == m) {
        ans++;
        return;
    }

    if (x == n + 1) {
        return;
    }

    if (cnt + (n - x + 1) < m) {
        return;
    }

    dfs(x + 1, cnt + 1);
    dfs(x + 1, cnt);
}
```

cnt 是已经选的数量，n-x+1 是从当前 x 到 n 还剩多少个数。如果即使全选也凑不够 m 个，这条分支不可能成功，可以直接返回。

若空缺为：

```
if (①) {
    return;
}
```

应填：

```
cnt + (n - x + 1) < m
```

注意是 `< m`，不是 `<= m`。如果刚好等于 m，说明后面必须全选，仍然有一个可行方案，不能剪掉。

再看求最小值：

```
void dfs(int k, int cost) {
    if (cost >= ans) {
        return;
    }

    if (k == n + 1) {
        ans = min(ans, cost);
        return;
    }

    ...
}
```

如果当前代价已经不小于已知答案，再继续搜索也不会得到更优结果，可以返回。

剪枝条件必须保证不会删掉可能成为答案的分支。完善程序题中，剪枝空缺常在不等号上，要用边界样例检查。

参数传递

递归参数有两类：一类表示当前层位置，一类表示当前累计状态。

看下面程序：

```
void dfs(int k, int sum) {
    if (k == n + 1) {
        if (sum == target) {
            ans++;
        }
        return;
    }

    dfs(k + 1, sum + a[k]);
    dfs(k + 1, sum);
}
```

这里：

参数	含义
k	当前考虑第几个数
sum	已选数字之和

选择第 k 个数时，下一层为：

```
dfs(k + 1, sum + a[k])
```

不选择时，下一层为：

```
dfs(k + 1, sum)
```

若空缺是：

```
dfs(①, ②);
```

就要看这条分支是否选择了当前元素。

搜索参数如果写错，会改变整个搜索对象。比如把 `sum+a[k]` 写成 `sum+a[k+1]`，就会把下一层元素提前加进去；把 `k+1` 写成 `k`，可能导致递归不能推进。

递归参数填空时，可以用最小样例检查：`k=1` 时下一层是否应该考虑 `2`，累计值是否加入了第 1 个元素。

记忆化搜索

记忆化搜索是递归加缓存。常见形式：

```
int f[N];

int dfs(int x) {
    if (x == 0) {
        return 1;
    }

    if (f[x] != -1) {
        return f[x];
    }

    int res = 0;

    for (int y : nxt[x]) {
        res += dfs(y);
    }

    return f[x] = res;
}
```

这里 `f[x]` 表示状态 `x` 的答案。`-1` 表示还没有计算过。

若空缺在：

```
if (①) {
    return f[x];
}
```

① 应填：

```
f[x] != -1
```

若数组初始化为 `0` 表示未计算，就要小心：有些状态的真实答案也可能是 0。此时不能用 `0` 判断是否计算过，通常要另开 `vis[x]`。

例如：

```
bool vis[N];
int f[N];

int dfs(int x) {
    if (vis[x]) {
        return f[x];
    }
```

```
    vis[x] = true;
    ...
    return f[x];
}
```

记忆化搜索填空常见位置：

空缺	常见内容
访问缓存	`if (vis[x]) return f[x];`
标记已算	`vis[x] = true;`
返回缓存	`return f[x] = res;`
初始化	`memset(f, -1, sizeof(f));`

记忆化和普通回溯的区别在于：记忆化中的 `vis[x]` 表示状态已经算过，通常不恢复；回溯中的 `vis[i]` 表示当前路径中已经使用，通常要恢复。

枚举子集

子集搜索常用递归，也常用二进制状态。

递归写法：

```
void dfs(int x) {
    if (x == n + 1) {
        ans++;
        return;
    }

    choose[x] = true;
    dfs(x + 1);

    choose[x] = false;
    dfs(x + 1);
}
```

这里共有 2^n 个子集。

二进制写法：

```
for (int s = 0; s < (1 << n); s++) {
    for (int i = 0; i < n; i++) {
        if (s & (1 << i)) {
            ...
        }
    }
}
```

若空缺在：

```
for (int s = 0; s < ①; s++)
```

应填：

```
(1 << n)
```

因为 n 个元素的子集数为 2^n，状态从 0 到 2^n-1。

若检查第 i 个元素是否被选：

```
if (①) {
    ...
}
```

应填：

```
s & (1 << i)
```

如果元素编号从 1 开始，状态位也可能从 1 开始写，但更常见的是第 i 个位置对应第 i-1 位：

```
s & (1 << (i - 1))
```

填空时要看循环中 i 的范围。若 i 从 0 到 $n-1$，用 1<<i；若 i 从 1 到 n，多半用 1<<(i-1)。

全排列

全排列搜索典型写法：

```
int p[N];
bool vis[N];

void dfs(int k) {
    if (k == n + 1) {
        output();
        return;
    }

    for (int i = 1; i <= n; i++) {
        if (vis[i]) {
            continue;
        }

        p[k] = i;
        vis[i] = true;
        dfs(k + 1);
        vis[i] = false;
    }
}
```

空缺常见在：

```
if (①) continue;
```

应填：

```
vis[i]
```

因为已经使用过的数字不能再选。

递归下一层：

```
dfs(②);
```

应填：

```
k + 1
```

恢复：

```
③;
```

应填：

```
vis[i] = false
```

如果题目要求字典序输出排列，循环必须从小到大枚举 `i`。如果要求逆字典序，则循环从大到小。

例如：

```
for (int i = n; i >= 1; i--)
```

输出顺序会改变。完善程序中如果空缺在循环起止和方向，要看题目对输出顺序是否有要求。

迷宫搜索

迷宫 DFS 常见代码：

```
int dx[4] = {-1, 1, 0, 0};
int dy[4] = {0, 0, -1, 1};

void dfs(int x, int y) {
    if (x == tx && y == ty) {
        ans++;
        return;
    }

    for (int d = 0; d < 4; d++) {
        int nx = x + dx[d];
        int ny = y + dy[d];

        if (nx < 1 || nx > n || ny < 1 || ny > m) {
            continue;
        }

        if (mp[nx][ny] == '#' || vis[nx][ny]) {
            continue;
        }

        vis[nx][ny] = true;
        dfs(nx, ny);
        vis[nx][ny] = false;
    }
}
```

入口通常是：

```
vis[sx][sy] = true;
dfs(sx, sy);
```

为什么入口前要标记起点？因为搜索过程中不能重复回到当前路径上的点。如果起点不标记，后续可能又走回起点，造成重复路径或无法结束。

若空缺在：

```
if (①) continue;
```

判断越界条件时应填：

```
nx < 1 || nx > n || ny < 1 || ny > m
```

若空缺在：

```
if (mp[nx][ny] == '#' || ②) continue;
```

② 应填：

```
vis[nx][ny]
```

若空缺在递归后：

```
dfs(nx, ny);
③;
```

③ 应填：

```
vis[nx][ny] = false
```

因为迷宫计数搜索中的 `vis` 表示当前路径是否经过，返回后要恢复。

如果题目是 BFS 求最短路，就不应恢复 `vis`。DFS 计数路径和 BFS 最短路对 `vis` 的含义不同。

真题式例题：容器分水

有两个容器，容量分别为 `a` 和 `b`。允许三种操作：

- `FILL(i)`：把容器 `i` 装满；
- `DROP(i)`：把容器 `i` 倒空；
- `POUR(i,j)`：把容器 `i` 的水倒入容器 `j`，直到 `i` 空或 `j` 满。

要求得到恰好 `c` 升水的最少操作数和操作序列。下面程序用记忆化搜索计算从状态 `(x,y)` 到目标状态的最少步数，其中 `x`、`y` 表示两个容器当前水量。

```
const int N = 110;

int f[N][N];
int ans;
int a, b, c;
int init;

int dfs(int x, int y) {
    if (f[x][y] != init) {
        return f[x][y];
    }

    if (x == c || y == c) {
        return f[x][y] = 0;
    }
```

```
    f[x][y] = init - 1;

    f[x][y] = min(f[x][y], dfs(a, y) + 1);
    f[x][y] = min(f[x][y], dfs(x, b) + 1);
    f[x][y] = min(f[x][y], dfs(0, y) + 1);
    f[x][y] = min(f[x][y], dfs(x, 0) + 1);

    int t = min(a - x, y);
    f[x][y] = min(f[x][y], ①);

    t = min(x, b - y);
    f[x][y] = min(f[x][y], ②);

    return f[x][y];
}
```

先看状态和操作。

`dfs(x,y)` 表示当前容器 1 有 x 升水、容器 2 有 y 升水时，到达目标的最少操作数。

前四个转移分别是：

操作	新状态
FILL(1)	(a,y)
FILL(2)	(x,b)
DROP(1)	(0,y)
DROP(2)	(x,0)

接着是倒水。

第一段：

```
int t = min(a - x, y);
```

`a-x` 是容器 1 剩余容量，`y` 是容器 2 当前水量。`t` 表示从容器 2 倒入容器 1 的水量。新状态为：

```
(x + t, y - t)
```

所以① 应填：

```
dfs(x + t, y - t) + 1
```

第二段：

```
t = min(x, b - y);
```

`x` 是容器 1 当前水量，`b-y` 是容器 2 剩余容量。`t` 表示从容器 1 倒入容器 2 的水量。新状态为：

```
(x - t, y + t)
```

所以② 应填：

```
dfs(x - t, y + t) + 1
```

继续看输出操作序列：

```
void go(int x, int y) {
    if (③) {
        return;
    }

    if (f[x][y] == dfs(a, y) + 1) {
        cout << "FILL(1)" << endl;
        go(a, y);
    } else if (f[x][y] == dfs(x, b) + 1) {
        cout << "FILL(2)" << endl;
        go(x, b);
    } else if (f[x][y] == dfs(0, y) + 1) {
        cout << "DROP(1)" << endl;
        go(0, y);
    } else if (f[x][y] == dfs(x, 0) + 1) {
        cout << "DROP(2)" << endl;
        go(x, 0);
    } else {
        int t = min(a - x, y);

        if (f[x][y] == ④) {
            cout << "POUR(2,1)" << endl;
            go(x + t, y - t);
        } else {
            t = min(x, b - y);

            if (f[x][y] == ⑤) {
                cout << "POUR(1,2)" << endl;
                go(x - t, y + t);
            }
        }
    }
}
```

go(x,y) 负责从当前状态输出一条最优操作路径。当已经达到目标，即某个容器水量为 c，就停止输出，所以③应填：

```
x == c || y == c
```

④ 对应 POUR(2,1)，与前面①相同：

```
dfs(x + t, y - t) + 1
```

⑤ 对应 POUR(1,2)，与前面②相同：

```
dfs(x - t, y + t) + 1
```

这道题的关键是把每个操作对应的新状态写出来。只要状态变化清楚，五个空缺都可以从操作含义推出。

真题式例题：选择数字组成目标和

补全下面程序。给定 n 个正整数，判断是否能选出若干个数，使它们的和等于 target。每个数最多选一次。

```
int n, target;
int a[25];
bool ok = false;
```

```
void dfs(int k, int sum) {
    if (sum == target) {
        ok = true;
        return;
    }

    if (①) {
        return;
    }

    if (sum > target) {
        return;
    }

    dfs(②, ③);
    dfs(④, ⑤);
}
```

k 表示当前考虑第几个数，sum 表示已经选中的数之和。

若已经考虑完所有数，即 k==n+1，还没有达到目标，就返回。① 应填：

```
k == n + 1
```

选择第 k 个数时，下一层考虑 k+1，和增加 a[k]：

```
dfs(k + 1, sum + a[k])
```

所以② 填 k + 1，③ 填 sum + a[k]。

不选择第 k 个数时，下一层仍是 k+1，和不变：

```
dfs(k + 1, sum)
```

所以④ 填 k + 1，⑤ 填 sum。

这段程序中 sum>target 的剪枝依赖 a[i] 都是正整数。如果允许负数，这个剪枝就不一定正确。

真题式例题：排列限制

补全下面程序。求 1 到 n 的排列数量，要求相邻两个数之差不能为 1。

```
int n, p[15], ans;
bool vis[15];

void dfs(int k) {
    if (①) {
        ans++;
        return;
    }

    for (int i = 1; i <= n; i++) {
        if (②) {
            continue;
        }

        if (k > 1 && ③) {
            continue;
```

```
        }

        p[k] = i;
        vis[i] = true;
        dfs(④);
        ⑤;
    }
}
```

k 表示当前填写第 k 个位置。填完第 n 个位置后，下一层是 k==n+1，所以① 填：

```
k == n + 1
```

已经使用过的数字不能再用，所以② 填：

```
vis[i]
```

相邻两个数之差不能为 1，当前尝试的数是 i，前一个数是 p[k-1]，禁止条件是：

```
abs(i - p[k - 1]) == 1
```

所以③ 填：

```
abs(i - p[k - 1]) == 1
```

填完当前位置后进入下一层，④ 填：

```
k + 1
```

返回后恢复标记，⑤ 填：

```
vis[i] = false
```

这类排列填空先看位置参数，再看合法性条件。相邻限制通常只在 k>1 时检查，因为第一个位置没有前一个数。

变式题：组合剪枝

补全下面程序，从 1 到 n 中选 m 个数，统计方案数。

```
int n, m, ans;

void dfs(int x, int cnt) {
    if (cnt == m) {
        ans++;
        return;
    }

    if (x == n + 1) {
        return;
    }

    if (①) {
        return;
    }
```

```
    dfs(x + 1, cnt + 1);
    dfs(x + 1, cnt);
}
```

cnt 是已经选择的数量，当前还可以考虑的数是 $x, x+1, \ldots, n$，共 $n-x+1$ 个。若这些全选也不够 m 个，就可以剪枝。

所以① 应填：

```
cnt + (n - x + 1) < m
```

如果刚好等于 m，后面全选仍是一个方案，不能剪掉。

变式题：迷宫 DFS

补全下面代码，统计从起点到终点的不重复路径数。`'.'` 表示可走，`'#'` 表示障碍。

```
void dfs(int x, int y) {
    if (x == tx && y == ty) {
        ans++;
        return;
    }

    for (int d = 0; d < 4; d++) {
        int nx = x + dx[d];
        int ny = y + dy[d];

        if (①) {
            continue;
        }

        if (mp[nx][ny] == '#' || ②) {
            continue;
        }

        vis[nx][ny] = true;
        dfs(nx, ny);
        ③;
    }
}
```

越界条件是：

```
nx < 1 || nx > n || ny < 1 || ny > m
```

所以① 填：

```
nx < 1 || nx > n || ny < 1 || ny > m
```

不能走已经在当前路径中的格子，所以② 填：

```
vis[nx][ny]
```

返回后恢复当前路径标记，所以③ 填：

```
vis[nx][ny] = false
```

入口处还应有：

```
vis[sx][sy] = true;
dfs(sx, sy);
```

否则可能从相邻格子又走回起点。

变式题：记忆化搜索

补全下面程序。`dfs(x)` 返回从状态 `x` 出发到终点的方案数。`vis[x]` 表示该状态是否已经计算过。

```
int dfs(int x) {
    if (x == target) {
        return 1;
    }

    if (①) {
        return f[x];
    }

    ②;

    int res = 0;

    for (int y : nxt[x]) {
        res += dfs(y);
    }

    return ③;
}
```

如果状态已经算过，直接返回缓存，所以① 填：

```
vis[x]
```

第一次计算状态 `x` 时，应标记它已经计算过，所以② 填：

```
vis[x] = true
```

最后把结果存入 `f[x]` 并返回，所以③ 填：

```
f[x] = res
```

或写成：

```
return f[x] = res
```

在本代码位置已经有 `return \circlednum{3};`，所以③ 填 `f[x] = res` 即可。

记忆化搜索中的 `vis[x]` 不恢复，因为它表示“这个状态答案已经算出”，不是当前路径标记。

变式题：存在性搜索

补全下面程序，判断能否从数组中选若干数，使和为 `target`。

```
bool dfs(int k, int sum) {
    if (sum == target) {
        return true;
    }

    if (k == n + 1 || sum > target) {
        return false;
    }

    if (dfs(k + 1, sum + a[k])) {
        ①;
    }

    if (dfs(k + 1, sum)) {
        ②;
    }

    return ③;
}
```

这是判断是否存在方案。只要某个分支成功，就应立即返回 `true`。

所以① 填：

```
return true
```

② 也填：

```
return true
```

两个分支都失败后返回 `false`，所以③ 填：

```
false
```

这类题不要继续统计所有方案。函数返回类型是 `bool`，说明目标是存在性判断。

变式题：全排列字典序输出

补全代码，按字典序输出 1 到 n 的所有排列。

```
void dfs(int k) {
    if (k == n + 1) {
        output();
        return;
    }

    for (int i = ①; ②; ③) {
        if (vis[i]) {
            continue;
        }

        p[k] = i;
        vis[i] = true;
        dfs(k + 1);
        vis[i] = false;
    }
}
```

字典序要求每一层从小到大尝试，所以：

① 填：

```
1
```

② 填：

```
i <= n
```

③ 填：

```
i++
```

若要逆字典序输出，则应从 `n` 到 `1` 枚举。

变式题：子集状态枚举

补全代码，枚举 n 个元素的所有子集。元素编号为 1 到 n，第 `i` 个元素对应二进制状态的第 `i-1` 位。

```
for (int s = 0; s < ①; s++) {
    int sum = 0;

    for (int i = 1; i <= n; i++) {
        if (②) {
            sum += a[i];
        }
    }

    ...
}
```

共有 2^n 个子集，状态从 0 到 2^n-1，所以① 填：

```
(1 << n)
```

第 `i` 个元素对应第 `i-1` 位，所以② 填：

```
s & (1 << (i - 1))
```

如果内层 `i` 从 0 开始，条件就会变成 `s & (1 << i)`。填空时要看编号范围。

小结

做搜索与递归填空时，可以按下面顺序检查：

检查点	要问的问题
递归参数	每个参数表示当前位置、已选数量、当前和，还是当前点
停止条件	什么时候形成完整方案，什么时候无路可走
选择范围	下一步从哪些值中枚举
递归推进	下一层参数是否真正向前推进
合法性判断	已用、越界、障碍、相邻限制是否检查完整
标记	标记表示当前路径，还是全局访问
恢复	当前路径标记是否在递归后恢复
答案更新	统计数量、最优值、存在性分别怎样处理
剪枝	是否会错误剪掉可行方案
记忆化	`vis` 是否表示状态已计算，是否不应恢复
子集状态	位编号与元素编号是否错位
输出顺序	枚举顺序是否符合题目要求

搜索与递归填空不适合靠模板硬套。先说清“这一层递归正在做什么”，再看它如何进入下一层。当前路径标记通常要恢复，全局访问标记通常不恢复；计数题要继续枚举，存在性题找到后可以返回。把这几条分清，第九编的完善程序专项就完成了。

第十编 综合训练

0x3B 选择题分类训练

本章进入第十编“综合训练”。前九编已经分别讲过知识点、程序阅读方法和完善程序方法；从本章开始，训练目标改为查漏补缺。题目不再服务某一个新知识点，而是尽量贴近第一轮试卷的表达方式，把容易混在一起考的内容放到同一章中。

本章训练选择题。为便于复习，题目按知识点分组。

本章题目为原创训练题，题干与选项采用第一轮常见表述方式。

一、单项选择题

复杂度与程序行为

1. 在 Linux 系统终端中，若要将当前目录下的 `main.cpp` 编译为名为 `main` 的可执行文件，以下命令正确的是（）。

   A. `g++ main.cpp -o main`

   B. `g++ main -o main.cpp`

   C. `g++ -run main.cpp main`

   D. `compile main.cpp main`

2. 对于给定的正整数 n，分析以下代码段的时间复杂度。

```
int s = 0;
for (int i = 1; i <= n; i *= 2) {
    for (int j = 1; j <= i; j++) {
        s++;
    }
}
```

该代码段的时间复杂度为（）。

   A. $O(\log n)$

   B. $O(n)$

   C. $O(n \log n)$

   D. $O(n^2)$

3. 对于给定的正整数 n，分析以下代码段的时间复杂度。

```
int cnt = 0;
for (int i = 1; i <= n; i++) {
    for (int j = i; j <= n; j += i) {
        cnt++;
    }
}
```

该代码段的时间复杂度为 ()。

A. $O(n)$

B. $O(n\sqrt{n})$

C. $O(n\log n)$

D. $O(n^2)$

4. 假设 n 为 2 的正整数幂，分析以下函数 `f(n)` 的时间复杂度。

```
int f(int n) {
    if (n <= 1) return 1;
    return f(n / 2) + f(n / 2);
}
```

该函数的时间复杂度为 ()。

A. $O(\log n)$

B. $O(\sqrt{n})$

C. $O(n)$

D. $O(n\log n)$

5. 有如下程序片段：

```
int x = 1;
for (int i = 1; i <= 5; i++) {
    int x = i;
}
cout << x << endl;
```

程序输出为 ()。

A. `1`

B. `5`

C. `6`

D. 编译错误

数学与计数

6. 有 5 个红色球和 4 个蓝色球，它们除了颜色之外完全相同。将这 9 个球排成一排，要求任意两个蓝色球都不能相邻，有多少种不同的排列方法？()

A. 10

B. 15

C. 20

D. 24

7. 1 到 300 之间，不能被 2、3、5 中任意一个数整除的整数有多少个？()

A. 70

B. 80

C. 90

D. 100

8. 一位玩家连续掷两次骰子。第一次掷出 x 点，第二次掷出 y 点。若 $y > x$，玩家获得 $y - x$ 元；否则收益为 0 元。假设骰子掷出任意一点的概率均为 $\frac{1}{6}$，玩家两次掷骰后的平均收益为 ()。

A. $\frac{5}{6}$

B. $\frac{35}{36}$

C. 1

D. $\frac{7}{6}$

9. 共有 8 名同学，要从中选出 3 人组成一个小组。小组内部不区分职务和顺序，共有多少种不同的选法？()

A. 24

B. 48

C. 56

D. 336

10. 数列满足 $a_1 = 1$，$a_2 = 2$，当 $n \geq 3$ 时有 $a_n = 2a_{n-1} + a_{n-2}$。则 a_5 的值为 ()。

A. 17

B. 24

C. 29

D. 34

数据结构

11. 元素 $1, 2, 3, 4$ 依次进栈，允许进栈和出栈操作交替进行。以下哪个出栈序列不可能出现？()

A. $2, 1, 4, 3$

B. $3, 2, 1, 4$

C. $4, 1, 2, 3$

D. $1, 3, 2, 4$

12. 给定地址区间为 0 到 10 的哈希表，哈希函数为 $h(x) = x \bmod 11$，采用线性探查解决冲突。哈希表初始为空，依次插入 $22, 1, 12, 23, 34$ 后，34 被存储在地址 ()。

A. 1

B. 2

C. 3

D. 4

13. 在一个初始为空的最小堆中，依次插入元素 $20, 12, 15, 8, 10, 5$，然后连续执行两次删除最小值操作。此时堆顶元素为 ()。

A. 8

B. 10

C. 12

D. 15

14. 令根结点的高度为 1，一棵含有 2025 个结点的二叉树的高度至少为 ()。

A. 10

B. 11

C. 12

D. 2025

15. 如果一棵二叉搜索树的后序遍历序列为 $1, 4, 3, 7, 9, 8, 5$，那么该树的前序遍历序列为 ()。

A. $5, 3, 1, 4, 8, 7, 9$

B. $5, 8, 7, 9, 3, 1, 4$

C. $1, 3, 4, 5, 7, 8, 9$

D. $5, 3, 4, 1, 8, 9, 7$

16. 将字符串 `to`、`tea`、`ted`、`ten`、`in`、`inn` 插入一棵初始为空的 Trie 树中。若根结点也计入结点总数，则构造完成后共有多少个结点？()

A. 8

B. 9

C. 10

D. 11

17. 设线段树维护下标 1 到 8 的数组，根结点表示区间 $[1, 8]$。若只考虑将查询区间 $[3, 6]$ 拆成若干个互不相交且完整覆盖的线段树结点区间，则最少可以拆成多少个区间？()

A. 1

B. 2

C. 3

D. 4

树与图

18. 一个非连通简单无向图共有 21 条边，则该图至少有多少个顶点？()

A. 7

B. 8

C. 9

D. 10

19. 对于一个至少有一个顶点的有向无环图，以下说法一定正确的是 ()。

A. 至少存在一个入度为 0 的顶点

B. 每个顶点的入度都为 0

C. 拓扑排序结果唯一

D. 任意两个顶点之间至多存在一条路径

20. 一个连通无向图中每个顶点的度数均为偶数。以下说法不一定正确的是 ()。

A. 该图存在欧拉回路

B. 该图中所有边可以被一条闭合路线经过且每条边恰好经过一次

C. 该图不存在度数为奇数的顶点

D. 该图的边数一定为偶数

21. 以下关于二分图的说法中，正确的是 ()。

A. 任意一棵树都不是二分图

B. 任意完全图都是二分图

C. 一个无向图是二分图，当且仅当它不含奇环

D. 只要图的边数为偶数，它就是二分图

22. 有一个包含 10 个顶点的完全图，顶点编号为 1 到 10。任意两点之间边权等于两顶点编号之差的绝对值，例如顶点 3 与顶点 8 之间的边权为 5。该图的最小生成树总权重为 ()。

A. 8

B. 9

C. 10

D. 45

23. 使用 Dijkstra 算法求单源最短路时，以下哪个条件是通常需要满足的？()

A. 所有边权均非负

B. 图必须是完全图

C. 图必须是有向无环图

D. 所有边权必须相等

字符串与序列

24. 在 KMP 算法中，对于模式串 `ababaca`，若 `next[i]` 定义为模式串前缀 `P[0..i]` 的最长公共前后缀长度，且下标从 0 开始，则 `next` 数组为 ()。

A. $\{0, 0, 1, 2, 3, 0, 1\}$

B. $\{0, 1, 2, 3, 0, 1, 2\}$

C. $\{0, 0, 1, 2, 0, 1, 2\}$

D. $\{0, 0, 1, 0, 1, 2, 3\}$

25. 序列 `ABCBDAB` 和 `BDCABA` 的最长公共子序列长度为 ()。

A. 2

B. 3

C. 4

D. 5

26. 定义一次操作为交换相邻两个字符。将字符串 `BDCA` 变为 `ABCD`，最少需要多少次操作？()

A. 3

B. 4

C. 5

D. 6

DP、贪心与搜索

27. 一个 0-1 背包问题，背包容量为 10。现有 4 个物品，它们的重量分别为 $6, 3, 4, 5$，价值分别为 $30, 14, 16, 20$。能够获得的最大总价值为 ()。

A. 34

B. 44

C. 46

D. 50

28. 给定整数序列 $-2, 3, -1, 5, -6, 4$，其最大连续子段和为 ()。

A. 6

B. 7

C. 8

D. 9

29. 从 $1, 2, 3, 4$ 中选出所有数排成一列，要求任意相邻两个数的差的绝对值都不等于 1。满足条件的排列有多少个？()

A. 0

B. 2

C. 4

D. 8

30. 有如下二分查找代码，其中数组 `a[1..n]` 单调不降，且保证存在至少一个元素大于等于 `x`。

```
int find(int a[], int n, int x) {
    int l = 1, r = n;
    while (l < r) {
        int mid = (l + r) >> 1;
        if (a[mid] >= x) r = mid;
        else l = mid + 1;
    }
    return l;
```

```
}
```

当 `n=5`，数组为 $1,3,3,5,7$，`x=4` 时，函数返回值为（）。

A. 3

B. 4

C. 5

D. 7

二、参考答案

题号	1	2	3	4	5	6	7	8	9	10
答案	A	B	C	C	A	B	B	B	C	C

题号	11	12	13	14	15	16	17	18	19	20
答案	C	D	B	B	A	C	B	B	A	D

题号	21	22	23	24	25	26	27	28	29	30
答案	C	B	A	A	C	B	C	B	B	B

三、解析

1. `g++ main.cpp -o main` 表示用 `main.cpp` 作为源文件，输出可执行文件 `main`。`-o` 后面跟输出文件名。
2. 外层循环中 i 依次为 $1,2,4,\ldots$，内层总执行次数为 $1+2+4+\cdots$，总和不超过 $2n$，时间复杂度为 $O(n)$。
3. 对固定的 i，内层循环执行 $\lfloor n/i \rfloor$ 次，总次数为 $\sum_{i=1}^{n}\lfloor n/i \rfloor$，量级为 $O(n\log n)$。
4. 每次调用产生两个规模为原来一半的子调用，递归树每层调用规模数量翻倍，一直到规模为 1。调用总数与 n 同阶，时间复杂度为 $O(n)$。
5. 循环内部的 `int x = i;` 定义的是新的局部变量，只在循环体内有效，不会改变外层的 `x`。循环结束后输出外层 `x`，结果为 `1`。
6. 先排 5 个红球，形成 6 个空位，包括两端和红球之间。为了使蓝球不相邻，只需从这 6 个空位中选 4 个放蓝球，方案数为 $\binom{6}{4}=15$。
7. 用容斥计算。1 到 300 中被 $2,3,5$ 中至少一个整除的个数为 $150+100+60-50-30-20+10=220$，所以不能被它们任意一个整除的个数为 $300-220=80$。
8. 只在 $y>x$ 时有收益。差值为 d 的情况有 $6-d$ 种，所以总收益为 $1\cdot 5+2\cdot 4+3\cdot 3+4\cdot 2+5\cdot 1=35$。共有 36 种等可能结果，平均收益为 $\frac{35}{36}$。
9. 从 8 人中选 3 人，不区分顺序，方案数为 $\binom{8}{3}=56$。
10. 逐项计算：$a_3=2a_2+a_1=5$，$a_4=2a_3+a_2=12$，$a_5=2a_4+a_3=29$。
11. 若第一个出栈元素为 4，说明 $1,2,3,4$ 都已经入栈，且 4 出栈后栈顶为 3，不可能接着输出 1。所以 $4,1,2,3$ 不可能出现。
12. 插入过程为：22 放入 0，1 放入 1，12 原地址为 1，冲突后放入 2；23 原地址为 1，依次探查到 3；34 原地址为 1，依次探查到 4。
13. 最小堆连续删除两次最小值，相当于先删去 5，再删去 8。剩余元素中最小的是 10，所以堆顶为 10。
14. 高度为 h 的二叉树最多有 2^h-1 个结点。$2^{10}-1=1023$，$2^{11}-1=2047$，要容纳 2025 个结点，高度至少为 11。

15. 二叉搜索树后序遍历的最后一个数是根，为 5。左子树元素小于 5，为 $1,4,3$；右子树元素大于 5，为 $7,9,8$。继续按后序拆分，得到前序遍历为 $5,3,1,4,8,7,9$。

16. Trie 结点为：根、`t`、`to` 的 `o`、`te` 的 `e`、`tea` 的 `a`、`ted` 的 `d`、`ten` 的 `n`、`i`、`in` 的 `n`、`inn` 的第二个 `n`，共 10 个。

17. 在线段树中，区间 $[3,6]$ 可以拆为 $[3,4]$ 和 $[5,6]$ 两个完整结点区间，且不能用一个线段树结点完整表示，所以答案为 2。

18. 非连通简单无向图若有 n 个顶点，最多可以让其中 $n-1$ 个顶点构成完全图，另一个顶点孤立。要有 21 条边，需要 $\binom{n-1}{2} \geq 21$。当 $n=8$ 时，$\binom{7}{2}=21$，可以达到。

19. 有向无环图中若每个顶点入度都大于 0，从任意点不断沿入边往前走，由于顶点数有限，最终会形成环，与无环矛盾。因此至少存在一个入度为 0 的顶点。

20. 连通无向图所有顶点度数为偶数时存在欧拉回路。边数不一定为偶数，例如三角形有 3 条边，每个顶点度数都是 2。

21. 二分图可以把顶点分成两部分，使每条边连接不同部分的顶点。它的等价判定是不存在奇环。树没有环，因此树是二分图，选项 A 错；完全图 K_3 不是二分图，选项 B 错；边数为偶数不能保证二分图，选项 D 错。

22. 选取边 $(1,2),(2,3),\ldots,(9,10)$ 可以连通所有顶点，总权重为 9。任意生成树需要 9 条边，每条边权至少为 1，所以最小生成树总权重不能小于 9。

23. Dijkstra 算法确定一个点的最短距离后，不再回头修改该点。这个性质依赖边权非负。若存在负边，后续路径可能使已经确定的距离变小。

24. 逐位计算最长公共前后缀长度：`a` 为 0，`ab` 为 0，`aba` 为 1，`abab` 为 2，`ababa` 为 3，`ababac` 为 0，`ababaca` 为 1，所以数组为 $\{0,0,1,2,3,0,1\}$。

25. 两个序列的最长公共子序列长度为 4，例如 `BCBA` 或 `BDAB` 都可以作为长度为 4 的公共子序列，无法得到长度为 5 的公共子序列。

26. 将 `BDCA` 中每个字符换成它在目标串 `ABCD` 中的位置，得到序列 $2,4,3,1$。相邻交换最少次数等于逆序对数。该序列逆序对为 $(2,1),(4,3),(4,1),(3,1)$，共 4 个。

27. 可选组合中，物品 1 和物品 3 的总重量为 $6+4=10$，总价值为 $30+16=46$。其他合法组合的价值不超过 46，所以最大总价值为 46。

28. 最大连续子段为 $3,-1,5$，和为 7。包含后面的 -6 会使和下降；单独的 4 小于 7。

29. 枚举或回溯可得满足条件的排列为 $2,4,1,3$ 和 $3,1,4,2$，共 2 个。其他排列中至少会出现一对相邻数差的绝对值为 1。

30. 该函数返回第一个满足 `a[pos] >= x` 的位置。数组为 $1,3,3,5,7$，第一个大于等于 4 的元素是 5，位置为 4。

0x3C 程序阅读分类训练

本章训练程序阅读。前面已经讲过变量追踪、循环程序、递归程序、数组字符串程序和树图程序；这里不再重新讲知识点，而是把它们合在一起，按试卷中的阅读程序形式训练。

阅读程序题建议先独立完成。做题时不要急着判断程序“像什么算法”，先把输入、变量、循环边界和输出位置看清楚。答案和解析统一放在本章末尾，便于先做题，再核对。

本章题目为原创训练题，题干和选项采用第一轮常见表述方式。

一、阅读程序

第一题

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

int n;
int a[105], s[105], b[105];

int main() {
    cin >> n;
    for (int i = 1; i <= n; i++) cin >> a[i];

    for (int i = 1; i <= n; i++) {
        s[i] = s[i - 1] + a[i];
        b[i] = b[i - 1];
        if (s[i] < s[b[i]]) b[i] = i;
    }

    int ans = -1000000000;
    int l = 1, r = 1;

    for (int i = 1; i <= n; i++) {
        int v = s[i] - s[b[i - 1]];
        if (v > ans) {
            ans = v;
            l = b[i - 1] + 1;
            r = i;
        }
    }

    cout << ans << " " << l << " " << r << endl;
    return 0;
}
```

假设输入的 n 为正整数，且输入数据不会使 `int` 类型溢出。完成下面的判断题和单选题。

判断题

1. 程序运行完**第二个 for 循环**后，`s[i]` 表示 $a_1 + a_2 + \cdots + a_i$。()

   A. 正确

B. 错误

2. 程序运行完**第二个 for 循环**后，`b[i]` 表示在 $0, 1, \ldots, i$ 中，使得 `s` 值最小的下标之一。()

A. 正确

B. 错误

3. 若输入数组中所有元素均为正数，则程序输出的第二个数一定为 1。()

A. 正确

B. 错误

单选题

4. 当输入为

```
6
-2 3 -1 5 -6 4
```

时，程序输出为 ()。

A. `7 2 4`

B. `7 1 4`

C. `8 2 6`

D. `5 4 4`

5. 如果将第二重循环中的 `s[b[i - 1]]` 改为 `s[b[i]]`，则程序输出的第一个数一定不变。()

A. 正确

B. 错误

6. 该程序的时间复杂度为 ()。

A. $O(1)$

B. $O(\log n)$

C. $O(n)$

D. $O(n^2)$

第二题

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

int n, ans;
char s[25];

bool ok(int k) {
    if (k < 3) return true;
    return !(s[k - 2] == '1' && s[k - 1] == '0' && s[k] == '1');
}
```

```
void dfs(int k, int cnt) {
    if (k == n + 1) {
        if (cnt % 2 == 0) ans++;
        return;
    }

    for (char c = '0'; c <= '1'; c++) {
        s[k] = c;
        if (ok(k)) dfs(k + 1, cnt + (c == '1'));
    }
}

int main() {
    cin >> n;
    dfs(1, 0);
    cout << ans << endl;
    return 0;
}
```

假设输入的 n 为正整数，且不超过数组定义范围。完成下面的判断题和单选题。

判断题

1. 当输入为 2 时，程序输出为 2。()

   A. 正确

   B. 错误

2. 该程序统计的是所有长度为 n、且字符 `'1'` 个数为偶数的 01 串数量。()

   A. 正确

   B. 错误

3. 在 `dfs` 函数运行过程中，参数 `k` 的取值可能为 $n+1$。()

   A. 正确

   B. 错误

单选题

4. 当输入为 4 时，程序输出为 ()。

   A. 4

   B. 5

   C. 6

   D. 8

5. 若删除 `dfs` 函数中的 `if (ok(k))` 判断，改为每次都递归调用 `dfs(k + 1, cnt + (c == '1'))`，当输入为 3 时，程序输出为 ()。

   A. 3

   B. 4

   C. 5

D. 8

6. 该程序的时间复杂度上界可以表示为 ()。

A. $O(n)$

B. $O(n^2)$

C. $O(2^n)$

D. $O(n!)$

第三题

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

const int N = 105;

int n, m;
int deg[N], dp[N];
vector<int> g[N];
queue<int> q;

int main() {
    cin >> n >> m;

    for (int i = 1; i <= m; i++) {
        int u, v;
        cin >> u >> v;
        g[u].push_back(v);
        deg[v]++;
    }

    dp[1] = 1;

    for (int i = 1; i <= n; i++) {
        if (deg[i] == 0) q.push(i);
    }

    int cnt = 0;

    while (!q.empty()) {
        int u = q.front();
        q.pop();
        cnt++;

        sort(g[u].begin(), g[u].end());

        for (int v : g[u]) {
            dp[v] += dp[u];
            deg[v]--;
            if (deg[v] == 0) q.push(v);
        }
    }

    cout << cnt << " " << dp[n] << endl;
    return 0;
}
```

假设输入的点编号均在 1 到 n 之间，且输入数据不会使 `int` 类型溢出。完成下面的判断题和单选题。

判断题

1. 若输入图是有向无环图，则程序输出的第一个数一定为 n。()

   A. 正确

   B. 错误

2. 删除 `sort(g[u].begin(), g[u].end());` 后，程序输出的第二个数一定不变。()

   A. 正确

   B. 错误

3. 若输入图是有向无环图，则程序输出的第二个数表示从 1 到 n 的最短路径条数。()

   A. 正确

   B. 错误

单选题

4. 当输入为

```
5 6
1 2
1 3
2 4
3 4
2 5
4 5
```

时，程序输出为 ()。

   A. `5 2`

   B. `5 3`

   C. `4 3`

   D. `5 4`

5. 当输入为

```
3 3
1 2
2 3
3 2
```

时，程序输出为 ()。

   A. `1 0`

   B. `2 1`

   C. `3 1`

   D. `3 0`

6. 计入 `sort` 的时间开销，该程序的时间复杂度上界可以表示为 ()。

A. $O(n+m)$

B. $O(nm)$

C. $O(n+m\log m)$

D. $O(2^n)$

二、参考答案

第一题

小题	1	2	3	4	5	6
答案	A	A	A	A	B	C

第二题

小题	1	2	3	4	5	6
答案	A	B	A	C	B	C

第三题

小题	1	2	3	4	5	6
答案	A	A	B	B	A	C

三、解析

第一题解析

这段程序的核心是前缀和。全局数组初值为 0，所以 `s[0]=0`，`b[0]=0`。第一重循环中，`s[i]=s[i-1]+a[i]`，所以 `s[i]` 表示前 i 个数的和。

数组 `b` 记录最小前缀和的位置。执行到第 i 位时，先令 `b[i]=b[i-1]`，表示沿用前面最小前缀和的位置；若 `s[i] < s[b[i]]`，再把位置改为 i。注意这里使用的是严格小于，所以遇到相同的最小值时保留较早的位置。

第二重循环中，`s[i]-s[b[i-1]]` 表示右端点为 i、左端点不超过 i 的一段连续子段和。这里用 `b[i-1]`，是因为子段必须至少包含 a_i，不能用当前的 `s[i]` 把长度为 0 的空段算进去。

对于第 4 小题，输入数组为 $-2,3,-1,5,-6,4$。前缀和依次为：

下标	0	1	2	3	4	5	6
s	0	-2	1	0	5	-1	3

最大连续子段为 $a_2+a_3+a_4=3+(-1)+5=7$，因此输出 `7 2 4`。

第 5 小题中，如果把 `b[i - 1]` 改成 `b[i]`，当 `s[i]` 本身成为新的最小前缀和时，程序会计算 `s[i]-s[i]=0`。例如数组只有一个元素 -5 时，原程序输出的最大子段和为 -5，修改后会得到 0，所以输出的第一个数不一定不变。

整个程序只有两个从 1 到 n 的循环，时间复杂度为 $O(n)$。

第二题解析

这段程序用深度优先搜索枚举长度为 n 的 01 串。数组 `s[1..n]` 记录当前构造出的字符串，`cnt` 记录其中字符 `'1'` 的个数。

函数 `ok(k)` 只检查最后三个字符。当 $k < 3$ 时，当前还不足三个字符，不可能出现长度为 3 的模式；当 $k \geq 3$ 时，若 `s[k-2]s[k-1]s[k]` 等于 `101`，这一分支被剪去。因此，程序统计的是长度为 n、不含连续片段 `101`、并且字符 `'1'` 个数为偶数的 01 串数量。

第 1 小题中，长度为 2 的 01 串为 `00`、`01`、`10`、`11`。其中 `'1'` 个数为偶数的有 `00` 和 `11`，且长度不足 3，不会被 `ok` 剪去，所以输出 2。

第 2 小题错在漏掉了 `ok(k)` 的限制。程序不是只看 `'1'` 的个数，还要求不能出现 `101`。

第 3 小题中，当已经填完 n 个位置后，会调用 `dfs(n+1, cnt)`，此时进入递归出口，所以 `k` 的取值可能为 $n+1$。

第 4 小题中，长度为 4、满足条件的字符串有：

```
0000
0011
0110
1001
1100
1111
```

共有 6 个，所以答案为 C。

第 5 小题中，删去 `ok(k)` 后，程序只统计长度为 3 且 `'1'` 个数为偶数的 01 串。数量为 $\binom{3}{0}+\binom{3}{2}=1+3=4$。

该搜索每一位最多分成两个分支，`ok(k)` 的检查为常数时间，所以时间复杂度上界可以写成 $O(2^n)$。

第三题解析

这段程序先读入一张有向图，`deg[v]` 表示点 v 当前剩余的入度。所有入度为 0 的点先进入队列。随后不断取出队首点 u，遍历从 u 出发的每条边 $u \to v$，把 `deg[v]` 减 1。若某个点的入度变为 0，它进入队列。

这正是拓扑排序的过程。若输入图是有向无环图，每个点最终都会入队并出队，因此 `cnt` 会等于 n。若图中存在环，环上的点可能一直无法变成入度为 0，因此 `cnt` 可能小于 n。

数组 `dp` 的含义是路径计数。初始 `dp[1]=1`，表示从点 1 到点 1 有一种空路径式的起始方式。处理边 $u \to v$ 时执行 `dp[v]+=dp[u]`，把所有到达 u 的路径接上一条边，变成到达 v 的路径。因此，在有向无环图中，`dp[n]` 表示从 1 到 n 的路径总数，而不是最短路径条数。

第 2 小题中，`sort(g[u].begin(), g[u].end())` 只改变同一个点的出边遍历顺序。对于有向无环图中的路径计数，每条边贡献一次 `dp[u]`，出边顺序不会改变最终结果。对于含环图，程序本身只处理能进入队列的点，排序仍不会改变被处理的边集合和加法结果。

第 4 小题的输入图中，从 1 到 5 的路径有：

```
1 -> 2 -> 5
1 -> 2 -> 4 -> 5
1 -> 3 -> 4 -> 5
```

共有 3 条。该图是有向无环图，所有 5 个点都会被处理，所以输出 `5 3`。

第 5 小题中，边为 $1 \to 2$、$2 \to 3$、$3 \to 2$。点 2 与点 3 构成环。初始只有点 1 的入度为 0，处理点 1 后，点 2 的入度仍不为 0，队列变空。此时只处理了 1 个点，`dp[3]` 仍为 0，所以输出 `1 0`。

时间复杂度中，读入和拓扑处理为 $O(n+m)$。排序部分是对每个点的出边分别排序，总开销不超过 $O(m \log m)$，所以整体上界可以写为 $O(n + m \log m)$。

0x3D 完善程序分类训练

本章训练完善程序。前面已经分别讲过初始化、边界、二分、排序、DP、图论、搜索和递归填空；这里不再重新讲知识点，而是按试卷中的完善程序形式做综合训练。

完善程序题的难点通常不在某一行语法，而在于判断这段代码到底想实现什么。读题时要先抓住程序框架：它是在二分答案、做状态转移、维护单调结构，还是在拓扑序上计数。看空缺时，再判断这一空影响的是初始化、边界、转移、更新、剪枝还是输出。

本章题目为原创训练题，题干和选项采用第一轮常见表述方式。

一、完善程序

第一题

（次短路）给定一个含 n 个点、m 条边的有向图，边权均为正整数。给定起点 s 和终点 t，求从 s 到 t 的次短路长度。这里的“次短路”指长度严格大于最短路长度的最短路径长度，路径中允许重复经过点和边。若不存在次短路，输出 `-1`。

以下程序使用 Dijkstra 的思想，同时维护每个点的最短距离和次短距离。试补全程序。

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

using ll = long long;

const int N = 200005;
const ll INF = (1LL << 60);

struct Edge {
    int to;
    int w;
};

int n, m, s, t;
vector<Edge> g[N];
ll d1[N], d2[N];

struct Node {
    ll d;
    int u;
    bool operator < (const Node &o) const {
        return d > o.d;
    }
};

void upd(int v, ll nd, priority_queue<Node> &q) {
    if (nd < d1[v]) {
        ①;
        d1[v] = nd;
        q.push({d1[v], v});
    } else if (②) {
        d2[v] = nd;
```

```
        q.push({d2[v], v});
    }
}

int main() {
    cin >> n >> m >> s >> t;

    for (int i = 1; i <= n; i++) {
        d1[i] = d2[i] = INF;
    }

    for (int i = 1; i <= m; i++) {
        int u, v, w;
        cin >> u >> v >> w;
        g[u].push_back({v, w});
    }

    priority_queue<Node> q;
    d1[s] = 0;
    q.push({0, s});

    while (!q.empty()) {
        Node now = q.top();
        q.pop();

        int u = now.u;
        ll dis = now.d;

        if (③) continue;

        for (auto e : g[u]) {
            int v = e.to;
            ll nd = dis + e.w;
            ④;
        }
    }

    if (⑤) cout << -1 << endl;
    else cout << d2[t] << endl;

    return 0;
}
```

1. ①处应填 ()

   A. `d2[v] = d1[v]`

   B. `d1[v] = d2[v]`

   C. `d2[v] = nd`

   D. `d1[v] = INF`

2. ②处应填 ()

   A. `nd <= d1[v] && nd < d2[v]`

   B. `nd > d1[v] && nd < d2[v]`

   C. `nd >= d1[v] && nd <= d2[v]`

   D. `nd < d1[v] || nd > d2[v]`

3. ③处应填（）

A. `dis > d2[u]`

B. `dis < d1[u]`

C. `dis == d1[u]`

D. `dis == d2[u]`

4. ④处应填（）

A. `upd(u, nd, q)`

B. `upd(v, nd, q)`

C. `d1[v] = min(d1[v], nd)`

D. `d2[u] = min(d2[u], nd)`

5. ⑤处应填（）

A. `d1[t] == INF`

B. `d2[t] == INF`

C. `d1[t] == d2[t]`

D. `d1[t] > d2[t]`

第二题

（所有子段最大值之和）给定一个长度为 n 的整数序列 $a_1, a_2, \ldots, a_n$，求所有非空连续子段的最大值之和。

例如，序列 $1, 2, 1$ 的非空连续子段为 $[1]$、$[2]$、$[1]$、$[1, 2]$、$[2, 1]$、$[1, 2, 1]$，最大值分别为 $1, 2, 1, 2, 2, 2$，答案为 10。

以下程序使用单调栈计算每个位置作为子段最大值时的贡献。为了避免相同数值被重复计算，程序约定：若一个子段中最大值出现多次，由最靠左的那个最大值代表这个子段。试补全程序。

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

using ll = long long;

const int N = 100005;

int n;
ll a[N];
int L[N], R[N], st[N], top;

int main() {
    cin >> n;
    for (int i = 1; i <= n; i++) cin >> a[i];

    top = 0;
    for (int i = 1; i <= n; i++) {
        while (top && ①) top--;
        if (top) L[i] = st[top];
        else L[i] = 0;
        st[++top] = i;
    }

    top = 0;
```

```
    for (int i = n; i >= 1; i--) {
        while (top && ②) top--;
        if (top) R[i] = st[top];
        else R[i] = n + 1;
        st[++top] = i;
    }

    ll ans = 0;
    for (int i = 1; i <= n; i++) {
        ll left = ③;
        ll right = ④;
        ans += ⑤;
    }

    cout << ans << endl;
    return 0;
}
```

1. ①处应填 ()

   A. `a[st[top]] < a[i]`

   B. `a[st[top]] <= a[i]`

   C. `a[st[top]] > a[i]`

   D. `a[st[top]] >= a[i]`

2. ②处应填 ()

   A. `a[st[top]] < a[i]`

   B. `a[st[top]] <= a[i]`

   C. `a[st[top]] > a[i]`

   D. `a[st[top]] >= a[i]`

3. ③处应填 ()

   A. `L[i]`

   B. `i - L[i]`

   C. `L[i] - i`

   D. `n - L[i]`

4. ④处应填 ()

   A. `R[i]`

   B. `R[i] - i`

   C. `i - R[i]`

   D. `R[i] - L[i]`

5. ⑤处应填 ()

   A. `a[i] + left + right`

   B. `a[i] * (left + right)`

C. `a[i] * left * right`

D. `left * right`

第三题

（字典序第 k 小路径）给定一张 n 个点 m 条边的有向无环图，顶点编号为 1 到 n。对于一条从点 1 出发的路径，定义它的路径序列为从起点开始依次经过的顶点编号构成的序列。路径可以只包含起点 1。

把所有从点 1 出发的路径序列按字典序从小到大排序，求第 k 小的路径序列。保证至少存在 k 条这样的路径。由于路径数量可能很大，程序中所有超过 10^{18} 的计数都用 10^{18} 表示。

例如，若从 1 出发的路径序列有：

```
1
1 2
1 2 4
1 3
```

则按字典序排列后，第 3 小路径为 `1 2 4`。

以下程序先用拓扑排序求出每个点出发的路径数量，再根据 k 逐步确定路径上的下一个点。试补全程序。

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

using ll = long long;

const int N = 100005;
const ll LIM = 1000000000000000000LL;

int n, m;
ll k, f[N];
int deg[N];
vector<int> g[N], ord;

ll add(ll x, ll y) {
    if (x + y >= LIM) return LIM;
    return x + y;
}

int choose_next(vector<int> &v, ll &k) {
    sort(v.begin(), v.end());
    for (int x : v) {
        if (①) return x;
        k -= f[x];
    }
    return -1;
}

int main() {
    cin >> n >> m >> k;

    for (int i = 1; i <= m; i++) {
        int u, v;
        cin >> u >> v;
        g[u].push_back(v);
        deg[v]++;
    }

    queue<int> q;
```

```
    for (int i = 1; i <= n; i++) {
        if (deg[i] == 0) q.push(i);
    }

    while (!q.empty()) {
        int u = q.front();
        q.pop();
        ord.push_back(u);

        for (int v : g[u]) {
            ②;
            if (deg[v] == 0) q.push(v);
        }
    }

    reverse(ord.begin(), ord.end());

    for (int u : ord) {
        f[u] = ③;
        for (int v : g[u]) {
            f[u] = ④;
        }
    }

    int u = 1;
    cout << u;

    while (⑤) {
        k--;
        u = choose_next(g[u], k);
        cout << " " << u;
    }

    cout << endl;
    return 0;
}
```

1. ①处应填 ()

   A. `k < f[x]`

   B. `k <= f[x]`

   C. `k == f[x]`

   D. `k > f[x]`

2. ②处应填 ()

   A. `deg[u]--`

   B. `deg[v]--`

   C. `deg[v]++`

   D. `deg[u]++`

3. ③处应填 ()

   A. `0`

   B. `1`

C. `f[u]`

D. `LIM`

4. ④处应填（）

A. `add(f[u], f[v])`

B. `add(f[u], f[v] + 1)`

C. `add(f[u] * f[v], 1)`

D. `min(f[u], f[v])`

5. ⑤处应填（）

A. `u != -1`

B. `k > 1`

C. `k > f[u]`

D. `g[u].empty()`

二、参考答案

第一题

小题	1	2	3	4	5
答案	A	B	A	B	B

第二题

小题	1	2	3	4	5
答案	A	B	B	B	C

第三题

小题	1	2	3	4	5
答案	B	B	B	A	B

三、解析

第一题解析

本题同时维护每个点的两类距离：`d1[v]` 表示当前找到的最短距离，`d2[v]` 表示当前找到的、严格大于 `d1[v]` 的最小距离。

当新的距离 `nd` 小于 `d1[v]` 时，原来的最短距离不应丢弃，而是成为当前候选的次短距离。因此先执行 `d2[v] = d1[v]`，再令 `d1[v] = nd`。这对应第 1 空。

如果 `nd` 不能成为新的最短距离，它仍可能成为新的次短距离。次短路要求长度严格大于最短路，所以判断条件应为 `nd > d1[v] && nd < d2[v]`。若写成 `nd >= d1[v]`，长度等于最短路的另一条路径会被误计为次短路；若写成 `nd <= d2[v]`，相同长度也可能重复入队。这对应第 2 空。

优先队列中可能保存已经失效的状态。当弹出的距离 `dis` 已经大于当前记录的 `d2[u]` 时，它既不是点 u 的最短距离，也不是次短距离，继续扩展不会得到更优答案。因此第 3 空应填 `dis > d2[u]`。

遍历边 $u \to v$ 时，新的路径长度为 `dis + e.w`，需要用它更新点 v 的最短距离和次短距离，所以第 4 空应填 `upd(v, nd, q)`。

程序结束后，若 `d2[t]` 仍为 `INF`，说明没有找到严格大于最短路的路径，应输出 `-1`；否则输出 `d2[t]`。这对应第 5 空。

第二题解析

本题用“每个位置作为最大值代表”的方式计算贡献。对位置 i，只要能确定它可以代表多少个连续子段，就能把这些子段的最大值贡献一次性加入答案。

题目约定：若一个子段中最大值出现多次，由最靠左的那个最大值代表。因此，位置 i 代表的子段必须满足两个条件：

第一，子段左端不能跨过左侧第一个大于等于 a_i 的位置。若左侧有一个值等于 a_i 的位置也被包含进来，那么最靠左的最大值就不是 i。所以 `L[i]` 应记录左侧第一个大于等于 a_i 的位置。

第二，子段右端可以跨过右侧等于 a_i 的位置，但不能跨过右侧第一个大于 a_i 的位置。右侧相等值不会改变“最靠左最大值”仍为 i；右侧更大的值会使最大值不再是 a_i。所以 `R[i]` 应记录右侧第一个大于 a_i 的位置。

第一遍从左到右维护单调栈。为了让栈顶成为左侧第一个大于等于当前值的位置，遇到小于当前值的元素就弹出，遇到等于当前值的元素则保留。因此第 1 空为：

```
a[st[top]] < a[i]
```

第二遍从右到左维护单调栈。为了让栈顶成为右侧第一个大于当前值的位置，遇到小于或等于当前值的元素都要弹出。因此第 2 空为：

```
a[st[top]] <= a[i]
```

确定边界后，左端点可以从 `L[i]+1` 到 `i` 中选择，共 `i - L[i]` 种；右端点可以从 `i` 到 `R[i]-1` 中选择，共 `R[i] - i` 种。位置 i 代表的子段数量为两者乘积，每个这样的子段最大值都是 a_i，所以贡献为：

```
a[i] * left * right
```

第 3、4、5 空分别为 `i - L[i]`、`R[i] - i`、`a[i] * left * right`。

以序列 $1, 2, 1$ 为例，位置 2 的值为 2，左端点可选 $1, 2$，右端点可选 $2, 3$，它代表 4 个子段，贡献为 8。两个值为 1 的位置分别只代表自己的单点子段，贡献合计为 2。总答案为 10。

第三题解析

设 `f[u]` 表示从点 u 出发的路径条数。由于路径可以只包含当前点，所以每个点至少对应一条路径 `[u]`，初始化应为 `f[u] = 1`。这对应第 3 空。

若存在一条边 $u \to v$，那么从 v 出发的每一条路径，都可以接在点 u 后面，形成一条从 u 出发、第二个点为 v 的路径。因此有：

```
f[u] = add(f[u], f[v]);
```

这对应第 4 空。函数 `add` 将超过 10^{18} 的计数截断为 10^{18}，因为题目只需要根据 k 比较路径数量，超过上界后无需保留精确值。

为了保证计算 f[u] 时，所有后继点的 f 值已经求出，程序先进行拓扑排序，再反转拓扑序。拓扑排序中处理边 $u \to v$ 时，应减少点 v 的剩余入度，所以第 2 空为 deg[v]--。

构造答案时，当前已经输出点 u。在所有从 u 出发的路径中，路径 [u] 本身排在以更长路径开头的方案之前。若当前 k == 1，答案就在此结束；若 k > 1，需要先跳过路径 [u]，所以循环条件为 k > 1，并在循环内执行 k--。这对应第 5 空。

接下来要选择路径的下一个点。函数 choose_next 将所有后继点按编号排序。对于某个后继点 x，以 x 为下一点的路径共有 f[x] 条。如果 k <= f[x]，答案的下一点就是 x；否则跳过这一整组，并令 k -= f[x]。所以第 1 空为 k <= f[x]。

这道题最容易出错的是把 f[u] 理解成“从 u 出发且至少走一条边的路径数”。程序中的 f[u] 包含路径 [u] 本身，因此后面构造答案时才需要在继续往后走之前执行 k--。

附录

附录 A Linux 命令

一、基础命令

命令	作用	备注
`pwd`	显示当前工作目录路径	CSP-S 2024 考过
`ls`	列出当前目录下的文件和子目录	CSP-S 2021 考过
`cd`	切换工作目录	CSP-S 2022 考过
`mkdir`	创建目录	CSP-S 2023 考过
`cp`	复制文件或目录	高频基础命令
`mv`	移动文件或重命名	高频基础命令
`rm`	删除文件或目录	高频基础命令
`cat`	显示文件内容	查看短文本
`head`	显示文件开头部分	查看前若干行
`tail`	显示文件末尾部分	查看后若干行
`wc`	统计行数、单词数、字节数	`wc -l` 表示统计行数
`chmod`	修改文件权限	常见于可执行权限

二、路径写法

写法	含义
`.`	当前目录
`..`	上一级目录
`~`	当前用户的主目录
`/`	根目录，或路径分隔符
`./main`	当前目录下的可执行文件 `main`
`a/b.cpp`	当前目录下 `a` 子目录中的 `b.cpp`
`/home/user/main.cpp`	从根目录开始写出的绝对路径

三、编译与运行

命令	作用
`g++ main.cpp`	编译 `main.cpp`，默认生成 `a.out`
`g++ main.cpp -o main`	编译 `main.cpp`，生成可执行文件 `main`
`g++ -o main main.cpp`	与上一条含义相同
`g++ -std=c++14 main.cpp -o main`	按 C++14 标准编译
`./main`	运行当前目录下的 `main`
`./main < in.txt`	从 `in.txt` 读入
`./main > out.txt`	将输出写入 `out.txt`，覆盖原内容
`./main >> out.txt`	将输出追加到 `out.txt` 末尾
`./main < in.txt > out.txt`	从 `in.txt` 读入，将输出写入 `out.txt`
`time ./main`	运行程序并显示耗时信息

四、查找与比较

命令	作用
`grep "abc" a.txt`	在 `a.txt` 中查找包含 `abc` 的行
`grep -n "abc" a.txt`	查找时显示行号
`find . -name "*.cpp"`	从当前目录开始查找所有 `.cpp` 文件
`diff a.out a.ans`	比较两个文件的差异
`sort a.txt`	按行排序
`uniq a.txt`	去掉相邻重复行

五、重定向与连接

写法	含义
`<`	输入重定向
`>`	输出重定向，覆盖原文件
`>>`	输出重定向，追加到文件末尾
`2>`	错误信息重定向
`&&`	前一个命令成功后，再执行后一个命令
`;`	依次执行两个命令，不要求前一个命令成功

六、time 输出

字段	含义
`real`	程序从开始到结束经过的实际时间
`user`	程序在用户态执行所用时间
`sys`	程序在系统态执行所用时间

七、易混对照

写法	注意点
`ls`	列出目录内容，不是切换目录
`cd`	切换目录，不是复制文件
`mkdir`	创建目录，不是创建普通文件
`cp`	复制后源文件仍保留
`mv`	移动或改名后，原位置不再保留同名文件
`rm`	删除文件或目录
`cat`	显示文件内容
`grep`	查找包含指定内容的行
`find`	按条件查找文件
`diff`	比较文件差异
`chmod`	修改权限，不负责编译
`g++ -o main main.cpp`	`main` 是输出文件名，`main.cpp` 是源文件
`g++ main.cpp -o main.cpp`	输出文件名也是 `main.cpp`，可能覆盖源文件，应避免

八、最小记忆表

命令	记忆
`pwd`	当前位置
`ls`	列目录
`cd`	换目录
`mkdir`	建目录
`cp`	复制
`mv`	移动或改名
`rm`	删除
`cat`	看文件
`grep`	找文本
`find`	找文件
`diff`	比文件
`g++`	编译
`./main`	运行
`time`	计时
`chmod`	改权限

附录 B C++ 运算符优先级速查

一、优先级总表

优先级	运算符	含义	结合方向
1	() 、[] 、. 、->	函数调用、下标、成员访问	从左到右
2	++ 、–	后置自增、后置自减	从左到右
3	++ 、– 、+ 、- 、! 、~ 、* 、& 、sizeof	前置自增自减、单目正负、逻辑非、按位取反、解引用、取地址、求大小	从右到左
4	* 、/ 、%	乘、除、取模	从左到右
5	+ 、-	加、减	从左到右
6	<< 、>>	左移、右移	从左到右
7	< 、<= 、> 、>=	大小比较	从左到右
8	== 、!=	相等、不等	从左到右
9	&	按位与	从左到右
10	^	按位异或	从左到右
11	|	按位或	从左到右
12	&&	逻辑与	从左到右
13	||	逻辑或	从左到右
14	?:	条件运算符	从右到左
15	= 、+= 、-= 、*= 、/= 、%= 、<<= 、>>=	赋值与复合赋值	从右到左
16	,	逗号运算符	从左到右

二、常用优先级链

顺序	运算类型
1	括号、下标、成员访问
2	单目运算
3	乘除取模
4	加减
5	移位
6	大小比较
7	相等比较
8	按位与
9	按位异或
10	按位或
11	逻辑与
12	逻辑或
13	条件运算符
14	赋值
15	逗号

三、易错表达式

表达式	实际理解
a + b * c	a + (b * c)
a << b + c	a << (b + c)
a + b < c	(a + b) < c
a == b < c	a == (b < c)
a & b == c	a & (b == c)
a ^ b == c	a ^ (b == c)
a | b == c	a | (b == c)
a && b == c	a && (b == c)
a || b && c	a || (b && c)
a = b = c	a = (b = c)
a ? b : c ? d : e	a ? b : (c ? d : e)

四、逻辑运算与位运算

运算符	名称	运算对象	结果特点
!	逻辑非	条件真假	结果为 true 或 false
&&	逻辑与	条件真假	左右都真才为真
||	逻辑或	条件真假	至少一个为真就为真
~	按位取反	整数二进制位	每一位取反
&	按位与	整数二进制位	每一位分别与
^	按位异或	整数二进制位	相同为 0，不同为 1
|	按位或	整数二进制位	每一位分别或

五、自增自减

表达式	含义
`++i`	先自增，再使用新值
`i++`	先使用旧值，再自增
`--i`	先自减，再使用新值
`i--`	先使用旧值，再自减

六、赋值与比较

写法	含义
`a = b`	赋值
`a == b`	判断是否相等
`a != b`	判断是否不等
`a += b`	等价于 `a = a + b`
`a -= b`	等价于 `a = a - b`
`a *= b`	等价于 `a = a * b`
`a /= b`	等价于 `a = a / b`
`a %= b`	等价于 `a = a % b`

七、考试速记

内容	记法
算术高于比较	a + b < c 先算加法
比较高于逻辑	a < b && c < d 先算两个比较
相等比较高于按位与、异或、或	a & b == c 不是先算按位与
逻辑与高于逻辑或	a && b || c 先算逻辑与
赋值从右向左	a = b = c 先算 b = c
条件运算符从右向左	a ? b : c ? d : e 先看右侧条件表达式
逗号优先级最低	a = 1, 2 先完成赋值，再算逗号表达式
不确定就加括号	读程序时先按优先级补括号

附录 C 复杂度速查

一、常见复杂度大小

复杂度	增长速度	常见来源
$O(1)$	常数级	直接访问数组元素、一次赋值、一次比较
$O(\log n)$	对数级	二分、倍增、堆的一次上浮或下沉
$O(\sqrt{n})$	根号级	枚举因数、试除判断素数
$O(n)$	线性级	扫描数组、单次 DFS、单次 BFS
$O(n \log n)$	线性对数级	归并排序、堆排序、一般比较排序的常见上界
$O(n^2)$	平方级	两层循环、简单 DP、枚举点对
$O(n^3)$	立方级	三层循环、Floyd、区间 DP 常见转移
$O(2^n)$	指数级	枚举子集、朴素递归搜索
$O(n!)$	阶乘级	枚举全排列

二、数据规模与可接受复杂度

数据规模	常见可接受复杂度
$n \le 10$	$O(n!)$、$O(2^n n)$
$n \le 20$	$O(2^n n)$
$n \le 30$	折半搜索、指数级需谨慎
$n \le 10^2$	$O(n^3)$
$n \le 10^3$	$O(n^2)$、较轻的 $O(n^2 \log n)$
$n \le 10^5$	$O(n \log n)$、$O(n)$
$n \le 10^6$	$O(n)$、较轻的 $O(n \log n)$
$n \le 10^9$	$O(\log n)$、$O(\sqrt{n})$
$n \le 10^{18}$	$O(\log n)$

三、循环复杂度速查

代码结构	复杂度
`for (int i = 1; i <= n; i++)`	$O(n)$
`for (int i = 1; i <= n; i += 2)`	$O(n)$
`for (int i = 1; i <= n; i *= 2)`	$O(\log n)$
`for (int i = n; i >= 1; i /= 2)`	$O(\log n)$
两层循环都从 1 到 n	$O(n^2)$
三层循环都从 1 到 n	$O(n^3)$
外层 n 次，内层 $\log n$ 次	$O(n \log n)$
外层 n 次，内层最多移动总计 n 次	$O(n)$

四、递归复杂度速查

递归式	复杂度
$T(n) = T(n-1) + O(1)$	$O(n)$
$T(n) = T(n-1) + O(n)$	$O(n^2)$
$T(n) = T(n/2) + O(1)$	$O(\log n)$
$T(n) = T(n/2) + O(n)$	$O(n)$
$T(n) = 2T(n/2) + O(1)$	$O(n)$
$T(n) = 2T(n/2) + O(n)$	$O(n \log n)$
$T(n) = 2T(n/2) + O(n^2)$	$O(n^2)$
$T(n) = 3T(n/2) + O(n)$	$O(n^{\log_2 3})$
$T(n) = T(n-1) + T(n-2) + O(1)$	指数级

五、分治主定理速查

设 $T(n) = aT(n/b) + O(n^c)$，其中 $a \geq 1$，$b > 1$。

比较关系	复杂度
$a < b^c$	$O(n^c)$
$a = b^c$	$O(n^c \log n)$
$a > b^c$	$O(n^{\log_b a})$

常见例子：

递归式	对应结果
$T(n) = 2T(n/2) + O(n)$	$O(n \log n)$
$T(n) = 2T(n/2) + O(1)$	$O(n)$
$T(n) = 2T(n/2) + O(n^2)$	$O(n^2)$
$T(n) = 4T(n/2) + O(n)$	$O(n^2)$
$T(n) = 4T(n/2) + O(n^2)$	$O(n^2 \log n)$

六、常见算法复杂度

算法或操作	时间复杂度
顺序查找	$O(n)$
二分查找	$O(\log n)$
冒泡排序	$O(n^2)$
插入排序	$O(n^2)$
选择排序	$O(n^2)$
归并排序	$O(n \log n)$
堆排序	$O(n \log n)$
快速排序平均情况	$O(n \log n)$
快速排序最坏情况	$O(n^2)$
DFS 遍历图	$O(n+m)$
BFS 遍历图	$O(n+m)$
拓扑排序	$O(n+m)$
Dijkstra 朴素写法	$O(n^2+m)$
Dijkstra 堆优化	$O((n+m)\log n)$
Floyd	$O(n^3)$
Kruskal	$O(m \log m)$
Prim 朴素写法	$O(n^2)$
KMP	$O(n+m)$
ST 表预处理	$O(n \log n)$
ST 表单次查询	$O(1)$

七、常见数据结构操作

数据结构	操作	时间复杂度
数组	按下标访问	$O(1)$
数组	插入或删除中间元素	$O(n)$
栈	入栈、出栈、取栈顶	$O(1)$
队列	入队、出队、取队首	$O(1)$
二叉堆	插入	$O(\log n)$
二叉堆	删除堆顶	$O(\log n)$
二叉堆	取堆顶	$O(1)$
哈希表	平均查找、插入	$O(1)$
哈希表	最坏查找、插入	$O(n)$
二叉搜索树	平衡时查找、插入	$O(\log n)$
二叉搜索树	退化时查找、插入	$O(n)$
线段树	单点修改、区间查询	$O(\log n)$
树状数组	单点修改、前缀查询	$O(\log n)$
Trie	插入或查询字符串	$O(L)$

其中 L 表示字符串长度。

八、复杂度化简

原式	常见写法
$O(3n+5)$	$O(n)$
$O(n+n\log n)$	$O(n\log n)$
$O(n^2+n\log n)$	$O(n^2)$
$O(100n)$	$O(n)$
$O(\log_2 n)$	$O(\log n)$
$O(\log n+\log m)$	$O(\log n+\log m)$
$O(nm+n)$	$O(nm)$
$O(n^2m+nm^2)$	保留为 $O(n^2m+nm^2)$

附录 D 数据结构速查

一、线性结构

数据结构	核心特点	常见操作	复杂度
数组	连续存储，支持下标访问	访问 `a[i]`	$O(1)$
数组	中间插入、删除需要移动元素	插入、删除	$O(n)$
栈	后进先出	入栈、出栈、取栈顶	$O(1)$
队列	先进先出	入队、出队、取队首	$O(1)$
双端队列	两端都能插入和删除	队首、队尾操作	$O(1)$
链表	不支持快速下标访问	已知位置插入、删除	$O(1)$
链表	需要从头查找位置	查找第 k 个元素	$O(n)$

二、栈与队列

结构	插入位置	删除位置	典型用途
栈	栈顶	栈顶	括号匹配、递归过程、单调栈
队列	队尾	队首	BFS、拓扑排序
双端队列	队首或队尾	队首或队尾	单调队列、滑动窗口
优先队列	按优先级维护	取出最大或最小元素	堆、Dijkstra、合并问题

三、堆

操作	含义	复杂度
建堆	将一组元素整理成堆	$O(n)$
插入元素	新元素上浮	$O(\log n)$
删除堆顶	堆尾补到堆顶后下沉	$O(\log n)$
查看堆顶	取最大值或最小值	$O(1)$

类型	堆顶含义
大根堆	堆顶是当前最大值
小根堆	堆顶是当前最小值

C++ 的 `priority_queue<int>` 默认是大根堆。

四、哈希表

内容	速查
核心思想	用哈希函数把关键字映射到表中位置
常见冲突处理	拉链法、开放地址法
线性探查	冲突后依次向后找空位置
平均查找复杂度	$O(1)$
最坏查找复杂度	$O(n)$
装载因子	已存元素数与表长的比值
装载因子过高	冲突增多，查找效率下降

五、二叉树

概念	含义
根结点	没有父亲的结点
叶结点	没有孩子的结点
深度	从根到该结点经过的层数
高度	从该结点到最深叶结点的层数
满二叉树	每一层结点数都达到最大
完全二叉树	除最后一层外都满，最后一层从左到右排列
二叉搜索树	左子树关键字小于根，右子树关键字大于根

六、二叉树遍历

遍历方式	顺序
前序遍历	根、左、右
中序遍历	左、根、右
后序遍历	左、右、根
层序遍历	从上到下、从左到右

已知信息	能否确定二叉树
前序和中序	通常可以确定
后序和中序	通常可以确定
前序和后序	通常不能唯一确定
只有一种遍历	通常不能唯一确定

七、二叉搜索树

操作	平衡时复杂度	退化时复杂度
查找	$O(\log n)$	$O(n)$
插入	$O(\log n)$	$O(n)$
删除	$O(\log n)$	$O(n)$

性质	说明
中序遍历	得到递增序列
最小值	一直向左走
最大值	一直向右走
退化情况	插入序列接近有序时，可能退化成链

八、Trie

内容	速查
适用对象	字符串集合
核心思想	相同前缀共用结点
插入复杂度	$O(L)$
查询复杂度	$O(L)$
结点数上界	所有字符串长度之和加 1
常见用途	前缀查询、字符串集合判重

其中 L 表示字符串长度。

九、线段树

内容	速查
维护对象	区间信息
常见信息	区间和、区间最大值、区间最小值
建树复杂度	$O(n)$
单点修改	$O(\log n)$
区间查询	$O(\log n)$
常用数组空间	约 $4n$
根结点区间	整个序列
左右儿子区间	将当前区间从中点分开

十、树状数组

内容	速查
维护对象	前缀信息
常见信息	前缀和
单点修改	$O(\log n)$
前缀查询	$O(\log n)$
区间和查询	两个前缀和相减
核心写法	`lowbit(x)` 取出最低位的 1 对应的值

十一、ST 表

内容	速查
适用问题	静态区间最值查询
是否支持修改	不支持普通动态修改
预处理复杂度	$O(n \log n)$
查询复杂度	$O(1)$
常见维护	最大值、最小值、gcd
关键思想	用长度为 2^k 的区间拼出查询区间

十二、并查集

内容	速查
维护对象	若干元素所属集合
`find(x)`	找到 x 所在集合的代表
`merge(x,y)`	合并 x 与 y 所在集合
路径压缩	让以后查找更快
按大小或秩合并	避免树过深
常见用途	连通性判断、Kruskal

十三、常见选择

需求	常用结构
按下标快速访问	数组
后进先出	栈
先进先出	队列
两端插入删除	双端队列
随时取最大或最小	堆
快速判断元素是否出现	哈希表
维护有序集合	二叉搜索树或平衡树
查询字符串前缀	Trie
动态区间查询与修改	线段树
前缀和动态维护	树状数组
静态区间最值查询	ST 表
判断连通块	并查集

十四、易错提醒

情况	注意
栈	出栈顺序不一定是入栈顺序的逆序，要看中途是否穿插出栈
队列	BFS 中每个点通常入队一次
堆	堆中只能保证堆顶最优，不能保证整体有序
哈希表	平均 $O(1)$ 不代表最坏 $O(1)$
二叉搜索树	中序遍历才是有序序列
Trie	根结点也要计入结点总数
线段树	查询区间可能被拆成多个结点区间
ST 表	适合静态查询，不适合频繁修改
并查集	只能快速判断集合关系，不能直接给出两点之间路径

附录 E 图论速查

一、基本概念

概念	含义
图	由点和边组成，常记为 $G=(V,E)$
点	图中的对象，也叫顶点
边	连接两个点的关系
无向边	两个方向都可以走
有向边	只能按指定方向走
边权	边上的数值，常表示距离、费用、时间
简单图	没有自环和重边的图
自环	一条边从某个点连向它自己
重边	两个点之间有多条边
完全图	任意两点之间都有边
稀疏图	边数与点数同阶，常见为 $m=O(n)$
稠密图	边数接近点数平方级，常见为 $m=O(n^2)$

二、度数

概念	适用图	含义
度数	无向图	与该点相连的边数
入度	有向图	指向该点的边数
出度	有向图	从该点指出的边数
孤立点	无向图或有向图	度数为 0 的点
叶结点	树	度数为 1 的非根结点，或没有儿子的结点

无向图中所有点度数之和等于 $2m$。
有向图中所有点入度之和等于所有点出度之和，且都等于 m。

三、路径与环

概念	含义
路径	从一个点沿边走到另一个点经过的点边序列
路径长度	可以指边数，也可以指边权和，要看题意
简单路径	路径中不重复经过点
环	从某个点出发，沿若干条边又回到该点
简单环	除起点和终点相同外，其余点不重复
无环图	不含环的图
DAG	有向无环图

DAG 常与拓扑排序、DP、路径计数一起出现。

四、存图方式

存图方式	空间复杂度	适合情况
邻接矩阵	$O(n^2)$	点数较小，或需要快速判断两点是否有边
邻接表	$O(n+m)$	稀疏图，遍历边更方便
边集数组	$O(m)$	Kruskal、按边排序
前向星	$O(n+m)$	静态数组存图，适合竞赛代码

五、遍历

算法	常见用途	时间复杂度
DFS	连通性、搜索、树上递归、拓扑相关	$O(n+m)$
BFS	最少边数路径、层次遍历、拓扑排序	$O(n+m)$

题目现象	常用判断
求从起点能到哪些点	DFS 或 BFS
无权图最短路	BFS
遍历所有点和边	$O(n+m)$
判断图是否连通	从一个点遍历，看是否访问所有点
判断是否存在环	DFS 或拓扑排序

六、连通与强连通

概念	适用图	含义
连通	无向图	任意两点之间都有路径
连通块	无向图	极大的连通部分
强连通	有向图	任意两点之间互相可达
强连通分量	有向图	极大的强连通部分
弱连通	有向图	忽略边方向后连通

判断对象	常用方法
无向图连通块	DFS 或 BFS
有向图强连通分量	Tarjan 或 Kosaraju
有向图是否强连通	所有点在同一个强连通分量中
缩点后的图	一定是 DAG

七、树

性质	内容
点数与边数	n 个点的树有 $n-1$ 条边
连通性	树是连通图
环	树中没有环
两点路径	树上任意两点之间有且只有一条简单路径
根	指定根后，树可以看成父子结构
子树	某个结点及其所有后代构成的部分
LCA	两个结点的最近公共祖先

等价判断：

条件	是否能说明是树
连通且有 $n-1$ 条边	是
无环且有 $n-1$ 条边	是
连通且无环	是
任意两点之间有唯一简单路径	是

八、二分图

概念	含义
二分图	点集能分成两部分，每条边都连接两部分中的点
染色判断	可以用两种颜色染点，使每条边两端颜色不同
奇环	长度为奇数的环
判定性质	无向图是二分图，当且仅当没有奇环

常见判断：

情况	结论
树	一定是二分图
偶环	是二分图
奇环	不是二分图
完全图 K_3	不是二分图
完全二分图	是二分图

九、欧拉图

概念	含义
欧拉路径	经过每条边恰好一次的路径
欧拉回路	经过每条边恰好一次，并回到起点
欧拉图	存在欧拉回路的图

无向图常用判定：

条件	结论
连通且所有点度数为偶数	存在欧拉回路
连通且恰有 2 个奇度点	存在欧拉路径，但不存在欧拉回路
奇度点个数不是 0 或 2	不存在欧拉路径

有向图常用判定：

条件	结论
所有点入度等于出度，并且相关点连通	存在欧拉回路
起点出度比入度多 1，终点入度比出度多 1，其余点入度等于出度	存在欧拉路径

十、拓扑排序

内容	速查
适用图	DAG
常用方法	反复取入度为 0 的点
时间复杂度	$O(n+m)$
是否唯一	不一定唯一
有环时	不能完成包含所有点的拓扑排序
常见用途	先后依赖关系、课程安排、DAG 上 DP

拓扑排序步骤：

步骤	操作
1	统计每个点入度
2	将所有入度为 0 的点加入队列
3	每次取出一个点，并删除它指出的边
4	若某个点入度变为 0，加入队列
5	若最终取出的点数小于 n，说明有环

十一、最小生成树

内容	速查
适用图	连通无向带权图
目标	用 $n-1$ 条边连通所有点，使边权和最小
Kruskal	按边权从小到大选边，用并查集判环
Prim	从一个点出发，不断加入距离当前集合最近的点
树边数	必为 $n-1$
图不连通	不存在覆盖所有点的生成树

算法	常见复杂度
Kruskal	$O(m\log m)$
Prim 朴素写法	$O(n^2)$
Prim 堆优化	$O(m\log n)$

十二、最短路

算法	适用情况	常见复杂度
BFS	无权图，或所有边权相同	$O(n+m)$
Dijkstra	边权非负	$O((n+m)\log n)$
Floyd	多源最短路，点数较小	$O(n^3)$
Bellman-Ford	可处理负边	$O(nm)$

题目条件	常用选择
无权图最少边数	BFS
正边权或非负边权	Dijkstra
任意两点最短路，n 较小	Floyd
出现负边	Bellman-Ford 或 SPFA 思想
状态中有额外条件	状态最短路

十三、图论易错提醒

情况	注意
有向图连通	要看题目说的是强连通还是弱连通
拓扑排序	只适用于 DAG
二分图	不是“分成两块”就行，还要所有边都跨两边
欧拉路径	看的是边是否恰好经过一次，不是点
最小生成树	只用于无向连通图
最短路	Dijkstra 不能直接处理负边
树	n 个点、$n-1$ 条边还需配合连通或无环判断
完全图边数	无向完全图有 $\frac{n(n-1)}{2}$ 条边
有向完全图边数	若每对点两个方向都有边，则有 $n(n-1)$ 条边

附录 F 数学公式速查

一、等差数列

内容	公式
通项公式	$a_n = a_1 + (n-1)d$
前 n 项和	$S_n = \frac{n(a_1+a_n)}{2}$
前 n 项和	$S_n = na_1 + \frac{n(n-1)}{2}d$
中项关系	若 a, b, c 成等差，则 $2b = a + c$

二、等比数列

内容	公式
通项公式	$a_n = a_1 q^{n-1}$
前 n 项和，$q \neq 1$	$S_n = \frac{a_1(1-q^n)}{1-q}$
前 n 项和，$q = 1$	$S_n = na_1$
中项关系	若 a, b, c 成等比，则 $b^2 = ac$

三、常见求和

内容	公式
前 n 个正整数和	$1 + 2 + \cdots + n = \frac{n(n+1)}{2}$
前 n 个正整数平方和	$1^2 + 2^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}$
前 n 个正整数立方和	$1^3 + 2^3 + \cdots + n^3 = \left(\frac{n(n+1)}{2}\right)^2$
奇数和	$1 + 3 + \cdots + (2n-1) = n^2$
偶数和	$2 + 4 + \cdots + 2n = n(n+1)$
二进制权值和	$1 + 2 + 4 + \cdots + 2^{n-1} = 2^n - 1$

四、排列组合

内容	公式
排列数	$A_n^m = \frac{n!}{(n-m)!}$
组合数	$C_n^m = \frac{n!}{m!(n-m)!}$
组合对称性	$C_n^m = C_n^{n-m}$
组合递推	$C_n^m = C_{n-1}^m + C_{n-1}^{m-1}$
全排列	$n!$
圆排列	$(n-1)!$
不相邻选法	从 n 个位置选 k 个且互不相邻，有 C_{n-k+1}^k 种
可重组合	n 类物品选 k 个，允许重复，有 C_{n+k-1}^k 种

五、二项式

内容	公式
二项式定理	$(a+b)^n=\sum_{k=0}^{n}C_n^k a^{n-k}b^k$
组合数总和	$\sum_{k=0}^{n}C_n^k=2^n$
奇偶项和	$\sum_{k\text{ 为偶数}}C_n^k=\sum_{k\text{ 为奇数}}C_n^k=2^{n-1}$
带符号和	$\sum_{k=0}^{n}(-1)^kC_n^k=0$

六、容斥原理

情况	公式或方法
两个集合	$\lvert A\cup B\rvert=\lvert A\rvert+\lvert B\rvert-\lvert A\cap B\rvert$
三个集合	$\lvert A\cup B\cup C\rvert=\lvert A\rvert+\lvert B\rvert+\lvert C\rvert-\lvert A\cap B\rvert-\lvert A\cap C\rvert-\lvert B\cap C\rvert+\lvert A\cap B\cap C\rvert$
至少满足一个条件	总数减去一个都不满足的数量
一个都不满足	总数减去至少满足一个条件的数量
不能被 a,b,c 整除	总数减去能被 a,b,c 中至少一个整除的数量
同时满足多个限制	先加单个条件数量，再减两个条件交集，再加三个条件交集

七、概率

内容	公式
古典概率	$P(A)=\frac{A\text{ 发生的基本事件数}}{\text{基本事件总数}}$
对立事件	$P(\overline{A})=1-P(A)$
加法公式	$P(A\cup B)=P(A)+P(B)-P(A\cap B)$
互斥事件	若 A,B 互斥，则 $P(A\cup B)=P(A)+P(B)$
独立事件	若 A,B 独立，则 $P(A\cap B)=P(A)P(B)$
期望定义	$E(X)=\sum x_iP(X=x_i)$
期望线性性	$E(X+Y)=E(X)+E(Y)$

八、二项分布

内容	公式
恰好成功 k 次	$P(X=k)=C_n^kp^k(1-p)^{n-k}$
期望	$E(X)=np$
失败概率	若成功概率为 p，失败概率为 $1-p$

二项分布常见条件：重复进行 n 次独立试验，每次只有成功和失败两种结果，每次成功概率相同。

九、进制与数位

内容	公式或规则
b 进制展开	$(x_k x_{k-1} \cdots x_0)_b = \sum_{i=0}^{k} x_i b^i$
二进制第 i 位权值	2^i
十进制第 i 位权值	10^i
十六进制第 i 位权值	16^i
十六进制数字	$A=10, B=11, C=12, D=13, E=14, F=15$
十进制取个位	$n \bmod 10$
十进制去掉个位	$\lfloor n/10 \rfloor$
数位和	逐位相加

十、模运算

内容	公式
加法	$(a+b) \bmod m = ((a \bmod m) + (b \bmod m)) \bmod m$
减法	$(a-b) \bmod m = ((a \bmod m) - (b \bmod m) + m) \bmod m$
乘法	$(ab) \bmod m = ((a \bmod m)(b \bmod m)) \bmod m$
幂	$a^b \bmod m$ 常用快速幂计算
同余	$a \equiv b \pmod m$ 表示 m 整除 $a-b$
逆元	若 $ax \equiv 1 \pmod m$，则 x 是 a 在模 m 意义下的逆元
逆元存在条件	$\gcd(a, m) = 1$

十一、gcd 与 lcm

内容	公式
最大公约数	$\gcd(a, b)$
最小公倍数	$\operatorname{lcm}(a, b)$
基本关系	$\gcd(a, b) \operatorname{lcm}(a, b) = ab$
欧几里得算法	$\gcd(a, b) = \gcd(b, a \bmod b)$
互质	$\gcd(a, b) = 1$
多个数 gcd	$\gcd(a, b, c) = \gcd(\gcd(a, b), c)$
多个数 lcm	$\operatorname{lcm}(a, b, c) = \operatorname{lcm}(\operatorname{lcm}(a, b), c)$

十二、图论计数

图	边数
n 个点的树	$n-1$
n 个点的无向完全图	$\frac{n(n-1)}{2}$
n 个点的有向完全图，双向都有边	$n(n-1)$
二分图两侧大小分别为 a, b 的完全二分图	ab
无向图度数和	$2m$
有向图入度和	m
有向图出度和	m

十三、常见边界

内容	数值
空集子集数	1
n 元集合子集数	2^n
长度为 n 的 01 串数量	2^n
长度为 n 的十进制串数量	10^n
0!	1
C_n^0	1
C_n^n	1
A_n^0	1
a^0，$a \neq 0$	1

十四、易错提醒

情况	注意
排列	选出的元素有顺序
组合	选出的元素无顺序
可重复选择	通常不能直接用 C_n^k
至少一个	可以用总数减去一个都没有
不能相邻	常用插空或 C_{n-k+1}^k
模除法	不能直接除，要考虑逆元
逆元	只有互质时一定存在
gcd 和 lcm	先乘可能溢出，计算时常先除后乘
概率平均收益	常转成期望求和
树的边数	$n-1$ 是树的必要性质，也要注意连通或无环条件

附录 G 考前检查表

一、考前准备

项目	检查内容	完成
证件	准考证、身份证件按当地要求准备	□
文具	黑色签字笔、2B 铅笔、橡皮	□
时间	确认考试日期、入场时间、考点位置	□
作答方式	确认是在答题纸还是答题卡上作答	□
禁带物品	不携带电子设备、资料、草稿纸等违规物品	□
状态	考前一天不熬夜，不临时刷大量新题	□

二、开考后先检查

项目	检查内容	完成
试卷	页数是否完整，印刷是否清楚	□
答题纸	姓名、准考证号等信息是否填写正确	□
题型	确认选择题、程序阅读、完善程序的位置	□
时间	心里分配好每部分大致用时	□
标记	遇到暂时不会的题先标记，不长时间卡住	□

三、选择题检查

项目	检查内容	完成
Linux 命令	`pwd`、`ls`、`cd`、`mkdir`、`g++` 等命令用途	□
进制数	二进制、十六进制、数位权值是否算对	□
复杂度	循环次数、递归式、排序复杂度是否判断正确	□
排列组合	是否区分排列与组合，是否考虑前导零	□
概率期望	是否把“平均收益”转成期望计算	□
树与图	连通、强连通、DAG、二分图、欧拉图概念是否混淆	□
数据结构	栈、队列、堆、哈希表、BST、Trie 的性质是否清楚	□
字符串	KMP、LCS、字符串哈希、编码过程是否看清定义	□
运算符	位运算、逻辑运算、比较运算优先级是否补括号	□
边界条件	题目问“最多”“最少”“一定”“不一定”是否看清	□

四、程序阅读检查

项目	检查内容	完成
输入	输入格式、变量初值是否看清	□
输出	输出几行、是否有空格或换行	□
全局变量	全局数组和变量默认初值是否利用	□
循环边界	起点、终点、是否包含端点	□
判断条件	`if`、`while` 的条件是否逐项看完	□
数组下标	从 `0` 开始还是从 `1` 开始	□
递归出口	递归何时返回，返回后是否继续循环	□
变量变化	关键变量是否列出追踪过程	□
函数作用	每个函数的输入、输出、修改了哪些变量	□
修改语句	删除或替换某一行后，影响的是哪一部分流程	□

五、完善程序检查

项目	检查内容	完成
题意	程序要解决的问题是否先读懂	□
变量	每个变量、数组的含义是否明确	□
空缺位置	空缺处是初始化、判断、转移、更新还是输出	□
上下文	空缺前后变量是否能对应上	□
边界	起点、终点、空区间、单点情况是否处理	□
选项排除	是否用类型、下标、变量含义排除错误项	□
算法意图	看出是二分、DP、图论、搜索还是排序流程	□
返回值	函数最后应返回什么，是否与题意一致	□
更新方向	DP 或最短路中是从旧状态更新新状态，还是反过来	□
输出	最终输出的是答案值、编号、路径还是判断结果	□

六、常见易错项

易错项	检查内容	完成
前导零	组成整数时，最高位不能为 `0`	□
取模	减法后可能为负，要补模	□
除法	模意义下不能随意除，要看逆元是否存在	□
哈希表	线性探查冲突后要继续往后找空位	□
堆	堆顶最优，不代表数组整体有序	□
BST	中序遍历有序，不是前序或后序有序	□
Trie	根结点也要计入结点数	□
二分	判断区间是闭区间还是左闭右开区间	□
图论	有向图要区分可达、弱连通、强连通	□
拓扑排序	有环图不能得到完整拓扑序	□
欧拉图	欧拉问题看边，不是看点	□
最短路	Dijkstra 需要非负边权	□
DP	状态含义不清时不要直接套转移式	□
搜索	回溯后是否恢复标记或变量	□
字符串	子串连续，子序列不要求连续	□

七、交卷前复查

项目	检查内容	完成
答案位置	是否全部写在规定答题区域	□
题号	答案是否和题号一一对应	□
选择题	是否漏选、错位、涂错	□
判断题	正误标记是否符合试卷要求	□
程序阅读	多小题答案是否都已填写	□
完善程序	每个空是否只选了一个选项	□
草稿推导	有疑问的计算题是否重新验算一次	□
极端词	“一定”“不一定”“最多”“最少”是否重新确认	□
改答案	修改过的题是否在答题纸上同步修改	□
时间	留出最后几分钟检查答题卡或答题纸	□

八、最后一遍快速确认

题型	最后确认
选择题	不空题，不错位
程序阅读	输入、循环、递归、数组变化再看一遍
完善程序	空缺处与上下文变量一致
计算题	小数字手算再验一次
图表题	点数、边数、入度、出度不要抄错
代码题	下标、边界、初值、返回值重点检查
答题纸	所有答案填在有效区域内

A GUIDE FOR CSP-S ROUND 1

关于本书

本书面向 CSP-S 第一轮备考，围绕选择题、程序阅读与完善程序展开，系统梳理高频考点、易错方法与典型训练，帮助学生建立稳定的解题节奏，完成考前查漏补缺。

01 **考点结构**

复杂度、数学计数、数据结构、树图、字符串、DP 与搜索等模块化梳理。

02 **题型能力**

选择题、程序阅读、完善程序分层训练，强调读题、追踪与排除。

03 **考前复习**

附录速查、检查表与分类训练帮助学生完成临考前查漏补缺。