DAMONTHRONE / CSP-JROUND1

0x22

CSP-J 第一轮

入门组初赛专项教材

选择题 · 程序阅读 · 完善程序

J1

目录

第一编 考试方法 **1**

0x01 CSP-J 第一轮考什么 2

0x02 选择题怎么做 6

0x03 阅读程序怎么读 11

0x04 完善程序怎么补 18

第二编 C++ 与程序行为 **25**

0x05 类型与变量 26

0x06 表达式与运算符 34

0x07 分支与循环 41

0x08 函数与递归 48

0x09 数组与字符串 56

0x0A 指针、结构体与链表 63

第三编 数学与计算机常识 **71**

0x0B 进制与存储单位 72

0x0C 计数基础 77

0x0D 简单数论 84

0x0E 复杂度 92

0x0F 计算机常识 100

第四编 基础数据结构 **108**

0x10 栈 109

0x11 队列 116

0x12 表达式 124

0x13 树与二叉树 132

0x14 图的基本概念 139

0x15 哈夫曼编码 146

0x16 排序基础 153

第五编 基础算法 **161**
0x17 模拟 162
0x18 枚举 170
0x19 二分 178
0x1A 递归问题 187
0x1B 搜索 196
0x1C 简单动态规划 204
0x1D 字符串处理 213
0x1E 贪心 222
第六编 专项训练 **230**
0x1F 选择题分类训练 231
0x20 程序阅读分类训练 240
0x21 完善程序分类训练 248
0x22 综合卷复盘 256
附录 **270**
附录 A 运算符优先级 271
附录 B 进制与存储单位 275
附录 C 数据结构速查 278
附录 D 考前检查表 283

第一编 考试方法

0x01 CSP-J 第一轮考什么

CSP-J 第一轮考的不是能不能在电脑上写出完整程序，而是“能不能在纸面上读懂题、判断规则、追踪程序、补全关键语句”。平时写程序时，错误可以通过运行、调试、修改来发现；第一轮考试中，学生面对的是题面、代码和选项，所有判断都要靠纸面分析完成。

本章先建立整张试卷的地图：题型有哪些，每类题型考什么，需要哪些能力。后面的章节会分别讲选择题、阅读程序和完善程序的具体做法。

一、三类题型

CSP-J 第一轮通常由三部分组成：单项选择题、阅读程序、完善程序。三部分分值不同，考查方式也不同。

题型	常见形式	主要考查
单项选择题	给出题干和四个选项	概念、计算、数据结构、算法常识、C++ 规则
阅读程序	给出一段完整程序，再回答判断题和选择题	程序执行过程、变量变化、边界条件、输出结果
完善程序	给出问题说明和带空缺的程序，在选项中补全	根据上下文判断初始化、条件、转移、更新和输出

这三类题型的共同点是：题目一般不会要求学生现场写一份完整程序，而是要求学生判断一段代码或一个说法是否正确。也就是说，第一轮更重视“理解程序行为”的能力。

很多学生误以为选择题只靠记忆，阅读程序只靠手算，完善程序只靠猜模板。这样的做法很容易失分。第一轮真正需要的是把“知识点”和“程序过程”连起来：知道规则，还要能看出规则在代码中怎样起作用。

二、选择题考什么

单项选择题看起来分散，但并不是随意出题。它常考以下几类内容。

类型	常见考法	做题重点
C++ 基础	类型、变量、表达式、循环、函数、结构体、指针	看清语法规则和程序行为
数学与常识	进制、存储单位、计数、简单数论、复杂度	纸面计算要稳
数据结构	栈、队列、树、图、哈夫曼编码、排序	把定义转化为判断过程
算法基础	枚举、二分、递归、搜索、动态规划思想	理解算法在做什么，不背长模板
计算机常识	操作系统、编译器、编码、基础概念	分清概念边界

选择题中有一类题很容易被低估：它不是直接问定义，而是把定义放进一个小场景里。例如，栈的题目可能不问“栈是什么”，而是给出入栈顺序，要求判断某个出栈序列是否可能；二叉树的题目可能不问遍历定义，而是给出前序和中序，要求推出后序；进制题可能同时混合二进制、八进制、十六进制和十进制。

因此，选择题不能只背句子。背“栈是后进先出”还不够，必须会在具体序列中判断后进先出有没有被破坏；背“二分查找复杂度是 $O(\log n)$”还不够，必须会算 1000 个元素最多比较多少次；背 1MB = 1024KB 还不够，必须知道字节和二进制位之间还差一个 8 倍。

三、阅读程序考什么

阅读程序是第一轮中最需要耐心的一部分。题目会给出一段完整程序，后面跟若干判断题和单选题。程序可能很短，也可能包含函数、数组、字符串、递归、循环嵌套或简单算法。

阅读程序最常考的不是“这段程序像哪个模板”，而是下面几件事：

关注点	常见失分原因
输入输出	没看清输入格式，或者漏看输出中的空格、换行、小数位数
变量含义	只盯着变量名，不看它在循环中如何变化
循环边界	把 `<` 看成 `<=`，或把起点、终点看错
数组下标	混淆从 0 开始和从 1 开始
字符串长度	混淆字符位置和字符串长度
函数返回值	只看函数名，不追踪 `return` 的结果
递归过程	只看最后一层，不看递归出口和回溯结果
修改代码后的影响	只判断语法，不判断行为是否改变

阅读程序时，最忌讳从头到尾“扫一遍”后直接看选项。这样会觉得每个选项都好像有道理。更稳的做法是先回答三个问题：输入是什么，程序输出什么，核心变量怎样变化。遇到循环和递归时，要写少量追踪过程，而不是在脑中硬记。

例如，程序中出现 `for (int i = 0; i < n; i++)`，就要立刻意识到它访问的是 0 到 $n-1$；如果出现 `for (int i = 1; i <= n; i++)`，访问范围就变成了 1 到 n。这类边界差异常常只差一个字符，却会直接改变答案。

四、完善程序考什么

完善程序题会先说明一个问题，再给出一段带空缺的程序。空缺处通常是表达式、条件、赋值语句、函数参数或返回值。题目要求从选项中选择最合适的内容。

完善程序比阅读程序多了一层要求：不仅要看懂已有代码，还要判断缺失的部分应该承担什么作用。空缺位置不同，思考方向也不同。

空缺位置	应优先判断
初始化位置	变量一开始应该是什么值
循环条件	什么时候继续，什么时候停止
判断条件	哪种情况进入这个分支
状态更新	本轮处理后哪个变量要改变
函数参数	当前递归或调用要把问题变成哪一部分
返回语句	函数最终应该交给外层什么结果

完善程序题不能只看某一个空。很多空缺单独看时都有多个选项像是可行，但放回整段程序后只有一个符合上下文。判断时要把题意、变量含义、循环范围和前后语句连起来。

例如，若题目说要判断一个数是否为完全平方数，程序中出现 `if (...) return true;`，那么这个条件通常应该表达“找到了某个整数平方等于这个数”。此时若选项中有赋值语句、比较语句、错误的倍数关系，就要根据 `if` 的需要排除。`if` 中需要的是能判断真假的条件，而不是把某个值改掉的赋值。

又如递归补全题，空缺如果出现在递归调用的参数里，就不能只看参数类型是否匹配。还要判断这次调用是在处理“规模减小的问题”，还是在移动某一部分，或者在交换两个角色。递归题的选项经常都能通过语法检查，但只有一个符合题意流程。

五、第一轮需要的能力

从题型看，CSP-J 第一轮主要考五种能力。

第一是概念判断能力。学生要知道基本概念的边界，例如基本数据类型和自定义结构的区别，操作系统和普通应用程序的区别，栈和队列的访问规则，树和图的基本术语。

第二是纸面计算能力。进制转换、存储单位、排列组合、简单复杂度、树的结点数量、二分比较次数，都需要在不使用计算器的情况下完成。计算不一定复杂，但很容易因为一个单位、一个指数或一个边界出错。

第三是程序追踪能力。阅读程序时，要能跟住变量变化，判断循环执行次数，追踪数组和字符串访问位置，看清函数返回值。程序追踪不要求把所有变量都列成大表，但关键变量必须落到纸上。

第四是结构识别能力。看到栈、队列、树、图、排序、二分、递归、搜索、动态规划时，要能识别它们在题目中的基本形态。这里的识别不是背模板，而是知道这种结构解决什么问题，代码中哪些语句体现它的规则。

第五是选项排除能力。第一轮的答案在选项中，很多题不需要从零推到最后。能先排除明显不符合语法、边界、类型、数量级或题意的选项，往往能减少计算量，也能避免被干扰项带偏。

六、本书怎样使用

本书按考试能力组织，而不是把 C++ 语法、数学、数据结构和算法机械分开。学习时建议按章节顺序推进，因为前面的考试方法会在后面的知识章节中反复使用。

第 1 章到第 4 章先讲题型方法。它们解决的是“拿到题后怎么想”的问题。不要急着跳到具体知识点，否则容易出现会背概念却不会做题的情况。

第 5 章到第 15 章补足 C++、数学和计算机常识。这里不会把 C++ 写成完整入门课，而是围绕第一轮常见考法讲清楚类型、表达式、循环、函数、数组、字符串、递归、进制、复杂度等内容。

第 16 章到第 30 章讲数据结构和基础算法。重点不是写很长的代码模板，而是理解栈、队列、树、图、排序、模拟、枚举、二分、搜索、动态规划、字符串处理、贪心在第一轮中怎样出现。

第 31 章到第 34 章进行专项训练和综合复盘。它们不再重复讲一遍知识点，而是帮助学生把前面学过的内容放回选择题、阅读程序和完善程序中使用。

学完一章后，不要只问“这一章我看懂了吗”，还要问三个问题：题目会怎样考，选项会怎样设陷阱，我在纸面上怎样判断。能回答这三个问题，才算把知识点转化成了第一轮能力。

0x02 选择题怎么做

选择题是整张试卷的第一部分，通常共 15 题，每题 2 分，共 30 分。它看起来像“会就选，不会就猜”，但实际并不是这样。很多选择题并不要求完整推导，只要能抓住题型、排除错误选项、把关键计算写清楚，就能减少失误。

选择题的难点不在题目篇幅，而在干扰项。一个选项可能只错在一个单位，一个边界，一个符号，或者一个概念的适用范围。做选择题时，不要急着凭第一印象勾答案，而要先判断这道题属于哪一类，再决定用哪种办法处理。

一、先判断题型

选择题大致可以分为四类：概念题、计算题、结构题、程序片段题。不同类型的题，做法不一样。

类型	题目特征	优先做法
概念题	问“以下说法正确的是”“哪个不是”	找关键词，检查概念边界
计算题	涉及进制、单位、组合、复杂度	写出中间量，不心算到底
结构题	涉及栈、队列、树、图、排序	按规则模拟或画小图
程序片段题	给出几行代码，问结果或行为	追踪关键变量和语句顺序

第一步不是算，而是分类。分类之后，注意力才会放对位置。

概念题要抓“绝对说法”。例如“总是”“一定”“只能”“所有”这类词，常常需要特别检查。计算机与程序设计中有很多规则带有前提，离开前提就不一定成立。看到这种选项时，不要只判断它像不像书上的话，还要想有没有反例。

计算题要避免只在脑中算。第一轮不能使用计算器，题目中的数字一般不会过分复杂，但很容易在进制、单位、指数、阶乘、下标上出错。只要题目涉及两步以上计算，就应在草稿处写出关键中间量。

结构题要按定义做，不要凭感觉判断。例如栈题要检查后进先出是否被破坏，队列题要检查先进先出是否被破坏，二叉树遍历题要从根、左子树、右子树的关系入手。结构题常常不难，但很怕“看起来差不多”。

程序片段题要看执行顺序。赋值、比较、自增、自减、函数调用、循环条件，哪怕只看错一个位置，结果也可能完全不同。对于很短的程序片段，最稳的办法是列出一两行变量变化，而不是直接看选项。

二、选项不是最后才看

很多学生做选择题时，会先把题完整算完，再去找答案。这样当然可以，但不一定高效。选择题的特点是答案已经给出，选项本身也是信息。合理利用选项，可以减少计算量。

选项有三种常见作用。

第一，选项能提示数量级。比如某题要求计算 1MB 等于多少二进制位，如果选项中有 1048576 和 8388608，就要意识到前者是字节数，后者才可能是二进制位数。此时不必把所有单位重新展开，只要抓住“字节到位还要乘 8”即可。

第二，选项能帮助排除不可能情况。比如排列组合题中，如果要求“至少包含某类元素”，答案通常不会比“不加限制”的总数还大。若某个选项超过总数，可以先排除。若题目问最坏情况下比较次数，答案也不应小到无法区分足够多的情况。

第三，选项能暴露干扰方向。四个选项往往不是随便写的，它们可能对应几种常见错误：少乘一个 8，把 0 下标当成 1 下标，把 < 看成 <=，把前序遍历和后序遍历混淆，把“子串”和“子序列”混淆。看出干扰方向，反而能提醒自己检查哪里。

但利用选项不等于猜。正确做法是“先理解题意，再用选项缩小范围”。如果还没读清题，就直接在选项中找熟悉的说法，很容易被最像答案的干扰项带走。

三、计算题要留下过程

计算题最常见的失分不是不会，而是算错。第一轮中的计算题常见于进制、存储单位、排列组合、树的结点数量、二分次数、复杂度判断等内容。它们一般步骤短，但每一步都可能出错。

做计算题时，可以遵守一个原则：凡是跨了两种表示，就写转换；凡是有乘除关系，就写单位；凡是有边界，就写范围。

场景	应写出的内容
进制转换	每一位的权值，或先转成十进制
存储单位	1KB = 1024B，1B = 8bit
二分次数	2^k 与数据规模的关系
计数问题	先分情况，检查有没有重复或遗漏
复杂度	看循环层数和每层次数，不只看代码长短

例如问“有序表中有 1000 个元素，二分查找最多比较多少次”，不要只凭感觉选 10。可以写出 $2^{10} = 1024$，这表示经过 10 次比较已经足够区分 1000 个位置附近的情况。这样写一步，能避免误选 7 或 25 这类干扰项。

再如存储单位题，1MB = 1024KB = 1024 × 1024B = 1048576B。题目若问二进制位，还要乘 8，得到 8388608bit。很多错误答案正是停在字节数，没有继续换成位数。

计算题的草稿不需要写成完整证明，只要能帮助自己检查。尤其在纸质考试中，留下中间量还有一个好处：回头检查时知道自己当时为什么这样选，不必重新从头算。

四、概念题要看边界

概念题最怕“差不多”。很多概念在日常学习中经常一起出现，但在选择题中必须分清边界。

例如，`int`、`char`、`float` 是 C++ 中的基本数据类型；`struct` 可以定义结构体类型，但它本身不是与 `int` 同类的基本数据类型。若题目问“哪个不是基本数据类型”，就要按“基本数据类型”的边界判断，而不是按“C++ 里能不能出现这个词”判断。

又如，编译器的主要作用是把源代码转换成机器能够执行的目标代码或相关形式，而不是负责直接运行程序，也不是主要负责调试。调试器、编辑器、操作系统、编译器这些概念在使用时可能连在一起，但选择题会要求分清各自职责。

概念题还常出现“不是”字。看到“以下哪个不是”“不正确的是”“错误的是”，一定要在题号旁边做一个小标记。很多学生不是不会，而是把否定问题当成肯定问题做。

可以把概念题拆成两步：

先找题干限定词。题目问的是“基本数据类型”，还是“循环语句”，还是“操作系统名字”；问的是“正确”，还是“不正确”。

再逐项检查选项。不要因为 A 选项熟悉就马上选。概念题经常有两个选项都很熟悉，但只有一个符合题干限定。

五、结构题按规则走

结构题常见于栈、队列、树、图、表达式、哈夫曼编码、排序等内容。这类题一般不要求写代码，而是要求判断某个序列、遍历结果、编码方式或说法是否符合规则。

做结构题时，不要把规则背成一句话后直接凭感觉选。要把规则转化为一个可以操作的过程。

例如栈的规则是后进先出。判断一个出栈序列是否可能时，可以按入栈顺序模拟：需要某个元素出栈时，就把还没入栈的元素依次压入，直到栈顶是它；如果栈顶不是它，并且它已经被压在下面，就说明这个出栈序列不可能。

二叉树遍历题也类似。前序遍历的第一个结点是根；中序遍历中，根左边是左子树，根右边是右子树。先确定根，再把左右子树切开，继续处理。不要直接在几个选项中找“看着顺”的序列。

图的题要注意“有向”和“无向”，“连通”和“强连通”，“度数”和“入度、出度”。这些词只差一两个字，规则却不同。若题目问无向图所有顶点度数之和，它与边数的关系是每条边贡献 2 个度；若换成有向图，就要分别看入度和出度。

结构题中，图示和小表格比长篇心算更可靠。栈题可以画一列栈，树题可以画根和左右子树，图题可以画点和边。图不需要美观，只要能帮助判断。

六、程序片段题只追关键变量

选择题中有时会给出几行代码，要求判断输出结果或程序行为。这类题虽然属于选择题，但做法接近小型阅读程序。

程序片段题的重点是追踪关键变量。不是所有变量都要列出来，而是要找出会影响答案的变量。循环变量、累加变量、数组下标、返回值通常最关键。

看下面这个示例。

```
int x = 3, y = 5;
x = y;
y = x + 2;
cout << x << " " << y;
```

这段代码的关键不是“有两个变量”，而是第一句赋值后 `x` 已经变成 `5`。第二句中的 `x + 2` 使用的是新值，所以最后输出 `5 7`。如果把赋值理解成“交换”，就会算错。

程序片段题要特别注意赋值语句和比较语句的区别。`x = y` 是把 `y` 的值赋给 `x`，不是判断二者相等；`x == y` 才是比较。若它们出现在 `if` 条件中，含义差别很大。

还要注意自增自减的位置。第 5 章到第 7 章会专门讲表达式和循环，本章只强调做题方法：遇到 `i++`、`++i`、`a[i++]` 这类写法，不要凭感觉跳过，必须确认先使用哪个值，再改变哪个值。

七、原创例题：用选项反查错误

例题 2-1 下列关于存储单位的说法，正确的是（）。

A. 1KB = 1000B
B. 1MB = 1024KB
C. 1B = 10bit
D. 1MB = 1024bit

答案：B

解析：
这道题是概念与计算混合题。先看单位关系：在第一轮常见语境中，1KB = 1024B，1MB = 1024KB，1B = 8bit。

A 把 1024 写成了 1000，不符合题目常见的二进制存储单位换算。C 把 8 位写成了 10 位。D 混淆了 MB 和 bit，1MB 先等于 1024 × 1024B，再乘 8 才是 bit 数。只有 B 正确。

这类题不能只看哪个选项“单位最大”。必须逐项检查单位之间的换算层级：KB 到 B 是 1024 倍，B 到 bit 是 8 倍。

变式题 下列哪个数等于 1KB 对应的二进制位数？（）

A. 1024
B. 8000
C. 8192
D. 1048576

答案：C

解析：
1KB = 1024B，1B = 8bit，所以 1KB = 1024 × 8 = 8192bit。A 是字节数，不是位数；B 把 1024

近似成了 1000；D 是 1MB 的字节数。变式题与原题的差异在于，它不问“哪句话正确”，而是直接要求算出 bit 数，更容易漏乘 8。

八、检查答案时看三件事

选择题做完后，不建议只检查“有没有空题”。更有效的检查是看三件事。

第一，检查否定词。把所有“不是”“错误”“不正确”“不可能”重新圈一遍，确认自己选的是题目要求的方向。选择题中因为看反方向而失分很常见。

第二，检查单位和边界。存储单位有没有从字节换成位，数组下标有没有从 0 开始，循环条件是 `<` 还是 `<=`，二分次数有没有覆盖全部元素。这些地方通常只差一步。

第三，检查选项是否符合数量级。计数题答案不应超过明显的总数；复杂度题不应只根据代码行数判断；树和图的题要看结点、边、度数之间的基本关系。数量级检查不能代替解题，但能发现不少明显错误。

选择题的目标不是每题都用最长方法推完，而是在保证正确率的前提下控制时间。会做的题要稳，不会的题要能排除，算过的题要能检查。做到这三点，选择题就不再只是“看记忆”，而是可以训练出来的一类题型。

0x03 阅读程序怎么读

阅读程序题要求学生在纸面上读懂一段完整程序，再回答判断题和选择题。它不像选择题那样只考一个概念，也不像完善程序那样只补某几个空。阅读程序题通常会同时考输入输出、变量变化、循环边界、数组下标、函数返回值、递归过程、复杂度和修改代码后的影响。

这类题不能靠“看代码感觉”。一段程序中，真正影响答案的语句可能只有几行；但如果没有方法，学生会把所有代码都看成一样重要，最后既浪费时间，又容易漏掉边界。本章讲阅读程序的基本读法：先读结构，再定变量，再追过程，最后看选项。

一、先看程序框架

拿到阅读程序题，不要从第一行开始逐字翻译。更好的做法是先看程序框架，判断它由哪些部分组成。

位置	要看什么	作用
头文件和全局变量	是否有数组、常量、全局计数器	判断数据范围和变量作用范围
函数定义	每个函数输入什么，返回什么	判断函数功能
`main` 函数	读入什么，调用什么，输出什么	确定程序主流程
循环和分支	循环次数、条件变化	判断执行过程
输出语句	输出变量、空格、换行、小数格式	确定最终答案

第一遍读程序，只需要回答三个问题：

程序读入了什么？程序调用了哪些函数？程序最后输出了什么？

这三个问题回答清楚后，再回头分析函数和循环。很多阅读程序题中，函数名字会带有提示，但不能完全相信函数名。比如 `check` 可能检查合法性，也可能检查某个数是否满足条件；`solve` 可能完成全部计算，也可能只处理一部分。函数真正做什么，要看参数、循环和 `return`。

读框架时，还要注意变量在哪里定义。全局变量会自动初始化为 0，局部变量如果没有初始化就使用，程序行为可能出问题。第一轮题目通常会避免过分依赖复杂的未定义行为，但读题时仍要警惕“变量是否已经有值”。

二、给变量写含义

阅读程序最怕只记变量名。变量名短时更容易混淆，例如 `i`、`j`、`k`、`x`、`y`、`t`、`ans`。读程序时，应给关键变量写一句含义。

不需要给每个变量都写长解释，只要写出它在程序中的作用。例如：

变量	可能含义
`n`	输入规模、循环上界、数组长度
`i`、`j`	循环变量或下标
`cnt`	计数器
`sum`	当前和
`ans`	当前答案
`flag`	是否满足某种条件
`dp[i]`	到位置 i 的某种最优值或方案数

变量含义不能只看名字，要看它怎样被修改。若 `cnt` 每次满足条件时加 1，它一般是计数器；若 `sum` 每轮加上数组元素，它一般是前缀和或累计和；若 `ans` 每轮与某个值取较大或较小，它通常记录当前最优结果。

有些变量的含义会随程序阶段改变。例如，`x` 读入时表示输入值，经过循环后可能变成处理后的结果。阅读时要区分“初始含义”和“当前含义”。如果题目问某一行执行后的值，就必须看它已经被前面的语句改过几次。

三、循环要看边界

阅读程序中，循环边界是高频失分点。循环的起点、终点、条件和更新方式都会影响执行次数。

看到循环时，要立刻检查四件事：

内容	例子	需要判断
起点	`int i = 0` 或 `int i = 1`	第一个访问位置是什么
终点	`i < n` 或 `i <= n`	最后一个访问位置是什么
更新	`i++`、`i += 2`、`i--`	每次变化多少
循环体	是否修改循环相关变量	循环次数是否会被改变

例如 `for (int i = 0; i < n; i++)` 访问的是 $0, 1, \ldots, n-1$，共 n 次；`for (int i = 1; i <= n; i++)` 访问的是 $1, 2, \ldots, n$，也是 n 次，但下标整体后移了一位。两种写法都常见，不能混在一起。

`while` 循环比 `for` 循环更需要关注变量变化。因为 `while` 的更新语句不一定写在循环头里，可能藏在分支中。读 `while` 时，要先找“循环条件中的变量在哪里变化”。如果条件变量没有变化，程序可能无法结束；如果只在某个分支中变化，就要判断输入是否会进入那个分支。

遇到嵌套循环时，不要马上把所有次数相乘。先看内层循环是否每次都从同一个起点开始，还是依赖外层变量。例如 `for (int j = 0; j < i; j++)` 的内层次数随 `i` 改变，不能直接写成 n^2，但在第一轮中通常只需要判断大致复杂度或具体执行过程。

四、只追关键过程

阅读程序并不要求把每一步都列成大表。纸面考试时间有限，变量追踪表要短而有效。追踪的目标是回答题目，不是重新执行整份程序。

追踪程序时，可以先找关键变量。通常包括：

- 最终输出的变量；
- 影响分支走向的变量；
- 影响循环是否继续的变量；
- 数组或字符串的访问下标；
- 函数的返回值。

若题目只问输出结果，重点追输出变量。若题目问某个判断题是否正确，就只追与该判断有关的变量。若题目问修改某一行代码后的影响，就要比较修改前后关键变量怎样变化。

下面是一段用于说明方法的程序。

```cpp
#include <iostream>
using namespace std;

int f(int n) {
    int s = 0;
    for (int i = 1; i <= n; i++) {
        if (i % 2 == 0) s += i;
        else s -= i;
    }
    return s;
}

int main() {
    int n;
    cin >> n;
    cout << f(n) << endl;
    return 0;
}
```

如果输入 `5`，不需要把所有语句重新解释一遍，只要追踪 `i` 和 `s`。

i	奇偶	对 s 的操作	操作后 s
1	奇数	`s -= i`	−1
2	偶数	`s += i`	1
3	奇数	`s -= i`	−2
4	偶数	`s += i`	2
5	奇数	`s -= i`	−3

最后 `f(5)` 返回 −3，程序输出 `-3`。这张表只追了两个量，没有列出 `n`、`main` 的执行细节，也没有重复写 `cout`。这种追踪在纸面上更实用。

五、函数要看返回值

阅读程序题中，函数常常是核心。函数可能用于判断、计算、递归、更新数组，也可能只是把一段循环封装起来。读函数时，重点不是函数名，而是参数和返回值。

读函数可以按四步进行：

第一，看参数。参数是函数从外部拿到的信息。例如 `f(int x)` 只拿到一个整数，`g(int a[], int n)` 同时拿到数组和长度。

第二，看局部变量。局部变量通常用于临时计算，函数结束后不会继续存在。若函数返回的是局部变量的值，则要追踪它最后变成什么。

第三，看 `return`。一个函数可能有多个 `return`。分支中提前 `return` 的语句尤其重要，它会让后面的代码不再执行。

第四，看调用处。函数本身算出结果后，还要看调用者怎样使用这个结果。结果可能直接输出，也可能参与下一步计算。

判断函数时，不要看到 `return true` 就认为函数一定返回 `true`，也不要看到最后有 `return false` 就忽略前面的提前返回。第一轮阅读程序中，很多判断题会围绕“某个输入是否会提前返回”设置。

对于递归函数，本章只给出读法，不展开递归专题。递归题先看出口，再看每次调用参数是否向出口靠近。若参数没有向出口靠近，可能无法结束；若每次减少 1，可以从出口倒着推，也可以从输入开始展开几层。

六、数组和字符串看下标

数组和字符串题最容易错在下标。C++ 中普通数组和 `string` 的下标通常从 0 开始。长度为 n 的字符串，合法下标是 0 到 $n-1$。

看到数组或字符串时，要检查三件事。

第一，数组开多大，访问到哪里。若数组是 `a[100]`，合法下标一般是 0 到 99。如果程序用 `for (int i = 1; i <= n; i++)` 访问 `a[i]`，就要确认 n 是否可能等于 100。

第二，题目输入是否保证不越界。阅读程序题常会说明“程序输入不超过数组或字符串定义的范围”。这句话的意思是，通常不需要从非法输入角度攻击程序，但仍要看题目中的判断题是否讨论越界情况。

第三，字符串的长度和最后一个字符的位置不同。若 `s.size()` 为 n，最后一个字符是 `s[n - 1]`，不是 `s[n]`。很多程序片段会使用 `s[i - 1]`，这是因为循环变量 `i` 从 1 开始，而字符串下标从 0 开始。

如果程序同时出现两个字符串或二维数组，最好在草稿上写清楚行、列或两个字符串的含义。不要把 `a[i][j]` 和 `a[j][i]` 混用，也不要把 `x[i - 1]` 和 `y[j - 1]` 看反。

七、选项要回到程序中验证

阅读程序题的判断题和选择题都不能脱离程序。选项中的说法可能看起来符合常识，但程序未必这样写；也可能看起来奇怪，但正好符合代码行为。

常见选项可以分成几类。

选项类型	判断方法
具体输入输出	代入输入，追踪关键变量
函数功能描述	检查函数对所有相关情况是否成立
修改代码影响	比较修改前后的条件、循环和返回值
复杂度判断	看循环层数、递归次数或数组规模
边界情况	代入最小值、最大值或特殊值

判断题尤其容易出现“部分正确”。例如“该函数计算 1 到 n 的所有整数之和”这句话，如果程序实际上从 0 加到 $n-1$，对于某些输入可能结果相近，但描述仍然错误。判断题要求的是说法是否成立，而不是是否“差不多”。

选择题中，若问某个输入的输出，最稳的方法是先独立算出结果，再看选项。若算到一半发现某些选项已经不可能，也可以排除。不要先看选项再把程序往某个选项上凑。

修改代码影响题要特别谨慎。比如把 `i * i <= n` 改成 `i <= n`，程序结果可能不变，但运行次数变多；把 `<=` 改成 `<`，结果可能只在平方数等边界情况下改变。不能只判断“代码还能运行”，还要判断“输出是否变、复杂度是否变、边界是否变”。

八、原创例题：读一段程序

例题 3-1 阅读下面的程序，回答问题。

```cpp
#include <iostream>
using namespace std;

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

int main() {
    int x;
    cin >> x;
    cout << calc(x) - x << endl;
    return 0;
}
```

判断题：

1. 当输入为 `6` 时，程序输出 `6`。

2. `calc(n)` 的返回值是 n 的所有正因数之和。

3. 若输入为任意正整数，程序输出一定小于输入值。

单选题：

4. 当输入为 12 时，程序输出为 ()。

A. 12
B. 16
C. 18
D. 28

答案：
1. ✓
2. ✓
3. ✗
4. B

解析：
先看函数 calc。循环变量 i 从 1 到 n，每次判断 n % i == 0。若成立，说明 i 是 n 的正因数，于是加入 ans。所以 calc(n) 返回 n 的所有正因数之和。

main 中输出的是 calc(x) - x，也就是所有正因数之和减去 x 本身。换句话说，输出的是 x 的所有真因数之和。

输入为 6 时，正因数为 $1, 2, 3, 6$，calc(6)=12，输出 $12 - 6 = 6$，所以第 1 小题正确。

第 2 小题直接对应函数功能，正确。

第 3 小题说 “输出一定小于输入值”。输入为 6 时，输出为 6，并不小于输入值；输入为 12 时，输出为 $1 + 2 + 3 + 4 + 6 = 16$，还大于输入值。所以第 3 小题错误。

输入为 12 时，正因数为 $1, 2, 3, 4, 6, 12$，calc(12)=28，程序输出 $28 - 12 = 16$，选择 B。

这道题的关键是区分 calc(x) 和最终输出。若只看函数返回值，会误选 D；若只把因数算到 6，漏掉 12，也会影响判断。

变式题 若把程序中的输出语句改为：

```
cout << calc(x) << endl;
```

当输入为 12 时，输出为 ()。

A. 12
B. 16
C. 28
D. 6

答案：C

解析：
变式只改了输出语句。原程序输出 calc(x) - x，变式输出 calc(x)。输入为 12 时，calc(12)=1+2+3+4+6+12=28，所以选 C。这里考的是 “修改某一行后的影响”，不能沿用原题的最终输出。

九、读题顺序

阅读程序题建议按下面的顺序完成。

先读 `main` 函数，知道输入、调用和输出。再读被调用的函数，弄清函数返回值。接着看题目给出的判断题和选择题，带着问题回到代码中追踪。最后检查边界、输出格式和修改代码的影响。

不要一开始就陷入某个复杂循环。阅读程序不是把代码逐行翻译成中文，而是找出“输入如何变成输出”。一旦抓住这条主线，很多判断题都能围绕主线解决。

对于较长程序，可以在草稿上写三行：

输入：……

核心过程：……

输出：……

这三行能防止读到后面忘记前面，也能帮助区分函数内部结果和最终输出结果。

十、本章要点

阅读程序题的核心是纸面追踪。读程序时，先看框架，再定变量含义，再追关键过程，最后用选项回到程序中验证。

做题时尤其要注意循环边界、数组下标、函数返回值和输出语句。很多错误不是因为不会算法，而是因为把 `i < n` 看成 `i <= n`，把 `calc(x)` 看成最终输出，或者把字符串长度当成最后一个下标。

阅读程序要追踪，但不能追得过满。草稿表格越大，越容易浪费时间。只追影响答案的变量，抓住输入到输出的主线，才是第一轮阅读程序题最稳的做法。

0x04 完善程序怎么补

完善程序题给出问题说明和一段带空缺的程序，要求从选项中补全缺失部分。它不像阅读程序题那样只判断已有代码的行为，而是要判断“这里应该写什么，整段程序才符合题意”。

这类题的难点不在语法本身。很多选项放进空缺处都能通过语法检查，但只有一个符合程序意图。做完善程序时，不能只盯着空缺本身，也不能凭记忆套模板。更稳的方法是：先读题意，再读框架，确定空缺所在位置的作用，最后用前后文和选项一起验证。

一、先定空缺作用

看到空缺后，不要马上看选项。第一步应判断这个空缺在程序中承担什么任务。

空缺位置	常见作用	判断重点
变量初值	初始化计数器、答案、边界	初始值是否会影响第一次循环
循环条件	控制循环是否继续	是否多跑一轮或少跑一轮
判断条件	决定是否进入分支	条件是否与题意一致
更新语句	改变下标、计数器、状态	是否能推进过程
函数参数	传递子问题或当前位置	参数顺序是否符合流程
返回值	交出函数结果	返回的是答案、布尔值还是中间量

空缺作用判断错了，后面选项很难选对。比如空缺在 `if (...)` 里面，它需要的是一个判断条件；空缺在 `return ...;` 里面，它需要的是返回值；空缺在递归调用参数中，它要体现问题规模如何变小，或者角色如何交换。

有些题目会把多个空缺放在同一个流程里。此时不能把每个空独立处理，而要看它们之间的配合。一个空负责初始化，另一个空负责循环更新，最后一个空负责返回结果。只要其中一个空的理解错了，后面的空也容易连带出错。

二、题意要落到变量上

完善程序题的题面通常会先说明问题，再给出代码。题面里的对象要和代码里的变量对应起来。

例如题面说“给定一个长度为 n 的数组，求其中所有偶数的个数”，代码里可能出现 `n`、`a[i]`、`cnt`。这时可以写出：

- `n` 表示数组长度；
- `a[i]` 表示第 i 个数；
- `cnt` 表示已经统计到的偶数个数。

这样做的目的不是翻译变量名，而是把题意放进代码。空缺处要填什么，往往取决于变量含义。若 cnt 是计数器，那么每找到一个偶数时应执行 cnt++，而不是把 cnt 赋成当前下标；若 ans 表示最小值，初始化和更新都要符合“越小越好”的方向。

题意和变量对应时，要特别注意三类信息。

第一，输入范围。输入范围可能决定数组下标从哪里开始，也可能决定初始答案能否设为某个特殊值。若数组至少有一个元素，可以把 ans 初始化为第一个元素；若可能没有元素，就需要额外处理空情况。

第二，目标结果。题目是求个数、求和、求最值，还是判断是否存在？不同目标对应不同的返回值和更新方式。求个数常见 cnt++，求和常见 sum += ...，求最小值常见 ans = min(ans, ...)，判断是否存在常见找到后返回 true。

第三，边界条件。题目是否包含第一个位置、最后一个位置、空串、长度为 1 的情况，这些都会影响循环范围和返回值。

三、上下文比选项更重要

完善程序题常会给出四个看起来相似的选项。如果只看选项，很容易觉得都像答案。正确做法是先读空缺前后的语句。

空缺前的代码通常告诉你“已经做到了哪一步”，空缺后的代码通常告诉你“接下来要依赖什么结果”。空缺处应当把前后两部分接起来。

例如：

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

若题目要求统计偶数个数，那么空缺 ① 应该是 a[i] % 2 == 0。原因不只是“偶数条件是这样写”，还因为 if 成立后执行的是 cnt++，这说明空缺处必须判断当前元素是否应该被计数。

再看一个空缺在更新位置的片段：

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

若题目要求输出 1 到 n 的和，空缺 ① 应填 s。如果选项中有 i、n、s + n，它们都可能是整数表达式，但只有 s 是循环累计出的结果。

上下文判断尤其适合处理动态规划、递归、二分、搜索这类题。它们的代码看起来更长，但空缺处

仍然有明确位置：初始化、转移、边界、递归调用、返回结果。只要先确定位置作用，就能缩小选择范围。

四、先补逻辑，再对选项

完善程序题是选择题，但不要一开始就被选项牵着走。更好的顺序是先在心里补出一句“这里应该表达什么逻辑”，再去找对应选项。

例如，题目要求判断 num 是否为完全平方数，程序中有：

```
for (int i = 1; i <= bound; i++) {
    if (①) {
        return true;
    }
}
return false;
```

先不看选项，也应知道空缺 ① 要表达“i 的平方等于 num”，也就是 `i * i == num`。如果选项中出现 `num = i * i`，它是赋值，不是比较；如果出现 `num == 2 * i`，它判断的是偶数关系，不是平方关系。

这种“先补逻辑”的做法，可以减少被干扰项影响。完善程序的错误选项常见有以下几类：

干扰方式	例子	问题
赋值冒充比较	`x = y`	改变了变量，不是判断相等
边界差一	`i < n` 与 `i <= n`	可能漏掉最后一个位置
方向相反	`left = mid` 与 `right = mid`	搜索区间被错误缩小
参数顺序错	`dfs(tmp, src, tgt)`	角色交换不符合流程
返回值类型不合	`return i` 与 `return true`	返回含义不一致
使用未更新变量	返回旧下标或旧答案	与前面的更新脱节

对选项时，要检查它是否同时满足语法和逻辑。语法正确只是最低要求，逻辑正确才是答案。

五、看空缺之间的配合

很多完善程序题一段程序有多个空。多个空之间往往不是并列关系，而是共同完成一个算法流程。

例如二分查找类程序，空缺可能包括：

- 初始左端点；
- 初始右端点；
- 中点计算；
- 判断条件；
- 左右端点更新；
- 最终返回值。

这些空不能分开猜。若程序采用左闭右闭区间，循环条件、端点更新和返回值要配套；若程序采用左闭右开区间，写法又会不同。只把某个空看成“二分常见写法”，可能会和整段程序不匹配。

递归题也是这样。递归程序常包含出口、处理当前层、递归调用三个部分。空缺如果出现在出口处，要看最小规模问题如何解决；如果出现在递归调用处，要看每次调用是否减少规模；如果出现在递归参数中，要看参数顺序是否表示题意中的角色变化。

动态规划题的多个空通常围绕“初始状态”和“状态转移”。先判断 `dp[i]` 或 `dp[i][j]` 的含义，再看空缺。若 `dp[i]` 表示到第 i 个位置的最小代价，那么转移通常要取较小值；若表示方案数，转移通常是加法。不要看到 `dp` 就直接套某个固定公式。

完善程序题中，空缺之间的配合比某个单独选项更重要。做题时可以在题旁写出“这段程序想做什么”，再把每个空放回这个目标中检查。

六、原创例题：补全计数程序

例题 4-1 给定一个正整数 n，统计 1 到 n 中有多少个数能被 3 整除。试补全程序。

```
#include <iostream>
using namespace std;

int main() {
    int n;
    cin >> n;
    int cnt = ①;
    for (int i = ②; i <= n; i++) {
        if (③) {
            ④;
        }
    }
    cout << ⑤ << endl;
    return 0;
}
```

选项：

①处应填（）

A. `1`
B. `0`
C. `n`
D. `3`

②处应填（）

A. `0`
B. `1`
C. `3`
D. `n`

③处应填（）

A. `i / 3 == 0`
B. `i % 3 == 0`
C. `n % i == 0`

D. `i % n == 0`

④处应填（）

A. `cnt++`

B. `i++`

C. `cnt = i`

D. `n++`

⑤处应填（）

A. `i`

B. `n`

C. `cnt`

D. `cnt + n`

答案：

① B，② B，③ B，④ A，⑤ C。

解析：

题目要求统计个数，所以需要一个计数器 `cnt`。一开始还没有统计任何数，①处应为 `0`。

统计范围是 1 到 n，循环变量 `i` 应从 1 开始，并且循环条件已经写成 `i <= n`，所以②处应为 `1`。

每次循环处理当前的 `i`。若 `i` 能被 3 整除，就把它计入答案。判断“能被 3 整除”的条件是 `i % 3 == 0`，所以③处选 B。

条件成立时，计数器增加 1，所以④处应为 `cnt++`。不能写 `i++`，因为 `i` 是循环变量；也不能写 `cnt = i`，那会把计数器改成当前数值，而不是增加个数。

循环结束后，`cnt` 保存统计结果，⑤处应输出 `cnt`。

这道题的每个空都不复杂，但五个空共同完成“初始化—枚举—判断—计数—输出”的流程。若只看单个空，可能觉得 `0`、`1`、`3` 都与题目有关；放回流程中，就能看出各自位置的含义。

变式题 若题目改为“统计 1 到 n 中所有能被 3 整除的数的和”，则上面程序至少需要怎样修改？

A. 把 `cnt++` 改为 `cnt += i`，输出仍为 `cnt`

B. 把 `cnt++` 改为 `i += cnt`，输出仍为 `cnt`

C. 把 `i % 3 == 0` 改为 `i / 3 == 0`

D. 把 `cnt` 初值改为 `3`

答案：A

解析：

变式题把“统计个数”改成“统计总和”。变量 `cnt` 虽然名字像计数器，但如果继续使用它保存结果，它的含义应改为“当前总和”。初值仍然是 `0`；判断条件仍然是 `i % 3 == 0`；条件成立时不再加 1，而是加当前数 `i`，所以应改为 `cnt += i`。输出的仍是保存结果的变量 `cnt`。

这说明完善程序不能只看变量名。变量名能提示含义，但最终要以题意和代码更新方式为准。

七、递归空缺看三件事

递归完善题在第一轮中较常见。它的代码不一定长，但空缺位置容易迷惑学生。处理递归空缺时，可以只看三件事：出口、规模、角色。

出口是递归停止的条件。比如 `if (n == 1)`、`if (i == 0)`、`if (l > r)`，这些条件说明最小问题如何直接求解。出口处的返回值必须和题意一致。求和时可能返回 0 或某个元素，求阶乘时可能返回 1，判断问题可能返回 `true` 或 `false`。

规模是下一次递归是否更接近出口。常见写法有 `n - 1`、`i + 1`、`l + 1`、`r - 1`。若参数没有变化，或变化方向远离出口，递归就无法完成题意。完善程序题的选项中常会出现看似类型正确但规模不变的递归调用，应优先排除。

角色是参数的意义是否发生交换。汉诺塔、搜索、树遍历等题中，多个参数可能表示起点、中转点、目标点，或者当前结点和父结点。递归调用时，参数顺序不能随意交换。判断这类空缺，要把每个参数写成中文含义，再看调用是否符合流程。

递归题不要一上来展开很多层。先判断出口是否正确，再判断每次递归是否向出口靠近，最后用一个小输入代入检查。小输入可以选 1、2 或 3，因为这几个值最容易暴露出口和参数顺序错误。

八、二分空缺看区间

二分完善题的难点在于区间写法。不同写法都可能正确，但一段程序内部必须一致。

若程序使用 `while (l <= r)`，通常表示当前区间两端都还可能包含答案，更新时常见 `l = mid + 1` 或 `r = mid - 1`。若程序使用 `while (l < r)`，可能是在不断缩小到一个位置，更新时常见 `l = mid + 1` 或 `r = mid`。这两类写法不能随意混用。

看二分空缺时，应先回答：

- `l` 和 `r` 表示的区间是否包含端点；
- `mid` 属于左半还是右半；
- 判断条件成立时，答案在左边还是右边；
- 更新后区间是否变小；
- 循环结束后应返回哪个变量。

例如某程序寻找第一个满足条件的位置，如果 `mid` 已经满足条件，那么答案可能是 `mid`，也可能在它左边，所以常见更新是 `r = mid`。如果 `mid` 不满足条件，答案只能在右边，常见更新是 `l = mid + 1`。这个判断来自“第一个满足条件”的题意，而不是来自某个固定模板。

二分题中，选项常会故意混淆 `l = mid`、`l = mid + 1`、`r = mid`、`r = mid - 1`。判断它们时，不要只看是否会编译，而要看区间是否缩小、是否漏掉可能答案、是否可能死循环。

九、检查补全结果

完善程序做完后，最好进行一次整体检查。检查不需要重新完整推导，只看以下四点。

第一，变量是否有合理初值。计数器通常从 0 开始，求和变量通常从 0 开始，乘积变量常从 1 开始，最值变量要结合题目范围确定。

第二，循环是否覆盖题目要求的范围。题目要求 1 到 n，循环就不能从 0 到 $n-1$，除非代码中另有转换。题目要求遍历字符串所有字符，就要覆盖下标 0 到 $n-1$。

第三，空缺处是否推进程序。循环和递归必须向结束条件靠近；搜索和二分必须让范围缩小；队列和栈的处理必须让待处理元素发生变化。

第四，输出是否是题目要求的答案。有些程序中间变量很多，最后输出的变量不一定是刚刚更新的变量。完善程序题常在最后一个空考“应该输出谁”或“应该返回谁”。

如果时间允许，可以用一个很小的样例代入整段程序。样例不一定来自题目，可以自己选。例如计数题选 $n=3$，二分题选长度 3 的数组，递归题选 $n=1$ 或 $n=2$。小样例能快速发现初值、边界和返回值错误。

十、本章要点

完善程序题的核心不是猜空，而是还原程序意图。先读题意，把题意对应到变量；再看空缺位置，判断它负责初始化、判断、更新、递归调用还是返回；然后结合上下文选择答案。

做题时不要被选项牵着走。先想清楚空缺处应该表达什么逻辑，再到选项中找对应写法。遇到多个空，要看它们是否共同完成同一流程，不要把每个空拆开孤立判断。

选择题可以靠排除，阅读程序可以靠追踪，完善程序则更依赖上下文。能读出“这段程序想完成什么”，并能判断“这一空在流程中负责什么”，就抓住了完善程序题的主要方法。

第二编 C++ 与程序行为

0x05 类型与变量

程序中的每一个值都要以某种类型存在。变量的类型决定它能保存什么样的数据，参与运算时会怎样变化，输出时会呈现什么结果。CSP-J 第一轮不会把类型系统考得很深，但会频繁考类型范围、字符与整数的关系、布尔值、变量初始化和简单类型转换。

本章不把 C++ 类型讲成完整语法课，只讲第一轮中最容易出题、最容易看错的部分。读程序时，类型决定了变量的“解释方式”；做选择题时，类型决定了某个说法是否成立；做完善程序时，类型决定了空缺处应该返回数字、字符还是真假值。

一、变量不是名字，而是带类型的值

变量可以理解成程序中保存数据的位置。变量名只是给这个位置取的名字，影响程序行为的是它的类型和值。

例如：

```
int x = 65;
char c = 'A';
bool ok = true;
double t = 3.5;
```

这几行代码中，`x` 保存整数，`c` 保存字符，`ok` 保存真假值，`t` 保存小数。它们都叫变量，但能参与的操作并不完全相同。

第一轮题目中常见的变量类型如下。

类型	常见含义	第一轮常见考点
`int`	整数	范围、溢出、整数除法
`long long`	更大范围整数	大数计算时避免超过 `int`
`char`	单个字符	字符编码、字符与整数相加
`bool`	真假值	条件判断、`true` 与 `false`
`float`、`double`	小数	精度、输出格式
`string`	字符串	长度、下标、字符访问

这里的 `string` 不是内置基本类型，但在第一轮程序阅读中非常常见。本章只说明它与变量类型有关的基本用法，字符串的下标、长度和处理方法会在第 9 章集中讲。

做题时，不要只看变量名。例如 `flag` 不一定是布尔变量，`ans` 不一定是整数，`ch` 也可能被当作整数参与运算。判断变量含义，要同时看类型、初值和后续语句。

二、整数类型先看范围

`int` 是第一轮中最常见的整数类型。很多题会直接说明“假设 `int` 为 32 位有符号整数类型”，此时它的取值范围通常按 -2147483648 到 2147483647 判断。

这个范围的两个端点很容易写错。最大值是 2147483647，最小值是 -2147483648。最小值的绝对值比最大值多 1，这是选择题常见干扰点。

类型	常见范围理解	做题提醒
`int`	约 -2.1×10^9 到 2.1×10^9	题目若指定 32 位，按精确范围判断
`unsigned int`	只能表示非负整数	与负数比较时容易出问题
`long long`	约 -9×10^{18} 到 9×10^{18}	常用于较大的整数计算

在第一轮中，判断是否会超出范围时，不一定需要记完整端点。很多时候只需要看数量级。例如 $10^5 \times 10^5 = 10^{10}$，已经超过 32 位 `int` 的范围，需要考虑 `long long`。但如果题目问的是精确范围，就必须记清楚端点。

还要注意，整数除法和数学中的除法不完全一样。两个整数相除，结果仍是整数，小数部分会被舍去。例如 `5 / 2` 的结果是 `2`，不是 `2.5`。如果想得到小数结果，至少要让其中一个操作数是小数类型，例如 `5.0 / 2`。

本章只提醒这一点，表达式和运算顺序会在第 6 章继续讲。

三、原码、反码与补码

整数在计算机中最终以二进制形式存储。对于非负整数，二进制表示比较直接。例如用 8 位表示 5，结果是 `00000101`。困难主要出现在负数：计算机需要用固定长度的二进制位表示正数、负数和零。

在第一轮中，原码、反码和补码通常只考最基本的表示方法，不会深入计算机组成原理。可以先看下面这张表。

表示方法	正数表示	负数表示	8 位表示 -5
原码	最高位为 0，其余位表示数值	最高位为 1，其余位表示绝对值	`10000101`
反码	与原码相同	符号位不变，其余位按位取反	`11111010`
补码	与原码相同	在反码基础上加 1	`11111011`

用 8 位表示 -5 时，可以按三步得到补码：

1. 5 的二进制是 `00000101`；
2. 按位取反得到 `11111010`；
3. 加 1 得到 `11111011`。

所以，-5 的 8 位补码是 `11111011`。

补码有一个重要结果：n 位有符号整数通常能表示 -2^{n-1} 到 $2^{n-1}-1$。例如 8 位有符号整数范围是

−128 到 127，32 位 `int` 的常见范围是 −2147483648 到 2147483647。正数少一个，是因为补码中只有一个零，而最小负数多占了一个位置。

这也解释了为什么 32 位 `int` 的最小值不是 −2147483647，而是 −2147483648。选择题常把这两个范围放在选项中混淆。

还可以记住几个常见例子：

位数	补码	含义
8 位	`00000000`	0
8 位	`00000101`	5
8 位	`11111111`	−1
8 位	`11111011`	−5
8 位	`10000000`	−128

做题时要注意题目是否说明“按补码表示”。如果题目只问 CSP-J 常见的 `int` 范围，通常直接按补码范围判断即可；如果题目专门问原码、反码、补码，就要按表中的规则逐步处理。

第一轮最常见的判断有三类：

- 正数的原码、反码、补码相同；
- 负数的补码是“绝对值二进制，按位取反，再加 1”；
- n 位补码有符号整数范围是 -2^{n-1} 到 $2^{n-1}-1$。

不要把原码、反码、补码扩展成复杂计算。本章只需要它们帮助解释整数范围和负数表示。

四、字符也可以参与整数运算

`char` 用来保存单个字符，例如 `'a'`、`'A'`、`'0'`。字符常量要使用单引号，字符串常量要使用双引号。`'a'` 是字符，`"a"` 是字符串，这一点在选择题中常见。

字符在程序中也有对应的编码值。因此，字符可以和整数相加减。例如在常见编码下，`'a' + 1` 对应 `'b'`，`'a' + 13` 对应 `'n'`。第一轮题目常用这种方式考字符与整数之间的关系。

看字符题时，要分清三个对象：

写法	含义	常见错误
`'3'`	字符 3	误认为整数 3
`3`	整数 3	误认为字符
`"3"`	长度为 1 的字符串	误认为字符

字符 `'3'` 和整数 `3` 不是同一个值。`'3'` 表示字符，它的编码值不是 3。如果题目问 `char` 变量输出什么，要看它是以字符形式输出，还是先转换成整数再输出。

示例：

```cpp
#include <iostream>
using namespace std;

int main() {
    char c = 'a';
    cout << char(c + 2) << endl;
    cout << int(c) << endl;
    return 0;
}
```

第一行输出把 `c + 2` 再转换为 `char`，得到字符 `c`。第二行把字符 `a` 转换为整数输出，在常见编码下会输出它的编码值。做题时，如果题目没有要求记具体编码表，通常只需要知道小写字母按顺序连续排列，大写字母按顺序连续排列，数字字符也按顺序连续排列。

五、布尔值用于判断

`bool` 类型只有两个值：`true` 和 `false`。它常用于保存判断结果，例如某个数是否为素数，某个位置是否访问过，某个条件是否成立。

在输出时，`cout << true` 通常输出 `1`，`cout << false` 通常输出 `0`。这也是阅读程序题中容易忽略的地方。程序里写的是 `true` 或 `false`，输出时不一定显示单词本身。

布尔变量常见写法如下：

```cpp
bool ok = true;
if (ok) {
    cout << 1 << endl;
}
```

`if (ok)` 的含义是“如果 `ok` 为真”。不要把它误读成“如果 `ok` 等于某个具体整数”。虽然 `true` 输出时常表现为 `1`，但从题意上看，`bool` 更适合表示真假状态。

布尔题还有一个常见陷阱：非零数转换为 `bool` 时通常为 `true`，`0` 转换为 `bool` 时为 `false`。因此，`bool b = 5;` 后，`b` 的值是真，而不是整数 `5`。若程序输出 `b`，输出的是 `1`。

第一轮中，`bool` 经常与函数返回值结合。例如 `isPrime(n)`、`check(x)` 这类函数通常返回真假值。读这种函数时，要看哪些情况返回 `true`，哪些情况返回 `false`，不要只看函数名。

六、变量要注意初始化

变量在使用前应当有确定的值。全局变量一般会被初始化为 `0`，局部变量如果没有初始化就直接使用，值可能不可预测。第一轮题目通常不会故意依赖复杂的异常情况，但变量是否初始化仍是阅读程序和完善程序中的重点。

例如：

```cpp
int sum = 0;
for (int i = 1; i <= n; i++) {
```

```
    sum += i;
}
```

这里 sum 必须从 0 开始，才能正确累计。如果没有给 sum 初值，后面的 sum += i 就是在一个不确定值的基础上累加，结果无法按正常求和理解。

不同任务对应的常见初值不同。

任务	常见变量	常见初值
计数	cnt	0
求和	sum	0
求乘积	prod	1
判断是否存在	ok	false
判断是否全部满足	ok	true
求最大值	mx	第一个元素或足够小的值
求最小值	mn	第一个元素或足够大的值

“是否存在” 和 “是否全部满足” 的初值容易混淆。判断是否存在某种元素时，一开始还没找到，通常设为 false；判断是否所有元素都满足条件时，一开始没有发现反例，通常设为 true，一旦发现不满足就改为 false。

完善程序题常把初值作为空缺。判断初值时，不要只看变量名，要看后面如何更新。如果后面是 ans = max(ans, a[i])，初值就不能随便设成 0，因为数组元素可能全是负数。此时更稳的方式是把 ans 初始化为第一个元素。

七、类型转换看结果丢了什么

程序中不同类型的值可能一起运算。C++ 会在很多情况下进行类型转换。第一轮不要求掌握所有转换规则，但要知道几个常见结果。

第一，小数赋给整数，小数部分会丢失。例如 int x = 3.8; 后，x 的值是 3，不是 4。

第二，整数参与小数运算时，结果可能变成小数。例如 5.0 / 2 得到 2.5，而 5 / 2 得到 2。

第三，字符可以转换成整数，整数也可以转换成字符。字符加减整数时，常用于得到相邻字符。

第四，整数可以转换成布尔值，0 为 false，非零为 true。

看下面的示例。

```
#include <iostream>
using namespace std;

int main() {
    int a = 5 / 2;
    double b = 5 / 2;
    double c = 5.0 / 2;
    bool d = -3;
```

```
    cout << a << " " << b << " " << c << " " << d << endl;
    return 0;
}
```

a 的值是 2。b 虽然是 double，但右边的 5 / 2 先按整数除法得到 2，再赋给 b，所以 b 是 2。c 的右边有 5.0，按小数除法得到 2.5。d 接收非零整数，所以为 true，输出时通常显示 1。

这类题的关键不是看左边变量类型，而是看右边表达式先怎样计算。第 6 章会继续讲表达式，本章先记住：赋值不会改变右边已经完成的整数除法结果。

八、常量与 const

const 用来声明不能被修改的变量。例如：

```
const int N = 1000;
```

声明后，N 的值不能再被赋新值。第一轮中，const 常用于数组大小、固定上界、不会变化的参数。选择题可能直接问哪个关键字用于声明值不能被修改的变量。

const 的意义是限制修改，不是让变量变得“更快”，也不是表示它一定是全局变量。它可以出现在函数内，也可以出现在函数外。读程序时，看到 const int N = 1000;，应把 N 理解为固定常量。

有些同学会把 const 和宏定义混在一起。本书不展开宏，只提醒一点：第一轮中如果题目问 C++ 关键字，const 是关键字；如果题目问某个变量能否被重新赋值，要看它是否被 const 修饰。

九、作用域：变量在哪里有效

变量并不是在整个程序中都能使用。它有自己的作用范围。定义在函数外的变量是全局变量，定义在函数或代码块内部的变量是局部变量。

例如：

```
#include <iostream>
using namespace std;

int x = 10;

int main() {
    int x = 3;
    cout << x << endl;
    return 0;
}
```

这段程序中，函数外有一个 x，main 函数内也有一个 x。在 main 内部直接写 x，使用的是局部变量 x，所以输出 3。这类题考的是变量作用域，而不是赋值计算。

第一轮不会深入复杂的作用域规则，但要掌握两点。

第一，在一个代码块 {} 内定义的变量，通常只在这个代码块内有效。离开代码块后，不能再用这个局部变量。

第二，如果内层变量和外层变量同名，内层变量会遮住外层变量。读程序时要看当前语句处在哪一层代码块里。

在阅读程序题中，同名变量容易造成误判。尤其是函数参数和全局变量同名时，要注意函数内部使用的是参数还是全局变量。不要因为名字相同，就认为它们一定是同一个变量。

十、原创例题：判断变量输出

例题 5-1 阅读下面的程序，判断输出结果。

```
#include <iostream>
using namespace std;

int main() {
    int a = 5 / 2;
    double b = 5 / 2;
    double c = 5.0 / 2;
    char ch = 'a' + 3;
    bool ok = a - 2;
    cout << a << " " << b << " " << c << " " << ch << " " << ok << endl;
    return 0;
}
```

程序的输出是（）。

A. `2 2 2.5 d 0`

B. `2 2.5 2.5 d 0`

C. `2 2 2.5 c 1`

D. `2 2.5 2.5 c 1`

答案：A

解析：

`a = 5 / 2` 中，`5` 和 `2` 都是整数，所以先做整数除法，结果为 2。

`b` 的类型是 `double`，但右边仍然是 `5 / 2`。右边先按整数除法得到 2，再赋给 `b`，所以输出时是 `2`，不是 `2.5`。

`c = 5.0 / 2` 中，`5.0` 是小数，运算结果为 2.5。

`ch = 'a' + 3`，小写字母连续排列，`'a'` 后面第 3 个字符是 `'d'`，所以 `ch` 为 `d`。

`ok = a - 2`。此时 `a` 是 2，所以 `a - 2` 为 0，转换为 `bool` 后是 `false`，输出时为 `0`。

因此输出为 `2 2 2.5 d 0`，选 A。

这道题同时考整数除法、赋值前的表达式计算、字符加法和布尔转换。不要只看变量左边的类型，关键要看右边表达式先算出什么。

十一、本章要点

类型决定变量如何保存和解释数据。第一轮中，`int` 常考范围和整数除法，原码、反码和补码常用于解释负数表示与整数范围，`char` 常考字符与整数的关系，`bool` 常考真假值与输出，`double` 常考小数运算，`const` 常考值不能被修改。

读程序时，要把变量的类型、初值、作用范围和更新方式放在一起看。不要只凭变量名判断含义，也不要只看赋值左边的类型。很多错误都发生在右边表达式已经按另一种类型算完之后。

做题时先问三件事：变量是什么类型，初值是什么，后面怎样变化。能回答这三点，类型与变量相关的选择题、阅读程序题和完善程序题就有了稳定的判断基础。

0x06 表达式与运算符

表达式是程序中用来计算值的部分。一个常量、一个变量、一次加法、一次比较、一次函数调用，都可以看成表达式。运算符决定表达式怎样计算，例如 `+` 表示加法，`==` 表示相等比较，`&&` 表示逻辑与，`=` 表示赋值。

CSP-J 第一轮不会要求学生背完整的 C++ 运算符表，但会频繁考几类问题：整数除法和取余、赋值与比较的区别、逻辑表达式的真假、复合赋值、自增自减、运算符优先级。做这类题时，不能只凭数学直觉，要按 C++ 的表达式规则判断。

本章只讲第一轮中常见、容易失分的运算符。分支和循环会在第 7 章继续展开，本章重点放在“表达式本身怎样算”。

一、表达式先产生一个值

表达式计算后会产生一个值。这个值可以是整数、小数、字符，也可以是真假值。

例如，若 `a = 7`，`b = 3`：

- `a + b` 的值是 10；
- `a > b` 的值是 `true`，输出时通常显示为 `1`；
- `a / b` 的值是 2，因为这里是整数除法；
- `a % b` 的值是 1，因为 $7 = 3 \times 2 + 1$。

表达式题的第一步是判断“这个表达式最后得到什么值”。不要只看它长得像数学式，还要看参与运算的变量类型。两个整数做除法，结果按整数处理；比较表达式的结果是真假值；赋值表达式会改变左边变量。

常见表达式可以这样理解：

- 算术表达式：例如 `a + b`、`x * y`，结果是数值。
- 比较表达式：例如 `a < b`、`x == y`，结果是真假值。
- 逻辑表达式：例如 `p && q`、`!ok`，结果也是真假值。
- 赋值表达式：例如 `x = y + 1`，会修改 `x` 的值。
- 复合赋值表达式：例如 `x += 3`，表示把 `x` 增加 3 后再存回 `x`。
- 自增自减表达式：例如 `i++`、`++i`，会改变变量自身。

阅读程序时，看到一个复杂表达式，不要急着从左到右全部心算。可以先把它拆成小块：哪些是算术，哪些是比较，哪些是逻辑连接。拆开后，再按括号和优先级计算。

二、整数除法和取余

整数除法是第一轮高频考点。若 / 两边都是整数，结果会舍去小数部分。例如 7 / 3 的结果是 2，10 / 4 的结果是 2。这里不是四舍五入，而是直接去掉小数部分。

取余运算 % 表示除法后的余数。例如 7 % 3 的结果是 1，10 % 4 的结果是 2。在第一轮中，取余常用于判断整除、奇偶和循环位置。

几个常见判断要分清：

- x % 2 == 0：判断 x 是否为偶数。
- x % 2 == 1：在 x 为非负整数时，常用于判断 x 是否为奇数。
- x % k == 0：判断 x 是否能被 k 整除。
- (i + 1) % n：常用于让下标在 0 到 $n-1$ 之间循环变化。

判断整除时，常用 x % k == 0。不要写成 x / k == 0。x / k == 0 判断的是整数商是否为 0，含义完全不同。例如 2 / 3 == 0 为真，但 2 并不能被 3 整除。

取余还有一个常见用途是限制范围。例如 i = (i + 1) % n 可以让 i 在 0 到 $n-1$ 之间循环变化。当 i 到达 $n-1$ 后，再加 1 得到 n，取余后回到 0。这种写法在环形处理、约瑟夫问题和循环模拟中常出现。

三、赋值不是比较

= 是赋值，== 是比较。这是表达式中最容易被看错的地方之一。

x = y 的意思是把 y 的值存入 x。执行后，x 的值改变。x == y 的意思是判断 x 和 y 是否相等，表达式结果是真或假，不会因为比较本身改变 x 或 y。

看下面的示例。

```
#include <iostream>
using namespace std;

int main() {
    int x = 3, y = 5;
    x = y;
    cout << x << endl;
    cout << (x == y) << endl;
    return 0;
}
```

执行 x = y 后，x 变为 5。第二个输出语句中，x == y 为真，通常输出 1。

在 if 或 while 的条件中，这两个符号的差别更明显。若写成 if (x = 3)，程序会先把 x 改成 3，然后把赋值表达式的结果作为条件。因为 3 是非零值，条件为真。若本意是判断 x 是否等于 3，应写成 if (x == 3)。

第一轮题目中，赋值和比较常作为干扰项出现。遇到 if、while、return 中的表达式时，要看清楚

是 = 还是 ==。只差一个等号，程序行为可能完全不同。

四、比较和逻辑

比较运算符包括 <、<=、>、>=、==、!=。比较表达式的结果是真假值。

逻辑运算符常见三个：

- &&：逻辑与。左右两边都为真时，结果才为真。
- ||：逻辑或。左右两边至少有一边为真时，结果为真。
- !：逻辑非。把真变成假，把假变成真。

例如，判断一个数 x 是否在 1 到 100 之间，应写成 x >= 1 && x <= 100。两个条件都满足时，整体才为真。

判断一个数不在这个范围内，可以写成 x < 1 || x > 100。只要小于 1 或大于 100，整体就为真。

逻辑表达式常见错误是把数学中的连续比较直接写进 C++。数学里可以写 $1 \le x \le 100$，但 C++ 中不能写成 1 <= x <= 100 来表达同样含义。这个表达式会先计算 1 <= x，得到真假值，再把这个真假值与 100 比较，含义已经改变。正确写法是 1 <= x && x <= 100。

另一个常见错误是混淆 “且” 和 “或”。题目说 “同时满足两个条件”，通常用 &&；题目说 “满足其中至少一个条件”，通常用 ||。如果题目要求 “不是某种情况”，要看否定的是单个条件，还是整个表达式。

例如 “不满足 x >= 1 && x <= 100” 可以写成 !(x >= 1 && x <= 100)，也可以写成 x < 1 || x > 100。不要误写成 x < 1 && x > 100，因为一个数不可能同时小于 1 又大于 100。

五、优先级只记常用层次

C++ 运算符有优先级。优先级高的部分先结合，括号可以改变计算顺序。第一轮不要求背完整表，但常用层次必须熟悉。

常见优先级可以按下面顺序理解，从高到低大致是：

1. 括号、数组下标、函数调用，例如 ()、[]、f(x)。
2. 自增自减和逻辑非，例如 ++i、i++、!ok。
3. 乘、除、取余，例如 *、/、%。
4. 加、减，例如 +、-。
5. 大小比较，例如 <、<=、>、>=。
6. 相等比较，例如 ==、!=。
7. 逻辑与 &&。
8. 逻辑或 ||。
9. 赋值和复合赋值，例如 =、+=、-=。

这不是完整优先级表，只保留第一轮中最常用的层次。遇到不确定的表达式，最稳的方法是加括号理解。平时写代码时，也建议用括号把重要逻辑写清楚。

例如：

- `a + b * c` 相当于 `a + (b * c)`。
- `x + 1 < y` 相当于 `(x + 1) < y`。
- `x % 2 == 0 && x > 10` 会先算 `x % 2`，再比较，再做逻辑与。
- `a == b || c == d` 会先分别计算两个相等比较，再做逻辑或。

不要把“从左到右看”当成唯一规则。同一优先级的多数算术运算按从左到右结合，但不同优先级要先按优先级处理。括号的优先级最高，看到括号先处理括号中的表达式。

六、自增自减看使用时机

`++` 表示自增 1，`--` 表示自减 1。它们有前置和后置两种写法。

`++i` 表示先让 `i` 增加 1，再使用新值。`i++` 表示先使用旧值，再让 `i` 增加 1。

`--i` 表示先让 `i` 减少 1，再使用新值。`i--` 表示先使用旧值，再让 `i` 减少 1。

如果自增语句单独出现，`i++` 和 `++i` 的最终效果一样，都会让 `i` 增加 1。例如在常见循环末尾写 `i++`，重点是每轮之后 `i` 增加 1。

如果自增出现在更大的表达式中，就要看使用时机。

例如：

- 若 `i` 原来是 3，执行 `a = i++` 后，`a` 得到旧值 3，然后 `i` 变成 4。
- 接着执行 `b = ++i`，先把 `i` 从 4 增加到 5，再把 5 赋给 `b`。
- 最后 `a=3`，`b=5`，`i=5`。

第一轮中，自增自减题通常不会需要很复杂的表达式。遇到这类题，只要抓住“前置先变再用，后置先用再变”，再按语句顺序追踪即可。

平时写程序时，应尽量避免在一个复杂表达式里多次修改同一个变量。第一轮阅读程序只需要处理题目给出的常见写法，不需要主动写出难以判断的表达式。

七、复合赋值看作先算后存

`+=`、`-=`、`*=`、`/=`、`%=` 是复合赋值运算符。它们的作用是把左边变量与右边表达式计算后，再存回左边变量。

可以这样理解：

- `x += 3` 相当于 `x = x + 3`。
- `x -= 2` 相当于 `x = x - 2`。
- `x *= k` 相当于 `x = x * k`。
- `x /= k` 相当于 `x = x / k`。
- `x %= k` 相当于 `x = x % k`。

例如，`sum += a[i]` 表示把当前 `a[i]` 加到 `sum` 中；`x %= 10` 表示只保留 `x` 除以 10 的余数。

复合赋值常出现在循环中，表示累计、更新或压缩状态。读程序时，不要把 `sum += a[i]` 看成普通加法后没有保存。它会改变 `sum` 的值，下一轮循环使用的是更新后的 `sum`。

例如：

- `s` 初始为 1；
- 执行 `s += 2` 后，`s` 变为 3；
- 执行 `s *= 3` 后，`s` 变为 9；
- 执行 `s %= 5` 后，`s` 变为 4。

复合赋值题的关键是按语句顺序更新变量。不要把所有右边表达式都用初始值计算。

八、短路求值

逻辑与 `&&` 和逻辑或 `||` 有短路求值的特点。

对于 `A && B`，如果 `A` 已经为假，整个表达式一定为假，`B` 不会再被计算。对于 `A || B`，如果 `A` 已经为真，整个表达式一定为真，`B` 不会再被计算。

短路求值在阅读程序中很重要。它不仅影响真假结果，还可能影响函数是否被调用、变量是否被修改。

看下面的示例。

```
#include <iostream>
using namespace std;

int main() {
    int x = 0;
    if (x != 0 && 10 / x > 1) {
        cout << "yes" << endl;
    }

    int y = 5;
    if (y > 0 || y++ > 10) {
        cout << y << endl;
    }
    return 0;
}
```

第一个 `if` 中，`x != 0` 为假。由于 `&&` 左边已经为假，右边的 `10 / x > 1` 不会被计算，所以不会执行除以 0 的计算。

第二个 `if` 中，`y > 0` 为真。由于 `||` 左边已经为真，右边的 `y++ > 10` 不会执行，所以 `y` 仍然是 5，输出 5。

第一轮中，短路求值一般不会考得很绕，但遇到 `&&`、`||` 两边有函数调用、自增自减或除法时，要特别检查右边是否真的会执行。

九、原创例题：表达式求值

例题 6-1 阅读下面的程序，判断输出结果。

```
#include <iostream>
using namespace std;

int main() {
    int a = 5, b = 2, c = 0;
    c = a / b + a % b;
    bool ok = c == 3 || a++ < 5;
    cout << c << " " << a << " " << ok << endl;
    return 0;
}
```

程序输出为（）。

A. `3 5 1`

B. `3 6 1`

C. `2 6 0`

D. `2 5 1`

答案：A

解析：

先计算 `c = a / b + a % b`。`a=5`，`b=2`，所以 `a / b` 为 2，`a % b` 为 1，`c` 变为 3。

接着计算 `ok = c == 3 || a++ < 5`。比较运算 `c == 3` 为真。对于逻辑或 `||`，左边已经为真时，右边不会再计算，所以 `a++ < 5` 不会执行，`a` 仍然是 5。

`ok` 为真，输出时通常显示为 `1`。因此程序输出 `3 5 1`，选 A。

这道题的关键有三点：整数除法、取余、逻辑或的短路求值。若误以为 `a++` 一定会执行，就会选 B。

变式题 若把程序中的 `||` 改为 `&&`，其他代码不变，输出为（）。

A. `3 5 1`

B. `3 6 0`

C. `3 6 1`

D. `2 6 0`

答案：B

解析：

`c` 仍然等于 3。此时表达式变为 `c == 3 && a++ < 5`。左边 `c == 3` 为真，`&&` 还需要继续判断右边，所以会执行 `a++ < 5`。后置自增先使用旧值，旧值 5 与 5 比较，`5 < 5` 为假，然后 `a` 增加为 6。整个逻辑与表达式为假，`ok` 输出为 `0`。所以输出 `3 6 0`。

十、本章要点

表达式题要先判断每一部分产生什么值，再看这些值如何组合。整数除法会舍去小数部分，取余常用于判断整除；`=` 是赋值，`==` 是比较；比较和逻辑表达式的结果是真假值；复合赋值会改变左边变

量；前置自增先变再用，后置自增先用再变。

优先级不确定时，可以用括号理解。做题时要特别注意 `&&` 和 `||` 的短路求值，因为右边表达式不一定会执行。阅读程序时，只要表达式里出现自增自减、赋值、函数调用或除法，就要判断它是否真的被计算。

表达式与运算符是后面分支、循环、数组、递归和算法程序的基础。很多程序阅读题的错误，并不是算法不会，而是某个表达式的值算错了。

0x07 分支与循环

程序不会总是从上到下一行一行执行。遇到分支时，它会根据条件选择执行哪一段；遇到循环时，它会在条件允许的情况下重复执行一段代码。CSP-J 第一轮中，分支与循环几乎贯穿所有阅读程序和完善程序题。很多题并不难在算法，而是难在看错了某个条件、漏算了一次循环，或者把 `break`、`continue` 的作用判断错。

一、分支看条件真假

`if` 语句根据条件决定是否执行某段代码。条件为真时执行对应语句，条件为假时跳过或进入 `else` 分支。

最基本的写法是：

```cpp
#include <iostream>
using namespace std;

int main() {
    int x;
    cin >> x;

    if (x % 2 == 0) {
        cout << "even" << endl;
    } else {
        cout << "odd" << endl;
    }

    return 0;
}
```

这段程序读入一个整数 `x`。如果 `x % 2 == 0` 为真，就输出 `even`；否则输出 `odd`。这里的关键不是记住“偶数程序怎么写”，而是看清条件表达式的结果。

分支题中，条件常见有以下几类：

- 比较条件：例如 `x > 0`、`a[i] <= m`、`x == y`。
- 取余条件：例如 `x % 2 == 0`、`n % i == 0`。
- 逻辑组合：例如 `x >= 1 && x <= 100`。
- 布尔变量：例如 `if (ok)`、`if (!vis[i])`。
- 函数返回值：例如 `if (check(x))`。

读分支时，要先算条件真假，再决定执行路径。不要只看 `if` 后面有没有某个变量名，也不要凭变量名判断。`if (cnt)` 的含义是 `cnt` 转换为真假值后是否为真：当 `cnt` 为 0 时是假，非零时是真。

`if` 后面如果没有花括号，只控制紧跟着的一条语句。例如：

```
if (x > 0) x++; cout << x;
```

这里无论 `x > 0` 是否成立，`cout << x;` 都会执行。纸面阅读时，花括号范围必须看清楚。为了减少误读，本书中的代码通常会保留花括号，但真题或程序片段中不一定总是这样写。

二、else 只和最近的未匹配 if 配对

嵌套分支中，`else` 的归属容易看错。在 C++ 中，一个 `else` 会和它前面最近的、还没有配对的 `if` 配对。若代码中没有花括号，这一点尤其容易出题。

例如：

```
if (a > 0) if (b > 0) cout << 1; else cout << 2;
```

这里的 `else` 配对的是里面的 `if (b > 0)`，不是外面的 `if (a > 0)`。也就是说：

- 若 `a <= 0`，整段代码不输出；
- 若 `a > 0` 且 `b > 0`，输出 `1`；
- 若 `a > 0` 且 `b <= 0`，输出 `2`。

如果想让 `else` 配对外层 `if`，必须用花括号明确范围。阅读程序题不要求学生改写代码，但必须能判断 `else` 到底属于哪一层。

多分支语句常写成 `if`、`else if`、`else`。读这种结构时，要记住它是从上到下依次判断，一旦某个条件成立并执行了对应分支，后面的分支就不会再判断。

例如：

- `if (x >= 90)` 处理 90 分及以上；
- `else if (x >= 60)` 处理的是小于 90 且不小于 60；
- `else` 处理的是小于 60。

第二个条件虽然只写了 `x >= 60`，但它处在 `else if` 中，前面的 `x >= 90` 已经不成立，所以实际范围是 $60 \le x < 90$。很多选择题会利用这一点设置干扰项。

三、循环先问执行几次

循环语句会重复执行一段代码。第一轮中最常见的是 `for` 和 `while`。读循环时，最先要问的不是“这段程序在做什么算法”，而是“循环体会执行几次”。

`for` 循环一般由初始化、循环条件、更新三部分组成。以 `for (int i = 1; i <= n; i++)` 为例：

- 初始化：`int i = 1`，循环变量从 1 开始；
- 条件：`i <= n`，每轮开始前判断；
- 更新：`i++`，每轮循环体执行完后，`i` 增加 1。

当 $n = 5$ 时，`i` 依次取 $1, 2, 3, 4, 5$，循环体执行 5 次。若写成 `i < n`，则 `i` 依次取 $1, 2, 3, 4$，循环体执行 4 次。只差一个等号，次数就不同。

常见循环范围要能迅速判断：

- `for (int i = 0; i < n; i++)`：`i` 取 0 到 $n - 1$，共 n 次。
- `for (int i = 1; i <= n; i++)`：`i` 取 1 到 n，共 n 次。
- `for (int i = 0; i <= n; i++)`：`i` 取 0 到 n，共 $n + 1$ 次。
- `for (int i = n; i >= 1; i--)`：`i` 取 n 到 1，共 n 次。
- `for (int i = 1; i <= n; i += 2)`：`i` 每次增加 2，只访问部分位置。

循环边界是阅读程序的高频失分点。做题时可以在草稿上写出循环变量前几项和最后一项，不要只凭印象判断。

四、while 看变量怎样变

`while` 循环在每轮开始前判断条件。条件为真，就执行循环体；条件为假，就退出循环。

`while` 的更新语句不写在循环头中，常常藏在循环体里。因此读 `while` 时，要找出“循环条件中的变量在哪里改变”。

例如：

```cpp
#include <iostream>
using namespace std;

int main() {
    int n;
    cin >> n;

    int cnt = 0;
    while (n > 0) {
        cnt++;
        n /= 10;
    }

    cout << cnt << endl;
    return 0;
}
```

这段程序中，循环条件是 `n > 0`。循环体里有 `n /= 10`，每一轮会删去十进制表示中的最后一位。`cnt` 记录循环执行次数。

若输入 `1234`：

- 第 1 轮后，`n` 变为 123，`cnt` 变为 1；
- 第 2 轮后，`n` 变为 12，`cnt` 变为 2；
- 第 3 轮后，`n` 变为 1，`cnt` 变为 3；
- 第 4 轮后，`n` 变为 0，`cnt` 变为 4；

- 再次判断 `n > 0` 为假，循环结束。

程序输出 4。这类题的关键是：条件在每轮开始前判断，更新发生在循环体中。最后一次把 n 改成 0 后，不会再执行下一轮。

如果 while 条件中的变量没有改变，就可能形成无法结束的循环。完善程序题中，空缺若负责更新循环变量，就必须保证循环能向结束条件靠近。阅读程序题中，若题目问某个修改是否会导致无法结束，也要看条件变量是否仍会变化。

五、do while 至少执行一次

do while 的特点是先执行循环体，再判断条件。因此它至少执行一次。

基本结构是：

```
do { 循环体 } while (条件);
```

最后的分号不能漏。第一轮中 do while 出现频率不如 for 和 while 高，但它的“先执行一次”很适合出判断题。

例如，若 x=0：

while (x > 0) 的循环体一次也不会执行；

do { ... } while (x > 0); 的循环体会先执行一次，然后再判断 x > 0。由于条件为假，执行一次后结束。

遇到 do while，不要直接按 while 的习惯先判断条件。先把循环体执行一次，再看条件是否继续。这是它和普通 while 的主要差别。

六、break 和 continue

break 用于跳出当前循环。执行到 break 后，当前循环立即结束，程序继续执行循环后面的语句。

continue 用于结束当前这一轮循环。执行到 continue 后，本轮循环体中后面的语句不再执行，程序进入下一轮循环的判断或更新。

二者的区别可以这样记：

- break：整个循环结束。
- continue：本轮结束，循环本身还可能继续。

例如，下面这段程序统计从 1 开始连续试除时，第一个能整除 n 的数。

for (int i = 2; i <= n; i++) 中，如果遇到 n % i == 0，执行 break，后面的 i 不再尝试。此时 break 的作用是提前结束循环。

而在统计某些数的和时，常常会用 continue 跳过不需要的数。例如遇到奇数就 continue，后面的累加语句只处理偶数。

读到 break 和 continue 时，要注意它们影响的是“当前所在的最近一层循环”。在嵌套循环中，内层循环里的 break 只跳出内层循环，不会直接跳出外层循环。若程序有两层循环，必须看 break 写在哪一层。

七、嵌套循环不要直接乱乘

嵌套循环是第一轮阅读程序和复杂度题的常见来源。外层循环每执行一次，内层循环可能执行若干次。若内层次数固定，可以相乘；若内层次数随外层变量变化，就要逐步计算。

例如：

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

里面有：

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

内层每次都执行 n 次，外层执行 n 次，总执行次数是 n^2。

但若内层写成：

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

当 i=1 时，内层执行 1 次；当 i=2 时，内层执行 2 次；一直到 i=n 时，内层执行 n 次。总次数是 $1+2+\cdots+n$，大致是 n^2 级别，但具体次数不是 $n \times n$。

若题目只问时间复杂度，通常判断数量级即可；若题目问具体输出，就要按具体输入追踪。对于小输入，可以把循环变量的取值列出来，避免漏掉某一轮。

嵌套循环还常用于二维数组、字符串比较、枚举数对。读这类程序时，要看清楚内外层变量分别控制什么。不要把 i 和 j 混用，也不要把 a[i][j] 看成 a[j][i]。

八、原创例题：分支与循环追踪

例题 7-1 阅读下面的程序，判断输出结果。

```cpp
#include <iostream>
using namespace std;

int main() {
    int n = 7;
    int s = 0;

    for (int i = 1; i <= n; i++) {
        if (i % 2 == 0) {
            continue;
        }
        if (i == 5) {
            break;
        }
        s += i;
```

```
    }

    cout << s << endl;
    return 0;
}
```

程序输出为（）。

A. 4
B. 9
C. 16
D. 25

答案：A

解析：
循环变量 i 从 1 到 7。每一轮先判断 `i % 2 == 0`。如果 i 是偶数，就执行 continue，本轮后面的语句不再执行。

逐轮看：

i=1 时，不是偶数，也不等于 5，执行 `s += i`，s 变为 1。

i=2 时，是偶数，执行 continue，不累加。

i=3 时，不是偶数，也不等于 5，执行 `s += i`，s 变为 4。

i=4 时，是偶数，执行 continue，不累加。

i=5 时，不是偶数，继续往下判断 `i == 5`，成立，执行 break。整个循环结束，后面的 i=6、i=7 都不会再处理。

最后输出 s，即 4，选 A。

这道题的关键是区分 continue 和 break。continue 只跳过当前这一轮，break 直接结束整个循环。

变式题 若把例题中的两个 if 交换顺序，改成先判断 `i == 5`，再判断 `i % 2 == 0`，输出是否改变？

A. 不改变，仍输出 4
B. 改变，输出 9
C. 改变，输出 16
D. 程序无法通过编译

答案：A

解析：
交换后，i=1 和 i=3 仍会累加，s 仍先变为 4。当 i=5 时，先判断 `i == 5`，成立后执行 break，循环结束。由于 5 不是偶数，原程序中也不会被 continue 跳过，而是会执行到 break。所以输出不改变。

这个变式提醒我们：改变判断顺序不一定改变结果，要看有没有输入会在两个判断之间产生不同路径。若交换的是“偶数判断”和“是否等于 4”之类的条件，结果就可能不同。

九、完善程序中的分支与循环

完善程序题常把空缺放在分支条件或循环条件中。补这类空时，要先判断空缺控制什么。

如果空缺在 `if (...)` 中，它控制是否进入分支。此时应问：题意中什么情况要执行分支里的语句？例如分支里写着 `cnt++`，那条件很可能是“当前元素满足计数要求”。

如果空缺在 `while (...)` 或 `for (...; ...; ...)` 的条件中，它控制循环是否继续。此时应问：程序什么时候还没有处理完？什么时候应该停止？循环条件不能只看某个变量是否存在，而要看它是否能覆盖题目要求的范围。

如果空缺在循环更新位置，它控制下一轮处理什么。此时应问：循环变量是否向结束方向变化？会不会跳过需要处理的位置？会不会卡在同一个值上？

例如，枚举 1 到 n 的整数，循环通常从 `1` 开始，到 `n` 结束，每次 `i++`。若题目要求处理数组下标，循环可能从 `0` 到 `n - 1`。这两类范围都常见，不能把它们混在一起。

分支与循环的空缺要结合上下文判断。只看选项中的某个条件是否“常见”，不一定能选对。要看分支内做了什么，循环体内更新了什么，最后输出依赖哪个变量。

十、本章要点

分支题先看条件真假，再看执行哪一段。`else` 会和最近的未匹配 `if` 配对；`else if` 是从上到下依次判断，前面成立后，后面不再判断。

循环题先看执行次数，再追踪关键变量。`for` 要看起点、条件和更新；`while` 要看条件变量在哪里改变；`do while` 至少执行一次。`break` 会结束当前循环，`continue` 只结束当前这一轮。

嵌套循环不能直接凭感觉相乘，要看内层次数是否固定。阅读程序时，循环边界、下标范围和提前结束位置是最需要检查的地方。掌握分支与循环，后面的数组、字符串、递归和基础算法才能读得稳。

0x08 函数与递归

函数把一段程序封装起来，用一个名字调用。递归是一种特殊的函数调用：函数在执行过程中又调用自己。CSP-J 第一轮中，函数和递归常出现在阅读程序题与完善程序题中。题目不一定要求写出复杂函数，但会要求判断函数返回什么、参数如何变化、递归什么时候停止。

很多学生读函数时只看函数名，例如看到 `check` 就认为它一定是在检查合法性，看到 `solve` 就认为它一定能求出最终答案。这样的读法不稳。函数名只能提供线索，真正决定程序行为的是参数、函数体和返回值。

本章重点讲三件事：怎样读普通函数，怎样读 `return`，怎样追踪简单递归。后面章节会继续使用这些方法分析数组、字符串、搜索和动态规划程序，本章不展开那些后续算法。

一、函数先看四件事

读一个函数时，不要一开始就逐行翻译。先看四件事：返回类型、函数名、参数列表、函数体。

例如：

```cpp
#include <iostream>
using namespace std;

int add(int a, int b) {
    int s = a + b;
    return s;
}

int main() {
    int x = 3, y = 5;
    cout << add(x, y) << endl;
    return 0;
}
```

这段程序中，`add` 是函数名，`int add(...)` 表示它返回一个 `int` 类型的值。参数列表是 `int a, int b`，说明调用它时要给两个整数。函数体中先计算 `s = a + b`，再 `return s`。因此 `add(3, 5)` 的结果是 8。

读函数时，可以按下面顺序问：

第一，函数返回什么类型？如果返回类型是 `int`，它应该返回整数；如果是 `bool`，它通常返回真假值；如果是 `void`，它不返回普通值。

第二，函数需要哪些参数？参数是函数从调用处拿到的信息。参数不同，函数结果可能不同。

第三，函数体中修改了哪些变量？局部变量只在函数内部使用，全局变量可能影响函数外部。

第四，return 在哪里？如果函数中有多个 return，要看不同输入会走到哪一个。

函数题最容易错在“只看函数名，不看函数体”。例如一个名叫 calc 的函数可能求和，也可能计数；一个名叫 check 的函数可能检查奇偶，也可能检查是否能被整除。题目中真正可靠的是代码本身。

二、调用时先算实参

函数调用时，括号中的表达式叫实参。函数定义中的变量名叫形参。调用发生时，实参的值会传给形参。

例如 add(x + 1, y * 2)，调用前要先计算 x + 1 和 y * 2 的值，再把这两个值传入函数。函数内部看到的是形参 a、b，不是调用处原来的表达式。

在第一轮中，普通函数参数通常按值传递。可以先理解为：函数得到的是传入值的一份副本。函数内部修改形参，一般不会改变调用处的原变量。

看这个片段：

```
#include <iostream>
using namespace std;

void change(int x) {
    x = x + 10;
}

int main() {
    int a = 5;
    change(a);
    cout << a << endl;
    return 0;
}
```

change(a) 调用时，a 的值 5 传给形参 x。函数内部把 x 改为 15，但这里的 x 只是函数内部的变量。回到 main 后，a 仍然是 5，所以输出 5。

这类题常出现在阅读程序中。若函数参数不是引用，也不是数组等特殊形式，函数内部对形参的普通赋值不会改变外面的变量。后面第 9 章会讲数组，第 10 章会讲指针，本章先掌握普通整数参数的情况。

如果函数返回了修改后的值，调用处要接住这个返回值才会影响外部变量。例如 a = f(a); 和单独写 f(a); 的效果可能不同。前者把函数返回值重新赋给 a，后者只是调用函数，返回值如果没有使用，就不会改变 a。

三、return 会结束当前函数

return 的作用是结束当前函数，并把结果交回调用处。它不是结束整个程序，除非它出现在 main 函数中，或者程序后续没有其他事情可做。

例如：

```
return x + 1;
```

这句会先计算 `x + 1`，然后把这个结果作为函数值返回。执行到 `return` 后，函数中后面的语句不会再执行。

函数中可以有多个 `return`，但一次调用只会执行其中一个。常见写法是先处理特殊情况，再处理一般情况。

例如，判断一个数是否为正数：

- 若 `x > 0`，返回 `true`；
- 否则返回 `false`。

读这种函数时，要按输入判断会走哪条路径。不要看到最后有 `return false`，就认为函数总是返回 `false`；也不要看到前面有 `return true`，就忽略它的条件。

`void` 函数没有普通返回值，但可以单独写 `return;`，表示提前结束函数。它不会返回一个整数或真假值。完善程序题中，如果空缺在 `void` 函数里，就不能填 `return 1;` 这类与返回类型不匹配的内容。

阅读程序时，要特别注意“函数返回值有没有被使用”。例如 `check(x);` 只是调用函数，若返回值没有参与赋值、输出或条件判断，它的返回值就被丢弃了。`if (check(x))` 则会用返回值决定是否进入分支。

四、局部变量和全局变量

定义在函数内部的变量是局部变量。它只在当前函数的这一部分代码中有效。不同函数中可以有同名局部变量，它们不是同一个变量。

定义在函数外部的变量是全局变量。多个函数都可以访问它。全局变量一般会被初始化为 0，这在阅读程序题中有时会影响结果。

例如，若函数外有 `int cnt;`，函数中每次执行 `cnt++`，修改的是同一个全局变量。若函数内写了 `int cnt = 0;`，那就是函数内部新定义的局部变量，会遮住外面的同名变量。

读函数时，要看变量定义在哪里。不要因为名字相同，就认为它们一定是同一个变量。尤其在递归函数中，形参和局部变量每次调用都会有本次调用自己的值；而全局变量会在多次调用之间一直保留变化结果。

第一轮不会要求深入掌握复杂作用域规则，但要能判断这几种常见情况：

局部变量只影响当前函数内部。

形参是函数调用时得到的变量，普通赋值一般不改变调用处的原变量。

全局变量可以被多个函数共享，函数中修改它后，后续代码看到的是修改后的值。

同名时，内层定义的变量会遮住外层变量。

五、递归先找出口

递归函数会调用自己。读递归时，第一件事不是展开很多层，而是找递归出口。递归出口就是不再继续调用自己的情况。

例如，下面的函数计算 $1+2+\cdots+n$：

```
#include <iostream>
using namespace std;

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

int main() {
    int n;
    cin >> n;
    cout << sum(n) << endl;
    return 0;
}
```

这里的出口是 `n == 1`。当 `n` 等于 1 时，函数直接返回 1，不再继续调用 `sum`。一般情况是 `return sum(n - 1) + n;`，它把问题变成更小的 `sum(n - 1)`。

若输入 `4`，可以这样追踪：

`sum(4)` 等于 `sum(3) + 4`。

`sum(3)` 等于 `sum(2) + 3`。

`sum(2)` 等于 `sum(1) + 2`。

`sum(1)` 返回 1。

然后往回代入，`sum(2)=3`，`sum(3)=6`，`sum(4)=10`。所以输出 `10`。

读递归时，要检查两件事：

第一，有没有出口。没有出口的递归无法按正常方式结束。

第二，每次递归调用是否更接近出口。例如 `sum(n - 1)` 会让 `n` 逐渐靠近 1；如果写成 `sum(n)`，参数没有变化，就无法到达出口；如果写成 `sum(n + 1)`，还可能离出口越来越远。

六、递归返回值要往回看

递归不是只看最深一层。到达出口后，结果还要一层一层返回。很多阅读程序题会让学生误以为“递归到出口就结束了”，从而漏掉回来的计算。

以上面的 `sum(4)` 为例，递归调用的顺序是一路从 4 到 1，但结果计算要从 `sum(1)` 往回返回：

`sum(1)` 返回 1。

`sum(2)` 返回 `sum(1) + 2`，即 3。

`sum(3)` 返回 `sum(2) + 3`，即 6。

`sum(4)` 返回 `sum(3) + 4`，即 10。

如果递归调用后面还有运算，例如 `return f(n - 1) + n;`，就必须等 `f(n - 1)` 的结果回来后再加 `n`。如果写成：

```
return n * f(n - 1);
```

就表示回来后还要乘上当前层的 `n`。

如果递归函数没有使用返回值，例如只写 `f(n - 1);`，后面又没有把它赋给变量或返回出去，那么这次调用的结果可能被丢弃。完善程序题中，递归调用前是否需要 `return` 是常见空缺。

例如一个返回 `int` 的函数中，若写：

```
f(n - 1);
```

这只是调用了函数，没有把结果返回给外层。若函数本意是递归求值，常常需要写：

```
return f(n - 1);
```

或者写：

```
return f(n - 1) + n;
```

具体填法取决于题意和后续计算。

七、递归参数看含义变化

有些递归不只是把一个数减小，还会改变多个参数。读这类递归时，要给每个参数写含义。

例如，汉诺塔程序中常见参数可能表示：当前要移动的圆盘数量、起始柱、中转柱、目标柱。一次递归调用不是随便交换参数，而是根据任务阶段变化：

先把上面一部分从起始柱移到中转柱。

再把最大圆盘从起始柱移到目标柱。

最后把上面一部分从中转柱移到目标柱。

如果不知道参数含义，就会觉得 `src`、`tmp`、`tgt` 的顺序很难记。若把它们写成“起点、中转、目标”，每次递归调用为什么这样交换，就容易判断。

搜索类递归中，参数可能表示当前位置、当前步数、当前和、是否访问过。读这类程序时，也要看参数是否表示“下一层状态”。本章不展开搜索算法，只强调读法：多参数递归必须先确定每个参数的含义，再判断递归调用是否符合题意。

完善程序题常把递归参数作为空缺。遇到这种题，不要只看参数类型是否匹配。类型匹配只能说明代码可能编译，不能说明逻辑正确。要把参数放回题意中，判断这次调用解决的是哪一个子问题。

八、函数与递归中的常见错误

函数与递归题常见错误有以下几类。

第一，只看函数名。函数名可能有提示，但不能代替代码分析。要看参数、条件、循环和返回值。

第二，忽略返回值。函数返回了结果，但调用处没有接收或使用，这个结果就不会自动改变外部变量。

第三，把形参和实参混成一个变量。普通整数参数传入函数后，函数内部修改形参，一般不会改变调用处变量。

第四，看错提前返回。执行到 return 后，当前函数结束，后面的语句不会再执行。

第五，递归只看下去，不看回来。递归出口给出最小问题的结果，最终答案还要沿着调用关系返回并继续计算。

第六，递归参数没有靠近出口。若参数不变或变化方向错误，递归无法按题意结束。

第七，多个参数含义不清。参数一多，不能只靠位置硬记，要给每个参数写出含义。

这些错误在选择题、阅读程序和完善程序中都会出现。函数章节虽然不像数组和循环那样到处都有下标，但它的错法往往更隐蔽：程序看起来能运行，结果却不是题意要求的值。

九、原创例题：递归返回值

例题 8-1 阅读下面的程序，判断输出结果。

```
#include <iostream>
using namespace std;

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

int main() {
    cout << f(4) << endl;
    return 0;
}
```

程序输出为（）。

A. 8

B. 14

C. 20

D. 28

答案：C

解析：
先找递归出口。`n == 1` 时，`f(1)` 返回 2。

再看一般情况：`f(n)` 返回 `f(n - 1) + 2 * n`。也就是说，每一层都要先求出前一层的结果，再加上当前的 $2n$。

逐层计算：

`f(1)=2`。

`f(2)=f(1)+4=6`。

`f(3)=f(2)+6=12`。

`f(4)=f(3)+8=20`。

所以程序输出 `20`，选 C。

这道题的关键是从出口往回算。若只看到每层有 `2 * n`，直接把 2×4 当成答案，就会误选 A；若漏掉出口返回的 2，也会算错。

变式题 若把函数中的递归语句改为：

```
return f(n - 1) * 2;
```

其他代码不变，程序输出为 ()。

A. `8`
B. `12`
C. `16`
D. `20`

答案：C

解析：
出口仍然是 `f(1)=2`。修改后，每一层返回前一层结果的 2 倍。

因此：

`f(1)=2`。

`f(2)=4`。

`f(3)=8`。

`f(4)=16`。

所以输出 `16`，选 C。变式题说明，递归出口不变时，一般递归式的变化会直接改变返回过程，不能沿用原题的加法计算。

十、完善程序中的函数空缺

完善程序题中，函数空缺常出现在四个位置。

第一，函数返回类型。若函数用于判断，返回类型常是 bool；若函数用于计算个数、和、最值，返回类型常是 int 或 long long。判断返回类型时，要看调用处怎样使用函数结果。

第二，函数参数。参数要包含解决问题所需的信息。若函数需要处理区间，常见参数可能有左右端点；若函数需要递归处理规模，常见参数可能有 n 或当前位置。

第三，递归出口。出口要能处理最小情况，并且必须与题意一致。求和问题的出口、计数问题的出口、判断问题的出口，返回值可能不同。

第四，递归调用与返回语句。若函数要把子问题结果继续交给外层，常常需要 return。如果只调用函数但不返回或不保存结果，结果可能不会传出去。

补函数空缺时，可以先用一个小输入检查。递归题常用 1、2、3 代入。若 $n = 1$ 时都不对，出口多半有问题；若 $n = 2$ 时不对，递归调用或返回表达式多半有问题；若参数顺序错了，小输入也容易暴露问题。

十一、本章要点

函数要看返回类型、参数、函数体和 return。函数名只能提示意图，不能代替分析。普通参数传入函数后，函数内部修改形参，一般不会改变调用处原变量；返回值如果没有被使用，也不会自动影响外部变量。

return 会结束当前函数，并把结果交回调用处。函数中有多个 return 时，要根据条件判断执行哪一个。void 函数不返回普通值，可以用单独的 return; 提前结束。

递归先找出口，再看参数是否靠近出口，最后从出口往回计算返回值。多参数递归要写清每个参数的含义，不能只按参数位置硬猜。读懂函数与递归，后面程序阅读中的许多“看起来很长”的代码，就能拆成清楚的调用关系。

0x09 数组与字符串

数组和字符串是 CSP-J 第一轮程序阅读中的高频内容。循环边界、下标起点、字符串长度、字符访问、数组初始化，都是常见失分点。很多题并不是算法复杂，而是把 `a[i]`、`a[i - 1]`、`s[j]`、`s.size()` 这些细节混在一起考。

数组可以看成一排同类型数据，字符串可以看成一排字符。读这类程序时，最重要的不是背函数，而是看清“有多少个元素”“下标从哪里开始”“最后一个位置在哪里”“循环有没有越界”。

本章只讲第一轮需要稳定掌握的数组与字符串行为。更深入的指针、结构体和链表会在下一章处理；字符串算法会在后面的字符串处理章节中再展开。

一、数组是一排同类型数据

数组用于保存多个同类型数据。例如 `int a[5];` 表示定义了一个名为 `a` 的数组，可以保存 5 个 `int` 类型的值。它的合法下标是 $0, 1, 2, 3, 4$。第一个元素是 `a[0]`，最后一个元素是 `a[4]`。

如果数组长度是 n，常见合法下标是 0 到 $n-1$。这是数组题中最基本、也最容易被看错的规则。

例如，`int a[10];` 有 10 个元素，合法下标通常是 0 到 9。`a[0]` 是第一个元素，`a[9]` 是最后一个元素，`a[10]` 不属于这 10 个元素。

CSP-J 第一轮题目中，经常会说明“程序输入不超过数组或字符串定义的范围”。这表示题目一般不会让你从非法输入角度去攻击程序。但即使如此，仍然要看代码本身访问了哪些下标，因为判断题可能会专门问某个输入是否会越界。

数组的每个元素都可以像普通变量一样使用。`a[i] = a[i] + 1` 会把第 `i` 个元素增加 1；`sum += a[i]` 会把第 `i` 个元素加到 `sum` 中；`if (a[i] > 0)` 会判断第 `i` 个元素是否大于 0。

读数组程序时，不要只看数组名，还要看当前访问的是哪个下标。真正参与计算的是 `a[0]`、`a[1]`、`a[i]` 这样的元素，而不是整个数组名。

二、从 0 开始和从 1 开始

虽然 C++ 数组的第一个位置是 `a[0]`，但竞赛程序中也常把数组开大一些，然后从 `a[1]` 开始使用。这两种写法都很常见。阅读程序时，必须先判断当前程序采用哪一种习惯。

看下面这段代码：

```
int a[105], n;
cin >> n;

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

```
    cin >> a[i];
}

int sum = 0;
for (int i = 0; i < n; i++) {
    sum += a[i];
}
cout << sum << endl;
```

这段程序读入并处理的是 a[0] 到 a[n - 1]，共 n 个元素。这是从 0 开始使用数组的写法。

如果代码改成 for (int i = 1; i <= n; i++) cin >> a[i];，那么有效数据就是 a[1] 到 a[n]。此时 a[0] 通常不用。

两种写法本身都可以成立，但不能混用。若输入时从 a[1] 到 a[n]，后面处理却写成 for (int i = 0; i < n; i++)，就会访问 a[0] 到 a[n - 1]，漏掉 a[n]，还多用了 a[0]。

阅读数组程序时，可以先在草稿上写一句：

有效元素是 a[0] 到 a[n - 1]。

或者：

有效元素是 a[1] 到 a[n]。

这句话很短，却能减少很多下标错误。尤其在程序较长时，前面读入用的是哪套下标，后面循环是否保持一致，一定要检查。

三、循环边界决定访问范围

数组通常和循环一起出现。看数组程序时，要同时看循环变量和数组下标。

若代码写成 for (int i = 0; i < n; i++) sum += a[i];，当 $n = 5$ 时，访问的是 a[0]、a[1]、a[2]、a[3]、a[4]。这正好是从 0 开始的 5 个元素。

若代码写成 for (int i = 1; i <= n; i++) sum += a[i];，当 $n = 5$ 时，访问的是 a[1] 到 a[5]。这适合从 1 开始存数据的程序，不适合只使用 a[0] 到 a[4] 的程序。

数组题中，循环边界常见错误有三种。

第一，把 < n 看成 <= n。这会多访问一个位置。对于从 0 开始、长度为 n 的数组，i < n 才会停在 a[n - 1]。

第二，把 i = 0 看成 i = 1。这会漏掉第一个位置，或者错用本来不用的位置。

第三，把 a[i] 和 a[i - 1] 混淆。若循环从 i = 1 开始，访问 a[i - 1] 实际上是从 a[0] 开始。

有些程序故意让循环变量和数组下标不同。例如 for (int i = 1; i <= n; i++) sum += a[i - 1];，循环变量 i 从 1 到 n，但访问的是 a[0] 到 a[n - 1]。判断这种程序时，不要只看循环变量范围，要看真正放在方括号里的表达式。

四、数组初始化与默认值

数组元素在使用前应当有确定的值。全局数组一般会被初始化为 0。局部数组如果没有初始化，元素值可能不可预测。第一轮题目通常不会故意依赖很偏的情况，但“数组有没有被赋值”仍然是程序阅读中的重点。

例如，`int cnt[10] = {};` 会把数组元素初始化为 0。若用于计数，初值为 0 很合理。

如果数组定义在函数外，例如 `int a[1000];`，它是全局数组，初始元素通常为 0。如果数组定义在函数内部，例如 `int a[1000];`，并且没有给初值，就不能按“全是 0”来理解，除非题目或代码明确进行了赋值。

阅读程序中，还要看数组是否被全部赋值。若只读入了 `a[1]` 到 `a[n]`，那么 `a[0]` 通常不是有效输入数据；若只对部分下标赋值，后面访问未赋值位置就要谨慎判断。

计数数组是第一轮常见写法。例如用 `cnt[x]++` 统计某个数出现次数。读这种程序时，要看 `cnt` 是否初始为 0，也要看 `x` 是否在数组范围内。题目若保证输入范围合法，就主要关注计数逻辑：每出现一次，对应位置是否增加 1。

五、二维数组看行和列

二维数组可以看成有行有列的数据。程序中用两个下标访问它，例如 `a[i][j]`。通常可以把第一个下标看成行，第二个下标看成列，但最终仍要结合题意和输入循环判断。

若代码中外层循环是 `i`，内层循环是 `j`，并且读入语句是 `cin >> a[i][j];`，通常表示按行读入。`i` 控制第几行，`j` 控制第几列。

二维数组题容易把行列看反。比如 `a[i][j]` 和 `a[j][i]` 不是同一个位置。若程序访问上下左右相邻位置，常见变化是：

上方位置改变行号，可能写成 `a[i - 1][j]`。

下方位置改变行号，可能写成 `a[i + 1][j]`。

左侧位置改变列号，可能写成 `a[i][j - 1]`。

右侧位置改变列号，可能写成 `a[i][j + 1]`。

本章只要求能读懂基本下标变化。搜索中的方向数组会在后面的搜索章节中再讲。当前阶段不要把二维数组题想得过深，先保证行列、边界和访问范围不出错。

六、字符串是一排字符

`string` 可以保存一串字符。例如 `string s = "csp";`，这个字符串长度为 3，其中 `s[0]` 是 `'c'`，`s[1]` 是 `'s'`，`s[2]` 是 `'p'`。

和数组一样，字符串下标也通常从 0 开始。若字符串长度为 n，最后一个字符的位置是 `s[n - 1]`，不是 `s[n]`。

字符串长度可以用 `s.size()` 或 `s.length()` 得到。在第一轮程序阅读中，这两个用法通常都表示字符串长度。

看下面这段程序：

```cpp
#include <iostream>
#include <string>
using namespace std;

int main() {
    string s = "hello";
    cout << s.size() << endl;
    cout << s[0] << " " << s[4] << endl;
    return 0;
}
```

`"hello"` 的长度是 5，所以第一行输出 `5`。第一个字符 `s[0]` 是 `'h'`，最后一个字符 `s[4]` 是 `'o'`，所以第二行输出 `h o`。

读字符串程序时，要特别注意字符和字符串的区别。`'a'` 是字符，`"a"` 是字符串。`s[i]` 取出的通常是一个字符，可以与 `'a'`、`'0'` 这类字符常量比较。

七、字符串遍历看长度

遍历字符串时，常见写法是 `for (int i = 0; i < s.size(); i++)`。若 `s.size()` 是 n，循环访问下标 0 到 $n-1$，正好覆盖所有字符。

字符串题中常见错误和数组类似。

第一，把 `i < s.size()` 看成 `i <= s.size()`。若字符串长度为 n，`s[n]` 不是最后一个字符。

第二，把字符位置和第几个字符混淆。第 1 个字符是 `s[0]`，第 k 个字符是 `s[k - 1]`。

第三，把 `s.size()` 当成最后下标。`s.size()` 表示长度，不表示最后一个合法位置。最后一个合法位置是 `s.size() - 1`，前提是字符串非空。

若程序倒序遍历字符串，常见起点是 `s.size() - 1`。此时要注意字符串是否可能为空。CSP-J 第一轮题目通常会给出输入范围，若保证字符串非空，就按非空处理；若没有保证，要谨慎判断。

八、字符与数字字符

字符串中常出现数字字符，例如 `'0'`、`'1'`、`'9'`。它们是字符，不是整数。若要把数字字符转成对应整数，常见写法是 `int x = s[i] - '0';`。

例如，`'7' - '0'` 的结果是整数 7。这是因为数字字符在常见编码中按顺序排列。

反过来，若有一个 0 到 9 的整数 `x`，想得到对应数字字符，可以写 `char c = char('0' + x);`。

字符题中容易把 `'1'` 和 `1` 混在一起。`'1'` 是字符，`1` 是整数。若程序输出字符 `'1'`，屏幕上显示的是 `1`；但它参与运算时，不能直接把它当成整数 1，除非做了类似 `s[i] - '0'` 的转换。

统计字符串中数字和字母时，常见判断有三类：

`s[i] >= '0' && s[i] <= '9'` 用于判断是否为数字字符。

`s[i] >= 'a' && s[i] <= 'z'` 用于判断是否为小写字母。

`s[i] >= 'A' && s[i] <= 'Z'` 用于判断是否为大写字母。

这些判断依赖字符按顺序排列。第一轮题目中通常只在数字字符、大小写字母范围内使用，不会要求掌握复杂编码细节。

九、输入字符串看空格

使用 `cin >> s` 读入 `string` 时，它会读入一个不含空白字符的单词。遇到空格、换行等空白字符时，当前字符串读入结束。

例如输入为 `hello world`，若代码是 `string s; cin >> s;`，那么 `s` 得到的是 `"hello"`，不是整行 `"hello world"`。

如果要读入整行，可以使用 `getline(cin, s)`。不过在第一轮阅读程序中，题目一般会给出具体代码和输入。读题时只要根据代码判断它读入了哪一部分。

如果程序先用 `cin >> n` 读入整数，再用 `getline(cin, s)` 读入整行，可能会受到前面换行符影响。这类细节在 CSP-J 第一轮中不是主线高频，但若题目给出相关代码，就要按实际读入过程判断。

本书后续讲字符串处理时，主要使用不含空格的字符串。第一轮常见题也多围绕单词式字符串、字符统计、回文判断、子串和长度展开。

十、原创例题：数组与字符串追踪

例题 9-1 阅读下面的程序，判断输出结果。

```cpp
#include <iostream>
#include <string>
using namespace std;

int main() {
    int a[6] = {2, 5, 1, 4, 3, 6};
    string s = "Csp2024";

    int x = 0;
    for (int i = 0; i < 6; i++) {
        if (a[i] % 2 == i % 2) {
            x += a[i];
        }
    }

    int y = 0;
    for (int i = 0; i < s.size(); i++) {
        if (s[i] >= '0' && s[i] <= '9') {
            y = y * 10 + (s[i] - '0');
```

```
        }
    }

    cout << x << " " << y << endl;
    return 0;
}
```

程序输出为（）。

A. `7 2024`

B. `12 2024`

C. `7 8`

D. `12 8`

答案：A

解析：

先看数组部分。数组 `a` 的下标从 0 到 5，对应值依次为 $2, 5, 1, 4, 3, 6$。

循环中判断 `a[i] % 2 == i % 2`，也就是判断 `a[i]` 的奇偶性是否和下标 `i` 的奇偶性相同。

当 `i=0` 时，`a[0]=2`，二者都是偶数，满足条件，`x` 加 2。

当 `i=1` 时，`a[1]=5`，二者都是奇数，满足条件，`x` 加 5。

当 `i=2` 时，`a[2]=1`，值是奇数，下标是偶数，不满足条件。

当 `i=3` 时，`a[3]=4`，值是偶数，下标是奇数，不满足条件。

当 `i=4` 时，`a[4]=3`，值是奇数，下标是偶数，不满足条件。

当 `i=5` 时，`a[5]=6`，值是偶数，下标是奇数，不满足条件。

所以数组部分结束后，`x=2+5=7`。

再看字符串部分。字符串 `s` 是 `"Csp2024"`。程序只处理数字字符。遇到数字字符时，执行 `y = y * 10 + (s[i] - '0')`，这相当于把当前数字接到 `y` 的末尾。

字符串中的数字字符依次为 `'2'`、`'0'`、`'2'`、`'4'`。

一开始 `y=0`。

读到 `'2'` 后，`y=2`。

读到 `'0'` 后，`y=20`。

读到 `'2'` 后，`y=202`。

读到 `'4'` 后，`y=2024`。

因此程序输出 `7 2024`，选 A。

这道题同时考数组下标、奇偶判断、字符串遍历和数字字符转换。数组部分不能只看元素值，还要比较下标奇偶；字符串部分不能把 `'2'` 当作整数 2 直接使用，要通过 `s[i] - '0'` 得到对应数字。

变式题 若把第一个循环中的条件改为 `a[i] % 2 != i % 2`，其他代码不变，程序输出为 ()。

A. `7 2024`

B. `14 2024`

C. `14 8`

D. `21 2024`

答案：B

解析：
修改后，数组部分累加的是“元素奇偶性和下标奇偶性不同”的位置。

这些位置是 `i=2,3,4,5`，对应值为 $1,4,3,6$。因此新的 `x=1+4+3+6=14`。

字符串部分没有改变，仍然得到 `y=2024`。所以输出 `14 2024`，选 B。

这个变式只改了判断条件中的 `==` 和 `!=`，累加范围就变成了原来的补集。做数组追踪题时，判断条件的方向必须看清楚。

十一、常见错误

数组与字符串题最常见的错误集中在下标和长度上。

第一，把长度当成最后下标。长度为 n 的数组或字符串，若从 0 开始使用，最后位置是 $n-1$。

第二，把第 k 个位置写成下标 `k`。从 0 开始时，第 k 个元素的下标是 $k-1$。

第三，混用从 0 开始和从 1 开始。输入、处理、输出必须使用同一套有效范围。

第四，看错循环边界。`i < n` 和 `i <= n` 只差一个等号，但访问范围不同。

第五，混淆字符和整数。`'5'` 是字符，若要得到整数 5，常用 `'5' - '0'`。

第六，字符串输入时忽略空格。`cin >> s` 只读入一个不含空白字符的单词，不会自动读入整行含空格内容。

第七，二维数组行列看反。`a[i][j]` 中哪个是行、哪个是列，要结合输入循环和题意判断。

这些错误大多不是知识难，而是读题时没有把下标写清楚。做数组和字符串题时，建议先写出有效范围，再追踪变量。

十二、本章要点

数组是一排同类型数据，常见下标从 0 到 $n-1$，但竞赛程序中也常从 1 到 n 使用。读程序时，要先判断有效数据范围，再看循环访问了哪些下标。

二维数组要分清行和列。字符串可以看成一排字符，`s.size()` 表示长度，最后一个字符位置是 `s.size() - 1`。`s[i]` 是字符，数字字符和整数之间需要用 `'0'` 转换。

数组与字符串题的关键是边界。看到循环、下标、长度、字符比较时，尽量把访问范围写在草稿上。只要下标不乱，很多程序阅读题和完善程序题都会变得清楚。

0x0A 指针、结构体与链表

指针、结构体与链表在 CSP-J 第一轮中不会考得很深，但经常以选择题或程序片段的形式出现。题目可能给出几行指针赋值，问变量值是否改变；也可能给出一个结构体结点，问怎样访问成员、怎样把新结点接到链表前面或后面。

这部分内容容易让学生紧张，是因为它不像数组那样直观。数组下标写成 `a[i]`，看起来像第几个位置；指针写成 `p`、`*p`、`&x`、`p->next`，符号更多。第一轮需要掌握的不是复杂写法，而是几个基本判断：指针保存的是地址，`*p` 表示指针指向的对象，结构体把多个成员合成一个整体，链表靠指针把结点连起来。

本章只讲纸面做题够用的内容，不展开复杂指针技巧。掌握本章后，遇到指针、结构体和链表题，至少能判断“改的是指针本身，还是指针指向的变量”。

一、指针保存地址

普通变量保存的是值。指针变量保存的是地址，也就是某个变量所在位置的标识。可以先把“地址”理解成变量的位置编号。

例如：

```cpp
#include <iostream>
using namespace std;

int main() {
    int x = 10;
    int *p = &x;

    cout << x << endl;
    cout << *p << endl;

    *p = 20;
    cout << x << endl;

    return 0;
}
```

这段程序中，`x` 是一个整数变量，保存值 10。`p` 是一个指向 `int` 的指针，`&x` 表示取出 `x` 的地址，所以 `p = &x` 后，`p` 指向 `x`。

`*p` 表示“`p` 指向的那个变量”。因为 `p` 指向 `x`，所以 `*p` 和 `x` 访问的是同一个整数。前两个输出都是 `10`。执行 `*p = 20` 后，被修改的是 `p` 指向的变量，也就是 `x`，所以最后输出 `20`。

指针题中，最重要的是分清这三个写法：

x 表示变量 x 的值。

&x 表示变量 x 的地址。

*p 表示指针 p 指向的那个对象。

如果题目给出 int *p = &x;，就可以在草稿上写一句：p 指向 x。后面只要出现 *p，就可以先替换成“p 指向的变量”。

二、改指针和改变量不是一回事

指针变量自己也有值，它的值是某个地址。给指针重新赋值，会改变它指向哪里；通过 *p 赋值，会改变它指向的变量。

这是第一轮中最常见的指针考法。

看下面的程序片段：

```
int x = 101;
int y = 201;
int *p = &x;
int *q = &y;
p = q;
```

执行到 int *p = &x; 时，p 指向 x。执行到 int *q = &y; 时，q 指向 y。最后一句 p = q; 的含义是把 q 中保存的地址赋给 p，于是 p 也指向 y。

这句不会把 x 改成 201，也不会把 y 改成 101。它只改变了 p 的指向。执行完后，p 和 q 都指向 y，x 仍是 101，y 仍是 201。

如果最后一句换成 *p = *q;，含义就不同了。此时 *p 是 x，*q 是 y，于是会把 y 的值赋给 x，执行后 x 变成 201，y 不变。

如果最后一句换成 *p = 300;，就是把 p 指向的变量改成 300。因为 p 指向 x，所以 x 变成 300。

做指针赋值题时，可以按下面三步判断：

第一，看每个指针当前指向谁。

第二，看赋值号左边是指针名，还是带 * 的表达式。

第三，如果左边是 p，改的是指针本身；如果左边是 *p，改的是 p 指向的变量。

这三步足够处理第一轮中大多数基础指针片段。

三、空指针只表示“不指向有效对象”

nullptr 表示空指针。它通常表示这个指针暂时不指向任何有效对象。

例如：

```
int *p = nullptr;
```

这表示 p 目前没有指向某个整数变量。此时不能把 *p 当成某个正常变量来读写。第一轮题目一般不会要求深入讨论异常情况，但会考“空指针不能正常解引用”的基本判断。

链表中常用空指针表示“后面没有结点”。例如一个结点的 next 为 nullptr，表示它是链表的最后一个结点。一个链表的头指针为 nullptr，表示这个链表当前为空。

做题时，看到 p == nullptr，应理解为“p 是否为空”。看到 p != nullptr，应理解为“p 是否指向某个结点或对象”。遍历链表时，经常用 while (p != nullptr) 判断还有没有结点可以处理。

注意：nullptr 不是整数 0 的普通替代品。第一轮通常只需要知道它表示空指针，不需要展开更多语法细节。

四、结构体把多个成员放在一起

结构体用于把多个相关数据合成一个整体。例如，一个学生可以有编号和成绩，一个点可以有横坐标和纵坐标，一个链表结点可以有数据和指向下一个结点的指针。

结构体的基本写法如下：

```
#include <iostream>
using namespace std;

struct Point {
    int x;
    int y;
};

int main() {
    Point p;
    p.x = 3;
    p.y = 5;

    cout << p.x + p.y << endl;
    return 0;
}
```

Point 是结构体类型，里面有两个成员：x 和 y。Point p; 定义了一个结构体变量 p。访问普通结构体变量的成员时，用点号 .。所以 p.x 表示 p 的 x 成员，p.y 表示 p 的 y 成员。

结构体题常考两个符号：

点号 . 用于结构体对象本身。

箭头 -> 用于指向结构体对象的指针。

若有 Point p;，访问成员写 p.x。

若有 Point *q = &p;，访问成员可以写 q->x。它表示通过指针 q 访问所指对象的 x 成员。也可以理解为 (*q).x，但写成 q->x 更常见。

要注意 *q.x 和 (*q).x 不是一回事。由于成员访问的优先级较高，*q.x 会先尝试访问 q.x，这不符合“q 是指针”的情况。若需要先解引用，再访问成员，必须写 (*q).x，或者直接写 q->x。

第一轮中，结构体不会只考定义格式，更常和指针、链表结合出现。只要分清 . 和 ->，很多题就能读懂。

五、结构体赋值会复制成员值

同类型结构体变量之间可以赋值。赋值时，会把各个成员的值复制过去。

例如：

```
Point a, b;
a.x = 1;
a.y = 2;
b = a;
b.x = 5;
```

执行 b = a 后，b.x 为 1，b.y 为 2。接着执行 b.x = 5，修改的是 b 的成员，不会自动修改 a.x。所以最后 a.x 仍为 1，b.x 为 5。

如果结构体成员中有指针，赋值会复制指针保存的地址，而不是复制指针指向的整条结构。这一点在链表中要小心。不过 CSP-J 第一轮通常只考基础判断：结构体变量赋值会让对应成员得到相同值；结构体指针赋值则只是改变指针指向。

例如，若 Node *p = head;，这不是复制一个新结点，而是让 p 指向 head 当前指向的结点。此后 p->data 和 head->data 访问的是同一个结点的数据。

读这类程序时，要分清“结构体对象赋值”和“结构体指针赋值”。前者复制成员值，后者复制地址。

六、链表由结点连成

链表是一种由结点连接起来的数据结构。每个结点通常保存两类信息：当前结点的数据，以及指向下一个结点的指针。

最基本的单链表结点可以写成：

```
struct Node {
    int data;
    Node *next;
};
```

这里 data 保存数据，next 指向下一个结点。如果当前结点后面没有结点，next 通常为 nullptr。

可以把链表想成一串结点：

第一个结点保存数据，并通过 next 找到第二个结点；

第二个结点保存数据，并通过 next 找到第三个结点；

最后一个结点的 next 为 nullptr，表示链表结束。

链表通常用一个头指针 head 保存第一个结点的位置。如果 head == nullptr，表示链表为空。如果 head != nullptr，就可以从 head 开始，沿着 next 一个一个往后走。

遍历链表时，常见写法是：

```
Node *p = head;

while (p != nullptr) { ...; p = p->next; }
```

其中 `p = p->next` 表示让 p 移到下一个结点。如果漏掉这句，p 一直指向同一个结点，循环就无法按正常方式结束。

第一轮中，链表题不要求写复杂操作，但会考基本连接关系。看到 next，要把它理解为“指向后继结点”。

七、在链表头部插入结点

链表选择题常考“把新结点插到链表头部”。假设已经有头指针 head，现在要创建一个新结点 p，并让它成为新的第一个结点。

常见步骤是：

先让新结点的数据成员保存值。

再让新结点的 next 指向原来的第一个结点。

最后让 head 指向新结点。

代码可以写成：

```
Node *p = new Node;
p->data = 42;
p->next = head;
head = p;
```

这几句的顺序很重要。`p->next = head;` 必须在 `head = p;` 之前执行。因为 head 原来保存的是旧链表的第一个结点位置，新结点要先接住这条旧链表；如果先写 `head = p;`，就会让 head 改指向新结点，原来第一个结点的位置就丢失了。

这类题的干扰项常见有几种。

第一，只写 `p->next = head;`，但没有写 `head = p;`。这样新结点虽然指向了旧链表，但头指针没有更新，新结点没有成为链表开头。

第二，先写 `head = p;`，再写 `p->next = head;`。这样会让新结点的 next 指向自己，旧链表没有被接上。

第三，把 `head->data = 42;` 写成给新结点赋值。这样修改的是旧头结点的数据，不是新结点的数据。

第四，漏写 `p->data = 42;`。链表连接可能对了，但新结点的数据没有按题意设置。

做头插题时，可以画三个对象：head、新结点 p、原来的第一个结点。先让 p->next 指向旧头，再让 head 指向 p。这个顺序不能反。

八、访问链表中的结点

链表不能像数组那样直接用下标访问第 k 个位置。它通常要从头指针开始，沿着 next 一步一步走。

例如，要输出链表中所有结点的 data，可以写：

```
Node *p = head;
```

然后不断访问 p->data，再执行 p = p->next。直到 p 变成 nullptr，说明已经走到链表末尾之后。

读链表遍历程序时，要看三个位置：

第一，p 初始指向哪里。通常是 head，表示从第一个结点开始。

第二，循环条件是什么。常见是 p != nullptr，表示只要还有结点就继续。

第三，循环中有没有移动 p。常见是 p = p->next，表示走向下一个结点。

如果循环条件写成 p->next != nullptr，含义就不同了。它表示当前结点后面还有结点时才进入循环，这样往往不会处理最后一个结点。第一轮题目可能会利用这一点考边界。

例如，若链表有三个结点，条件 p != nullptr 会依次处理第一个、第二个、第三个结点；条件 p->next != nullptr 只会在当前结点不是最后一个时进入，可能只处理前两个结点，最后一个结点被漏掉。具体是否漏掉，还要看循环体里先处理还是先移动。

链表题的边界通常是空链表、只有一个结点、最后一个结点。判断链表程序时，用这几个小情况代入，比只看一般情况更稳。

九、数组和链表的区别

数组和链表都能保存一串数据，但访问方式不同。

数组中的元素排在连续编号的位置上，适合用下标访问。例如 a[3] 可以直接表示第 4 个位置。第一轮中，数组题主要考下标范围、循环边界和元素更新。

链表中的结点靠指针连接。要找到后面的结点，通常要从某个已知结点开始，沿着 next 走。第一轮中，链表题主要考指针指向、结点连接和头指针更新。

选择题可能会问链表和数组的区别。可以把重点记成：

数组大小通常在定义时确定，按下标访问方便。

链表通过结点和指针连接，插入、删除时常需要改指针。

数组访问某个位置常依赖下标，链表访问后续结点常依赖 next。

本书不展开数组和链表的性能细节。CSP-J 第一轮中，能判断基本结构和基本操作即可。

十、原创例题：指针与链表头插

例题 10-1 阅读下面的程序片段。假设 head 原来指向一个非空链表的第一个结点，结点定义如下：

```
struct Node {
    int data;
    Node *next;
};

Node *p = new Node;
p->data = 7;
p->next = head;
head = p;
```

执行这段代码后，下列说法正确的是（）。

- **A.** 新结点 p 成为链表的第一个结点，原来的链表接在它后面。
- **B.** 原来的第一个结点被删除。
- **C.** head 没有变化，仍然指向原来的第一个结点。
- **D.** 新结点 p 的 next 指向它自己。

答案：A

解析：

先看 `Node *p = new Node;`，它创建了一个新结点，并让 p 指向这个新结点。

`p->data = 7;` 把新结点的数据设为 7。

`p->next = head;` 让新结点的 next 指向原来的第一个结点。此时新结点已经接到了旧链表前面，但头指针还没有更新。

`head = p;` 让头指针改为指向新结点。执行完后，链表的第一个结点就是 p 指向的新结点，它后面接着原来的链表。

因此 A 正确。B 错在原来的第一个结点没有被删除，只是变成了第二个结点。C 错在 head 已经改为指向新结点。D 错在 `p->next` 指向的是旧的 head，不是 p 自己。

变式题 若把最后两句交换，写成：

```
head = p;
p->next = head;
```

则执行后最可能出现的情况是（）。

- **A.** 新结点正确成为链表头，且旧链表接在后面。
- **B.** 新结点的 next 指向自己，旧链表没有被正确接上。
- **C.** head 仍然指向旧链表第一个结点。
- **D.** 程序一定无法通过编译。

答案：B

解析：

先执行 `head = p;` 后，head 已经指向新结点。接着执行 `p->next = head;`，此时 head 也是 p，所以 `p->next` 会指向 p 自己。旧链表没有被新结点接住。

这说明链表头插时，必须先保存旧头结点的位置，也就是先写 p->next = head;，再写 head = p;。顺序一旦反过来，指针关系就会改变。

十一、常见错误

指针、结构体和链表题常见错误有以下几类。

第一，把 p = q 理解成复制 q 指向的值。实际上它只是让 p 和 q 指向同一个位置。

第二，把 *p = *q 和 p = q 混淆。前者修改 p 指向的变量值，后者修改 p 的指向。

第三，忘记 &x 表示地址。若要让指针 p 指向变量 x，常见写法是 p = &x，不是 p = x。

第四，把 . 和 -> 混用。结构体对象访问成员用 .，结构体指针访问成员常用 ->。

第五，链表头插顺序写反。应先让新结点的 next 指向旧头，再更新 head。

第六，遍历链表时忘记移动指针。若没有 p = p->next，循环可能无法按预期结束。

第七，用 p->next != nullptr 代替 p != nullptr 时没检查边界。前者可能漏掉最后一个结点，也不能直接用于空链表而不做额外判断。

遇到这类题，不要凭符号多少判断难度。只要把每个指针当前指向谁写清楚，再按语句顺序更新，就能处理大多数基础题。

十二、本章要点

指针保存地址。&x 表示取 x 的地址，*p 表示访问 p 指向的对象。p = q 改的是指针本身，让 p 指向 q 指向的位置；*p = *q 改的是 p 指向的变量值。

结构体把多个成员放在一起。普通结构体变量访问成员用 .，结构体指针访问成员常用 ->。结构体对象赋值会复制成员值，结构体指针赋值只是复制地址。

链表由结点和 next 指针连接。头插时，先让新结点接住旧头，再更新头指针。遍历链表时，从 head 出发，沿着 next 移动，直到遇到 nullptr。第一轮中，链表题的关键不是写复杂操作，而是判断指针关系有没有接对。

第三编 数学与计算机常识

0x0B 进制与存储单位

进制与存储单位属于 CSP-J 第一轮中稳定出现的基础知识。它们通常不会单独考很长的推导，但常常出现在选择题中：给出一个二进制、八进制或十六进制数，要求转换成十进制；给出若干不同进制的数，要求计算结果；给出 KB、MB、byte、bit，要求判断容量大小。

这类题看起来简单，却很容易丢分。常见错误有三种：把进制标记看漏，把字节和位混淆，把 1024 当成 1000。本章不展开复杂编码，只把第一轮最常用的转换、计算和检查方法讲清楚。

一、进制的含义

十进制是平时最常用的计数方式。一个十进制数每一位的权值是 $1, 10, 100, 1000, \ldots$。例如 345 表示 $3 \times 100 + 4 \times 10 + 5$。

二进制、八进制、十六进制的区别只在于“每一位逢几进一”。

二进制每一位只能是 0 或 1，权值是 $1, 2, 4, 8, 16, \ldots$。

八进制每一位可以是 0 到 7，权值是 $1, 8, 64, 512, \ldots$。

十六进制每一位可以是 0 到 9 以及 A, B, C, D, E, F，其中 A 到 F 分别表示 10 到 15，权值是 $1, 16, 256, 4096, \ldots$。

教材和试题中常用下标表示进制，例如 1010_2 表示二进制数 1010，14_8 表示八进制数 14，D_{16} 表示十六进制数 D。做题时，先看下标，再计算。没有下标时，要根据题面说明判断，不要直接按十进制处理。

C++ 程序中也有进制写法。整数常量 `10` 是十进制，`010` 是八进制，`0x10` 是十六进制。第一轮如果考 C++ 字面量，要特别注意前导 `0` 和前缀 `0x`。

二、转成十进制

进制转换最稳的方法，是按权值展开。对整数部分，从右向左依次乘以 1、进制、进制平方。

例如 101101_2：

$101101_2 = 1 \times 32 + 0 \times 16 + 1 \times 8 + 1 \times 4 + 0 \times 2 + 1 = 45$。

例如 32_8：

$32_8 = 3 \times 8 + 2 = 26$。

例如 $2F_{16}$：

$2F_{16} = 2 \times 16 + 15 = 47$。

这一步不要跳得太快。很多题给出的数并不长，按权值写两三项就能算清楚。比起心算，写出权值更能避免把 D_{16} 当成十进制 D，或者把 1010_2 当成十进制 1010。

如果有小数部分，小数点右边的权值依次是进制的 -1 次方、-2 次方、-3 次方。比如 101.11_2：

整数部分 $101_2 = 5$。小数部分 $.11_2 = 1 \times \frac{1}{2} + 1 \times \frac{1}{4} = 0.75$。所以 $101.11_2 = 5.75$。

再如 32.1_8，整数部分是 26，小数部分是 $1 \times \frac{1}{8} = 0.125$，所以结果是 26.125。

第一轮中的进制小数一般不复杂，关键是不要把小数点右侧也按正整数权值处理。

三、从十进制转到其他进制

把十进制整数转成其他进制，常用“除进制取余数”的方法。每次除以目标进制，记录余数，最后把余数倒过来读。

例如把十进制 26 转成二进制：

$26 \div 2 = 13$ 余 0；

$13 \div 2 = 6$ 余 1；

$6 \div 2 = 3$ 余 0；

$3 \div 2 = 1$ 余 1；

$1 \div 2 = 0$ 余 1。

余数从下往上读，得到 11010_2。

转八进制、十六进制也是同样方法，只是除数分别变成 8 和 16。十六进制中，余数 10 到 15 要写成 A 到 F。

在选择题中，如果选项已经给出，可以不必完整转换每一个选项。可以先把题目中的数转成十进制，再把选项逐个估算或转换。若只差最后一步，选择题选项也能帮助检查结果。

四、二、八、十六进制之间的快速转换

二进制、八进制、十六进制之间有更快的方法，因为 $8 = 2^3$，$16 = 2^4$。

二进制转八进制时，从右向左每 3 位分成一组，不足的高位补 0。每组二进制对应一个八进制数字。

例如 101101_2 分成 101 和 101，每组都是十进制 5，所以 $101101_2 = 55_8$。

二进制转十六进制时，从右向左每 4 位分成一组。

例如 101101_2 可以补成 0010 1101_2。$0010_2 = 2$，$1101_2 = 13$，十六进制中 13 写作 D，所以 $101101_2 = 2D_{16}$。

反过来，八进制转二进制时，每个八进制数字写成 3 位二进制；十六进制转二进制时，每个十六进制数字写成 4 位二进制。

例如 $3A_{16}$ 中，3 写成 0011，A 写成 1010，所以 $3A_{16} = 111010_2$。最前面的多余 0 可以省略。

这个方法适合二、八、十六进制之间互转，不适合直接处理十进制。遇到十进制时，仍然要按权值展开或除进制取余。

五、混合进制计算

第一轮常见考法是把不同进制的数放在同一个算式里。处理这类题时，不要在原进制下直接加减乘除。最稳的方法是先全部转成十进制，再计算。

真题例题 计算 $(14_8 - 1010_2) \times D_{16} - 1101_2$ 的结果，并选择答案的十进制值。

A. 13
B. 14
C. 15
D. 16

答案：A

解析：
先把每个数转成十进制。$14_8 = 1\times8+4 = 12$。$1010_2 = 8+2 = 10$。$D_{16} = 13$。$1101_2 = 8+4+1 = 13$。

原式变为 $(12 - 10) \times 13 - 13$，也就是 $2 \times 13 - 13 = 13$。因此答案是 A。

这道题的陷阱不在计算本身，而在进制混合。若把 14_8 当成十进制 14，第一步就会错；若忘记 $D_{16} = 13$，也会得到错误结果。

再看一个示例：计算 $101010_2 + 166_8$。

$101010_2 = 32 + 8 + 2 = 42$，$166_8 = 1 \times 64 + 6 \times 8 + 6 = 118$，两者之和是 160。若要写成十六进制，$160 \div 16 = 10$ 余 0，所以结果是 $A0_{16}$。

混合进制题的草稿建议写成三行：第一行列出每个数对应的十进制值，第二行做普通十进制计算，第三行按题目要求转回目标进制或选择答案。

六、存储单位

存储单位题常考 bit、byte、KB、MB、GB 的换算。

bit 表示二进制位，一个 bit 只能表示 0 或 1。

byte 表示字节，1byte = 8bit。中文题目中常把 byte 写作 “字节”。

常见换算是：

1KB = 1024byte；

1MB = 1024KB；

1GB = 1024MB。

如果题目明确说明 1KB = 1024 字节，就按 1024 计算。CSP-J 第一轮中，存储单位题通常采用这个换算方式，不要把 1KB 直接当成 1000 字节。

示例：1MB 是多少 bit?

先换成字节：1MB = 1024 × 1024byte = 1048576byte。再换成 bit：1048576 × 8 = 8388608bit。

这里最容易漏掉最后的乘 8。如果选项中同时有 1048576 和 8388608，前者是字节数，后者才是位数。

做单位题时，要先判断题目问的是 byte 还是 bit。B 和 b 在英文中也可能分别表示 byte 和 bit，但中文题目一般会写清 “字节” 或 “二进制位”。不要只看数字大小。

七、与二进制有关的常见判断

有些选择题虽然不直接要求进制转换，但和二进制有关。

第一，n 个二进制位最多能表示 2^n 种不同状态。例如 4 位二进制共有 16 种编码，从 0000_2 到 1111_2。

第二，若只表示非负整数，n 位二进制能表示的范围是 0 到 $2^n - 1$。例如 8 位无符号二进制能表示 0 到 255。

第三，格雷码是一种相邻编码只改变一位的编码方式。第一轮如果出现格雷码选择题，通常可以按相邻项检查：从前一个编码到后一个编码，是否只有一位不同。不要把它当作普通二进制从小到大直接判断。

第四，二进制位数和数量级有关。$2^{10} = 1024$，所以 $2^{20} = 1048576$。这也是 KB、MB 换算中经常出现 1024 的原因。遇到二分查找次数、存储容量或编码数量时，2^{10} 是一个很常用的检查点。

这些内容不需要背很复杂的公式。第一轮更常考的是 “能否用二进制位数判断数量”，以及 “能否看出单位或编码规则中的陷阱”。

八、变式题

变式题 计算 $2F_{16} + 1011_2$ 的十进制值。

A. 56
B. 57
C. 58
D. 59

答案：C

解析：
$2F_{16} = 2 \times 16 + 15 = 47$。$1011_2 = 8 + 2 + 1 = 11$。两者相加得到 58，所以选 C。

这道题与前面的真题例题类似，都需要先统一转成十进制。差别在于这里没有减法和乘法，主要考十六进制中字母 F 的含义。若把 F 当成 14，就会得到 57，这正好对应干扰项 B。

九、本章要点

进制题先看下标。1010_2、14_8、D_{16} 不能按十进制直接读。不同进制混合计算时，先统一转成十进制，再按题目要求计算或转换。

二进制、八进制、十六进制之间可以利用分组快速转换：3 位二进制对应 1 位八进制，4 位二进制对应 1 位十六进制。涉及小数时，小数点右侧的权值是进制的负次方。

存储单位题要分清 bit 和 byte。1byte = 8bit，1KB = 1024byte，1MB = 1024KB。如果题目问 bit，最后通常还要乘 8。

做这类题不要省略草稿。进制和单位本身不难，但只要看漏一个下标、一个字母或一个单位，答案就会偏离。把每个中间量写清楚，是进制与存储单位题最稳的得分方法。

0x0C 计数基础

计数题问的是“有多少种方案”。在 CSP-J 第一轮中，它常出现在选择题里：选人、排队、分组、相邻、不相邻、至少有一个、每类至少一个。题目通常不长，但很容易因为少算一类、多算一次而选错。

这一章先把最基本的工具讲清楚：阶乘、排列、组合、分类、分步、捆绑法、插空法。后面遇到更复杂的递推计数或动态规划计数时，再在对应章节继续处理。本章只解决第一轮中最常见的纸面计数问题。

一、阶乘：连续往下乘

排列和组合都会用到阶乘。阶乘记作 $n!$，意思是从 n 一直乘到 1。

例如：

$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$。

$4! = 4 \times 3 \times 2 \times 1 = 24$。

$3! = 3 \times 2 \times 1 = 6$。

特别地，$1! = 1$，$0! = 1$。在 CSP-J 第一轮中，通常只需要计算较小的阶乘，不需要处理很大的数。

阶乘最常见的含义是“把 n 个互不相同的对象排成一排”。例如 5 个人站成一排，第一个位置有 5 种选法，第二个位置剩 4 种，第三个位置剩 3 种，第四个位置剩 2 种，最后一个位置剩 1 种，所以共有 5! 种站法。

这就是全排列。

二、排列：顺序不同，方案不同

如果题目关心顺序，就用排列。

从 n 个不同对象中选出 k 个，并且排成有顺序的一列，方案数记作 $A(n,k)$，也常写作 $P(n,k)$。本书统一写作 $A(n,k)$。

计算方法是：

$A(n,k) = n \times (n-1) \times (n-2) \times \cdots \times (n-k+1)$。

也就是从 n 开始，连续乘 k 个数。

例如，从 5 名同学中选 3 人，分别担任队长、副队长、记录员。三个职务不同，所以同样三个人换了职务，也算不同方案。队长有 5 种选法，副队长还剩 4 种，记录员还剩 3 种，共 $5 \times 4 \times 3 = 60$

种。也就是 $A(5,3)=60$。

如果 $k=n$，就是把 n 个对象全部排好，方案数是 $A(n,n)=n!$。

做排列题时，可以先问一句：同样几个对象换个顺序，算不算新方案？如果算，就要考虑排列。

例如：

从 8 人中选出 3 人参加活动，不分职务，是组合，不是排列。

从 8 人中选出 3 人分别担任一号、二号、三号选手，是排列，因为位置不同。

“排队”“站成一排”“第几名”“第几个位置”“分别担任不同职务”这些说法，通常都表示顺序重要。

三、组合：只看选谁，不看顺序

如果题目只关心选出了哪些对象，不关心顺序，就用组合。

从 n 个不同对象中选出 k 个，不考虑顺序，方案数记作 $C(n,k)$。

组合可以由排列得到。先按排列算，得到 $A(n,k)$。但同一组选出的 k 个对象，在排列中被按不同顺序重复算了 $k!$ 次。所以组合数是：

$C(n,k)=A(n,k)\div k!$。

例如，从 5 名同学中选 3 人组成小组，不分职务。若按排列算，有 $5\times4\times3=60$ 种。但同一个三人小组内部可以排成 $3!=6$ 种顺序，而这些顺序都对应同一个小组。所以实际方案数是 $60\div6=10$，即 $C(5,3)=10$。

常用计算方法是：

$$C(n,k)=\frac{n\times(n-1)\times\cdots\times(n-k+1)}{k\times(k-1)\times\cdots\times1}。$$

例如：

$$C(8,3)=\frac{8\times7\times6}{3\times2\times1}=56。$$

$$C(10,2)=\frac{10\times9}{2\times1}=45。$$

$C(6,4)$ 也可以算成 $C(6,2)$，因为从 6 个里选 4 个，等价于决定哪 2 个不选。所以 $C(6,4)=C(6,2)=15$。

这个性质在第一轮中很有用：$C(n,k)=C(n,n-k)$。如果 k 较大，可以改算 $n-k$，数字更小。

做组合题时，也先问一句：同样几个对象换个顺序，算不算新方案？如果不算，就用组合。

四、分类用加法，分步用乘法

计数题不只靠公式，还要先看清方案结构。

如果所有方案可以分成几类，并且每个方案只属于其中一类，那么总数用加法。

例如，从 5 个男生和 3 个女生中选 1 人。选男生有 5 种，选女生有 3 种，二者不会重叠，所以共有 $5+3=8$ 种。

如果一个完整方案要分几步完成，那么总数用乘法。

例如，从 5 个男生中选 1 人，再从 3 个女生中选 1 人，组成一男一女小组。男生有 5 种选法，女生有 3 种选法，每种男生选择都能搭配每种女生选择，所以共有 $5\times3=15$ 种。

分类和分步经常同时出现。

例如，从 A、B、C 三个部门中选 4 人，要求每个部门至少 1 人。因为一共只有 4 人，而有 3 个部门，所以人数分配只能是 $2,1,1$。接着要分类讨论：哪个部门出 2 人。每一类内部再用乘法计算各部门选人的方案数。

这类题不要一上来写公式。先把“方案长什么样”说清楚，再决定加法还是乘法。

五、相邻问题：捆绑法

如果题目要求几个对象必须相邻，常用捆绑法。

捆绑法的做法是：把必须相邻的对象先看成一个整体，和其他对象一起排列；最后再排列这个整体内部的对象。

例如，5 个男生和 3 个女生站成一排，要求 3 个女生必须相邻。

第一步，把 3 个女生捆成一个“女生块”。这样外部一共有 6 个对象：5 个男生和 1 个女生块。它们排成一排，有 $6!$ 种。

第二步，女生块内部的 3 个女生还可以自己排列，有 $3!$ 种。

所以总数是 $6!\times3!=720\times6=4320$。

捆绑法最容易漏掉“块内部排列”。如果只是把女生块和男生排列，得到的是女生作为整体的位置，还没有决定这三个女生在块内的先后顺序。

再看一个示例：4 个不同字母 A、B、C、D 排成一排，要求 A、B 相邻。

把 A、B 捆成一个块。外部对象是 AB 块、C、D，一共 3 个对象，有 $3!$ 种。AB 块内部有 AB、BA 两种顺序，所以总数是 $3!\times2=12$。

捆绑法适合“必须挨在一起”。如果题目说“不能相邻”，通常不能捆绑，要用插空法。

六、不相邻问题：插空法

如果题目要求某些对象互不相邻，常用插空法。

插空法的做法是：先排没有限制或数量较多的一类对象，排好后在它们之间和两端形成若干空位，再把受限制的对象放进这些空位。为了保证不相邻，每个空位最多放一个受限制对象。

例如，5 个男生和 3 个女生站成一排，要求任意两个女生不相邻。

第一步，先排 5 个男生。男生互不相同，有 $5!$ 种排法。

第二步，男生排好后，会形成 6 个空位：

第一个男生前面有一个空位；

相邻两个男生之间有 4 个空位；

最后一个男生后面有一个空位。

可以记成：空 男 空 男 空 男 空 男 空 男 空。

一共有 6 个空位。

第三步，把 3 个女生放入这 6 个空位中，并且每个空位最多放一个女生，这样女生之间就不会相邻。先选出 3 个空位，有 $C(6,3)$ 种；再把 3 个女生排列到这些空位中，有 3! 种。

所以总数是 $5! \times C(6,3) \times 3!$。

计算得到 $120 \times 20 \times 6 = 14400$。

插空法的核心有两点。

第一，先排不受“不相邻”限制的对象。常见的是先排男生，再插女生；先排普通物品，再插特殊物品。

第二，空位数要数对。m 个对象排成一排，会形成 $m+1$ 个空位，包括两端。如果只允许插在中间，那就是 $m-1$ 个空位。

再看一个较小示例：4 个不同字母 A、B、C、D 排成一排，要求 A、B 不相邻。

可以先排 C、D，有 2! 种。排好后有 3 个空位：空 C 空 D 空。把 A、B 放入其中 2 个空位，并排列 A、B，有 $C(3,2) \times 2!$ 种。总数是 $2! \times C(3,2) \times 2! = 12$。

也可以用补集检查：4 个字母全排列有 $4! = 24$ 种。A、B 相邻时，用捆绑法有 $3! \times 2 = 12$ 种。因此 A、B 不相邻有 $24 - 12 = 12$ 种。两种方法结果一致。

第一轮中，如果只限制两个对象不相邻，补集法往往更短；如果限制一类对象两两不相邻，插空法通常更清楚。

七、至少与至多

“至少”和“至多”是计数题中的高频条件。

“至少 1 个”可以直接分类，也可以用补集。补集是用全部方案数减去不满足条件的方案数。

例如，一个班级有 10 个男生和 12 个女生，选出 3 人的小组，要求至少有 1 个女生。

全部方案数是 $C(22,3)$。不满足条件的情况是没有女生，也就是 3 人全是男生，有 $C(10,3)$ 种。所以答案是 $C(22,3) - C(10,3) = 1540 - 120 = 1420$。

如果直接分类，也可以分成 1 个女生、2 个女生、3 个女生：

$C(12,1)C(10,2) + C(12,2)C(10,1) + C(12,3)$。

这个结果也等于 1420，但计算更长。

所以遇到“至少 1 个”，先想补集：全部减去一个都没有。

“每类至少一个”则常常要按人数分配分类。例如三个部门选 4 人，每个部门至少 1 人，人数只能是 $2,1,1$。这时补集也能做，但容易多减、少加；直接按人数分配更稳。

“至多 2 个”可以直接数 0 个、1 个、2 个，也可以用全部减去 3 个及以上。选哪种方法，取决于哪边分类更少。

不要把“至少”机械地当成补集，也不要把“至多”机械地一项一项加。第一轮最重要的是少写、稳算、不漏项。

八、分组问题：注意组内和组间重复

分组题常见错误是重复计算。

例如 6 个人，两个人组一队，总共组成三队，不区分队伍编号。问有多少种不同的组队方式？

可以这样想：先把 6 个人排成一排，有 $6!$ 种。把第 $1,2$ 个看成第一队，第 $3,4$ 个看成第二队，第 $5,6$ 个看成第三队。

但这样多算了。

每一队内部两个人交换顺序，队伍没有变，所以每队多算了 2 倍，三队共多算了 2^3 倍。

三队之间交换顺序，也不是新的分组，所以又多算了 $3!$ 倍。

因此答案是 $6! \div (2^3 \times 3!) = 720 \div 48 = 15$。

分组题不要急着套公式，要先问两件事：

组内有没有顺序？如果没有，组内排列要除掉。

组与组之间有没有编号？如果没有，组间排列也要除掉。

若题目说“分成甲队、乙队、丙队”，队伍有名字，组间顺序就不能除掉；若题目说“组成三队，不区分队伍编号”，组间顺序就要除掉。

九、真题例题

真题例题 某公司有 10 名员工，分为 3 个部门：A 部门有 4 名员工、B 部门有 3 名员工、C 部门有 3 名员工。现需要从这 10 名员工中选出 4 名组成一个工作小组，且每个部门至少要有 1 人。问有多少种选择方式？

A. 120
B. 126
C. 132
D. 238

答案：B

解析：
这道题不关心 4 个人的顺序，只关心选了哪些人，所以用组合。

一共选 4 人，来自 3 个部门，且每个部门至少 1 人。因此人数分配只能是 $2,1,1$。也就是说，恰好有一个部门出 2 人，另外两个部门各出 1 人。

按“哪个部门出 2 人”分类。

A 部门出 2 人：从 A 的 4 人中选 2 人，有 $C(4,2)=6$ 种；B 部门选 1 人，有 $C(3,1)=3$ 种；C 部门选 1 人，有 $C(3,1)=3$ 种。这一类有 $6\times 3\times 3=54$ 种。

B 部门出 2 人：A 部门选 1 人，有 $C(4,1)=4$ 种；B 部门选 2 人，有 $C(3,2)=3$ 种；C 部门选 1 人，有 $C(3,1)=3$ 种。这一类有 $4\times 3\times 3=36$ 种。

C 部门出 2 人：A 部门选 1 人，有 $C(4,1)=4$ 种；B 部门选 1 人，有 $C(3,1)=3$ 种；C 部门选 2 人，有 $C(3,2)=3$ 种。这一类有 $4\times 3\times 3=36$ 种。

三类互不重叠，总数是 $54+36+36=126$，所以选 B。

这道题的关键不是计算 $C(4,2)$，而是先判断人数分配。只要看出 4 人分给 3 个部门且每部门至少 1 人，分配只能是 $2,1,1$，后面的计算就很直接。

十、变式题

变式题 有 5 个男生和 3 个女生站成一排，要求任意两个女生不相邻。共有多少种站法？

A. 4320
B. 7200
C. 14400
D. 20160

答案：C

解析：
这是“不相邻”问题，用插空法。

先排 5 个男生，有 $5!=120$ 种排法。男生排好后有 6 个空位，包括最左端、最右端和男生之间的 4 个空位。

为了让女生不相邻，3 个女生必须放在不同空位中。先从 6 个空位中选 3 个，有 $C(6,3)=20$ 种；再把 3 个女生排进去，有 $3!=6$ 种。

总数是 $5!\times C(6,3)\times 3!=120\times 20\times 6=14400$，所以选 C。

如果把女生先排好，再插男生，会让限制变得不清楚。插空法通常先排数量较多且不受“不相邻”限制的一类，再把受限制的一类插入空位。

十一、本章要点

排列和组合的区别，是顺序是否重要。顺序重要，用 $A(n,k)$；顺序不重要，用 $C(n,k)$。全排列是 $n!$。

$A(n,k)$ 是从 n 开始连续乘 k 个数。$C(n,k)$ 是先算 $A(n,k)$，再除以 $k!$，因为同一组选出的 k 个对象在排列中被重复算了 $k!$ 次。

分类用加法，分步用乘法。一个计数题中经常既有分类，也有分步。分类时要保证几类互不重叠；分步时要保证每一步共同组成一个完整方案。

必须相邻，用捆绑法：先把相邻对象看成整体，再乘上块内部排列。互不相邻，用插空法：先排其他对象，数出空位，再把受限制对象放进不同空位。

“至少 1 个” 常用补集：全部减去一个都没有。“每类至少一个” 常按人数分配分类。分组题要检查组内顺序和组间顺序是否被重复计算。

计数题不是背一个公式就结束。每次做题都要先判断：顺序重不重要，方案分几类，每类分几步，有没有相邻限制，有没有重复计算。把这些问题问清楚，公式才会用得稳。

0x0D 简单数论

简单数论在 CSP-J 第一轮中经常以两种形式出现：一种是选择题，直接考整除、质数、最大公约数、最小公倍数、因数个数等概念；另一种是阅读程序或完善程序，代码中出现 `%`、质数判断、枚举因数、辗转相除、平方数判断等过程。

这一章不讲复杂数论。对 CSP-J 第一轮来说，更重要的是能看懂这些程序在做什么，能手算小数据，能判断循环边界是否正确，能发现“少算一个因数”“平方根算重了”“把 1 当成质数”等错误。

一、整除、因数和倍数

如果整数 a 除以整数 b 的余数为 0，就说 b 整除 a，也可以说 b 是 a 的因数，a 是 b 的倍数。在 C++ 中，常用 `a % b == 0` 判断 b 是否整除 a。

例如，$12\%3=0$，所以 3 是 12 的因数，12 是 3 的倍数。$12\%5=2$，所以 5 不是 12 的因数。

一个正整数的因数通常成对出现。若 i 是 n 的因数，那么 n/i 也是 n 的因数。例如 36 的因数可以按配对来看：

1 和 36；

2 和 18；

3 和 12；

4 和 9；

6 和 6。

其中 6 比较特殊，因为 $6\times6=36$。当 n 是完全平方数时，平方根这一对其实是同一个数，只能算一次。

这个“成对出现”的性质非常重要。它解释了为什么很多程序只枚举到 $\sqrt{n}$，也解释了为什么判断平方数、统计因数时要特别处理 `i * i == n` 的情况。

例如要枚举 n 的所有因数，可以从 1 枚举到满足 $i\times i\le n$ 的位置。若 `n % i == 0`，就得到一对因数 `i` 和 `n / i`。如果 `i * i == n`，说明这两个因数相同，只记录一次。

```cpp
for (int i = 1; i * i <= n; i++) {
    if (n % i == 0) {
        cout << i << " ";
        if (i * i != n) cout << n / i << " ";
    }
}
```

这段代码能找到所有因数，但输出顺序不一定从小到大。若题目要求从小到大输出，通常要先保存较小因数，再反向输出较大因数。第一轮完善程序中经常考这种“先存一半，再输出另一半”的结构。

二、奇偶性与余数

奇数和偶数是最基础的数论判断。一个整数能被 2 整除，就是偶数；不能被 2 整除，就是奇数。C++ 中常写作 `x % 2 == 0` 或 `x % 2 == 1`。

做纸面题时，奇偶性常用来判断表达式结果。

偶数加偶数仍是偶数，奇数加奇数也是偶数，奇数加偶数是奇数。乘法中，只要有一个因数是偶数，乘积就是偶数；只有所有因数都是奇数，乘积才是奇数。

余数不仅能判断奇偶，也能判断周期。例如一个数除以 3 的余数只能是 $0, 1, 2$。如果程序中有 `i % 3`，它通常在按三种情况循环。

例如：

```
for (int i = 1; i <= 10; i++) {
    cout << i % 3 << " ";
}
```

输出是 `1 2 0 1 2 0 1 2 0 1`。这说明余数会周期性重复。阅读程序时，只要看到 `%`，就要马上问：它在判断整除，还是在制造周期？

注意，C++ 中负数取余在第一轮中不常作为主要考点。若题目没有特别说明，通常输入会避开复杂负数取余。若真的出现负数，应按 C++ 的运算规则判断，不要直接套小学里的“余数总是非负”的习惯。

三、质数和合数

质数也叫素数。一个大于 1 的正整数，如果只有 1 和它本身两个正因数，就是质数。一个大于 1 的正整数，如果除了 1 和它本身之外还有其他正因数，就是合数。

$2, 3, 5, 7, 11, 13$ 是质数。$4, 6, 8, 9, 10, 12$ 是合数。

1 不是质数，也不是合数。0 也不是质数。很多程序判断质数时，第一句就是：

```
if (n <= 1) return false;
```

这是为了排除 0、1 和负数。若少了这句，`1` 可能被误判为质数。

判断一个数 n 是否为质数，不需要从 2 一直试到 $n-1$。只要检查到 $\sqrt{n}$ 即可。原因来自因数成对出现：如果 n 有一个大于 $\sqrt{n}$ 的因数，那么它一定还有一个小于 $\sqrt{n}$ 的配对因数。只要较小的那个已经被检查过，就不需要再检查较大的那个。

常见写法如下：

```cpp
bool isPrime(int n) {
    if (n <= 1) return false;
    for (int i = 2; i * i <= n; i++) {
        if (n % i == 0) return false;
    }
    return true;
}
```

这里最重要的是循环条件 `i * i <= n`。如果写成 `i * i < n`，对于完全平方数可能出错。例如 $49 = 7 \times 7$，必须检查到 $i = 7$ 才能发现它不是质数。若只检查 `i * i < n`，当 $i = 7$ 时条件不成立，49 会被误判为质数。

如果把循环改成 `i <= n`，也不是简单地“更稳”。因为当 $i = n$ 时，`n % i == 0` 一定成立，这会把所有大于 1 的数都判成合数。若要枚举到更远，也应该是 `i < n`，但这样循环次数更多，不适合纸面阅读。第一轮中更常见、更合适的是 `i * i <= n`。

四、真题片段：统计质数

下面是一段 CSP-J 第一轮中的程序片段，用来判断质数、统计质数个数和求质数和。

```cpp
bool isPrime(int n) {
    if (n <= 1) {
        return false;
    }
    for (int i = 2; i * i <= n; i++) {
        if (n % i == 0) {
            return false;
        }
    }
    return true;
}

int countPrimes(int n) {
    int count = 0;
    for (int i = 2; i <= n; i++) {
        if (isPrime(i)) {
            count++;
        }
    }
    return count;
}

int sumPrimes(int n) {
    int sum = 0;
    for (int i = 2; i <= n; i++) {
        if (isPrime(i)) {
            sum += i;
        }
    }
    return sum;
}
```

读这段程序时，不要先算答案，而要先分清三个函数的作用。

`isPrime(n)` 判断 n 是否为质数，返回 `true` 或 `false`。

`countPrimes(n)` 从 2 到 n 逐个检查，统计质数个数。

`sumPrimes(n)` 从 2 到 n 逐个检查，把所有质数加起来。

例如输入 10 时，2 到 10 中的质数是 $2, 3, 5, 7$，共有 4 个，和为 17。因此输出中对应的两个数是 4 和 17。

若题目问 50 以内质数和，可以先列出质数：

$2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47$。

它们的和是 328。

这类题不能只盯着主函数输出。阅读程序时，应先给每个函数写一句“它做什么”。函数作用清楚后，手算就会少很多混乱。

这段程序还常被改动来考判断能力。例如把 `i * i <= n` 改成 `i <= n`，就会把质数本身也作为因数检查到，导致质数被判为合数。若把它改成 `i <= n / 2`，虽然比 `i * i <= n` 做了更多检查，但不会把质数本身检查进去，判断结果仍然正确，只是循环次数增加。读题时要看清具体改的是哪一句。

五、最大公约数

两个正整数公共的因数叫公因数，其中最大的一个叫最大公约数，常记作 $\gcd(a, b)$。

例如 12 的正因数有 $1, 2, 3, 4, 6, 12$，18 的正因数有 $1, 2, 3, 6, 9, 18$。公共因数是 $1, 2, 3, 6$，最大公约数是 6，所以 $\gcd(12, 18) = 6$。

最大公约数常用辗转相除法计算。它的核心规则是：$\gcd(a, b) = \gcd(b, a \bmod b)$。不断把较大的问题变成较小的问题，直到第二个数变成 0。这时第一个数就是最大公约数。

例如求 $\gcd(48, 18)$：

$48 \% 18 = 12$，所以转成 $\gcd(18, 12)$；

$18 \% 12 = 6$，所以转成 $\gcd(12, 6)$；

$12 \% 6 = 0$，所以转成 $\gcd(6, 0)$；

结果是 6。

代码常写成：

```
int gcd(int a, int b) {
    while (b != 0) {
        int r = a % b;
        a = b;
        b = r;
    }
    return a;
```

```
}

int lcm(int a, int b) {
    return a / gcd(a, b) * b;
}
```

阅读这段程序时，要看懂三个变量的变化：`r` 保存余数，`a` 变成原来的 `b`，`b` 变成余数。只要 `b` 没有变成 0，就继续做。

第一轮中常给出小数值要求手算。手算时按上面的链条写即可，不需要展开所有因数。

还要注意一个常见错误：`a % b` 中的 `b` 不能为 0。所以循环条件必须是 `b != 0`，而不是进入循环后再不加判断地取余。

六、最小公倍数

两个正整数公共的倍数叫公倍数，其中最小的正公倍数叫最小公倍数，常记作 $\text{lcm}(a,b)$。

例如 4 的倍数有 $4,8,12,16,20,\ldots$，6 的倍数有 $6,12,18,24,\ldots$，最小公共倍数是 12，所以 $\text{lcm}(4,6)=12$。

最大公约数和最小公倍数之间有一个常用关系：

$a\times b=\gcd(a,b)\times\text{lcm}(a,b)$。

因此：

$\text{lcm}(a,b)=a\div\gcd(a,b)\times b$。

例如 $a=12,b=18$，$\gcd(12,18)=6$，所以最小公倍数是 $12\div 6\times 18=36$。

写程序时通常写 `a / gcd(a, b) * b`，而不是 `a * b / gcd(a, b)`。在较大数据中，先乘可能使中间结果过大。CSP-J 第一轮一般不会把这个点考得很深，但阅读程序时要知道这两种写法的数学含义相同。

最大公约数偏向“共同能分成多大的块”，最小公倍数偏向“多久再次同时出现”。如果题目中出现“同时”“周期重合”“第一次一起出现”，常常和最小公倍数有关。

七、因数个数与平方数

枚举因数时，最常见的边界是 `i * i <= n`。它不仅用于判断质数，也用于统计因数、找平方数。

如果 i 是 n 的因数，那么通常会得到两个因数：i 和 n/i。当 `i * i == n` 时，这两个因数相同，只能算一次。

例如 12 的因数配对是 $1,12$，$2,6$，$3,4$。一共 6 个。

例如 36 的因数配对是 $1,36$，$2,18$，$3,12$，$4,9$，$6,6$。前四对各贡献两个因数，最后一对只贡献一个因数，所以一共 9 个。

因此，统计因数个数可以这样想：

遇到 `n % i == 0` 时，若 `i * i == n`，答案加 1；否则答案加 2。

平方数判断也是同一件事。若存在整数 i 使得 `i * i == n`，则 n 是完全平方数。比如 49 是完全平方数，因为 $7 \times 7 = 49$；50 不是，因为没有整数平方等于 50。

在完善程序中，如果题目要求判断平方数，空缺位置常出现在三类地方：初始值从多少开始，循环边界到哪里，判断条件是不是 `i * i == num`。这三处都和因数配对有关。

八、同余和周期

如果两个整数除以同一个正整数 m 的余数相同，就说它们对 m 同余。CSP-J 第一轮不一定直接使用“同余”这个词，但程序中经常使用 `%` 来表达同余关系。

例如判断一个数是否能被 3 整除，就是判断 `x % 3 == 0`。判断两个数是否在同一类中，可以写 `a % m == b % m`。

同余最常见的应用是周期。

例如星期问题中，每过 7 天星期重复。若今天是星期一，那么 10 天后相当于过了 $10\%7 = 3$ 天，是星期四。

程序中数组循环也经常用取余。例如 `i = (i + 1) % n` 表示从当前位置走到下一个位置，若超过最后一个位置，就回到 0。这在约瑟夫问题、循环队列和环形模拟中经常出现。

阅读这类程序时，要把 `% n` 看成“回到范围内”。如果下标范围是 0 到 $n-1$，那么 `(i + 1) % n` 的结果一定仍在这个范围内。

但如果下标范围是 1 到 n，写法就不同，常见处理是先加再判断，或使用 `(i % n) + 1`。不要把 0 开始的循环下标和 1 开始的编号混在一起。

九、例题

例题 输入一个正整数 n，判断下面程序输出的值。若输入为 36，程序输出什么？

```
int n, ans = 0;
cin >> n;
for (int i = 1; i * i <= n; i++) {
    if (n % i == 0) {
        if (i * i == n) ans++;
        else ans += 2;
    }
}
cout << ans << endl;
```

答案：
输出 9。

解析：
这段程序统计 n 的正因数个数。循环从 1 枚举到 $\sqrt{n}$，每找到一个因数 i，就同时考虑它的配对因数 n/i。

当 $n = 36$ 时：

$i = 1$，得到 1 和 36，答案加 2；

$i = 2$，得到 2 和 18，答案加 2；

$i = 3$，得到 3 和 12，答案加 2；

$i = 4$，得到 4 和 9，答案加 2；

$i = 5$，不是因数；

$i = 6$，得到 6 和 6，因为 `i * i == n`，只能加 1。

总数是 $2 + 2 + 2 + 2 + 1 = 9$。所以程序输出 `9`。

这道题的易错点是把 6 算两次。只要看到 `i * i == n`，就要想到完全平方数的中间因数只能算一次。

十、变式题

变式题 若把上面程序中的循环条件改成 `i * i < n`，输入仍为 36，输出会变成多少？

A. `8`
B. `9`
C. `10`
D. `12`

答案：A

解析：
原程序使用 `i * i <= n`，会枚举到 $i = 6$。修改为 `i * i < n` 后，当 $i = 6$ 时，$6 \times 6 < 36$ 不成立，所以循环不会处理 $i = 6$。

这样程序只会统计前四组因数：$1, 36$，$2, 18$，$3, 12$，$4, 9$。每组贡献 2 个因数，总数为 8。

因此输出变成 `8`。这说明在处理平方数、质数判断、因数枚举时，边界上的等号常常决定答案是否正确。

十一、本章要点

整除用余数判断。`n % i == 0` 表示 i 是 n 的因数。因数通常成对出现：若 i 是 n 的因数，n/i 也是 n 的因数。完全平方数的平方根只算一次。

质数是大于 1 且只有 1 和自身两个正因数的整数。0 和 1 都不是质数。判断质数时常枚举到 `i * i <= n`，因为若存在较大的因数，就一定存在一个较小的配对因数。

最大公约数用辗转相除法，核心变化是 $\gcd(a, b) = \gcd(b, a \bmod b)$。最小公倍数可由最大公约数得到，常用写法是 `a / gcd(a, b) * b`。

取余不仅用于判断整除，也用于表示周期。看到 `%` 时，要判断它是在检查因数、判断奇偶，还是让编号循环回到某个范围内。

简单数论题的主要得分点不在公式多，而在边界清楚。`n <= 1`、`i * i <= n`、`i * i == n`、`b != 0` 这些判断看起来很短，却经常决定程序是否正确。读程序时要把它们单独圈出来检查。

0x0E 复杂度

复杂度用来描述程序规模变大时，运行时间和占用空间怎样增长。CSP-J 第一轮不会要求严格证明复杂度，但经常要求判断一段程序的时间复杂度，或者判断某个算法在给定规模下是否合理。

这一章要解决三个问题：怎么看循环复杂度，怎么看递归复杂度，怎么看空间复杂度。循环部分是第一轮最常见的考法；递归部分除了会看一条调用链，还要能初步使用主定理判断分治递归；空间部分则重点看数组、二维数组和递归调用带来的额外存储。

复杂度题不是精确计算程序运行了多少秒，而是判断增长级别。只要能稳定区分 $O(1)$、$O(\log n)$、$O(n)$、$O(n\log n)$、$O(n^2)$、$O(2^n)$ 等常见形式，第一轮中的大多数复杂度题就不会失分。

一、复杂度看增长速度

假设一个程序处理 n 个数据。若 n 变成原来的 2 倍，程序运行时间大约也变成 2 倍，这类程序常写作 $O(n)$。若 n 变成原来的 2 倍，运行时间大约变成 4 倍，这类程序常写作 $O(n^2)$。

$O(\cdot)$ 记号关心的是增长速度，不关心具体机器上运行多少秒，也不关心常数倍。比如一个程序执行 $3n+10$ 次基本操作，复杂度写作 $O(n)$；一个程序执行 $5n^2+7n+100$ 次基本操作，复杂度写作 $O(n^2)$。

当 n 变大时，最高阶项决定增长速度。$5n^2+7n+100$ 中，n^2 项增长最快，所以写成 $O(n^2)$。常数 5、低阶项 $7n$ 和常数项 100 通常省略。

复杂度不是程序运行时间本身。两个同为 $O(n)$ 的程序，实际速度可能相差几倍；但从第一轮判断角度看，它们都属于线性增长。考试更关心学生能否看出增长级别，而不是精确数出每次赋值语句。

二、常见复杂度顺序

CSP-J 第一轮常见复杂度从小到大大致如下。

$O(1)$ 是常数复杂度。无论 n 多大，操作次数都差不多。例如访问数组中的一个元素 `a[i]`，或者交换两个变量。

$O(\log n)$ 是对数复杂度。每次把范围缩小一半，常见于二分查找，也常见于循环变量不断乘 2 或除以 2 的程序。

$O(n)$ 是线性复杂度。通常是一层从 1 到 n 的循环。

$O(n\log n)$ 常见于一些排序算法，也常见于“做 n 次，每次需要 $\log n$”的结构。它比 $O(n)$ 大，比 $O(n^2)$ 小。

$O(n^2)$ 是平方复杂度。常见于两层循环，枚举所有二元组。

$O(n^3)$ 是立方复杂度。常见于三层循环，枚举所有三元组。

$O(2^n)$ 是指数复杂度。常见于每个元素选或不选的搜索。

$O(n!)$ 是阶乘复杂度。常见于枚举所有排列。

第一轮中，常用判断是：$O(n^2)$ 通常远大于 $O(n \log n)$；$O(2^n)$ 和 $O(n!)$ 增长很快，只能处理较小的 n。但不要只背顺序，还是要回到程序结构：循环执行多少次，范围缩小多少次，递归分成多少个子问题，数组开了多大。

三、顺序结构：相加后取最大

如果程序先做一段 $O(n)$ 的循环，再做一段 $O(n^2)$ 的循环，总复杂度写作 $O(n^2)$。

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

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

第一段循环执行 n 次，是 $O(n)$。第二段双重循环执行 $n \times n$ 次，是 $O(n^2)$。两段顺序执行，总次数大约是 $n + n^2$，最终写作 $O(n^2)$。

这就是“顺序结构相加，保留增长最快的一项”。

如果有两个互不嵌套的循环，一个是 $O(n)$，一个是 $O(m)$，并且 n 和 m 是两个不同规模，那么总复杂度可以写作 $O(n + m)$。例如先读入 n 个点，再读入 m 条边，复杂度常写作 $O(n + m)$。不要把所有字母都强行合并成同一个 n。

四、嵌套循环：相乘，但要看边界

如果一层循环里面还有一层循环，并且内层每次都执行差不多完整的范围，通常用乘法。

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

外层执行 n 次。每次外层循环中，内层执行 m 次。所以总次数是 nm，复杂度是 $O(nm)$。若 $m = n$，就是 $O(n^2)$。

但不能只数“有两层循环”就写 $O(n^2)$。要看内层循环的边界。

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

这段代码中，内层循环次数不是每次 n，而是第 1 次执行 1 次，第 2 次执行 2 次，直到第 n 次执行 n 次。总次数是 $1+2+\cdots+n=\frac{n(n+1)}{2}$，复杂度仍然是 $O(n^2)$。

如果内层循环只执行固定次数，就不会多出一个 n。

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

总次数是 $3n$，复杂度是 $O(n)$。这里也有两层循环，但内层次数是常数。

判断嵌套循环时，要问三句话：外层多少次？内层每次多少次？内层次数是否随输入规模增长？

五、折半循环：对数复杂度

如果每次循环都把问题规模缩小一半，通常是 $O(\log n)$。

最常见的是二分查找。设有序数组长度为 n，每比较一次，就可以丢掉大约一半范围。最多比较次数是 $\log_2 n$ 级别，所以时间复杂度是 $O(\log n)$。

还有一种常见写法是循环变量不断乘 2：

```
int x = 1;
while (x <= n) {
    x *= 2;
}
```

x 的变化是 $1, 2, 4, 8, 16, \ldots$。执行 k 次后，x 大约变成 2^k。当 $2^k > n$ 时循环停止，所以 k 大约是 $\log_2 n$，复杂度是 $O(\log n)$。

如果循环变量不断除以 2，也是类似的：

```
while (n > 1) {
    n /= 2;
}
```

n 每次减半，执行次数也是 $O(\log n)$。

复杂度中的对数底数通常省略。$\log_2 n$、$\log_3 n$、$\log_{10} n$ 之间只差常数倍，所以在复杂度中统一写作 $O(\log n)$。

看到“范围减半”“每次乘 2”“每次除以 2”，就要想到对数复杂度。

六、分支结构：通常取较大一支

如果程序中有 `if` 和 `else`，一次运行只会进入其中一支。判断复杂度时，通常看最坏情况下较大的那一支。

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

第一支是 $O(n)$，第二支是 $O(n^2)$。若题目问最坏时间复杂度，应写 $O(n^2)$。

如果题目明确说明某个分支一定不会执行，就按实际会执行的分支判断。但多数第一轮题目默认问一般或最坏情况。除非题面有明确输入条件，不要随意假设程序只走较短的一支。

七、递归先看调用次数

递归复杂度不能只看函数写了几行，要看它产生了多少次函数调用。

如果一个递归函数每次只调用自己一次，并且参数每次减少 1，通常是 $O(n)$。

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

这段程序从 `f(n)` 调到 `f(n - 1)`，再调到 `f(n - 2)`，直到 `f(0)` 停止，调用链长度大约是 n，所以是 $O(n)$。

如果每次只调用自己一次，但参数每次减半，通常是 $O(\log n)$。

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

这里参数变化是 $n, n/2, n/4, n/8, \ldots$，所以调用次数是 $O(\log n)$。

如果一个递归函数每次调用自己两次，并且每次都让参数减少 1，调用数量会增长很快，常见为指数级。

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

`h(n)` 会产生两个 `h(n - 1)`，每个 `h(n - 1)` 又产生两个 `h(n - 2)`，调用树不断分叉。这类结构通常不是 $O(n)$，而是 $O(2^n)$ 级别。

如果递归中使用数组记录已经算过的结果，避免重复计算，复杂度可能大幅降低。这属于动态规划的思想，后续章节会继续讲。本章先记住：递归复杂度要看“每层有多少个调用”和“递归有多少层”。

八、主定理：分治递归的快速判断

有一类递归不是每次减 1，而是把规模分成若干个相同大小的子问题。这类递归常出现在分治算法中，例如归并排序。

主定理用于判断下面这种递归式：

$T(n) = aT(n/b) + f(n)$。

这里 a 表示每次分成几个子问题，n/b 表示每个子问题的规模，$f(n)$ 表示当前这一层除了递归调用之外还要做多少工作。

使用主定理时，先做三步：

第一步，找出 a、b 和 $f(n)$。

第二步，计算或识别 $n^{\log_b a}$。

第三步，比较 $f(n)$ 和 $n^{\log_b a}$ 哪个更大，或者是否同级。

为了适合第一轮，本书只保留最常见的三种判断。

第一种，若 $f(n)$ 比 $n^{\log_b a}$ 小很多，那么递归子问题占主导，答案是 $O(n^{\log_b a})$。

例如 $T(n) = 2T(n/2) + 1$。这里 $a = 2$，$b = 2$，所以 $n^{\log_b a} = n^{\log_2 2} = n$。当前层额外工作是 1，比 n 小，所以 $T(n) = O(n)$。

第二种，若 $f(n)$ 和 $n^{\log_b a}$ 同级，那么每一层工作量差不多，总复杂度多一个 $\log n$，答案是 $O(n^{\log_b a} \log n)$。

例如 $T(n) = 2T(n/2) + n$。这里 $a = 2$，$b = 2$，$n^{\log_b a} = n$，而 $f(n) = n$，二者同级，所以 $T(n) = O(n \log n)$。

归并排序就是这种情况：分成两个规模为 $n/2$ 的子问题，合并时需要 $O(n)$ 的工作，所以复杂度是 $O(n \log n)$。

第三种，若 $f(n)$ 比 $n^{\log_b a}$ 大很多，并且工作量主要集中在当前层，那么答案通常是 $O(f(n))$。

例如 $T(n) = 2T(n/2) + n^2$。这里 $n^{\log_2 2} = n$，而 $f(n) = n^2$ 更大，所以复杂度是 $O(n^2)$。

这三种情况可以先不用证明。第一轮使用时，重点是会认形式、会代入、会比较。

再看几个常见例子。

$T(n)=T(n/2)+1$。这里 $a=1, b=2, n^{\log_2 1}=n^0=1, f(n)=1$，二者同级，所以 $T(n)=O(\log n)$。这对应二分查找。

$T(n)=T(n/2)+n$。这里 $a=1$，$b=2$，$n^{\log_2 1}=1$，$f(n)=n$ 更大，所以 $T(n)=O(n)$。直观理解是：第一层做 n，下一层做 $n/2$，再下一层做 $n/4$，总和仍是 $O(n)$。

$T(n)=4T(n/2)+n$。这里 $a=4$，$b=2$，$n^{\log_2 4}=n^2$，$f(n)=n$ 更小，所以 $T(n)=O(n^2)$。

$T(n)=4T(n/2)+n^2$。这里 $n^{\log_2 4}=n^2$，$f(n)=n^2$，二者同级，所以 $T(n)=O(n^2\log n)$。

主定理不是所有递归都能直接套。比如 $T(n)=T(n-1)+1$ 不是把规模除以一个固定数，不能用这个形式。再如两个子问题规模不同，像 $T(n)=T(n/2)+T(n/3)+n$，也不属于本章这个轻量版主定理。遇到不能套的情况，就回到递归树或直接数调用次数。

九、空间复杂度

空间复杂度描述程序额外占用的存储空间怎样随规模增长。

若程序只使用几个普通变量，例如 `int x, y, ans;`，空间复杂度是 $O(1)$。

若程序开了一个长度为 n 的数组，例如 `int a[n]` 或等价规模的数组，空间复杂度是 $O(n)$。

若程序开了一个 $n\times n$ 的二维数组，空间复杂度是 $O(n^2)$。

递归程序还要注意调用深度。若递归深度是 n，除了显式数组之外，还会有 $O(n)$ 级别的调用记录。若递归深度是 $\log n$，这部分额外空间通常是 $O(\log n)$。在 CSP-J 第一轮中，一般只需按递归层数粗略判断。

空间复杂度题一般比时间复杂度题少，但阅读程序和完善程序中常会结合数组大小一起考。

十、真题片段：三层循环怎样数

下面这个函数来自 CSP-J 第一轮阅读程序中的典型结构。这里不讨论它具体解决什么问题，只看 `g(n, m)` 的时间复杂度。

```
int g(int n, int m) {
    for (int i = 1; i <= n; i++)
        h[i][1] = i;
    for (int j = 1; j <= m; j++)
        h[0][j] = 0;

    for (int i = 1; i <= n; i++) {
        for (int j = 2; j <= m; j++) {
            h[i][j] = INF;
            for (int k = 1; k <= i; k++)
                h[i][j] = min(h[i][j],
                              max(h[i - k][j], h[k - 1][j - 1]) + 1);
```

```
        }
    }

    return h[n][m];
}
```

前两个循环分别是 $O(n)$ 和 $O(m)$。

关键在后面的三层循环。外层 `i` 从 1 到 n。中层 `j` 从 2 到 m，数量是 $m-1$，复杂度中可看作 $O(m)$。内层 `k` 从 1 到 `i`，次数随 `i` 变化。

总次数可以这样看：

当 `i=1` 时，内层大约执行 m 次；

当 `i=2` 时，内层大约执行 $2m$ 次；

一直到 `i=n` 时，内层大约执行 nm 次。

所以总次数大约是 $m(1+2+\cdots+n)$，也就是 $O(mn^2)$。通常写作 $O(n^2m)$。

这段程序很容易被误判为 $O(nm)$，因为有人只看到了 `i` 和 `j` 两层主要循环，忽略了最里面的 `k`。也有人直接看到三层循环就写 $O(n^3)$，但这里三个变量不是都到 n，中间一层到 m，所以更准确是 $O(n^2m)$。

判断复杂度时，不要只数缩进层数。每一层循环的范围都要看清楚。

十一、例题

例题 判断下面程序片段的时间复杂度。

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

A. $O(n)$
B. $O(n\log n)$
C. $O(n^2)$
D. $O(2^n)$

答案：C

解析：
外层 `i` 从 1 到 n。当 `i=1` 时，内层执行 n 次；当 `i=2` 时，内层执行 $n-1$ 次；当 `i=n` 时，内层执行 1 次。

总次数是 $n+(n-1)+\cdots+1=\frac{n(n+1)}{2}$，最高阶是 n^2，所以复杂度是 $O(n^2)$。

这道题的关键是内层不是每次都从 1 到 n，但总量仍然是平方级。若只看到内层范围在变，就误以为不是 $O(n^2)$，也会选错。

十二、变式题

变式题 判断递归式 $T(n) = 2T(n/2) + n$ 对应的时间复杂度。

A. $O(n)$
B. $O(\log n)$
C. $O(n \log n)$
D. $O(n^2)$

答案：C

解析：
这个递归式符合主定理形式 $T(n) = aT(n/b) + f(n)$。

其中 $a = 2$，$b = 2$，$f(n) = n$。先计算 $n^{\log_b a} = n^{\log_2 2} = n$。

现在 $f(n)$ 和 $n^{\log_b a}$ 同级，所以总复杂度多一个 $\log n$，为 $O(n \log n)$。

这类式子常对应“把问题分成两个一半大小的子问题，再用线性时间合并”的分治算法。归并排序就是典型例子。

十三、本章要点

复杂度描述的是规模变大时程序运行时间或空间的增长速度，不是具体运行秒数。$O(\cdot)$ 记号会省略常数倍和低阶项，只保留主要增长级别。

顺序执行的几段代码，复杂度相加后保留增长最快的一项。嵌套循环通常相乘，但要看每层循环的实际范围。内层若只执行常数次，就不会多出一个 n。

每次把范围缩小一半，通常是 $O(\log n)$。一层从 1 到 n 的循环通常是 $O(n)$。两层都和 n 有关的循环常见为 $O(n^2)$。三层循环要看每层变量范围，不能只按层数机械判断。

递归复杂度要看调用次数和递归层数。单链递归常见为 $O(n)$ 或 $O(\log n)$；不断分叉的递归可能达到指数级。分治递归若符合 $T(n) = aT(n/b) + f(n)$，可以用主定理轻量判断：比较 $f(n)$ 与 $n^{\log_b a}$，小则看子问题，同级则多乘 $\log n$，大则通常看 $f(n)$。

空间复杂度看额外存储空间随规模怎样增长。几个变量是 $O(1)$，一维数组常是 $O(n)$，二维数组常是 $O(n^2)$，递归还要考虑调用深度。

做复杂度题时，最稳的方法是写出“哪一层循环执行多少次”或“递归每层有多少工作”。只要把执行次数看清楚，再去掉常数和低阶项，答案通常就不会偏离。

0x0F 计算机常识

计算机常识在 CSP-J 第一轮中通常出现在单项选择题里，题目短、分值不高，但覆盖面比较杂。它可能考硬件和软件的区别，考操作系统和应用软件的区别，考编译器的作用，也可能考计算机奖项、信息表示、网络安全等基本概念。

这类题不适合展开成完整的计算机组成原理或操作系统课程。本章只整理第一轮中常见、容易混淆、适合纸面判断的内容。遇到这类题时，先判断它属于哪一类常识，再用最基本的定义排除错误选项。

计算机又常被称为电脑，本质上是一种能够快速、自动地处理信息的电子设备。“数字化”是指计算机处理的信息需要用编码形式表示；“电子”是指现代计算机的工作基础主要依赖电子逻辑部件。对于第一轮来说，不需要深入了解电路原理，只要知道计算机能够按照程序对数据进行输入、存储、处理和输出即可。

计算机的几个主要特点可以这样记：运算速度快，计算精度高，具有逻辑判断能力，存储容量大，能在程序控制下自动工作。这些特点解释了为什么计算机能用于天气预报、工程设计、企业管理、网络通信、图像处理和程序评测等场景。

一、硬件与软件

硬件是看得见、摸得着的计算机设备或部件。常见硬件包括 CPU、内存、硬盘、显示器、键盘、鼠标、打印机、网卡等。

软件是运行在计算机上的程序及相关数据。常见软件包括操作系统、编译器、浏览器、文本编辑器、办公软件、游戏程序等。

判断硬件和软件时，可以抓住一句话：硬件是设备，软件是程序。键盘、鼠标、显示器是硬件；Windows、Linux、浏览器、记事本是软件。

输入设备用于把信息输入计算机，例如键盘、鼠标、扫描仪、摄像头、传感器。输出设备用于把计算机处理结果呈现出来，例如显示器、打印机、音箱。某些设备既可能输入也可能输出，例如触摸屏既能显示信息，也能接收触摸操作。

CPU 通常被看作计算机的核心运算部件，负责执行指令。内存用于临时存放正在运行的程序和数据，断电后其中内容通常会丢失。硬盘、U 盘等外部存储设备用于长期保存数据，断电后数据仍可保留。

还要分清内存和外存。内存直接服务于正在运行的程序，速度较快，但断电后内容通常不保留；外存用于长期保存文件，例如硬盘、U 盘、光盘等。题目如果问“长期保存数据”，通常要想到外存；如果问“程序运行时临时存放数据”，通常要想到内存。

做选择题时，不要只凭名字判断。比如“音箱”不是输入设备，而是输出设备；“传感器”不是输出设备，而是输入设备；“操作系统”不是硬件，而是软件。

二、操作系统与应用软件

操作系统是管理计算机硬件资源、为其他程序提供运行环境的软件。常见操作系统有 Windows、Linux、macOS、Android、iOS 等。

操作系统负责的事情很多，例如管理文件、管理内存、管理输入输出设备、调度程序运行、提供用户界面。学生不需要在第一轮里展开这些细节，但要知道：操作系统不是普通应用软件，它位于硬件和应用程序之间，负责让各种程序能够使用计算机资源。

应用软件是直接面向用户完成具体任务的软件，例如浏览器、文本编辑器、画图软件、办公软件、聊天软件、播放器等。

例如 Notepad 是文本编辑器，不是操作系统；Linux、Windows、macOS 是操作系统。遇到“下面哪个不是操作系统”这类题时，就按这个区别判断。

还要注意，Android 和 iOS 也是操作系统，只是常用于移动设备。不要把“操作系统”只理解为电脑上的 Windows。

系统软件和应用软件也要区分。操作系统、编译器、数据库管理系统、驱动程序等通常属于系统软件或与系统运行密切相关的软件；浏览器、播放器、文字处理软件、画图软件等通常属于应用软件。第一轮选择题不要求划分得特别细，但会考“哪个不是同一类”的基本判断。

三、编译器、解释器和程序运行

C++ 程序不能直接被计算机按源代码文本执行。通常需要先经过编译，把源代码转换成计算机能够执行的目标代码或可执行程序。完成这件事的软件叫编译器。

编译器的主要作用，是把源代码转换为机器能够执行的低层代码，而不是直接管理文件，也不是负责编辑源代码。编辑源代码通常由编辑器或集成开发环境完成。

例如，学生在写 C++ 时，常见流程是：

先写出 `.cpp` 源文件；

再用编译器编译；

若没有编译错误，得到可执行程序；

最后运行程序，输入数据，查看输出。

如果程序语法错误，通常会在编译阶段报错。例如少写分号、括号不匹配、变量未声明等。若程序能编译但运行结果错误，就属于运行过程或逻辑过程中的问题。

解释器和编译器不同。解释器通常一边读取源程序，一边执行对应操作。第一轮中，通常只需知道 C++ 常按编译方式运行，Python 常按解释方式运行。不要把“编译器”误认为“调试器”或“操作系统”。

调试器用于帮助检查程序运行过程，例如观察变量值、设置断点。它不是编译器。IDE 是集成开发环境，可能同时包含编辑、编译、调试等功能，但它本身不是某一个单独概念。

选择题有时会把“编辑器、编译器、解释器、调试器、操作系统”放在同一组选项里。判断时先看动词：写代码对应编辑器，翻译源代码对应编译器，逐步执行源程序对应解释器，检查运行状态对应调试器，管理资源对应操作系统。

四、程序、算法和数据

程序是按照一定规则写成的指令序列，用来让计算机完成任务。算法是解决问题的方法和步骤。数据是程序处理的对象。

同一个算法可以用不同语言写成程序。例如排序可以用 C++ 写，也可以用 Python 写。反过来，同一种语言也可以实现许多不同算法。

CSP-J 第一轮更关心学生能否分清“算法思想”和“程序实现”。比如“二分查找”是一种算法，“用 C++ 写出的二分代码”是程序实现。若题目问某段代码的时间复杂度，考的是算法或实现的效率；若题目问 `int` 范围、循环语句、函数返回值，考的是程序语言规则。

数据在计算机中最终都要以二进制形式存储和处理。文本、图片、声音、视频看起来形式不同，但在计算机内部都需要编码成二进制数据。前面进制章节已经讲过二进制与存储单位，本章只强调这个基本认识：计算机处理的底层信息最终离不开 0 和 1。

位和字节也常出现在常识题里。位通常写作 bit，是二进制信息的最小单位；字节通常写作 Byte，1 个字节等于 8 位。若题目问“最小的信息单位”，通常选 bit；若题目问“文件大小常用单位”，通常会出现 Byte、KB、MB 等。

五、文件、目录和扩展名

文件用于保存数据或程序。目录也叫文件夹，用来组织文件。

扩展名通常能提示文件类型。例如 `.cpp` 常表示 C++ 源代码文件，`.txt` 常表示纯文本文件，`.pdf` 常表示文档文件，`.jpg`、`.png` 常表示图片文件。但扩展名只是文件名的一部分，不能完全保证文件内容一定正确。

路径用于描述文件在目录中的位置。不同系统路径写法可能不同，但第一轮一般不会考复杂路径规则。只需要知道：文件可以放在目录中，目录中还可以有子目录。

选择题有时会问“源程序”“目标程序”“可执行程序”的区别。源程序是程序员写的代码文本，例如 `.cpp` 文件。目标程序或可执行程序是经过编译、链接后生成的程序文件，可以被操作系统加载运行。

不要把“文件大小”和“文件内容”混淆。一个文本文件、图片文件、可执行文件都可以占用存储空间，但它们保存的数据含义不同。

文件管理也属于操作系统的重要功能。创建、删除、复制、移动文件，查看文件所在目录，设置文件权限，这些都属于操作系统或文件管理程序提供的能力。应用软件可以读写文件，但应用软件本身通常不负责管理整个计算机的文件系统。

六、网络与安全常识

网络用于连接不同计算机，使它们能够交换数据。局域网通常范围较小，例如学校机房、家庭网络；互联网范围更大，连接了大量网络和设备。

浏览网页、发送邮件、在线提交程序、下载文件，都离不开网络通信。网址用于定位网络上的资源，IP 地址用于标识网络中的设备或服务位置。第一轮一般不会要求深入掌握网络协议，只会考非常基础的概念判断。

“网络化”也是计算机发展的重要方向之一。它强调把不同地点的计算机连接起来，实现信息交换和资源共享。在线评测系统、网上报名、云端文件同步、远程协作，都属于网络应用的具体表现。

信息安全常识也可能出现在选择题中。常见原则包括：不要随意打开未知来源的文件或链接；不要把密码告诉别人；不要在不可信网站输入重要账号；重要数据要备份；发现账号异常要及时修改密码。

病毒、木马等恶意程序可能破坏数据、窃取信息或影响系统运行。防范这类风险，常见做法是使用可信来源的软件，及时更新系统和安全软件，不随意运行陌生程序。

这部分题通常不考技术细节，而考基本判断。若某个选项鼓励泄露密码、关闭所有安全防护、随意执行陌生文件，通常就是错误做法。

七、计算机领域常识

计算机发展和人物奖项偶尔也会出现在第一轮选择题中。

图灵奖是计算机领域中非常重要的奖项。若题目问“哪个奖项与计算机领域最相关”，图灵奖通常是正确方向。奥斯卡奖与电影相关，普利策奖与新闻、文学等相关，诺贝尔奖虽然影响很大，但没有专门的计算机科学奖项。

冯·诺依曼与现代计算机体系结构关系密切。通常所说的冯·诺依曼体系，核心思想包括程序和数据都存放在存储器中，计算机按照程序指令自动执行。第一轮不会要求展开体系结构细节，但“冯·诺依曼体系”这个名字需要认识。

阿兰·图灵是计算机科学发展史上的重要人物。图灵奖以他的名字命名，用来奖励对计算机科学和计算技术发展作出重要贡献的研究者。姚期智是 2000 年图灵奖获得者，在计算理论、密码学等方向有重要贡献。若题目出现人物和奖项的对应关系，要注意把人物、领域、奖项分清。

还要分清计算机科学中的一些基本术语。人工智能是研究让机器表现出某些智能行为的领域；数据库用于组织和管理大量数据；程序设计语言用于描述程序；算法是解决问题的步骤；数据结构是组织数据的方式。

这些概念不需要背很长定义。第一轮选择题往往只要求排除明显不属于同一类的选项。

八、发展历史、分类和应用

计算机发展史偶尔会以常识题形式出现。早期教材常把 ENIAC 作为早期通用电子计算机的代表，它诞生于 20 世纪 40 年代，使用大量电子管，体积大、耗电多，但在当时具有很强的计算能力。对

于第一轮来说，记住 “ENIAC、电子计算机、电子管、早期代表” 这一组对应关系即可。

计算机通常按主要逻辑元件划分为几代。第一代是电子管计算机，体积大、速度较低，主要使用机器语言，常用于科学计算。第二代是晶体管计算机，体积和功耗降低，可靠性提高，汇编语言开始广泛使用。第三代是集成电路计算机，高级语言和操作系统逐渐发展。第四代是大规模和超大规模集成电路计算机，微型计算机、个人计算机和计算机网络得到广泛发展。

这类历史题不要死背大量年份。更稳定的判断方法是看 “元件” 和 “时代” 的对应：电子管对应早期，晶体管在电子管之后，集成电路在晶体管之后，大规模集成电路对应现代微型计算机的发展。若题目只问 “微型计算机属于哪一代”，通常按第四代理解。

计算机的发展趋势常概括为高性能化、微型化、网络化、智能化和多媒体化。高性能化强调更强的计算能力和更大的存储能力；微型化强调体积更小、使用更方便；网络化强调互联与资源共享；智能化强调模拟人的部分感知、判断和推理能力；多媒体化强调综合处理文字、图像、声音、视频等信息。

计算机也可以按不同标准分类。按工作原理，可以分为数字计算机、模拟计算机和混合计算机；按用途，可以分为通用计算机和专用计算机；按规模，可以分为巨型机、大型机、小型机、微型机等；按一次处理的二进制位数，旧资料中会出现 8 位机、16 位机、32 位机等说法。CSP-J 第一轮通常不会考很细，只需能判断 “分类标准不同，分出的类别也不同”。

计算机的主要应用领域可以分为几类。科学计算是较早的应用领域，常用于弹道计算、天气预报、工程计算等。信息处理是应用最广泛的领域，常用于企业管理、资料管理、办公事务等。过程控制强调实时采集信息并控制设备运行，例如工业自动化控制。计算机辅助系统常见缩写有 CAD、CAM、CAI、CAT 等，分别可对应计算机辅助设计、计算机辅助制造、计算机辅助教学、计算机辅助测试等语境。办公自动化常写作 OA，管理信息系统常写作 MIS，决策支持系统常写作 DSS。

这些缩写不需要全部展开成专业课程，只要能在选择题中识别常见对应关系。若题目给出 “CAD——计算机辅助设计” 这样的选项，一般是正确搭配；若把 CAD 说成 “计算机辅助下载” 之类，就可以排除。

我国计算机发展也可能作为常识背景出现。旧资料中常见的关键词包括我国 1958 年研制出第一台电子数字计算机，“银河” 巨型计算机，“神威” 计算机，以及 “龙芯” 通用 CPU。对于第一轮来说，这些内容通常只会作为低频常识出现，不需要背成长时间表。

九、真题片段

下面是一类常见选择题：

下列哪一个不是操作系统名字？

- **A.** Notepad
- **B.** Linux
- **C.** Windows
- **D.** macOS

答案是 A。

Linux、Windows、macOS 都是操作系统。Notepad 是文本编辑器，用于编辑文本，不负责管理计算机硬件资源，也不是其他程序运行的基础环境。

这道题的关键不是背很多软件名字，而是判断“它是管理计算机资源的系统软件，还是完成具体任务的应用软件”。操作系统是平台，文本编辑器是应用。按这个思路，即使题目把 Notepad 换成浏览器、播放器、画图软件，也能判断它们不是操作系统。

再看另一个常见判断：

编译器的主要作用是什么？

A. 直接执行源代码
B. 将源代码转换为机器代码
C. 进行代码调试
D. 管理程序运行时的内存

答案是 B。

编译器的核心作用是转换程序，把高级语言源代码转为机器能够执行或进一步处理的低层形式。调试程序是调试器的任务；管理内存、文件和设备主要属于操作系统职责；直接执行源代码不是 C++ 编译器的主要描述。

再看一组不需要计算的常识判断：

以下奖项与计算机领域最相关的是哪一个？

A. 奥斯卡奖
B. 图灵奖
C. 诺贝尔奖
D. 普利策奖

答案是 B。

判断这类题时，不要被奖项名气影响。奥斯卡奖主要与电影相关，普利策奖主要与新闻、文学等相关，诺贝尔奖没有专门的计算机科学奖项。图灵奖与计算机领域关系最密切。

还有一种题会问信息表示：

目前主流的计算机存储数据最终都要转换成哪类数据？

A. 二进制
B. 十进制
C. 八进制
D. 十六进制

答案是 A。

十进制是人们日常最常用的计数方式，八进制和十六进制常用来简洁表示二进制数据，但计算机底层存储和处理信息离不开二进制。因此这类题选二进制。

十、易错辨析

“操作系统”和“应用软件”容易混淆。Windows、Linux、macOS、Android、iOS 是操作系统；浏览器、Notepad、播放器、画图软件是应用软件。

“系统软件”和“应用软件”容易混淆。操作系统、编译器、数据库管理系统等更接近系统软件；办公软件、聊天软件、播放器等直接服务用户任务，通常属于应用软件。

“编译器”和“编辑器”容易混淆。编辑器用来写代码，编译器用来把代码转换成可执行形式。IDE 可能同时包含编辑器、编译器调用和调试功能，但不能把它们混为一个概念。

“编译器”和“调试器”容易混淆。编译器负责翻译源代码，调试器负责辅助观察程序运行过程。设置断点、单步运行、查看变量值，通常属于调试功能。

“输入设备”和“输出设备”容易混淆。键盘、鼠标、扫描仪、传感器偏输入；显示器、打印机、音箱偏输出。触摸屏这类设备可能同时具有输入和输出功能，要结合题目问法判断。

“内存”和“外存”容易混淆。内存用于程序运行时临时存放数据，外存用于长期保存数据。硬盘、U 盘属于外存，不要把它们直接等同于内存。

“位”和“字节”容易混淆。bit 是位，Byte 是字节，1 Byte 等于 8 bit。若题目问“最小单位”，要看它问的是信息表示还是常用文件大小单位。

“算法”和“程序”容易混淆。算法是方法，程序是用某种语言写出的实现。算法可以用不同程序语言实现，同一程序语言也可以写出不同算法。

“二进制”和“十六进制”容易混淆。十六进制常用于简洁书写二进制数据，但它不是计算机底层存储的根本形式。题目问“最终转换成什么数据存储”时，应想到二进制。

“发展历史”和“现实性能”容易混淆。电子管、晶体管、集成电路、大规模集成电路表示不同时代的主要逻辑元件，不是让学生比较现在电脑的具体性能参数。做题时按历史顺序判断即可。

十一、本章要点

计算机常识题范围较杂，但判断方法很稳定。先分清它考的是硬件软件、操作系统、编译器、文件、网络安全，还是计算机领域常识。

硬件是设备，软件是程序。操作系统是管理资源、提供运行环境的系统软件；应用软件完成具体任务。编译器的主要作用是把源代码转换成机器能够执行或进一步处理的低层代码。

输入设备负责把信息送入计算机，输出设备负责把结果呈现出来。内存偏运行时临时存储，外存偏长期保存。程序是指令序列，算法是解决问题的方法，数据是被处理的对象。

计算机具有运算速度快、计算精度高、能进行逻辑判断、存储容量大、能按程序自动工作的特点。计算机发展史中常见对应关系是：电子管、晶体管、集成电路、大规模集成电路依次发展；现代计算机还体现出高性能化、微型化、网络化、智能化和多媒体化等趋势。

图灵奖与计算机领域关系最密切。冯·诺依曼体系、阿兰·图灵、姚期智、ENIAC、CAD、CAM、OA、MIS 等名称属于常识题中可能出现的低频内容，不需要展开成大段知识，只需知道基本对应关系。

这类题不要过度联想。多数题只考基本定义和类别判断。看清选项属于哪一类，再排除明显不符合定义的选项，通常就能稳定得分。

第四编 基础数据结构

0x10 栈

栈是一种只允许在一端进行插入和删除的数据结构。CSP-J 第一轮中，栈常出现在选择题、阅读程序和完善程序里：判断一个出栈序列是否合法，判断表达式括号是否匹配，分析递归调用过程，或者看懂程序中 `push`、`pop`、`top` 的含义。

这一章只讲第一轮需要掌握的栈。重点不是写复杂数据结构模板，而是看懂“后进先出”的规则，并能在纸面上模拟入栈、出栈过程。

一、栈是什么

可以把栈想成一摞书。新放进去的书只能放在最上面，拿书时也只能先拿最上面的那本。后放进去的先拿出来，先放进去的后拿出来，这就是栈的核心规则。

这个规则叫后进先出，也可以记作 LIFO，即 Last In First Out。

栈只关心一端，这一端叫栈顶。插入元素叫入栈，删除栈顶元素叫出栈，查看栈顶元素叫取栈顶。

如果依次把 $1, 2, 3$ 入栈，栈的变化可以这样写：

```
入栈 1：底 [1] 顶
入栈 2：底 [1, 2] 顶
入栈 3：底 [1, 2, 3] 顶
```

此时若连续出栈，出栈顺序一定是 $3, 2, 1$，因为 3 在栈顶，必须最先出来。

栈的题目通常不难，但要求严格遵守“只能操作栈顶”。很多错误出栈序列，看起来只是交换了两个元素，其实违反了栈顶规则。

二、栈的基本操作

栈的基本操作有四类。

`push(x)` 表示把元素 `x` 放到栈顶。

`pop()` 表示弹出栈顶元素。注意，`pop()` 通常只负责删除，不返回这个元素。

`top()` 表示查看当前栈顶元素，但不删除它。

`empty()` 表示判断栈是否为空。

在 C++ 中可以使用标准库 `stack`：

```
stack<int> s;
s.push(1);
s.push(2);
cout << s.top() << endl;
s.pop();
cout << s.top() << endl;
```

这段程序先把 1 入栈，再把 2 入栈。第一次 top() 得到 2。执行 pop() 后，2 被删除，栈顶变成 1，第二次 top() 得到 1。

阅读程序时，要特别注意两点。

第一，不能对空栈执行 top() 或 pop()。如果题目没有保证栈非空，程序就可能出错。

第二，pop() 删除的是当前栈顶，而不是任意元素。若题目说要删除某个特定元素，不能直接用栈完成，除非它正好在栈顶。

CSP-J 第一轮更常考规则理解，不会要求手写完整的栈类。若题目使用数组模拟栈，只要看懂栈顶指针如何变化即可。

三、数组模拟栈

不用标准库时，可以用数组模拟栈。设数组 st 存栈中元素，变量 top 表示栈顶位置。

常见写法有两种。第一种让 top 表示当前栈内元素个数，也就是下一个可插入位置：

```
int st[1005], top = 0;

st[top++] = x;      // 入栈
top--;              // 出栈
st[top - 1];        // 栈顶元素
```

这种写法中，栈为空时 top == 0。入栈后先放到 st[top]，再让 top 增加。栈顶元素的位置是 top - 1。

第二种让 top 表示当前栈顶下标：

```
int st[1005], top = -1;

st[++top] = x;      // 入栈
top--;              // 出栈
st[top];            // 栈顶元素
```

这种写法中，栈为空时 top == -1。入栈时先把 top 加 1，再写入元素。栈顶元素的位置就是 top。

两种写法都可以，但不能混用。阅读程序时，先判断 top 表示“元素个数”还是“栈顶下标”。这是数组模拟栈最容易出错的地方。

例如第一种写法中，如果写 st[top] 作为栈顶元素，就错了，因为 top 指向的是下一个空位置。第二种写法中，如果用 top == 0 判断空栈，也可能错，因为此时栈里还有一个元素。

纸面分析时，可以在草稿上写一句：

`top` 是数量，栈顶在 `top - 1`；

或者：

`top` 是下标，栈顶在 `top`。

先确认这句话，再追踪程序。

四、出栈序列怎么判断

出栈序列题是栈的高频考法。

题目通常给定一个固定的入栈顺序，例如 $1,2,3,4,5$ 依次入栈；中途可以随时出栈；问某个出栈顺序是否可能。

判断方法是：按目标出栈序列从左到右模拟。为了得到当前目标元素，就不断把还没入栈的元素压入栈中，直到目标元素出现在栈顶。若目标元素已经不可能到达栈顶，则该序列不合法。

例如入栈顺序为 $1,2,3,4,5$，判断出栈序列 $2,4,5,3,1$ 是否可能。

目标第一个是 2。先入栈 1，栈顶是 1，不是 2；再入栈 2，栈顶是 2，出栈 2。

目标第二个是 4。继续入栈 3，栈顶是 3，不是 4；再入栈 4，栈顶是 4，出栈 4。

目标第三个是 5。继续入栈 5，栈顶是 5，出栈 5。

目标第四个是 3。此时栈中从底到顶是 $1,3$，栈顶正好是 3，出栈 3。

目标第五个是 1。栈顶是 1，出栈 1。

所有目标都能依次完成，所以这个出栈序列合法。

判断出栈序列时，最重要的不是“感觉顺不顺”，而是模拟栈顶。只要当前目标不在栈顶，就不能出栈它。

五、快速发现不合法序列

有些不合法序列可以用模拟直接发现。

仍以入栈顺序 $1,2,3,4,5$ 为例，判断出栈序列 $3,1,2,5,4$。

为了先出 3，必须先把 $1,2,3$ 依次入栈，然后弹出 3。此时栈从底到顶是 $1,2$，栈顶是 2。

目标第二个是 1，但 1 在 2 下面。栈只能从栈顶弹出，不能越过 2 直接弹出 1。如果弹出 2，出栈序列第二个就变成 2，不是目标中的 1。所以这个序列不合法。

这种题有一个常用判断：当某个较大的元素已经入栈并在某个较小元素之上时，较小元素不可能先于它出栈。因为中间不能跳过栈顶。

但考试时不要只靠这个口头规则。最稳的做法仍然是按目标序列模拟，直到出现“目标不在栈顶，且已经没有办法把它压到栈顶”的位置。

六、真题例题

真题例题 给定一个空栈，支持入栈和出栈操作。若入栈操作的元素依次是 $1,2,3,4,5,6$，其中 1 最先入栈，6 最后入栈，下面哪种出栈顺序是不可能的？

- **A.** $6,5,4,3,2,1$
- **B.** $1,6,5,4,3,2$
- **C.** $2,4,6,5,3,1$
- **D.** $1,3,5,2,4,6$

答案：D

解析：

A 可以实现。把 $1,2,3,4,5,6$ 全部入栈，再依次出栈，得到 $6,5,4,3,2,1$。

B 可以实现。先入栈 1，立即出栈 1。再把 $2,3,4,5,6$ 依次入栈，然后出栈 $6,5,4,3,2$。

C 可以实现。先入栈 $1,2$，出栈 2；再入栈 $3,4$，出栈 4；再入栈 $5,6$，出栈 6；此时栈顶依次是 $5,3,1$，继续出栈，得到 $5,3,1$。完整顺序为 $2,4,6,5,3,1$。

D 不可能。按照目标序列，先入栈 1 并出栈 1。接着目标是 3，需要入栈 $2,3$，然后出栈 3。此时栈中有 2。

接着目标是 5，需要继续入栈 $4,5$，然后出栈 5。此时栈从底到顶是 $2,4$，栈顶是 4。

但目标下一个是 2。由于 2 被 4 压在下面，不能在 4 出栈之前先出栈 2。所以 D 无法实现。

这道题的关键位置是前半段 $1,3,5$。为了依次弹出 $1,3,5$，必须把 2 和 4 留在栈中，而且 4 在 2 上方。后面若要求先弹出 2，就违反了后进先出规则。

七、括号匹配

括号匹配是栈的另一类常见应用。遇到左括号时入栈，遇到右括号时检查栈顶是否是对应的左括号。

例如表达式中只考虑圆括号。扫描字符串：

遇到 `(`，入栈；

遇到 `)`，如果栈为空，则不匹配；如果栈顶是 `(`，弹出栈顶；

扫描结束后，如果栈为空，则匹配；否则还有左括号没被匹配。

对于多种括号，如 `()`、`[]`、`{}`，规则也类似。右括号必须和最近的未匹配左括号配对。这个“最近”正好符合栈的后进先出规则。

```
bool match(string s) {
    stack<char> st;
    for (char c : s) {
        if (c == '(' || c == '[' || c == '{') {
            st.push(c);
        } else if (c == ')' || c == ']' || c == '}') {
            if (st.empty()) return false;
```

```
            char t = st.top();
            if (c == ')' && t != '(') return false;
            if (c == ']' && t != '[') return false;
            if (c == '}' && t != '{') return false;
            st.pop();
        }
    }
    return st.empty();
}
```

读这段程序时，可以只抓住三件事。

第一，左括号入栈，等待未来匹配。

第二，右括号只和栈顶比较，因为它必须匹配最近的未匹配左括号。

第三，扫描结束后栈必须为空，否则说明还有左括号没有被匹配。

例如字符串 `([{}])` 是匹配的。扫描到右括号时，栈顶依次是 `{`、`[`、`(`，都能正确弹出。

字符串 `([)]` 不匹配。扫描到 `)` 时，栈顶是 `[`，不是 `(`，说明括号交叉了。

括号匹配题并不要求记长代码，只要理解“最近未匹配”的含义，就能想到栈。

八、栈和递归

递归调用也具有栈的特点。一个函数调用另一个函数时，后调用的函数要先结束，前面的函数才能继续执行。这个过程与后进先出一致。

例如：

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

调用 `f(3)` 时，先输出 3，再调用 `f(2)`；`f(2)` 先输出 2，再调用 `f(1)`；`f(1)` 先输出 1，再调用 `f(0)`。

当 `f(0)` 返回后，程序回到 `f(1)`，输出 1；再回到 `f(2)`，输出 2；最后回到 `f(3)`，输出 3。

所以输出是：

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

这与栈的过程类似：`f(3)` 最先开始，却最后结束；`f(1)` 后开始，却先完成。阅读递归程序时，可以把每次函数调用看成一次入栈，把函数返回看成一次出栈。

本章只用栈帮助理解递归调用的后进先出。递归题的完整分析方法会在后续章节继续讲。

九、栈和表达式

表达式处理中也常用栈。第 18 章会专门讲表达式，本章只说明栈为什么适合这类问题。

中缀表达式是平时常见写法，例如 `a + b * c`。后缀表达式把运算符放在操作数后面，例如 `a b c * +`。计算后缀表达式时，可以用栈保存还没有参与运算的数。

例如计算后缀表达式 `2 3 4 * +`：

读到 2，入栈；

读到 3，入栈；

读到 4，入栈；

读到 *，弹出 4 和 3，计算 $3 \times 4 = 12$，把 12 入栈；

读到 +，弹出 12 和 2，计算 $2 + 12 = 14$，把 14 入栈。

最后栈中剩下的 14 就是结果。

这里的关键仍然是后进先出。最近读到的操作数，往往最先参与当前运算。表达式章节会继续讲中缀、前缀、后缀之间的转换，本章只需要知道：栈是表达式处理中的常用工具。

十、变式题

变式题 给定一个空栈，入栈顺序为 $1, 2, 3, 4, 5$，下面哪个出栈序列不可能？

- **A.** $5, 4, 3, 2, 1$
- **B.** $2, 1, 4, 3, 5$
- **C.** $3, 5, 4, 2, 1$
- **D.** $2, 4, 1, 3, 5$

答案：D

解析：
A 可以实现。把 $1, 2, 3, 4, 5$ 全部入栈，再全部出栈即可。

B 可以实现。先入栈 $1, 2$，出栈 $2, 1$；再入栈 $3, 4$，出栈 $4, 3$；最后入栈并出栈 5。

C 可以实现。先入栈 $1, 2, 3$，出栈 3；再入栈 $4, 5$，出栈 $5, 4$；此时栈顶依次是 $2, 1$，继续出栈即可。

D 不可能。为了先出 2，必须先入栈 $1, 2$，然后出栈 2，此时栈中有 1。目标下一个是 4，需要继续入栈 $3, 4$，然后出栈 4。此时栈从底到顶是 $1, 3$。

目标下一个是 1，但栈顶是 3，1 被压在 3 下面，不能先出栈。因此 D 不可能。

这道题和真题的判断方法相同。不要试图凭选项形状猜答案，逐步模拟更稳。

十一、本章要点

栈的核心规则是后进先出。所有栈题都要围绕“只能操作栈顶”来判断。

入栈是把元素放到栈顶，出栈是删除栈顶元素，取栈顶只查看不删除。不能对空栈执行取栈顶或出栈操作。数组模拟栈时，要先判断 top 表示元素个数还是栈顶下标。

判断出栈序列是否合法，最稳的方法是按目标序列模拟：为了得到当前目标，不断把还未入栈的元素压入栈；若目标在栈顶，就出栈；若目标被其他元素压住，就不合法。

括号匹配使用栈，是因为右括号必须匹配最近的未匹配左括号。递归调用过程也具有栈的特点，后调用的函数先返回。表达式处理中，栈常用于保存暂时还没有参与运算的操作数或运算符。

做栈题时，只要草稿上始终写清当前栈从底到顶的内容，就不容易乱。每一步只问一个问题：目标元素是不是栈顶？如果不是，它还能不能通过继续入栈变成栈顶？这就是栈题最可靠的判断方法。

0x11 队列

队列是一种只允许从一端加入、从另一端取出的数据结构。它和上一章的栈很像，都属于线性数据结构，但规则正好不同：栈是后进先出，队列是先进先出。

CSP-J 第一轮中，队列常出现在选择题和程序阅读中。常见考法包括：判断队列的访问规则，区分队列和栈，理解 `push`、`pop`、`front` 的含义，分析数组模拟队列的头尾指针变化，判断队列在广度优先搜索中的作用。

这一章不写复杂模板，只讲第一轮需要掌握的队列规则和纸面模拟方法。第 27 章搜索会继续使用队列，本章先把队列本身讲清楚。

一、队列是什么

队列可以理解为排队。先来的人排在前面，后到的人排在后面；办理业务时，只能从队首离开，新来的人只能从队尾加入。

这个规则叫先进先出，也可以记作 FIFO，即 First In First Out。

队列有两个重要位置：队首和队尾。队首是即将被取出的元素所在位置，队尾是新元素加入的位置。

例如依次把 $1, 2, 3$ 加入队列，队列变化如下：

```
入队 1：队首 [1] 队尾
入队 2：队首 [1, 2] 队尾
入队 3：队首 [1, 2, 3] 队尾
```

此时若连续出队，出队顺序一定是 $1, 2, 3$。因为 1 最先进队，所以最先出队；3 最后进队，所以最后出队。

这和栈不同。若 $1, 2, 3$ 依次入栈，出栈顺序可能是 $3, 2, 1$；但若 $1, 2, 3$ 依次入队，且中间不改变规则，出队顺序一定保持 $1, 2, 3$。

队列题的核心判断只有一句话：队首出，队尾进。任何试图从中间取元素、从队首插入新元素、从队尾删除旧元素的操作，都不是普通队列的规则。

二、队列的基本操作

队列的常见操作有四类。

`push(x)` 表示把元素 `x` 加到队尾。

`pop()` 表示删除队首元素。注意，`pop()` 通常只删除，不返回队首元素。

`front()` 表示查看队首元素，但不删除它。

`empty()` 表示判断队列是否为空。

C++ 标准库中可以使用 `queue`：

```
queue<int> q;
q.push(1);
q.push(2);
cout << q.front() << endl;
q.pop();
cout << q.front() << endl;
```

这段程序先把 1 入队，再把 2 入队。第一次 `front()` 得到 1，因为 1 在队首。执行 `pop()` 后，1 被删除，队首变成 2，第二次 `front()` 得到 2。

还可以使用 `back()` 查看队尾元素，使用 `size()` 查看队列中元素个数。但第一轮中最常考的是 `push`、`pop`、`front` 和 `empty`。

阅读队列程序时，要注意两点。

第一，不能对空队列执行 `front()` 或 `pop()`。如果队列为空还取队首，程序行为就不可靠。题目若没有说明队列非空，要检查程序是否先用了 `empty()` 判断。

第二，`pop()` 删除的是队首，不是队尾。很多学生受栈的影响，会把 `pop()` 理解成删除最后加入的元素。对队列来说，最后加入的元素在队尾，不能被普通 `pop()` 直接删除。

三、队列和栈的区别

栈和队列都能保存一串数据，但访问规则不同。

栈只在栈顶操作，后进入的元素先出来。队列在队尾加入，在队首删除，先进入的元素先出来。

例如按顺序加入 $1, 2, 3, 4$。

如果是栈，全部加入后再全部取出，顺序是 $4, 3, 2, 1$。

如果是队列，全部加入后再全部取出，顺序是 $1, 2, 3, 4$。

可以用一个简单判断区分它们：

栈像一摞盘子，最后放上去的最先拿走；

队列像排队买票，先排的人先办理。

选择题中常出现类似说法：

“栈的访问原则是后进先出，队列的访问原则是先进先出。”这是正确的。

“队列常用于广度优先搜索。”这是正确的。

“栈和队列规则不同，所以栈不能实现队列。”这句话要谨慎。普通栈和普通队列的访问规则不同，但可以用两个栈配合模拟队列。第一轮如果问“表述不恰当的一项”，这类过于绝对的说法常是干扰

点。

本章不展开两个栈实现队列的完整代码，只记住结论：栈和队列是不同的数据结构，但数据结构之间可以通过额外操作互相模拟。不能因为规则不同，就说完全无法实现。

四、数组模拟队列

除了使用标准库，也可以用数组模拟队列。队列需要维护两个位置：队首和队尾。

常见写法是用 `head` 表示队首位置，用 `tail` 表示队尾后面的第一个空位置。也就是说，队列中的元素位于区间 `[head, tail)`。

```
int q[1005];
int head = 0, tail = 0;

q[tail++] = x;      // 入队
head++;             // 出队
q[head];            // 队首元素
```

这种写法中，队列为空的条件是 `head == tail`。入队时，把元素放到 `q[tail]`，再让 `tail` 向后移动。出队时，不需要真的清空数组，只要让 `head` 向后移动，表示原队首已经离开。

例如依次入队 $10, 20, 30$，再出队一次：

开始：`head = 0`，`tail = 0`，队列为空。

入队 10：放入 `q[0]`，`tail` 变为 1，队首是 `q[0]`。

入队 20：放入 `q[1]`，`tail` 变为 2。

入队 30：放入 `q[2]`，`tail` 变为 3。

出队一次：`head` 从 0 变为 1，队首变为 `q[1]`，也就是 20。

此时数组中 `q[0]` 可能还保存着 10，但它已经不属于队列。判断队列内容时，只看从 `head` 到 `tail - 1` 的位置。

数组模拟队列时，最容易混淆的是 `tail` 的含义。上面的写法中，`tail` 不是当前队尾元素的位置，而是队尾后面的空位置。因此队尾元素在 `tail - 1`。若把 `q[tail]` 当作队尾元素，就会读到还没有加入队列的位置。

也有写法让 `tail` 表示当前队尾位置，但本书建议在纸面分析中优先使用 `[head, tail)` 这种写法，因为它和很多循环区间一致，空队列判断也简单。

五、循环队列

普通数组模拟队列时，`head` 和 `tail` 会不断向右移动。即使前面的元素已经出队，数组前部空间也不会再次使用。如果操作次数很多，`tail` 可能超过数组范围。

循环队列就是为了解决这个问题。它把数组看成一个环，走到末尾后回到开头。

设队列数组大小为 N，位置向后移动一格时，可以写成：

```
x = (x + 1) % N;
```

当 x 还没到末尾时，它正常加 1；当 x 等于 N - 1 时，(x + 1) % N 变成 0，回到数组开头。

循环队列仍然可以用 head 表示队首位置，用 tail 表示队尾后面的空位置。

```
int q[1005];
int head = 0, tail = 0;

q[tail] = x;
tail = (tail + 1) % N;

head = (head + 1) % N;
```

循环队列中，head == tail 通常表示队列为空。但若把数组全部放满，也可能出现 head == tail。为了避免空和满无法区分，常见做法是少用一个位置：当 (tail + 1) % N == head 时，认为队列已满。

CSP-J 第一轮不常要求完整实现循环队列，但常会考取余让下标回到范围内。看到 (i + 1) % n，就要想到“走到末尾后回到开头”。这在循环队列、约瑟夫问题、环形模拟中都很常见。

判断循环队列时，不要只看数组下标大小。下标可能从末尾回到 0，队列顺序由 head 到 tail 沿着环走，而不是简单从小到大排列。

六、队列的纸面追踪

阅读队列程序时，最稳的方法是画一行表，记录每一步的操作、队列内容、队首和队尾。

例如：

```
queue<int> q;
q.push(3);
q.push(5);
q.pop();
q.push(7);
cout << q.front() << endl;
```

逐步追踪：

```
开始：[]
push 3：[3]
push 5：[3, 5]
pop：[5]
push 7：[5, 7]
front：5
```

所以输出是 5。

如果使用数组模拟队列，也要同时记录 head 和 tail。

```
int q[10], head = 0, tail = 0;
q[tail++] = 4;
q[tail++] = 6;
head++;
q[tail++] = 8;
cout << q[head] << endl;
```

追踪如下：

```
开始：head = 0, tail = 0
入队 4：q[0] = 4, head = 0, tail = 1
入队 6：q[1] = 6, head = 0, tail = 2
出队一次：head = 1, tail = 2
入队 8：q[2] = 8, head = 1, tail = 3
队首 q[head] = q[1] = 6
```

输出是 6。

这类题不要在脑中直接跳结果。队列不像普通数组那样总从 0 开始读，队首会随着出队变化。只要 head 移动过，就要按当前 head 找队首。

七、队列与广度优先搜索

队列最典型的算法应用是广度优先搜索，简称 BFS。搜索章节会专门讲 BFS，本节只说明队列在其中起到什么作用。

广度优先搜索的基本思想是：先处理距离起点近的状态，再处理距离更远的状态。队列的先进先出规则正好适合这个过程。

以网格中从起点向上下左右扩展为例：

先把起点入队；

每次取出队首位置；

把它能到达、还没访问过的相邻位置加入队尾；

重复直到队列为空或找到目标。

因为先入队的是较早发现的位置，所以会先被处理；后发现的位置排在队尾，等前面的位置处理完再处理。这样就形成了“一层一层向外扩展”的效果。

示意代码如下：

```
queue<int> q;
q.push(start);
vis[start] = true;

while (!q.empty()) {
    int u = q.front();
    q.pop();
```

```
    for (int v : next[u]) {
        if (!vis[v]) {
            vis[v] = true;
            q.push(v);
        }
    }
}
```

在这段代码中，`front()` 取出当前要处理的状态，`pop()` 删除它，`push()` 把新发现的状态放到队尾。

第一轮中，如果题目问“队列常用于哪种搜索”，通常应想到广度优先搜索。深度优先搜索更常用递归或栈。二者不是绝对只能这样实现，但在基础考法中，BFS 和队列的对应关系很常见。

八、双端队列和优先队列

普通队列只能队尾进、队首出。还有一些名字里带“队列”的结构，规则并不完全相同。

双端队列可以在两端插入和删除。C++ 中常见容器是 `deque`。它既可以从前端操作，也可以从后端操作。双端队列比普通队列灵活，但第一轮通常只要求知道它不是普通 FIFO 队列。

优先队列每次取出的不是最早进入的元素，而是优先级最高的元素。C++ 中常见容器是 `priority_queue`。默认情况下，较大的元素优先取出。

例如依次加入 $3, 1, 5$。普通队列的出队顺序是 $3, 1, 5$；优先队列的取出顺序通常是 $5, 3, 1$。

所以看到 `priority_queue` 时，不能按先进先出分析。它虽然名字中有队列，但规则更接近“按优先级取元素”。CSP-J 第一轮中若出现这一点，通常是选择题或简单程序阅读，不会要求深入实现。

本章重点仍是普通队列。双端队列和优先队列只需要知道它们与普通队列不同，避免把所有“队列”都理解成 FIFO。

九、真题例题

真题例题 以下对数据结构的表述不恰当的一项为：

- **A.** 图的深度优先遍历算法常使用的数据结构为栈。
- **B.** 栈的访问原则为后进先出，队列的访问原则是先进先出。
- **C.** 队列常常被用于广度优先搜索算法。
- **D.** 栈与队列存在本质不同，无法用栈实现队列。

答案：D

解析：
A 是正确的。深度优先遍历需要沿着一条路径不断深入，再回退到之前的位置继续尝试。这个过程和栈的后进先出特点相符。递归实现深度优先搜索时，也可以理解为使用了类似栈的调用过程。

B 是正确的。栈是后进先出，队列是先进先出，这是二者最基本的区别。

C 是正确的。广度优先搜索需要按发现顺序逐层处理状态，队列的先进先出规则正好适合这一过程。

D 不恰当。栈和队列的访问规则确实不同，但不能推出“无法用栈实现队列”。通过两个栈配合，可以模拟普通队列的入队和出队过程。第一轮中不需要写出完整实现，但要知道“规则不同”和“无法实现”不是同一件事。

这道题的考点不是某个代码细节，而是基础数据结构的性质辨析。遇到“无法”“一定”“只能”这类绝对表述时，要格外小心。

十、变式题

变式题 下面程序的输出是什么？

```cpp
#include <iostream>
#include <queue>
using namespace std;

int main() {
    queue<int> q;
    q.push(2);
    q.push(4);
    q.pop();
    q.push(6);
    q.push(8);
    q.pop();
    cout << q.front() << endl;
    return 0;
}
```

A. 2

B. 4

C. 6

D. 8

答案：C

解析：
按队列内容逐步追踪。

开始队列为空。

执行 `q.push(2)` 后，队列为 `[2]`。

执行 `q.push(4)` 后，队列为 `[2, 4]`，队首是 2。

执行第一次 `q.pop()`，删除队首 2，队列变为 `[4]`。

执行 `q.push(6)`，队列为 `[4, 6]`。

执行 `q.push(8)`，队列为 `[4, 6, 8]`。

执行第二次 `q.pop()`，删除队首 4，队列变为 `[6, 8]`。

此时 `q.front()` 是 6，所以输出 6。

这道题的易错点是把 pop() 当成删除队尾。如果误以为第二次 pop() 删除 8，就会得到错误答案。普通队列的 pop() 删除的是队首。

十一、本章要点

队列的核心规则是先进先出。队尾入队，队首出队。普通队列不能从中间取元素，也不能直接删除队尾元素。

C++ 中，push(x) 把元素加入队尾，pop() 删除队首元素，front() 查看队首元素，empty() 判断是否为空。不能对空队列执行 front() 或 pop()。

数组模拟队列时，常用 head 表示队首位置，tail 表示队尾后面的第一个空位置。队列为空时 head == tail，队首元素是 q[head]。若使用循环队列，位置移动常写成 (x + 1) % N。

栈和队列的区别要清楚：栈是后进先出，队列是先进先出。它们规则不同，但不能因此说二者完全不能互相模拟。

队列常用于广度优先搜索，因为 BFS 需要按发现顺序逐层处理状态。深度优先搜索更常用递归或栈。普通队列、双端队列、优先队列规则不同，读题时要看清使用的是哪一种结构。

做队列程序阅读题时，草稿上只需要维护当前队列从队首到队尾的内容。每遇到一次 push 就在队尾加入，每遇到一次 pop 就删除队首，遇到 front 就读当前队首。只要按这个规则追踪，队列题通常不会复杂。

0x12 表达式

表达式题在 CSP-J 第一轮中常和栈一起出现。题目可能要求判断一个中缀表达式对应的后缀表达式，也可能给出一个后缀表达式，要求还原成中缀表达式，或者直接计算后缀表达式的值。

这类题的核心不是记住很多符号，而是理解三件事：运算顺序由什么决定，前缀、中缀、后缀表达式有什么区别，为什么栈适合处理表达式。

本章不展开复杂语法分析，只讲第一轮中最常见的表达式规则和纸面转换方法。

一、表达式由什么组成

表达式通常由操作数、运算符和括号组成。

操作数可以是数字，也可以是变量。例如 `3`、`a`、`b` 都可以作为操作数。

运算符表示要进行的运算。例如 `+`、`-`、`*`、`/` 都是常见运算符。

括号用于改变或明确运算顺序。例如 `a + b * c` 中，乘法先算；而 `(a + b) * c` 中，括号内的加法先算。

在普通中缀表达式中，运算符写在两个操作数中间，例如 `a + b`。这是最符合日常书写习惯的形式。

表达式题最重要的是运算顺序。不同写法如果运算顺序不同，就不是同一个表达式。例如 `a - b - c` 通常按从左到右计算，即 `(a - b) - c`，不等于 `a - (b - c)`。

在 CSP-J 第一轮的表达式题中，常见优先级可以按下面理解：

括号优先级最高；

乘法、除法优先于加法、减法；

同一优先级通常从左到右结合；

题目若使用 `^` 表示乘方，要按题目给出的数学表达式规则理解，不要和 C++ 中的 `^` 混淆。

特别注意：在 C++ 中，`^` 不是乘方运算符，而是按位异或运算符。表达式选择题中若把 `^` 当作数学乘方，通常是题目在讨论表达式表示法，不是在考 C++ 运算符含义。读题时要看清题目语境。

二、中缀、前缀和后缀

同一个运算，可以有不同的表达式写法。

中缀表达式把运算符写在两个操作数中间。例如：

```
a + b
```

```
a * (b + c)
```

前缀表达式把运算符写在操作数前面。例如 `a + b` 的前缀表达式是：

```
+ a b
```

后缀表达式把运算符写在操作数后面。例如 `a + b` 的后缀表达式是：

```
a b +
```

再看一个稍复杂的表达式：`a + b * c`。

因为乘法先算，所以它等价于 `a + (b * c)`。

其中 `b * c` 的后缀表达式是 `b c *`。再与 `a` 做加法，后缀表达式是：

```
a b c * +
```

它的前缀表达式是：

```
+ a * b c
```

这三种写法表达的是同一个运算结构：

中缀：`a + b * c`

前缀：`+ a * b c`

后缀：`a b c * +`

中缀表达式需要依靠括号和优先级判断顺序。前缀和后缀表达式把运算符位置固定后，通常不需要括号也能表达清楚运算顺序。

三、从中缀到后缀：先看运算树

从中缀表达式转后缀表达式，最适合纸面题的方法是先确定运算结构。

例如表达式 `a * (b + c) * d`。

先看括号，`b + c` 要先算。原式可以看作：

```
(a * (b + c)) * d
```

最外层最后做的是乘以 `d` 的乘法。左边是 `a * (b + c)`，右边是 `d`。

先处理左边 `a * (b + c)`。其中 `(b + c)` 的后缀表达式是 `b c +`，再与 `a` 相乘，得到：

```
a b c + *
```

最后再与 `d` 相乘，得到：

```
a b c + * d *
```

所以 `a * (b + c) * d` 的后缀表达式是：

a b c + * d *

这也是第一轮中非常常见的考法。选项里如果出现 abc+*d*，就对应这个结果。

把中缀表达式转后缀表达式，可以记住一个简洁规则：

先写左子表达式的后缀；

再写右子表达式的后缀；

最后写当前运算符。

例如 (a + b) * c，当前最后做乘法。左边 (a + b) 的后缀是 a b +，右边 c 的后缀是 c，最后写 *，得到：

a b + c *

如果是 a + b * c，当前最后做加法。左边是 a，右边 b * c 的后缀是 b c *，最后写 +，得到：

a b c * +

做题时，先给中缀表达式按优先级加上必要括号，再按“左、右、根”的顺序写，就不容易乱。

四、从后缀到中缀：用栈合并

后缀表达式转中缀表达式，最稳的方法是用栈。

从左到右扫描后缀表达式：

遇到操作数，就把它入栈；

遇到运算符，就从栈中弹出两个表达式，先弹出的是右操作数，后弹出的是左操作数；

把它们合成一个新的中缀表达式，再入栈。

例如后缀表达式：

a b c + * d *

扫描过程如下。

读到 a，入栈，栈中是 a。

读到 b，入栈，栈中是 a，b。

读到 c，入栈，栈中是 a，b，c。

读到 +，弹出 c 和 b，合成 (b + c)，入栈。栈中是 a，(b + c)。

读到 *，弹出 (b + c) 和 a，合成 (a * (b + c))，入栈。

读到 d，入栈。

读到最后一个 *，弹出 d 和 (a * (b + c))，合成 ((a * (b + c)) * d)。

所以对应中缀表达式可以写作 `(a * (b + c)) * d`。

这里有一个很重要的顺序：遇到运算符时，先弹出的表达式是右操作数，后弹出的表达式是左操作数。对于 `+` 和 `*`，左右交换结果可能一样；但对于 `-` 和 `/`，左右交换会改变结果。

例如后缀表达式 `6 2 -` 表示 `6 - 2`，不是 `2 - 6`。扫描到 `-` 时，先弹出的是 `2`，它是右操作数；后弹出的是 `6`，它是左操作数。

五、计算后缀表达式

后缀表达式也可以直接用栈计算。

规则和后缀转中缀相似：

遇到数字就入栈；

遇到运算符就弹出两个数；

后弹出的数作为左操作数，先弹出的数作为右操作数；

计算结果再入栈。

例如计算后缀表达式：

```
2 3 4 * +
```

读到 2，入栈；

读到 3，入栈；

读到 4，入栈；

读到 `*`，弹出 4 和 3，计算 $3 \times 4 = 12$，把 12 入栈；

读到 `+`，弹出 12 和 2，计算 $2 + 12 = 14$，把 14 入栈。

最后栈中剩下的 14 就是表达式的值。

再看一个涉及减法的例子：

```
8 3 2 * -
```

先算 `3 2 *` 得到 6，再算 `8 - 6`，结果是 2。用栈模拟时，遇到 `-` 后，先弹出 6，后弹出 8，所以是 $8 - 6$。

如果把顺序弄反，会算成 $6 - 8$。这是后缀计算中最常见的错误。

六、从中缀到后缀：栈方法

对于较长表达式，可以用运算符栈把中缀表达式转成后缀表达式。第一轮一般不会要求写完整程序，但理解这个过程有助于判断选项。

从左到右扫描中缀表达式：

遇到操作数，直接输出；

遇到左括号，入栈；

遇到右括号，不断弹出并输出栈顶运算符，直到遇到左括号，再把左括号弹出但不输出；

遇到运算符，若栈顶运算符优先级不低于它，就先弹出栈顶运算符并输出，然后再把当前运算符入栈；

扫描结束后，把栈中剩余运算符依次弹出并输出。

例如中缀表达式 `a + b * c`。

读到 `a`，输出 `a`。

读到 `+`，运算符栈为空，`+` 入栈。

读到 `b`，输出 `b`。

读到 `*`，栈顶是 `+`，乘法优先级高于加法，所以 `*` 入栈。

读到 `c`，输出 `c`。

扫描结束，依次弹出运算符栈中的 `*` 和 `+`，得到后缀表达式：

```
a b c * +
```

再看 `a * (b + c)`。

读到 `a`，输出；

读到 `*`，入栈；

读到 `(`，入栈；

读到 `b`，输出；

读到 `+`，入栈；

读到 `c`，输出；

读到 `)`，弹出 `+` 并输出，再弹出左括号；

扫描结束，弹出 `*`。

得到后缀表达式：

```
a b c + *
```

这个方法的关键是：操作数直接输出，运算符暂时入栈，等到它确定应该计算时再输出。括号本身不出现在后缀表达式中，它只负责控制运算符什么时候出栈。

七、前缀表达式

前缀表达式把运算符写在操作数前面。它不如后缀表达式在 CSP-J 第一轮中常见，但选择题中可能出现。

例如中缀表达式 `a + b * c` 的前缀表达式是：

```
+ a * b c
```

它的结构可以这样看：最后做加法，所以最前面写 `+`；左操作数是 `a`；右操作数是 `b * c`，它的前缀是 `* b c`。

从中缀转前缀，可以记住一个规则：

先写当前运算符；

再写左子表达式的前缀；

最后写右子表达式的前缀。

例如 `(a + b) * c`。最后做乘法，所以先写 `*`；左边 `(a + b)` 的前缀是 `+ a b`；右边是 `c`。因此前缀表达式是：

```
* + a b c
```

前缀表达式也可以用栈计算，不过通常从右向左扫描。第一轮中如果只是选择题，更多时候只需按结构判断，不需要写完整计算过程。

八、真题例题

真题例题 后缀表达式 `6 2 3 + - 3 8 2 / + * 2 ^ 3 +` 对应的中缀表达式是哪一个？

A. `((6 - (2 + 3)) * (3 + 8 / 2)) ^ 2 + 3`
B. `6 - 2 + 3 * 3 + 8 / 2 ^ 2 + 3`
C. `(6 - (2 + 3)) * ((3 + 8 / 2) ^ 2) + 3`
D. `6 - ((2 + 3) * (3 + 8 / 2)) ^ 2 + 3`

答案：A

解析：
用栈从左到右扫描后缀表达式。

先看前半段 `6 2 3 + -`。

读到 `2 3 +`，合成 `(2 + 3)`。再读到 `-`，左操作数是 `6`，右操作数是 `(2 + 3)`，合成：

```
(6 - (2 + 3))
```

再看中间的 `3 8 2 / +`。

读到 `8 2 /`，合成 `(8 / 2)`。再与 `3` 做加法，合成：

```
(3 + 8 / 2)
```

接着读到 *，把前面两个大表达式相乘，得到：

```
((6 - (2 + 3)) * (3 + 8 / 2))
```

再读到 2 ^，表示把上面的结果与 2 做乘方，得到：

```
((6 - (2 + 3)) * (3 + 8 / 2)) ^ 2
```

最后读到 3 +，得到：

```
((6 - (2 + 3)) * (3 + 8 / 2)) ^ 2 + 3
```

所以答案是 A。

这道题的关键不是从选项硬猜，而是把后缀表达式按栈合并成几个大块。先合并 2 3 +，再合并 8 2 /，再合并乘法、乘方和最后的加法，结构就会清楚。

九、变式题

变式题 中缀表达式 (a + b) * (c - d) 的后缀表达式是哪个？

A. a b + c d - *
B. a b c d + - *
C. a b + * c d -
D. * + a b - c d

答案：A

解析：
先看整个表达式。最外层最后做乘法，左边是 (a + b)，右边是 (c - d)。

左边 (a + b) 的后缀表达式是 a b +。

右边 (c - d) 的后缀表达式是 c d -。

最后把左右两部分相乘，所以在最后写 *。

得到：

```
a b + c d - *
```

因此答案是 A。

B 把左边和右边的结构打乱了；C 把乘号放得太早；D 是前缀表达式，不是后缀表达式。

十、易错点

第一，后缀表达式中遇到运算符时，先弹出的是右操作数，后弹出的是左操作数。减法、除法尤其不能反。

例如 10 4 - 是 $10-4$，不是 $4-10$。

第二，括号不会出现在后缀表达式里。括号的作用是改变中缀表达式中的运算顺序。转成后缀表达式后，顺序已经由运算符位置体现，括号就不需要写出来。

第三，不要把表达式题中的 ^ 自动当成 C++ 乘方。C++ 中 ^ 是按位异或。若题目讨论数学表达式中的乘方，会在语境中体现。做题时按题面语境处理。

第四，同一优先级的运算要注意结合方向。普通加减乘除中，加减和乘除通常按从左到右处理。例如 `a - b - c` 是 `(a - b) - c`，不是 `a - (b - c)`。

第五，前缀和后缀不要看反。前缀是运算符在前，如 `+ a b`；后缀是运算符在后，如 `a b +`。如果选项以运算符开头，很可能是前缀；如果选项以运算符结尾，很可能是后缀，但仍要检查完整结构。

十一、本章要点

中缀表达式把运算符写在操作数中间，前缀表达式把运算符写在操作数前面，后缀表达式把运算符写在操作数后面。

从中缀转后缀，可以先按优先级和括号确定运算结构，再按“左子表达式、右子表达式、当前运算符”的顺序写。前缀表达式则按“当前运算符、左子表达式、右子表达式”的顺序写。

从后缀转中缀或计算后缀表达式时，使用栈。遇到操作数入栈，遇到运算符弹出两个元素并合并。先弹出的是右操作数，后弹出的是左操作数。

表达式题常考的不是复杂计算，而是运算顺序。做题时先加括号、分清左右，再写前缀或后缀。只要结构判断正确，选项通常可以很快排除。

0x13 树与二叉树

树是一类非常重要的数据结构。它不像数组那样排成一条线，而是从一个起点向下分出许多分支。CSP-J 第一轮中，树与二叉树常出现在选择题里：判断树的节点数和边数关系，求二叉树遍历序列，根据前序和中序推出后序，判断完全二叉树的形态，或者根据数组下标判断父子节点位置。

这一章不讲复杂树上算法，只讲第一轮最常考、最容易手算出错的内容。重点是四件事：树的基本概念，二叉树的几种形态，三种深度优先遍历，完全二叉树的数组表示。

一、树的基本概念

树由节点和边组成。节点表示对象，边表示节点之间的连接关系。一棵树通常有一个根节点，从根节点出发向下连接其他节点。

如果一个节点直接连到下面的节点，那么上面的节点叫父节点，下面的节点叫子节点。没有子节点的节点叫叶子节点，也叫终端节点。不是叶子的节点叫非叶节点。

例如一棵树中，A 是根节点，A 有两个子节点 B、C；B 有两个子节点 D、E；C 有一个子节点 F。那么 A 是 B、C 的父节点，B 是 D、E 的父节点，D、E、F 都是叶子节点。

树的层数可以按根节点开始计数。本书中，如果题目说“根节点高度为 1”，就按根节点在第 1 层处理。根节点的子节点在第 2 层，再下面是第 3 层。不同资料有时会把根节点深度记为 0，但考试题会说明口径。做题时以题面为准。

节点的度指它有多少个子节点。树的度指整棵树中节点度数的最大值。若每个节点最多有 2 个子节点，这种树就是二叉树。

树有一个非常常用的性质：如果一棵树有 n 个节点，那么它有 $n-1$ 条边。原因是除根节点外，每个节点都有唯一一条边连接到它的父节点，所以边数正好等于非根节点数。

反过来，如果一个连通结构中有 n 个节点、$n-1$ 条边，并且没有环，它就是一棵树。CSP-J 第一轮不会要求证明，但会用这个性质判断简单选择题。

二、二叉树

二叉树是每个节点最多有两个子节点的树。二叉树中，两个子节点通常分为左孩子和右孩子。左、右的位置有区别。

例如，根节点 A 的左孩子是 B、右孩子是 C，与根节点 A 的左孩子是 C、右孩子是 B，是两棵不同的二叉树。即使节点集合一样，只要左右结构不同，二叉树的形态就不同。

这点很容易被忽略。普通树有时只关心父子关系，而二叉树还要关心左孩子和右孩子的位置。

二叉树中的节点可以按度数分为三类：

度为 0 的节点：没有孩子，也就是叶子节点；

度为 1 的节点：只有一个孩子，可能只有左孩子，也可能只有右孩子；

度为 2 的节点：有两个孩子。

二叉树有一个常用结论：在任意非空二叉树中，若叶子节点数为 n_0，度为 2 的节点数为 n_2，则 $n_0 = n_2 + 1$。第一轮中如果出现这类计数题，可以直接使用。这个结论不要求长篇证明，只要记住它只适用于二叉树，并且这里的 n_2 是有两个孩子的节点数。

例如一棵二叉树中有 5 个度为 2 的节点，那么叶子节点数是 6。度为 1 的节点数量不影响这个等式。

三、满二叉树、完全二叉树和二叉搜索树

二叉树的名称很多，第一轮中最常见的是满二叉树、完全二叉树和二叉搜索树。

满二叉树常指每一层都放满节点的二叉树。若根节点为第 1 层，高度为 h，则满二叉树共有 $2^h - 1$ 个节点。第 h 层有 2^{h-1} 个节点。

例如高度为 3 的满二叉树，共有 $2^3 - 1 = 7$ 个节点。第 1 层 1 个，第 2 层 2 个，第 3 层 4 个。

完全二叉树比满二叉树稍宽松。它要求除了最后一层外，前面各层都要放满；最后一层的节点必须从左到右连续排列，不能中间空着，也不能右边有节点而左边缺节点。

例如高度为 4 的完全二叉树，前 3 层必须放满，共有 $1 + 2 + 4 = 7$ 个节点。第 4 层可以有 1 到 8 个节点，但必须从左向右依次出现。因此高度为 4 的完全二叉树，节点总数可以是 8 到 15。

这也解释了一个常见选择题：若一棵完全二叉树高度为 5，前 4 层必须放满，共有 $1+2+4+8 = 15$ 个节点。第 5 层可以有 1 到 16 个节点。每一种最后一层节点数量，对应一种从左到右填充的形态，所以共有 16 种不同形态。

二叉搜索树是一种带有大小顺序的二叉树。通常要求左子树中的值小于根节点，右子树中的值大于根节点。CSP-J 第一轮中可能会把二叉搜索树和中序遍历联系起来：对二叉搜索树进行中序遍历，通常会得到从小到大的序列。本章只先记住这一点，不展开插入、删除等操作。

四、二叉树的三种遍历

遍历就是按某种顺序访问树中的所有节点。二叉树最常考三种深度优先遍历：前序、中序、后序。

前序遍历的顺序是：根，左，右。

中序遍历的顺序是：左，根，右。

后序遍历的顺序是：左，右，根。

这三个顺序一定要熟。它们名字中的“前、中、后”，说的是根节点在遍历中的位置：根在前面，就是前序；根在中间，就是中序；根在后面，就是后序。

看下面这棵二叉树：

```
      A
    /   \
   B     C
  / \   / \
 D   E F   G
```

前序遍历先访问根 A，再访问左子树 B、D、E，最后访问右子树 C、F、G，所以结果是：

A B D E C F G

中序遍历先访问左子树 D、B、E，再访问根 A，最后访问右子树 F、C、G，所以结果是：

D B E A F C G

后序遍历先访问左子树 D、E、B，再访问右子树 F、G、C，最后访问根 A，所以结果是：

D E B F G C A

做遍历题时，不要凭图上从左到右读。每一种遍历都有固定规则。尤其是后序遍历，根节点最后访问；前序遍历，根节点最先访问。

五、层序遍历

除了前序、中序、后序，还有一种常见遍历叫层序遍历。它按照从上到下、从左到右的顺序访问节点。

对上面的二叉树，层序遍历结果是：

A B C D E F G

层序遍历通常用队列实现。先把根节点入队；每次取出队首节点并访问，再把它的左孩子、右孩子依次入队。因为队列是先进先出，所以会先处理较早发现的上层节点，再处理下一层节点。

第一轮中，层序遍历可能和队列一起出现。若题目问二叉树按层输出，或者广度优先访问树中节点，就要想到队列。

本章不展开层序遍历代码，只要求能在图上按层写出顺序，并知道它和队列关系密切。

六、由前序和中序确定二叉树

CSP-J 第一轮常考：给出一棵二叉树的前序遍历和中序遍历，要求求出后序遍历。

关键规则是：

前序遍历的第一个节点一定是根；

中序遍历中，根左边的节点属于左子树，根右边的节点属于右子树。

知道根和左右子树范围后，再分别处理左子树和右子树。

例如前序遍历为：

A B D E C F G

中序遍历为：

D B E A F C G

第一步，前序第一个是 A，所以根是 A。

在中序中找到 A：

左边是 D B E，所以左子树节点是 D、B、E；

右边是 F C G，所以右子树节点是 F、C、G。

第二步，看左子树。左子树在前序中对应 B D E，第一个是 B，所以左子树根是 B。在左子树的中序 D B E 中，B 左边是 D，右边是 E。所以 B 的左孩子是 D，右孩子是 E。

第三步，看右子树。右子树在前序中对应 C F G，第一个是 C，所以右子树根是 C。在右子树的中序 F C G 中，C 左边是 F，右边是 G。所以 C 的左孩子是 F，右孩子是 G。

因此整棵树就是：

```
      A
    /   \
   B     C
  / \   / \
 D   E F   G
```

后序遍历是左、右、根，所以结果是：

D E B F G C A

这类题的关键是不要一次性硬想整棵树。每次只找当前子树的根，再用中序把左右子树切开。对子树重复这个过程。

七、由后序和中序确定二叉树

如果给出后序遍历和中序遍历，也能确定二叉树。

后序遍历的最后一个节点一定是根；中序遍历中，根左边是左子树，根右边是右子树。

例如后序遍历为：

D E B F G C A

中序遍历为：

D B E A F C G

后序最后一个是 A，所以根是 A。中序中 A 左边是 D B E，右边是 F C G。左子树的后序是 D E B，最后一个是 B，所以左子树根是 B；右子树的后序是 F G C，最后一个是 C，所以右子树根是 C。继续分下去，就能还原整棵树。

前序加中序，或者后序加中序，通常可以确定一棵二叉树。

但只有前序和后序，一般不能唯一确定二叉树。原因是一个节点若只有一个孩子，这个孩子是左孩子还是右孩子，前序和后序可能无法区分。CSP-J 第一轮如果问“哪些遍历可以唯一确定二叉树”，要记住：中序通常是关键。

八、完全二叉树的数组表示

完全二叉树可以很方便地用数组表示。若根节点放在下标 1，那么对于下标为 i 的节点：

左孩子下标是 $2i$；

右孩子下标是 $2i+1$；

父节点下标是 $\lfloor i/2 \rfloor$。

例如节点 4 的左孩子是 8，右孩子是 9，父节点是 2。节点 9 的兄弟节点是 8，因为它们的父节点都是 4。

若题目问数组第 9 个位置的节点的兄弟节点和右子节点位置，可以这样算：

第 9 个节点的父节点是 $\lfloor 9/2 \rfloor = 4$。父节点 4 的左孩子是 8，右孩子是 9。所以 9 的兄弟节点是 8。

第 9 个节点的右孩子位置是 $2 \times 9 + 1 = 19$。

所以答案是兄弟节点 8，右子节点 19。

这类题一定要看清根节点放在哪个下标。CSP-J 第一轮常见说法是“根节点存储在数组第 1 个位置”，这时用 $2i$ 和 $2i+1$。如果根节点从下标 0 开始，公式会变成左孩子 $2i+1$、右孩子 $2i+2$。题目用哪种编号，就按哪种公式。

完全二叉树的数组表示之所以简单，是因为它的节点从上到下、从左到右连续排列，中间没有空缺。

九、真题例题

真题例题 已知二叉树的前序遍历为 `[A, B, D, E, C, F, G]`，中序遍历为 `[D, B, E, A, F, C, G]`，请问该二叉树的后序遍历结果是？

A. `[D, E, B, F, G, C, A]`
B. `[D, E, B, F, G, A, C]`
C. `[D, B, E, F, G, C, A]`
D. `[D, E, B, F, G, A, C]`

答案：A

解析：
前序遍历第一个节点是 A，所以整棵树的根是 A。

在中序遍历 `[D, B, E, A, F, C, G]` 中，A 左边是 `[D, B, E]`，右边是 `[F, C, G]`。所以左子树由 D、B、E 构成，右子树由 F、C、G 构成。

左子树在前序遍历中对应 [B, D, E]，第一个节点是 B，所以左子树根是 B。在左子树的中序 [D, B, E] 中，B 左边是 D，右边是 E，所以 B 的左孩子是 D，右孩子是 E。

右子树在前序遍历中对应 [C, F, G]，第一个节点是 C，所以右子树根是 C。在右子树的中序 [F, C, G] 中，C 左边是 F，右边是 G，所以 C 的左孩子是 F，右孩子是 G。

于是整棵树的结构为：

```
      A
    /   \
   B     C
  / \   / \
 D   E F   G
```

后序遍历顺序是左、右、根。左子树后序为 D E B，右子树后序为 F G C，最后访问根 A，所以结果是：

D E B F G C A

因此选 A。

这道题的关键是用前序确定根，用中序切分左右子树。不要直接在选项中找相似字符串。只要树结构还原正确，后序遍历就很容易写出。

十、变式题

变式题 某二叉树的前序遍历为 A B C D，中序遍历为 B A D C，它的后序遍历是哪个？

A. B D C A
B. B C D A
C. D C B A
D. B D A C

答案：A

解析：
前序第一个是 A，所以根是 A。

在中序 B A D C 中，A 左边是 B，右边是 D C。所以左子树只有 B，右子树由 D、C 构成。

右子树在前序中对应 C D，第一个是 C，所以右子树根是 C。在右子树中序 D C 中，C 左边是 D，右边为空，所以 C 的左孩子是 D。

整棵树可以表示为：

```
  A
 / \
B   C
   /
  D
```

后序遍历先访问左子树 B，再访问右子树的后序 D、C，最后访问根 A，得到：

```
B D C A
```

所以选 A。

这道变式特意安排了一个只有左孩子的节点 C。做题时不要默认每个节点都有两个孩子。中序中根左边为空或右边为空，都表示对应方向没有子树。

十一、易错点

第一，前序、中序、后序的“前、中、后”说的是根节点的位置。前序是根、左、右；中序是左、根、右；后序是左、右、根。

第二，二叉树要区分左孩子和右孩子。两个孩子交换后，通常就是不同的二叉树。画图时要把左右位置固定，不要只写成一团节点。

第三，只有前序和后序通常不能唯一确定二叉树。若要还原二叉树，常见组合是前序加中序，或者后序加中序。

第四，完全二叉树最后一层必须从左到右连续排列。不能左边缺一个节点，右边却还有节点。

第五，完全二叉树数组表示时，要看根节点下标。根在 1，左孩子是 $2i$，右孩子是 $2i+1$；根在 0，公式不同。

第六，树的高度口径要看题面。有些题规定只有根节点的树高度为 1，那就按第 1 层开始计数。不要把不同资料中的口径混用。

十二、本章要点

树由节点和边组成。根节点没有父节点，叶子节点没有子节点。一棵有 n 个节点的树有 $n-1$ 条边。

二叉树中，每个节点最多有两个孩子，并且左孩子和右孩子有区别。任意非空二叉树中，叶子节点数等于度为 2 的节点数加 1。

满二叉树每层都放满，高度为 h 时有 2^h-1 个节点。完全二叉树除最后一层外都放满，最后一层从左到右连续排列。

前序遍历是根、左、右；中序遍历是左、根、右；后序遍历是左、右、根；层序遍历是从上到下、从左到右。由前序和中序可以确定二叉树，由后序和中序也可以确定二叉树。

完全二叉树用数组存储时，如果根在下标 1，节点 i 的左孩子是 $2i$，右孩子是 $2i+1$，父节点是 $\lfloor i/2 \rfloor$。

做二叉树题时，草稿上一定要画结构。先找根，再分左右子树，再分别处理。只要结构画对，遍历序列、节点位置和形态判断都会清楚很多。

0x14 图的基本概念

图是一种用“点”和“边”描述关系的数据结构。它比树更一般：树可以看作一种特殊的图，而图中可以有环，可以不连通，也可以有方向。

CSP-J 第一轮中，图的题通常不会要求写复杂算法，而是考基础概念：无向图和有向图、顶点和边、度数、连通、路径、环、树和图的关系、邻接矩阵与邻接表。题目常以选择题出现，也可能在程序阅读中作为背景。

本章只讲图的基本概念。深度优先搜索和广度优先搜索会在搜索章节继续讲，本章只保留必要说明。

一、图由顶点和边组成

图由顶点和边组成。顶点也叫点，边表示两个顶点之间存在某种关系。

例如，有 4 个城市 A、B、C、D。如果 A 和 B 之间有道路，B 和 C 之间有道路，C 和 D 之间有道路，就可以画成一个图。城市是顶点，道路是边。

若一条边连接顶点 u 和顶点 v，可以写作 (u, v)。在无向图中，(u, v) 和 (v, u) 表示同一条边；在有向图中，(u, v) 表示从 u 指向 v 的边，和 (v, u) 不一样。

图中顶点的个数常记作 n，边的条数常记作 m。CSP-J 第一轮中，如果题目说“有 n 个顶点、m 条边”，后面常会问度数和、是否连通、至少需要多少边、删多少边能成为树等问题。

有些图允许两个顶点之间有多条边，这叫重边；也可能允许一条边从一个顶点连回自己，这叫自环。第一轮基础选择题中，如果没有特别说明，通常默认讨论简单图：没有重边，没有自环。若题面明确出现重边或自环，就要按题面处理。

二、无向图和有向图

无向图的边没有方向。若 A 和 B 之间有一条无向边，就表示 A 与 B 相连，同时也可以从 B 到 A。道路、朋友关系、同学关系等，常用无向图表示。

有向图的边有方向。若有一条从 A 指向 B 的边，就表示可以从 A 到 B，但不一定能从 B 到 A。网页链接、任务依赖、比赛胜负关系等，常用有向图表示。

例如：

```
无向边：A --- B

有向边：A ---> B
```

做题时，先判断图是否有方向。很多性质只适用于无向图，不能直接搬到有向图中。

比如“所有顶点度数之和等于边数的两倍”是无向图中的结论。对有向图，更常用的是入度之和等于边数，出度之和也等于边数。把有向图和无向图混在一起，是图论选择题中常见错误。

三、度数、入度和出度

在无向图中，一个顶点连接了多少条边，它的度数就是多少。

例如顶点 A 分别与 B、C、D 相连，那么 A 的度数是 3。

一条无向边连接两个端点，会给这两个端点的度数各贡献 1。因此，在无向图中，所有顶点的度数之和等于边数的两倍，也就是 $2m$。

例如一个无向图有 5 条边，那么所有顶点的度数加起来一定是 10。不管这些边怎样连接，只要是普通无向边，每条边都会被两个端点各数一次。

这是 CSP-J 第一轮的高频结论。

在有向图中，边有方向。指向某个顶点的边数叫入度，从某个顶点指出去的边数叫出度。

例如有向边 A 到 B，会让 A 的出度加 1，让 B 的入度加 1。

在有向图中，所有顶点入度之和等于边数，所有顶点出度之和也等于边数。因为每条有向边只有一个起点和一个终点，给一个顶点贡献出度，给另一个顶点贡献入度。

注意这三个结论不要混用：

无向图中，度数之和是 $2m$；

有向图中，入度之和是 m；

有向图中，出度之和也是 m。

如果题目只说“度数之和等于边数两倍”，要确认它说的是无向图。

四、路径和环

路径表示从一个顶点出发，沿着边走到另一个顶点的过程。

例如无向图中有边 A-B、B-C、C-D，那么从 A 可以沿 A、B、C、D 走到 D，这就是一条从 A 到 D 的路径。

路径的长度通常指经过的边数。上面的路径经过 3 条边，所以长度为 3。

如果从一个顶点出发，沿若干条边走一圈又回到原来的顶点，并且中间没有随意重复绕路，这样的结构常叫环。

例如 A-B、B-C、C-A 形成一个三角形，这就是一个环。

树中没有环。图中可以有环。这个区别非常重要。很多图题会问“删掉多少条边才能成为一棵树”，本质上就是要删掉多余的环，同时保持连通。

在有向图中，路径和环还要考虑方向。若边是 A 到 B，再 B 到 C，那么可以从 A 到 C；但如果只有 B 到 A，就不能沿这条边从 A 到 B。做有向图题时，不能把箭头忽略。

五、连通图和连通分量

在无向图中，如果任意两个顶点之间都能通过若干条边互相到达，就说这个图是连通图。

例如：

```
A --- B --- C --- D
```

这个图是连通的，因为任意两个点之间都能沿边走到。

如果图分成几块，块与块之间没有边相连，就不是连通图。例如：

```
A --- B      C --- D
```

这里 A、B 在一块，C、D 在另一块，从 A 无法走到 C，所以这个图不连通。

无向图中，每一个内部连通、并且不能再向外扩大的部分，叫一个连通分量。上面的图有两个连通分量：一个是 A、B，另一个是 C、D。

对于一个有 n 个顶点的无向图，要保证它连通，至少需要 $n-1$ 条边。这些边可以把所有顶点连成一棵树。如果边数少于 $n-1$，就一定无法把 n 个顶点全部连起来。

但有 $n-1$ 条边并不一定连通。比如 4 个点有 3 条边，如果三条边都集中在 A、B、C 之间，D 仍然孤立，那么图不连通。判断连通不能只看边数，还要看边是否把所有顶点接到一起。

六、树是特殊的图

树可以看作一种特殊的无向图。它有两个重要特点：连通，没有环。

如果一棵树有 n 个顶点，那么它一定有 $n-1$ 条边。这是树的基本性质。

反过来，一个无向图如果连通并且有 $n-1$ 条边，那么它一定是一棵树。一个无向图如果连通但边数超过 $n-1$，就一定存在多余的边，这些多余的边会形成环。

因此，对于一个有 n 个顶点、m 条边的无向连通图，要删掉多少条边才能成为一棵树？

成为树后必须还剩 $n-1$ 条边。原来有 m 条边，所以要删：

$m-(n-1)=m-n+1$ 条边。

这个式子在第一轮中很常见。注意条件里必须有“连通”。如果原图不连通，只删边不能让它变成一棵覆盖所有顶点的树，因为删边不会把不同连通块接起来。

例如一个无向连通图有 6 个顶点、9 条边。若要删边成为一棵树，最后应剩 $6-1=5$ 条边，所以要删 $9-5=4$ 条边，也就是 $9-6+1=4$ 条。

树的性质也可以帮助判断边数。如果一个连通无向图有 n 个点、n 条边，那么它比树多 1 条边，至少有一个环。删掉环上的一条边后，仍然可以保持连通。

七、完全图和边数上限

在简单无向图中，如果任意两个不同顶点之间都有一条边，这样的图叫完全图。

n 个顶点的简单无向完全图有 $C(n,2)$ 条边。因为每一条边对应一对不同顶点，而从 n 个顶点中选 2 个顶点的方法数就是 $C(n,2)$。

例如 4 个顶点的简单无向完全图有 $C(4,2)=6$ 条边。

可以列成：

A-B，A-C，A-D，B-C，B-D，C-D。

若是简单有向图，并且任意两个不同顶点之间两个方向都可以分别存在一条边，那么最多有 $n(n-1)$ 条有向边。因为每个顶点都可以向其他 $n-1$ 个顶点连边。

第一轮中若问“最多有多少条边”，要先判断是无向图还是有向图，还要看是否允许重边和自环。若是简单无向图，通常是 $C(n,2)$；若是简单有向图且不允许自环，通常是 $n(n-1)$。

不要把 $C(n,2)$ 和 $n(n-1)$ 混用。无向边没有方向，一对点只算一条；有向边有方向，一对点可能有两条方向相反的边。

八、邻接矩阵和邻接表

图在程序中需要存储。CSP-J 第一轮常见两种存图方式：邻接矩阵和邻接表。

邻接矩阵用一个二维数组表示点与点之间是否有边。设有 n 个顶点，可以用 `g[i][j]` 表示顶点 i 到顶点 j 是否有边。

无向图中，如果 i 和 j 之间有边，那么通常 `g[i][j]` 和 `g[j][i]` 都要标记。有向图中，若只有从 i 到 j 的边，只标记 `g[i][j]`，不一定标记 `g[j][i]`。

邻接矩阵的优点是判断两个点之间是否有边很方便，只要查一个位置。缺点是占用空间是 $O(n^2)$，当点很多、边很少时比较浪费。

邻接表则为每个顶点保存它能直接到达的顶点列表。例如顶点 A 连向 B、C，就在 A 的列表中记录 B、C。邻接表适合边数较少的图，空间通常是 $O(n+m)$。

第一轮中，不一定要求写出完整邻接表代码，但要能判断两种存法的特点：

邻接矩阵适合快速判断两点是否相连，空间是 $O(n^2)$；

邻接表适合遍历一个点连出去的边，空间通常是 $O(n+m)$。

如果题目给出一个二维数组并用 `g[u][v] = 1` 表示有边，通常是在使用邻接矩阵。如果给出很多 `next`、`to` 或者每个点后面跟着若干相邻点，可能是在使用邻接表。

九、遍历图的基本想法

遍历图，就是从某个顶点出发，按照一定规则访问能到达的顶点。

最常见的两种遍历是深度优先搜索和广度优先搜索。

深度优先搜索通常沿着一条路尽量往深走，走不下去再回退。它常用递归或栈实现。

广度优先搜索通常按距离从近到远一层一层访问。它常用队列实现。

本章不展开 DFS 和 BFS 的完整算法，只强调它们与图概念的关系：遍历图时必须避免重复访问同一个顶点。因为图中可能有环，如果不记录已经访问过的点，程序可能在环中反复走。

例如图中有 A-B、B-C、C-A。若从 A 走到 B，再走到 C，又能回到 A。如果没有 `visited` 标记，就可能不断重复。

所以图的遍历程序中常会出现 `vis[u]`、`visited[u]` 之类数组，表示顶点 u 是否已经访问过。阅读图程序时，只要看到这种数组，就要想到它通常用于防止重复访问。

十、真题例题

真题例题 在无向图中，所有顶点的度数之和等于什么？

- **A.** 图的边数
- **B.** 图的边数的两倍
- **C.** 图的顶点数
- **D.** 图的顶点数的两倍

答案：B

解析：
在无向图中，一条边连接两个端点。计算所有顶点度数之和时，这条边会在一个端点处贡献 1，在另一个端点处也贡献 1。所以每条边总共贡献 2。

若图中有 m 条边，所有顶点的度数之和就是 $2m$。因此答案是“图的边数的两倍”。

可以用小图检查。例如三角形有 3 个顶点、3 条边，每个顶点度数都是 2，度数之和是 6，正好是边数 3 的两倍。

这道题只适用于无向图。如果是有向图，应分别讨论入度和出度：入度之和等于边数，出度之和也等于边数。

十一、变式题

变式题 对于有 n 个顶点、m 条边的无向连通图，若 $m > n$，需要删掉多少条边才能使其成为一棵树？

- **A.** $n-1$
- **B.** $m-n$
- **C.** $m-n-1$
- **D.** $m-n+1$

答案：D

解析：
一棵有 n 个顶点的树有 $n-1$ 条边。题目中的图是无向连通图，要通过删边变成一棵树，就要让最后剩下的边数为 $n-1$。

原来有 m 条边，最后保留 $n-1$ 条边，所以需要删：

$m-(n-1)=m-n+1$。

因此答案是 D。

这里的“连通”条件很重要。因为原图连通，所以可以通过删掉多余边去掉环，同时仍保持所有顶点连在一起。如果原图不连通，只靠删边不能把不同连通块连起来，也就不能保证得到一棵包含所有顶点的树。

十二、易错点

第一，无向图和有向图要分清。无向边没有方向，有向边有方向。无向图中度数之和是 $2m$；有向图中入度之和是 m，出度之和也是 m。

第二，连通图和树不能混淆。树一定连通且无环；连通图可以有环。一个有 n 个顶点的连通无向图如果有 $n-1$ 条边，就是树；如果边数更多，就有多余边。

第三，边数达到 $n-1$ 不一定连通。边数少于 $n-1$ 一定不可能连通，但边数等于 $n-1$ 还要看这些边是否把所有顶点连起来。

第四，完全图的边数要看是否有方向。简单无向完全图有 $C(n,2)$ 条边，简单有向图最多有 $n(n-1)$ 条边。

第五，邻接矩阵和邻接表用途不同。邻接矩阵查两点是否相连方便，但空间是 $O(n^2)$；邻接表遍历相邻点方便，空间通常是 $O(n+m)$。

第六，遍历图要记录访问状态。图可能有环，如果不记录哪些顶点已经访问过，程序可能重复走同一批点。

十三、本章要点

图由顶点和边组成。无向图的边没有方向，有向图的边有方向。顶点数常记作 n，边数常记作 m。

无向图中，一个顶点连接多少条边，它的度数就是多少。所有顶点度数之和等于 $2m$。有向图中，每条边贡献一个出度和一个入度，所以入度之和、出度之和都等于 m。

路径表示沿边从一个顶点走到另一个顶点。环表示从某个顶点出发又回到它本身的闭合结构。无向图中，任意两个顶点都能互相到达时，图是连通的。

树是连通且无环的无向图。有 n 个顶点的树有 $n-1$ 条边。一个有 n 个顶点、m 条边的无向连通图，要删成树，需要删 $m-n+1$ 条边。

简单无向完全图有 $C(n,2)$ 条边。简单有向图在不允许自环时最多有 $n(n-1)$ 条边。邻接矩阵空间是 $O(n^2)$，邻接表空间通常是 $O(n+m)$。

做图的基础题时，先确定图是否有方向，再判断它是否连通、是否有环、边数和度数之间有什么关系。只要这些基本概念不混，第一轮中的图论选择题通常能稳定得分。

0x15 哈夫曼编码

哈夫曼编码是一种常见的数据压缩编码方法。CSP-J 第一轮中，它通常以选择题出现：给出若干字符的出现频率，要求判断哪个字符编码更短，某个字符的编码长度是多少，或者从几个选项中选出一组可能的哈夫曼编码。

这一章不讲复杂证明，也不要求写哈夫曼树程序。第一轮需要掌握的是：什么是带权路径长度，哈夫曼树怎样构造，哈夫曼编码为什么是前缀编码，怎样在纸面上根据频率求出编码长度。

一、为什么需要变长编码

如果每个字符都用相同长度的编码表示，叫定长编码。例如有 8 个不同字符，至少需要 3 位二进制编码，因为 $2^3 = 8$。

但在真实文本中，不同字符出现的频率往往不同。有些字符出现很多，有些字符出现很少。若给高频字符较短编码，给低频字符较长编码，整体编码长度可能减少。这就是变长编码的基本想法。

例如有两个字符 A 和 B，A 出现很多，B 出现很少。如果 A 用 `0` 表示，B 用 `111` 表示，虽然 B 的编码更长，但由于 B 出现少，总长度可能仍然更短。

哈夫曼编码就是一种构造变长编码的方法。它的目标是让所有字符按出现频率编码后，总长度尽量短。这里的“出现频率”也可以理解成权值。

要注意，哈夫曼编码不是简单地把频率最高的字符随便写成 `0`，把频率第二高的字符写成 `1`。编码之间还必须能唯一解码，这就需要前缀编码的规则。

二、前缀编码

前缀编码要求：任何一个字符的编码，都不能是另一个字符编码的前缀。

例如下面这组编码不是前缀编码：

```
A: 0
B: 01
C: 10
```

因为 A 的编码 `0` 是 B 的编码 `01` 的前缀。若收到一串编码 `01`，就会产生歧义：它可能先读出 A 的 `0`，也可能整体读作 B 的 `01`。

下面这组编码是前缀编码：

```
A: 0
B: 10
C: 11
```

没有任何一个编码是另一个编码的前缀。收到编码串时，可以从左到右唯一地拆分。

哈夫曼编码是一种前缀编码。它通常用一棵二叉树表示：每个字符放在叶子节点上，从根到叶子的路径决定编码。向左走记作 `0`，向右走记作 `1`；也可以反过来，左边记作 `1`，右边记作 `0`。只要整棵树规则一致，就能得到一组编码。

为什么叶子节点编码不会互为前缀？因为一个叶子节点不可能是另一个叶子节点的祖先。若一个编码是另一个编码的前缀，就意味着一个字符对应的节点还可以继续向下走到另一个字符，这与“字符放在叶子节点”矛盾。

因此，判断一组选项是否可能是哈夫曼编码时，第一步可以先检查它是不是前缀编码。若某个编码是另一个编码的前缀，这组选项一定不对。

三、带权路径长度

哈夫曼树的目标，是让带权路径长度尽量小。

树中从根到某个叶子节点经过的边数，叫这个叶子节点的路径长度。若这个叶子节点对应字符的权值为 w，路径长度为 d，它对总长度的贡献就是 $w \times d$。

所有叶子节点的 $w \times d$ 相加，就是整棵树的带权路径长度，常记作 WPL。

例如有三个字符 A、B、C，权值分别是 $5, 2, 1$。如果它们的编码长度分别是 $1, 2, 2$，那么总带权路径长度是 $5 \times 1 + 2 \times 2 + 1 \times 2 = 11$。

这里的编码长度就是路径长度。编码越短，路径越短；字符权值越大，短路径带来的节省越明显。

哈夫曼树就是在所有可能的前缀编码树中，使 WPL 最小的二叉树。CSP-J 第一轮不要求证明它为什么最优，但要求会按规则构造它。

四、哈夫曼树的构造规则

构造哈夫曼树的规则很简单：每次选出当前权值最小的两个节点，合并成一个新节点，新节点权值等于这两个节点权值之和；再把新节点放回集合中，继续合并，直到只剩一个根节点。

这个过程本质上是一种贪心策略：每一步都先处理当前最小的两个权值。

例如给出权值 $5, 9, 12, 13, 16, 45$。

第一次选最小的 5 和 9，合并为 14。剩下：

```
12, 13, 14, 16, 45
```

第二次选最小的 12 和 13，合并为 25。剩下：

```
14, 16, 25, 45
```

第三次选最小的 14 和 16，合并为 30。剩下：

```
25, 30, 45
```

第四次选最小的 25 和 30，合并为 55。剩下：

```
45, 55
```

第五次选 45 和 55，合并为 100。构造结束。

从这个合并过程可以看出，45 只在最后一次合并时接到根下面，所以它的编码长度最短，为 1。5 和 9 先合并成 14，之后 14 又参与后续合并，它们一路向上经过多层，所以编码长度较长。

哈夫曼树中，权值较小的字符通常会处在较深的位置，编码较长；权值较大的字符通常会处在较浅的位置，编码较短。这里说“通常”，是因为遇到权值相同或接近时，左右方向和同权合并顺序可能不同，但总的 WPL 仍然最小。

五、编码不唯一，长度更关键

哈夫曼编码不是唯一的。原因至少有两个。

第一，左右孩子可以交换。若一棵哈夫曼树中左边记作 `0`、右边记作 `1`，把某个节点的左右子树交换后，也能得到一组合法哈夫曼编码，只是某些编码中的 `0` 和 `1` 会变化。

第二，若有相同权值，选择哪两个先合并可能有多种合法方式。只要每一步都符合“取当前最小两个”的规则，可能得到不同形状但同样最优的树。

因此，做选择题时，不要只记某个字符一定是 `0` 或一定是 `1111`。更可靠的判断是：

先根据合并过程判断每个字符的编码长度；

再检查选项是否满足这些长度；

最后检查是否为前缀编码。

例如权值 $5, 9, 12, 13, 16, 45$ 的构造中，45 的编码长度为 1，5 和 9 的编码长度为 4，$12, 13, 16$ 的编码长度为 3。至于 45 的编码是 `0` 还是 `1`，取决于最后根节点左右怎么标。只要整组编码符合树结构，都可能成立。

这一点很重要。若只背“频率最高的字符编码为 0”，遇到左右交换就会误判。

六、用合并过程求编码长度

有些题只问某个字符的编码长度，不要求写出完整编码。此时可以直接从合并过程看它参与了多少次合并。

每当某个节点所在的合并块再次被合并，块中所有字符的编码长度都会增加 1。最终，一个字符从叶子到根经过几次合并，它的编码长度就是几。

例如字符权值为：

```
a:10, b:15, c:30, d:16, e:29
```

按哈夫曼规则合并。

第一次，10 和 15 合并成 25。此时 a、b 先进入一个块。

剩下：

```
16, 25, 29, 30
```

第二次，16 和 25 合并成 41。此时 d 进入一个块，a、b 所在块又向上一层。

剩下：

```
29, 30, 41
```

第三次，29 和 30 合并成 59。此时 e、c 进入一个块。

剩下：

```
41, 59
```

第四次，41 和 59 合并成根。

现在看 d。它在第二次合并进入 41，又在第四次合并接到根，所以它经过两层，编码长度为 2。

这类题不需要把整棵树画得很复杂。可以在草稿上记录“某个字符所在的块被合并了几次”。问编码长度时，数层数即可。

七、用合并权值求 WPL

哈夫曼树的 WPL 可以通过最终树计算，也可以直接用每次合并出来的新权值相加。

例如权值 $5, 9, 12, 13, 16, 45$ 的合并过程是：

$5 + 9 = 14$；

$12 + 13 = 25$；

$14 + 16 = 30$；

$25 + 30 = 55$；

$45 + 55 = 100$。

那么 WPL 是 $14 + 25 + 30 + 55 + 100 = 224$。

为什么可以这样算？每次合并时，被合并的两个块中所有字符的深度都会增加 1。这一次增加的总贡献，正好等于这两个块的权值之和。因此把每次合并产生的新权值加起来，就是最终所有叶子的 $w \times d$ 总和。

CSP-J 第一轮中，直接考 WPL 的题不一定多，但这个方法很实用。它比画完整树后再逐个计算路径长度更快，也更不容易漏掉某个叶子。

若题目只问编码长度，不需要算 WPL；若题目问“带权路径长度最小是多少”，就可以用合并权值求和。

八、真题例题

真题例题 假设有一组字符 `{a,b,c,d,e,f}`，对应的频率分别为 $5\%, 9\%, 12\%, 13\%, 16\%, 45\%$。请问以下哪个选项是字符 `a,b,c,d,e,f` 分别对应的一组哈夫曼编码？

A. `1111, 1110, 101, 100, 110, 0`

B. `1010, 1001, 1000, 011, 010, 00`

C. `000, 001, 010, 011, 10, 11`

D. `1010, 1011, 110, 111, 00, 01`

答案：A

解析：

先只看频率，也就是权值：

```
a:5, b:9, c:12, d:13, e:16, f:45
```

按哈夫曼规则合并：

5 和 9 合并为 14；

12 和 13 合并为 25；

14 和 16 合并为 30；

25 和 30 合并为 55；

45 和 55 合并为 100。

由合并层数可知，`f` 的权值最大 45，只在最后一次合并中接到根下面，所以编码长度应为 1。

`a` 和 `b` 先合并成 14，然后 14 又与 16 合并成 30，再与 25 合并成 55，最后 55 与 45 合并成根。它们经历 4 层，所以 `a` 和 `b` 的编码长度都是 4。

`c`、`d`、`e` 的编码长度都是 3。

所以编码长度模式应为：

```
a:4, b:4, c:3, d:3, e:3, f:1
```

检查选项 A：

```
a:1111 长度 4
b:1110 长度 4
c:101  长度 3
d:100  长度 3
e:110  长度 3
f:0    长度 1
```

长度完全符合。同时，`0` 不是其他编码的前缀，因为其他编码都以 `1` 开头；以 `1` 开头的几个编码之间也没有谁是另一个的前缀。因此 A 可以是一组哈夫曼编码。

B 中 `f` 的编码 `00` 长度为 2，不符合最大频率字符应在这一构造中长度为 1 的结果。C 中所有低频字符长度都为 3，不符合 `a`、`b` 应为长度 4。D 中 `e` 和 `f` 长度都是 2，也不符合合并过程。答案选 A。

这道题的关键不是画出唯一的编码，而是先求出每个字符的编码长度，再检查前缀编码规则。

九、变式题

变式题 字母表 `{a,b,c,d,e}` 在字符串中出现的频率分别为 $10\%, 15\%, 30\%, 16\%, 29\%$。若使用哈夫曼编码方式进行不定长二进制编码，字符 `d` 的编码长度为多少？

- **A.** 1
- **B.** 2
- **C.** 2 或 3
- **D.** 3

答案：B

解析：
只需要求编码长度，不需要写出完整编码。

权值为：

```
a:10, b:15, c:30, d:16, e:29
```

按哈夫曼规则合并：

10 和 15 合并为 25；

此时剩下 $16, 25, 29, 30$。

接着 16 和 25 合并为 41；

此时剩下 $29, 30, 41$。

再把 29 和 30 合并为 59；

最后把 41 和 59 合并为根。

字符 `d` 的权值是 16。它在第二次合并中与 25 合并成 41，又在最后一次合并中与 59 合并成根。因此 `d` 从叶子到根经过 2 条边，编码长度为 2。

所以选 B。

这道题若只凭“频率越小编码越长”粗略判断，可能会误选。d 的频率 16% 不算最大，但它较早和一个合并块组成 41，最终深度为 2。哈夫曼编码长度要按合并过程确定，不能只按频率排名直接写答案。

十、易错点

第一，哈夫曼编码不唯一。左右子树交换会改变具体的 0 和 1，但不改变编码长度和 WPL。选择题中要更多关注长度和前缀关系。

第二，频率最高的字符通常编码较短，但不要把它绝对化为“最高频一定是某个具体编码”。具体编码取决于左右方向。

第三，每次合并必须选当前最小的两个权值。不是先把相邻的两个合并，也不是先把看起来接近的两个随便合并。若草稿中某一步没有重新排序，很容易错。

第四，问 WPL 时，可以把每次合并得到的新权值相加。问编码长度时，要看该字符所在的块被合并了几次。两个问题相关，但不要混淆。

第五，前缀编码要求任何一个编码都不能是另一个编码的开头。若出现 0 和 01 同时作为两个字符编码，这组编码一定不是合法前缀编码，也不可能是哈夫曼编码。

第六，所有字符应当对应叶子节点。不要把内部节点当作某个字符的编码位置。只有叶子节点对应字符，才能保证前缀编码性质。

十一、本章要点

哈夫曼编码是一种变长前缀编码。高频字符通常使用较短编码，低频字符通常使用较长编码，从而减少总编码长度。

哈夫曼树的构造规则是：每次选出当前权值最小的两个节点合并，新节点权值为二者之和，再放回集合中继续合并。这个过程体现的是贪心策略。

编码长度等于字符所在叶子节点到根的路径长度。字符所在的合并块每向上合并一次，它的编码长度就增加 1。如果只问某个字符编码长度，可以直接数它参与了几层合并。

带权路径长度 WPL 是所有叶子节点的 权值 × 路径长度 之和。用哈夫曼构造时，WPL 也可以通过每次合并得到的新权值相加求出。

做哈夫曼编码选择题时，按三个步骤检查：先按权值构造合并过程；再确定每个字符的编码长度；最后检查选项是否满足长度要求和前缀编码规则。不要只盯着某个具体编码串，因为哈夫曼编码的具体 0、1 分配并不唯一。

0x16 排序基础

排序是把一组数据按照某种规则重新排列。CSP-J 第一轮中，排序常以选择题和程序阅读形式出现：判断某个排序算法的过程，判断排序是否稳定，判断时间复杂度，或者读懂一段简单排序代码。

本章不讲复杂排序优化，也不要求背大量模板。第一轮需要掌握的是：排序规则是什么，什么叫稳定，冒泡排序、选择排序、插入排序的过程怎样走，归并排序和快速排序的大致特点是什么。

排序题看起来熟悉，但很容易因为“稳定性”“比较次数”“交换位置”这些细节失分。读排序程序时，不要只看算法名字，要看代码到底怎样比较、怎样交换、怎样移动。

一、什么是排序

排序就是把若干元素按某个关键字排列成指定顺序。

如果从小到大排列，叫升序排序。例如：

```
原序列：5 1 4 2 3
升序后：1 2 3 4 5
```

如果从大到小排列，叫降序排序。例如：

```
原序列：5 1 4 2 3
降序后：5 4 3 2 1
```

有些排序题中的元素不只是一个数字，而是包含多个信息。例如学生记录中有姓名和分数，要求按分数排序。此时“分数”就是关键字。若关键字相同，题目可能要求保持原顺序，也可能要求再按另一个关键字排序。

CSP-J 第一轮中，多数排序选择题使用整数序列，重点考算法过程和稳定性。遇到排序题时，先看三件事：

排序方向是升序还是降序；

比较的是哪个关键字；

关键字相等时是否要求保持原顺序。

如果题面没有特别说明，基础排序通常默认按数值从小到大讨论。但程序阅读题要按代码中的比较符号判断，`>` 和 `<` 的方向不同，结果也不同。

二、稳定性

排序稳定性是第一轮常考概念。

如果两个元素的关键字相等，排序后它们的相对先后顺序没有改变，就说这个排序算法是稳定的。若可能改变相对顺序，就说它是不稳定的。

例如有三个学生记录：

```
甲 90
乙 80
丙 90
```

按分数从小到大排序后，乙应该在最前面。甲和丙分数相同，若排序后仍然是甲在丙前面，说明这次排序保持了相等关键字的相对顺序。若变成丙在甲前面，就破坏了稳定性。

稳定性只讨论“关键字相等”的元素。如果两个元素关键字不同，排序后先后变化是正常的，不涉及稳定性。

很多学生会误以为“排序结果正确”就一定稳定。不是这样。对于只看数字的序列，两个相同的数字看起来没有区别；但如果它们背后还有姓名、编号或原始位置，稳定性就能体现出来。

为了在纸面上看清稳定性，可以把相等元素加上下标。例如原序列中有两个值为 2 的元素，靠前的记为 2_a，靠后的记为 2_b：

原序列：$2_a, 1, 2_b$。

稳定排序后应为：$1, 2_a, 2_b$。

如果某种排序可能得到：$1, 2_b, 2_a$，就说明它改变了两个相等关键字元素的相对顺序，是不稳定的。

CSP-J 第一轮常考结论是：冒泡排序的常见实现是稳定的，插入排序的常见实现是稳定的，归并排序的常见实现可以是稳定的，简单选择排序的常见实现是不稳定的。

这里说“常见实现”，是因为稳定性也和具体写法有关。例如冒泡排序若在相等时也交换，就可能破坏稳定性。考试中若问“常见实现”，按常见写法判断；若给出代码，就必须按代码判断。

三、冒泡排序

冒泡排序的思想是：反复比较相邻元素，如果顺序不对，就交换它们。每一轮结束后，一个较大的元素会被交换到靠后位置，就像气泡不断向上冒。

以升序排序为例：

```
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]);
        }
    }
}
```

内层循环比较相邻的 `a[j]` 和 `a[j + 1]`。若前者大于后者，就交换。第一轮结束后，最大值会到达最后一个位置；第二轮结束后，次大值会到达倒数第二个位置。

示例：对 `5 1 4 2` 做升序冒泡排序。

第一轮：

比较 5 和 1，交换，得到 `1 5 4 2`；

比较 5 和 4，交换，得到 `1 4 5 2`；

比较 5 和 2，交换，得到 `1 4 2 5`。

第一轮后，最大值 5 已经在最后。

第二轮：

比较 1 和 4，不交换；

比较 4 和 2，交换，得到 `1 2 4 5`。

第三轮：

比较 1 和 2，不交换。排序完成。

冒泡排序的常见实现是稳定的。原因是它只在 `a[j] > a[j + 1]` 时交换，若两个元素相等，不会交换它们的相对顺序。

如果代码写成 `a[j] >= a[j + 1]` 时也交换，那么相等元素可能被交换，稳定性就会被破坏。阅读程序时一定要看比较符号，不要只看算法名字。

冒泡排序有两层循环，常见时间复杂度是 $O(n^2)$。它只需要少量临时变量，额外空间复杂度是 $O(1)$。

四、选择排序

简单选择排序的思想是：每一轮从还没有排好的部分中选出最小元素，放到当前应该放的位置。

升序排序的常见写法如下：

```
for (int i = 1; i <= n; i++) {
    int pos = i;
    for (int j = i + 1; j <= n; j++) {
        if (a[j] < a[pos]) {
            pos = j;
        }
    }
    swap(a[i], a[pos]);
}
```

第 i 轮中，前 $i-1$ 个位置已经排好。程序在 i 到 n 中找到最小元素的位置 `pos`，然后把它和 `a[i]` 交换。这样第 i 个位置就放好了。

示例：对 `5 1 4 2` 做升序选择排序。

第一轮，从 `5 1 4 2` 中选最小值 1，与第一个位置 5 交换，得到 `1 5 4 2`。

第二轮，在剩下的 `5 4 2` 中选最小值 2，与第二个位置 5 交换，得到 `1 2 4 5`。

第三轮，在剩下的 `4 5` 中选最小值 4，不需要改变。

排序完成。

简单选择排序的常见实现是不稳定的。原因是它可能把后面的较小元素直接交换到前面，从而把某些相等元素的相对顺序打乱。

例如原序列是 $2_a, 2_b, 1$。按升序选择排序时，第一轮会选出最小值 1，把它和第一个位置的 2_a 交换，得到 $1, 2_b, 2_a$。原来 2_a 在 2_b 前面，排序后 2_b 在 2_a 前面。两个关键字都为 2 的元素相对顺序改变了，所以简单选择排序不稳定。

选择排序的常见时间复杂度是 $O(n^2)$。因为每一轮都要在未排序部分中扫描寻找最小值。额外空间复杂度是 $O(1)$。

选择排序的交换次数较少，通常每轮最多交换一次。但比较次数仍然是平方级。选择题中若只问时间复杂度，不要因为“交换少”就选 $O(n)$。

五、插入排序

插入排序的思想是：把前面一段看作已经排好，每次取下一个元素，把它插入到前面有序部分的合适位置。

升序排序的常见写法如下：

```
for (int i = 2; i <= n; i++) {
    int x = a[i];
    int j = i - 1;
    while (j >= 1 && a[j] > x) {
        a[j + 1] = a[j];
        j--;
    }
    a[j + 1] = x;
}
```

第 i 轮开始时，`a[1]` 到 `a[i - 1]` 已经有序。现在把 `a[i]` 保存到 `x`，从右向左扫描有序部分。比 `x` 大的元素都向右移动一格，最后把 `x` 放到空出来的位置。

示例：对 `5 1 4 2` 做升序插入排序。

开始时，把第一个元素 `5` 看作已经有序。

处理 1：5 比 1 大，5 右移，把 1 放到最前，得到 `1 5 4 2`。

处理 4：5 比 4 大，5 右移；1 不比 4 大，把 4 放到 1 后面，得到 `1 4 5 2`。

处理 2：5、4 都比 2 大，依次右移；1 不比 2 大，把 2 放到 1 后面，得到 `1 2 4 5`。

插入排序的常见实现是稳定的。原因是移动条件通常是 `a[j] > x`。当 `a[j] == x` 时，不会把前面的相等元素移到后面，所以相等关键字的相对顺序保持不变。

若写成 `a[j] >= x`，相等元素也会右移，就可能破坏稳定性。和冒泡排序一样，稳定性要看相等时是否交换或移动。

插入排序的最坏时间复杂度是 $O(n^2)$，例如原序列完全逆序时，每个元素都要向前移动很多位置。若原序列已经有序，常见实现中内层 `while` 很快停止，时间接近 $O(n)$。第一轮若问“最坏情况”或“一般常见复杂度”，通常按 $O(n^2)$ 判断。

六、归并排序

归并排序是一种分治排序。它的基本思想是：先把序列分成两半，分别排序，再把两个有序序列合并成一个有序序列。

例如要排序：

```
5 1 4 2
```

可以先分成两半：

```
5 1    4 2
```

分别排序成：

```
1 5    2 4
```

再合并两个有序序列。比较 1 和 2，先放 1；比较 5 和 2，放 2；比较 5 和 4，放 4；最后放 5。得到：

```
1 2 4 5
```

归并排序的时间复杂度是 $O(n \log n)$。因为它把序列不断二分，层数是 $O(\log n)$；每一层合并所有元素，总工作量是 $O(n)$。合起来就是 $O(n \log n)$。

归并排序的常见实现可以保持稳定。合并两个有序序列时，如果左右两边当前元素相等，先取左边原来更靠前的元素，就能保持稳定性。

不过归并排序通常需要额外数组保存合并结果，空间复杂度常见为 $O(n)$。这一点和冒泡、选择、插入不同。

CSP-J 第一轮中，归并排序更多用于考复杂度和稳定性，不太会要求完整手写代码。只要知道它是分治思想、常见时间复杂度是 $O(n \log n)$、可以稳定，就足够应对基础题。

七、快速排序

快速排序也是一种分治排序。它的基本思想是：选择一个基准值，把比它小的元素放到一边，比它大的元素放到另一边，然后分别处理两边。

例如以某个元素为基准，把序列分成“小于基准”“基准”“大于基准”三部分。若左右两部分再分别排好，整个序列就有序。

快速排序平均情况下时间复杂度常见为 $O(n \log n)$，但在某些不利情况下可能退化到 $O(n^2)$。例如每次选到的基准都使划分很不均衡，就会出现较差情况。

快速排序的常见实现通常不稳定。因为划分过程中可能交换相距较远的元素，导致相等关键字的相对顺序改变。

CSP-J 第一轮对快速排序一般考概念，不会要求深入掌握划分细节。需要记住的是：快速排序常用分治思想，平均效率较好，但最坏情况可能是 $O(n^2)$，常见实现不稳定。

八、计数排序的基本想法

有些排序不是通过比较元素大小完成的。计数排序就是其中一种。

如果所有元素都是范围较小的整数，可以统计每个值出现了多少次，再按顺序输出。

例如序列：

```
3 1 2 3 1
```

统计次数：

```
1 出现 2 次
2 出现 1 次
3 出现 2 次
```

按升序输出就是：

```
1 1 2 3 3
```

计数排序的时间复杂度通常与数据个数和数值范围有关。如果有 n 个数，值域大小为 K，常见复杂度是 $O(n+K)$。

但计数排序只有在值域不大时才合适。若数据范围很大，例如数值可能达到 10^9，直接开一个巨大计数数组就不合适。

CSP-J 第一轮中，计数排序常作为概念出现。看到“数值范围较小”“统计每个数出现次数”，可以想到计数排序。它和冒泡、选择、插入、归并、快速这些比较排序不同。

九、排序稳定性辨析

排序稳定性是 CSP-J 第一轮高频考点。下面把常见结论用文字整理一次。

冒泡排序的常见实现稳定。它只交换相邻逆序元素，若相等时不交换，就不会改变相等元素相对顺序。

插入排序的常见实现稳定。它把较大的元素右移，若相等时不右移，就会把新元素插到相等元素后面。

归并排序的常见实现可以稳定。合并时相等元素优先取左边，就能保持原顺序。

简单选择排序的常见实现不稳定。因为它每轮可能把后面的最小元素和前面的元素直接交换，跨过相等元素，打乱相对顺序。

快速排序的常见实现通常不稳定。因为划分时可能交换较远位置的元素，相等元素顺序可能改变。

注意，“稳定”不是排序正确性的必要条件。不稳定排序也能把数据排成正确顺序，只是相等关键字的相对顺序可能改变。只有当元素除了关键字之外还带有其他信息时，稳定性才会体现出来。

十、真题例题

真题例题 以下排序算法的常见实现中，哪个选项的说法是错误的？

- **A.** 冒泡排序算法是稳定的
- **B.** 简单选择排序是稳定的
- **C.** 简单插入排序是稳定的
- **D.** 归并排序算法是稳定的

答案：B

解析：
冒泡排序常见实现中，只有前一个元素大于后一个元素时才交换。相等元素不会交换，所以稳定。A 正确。

简单插入排序常见实现中，把比当前元素大的元素后移。遇到相等元素时不继续移动，当前元素会插到相等元素后面，所以稳定。C 正确。

归并排序在合并两个有序序列时，如果遇到相等元素，先取左边序列中的元素，就可以保持稳定。常见稳定实现下，D 正确。

简单选择排序常见实现中，每一轮从未排序部分选出最小元素，再和当前位置交换。这种交换可能跨过相等元素，改变它们的相对顺序，所以简单选择排序不稳定。B 的说法错误。

可以用 $2_a, 2_b, 1$ 检查选择排序。第一轮选出 1，与第一个位置的 2_a 交换，得到 $1, 2_b, 2_a$。两个关键字为 2 的元素相对顺序被改变，因此不稳定。

十一、变式题

变式题 对数组 `3 1 2 1` 使用下面的排序代码，排序后两个值为 1 的元素相对顺序是否会改变？

```
for (int i = 2; i <= n; i++) {
    int x = a[i];
    int j = i - 1;
    while (j >= 1 && a[j] > x) {
        a[j + 1] = a[j];
        j--;
    }
    a[j + 1] = x;
}
```

- **A.** 一定会改变
- **B.** 一定不会改变
- **C.** 只有数组长度为偶数时会改变
- **D.** 无法根据代码判断

答案：B

解析：
这段代码是插入排序的常见升序实现。关键在 `while` 条件：

`a[j] > x`

只有当前面元素严格大于 `x` 时，才把它向右移动。若前面元素与 `x` 相等，条件不成立，循环停止，新的相等元素会被放在原来相等元素的后面。

因此，两个值为 1 的元素相对顺序不会改变。这个实现是稳定的。

如果条件改成 `a[j] >= x`，那么遇到相等元素时也会移动，新的 1 可能被插到旧的 1 前面，稳定性就会被破坏。判断稳定性时要看代码如何处理相等关键字。

十二、易错点

第一，不要把“排序正确”和“排序稳定”混为一谈。不稳定排序也能得到升序或降序结果，只是相等关键字的相对顺序可能改变。

第二，不要只凭算法名称判断稳定性。若题目给出代码，要看相等时是否交换或移动。比较符号是 `>` 还是 `>=`，会影响稳定性。

第三，选择排序虽然每轮只交换一次，但时间复杂度仍然是 $O(n^2)$。交换次数少不等于比较次数少。

第四，冒泡排序每轮把较大元素推到后面，选择排序每轮选出最小元素放到前面，插入排序每轮把新元素插入前面有序部分。三者过程不要混淆。

第五，归并排序常见时间复杂度是 $O(n\log n)$，但需要额外数组；快速排序平均表现较好，但最坏情况可能为 $O(n^2)$，常见实现通常不稳定。

第六，计数排序依赖值域。若值域很小，可以统计次数；若值域很大，直接计数不合适。

十三、本章要点

排序是按照关键字把元素重新排列。升序是从小到大，降序是从大到小。若元素还有其他信息，关键字相同的元素是否保持原相对顺序，就是稳定性问题。

冒泡排序反复比较相邻元素并交换逆序对，常见实现稳定，时间复杂度 $O(n^2)$。选择排序每轮选择未排序部分的最小元素放到当前位置，常见实现不稳定，时间复杂度 $O(n^2)$。插入排序把当前元素插入前面有序部分，常见实现稳定，最坏时间复杂度 $O(n^2)$。

归并排序使用分治思想，常见时间复杂度 $O(n\log n)$，可以稳定，但通常需要 $O(n)$ 额外空间。快速排序也使用分治思想，平均常见为 $O(n\log n)$，最坏可能为 $O(n^2)$，常见实现通常不稳定。计数排序适合整数值域较小的情况。

做排序题时，要把“比较条件”“交换或移动方式”“相等元素如何处理”圈出来。判断过程题时，按每一轮排序后的序列追踪；判断稳定性题时，用带下标的相等元素，例如 2_a、2_b，检查它们的相对顺序是否可能改变。

第五编 基础算法

0x17 模拟

模拟是最基础的算法思想：题目怎样描述过程，程序就按这个过程一步一步维护状态。它不像二分、搜索、动态规划那样有明显的算法标志，却经常出现在 CSP-J 第一轮的阅读程序和完善程序中。

第一轮考模拟，通常不是让学生写很长的程序，而是让学生看懂一个过程：变量怎样变化，循环执行几次，某个状态什么时候更新，最后输出什么。模拟题最怕“脑中跳步”。一旦少追踪一次更新，后面的结果就会全部偏离。

本章重点讲三件事：怎样把题意变成状态，怎样按时间或步骤推进，怎样在纸面上追踪变量。后续章节会讲枚举、二分、搜索等更明确的算法；本章只讨论“按规则走过程”的模拟。

一、模拟先找状态

模拟题的第一步不是写循环，而是找状态。

状态就是程序在某一时刻需要记住的信息。不同题目中的状态不同。比如：

当前时间；

当前位置；

当前方向；

当前分数；

数组中每个位置的值；

队列或栈中的内容；

某个开关是否打开；

当前轮到谁操作。

如果题目描述“一个人沿路前进，遇到红灯要等待”，状态至少包括当前时间和当前走到哪里。如果题目描述“数组中若干元素反复交换”，状态就是数组当前内容。如果题目描述“游戏按回合进行”，状态可能包括双方分数、当前回合、当前行动者。

找状态时可以问一句：为了继续执行下一步，我必须知道哪些量？这些量就是需要维护的状态。

例如一个简单过程：

初始有 $x = 0$。依次读入 n 个数。若读入的是偶数，就把它加到 x 上；若读入的是奇数，就把 x 减去它。最后输出 x。

这里的状态很少，只需要维护当前答案 x。循环每读一个数，就根据奇偶性更新一次 x。如果输入是 3 2 5 4，过程是：

开始 $x = 0$；

读到 3，奇数，$x = 0 - 3 = -3$；

读到 2，偶数，$x = -3 + 2 = -1$；

读到 5，奇数，$x = -1 - 5 = -6$；

读到 4，偶数，$x = -6 + 4 = -2$。

最后输出 -2。

这就是最基本的模拟。程序不需要想“有没有更巧妙的方法”，只要按规则维护状态即可。

二、按步骤推进

模拟的第二步，是确定每一步做什么。

题目中的过程通常有明显顺序。例如“先移动，再判断是否越界”“先加分，再检查是否结束”“先取出队首，再加入新状态”。顺序一旦变了，答案可能完全不同。

阅读程序时要特别注意更新顺序。

例如下面两段代码看起来很像，但结果不同：

```
x = x + 1;
if (x == 5) ans++;
```

和

```
if (x == 5) ans++;
x = x + 1;
```

第一段是先让 x 增加，再判断它是否等于 5；第二段是先判断旧的 x，再增加。若开始时 x=4，第一段会计数，第二段不会计数。

模拟题中，更新顺序常常比公式更重要。草稿上不要只写变量最终值，最好按步骤写出“先做什么，后做什么”。

对于循环模拟，可以把一次循环看成一个完整步骤。每次循环开始时状态是什么？循环中改了哪些变量？循环结束后状态变成什么？如果能把这三点写清楚，阅读程序题就容易很多。

三、按时间推进

有些模拟题以时间为主线。每过一秒、一天、一轮或一段路程，状态发生变化。

例如交通灯问题：从起点出发，依次经过若干个红绿灯。到达每个红绿灯时，如果是红灯就等待到绿灯；如果是绿灯就直接通过。这样的题不需要搜索路线，只需要按顺序模拟时间变化。

关键状态是当前时间 t。每走一段路，t 增加路程耗时；到达红绿灯后，根据周期判断是否等待。

判断周期时常用取余。若某个红绿灯红灯持续 r 秒，绿灯持续 g 秒，并且从时刻 0 开始先红后绿，那么周期是 r + g。到达时刻为 t，令 p = t % (r + g)。如果 p < r，说明当前处在红灯，需要等待 r - p 秒；否则处在绿灯，可以直接通过。

这个过程可以写成：

```
t += d;
int p = t % (r + g);
if (p < r) t += r - p;
```

这里顺序不能颠倒。必须先走到红绿灯，也就是先让 t += d，再判断灯的状态。若先判断再走路，就判断了错误时刻的灯。

时间模拟题的草稿建议按事件写：

到达第一个位置时，时间是多少；

是否等待，等待多久；

离开时，时间是多少；

再进入下一个位置。

这样比直接在脑中算最终时间更稳。

四、按位置推进

另一类模拟以位置为主线。常见状态是当前位置 x、y，或者数组下标 i。

例如在一维数组上移动，字符 L 表示向左走一步，R 表示向右走一步。若初始位置为 0，指令为 R R L R，则位置变化是：

开始 $x = 0$；

执行 R，$x = 1$；

执行 R，$x = 2$；

执行 L，$x = 1$；

执行 R，$x = 2$。

最后位置是 2。

二维网格中也类似。若用 x 表示行，y 表示列，向上可能是 x--，向下可能是 x++，向左是 y--，向右是 y++。但不同题目可能使用不同坐标习惯，不能只凭印象。题面或代码中怎样定义，就按怎样追踪。

位置模拟题最容易错在边界。数组下标是从 0 开始还是从 1 开始？网格行数是 n，合法行号是 0 到 $n-1$，还是 1 到 n？方向数组中 dx 和 dy 是否对应正确？这些都要在纸面上写清楚。

例如合法下标是 0 到 $n-1$ 时，判断越界常写作：

```
if (x < 0 || x >= n) {
    // 越界
}
```

若合法下标是 1 到 n，判断就应是 `x < 1 || x > n`。同样是越界判断，下标口径不同，条件也不同。

五、数组状态的模拟

数组状态模拟常出现在排序、交换、前缀处理、游戏过程等题中。做这类题时，草稿上要保留每一步后的数组，而不是只记某个变量。

例如有数组：

```
3 1 4 2
```

程序从左到右扫描，若相邻两个数逆序，就交换它们一次。

第一步比较 3 和 1，逆序，交换，数组变为：

```
1 3 4 2
```

第二步比较 3 和 4，不交换，数组仍为：

```
1 3 4 2
```

第三步比较 4 和 2，逆序，交换，数组变为：

```
1 3 2 4
```

这一轮结束后，数组不是完全有序，但最大值已经移动到最后。若题目问“一轮后数组是什么”，答案就是 `1 3 2 4`，不能直接写最终排序结果。

数组模拟题常见陷阱是“题目问的是过程中的某一步”，而不是最终状态。阅读时要看清问题：是问循环执行完一次后的数组，还是全部执行完后的数组？是问第 i 次操作后，还是输出前？

如果数组长度不大，建议每一次关键交换后都写一行新数组。纸面上多写两行，能避免许多选择题误判。

六、计数器和标志变量

模拟程序中常出现计数器和标志变量。

计数器用于记录某种事件发生了多少次。比如 `cnt` 统计满足条件的数有几个，`ans` 统计答案，`sum` 统计总和。

标志变量用于表示某种状态是否出现。比如 `ok` 表示当前过程是否有效，`vis[i]` 表示第 i 个位置是否访问过，`flag` 表示是否找到目标。

例如：

```
bool ok = true;
for (int i = 1; i <= n; i++) {
    if (a[i] < 0) ok = false;
}
if (ok) cout << "YES";
else cout << "NO";
```

这段代码中，ok 一开始为 true，表示暂时没有发现问题。一旦遇到负数，就变成 false。后面即使再遇到正数，也不会把它改回 true。所以它记录的是“是否从头到尾都没有负数”。

阅读标志变量时，要看它会不会被改回去。有些变量只要变成 false 就不会恢复；有些变量每一轮都会重新初始化。若看漏初始化位置，就会误判程序行为。

例如 ok 若写在循环外，只初始化一次；若写在循环内，每一轮都会重新开始。两个位置差别很大。

七、模拟题的检查顺序

做模拟题，可以按固定顺序检查。

第一，看输入和初始状态。变量一开始是多少？数组初始内容是什么？队列、栈是否为空？

第二，看循环边界。循环从哪里开始，到哪里结束？条件是 < 还是 <=？每次增加多少？

第三，看每一步的更新顺序。先判断还是先修改？先输出还是先更新？先入队还是先出队？

第四，看特殊情况。第一个元素、最后一个元素、空数组、只有一个元素时是否会不同？

第五，看输出。输出的是最终状态，还是某个中间状态？有没有固定小数、空格、换行等格式要求？

在 CSP-J 第一轮中，很多模拟题输在第二步和第三步。学生知道题意，却把循环边界看错；知道变量含义，却把更新顺序反了。养成按顺序检查的习惯，比临时找技巧更可靠。

八、例题

例题 阅读下面程序，若输入为 5 3 1 4 2 5，输出是什么？

```
#include <iostream>
using namespace std;

int main() {
    int n, a[105];
    cin >> n;
    for (int i = 1; i <= n; i++) cin >> a[i];

    int cnt = 0;
    for (int i = 1; i < n; i++) {
        if (a[i] > a[i + 1]) {
            int t = a[i];
            a[i] = a[i + 1];
            a[i + 1] = t;
            cnt++;
```

```
        }
    }

    for (int i = 1; i <= n; i++) cout << a[i] << " ";
    cout << cnt << endl;
    return 0;
}
```

A. 1 3 2 4 5 2

B. 1 2 3 4 5 3

C. 3 1 2 4 5 1

D. 1 3 4 2 5 1

答案：A

解析：

输入中 n=5，数组初始为：

3 1 4 2 5

程序只进行一轮相邻比较交换，循环条件是 i < n，所以 i 从 1 到 4。

当 i=1 时，比较 3 和 1，因为 $3 > 1$，交换，数组变为：

1 3 4 2 5

cnt 变为 1。

当 i=2 时，比较当前位置的 3 和 4，不交换。

数组仍为：

1 3 4 2 5

当 i=3 时，比较 4 和 2，因为 $4 > 2$，交换，数组变为：

1 3 2 4 5

cnt 变为 2。

当 i=4 时，比较 4 和 5，不交换。

最后输出数组，再输出 cnt。因此输出为：

1 3 2 4 5 2

选 A。

这道题的关键是：程序只做一轮相邻交换，不是完整排序。若直接把数组排成有序，就会选错。

九、变式题

变式题 若把例题中的第二个循环改为下面形式，输入仍为 5 3 1 4 2 5，输出中的 cnt 是多少？

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

A. 1

B. 2

C. 3

D. 4

答案：C

解析：

修改后，只有当前一个数小于后一个数时才交换，也就是把较大的数向前移动。

初始数组为：

`3 1 4 2 5`

当 `i=1` 时，比较 3 和 1，$3 < 1$ 不成立，不交换。

当 `i=2` 时，比较 1 和 4，$1 < 4$ 成立，交换，数组变为：

`3 4 1 2 5`

`cnt=1`。

当 `i=3` 时，比较当前位置的 1 和 2，$1 < 2$ 成立，交换，数组变为：

`3 4 2 1 5`

`cnt=2`。

当 `i=4` 时，比较 1 和 5，$1 < 5$ 成立，交换，数组变为：

`3 4 2 5 1`

`cnt=3`。

因此 `cnt` 为 3，答案是 C。

这道变式的重点是数组会在循环中被修改。不能用初始数组一直判断每一对相邻元素，要使用修改后的数组继续往下模拟。

十、本章要点

模拟就是按题目或程序给出的规则一步一步维护状态。它不靠复杂公式，靠状态清楚、顺序清楚、边界清楚。

做模拟题先找状态：当前时间、当前位置、数组内容、计数器、标志变量等。再确定每一步如何推进：按时间、按位置、按数组下标，还是按回合。程序中的更新顺序不能随意改变。

阅读模拟程序时，要重点检查循环边界、变量更新、数组下标、特殊位置和输出内容。`<` 与 `<=`、先判断与先修改、初始化在循环内还是循环外，都可能改变答案。

数组模拟题要保留每一步后的数组状态。队列、栈模拟题要保留当前容器内容。时间模拟题要保留到达时刻、等待时间和离开时刻。

CSP-J 第一轮中的模拟题通常不难在思路，而难在细节。纸面追踪时不要跳步，尤其不要把“一轮操作”误认为“完整处理”。只要每一步都按当前状态继续，模拟题就能稳定得分。

0x18 枚举

枚举是最基础的算法思想之一：把可能的情况一一列出来，再检查哪些情况满足题意。它看起来简单，却是 CSP-J 第一轮中非常重要的程序阅读和完善程序考点。

模拟强调“按过程走”，枚举强调“把候选试完”。题目如果问“是否存在某个数”“有多少对数满足条件”“最小的合法答案是多少”，程序里往往会出现枚举。阅读这类程序时，重点不是看循环有几层，而是看它枚举的对象是什么，范围是否完整，条件是否正确，是否重复计算。

本章只讲第一轮常见枚举：一重枚举、二重枚举、去重、枚举答案、范围缩小和完善程序中的空缺判断。搜索章节会继续讲更复杂的递归枚举，本章不提前展开。

一、枚举在做什么

枚举可以理解为“把候选答案都试一遍”。

例如要在 1 到 100 中找出所有能被 7 整除的数，可以从 1 到 100 逐个检查：

```
for (int x = 1; x <= 100; x++) {
    if (x % 7 == 0) {
        cout << x << " ";
    }
}
```

这段程序枚举的对象是 `x`，范围是 1 到 100，检查条件是 `x % 7 == 0`。只要这三点清楚，就能看懂程序意图。

枚举题通常由三部分组成：

枚举对象：要试的是数、下标、区间、两个位置，还是某种方案；

枚举范围：从哪里开始，到哪里结束，是否包含端点；

检查条件：当前候选是否满足题目要求。

做题时不要先急着算答案。先在草稿上写一句：“这个循环枚举的是谁？”很多程序阅读题，只要这句话写对，后面的变量追踪就会轻很多。

例如：

```
int ans = 0;
for (int i = 1; i <= n; i++) {
    if (a[i] % 2 == 0) ans++;
}
```

这段程序枚举的是数组下标 `i`，范围是 1 到 n，检查条件是 `a[i]` 是否为偶数。它统计的是数组中偶数的个数，而不是统计下标中的偶数个数。若把 `i % 2 == 0` 和 `a[i] % 2 == 0` 看混，就会误判程序作用。

二、一重枚举：逐个检查

一重枚举只枚举一个变量。常见形式是从小到大试每一个数、每一个位置或每一个候选答案。

例如，统计 1 到 n 中有多少个数能同时被 3 和 5 整除：

```
int cnt = 0;
for (int x = 1; x <= n; x++) {
    if (x % 3 == 0 && x % 5 == 0) {
        cnt++;
    }
}
```

这里枚举的是 `x`，范围是 1 到 n。检查条件是 `x % 3 == 0 && x % 5 == 0`。因为要“同时”被 3 和 5 整除，所以使用逻辑与 `&&`。

若条件改成 `x % 3 == 0 || x % 5 == 0`，含义就变成“能被 3 或 5 至少一个整除”。两个条件得到的答案不同。

一重枚举题的易错点主要在范围和条件。

循环写成 `x < n`，就不会检查 n；写成 `x <= n`，会检查 n。如果题目要求“1 到 n”，通常应包含 n。如果题目要求“小于 n”，就不应包含 n。

还有一些枚举从 0 开始。例如数组下标若是 0 到 $n-1$，循环应写作 `i = 0; i < n; i++`。若数组下标是 1 到 n，循环常写作 `i = 1; i <= n; i++`。第一轮阅读程序时，必须先确认下标口径。

三、二重枚举：枚举一对对象

如果题目要求找两个数、两个位置、两个端点，常用二重枚举。

例如给定 n 个数，统计有多少对下标 i, j 满足 $i < j$ 且 `a[i] + a[j] == k`。

可以这样写：

```
int cnt = 0;
for (int i = 1; i <= n; i++) {
    for (int j = i + 1; j <= n; j++) {
        if (a[i] + a[j] == k) {
            cnt++;
        }
    }
}
```

外层枚举第一个下标 `i`，内层枚举第二个下标 `j`。由于 `j` 从 `i + 1` 开始，所以一定满足 $i < j$，同一对下标不会被重复统计。

如果内层写成 `j = 1; j <= n; j++`，就会出现两个问题。第一，可能统计到 $i = j$ 的情况，也就是同一个位置被选两次；第二，(i, j) 和 (j, i) 会被算成两次。除非题目明确把顺序不同看作不同方案，否则这样会重复。

二重枚举最常见的范围设计有三种。

枚举所有有序对：`i` 从 1 到 n，`j` 从 1 到 n。这会包含 $i = j$，也会区分 (i, j) 和 (j, i)。

枚举两个不同位置的有序对：`i` 从 1 到 n，`j` 从 1 到 n，并检查 `i != j`。这不包含同一位置，但仍然区分顺序。

枚举无序对：`i` 从 1 到 n，`j` 从 `i + 1` 到 n。这既不包含同一位置，也不会重复统计同一对。

做题时要看题意。若题目说“选出两个不同位置”，但不区分先后，通常用 $i < j$。若题目说“第一个位置和第二个位置”，或者两者角色不同，可能需要有序对。

四、多重枚举和复杂度

三重枚举可以枚举三个对象。例如枚举三个下标 $i < j < k$，可以写成三层循环。若每一层都和 n 同级，时间复杂度通常是 $O(n^3)$。

枚举层数越多，复杂度增长越快。CSP-J 第一轮常考的不是让学生优化很复杂的枚举，而是判断程序是否在枚举所有情况，以及复杂度是多少。

例如：

```
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++) {
            if (a[i] + a[j] + a[k] == s) cnt++;
        }
    }
}
```

这段程序枚举的是三个不同下标，并且满足 $i < j < k$。它不会重复统计同一组三个位置。复杂度是 $O(n^3)$。

若 n 很小，这样枚举可以接受；若 n 很大，就可能太慢。第一轮中，复杂度题会要求判断这类循环的增长级别。不要只说“三层循环就是 $O(n^3)$”，还要看每层范围是否真的随 n 增长。若某层只到固定常数，就不是多乘一个 n。

本章不展开优化方法。遇到枚举程序时，先能看懂它枚举了什么，再判断复杂度，这已经是第一轮的核心要求。

五、枚举时避免重复

枚举题常见错误是重复计算。同一个方案被算了两次，答案就会偏大。

例如从 n 个数中选两个数，若不区分顺序，那么选第 2 个和第 5 个，与选第 5 个和第 2 个是同一种方案。此时应该用 $i < j$ 限制，只统计一次。

再例如从 n 个数中选三个数，若不区分顺序，就可以使用 $i < j < k$。这样每组三个位置只会出现一次。

去重的核心是给方案规定一个唯一表示。对于一对下标，可以规定较小的下标在前，即 $i < j$。对于三个下标，可以规定 $i < j < k$。对于一段区间，可以规定左端点 l 不大于右端点 r。

如果题目要求统计区间，常见枚举方式是：

左端点 `l` 从 1 到 n；

右端点 `r` 从 `l` 到 n。

这样每个区间 `[l,r]` 都会被枚举一次，且不会出现 `[3,2]` 这种非法区间。

如果题目说“两个数的先后顺序有意义”，就不能随便加 $i < j$。比如选一个数作被减数、另一个数作减数，$a_i - a_j$ 和 $a_j - a_i$ 可能不同。此时两个位置的角色不同，可能需要枚举有序对。

所以，是否去重不是看循环好不好写，而是看题目中的方案是否区分顺序。

六、枚举答案

有些题可以直接枚举答案。也就是说，不是枚举输入中的元素，而是枚举最终答案可能是多少，再检查它是否满足条件。

例如要找最小的正整数 x，使得 x 能被 3 整除，也能被 5 整除，并且 $x > 20$。可以从 21 开始枚举：

```
for (int x = 21; ; x++) {
    if (x % 3 == 0 && x % 5 == 0) {
        cout << x;
        break;
    }
}
```

这段程序没有写循环上界，因为理论上一定能找到。找到第一个满足条件的 `x` 后，用 `break` 结束循环。由于是从小到大枚举，所以第一个合法值就是最小合法答案。

枚举答案时，要特别注意枚举顺序。

若要找最小值，通常从小到大枚举，找到第一个合法答案就停止。

若要找最大值，通常从大到小枚举，找到第一个合法答案就停止。

如果从小到大枚举最大值，就不能遇到第一个合法值就停止，因为后面可能还有更大的合法值。此时要么继续枚举并更新答案，要么改成从大到小枚举。

例如在 1 到 100 中找最大的能被 7 整除的数，可以从 100 向下枚举。第一次遇到合法数就是答案。

枚举答案的程序中，`break` 很常见。阅读时要看清它跳出的是哪一层循环。如果只跳出内层循环，外层可能还会继续执行。多层循环中的 `break` 是第一轮程序阅读常见细节。

七、缩小枚举范围

枚举不是乱试。一个好的枚举程序会先根据题意缩小范围。

例如要找 n 的因数，不需要从 1 枚举到很大的数，只需要枚举 1 到 n，甚至可以枚举到满足 $i \times i \le n$ 的位置，再利用因数成对出现。

例如要判断一个数是否为质数，不需要检查所有比它小的数，只要检查从 2 到 $\sqrt{n}$ 的整数。这一点在简单数论章节已经讲过。

再例如要枚举两位数，可以让十位从 1 到 9，个位从 0 到 9。这样比从 0 到 99 再判断是不是两位数更直接。

枚举范围缩小后，程序更短，复杂度也更低。但缩小范围不能漏掉合法情况。每次缩小范围都要能解释：为什么范围外的情况一定不可能成为答案？

CSP-J 第一轮完善程序中，空缺经常出现在循环边界处。若题目要求枚举所有正因数，边界可能是 `i <= n` 或 `i * i <= n`。若题目要求枚举所有两位数，十位不能从 0 开始。若题目要求枚举数组相邻元素，下标通常不能到 `n` 后还访问 `a[i + 1]`。

看边界时要同时看循环体。如果循环体访问了 `a[i + 1]`，那么 `i` 通常最多到 `n - 1`。如果循环体访问了 `a[j]` 且 `j` 从 `i + 1` 开始，那么外层 `i` 通常最多到 `n - 1` 即可，但写到 `n` 也可能只是让内层不执行。判断时要看具体代码。

八、枚举与条件判断

枚举程序通常由循环和条件判断组成。循环负责把候选列出来，条件判断负责筛掉不合法候选。

如果条件少写，答案会偏大；如果条件多写或写错，答案会偏小。

例如要统计 1 到 n 中满足“能被 3 整除但不能被 5 整除”的数，条件应是：

```
x % 3 == 0 && x % 5 != 0
```

如果写成 `x % 3 == 0 || x % 5 != 0`，含义就完全变了。

再例如，要枚举三位数 abc，要求三个数字互不相同。若用 `a`、`b`、`c` 分别表示百位、十位、个位，那么条件至少要包括：

```
a != b && a != c && b != c
```

少写一个不等关系，就可能允许两个数字相同。

枚举程序中的条件判断常常比循环本身更容易出错。阅读时可以把条件逐条翻译成中文，检查是否与题意一致。若题意有多个要求，代码中也应有对应的多个判断。若某个要求没有在循环范围中体现，也没有在 `if` 中体现，程序很可能漏判。

九、完善程序中的枚举空

完善程序题中，枚举相关空缺常出现在四个位置。

第一，循环初值。比如枚举正整数通常从 1 开始，枚举数组下标可能从 0 或 1 开始，枚举质数因子通常从 2 开始。

第二，循环终止条件。比如是否写 `i <= n`，是否写 `i < n`，是否写 `i * i <= n`。这类空缺最容易因为边界少一个而出错。

第三，筛选条件。也就是 `if` 中判断当前候选是否合法。若题目要求“同时满足”，通常要用 `&&`；若要求“至少满足一个”，通常要用 `||`。

第四，答案更新。比如 `cnt++`、`ans = max(ans, x)`、`break`。题目问数量、最大值、最小值、是否存在，更新方式都不同。

例如题目要求“判断是否存在一个数满足条件”。程序可能用 `bool ok = false`，找到后设为 `true`。若题目要求“统计有多少个数满足条件”，程序就应使用计数器，而不是找到一个就停止。

完善程序时，不要只看空缺所在一行。要先判断整段程序在枚举什么，再判断这个空缺承担的是范围、条件还是更新。

十、例题

例题 阅读下面程序，若输入为 `6 1 4 2 3 5 6 7`，输出是什么？

```cpp
#include <iostream>
using namespace std;

int main() {
    int n, k, a[105];
    cin >> n;
    for (int i = 1; i <= n; i++) cin >> a[i];
    cin >> k;

    int cnt = 0;
    for (int i = 1; i <= n; i++) {
        for (int j = i + 1; j <= n; j++) {
            if (a[i] + a[j] == k) {
                cnt++;
            }
        }
    }

    cout << cnt << endl;
    return 0;
}
```

A. 1

B. 2

C. 3

D. 4

答案：C

解析：
输入中 $n=6$，数组为：

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

目标和 $k=7$。

程序用二重循环枚举下标对 (i,j)，其中 `j` 从 `i + 1` 开始，所以枚举的是所有满足 $i<j$ 的无序下标对。每一对只会被检查一次。

逐一找和为 7 的数对：

$1+6=7$，对应下标 $(1,6)$；

$4+3=7$，对应下标 $(2,4)$；

$2+5=7$，对应下标 $(3,5)$。

没有其他满足条件的下标对，所以 `cnt=3`，输出 3。

这道题的关键是看清内层循环从 `i + 1` 开始。程序不是统计有序对，也不会把 $1+6$ 和 $6+1$ 算成两次。

十一、变式题

变式题 若把例题中的内层循环改成下面形式，其他代码和输入不变，输出会变成多少？

```
for (int j = 1; j <= n; j++) {
    if (i != j && a[i] + a[j] == k) {
        cnt++;
    }
}
```

A. 3
B. 4
C. 5
D. 6

答案：D

解析：
修改后，外层 `i` 仍然从 1 到 n，内层 `j` 也从 1 到 n。条件 `i != j` 排除了同一个位置被选两次，但并没有规定 $i<j$。

因此，若 (i,j) 满足条件，那么 (j,i) 也会被统计一次。

原来满足条件的无序下标对有三组：

$(1,6)$，$(2,4)$，$(3,5)$。

修改后会统计为：

$(1,6)$ 和 $(6,1)$；

$(2,4)$ 和 $(4,2)$；

$(3,5)$ 和 $(5,3)$。

一共 6 次，所以输出 6。

这道变式说明，`i != j` 只能保证两个位置不同，不能避免顺序重复。若题目不区分先后，通常要用 $i<j$ 这样的限制。

十二、本章要点

枚举就是把候选情况逐一试完，再检查哪些满足条件。做枚举题先看三件事：枚举对象、枚举范围、检查条件。

一重枚举常用于逐个检查数或位置。二重枚举常用于枚举两个对象。若不区分顺序，常用 $i<j$；若区分顺序，可能需要枚举有序对。多重枚举要注意复杂度随层数增长。

枚举时要避免重复。给方案规定唯一表示，是去重的常用方法。例如用 $i<j$ 表示一对下标，用 $i<j<k$ 表示三个下标，用 $l \le r$ 表示区间。

枚举答案时，要根据题目要求选择顺序。找最小合法答案通常从小到大枚举，遇到第一个合法值即可停止；找最大合法答案通常可以从大到小枚举。

完善程序中的枚举空常出现在循环初值、循环终止条件、筛选条件和答案更新处。判断空缺时，要先看整段程序枚举的是什么，再看空缺负责控制范围、筛选合法性，还是更新答案。

枚举题看似简单，但边界和去重很容易错。只要把“试的是谁、试到哪里、怎样判断、是否重复”四个问题写清楚，第一轮中的枚举题通常能稳定处理。

0x19 二分

二分是一种把查找范围不断缩小一半的方法。CSP-J 第一轮中，二分常出现在选择题、程序阅读和完善程序中：判断二分查找能否使用，追踪 `l`、`r`、`mid` 的变化，判断循环边界是否正确，或者看懂 `lower_bound`、`upper_bound` 这类函数的含义。

二分最容易错的不是思想，而是边界。`while (l <= r)` 和 `while (l < r)` 的含义不同，`r = mid - 1` 和 `r = mid` 也不能随便替换。本章重点讲清楚二分的使用条件、常见写法和纸面追踪方法。

一、二分能解决什么

最常见的二分查找用于有序数组。若数组已经从小到大排好，要找某个数 `x` 是否出现，就可以每次检查中间位置。

如果中间元素等于 `x`，查找成功。

如果中间元素小于 `x`，由于数组升序，左半边都不可能是 `x`，只需要到右半边继续找。

如果中间元素大于 `x`，右半边都不可能是 `x`，只需要到左半边继续找。

例如数组为：

```
1 3 5 7 9 11 13
```

要找 9。先看中间的 7，因为 $7 < 9$，所以只在右边 `9 11 13` 中查找。再看中间的 11，因为 $11 > 9$，所以只在左边 `9` 中查找。最后找到 9。

二分的关键条件是：每次比较后，都能排除一半左右的范围。如果数组无序，看到中间元素并不能判断目标在左边还是右边，就不能直接使用二分查找。

例如数组 `5 1 9 3 7` 中，中间元素是 9。若要找 7，不能因为 $9 > 7$ 就去左边，因为 7 实际在右边。无序数组不满足二分查找的基本条件。

所以，做二分题先问一句：范围是否有序，或者判断条件是否具有单调性？如果没有，就不能随便二分。

二、左闭右闭写法

最容易理解的二分查找写法，是左闭右闭区间。区间 `[l,r]` 表示当前可能包含答案的位置从 `l` 到 `r`，两端都包含。

```
int l = 1, r = n;
bool found = false;
```

```
while (l <= r) {
    int mid = (l + r) / 2;
    if (a[mid] == x) {
        found = true;
        break;
    } else if (a[mid] < x) {
        l = mid + 1;
    } else {
        r = mid - 1;
    }
}
```

这里循环条件是 `l <= r`，因为只要区间中还有一个位置，就要继续检查。当 `l > r` 时，区间为空，查找失败。

若 `a[mid] < x`，说明 `mid` 及其左侧都不可能是答案。新的左边界应为 `mid + 1`。

若 `a[mid] > x`，说明 `mid` 及其右侧都不可能是答案。新的右边界应为 `mid - 1`。

这两个 `+1` 和 `-1` 很重要。因为 `mid` 已经被检查过，并且已经确定不是答案，所以新范围不应再包含 `mid`。如果写成 `l = mid` 或 `r = mid`，在某些情况下范围不会缩小，循环可能无法结束。

例如 `l=1,r=2` 时，`mid=(1+2)/2=1`。若需要右半边却写成 `l=mid`，那么 `l` 仍是 1，区间没有变化，下一轮还是同样的 `mid`，程序就可能卡住。

左闭右闭写法适合判断某个值是否存在。阅读程序时，只要看到 `while (l <= r)`，就要检查更新是否是 `l = mid + 1` 或 `r = mid - 1`。

三、二分过程追踪

二分阅读题经常要求判断循环执行几次、最后 `l` 和 `r` 是多少，或者输出哪个位置。此时要按表格追踪。

例如数组：

`a[1..8] = 2 4 6 8 10 12 14 16`

查找 12，使用左闭右闭写法。

```
初始：l = 1, r = 8
mid = 4, a[4] = 8, 8 < 12，所以 l = 5

现在：l = 5, r = 8
mid = 6, a[6] = 12，找到
```

只执行两轮就找到。

若查找 5：

```
初始：l = 1, r = 8
mid = 4, a[4] = 8, 8 > 5，所以 r = 3

现在：l = 1, r = 3
mid = 2, a[2] = 4, 4 < 5，所以 l = 3

现在：l = 3, r = 3
mid = 3, a[3] = 6, 6 > 5，所以 r = 2

现在：l = 3, r = 2，循环结束，未找到
```

这类题不要在脑中直接跳答案。每一轮写出 `l`、`r`、`mid` 和比较结果，就不会把边界追丢。

如果题目问“循环结束后 `l` 的值”，要注意即使没有找到目标，`l` 也不是随便的。对于升序数组的左闭右闭查找，查找失败后，`l` 常停在第一个大于目标值的位置附近。这也是后面 `lower_bound` 的思想基础。

四、左闭右开区间

另一种常见写法是左闭右开区间 `[l,r)`。它表示可能范围包含 `l`，不包含 `r`。数组中要查找全部位置时，若下标从 0 到 $n-1$，可以令 `l=0,r=n`。

左闭右开写法常用于寻找“第一个满足条件的位置”。

例如在升序数组中，找第一个大于等于 `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;
    }
}
```

循环结束时，`l == r`，这个位置就是第一个满足 `a[pos] >= x` 的位置。如果所有元素都小于 `x`，结果会是 `n`，表示不存在这样的数组位置。

这里为什么是 `r = mid`，不是 `r = mid - 1`？

在这类写法中，`r` 不是“最后一个还要检查的下标”，而是右边界。当 `a[mid] >= x` 时，`mid` 可能就是第一个满足条件的位置，所以不能用 `mid - 1` 把它越过去。令 `r = mid`，表示把当前候选边界收缩到 `mid`，接下来继续在左侧寻找是否还有更靠前的满足位置。

若左侧没有更早的位置，循环最终会在 `mid` 处收敛；若左侧还有满足条件的位置，`r` 会继续向左收缩。因此，左闭右开写法中找第一个满足条件的位置时，条件成立写 `r = mid`。

若觉得这句话绕，可以直接记住左闭右开找第一个满足条件的位置：

条件成立时，`r = mid`；

条件不成立时，`l = mid + 1;`

循环条件是 `l < r;`

结束时答案是 `l`。

这套写法只要条件具有单调性，就很稳定。

五、找第一个和最后一个

二分不只可以判断是否存在，还可以找边界。第一轮中最常见的是找第一个不小于 x 的位置，以及找第一个大于 x 的位置。

在升序数组中，找第一个大于等于 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;
}
```

这就是 lower_bound 的含义。

找第一个大于 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;
}
```

这就是 upper_bound 的含义。

两段代码只差一个等号。第一段条件是 `a[mid] >= x`，第二段条件是 `a[mid] > x`。这个等号决定了相等元素被归到哪一边。

例如数组：

```
1 2 2 2 4 5
```

若 x=2，第一个大于等于 2 的位置是第 2 个元素；第一个大于 2 的位置是第 5 个元素。

因此，值为 2 的元素个数可以用：

```
upper_bound(2) - lower_bound(2)
```

从位置角度看，就是第一个大于 2 的位置减去第一个不小于 2 的位置。

CSP-J 第一轮若出现 lower_bound 和 upper_bound，通常不会要求深入 STL 细节，而是要求知道它们返回的边界含义。记住：lower_bound 找“不小于”，upper_bound 找“大于”。

六、二分答案

除了在数组中查找位置，二分还可以用来查找答案。这叫二分答案。

二分答案的前提不是数组有序，而是“答案是否可行”具有单调性。也就是说，随着候选答案变大或变小，可行性会从一边连续变到另一边。

例如要找最小的整数 x，使得 $x^2 \geq n$。可以对 x 二分。若某个 x 已经满足 $x^2 \geq n$，那么比它更大的数也都满足。若某个 x 不满足，那么比它更小的数也不满足。条件具有单调性。

可以写成：

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

这段程序找到的是最小的 x，使得 x * x >= n。

例如 $n = 20$。$4^2 = 16 < 20$，$5^2 = 25 \geq 20$，所以答案是 5。

二分答案题最重要的是写出判断函数或判断条件。这个条件必须能把候选答案分成两段：一段不满足，一段满足；或者一段满足，一段不满足。若可行性一会儿真、一会儿假，没有连续分界，就不能直接二分。

CSP-J 第一轮中，二分答案常以完善程序形式出现。空缺可能是判断条件，也可能是边界更新。做这类题时，先判断程序要找的是“最小可行值”还是“最大可行值”。

七、找最小可行值

若要找最小可行值，常用左闭右开写法：

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

这里假设 check(x) 的含义是“候选答案 x 是否可行”，并且当某个 x 可行时，更大的值也可行。也就是结果形如：

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

此时第一个 `true` 就是最小可行值。

当 `check(mid)` 为真时，`mid` 可能就是答案，所以不能跳过它，令 `r = mid`。

当 `check(mid)` 为假时，`mid` 和更小的值都不可能是答案，所以令 `l = mid + 1`。

循环结束时，`l` 就是最小可行值。

例如找最小的 x，使得 $x^2 \geq 20$。`check(x)` 是 `x * x >= 20`。可行性从 $x = 5$ 开始变为真，所以答案是 5。

八、找最大可行值

若要找最大可行值，可以把条件方向反过来。假设 `check(x)` 表示 `x` 是否可行，并且当某个 `x` 可行时，更小的值也可行。结果形如：

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

此时要找最后一个 `true`。

一种常见写法是：

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

这里 `mid` 使用 `(l + r + 1) / 2`，也就是偏右的中点。

为什么要偏右？因为当只剩两个数时，如果还用 `(l + r) / 2`，`mid` 会等于左边的 `l`。若 `check(mid)` 为真并执行 `l = mid`，范围不会缩小，可能循环不结束。使用偏右中点，可以保证在保留 `mid` 时仍然推进边界。

找最大可行值时：

`check(mid)` 为真，说明 `mid` 可行，答案可能是 `mid` 或更大，所以 `l = mid`；

`check(mid)` 为假，说明 `mid` 太大，答案在左边，所以 `r = mid - 1`。

这套写法和找最小可行值成对出现。做完善程序题时，要先看题目是在找最小，还是找最大，不能把两个模板混在一起。

九、中点和溢出

中点常写作：

```
int mid = (l + r) / 2;
```

在 CSP-J 第一轮的纸面题中，这种写法足够常见，也便于手算。但在较大整数范围中，`l + r` 可能超过 `int` 能表示的范围。更稳的写法是：

```
int mid = l + (r - l) / 2;
```

如果要取偏右中点，可以写作：

```
int mid = l + (r - l + 1) / 2;
```

第一轮通常不会专门难为中点溢出，但阅读程序时要知道这两种写法表达的是同一个意思：取当前范围的中间位置。若题目问哪种写法更安全，`l + (r - l) / 2` 更能避免 `l + r` 过大。

还要注意整除。C++ 中整数除法会舍去小数部分。例如 `(1 + 2) / 2` 得到 1，不是 1.5。这就是左中点。若需要右中点，需要写 `(l + r + 1) / 2` 或 `l + (r - l + 1) / 2`。

十、例题

例题 已知数组 `a[1..7] = 1 3 5 7 9 11 13`，执行下面程序查找 `x=8`，程序结束时 `l` 和 `r` 分别是多少？

```
int l = 1, r = 7, x = 8;
while (l <= r) {
    int mid = (l + r) / 2;
    if (a[mid] == x) {
        break;
    } else if (a[mid] < x) {
        l = mid + 1;
    } else {
        r = mid - 1;
    }
}
cout << l << " " << r << endl;
```

A. `4 4`

B. `5 4`

C. `4 5`

D. `5 5`

答案：B

解析：

按二分过程追踪。

初始 `l=1,r=7`，`mid=(1+7)/2=4`，`a[4]=7`。因为 $7 < 8$，所以 `l=mid+1=5`。

此时 `l=5,r=7`，`mid=(5+7)/2=6`，`a[6]=11`。因为 $11 > 8$，所以 `r=mid-1=5`。

此时 `l=5,r=5`，`mid=5`，`a[5]=9`。因为 $9 > 8$，所以 `r=4`。

此时 l=5,r=4，循环条件 l <= r 不成立，循环结束。输出为：

5 4

所以选 B。

这道题没有找到 8，但仍然要继续追踪到区间变空。左闭右闭写法查找失败时，常见结束状态是 l = r + 1。

十一、变式题

变式题 已知数组 a[0..5] = 1 2 2 2 4 5，执行下面程序，输出是多少？

```
int a[6] = {1, 2, 2, 2, 4, 5};
int l = 0, r = 6, x = 2;
while (l < r) {
    int mid = (l + r) / 2;
    if (a[mid] >= x) r = mid;
    else l = mid + 1;
}
cout << l << endl;
```

A. 0

B. 1

C. 3

D. 4

答案：B

解析：
这段程序使用左闭右开区间 [l,r)，寻找第一个满足 a[pos] >= x 的位置，也就是 lower_bound 的含义。

数组为：

```
下标：0 1 2 3 4 5
数值：1 2 2 2 4 5
```

当 x=2 时，第一个大于等于 2 的位置是下标 1。

追踪程序也能得到同样结果。

初始 l=0,r=6，mid=3，a[3]=2，满足 a[mid] >= x，所以 r=3。

此时 l=0,r=3，mid=1，a[1]=2，仍满足条件，所以 r=1。

此时 l=0,r=1，mid=0，a[0]=1，不满足 a[mid] >= x，所以 l=1。

循环结束，l=1,r=1，输出 1。答案选 B。

这道题的易错点是把它当成普通二分查找某个等于 2 的位置。事实上程序找的是第一个不小于 2 的位置，不是任意一个 2 的位置。

十二、易错点

第一，二分查找要求范围有序，或者判断条件具有单调性。无序数组不能直接使用普通二分查找。

第二，左闭右闭区间 `[l,r]` 中，循环条件通常是 `l <= r`。若排除 `mid`，应写 `l = mid + 1` 或 `r = mid - 1`。

第三，找第一个满足条件的位置时，常用左闭右开写法。条件成立时，`r = mid`；条件不成立时，`l = mid + 1`。

第四，找最大可行值时，常使用右中点 `(l + r + 1) / 2`。若使用左中点并写 `l = mid`，在只剩两个数时可能无法推进。

第五，`lower_bound` 找第一个不小于目标值的位置，`upper_bound` 找第一个大于目标值的位置。二者只差一个等号，含义却不同。

第六，`break` 会提前结束循环。若二分代码中找到目标后使用 `break`，后续边界不再继续变化。追踪程序时要注意是否已经跳出循环。

十三、本章要点

二分的核心是每次排除一半左右的范围。它能用于有序数组查找，也能用于具有单调性的答案范围。

普通二分查找常用左闭右闭区间 `[l,r]`。循环条件是 `l <= r`，比较后用 `l = mid + 1` 或 `r = mid - 1` 排除已经检查过的 `mid`。

寻找第一个满足条件的位置，常用左闭右开区间 `[l,r)`。若 `check(mid)` 成立，保留 `mid`，令 `r = mid`；若不成立，令 `l = mid + 1`。循环结束时 `l == r`，即为边界位置。

二分答案时，要先确认可行性具有单调性。找最小可行值和找最大可行值的模板不同，特别是最大可行值常用右中点。

做二分题时，不要只背模板。先判断区间含义，再看循环条件，最后检查边界更新是否能让范围缩小。只要 `l`、`r`、`mid` 的含义清楚，二分题就不会只靠猜。

0x1A 递归问题

递归是 CSP-J 第一轮中常见的程序阅读考点。它既可能出现在选择题中，也可能藏在阅读程序和完善程序里：给出一个函数，要求判断返回值；给出一段递归代码，要求判断输出顺序；或者让学生补出递归出口、递归参数和返回表达式。

第 8 章已经介绍过函数与递归的基本写法。本章不再把递归当作语法点，而是把它当作一种读程序的方法来讲。读递归程序时，不要只盯着函数名反复出现，而要抓住三件事：递归出口是什么，每一次递归后问题是否变小，返回值怎样一层一层传回来。

后续搜索章节会大量使用递归。本章先把递归本身讲清楚，尤其是 CSP-J 第一轮中最容易出错的手算过程。

一、递归是什么

递归，就是函数在求解问题时直接或间接调用自身。

例如，求 $n!$ 可以这样理解：$n!$ 等于 $n \times (n-1)!$。要算 5!，可以先算 4!；要算 4!，可以先算 3!；一直到 1!，它可以直接得到。

这就是递归的基本思想：把一个较大的问题，转化为一个或几个规模更小、形式相同的问题。

下面是一段求阶乘的递归函数：

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

这里 `fac(n)` 调用了 `fac(n - 1)`，所以它是递归函数。`n == 1` 时直接返回 1，不再继续调用，这是递归出口。若没有出口，函数会一直调用下去，程序无法正常结束。

递归函数通常包含两个部分。

第一部分是递归出口，也叫边界情况。当问题足够小时，可以直接给出答案。

第二部分是递归步骤，也叫递归关系。它把当前问题转化为更小的同类问题。

阅读递归代码时，第一句话要问：出口在哪里？第二句话要问：每次调用后，参数是否向出口靠近？如果参数没有变小，也没有接近出口，递归很可能出错。

二、出口决定能不能停

递归出口是递归程序最重要的条件。没有出口，递归无法停止；出口写错，答案会错；出口太窄，某些输入可能无法结束。

例如上面的阶乘函数使用 `if (n == 1) return 1;` 作为出口。若输入保证 $n \geq 1$，这是可以的。

但如果题目允许 $n = 0$，就要考虑 $0! = 1$。这时出口可以写成：

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

若仍然只写 `n == 1`，输入 0 时会调用 `fac(-1)`，再调用 `fac(-2)`，无法向正确出口靠近。

有些递归出口不是一个值，而是一段范围。例如斐波那契数列常写作：$f(1) = 1$，$f(2) = 1$，当 $n \geq 3$ 时，$f(n) = f(n-1) + f(n-2)$。

它的出口有两个：$n = 1$ 和 $n = 2$。若只写一个出口，另一个小规模情况就可能处理错误。

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

完善程序题中，出口经常是空缺位置。判断出口时，要先看递归调用的参数怎样变化。如果每次调用 `f(n - 1)`，出口通常在较小的 n 处；如果每次调用 `f(n / 2)`，出口通常和 $n \leq 1$ 之类条件有关；如果递归处理数组区间，出口常是区间为空或只剩一个元素。

不要只看函数名，要看参数如何变小。

三、返回值怎样传回

递归程序的难点在于：调用时一层一层向下走，返回时又一层一层向上算。

以 `fac(4)` 为例。数学上可以写成：

$\text{fac}(4) = 4 \times \text{fac}(3)$；

$\text{fac}(3) = 3 \times \text{fac}(2)$；

$\text{fac}(2) = 2 \times \text{fac}(1)$；

$\text{fac}(1) = 1$。

到达出口后，再倒着计算：

$\text{fac}(2) = 2 \times 1 = 2$；

$\text{fac}(3) = 3 \times 2 = 6$；

$\text{fac}(4) = 4 \times 6 = 24$。

读递归返回值时，可以先向下展开到出口，再从出口往回代值。不要一看到 `fac(4)` 就直接心算出 24，尤其在题目稍复杂时，这样很容易漏掉某一层额外操作。

再看下面的函数：

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

求 `f(4)`。

向下展开：

$f(4) = f(3) + 16$；

$f(3) = f(2) + 9$；

$f(2) = f(1) + 4$；

$f(1) = 1$。

向上代回：

$f(2) = 1 + 4 = 5$；

$f(3) = 5 + 9 = 14$；

$f(4) = 14 + 16 = 30$。

所以 `f(n)` 实际上计算的是 $1^2 + 2^2 + \cdots + n^2$。但阅读程序时，不必一开始就看出这个整体意义。先展开、再代回，是更稳的办法。

四、先操作还是后操作

递归函数中，递归调用前后的语句顺序会影响输出。

看下面这段程序：

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

调用 `f(3)` 时，先输出 3，再调用 `f(2)`；`f(2)` 先输出 2，再调用 `f(1)`；`f(1)` 先输出 1，再调用 `f(0)`。

`f(0)` 直接返回后，程序回到 `f(1)` 中递归调用后面的语句，输出 1；再回到 `f(2)` 输出 2；最后回到 `f(3)` 输出 3。

所以输出是：

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

如果把递归调用前的输出删掉，只保留递归调用后的输出：

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

调用 g(3) 时，输出是：

```
1 2 3
```

因为所有输出都发生在递归返回之后。

递归阅读题中，最常见的错误就是把递归调用前后的语句混在一起。要明确区分：

递归调用前的语句，会在向下调用时执行；

递归调用后的语句，会在返回时执行。

如果一个函数中既有递归前操作，又有递归后操作，手算时最好写成“进入时做什么，返回时做什么”。

五、多个递归调用

有些递归函数一次会调用自己多次。最常见的例子是斐波那契递归：

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

要求 fib(5)，可以展开：

$f(5) = f(4) + f(3)$；

$f(4) = f(3) + f(2)$；

$f(3) = f(2) + f(1)$。

继续代入可得：

$f(3) = 1 + 1 = 2$；

$f(4) = 2 + 1 = 3$；

$f(5) = 3 + 2 = 5$。

这类函数与阶乘不同。阶乘每层只调用一次，形成一条链；斐波那契每层可能分成两个分支，形成一棵调用树。调用次数会明显增加。

阅读多个递归调用时，要注意两点。

第一，两个调用都会执行。不能只算 `fib(n - 1)`，漏掉 `fib(n - 2)`。

第二，相同子问题可能被重复计算。例如计算 `fib(5)` 时，`fib(3)` 会被多次计算。这就是朴素递归效率较低的原因。

CSP-J 第一轮中，多个递归调用常用来考输出顺序、返回值和复杂度。若题目只让手算小输入，按调用树展开即可；若题目问复杂度，要看每层分出多少次调用。

六、递归与递推

递归和递推很容易混淆。

递归通常是“从大问题出发”，需要时调用更小的问题。例如算 `fib(5)`，会去调用 `fib(4)` 和 `fib(3)`。

递推通常是“从小问题开始”，按顺序推出更大的结果。例如先算 f_1、f_2，再算 f_3、f_4、f_5。

斐波那契数列的递推写法如下：

```
int f[105];
f[1] = f[2] = 1;
for (int i = 3; i <= n; i++) {
    f[i] = f[i - 1] + f[i - 2];
}
cout << f[n];
```

递归写法更接近数学定义，代码短，但可能重复计算。递推写法按顺序保存结果，通常更容易控制时间复杂度。

第一轮中不要求把所有递归都改成递推，但要会判断两者区别。若程序中函数调用自身，就是递归。若程序用数组从小到大填值，一般是递推或动态规划的形式。

完善程序中，如果题目已经给出递推数组，就不要误填成递归调用；如果题目给出函数自己调用自己，就要按递归出口和递归关系补全。

七、递归与树结构

树和递归关系很密切。因为一棵树的子树仍然是树，所以很多树上问题适合用递归描述。

例如计算二叉树的节点数。若一棵树为空，节点数是 0；否则，节点数等于左子树节点数加右子树节点数再加 1。

用递归思想可以写成：

空树返回 0；

非空树返回左子树结果、右子树结果和根节点本身。

这类递归的结构与二叉树遍历类似。前序遍历是先处理根，再递归处理左子树和右子树；中序遍历是先递归左子树，再处理根，再递归右子树；后序遍历是先递归左右子树，最后处理根。

本章不展开树上算法，只强调一点：递归常用于处理“整体由若干个相同结构的部分组成”的问题。树的子树仍然是树，区间的左右两半仍然是区间，搜索的下一步仍然是同类状态。这些结构都适合递归。

八、递归复杂度的粗略判断

递归复杂度要看调用次数，而不是只看函数里有几行代码。

若每次只调用自己一次，并且参数每次减少 1，递归深度大约是 n，常见时间复杂度是 $O(n)$。例如 `f(n)` 调用 `f(n - 1)`，直到 $n = 0$。

若每次只调用自己一次，并且参数每次减半，递归深度大约是 $\log n$，常见时间复杂度是 $O(\log n)$。例如 `f(n)` 调用 `f(n / 2)`，直到 $n \le 1$。

若每次调用自己两次，并且规模都接近 $n - 1$，调用数量会增长很快，常见是指数级。例如朴素的 `fib(n)` 同时调用 `fib(n - 1)` 和 `fib(n - 2)`，会产生大量重复调用。第一轮中不一定要求写出精确表达式，但要知道它不是 $O(n)$。

若递归把问题分成两个大小为 $n/2$ 的子问题，并且每层还有 $O(n)$ 的合并工作，常见复杂度是 $O(n \log n)$。归并排序就是这种类型。第 14 章已经讲过主定理的轻量用法，本章只提醒：递归复杂度要看“分几支”和“每支规模怎样变小”。

空间复杂度方面，递归还要考虑调用深度。若递归深度是 n，就会有 $O(n)$ 级别的调用记录；若递归深度是 $\log n$，这部分通常是 $O(\log n)$。在第一轮中，粗略按递归层数判断即可。

九、完善程序中的递归空

完善递归程序时，空缺通常出现在四个位置。

第一，递归出口。例如 `if (n == 0) return 1;`，或者 `if (l > r) return;`。判断出口时，要看递归参数如何变化。

第二，递归参数。例如调用 `f(n - 1)`、`f(l, mid)`、`f(mid + 1, r)`。参数必须让问题规模变小，逐步接近出口。

第三，返回表达式。例如 `return f(n - 1) + f(n - 2);` 或 `return n * f(n - 1);`。要根据题意判断当前层怎样使用子问题结果。

第四，递归调用前后的操作。例如先输出再递归，还是先递归再输出。这个位置决定输出顺序。

补递归空时，不要只看空缺一行。要先写清函数含义。例如 `sum(n)` 表示什么？是 1 到 n 的和，还是数组前 n 项的和？`dfs(x)` 表示从哪个状态继续处理？函数含义明确后，出口和递归关系才容易判断。

如果函数名不明显，就从返回值和调用位置反推。主函数如何调用它？调用后结果用于什么？这些信息都能帮助判断空缺。

十、真题例题

真题例题 以下对递归方法的描述中，正确的是哪一项？

A. 递归是允许使用多组参数调用函数的编程技术。
B. 递归是通过调用自身来求解问题的编程技术。
C. 递归是面向对象和数据而不是功能和逻辑的编程语言模型。
D. 递归是将某种高级语言转换为机器代码的编程技术。

答案：B

解析：
递归的核心特征是函数直接或间接调用自身，并通过更小规模的同类问题求解原问题。因此 B 正确。

A 只是在描述函数可以带参数，和递归没有直接关系。一个函数即使有很多参数，只要它不调用自身，也不是递归。

C 描述的是某些编程思想或程序设计模型，与递归无关。

D 描述的是编译器的作用。把高级语言转换为机器能够执行或进一步处理的低层代码，是编译相关内容，不是递归。

这类概念题看似简单，但容易被“多组参数”“高级语言”等词干扰。判断递归，只抓住一点：是否调用自身。

十一、例题

例题 阅读下面程序，若输入为 4，输出是什么？

```cpp
#include <iostream>
using namespace std;

int f(int n) {
    if (n == 0) return 0;
    if (n % 2 == 0) return f(n - 1) + n;
    return f(n - 1) - n;
}

int main() {
    int n;
    cin >> n;
    cout << f(n) << endl;
    return 0;
}
```

A. -2
B. 0
C. 2
D. 4

答案：C

解析：
函数 `f(n)` 每次调用 `f(n - 1)`，直到 `n == 0` 时返回 0。返回过程中，如果当前 n 是偶数，就加上 n；如果当前 n 是奇数，就减去 n。

输入为 4，展开为：

$f(4) = f(3) + 4$；

$f(3) = f(2) - 3$；

$f(2) = f(1) + 2$；

$f(1) = f(0) - 1$；

$f(0) = 0$。

向上代回：

$f(1) = 0 - 1 = -1$；

$f(2) = -1 + 2 = 1$；

$f(3) = 1 - 3 = -2$；

$f(4) = -2 + 4 = 2$。

所以输出 2，选 C。

这道题的关键是递归返回时根据当前层的 n 判断奇偶。不能只看输入 4 是偶数，就直接认为结果是正数或等于 4。每一层都有自己的 n，每一层都会参与计算。

十二、变式题

变式题 若把上题函数中的出口改为下面形式，输入仍为 4，输出是什么？

```
if (n == 1) return 1;
```

其他代码不变。

A. 1
B. 3
C. 4
D. 5

答案：C

解析：
修改后，`f(1)` 不再继续调用 `f(0)`，而是直接返回 1。

输入为 4 时：

$f(4) = f(3) + 4$；

$f(3) = f(2) - 3$；

$f(2) = f(1) + 2$；

$f(1) = 1$。

向上代回：

$f(2) = 1 + 2 = 3$；

$f(3) = 3 - 3 = 0$；

$f(4) = 0 + 4 = 4$。

所以输出 4，选 C。

这道变式说明，递归出口改变后，整个返回链都会改变。即使递归关系没有变，只要最底层返回值不同，最终答案也可能不同。

十三、本章要点

递归是函数调用自身来求解问题的方法。递归程序必须有出口，并且递归调用要让问题规模逐步接近出口。

读递归程序时，先找函数含义，再找出口，最后看递归关系。若函数有返回值，就把调用展开到出口，再从出口向上代回。若函数有输出语句，要分清输出在递归调用前还是递归调用后。

一次递归调用形成一条调用链；多次递归调用会形成调用树。像阶乘这类单链递归通常容易追踪；像斐波那契这类多分支递归会出现重复计算。

递归和递推都能描述从小问题到大问题的关系。递归是函数调用自身，递推通常用循环和数组从小到大计算。阅读程序时，要按代码形式判断，不要只凭数学公式判断。

递归复杂度要看调用次数和递归层数。每次减 1 的单链递归常见为 $O(n)$；每次减半的单链递归常见为 $O(\log n)$；多分支且重复计算的递归可能达到指数级。

完善递归程序时，常见空缺在出口、递归参数、返回表达式和递归前后操作。补空前先说明函数代表什么，再让代码和这个含义对齐。递归题不怕层数多，怕的是出口、参数和返回方向没有想清楚。

0x1B 搜索

搜索是在一组可能状态中，按照一定规则不断扩展，直到找到目标或把所有可能情况都检查完。它和枚举关系很近：枚举常把候选对象直接列出来，搜索则常从一个起点出发，沿着可行的转移一步一步走到新的状态。

CSP-J 第一轮中的搜索题通常不会要求写很长的搜索程序，而是要求看懂搜索过程：从哪里开始，能走到哪里，哪些状态已经访问过，递归什么时候返回，队列中先处理谁，某个变量最后是多少。搜索章节的重点不是模板，而是状态、转移、边界、访问标记和搜索顺序。

本章只讲最基础的深度优先搜索和广度优先搜索。更复杂的剪枝、记忆化和图论算法不在本章展开。

一、搜索先看状态

搜索中的状态，就是程序当前所在的情况。状态可以是一个数、一个位置、一段区间，也可以是多个变量组成的一组信息。

例如在网格中从起点走到终点，状态可以是当前位置 (x, y)。

在一个图中从顶点 s 出发，状态可以是当前顶点 u。

在一个数值变化问题中，状态可以是当前数值 x。

搜索题的第一步是问：程序正在搜索什么？如果状态没有看清，后面的递归、队列、访问标记都会变得混乱。

例如有一个 $n \times m$ 的网格，`'.'` 表示可以走，`'#'` 表示障碍，要求从起点走到终点。这里状态就是“当前格子”。从一个格子出发，可以向上、下、左、右四个方向尝试移动。若新位置没有越界、不是障碍、没有访问过，就可以进入这个新状态。

搜索题通常有三个核心部分：

起点：从哪里开始搜索；

转移：从当前状态能到哪些新状态；

终止：什么时候停止搜索或返回。

读搜索程序时，不要先盯着代码细节。先用一句话说出状态含义，再去看转移规则。比如“`dfs(x, y)` 表示从格子 (x, y) 继续搜索”，或者“`bfs(s)` 表示从顶点 s 出发按层访问图中的点”。

二、访问标记

搜索中常见数组 `vis` 或 `visited`，表示某个状态是否已经访问过。它的作用是避免重复搜索。

在图或网格中，状态之间可能形成环。如果没有访问标记，程序可能在几个状态之间来回走，无法结束。

例如无向图中有边 A-B、B-C、C-A。从 A 出发，走到 B，再走到 C，又能回到 A。如果不记录 A 已经访问过，程序可能继续从 A 出发重复走同样的路。

常见写法是：

```
vis[x][y] = true;
```

或：

```
vis[u] = true;
```

访问标记要在合适的时刻设置。对 DFS 来说，通常在进入一个状态时立刻标记，防止后续再次进入。对 BFS 来说，通常在新状态入队时标记，防止同一个状态被多次加入队列。

这一点很容易出现在完善程序题中。若 BFS 等到出队时才标记，同一个格子可能被多个方向重复入队，虽然有时仍能得到答案，但队列会多出许多重复状态，程序逻辑也更难判断。基础写法中，发现新状态并准备入队时就标记，是更清楚的做法。

但并不是所有搜索都永久标记。有些回溯问题需要在离开当前分支时撤销标记，例如排列生成、路径枚举。是否撤销，要看题目问的是“访问一次即可”，还是“在不同方案中可以重新使用”。

三、深度优先搜索

深度优先搜索常简称 DFS。它的特点是：从当前状态出发，能往下走就先往下走，走不下去再退回上一步，尝试其他方向。

DFS 很适合用递归表示。因为从当前状态进入下一个状态，再从下一个状态继续搜索，形式相同。

下面是一段网格 DFS 的典型结构：

```
void dfs(int x, int y) {
    vis[x][y] = true;
    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;
        dfs(nx, ny);
    }
}
```

这段代码中，`dfs(x, y)` 表示从格子 (x, y) 出发，把能到达的格子都访问一遍。`dx` 和 `dy` 表示四个方向。每次得到新格子 (nx, ny) 后，先检查三件事：

是否越界；

是否是障碍；

是否已经访问过。

只有三项检查都通过，才递归进入新格子。

读 DFS 程序时，要注意搜索顺序。四个方向的排列不同，访问顺序就不同。如果题目问“输出访问顺序”，必须按代码中 `dx`、`dy` 的顺序模拟，不能按自己习惯随便走。

DFS 常用于判断能不能到达、统计连通块大小、遍历树或图。它不保证第一次到达某个点时步数最少。如果题目要求无权图中的最短步数，通常更适合用 BFS。

四、DFS 的返回

有些 DFS 只负责把所有能到达的状态标记出来，不需要返回值。有些 DFS 需要返回是否找到目标，或返回某个数量。

例如判断从当前格子能否走到终点，可以写成返回 `bool` 的形式：

```
bool dfs(int x, int y) {
    if (x == tx && y == ty) return true;
    vis[x][y] = true;
    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;
        if (dfs(nx, ny)) return true;
    }
    return false;
}
```

这里的出口有两个。

第一个出口是已经到达终点，直接返回 `true`。

第二个出口是在所有方向都试完后仍找不到终点，返回 `false`。

阅读这种程序时，要特别注意 `return true` 的位置。它出现在递归调用之后，表示“只要某个方向能找到终点，当前函数也返回找到”。如果把它写成只调用 `dfs(nx, ny);` 而不使用返回值，外层函数就无法知道深处已经找到终点。

DFS 返回值题常考这类传递关系。发现目标后，结果要一层一层传回起点。若某一层没有把结果继续返回，程序就会丢失搜索结果。

五、回溯

回溯是一类特殊的 DFS。它常用于枚举方案：选择一个对象，进入下一层；这一分支结束后，撤销选择，再尝试其他选择。

例如从 1 到 n 中选若干个数，或者生成一个排列，都可以用回溯。

回溯中常出现这样的结构：

先标记当前选择；

递归进入下一层；

递归返回后撤销当前选择。

撤销这一步非常重要。因为当前选择只属于这一条分支，不能影响其他分支。

例如生成 1 到 n 的排列时，`used[i]` 表示数字 i 是否已经被当前排列使用。选择 i 后要设为已用；递归返回后，要把它改回未用，让其他位置或其他分支还能选择它。

如果忘记撤销标记，后面的分支会误以为某些数字已经不能使用，方案数会变少。若不该撤销却撤销了，则可能重复访问状态，甚至导致程序在图中反复走。

判断是否需要回溯，可以问：当前标记是否只对当前方案有效？如果是，离开当前分支时通常要撤销。如果一个状态在整次搜索中只需要访问一次，比如图的连通性搜索，则通常不撤销。

六、广度优先搜索

广度优先搜索常简称 BFS。它的特点是：从起点开始，先访问距离起点为 1 步的状态，再访问距离为 2 步的状态，依次一层一层扩展。

BFS 通常使用队列。队列先进先出，先发现的状态先处理，这正好符合按层扩展的顺序。

网格 BFS 的基本过程是：

把起点入队，并标记已访问；

取出队首状态；

检查它能到达的相邻状态；

把新发现的状态入队，并记录距离；

重复直到队列为空或找到目标。

与 DFS 不同，BFS 在无权图或普通网格中，第一次到达某个状态时，通常就是从起点到它的最短步数。这里的 “无权” 指每一步代价相同，例如每次上下左右移动一步。

如果题目问 “能否到达”，DFS 和 BFS 都可以；如果题目问 “最少需要几步”，BFS 更常用。

七、BFS 的距离

BFS 中常用数组 `dist` 记录距离。起点距离为 0，从某个点扩展到相邻点时，相邻点距离等于当前点距离加 1。

例如从起点 (sx, sy) 出发，可以写作：

```
queue<pair<int, int> > q;
q.push(make_pair(sx, sy));
vis[sx][sy] = true;
dist[sx][sy] = 0;

while (!q.empty()) {
    int x = q.front().first;
    int y = q.front().second;
    q.pop();

    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;
        dist[nx][ny] = dist[x][y] + 1;
        q.push(make_pair(nx, ny));
    }
}
```

读这段程序时，可以抓住四个动作：取队首，弹队首，检查相邻状态，合法则入队。

`dist[nx][ny] = dist[x][y] + 1` 表示从当前位置走一步到新位置，所以距离增加 1。这条语句通常出现在入队之前或入队附近，因为新状态第一次被发现时，就能确定它的最短距离。

若题目问某个格子的最短步数，就要看它第一次被赋值的 `dist`。不要等到它再次被其他路径尝试到达，因为基础 BFS 中已经用 `vis` 阻止重复访问。

八、DFS 和 BFS 的区别

DFS 和 BFS 都是在图或状态空间中搜索，但顺序不同。

DFS 沿着一条路径尽量深入，走不下去再回退。它常用递归或栈实现。它适合判断连通、枚举方案、遍历树结构。

BFS 一层一层扩展，先访问离起点近的状态。它常用队列实现。它适合求无权图中的最短步数。

例如从一个格子出发，四个方向都能走。DFS 可能先一直向上走，走到不能走再换方向；BFS 会先访问一步能到的格子，再访问两步能到的格子。

做选择题时，可以记住：

看到递归、回溯、不断深入，常与 DFS 有关；

看到队列、按层、最短步数，常与 BFS 有关。

但不要机械判断。若题目给出具体代码，就按代码实际结构分析。使用栈的迭代写法也可以实现 DFS；使用队列通常对应 BFS。

九、搜索中的边界检查

搜索程序中，边界检查很重要。尤其是网格题，如果先访问数组再判断越界，就可能访问非法位置。

例如应先判断：

```
if (nx < 1 || nx > n || ny < 1 || ny > m) continue;
```

再判断：

```
if (a[nx][ny] == '#') continue;
```

如果顺序反过来，先访问 a[nx][ny]，而 nx 或 ny 已经越界，程序就可能出错。

边界还包括状态是否允许进入。网格中的障碍、图中的不可走边、已经访问过的状态，都应在进入新状态前检查。

完善程序中，搜索空缺常出现在这些位置：

新坐标如何计算；

越界条件是什么；

是否是障碍；

是否已经访问；

访问标记何时设置；

是否把新状态入队或递归进入。

判断空缺时，要先看它是在“生成新状态”还是“检查新状态”。不要把方向数组、边界条件、访问标记混在一起。

十、例题

例题 阅读下面程序片段。若图中有 5 个顶点，边为 $1-2$，$1-3$，$2-4$，$3-5$，并且从 1 开始 BFS，邻接点按编号从小到大入队，访问顺序是什么？

- **A.** $1,2,3,4,5$
- **B.** $1,3,2,5,4$
- **C.** $1,2,4,3,5$
- **D.** $1,4,2,5,3$

答案：A

解析：
从顶点 1 开始。先访问 1，它的邻接点按编号从小到大为 $2,3$，所以把 $2,3$ 入队。

队列中先出的是 2。访问 2 后，它能到达 4，把 4 入队。此时队列中是 $3,4$。

接着访问 3。它能到达 5，把 5 入队。此时队列中是 $4,5$。

接着访问 4，再访问 5。

所以访问顺序为 $1, 2, 3, 4, 5$，选 A。

这道题的关键是 BFS 使用队列，先入队的顶点先处理。虽然 4 是从 2 扩展出来的，但在访问 4 之前，队列中已经有更早入队的 3，所以要先访问 3。

十一、变式题

变式题 若仍在上题的图中从 1 开始 DFS，邻接点按编号从小到大尝试，访问顺序是什么？

- **A.** $1, 2, 3, 4, 5$
- **B.** $1, 2, 4, 3, 5$
- **C.** $1, 3, 5, 2, 4$
- **D.** $1, 4, 2, 3, 5$

答案：B

解析：
DFS 从 1 出发，按编号从小到大尝试邻接点。

先访问 1。它的邻接点为 $2, 3$，先走 2。

访问 2 后，它还能走到 4，于是继续深入访问 4。4 没有新的未访问邻接点，返回 2，再返回 1。

回到 1 后，再尝试另一个未访问邻接点 3。访问 3 后继续访问 5。

因此 DFS 访问顺序是 $1, 2, 4, 3, 5$，选 B。

这道题和例题使用同一个图，但搜索顺序不同。BFS 按层访问，所以先访问 $2, 3$；DFS 先沿 $1 \to 2 \to 4$ 深入，再回来访问 $3, 5$。同一张图，换一种搜索方式，访问顺序可能不同。

十二、易错点

第一，搜索前要先明确状态。网格题状态常是坐标，图题状态常是顶点，数值变化题状态常是当前数值。

第二，DFS 和 BFS 的访问顺序不同。DFS 先深入，BFS 按层扩展。若题目问访问顺序，必须按代码中的方向或邻接点顺序模拟。

第三，BFS 求最短步数的前提是每一步代价相同。若不同边代价不同，普通 BFS 不能直接保证最短代价。

第四，访问标记的时机要看清。DFS 常在进入状态时标记，BFS 常在入队时标记。回溯问题中，某些标记还需要在递归返回后撤销。

第五，边界检查要在访问数组前完成。先判断新位置是否合法，再访问数组内容。

第六，`return` 和 `continue` 的作用不同。`continue` 只是跳过当前方向，继续试下一个方向；`return` 会结束当前函数调用。搜索程序中看错这两个语句，会导致流程判断错误。

十三、本章要点

搜索是在状态之间不断扩展。读搜索程序时，先确定状态、起点、转移和终止条件。

DFS 深度优先，常用递归或栈实现。它适合遍历、连通性判断和方案枚举。递归调用前后的操作顺序会影响搜索结果和输出顺序。

BFS 广度优先，常用队列实现。它按层扩展，在每一步代价相同的图或网格中，第一次到达某个状态时通常得到最短步数。

访问标记用于避免重复访问。图和网格中若没有 `vis`，程序可能在环中反复走。回溯中有时需要撤销标记，图的普通遍历中通常不撤销。

搜索题的易错点集中在方向顺序、边界条件、访问标记和返回值传递。做题时不要只记 DFS、BFS 名字，而要按当前队列、当前递归层和当前状态逐步追踪。

0x1C 简单动态规划

动态规划常简称 DP。对 CSP-J 第一轮来说，本章不需要把动态规划讲成复赛算法专题，更不需要堆很多模板。第一轮更常考的是：给出一段程序，要求判断 `dp[i]` 表示什么，某个初值是否正确，转移式中 `min`、`max` 或加法的含义是什么，循环顺序是否会影响结果。

在进入动态规划之前，要先补一个更基础的概念：递推。递推就是从已经算出的结果推出后面的结果。很多简单 DP 题，本质上就是“带有明确状态含义的递推”。如果学生看不懂递推式，后面看到 `dp[i]`、`dp[i][j]` 就会只会背模板，不会读程序。

本章先讲递推，再讲简单动态规划。递推解决“后一个值怎么由前面的值得到”，动态规划进一步要求说清“数组中的每个状态表示什么”。

一、递推先看顺序

递推的基本形式是：先给出前几个值，再按规则推出后面的值。

例如斐波那契数列：

$f_1 = 1$，$f_2 = 1$，当 $i \geq 3$ 时，$f_i = f_{i-1} + f_{i-2}$。

这句话翻译成程序，就是先设置初值，再从小到大计算：

```
int f[105];
f[1] = 1;
f[2] = 1;
for (int i = 3; i <= n; i++) {
    f[i] = f[i - 1] + f[i - 2];
}
cout << f[n];
```

递推程序最重要的不是公式本身，而是顺序。计算 `f[i]` 时，`f[i - 1]` 和 `f[i - 2]` 必须已经算好。因此循环要从小到大。

如果把循环写成从大到小：

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

当程序计算 `f[n]` 时，`f[n - 1]` 和 `f[n - 2]` 还没有按照递推规则算出来，结果就不可靠。

阅读递推程序时，可以先做三件事：

先找初值，看最开始哪些位置被直接赋值；

再找递推式，看当前位置用到了哪些旧位置；

最后看循环方向，判断旧位置在使用前是否已经算好。

这三个问题弄清楚，很多一维 dp 程序就能继续读下去。

二、递推表怎么手算

第一轮考试中，递推常以程序阅读形式出现。题目不会让学生计算很大的 n，而是让学生根据程序手算几个值，或者判断修改某一行后结果会不会变。

例如：

```
int f[10];
f[0] = 1;
f[1] = 2;
for (int i = 2; i <= 6; i++) {
    f[i] = f[i - 1] + 2 * f[i - 2];
}
```

手算时不要在脑中跳步，最好列出短表：

i	f[i]
0	1
1	2
2	$2+2\times1=4$
3	$4+2\times2=8$
4	$8+2\times4=16$
5	$16+2\times8=32$
6	$32+2\times16=64$

表不必很大，但要写清楚每一项从哪里来。尤其是出现 `i - 1`、`i - 2`、`i + 1` 这类下标时，学生很容易把当前项和上一项看反。

递推题还有一个常见陷阱：数组下标从 0 开始还是从 1 开始。

如果程序写的是：

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

那么输出 `f[5]` 得到的是第 0 项起算下的结果。若题目把第 1 项称为 1，就要小心题意和程序下标是否一致。

做题时不要只记“斐波那契第五项是多少”，而要按程序中的下标重新算。

三、从递推到动态规划

递推只强调“怎样由前面的值推出后面的值”。动态规划还要强调“每个值表示什么”。

例如每次可以爬 1 级或 2 级台阶，问爬到第 n 级有多少种方法。

可以设 `dp[i]` 表示到达第 i 级台阶的方法数。为了到达第 i 级，最后一步可能从第 $i-1$ 级爬 1 级，也可能从第 $i-2$ 级爬 2 级。因此：

$dp_i = dp_{i-1} + dp_{i-2}$。

程序可以写成：

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

这段代码和普通递推很像，但多了一层含义：`dp[i]` 不是一个普通数列的第 i 项，而是“到达第 i 级的方法数”。状态含义一旦确定，初值和转移都要围绕它解释。

动态规划题通常有四个部分：

名称	要问的问题
状态	`dp[i]` 或 `dp[i][j]` 表示什么
初值	最小规模的问题答案是多少
转移	当前状态由哪些旧状态推出
答案	最后输出哪个状态

第一轮中，很多题并不要求学生独立设计 DP，而是要求读懂已经写好的程序。所以看 DP 程序时，第一句话应是：这个 `dp` 数组每个位置表示什么？

四、初值不能靠背

DP 需要从最小规模的问题开始。初值错了，后面所有状态都会跟着错。

仍以爬楼梯为例。若设 `dp[i]` 表示到达第 i 级的方法数，那么：

到达第 1 级，只有一种方法：爬 1 级，所以 $dp_1 = 1$；

到达第 2 级，有两种方法：$1+1$，或者 2，所以 $dp_2 = 2$。

因此可以写：

```
dp[1] = 1;
dp[2] = 2;
```

但有些程序会设 `dp[0] = 1`。这不一定错。若 `dp[i]` 表示从起点走到位置 i 的方法数，那么 `dp[0] = 1` 可以理解为“从起点到起点有一种空方案”。

例如：

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

这里也能得到爬楼梯的方法数。它和 `dp[1] = 1, dp[2] = 2` 的写法不同，但状态含义和转移是配套的。

完善程序题很喜欢考初值。判断初值时，不要只看它像不像模板，而要回到状态含义：最小规模时答案到底是多少？

五、转移看最后一步

转移式通常来自“最后一步”或“最后一次选择”。

例如有 n 个数，每次可以选若干个数，但不能选相邻位置，问最大和是多少。

设 `dp[i]` 表示只考虑前 i 个数时，能得到的最大和。对于第 i 个数，有两种情况：

不选第 i 个数，答案是 `dp[i - 1]`；

选第 i 个数，那么第 $i-1$ 个数不能选，答案是 `dp[i - 2] + a[i]`。

所以：

$dp_i = \max(dp_{i-1}, dp_{i-2} + a_i)$。

代码可以写成：

```
dp[1] = a[1];
dp[2] = max(a[1], a[2]);
for (int i = 3; i <= n; i++) {
    dp[i] = max(dp[i - 1], dp[i - 2] + a[i]);
}
```

这段程序里，`max` 的含义是“选第 i 个数”和“不选第 i 个数”两种情况取较大值。若题目问的是方法数，常用加法；若题目问最大值，常用 `max`；若题目问最小代价，常用 `min`。

看到转移式时，可以把它翻译成中文。只要能说出每一项代表哪种情况，就不容易把公式背错。

六、最小代价模型

CSP-J 第一轮中，最小代价类 DP 很适合做阅读程序题。它通常给出一个数组 `cost`，再用 `dp` 保存到达某个位置的最小代价。

例如：

```
int compute(int cost[], int n) {
    int dp[105];
    dp[0] = cost[0];
    dp[1] = cost[1];
    for (int i = 2; i < n; i++) {
        dp[i] = min(dp[i - 1], dp[i - 2]) + cost[i];
    }
    return min(dp[n - 1], dp[n - 2]);
}
```

这段程序可以理解为：到达第 i 个位置前，可能来自第 $i-1$ 个位置，也可能来自第 $i-2$ 个位置。为了让总代价最小，先取这两个来源中代价较小的一个，再加上当前位置的代价 `cost[i]`。

注意最后返回的是：

```
return min(dp[n - 1], dp[n - 2]);
```

这表示终点可以从最后一个位置或倒数第二个位置到达，最后一步不再额外付代价。若题目把终点定义成必须站在最后一个位置上，那么答案就可能是 `dp[n - 1]`。所以返回值要看题意，不能只看数组末尾。

这类题常问：

输入某个数组时输出多少；

把 `min` 改成 `max` 会怎样；

把 `dp[i - 2]` 改成 `dp[i - 3]` 是否会越界；

最后返回 `dp[n]`、`dp[n - 1]` 还是 `min(dp[n], dp[n - 1])`。

做这种题时，先写出 `dp` 表的前几项，比直接心算更稳。

七、二维 DP 表

有些 DP 需要两个维度。二维 DP 的难点不是数组大，而是两个下标分别表示什么。

例如从左上角走到右下角的网格计数问题，每次只能向右或向下走。设 `dp[i][j]` 表示到达第 i 行第 j 列的方法数。

若当前位置不在第一行或第一列，那么它只能从上方或左方到达：

$dp_{ij} = dp_{i-1,j} + dp_{i,j-1}$。

代码可以写成：

```
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];
        }
    }
```

第一行只能一直向右走到达，所以方法数为 1；第一列只能一直向下走到达，所以方法数也为 1。中间格子由上方和左方相加。

二维 DP 阅读题中，最容易错的是下标含义。例如 `dp[i][j]` 可能表示第 i 行第 j 列，也可能表示第一个字符串前 i 个字符和第二个字符串前 j 个字符，还可能表示前 i 个物品、容量为 j。不能只看到二维数组就套同一种解释。

八、字符串中的 DP

字符串题中常见的 DP，是用 `dp[i][j]` 表示两个前缀之间的关系。

例如最长公共子序列问题。给定字符串 `a` 和 `b`，设 `dp[i][j]` 表示 `a` 的前 i 个字符和 `b` 的前 j 个字符的最长公共子序列长度。

如果 `a[i - 1] == b[j - 1]`，说明两个前缀的最后一个字符可以一起使用：

```
dp[i][j] = dp[i - 1][j - 1] + 1;
```

否则，最后一个字符不能同时使用，只能丢掉其中一个前缀的末尾字符：

```
dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
```

完整片段如下：

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

这里有一个常见下标点：字符串下标从 0 开始，所以第 i 个字符写作 `a[i - 1]`。而 `dp[i][j]` 中的 i 和 j 表示“前 i 个”“前 j 个”。这两个下标体系放在同一段程序里，学生很容易看错。

如果题目把 `a[i - 1]` 改成 `a[i]`，就可能访问到错误位置；如果把 `dp[i - 1][j - 1]` 改成 `dp[i][j - 1]`，状态含义也会变。

九、完善程序怎么补 DP

DP 完善程序的空缺常出现在四类位置。

第一类是初始化。比如第一行、第一列、`dp[0]`、`dp[1]`。补初始化时，要问最小规模的答案是什么。

第二类是判断条件。字符串 DP 中，常见条件是：

```
a[i - 1] == b[j - 1]
```

这里不能写成 `a[i] == b[j]`，因为字符串下标和前缀长度差 1。

第三类是转移式。计数题通常把不同来源相加；最值题通常用 `min` 或 `max`；编辑距离这类题还会同时比较插入、删除、替换三种操作。

第四类是返回值。程序可能返回 `dp[n]`，也可能返回 `dp[m][n]`，还可能返回两个状态的较小值。返回什么，取决于状态含义和题目问的答案。

补 DP 程序时，不要先看选项中哪一项最像模板。更可靠的顺序是：

先读题意，确认程序要解决什么问题；

再读变量名和数组含义，判断状态表示什么；

然后看循环范围，确认下标从哪里开始；

最后用一个小样例手算，检查空缺填入后能否得到合理结果。

十、例题

例题：阅读下面程序，若输入为 `5`，程序输出什么？

```cpp
#include <iostream>
using namespace std;

int main() {
    int n;
    int dp[105];
    cin >> n;
    dp[0] = 1;
    dp[1] = 1;
    for (int i = 2; i <= n; i++) {
        dp[i] = dp[i - 1] + dp[i - 2];
    }
    cout << dp[n] << endl;
    return 0;
}
```

A. `3`

B. `5`

C. `8`

D. `13`

答案：C

解析：程序中 `dp[0] = 1`，`dp[1] = 1`。从 $i = 2$ 开始递推：

`dp[2] = dp[1] + dp[0] = 2`；

`dp[3] = dp[2] + dp[1] = 3`；

`dp[4] = dp[3] + dp[2] = 5`；

`dp[5] = dp[4] + dp[3] = 8`。

所以输出 `8`。这道题的关键是看清程序从 `dp[0]` 开始定义，而不是直接套“第 5 项斐波那契数”的口头记忆。

变式：如果把循环条件改成 `i < n`，其他代码不变，输入仍为 `5`，输出是什么？

A. `3`
B. `5`
C. `8`
D. 结果不确定

答案：D

解析：循环条件变成 `i < n` 后，最后一次执行是 $i = 4$，不会计算 `dp[5]`。由于 `dp[5]` 没有在程序中被正确赋值，这段程序严格来说依赖未初始化数组值，结果不可靠。`dp[5]` 未初始化，结果不确定。

十一、常见错误

第一，把递推式当成孤立公式。

`dp[i] = dp[i - 1] + dp[i - 2]` 只是形式。它可能表示斐波那契数列，也可能表示爬楼梯方法数，还可能表示其他计数问题。做题时要结合状态含义解释。

第二，初值和状态不配套。

`dp[0] = 1` 有时正确，有时不正确。它是否正确，取决于 `dp[0]` 代表什么。若它表示空方案，通常可以是 1；若它表示某个不存在的位置答案，就要重新判断。

第三，看错字符串下标。

`dp[i][j]` 表示前 i 个和前 j 个字符时，当前字符通常是 `s[i - 1]` 和 `t[j - 1]`。这里的 $i - 1$ 不是随便减的，而是因为字符串下标从 0 开始。

第四，循环方向不看转移依赖。

递推和 DP 都要求先算旧状态，再算新状态。若当前状态依赖 `dp[i - 1]`，循环通常要从小到大；若程序使用一维数组压缩二维 DP，循环方向还可能影响是否重复使用同一轮状态。第一轮只需先掌握：使用某个状态前，它必须已经按题意算好。

第五，把 DP 和贪心混在一起。

贪心通常按一个规则直接做选择，较少保存完整状态表；DP 会保存多个子问题答案，再由它们推出新答案。看到 `dp` 数组、初值和转移式，一般应先按动态规划阅读。

本章只要求掌握最基本的递推和 DP 阅读方法：先看状态，后看初值，再看转移和答案。只要能把 `dp[i]` 或 `dp[i][j]` 翻译成一句清楚的话，大多数 CSP-J 第一轮中的简单 DP 题就不会只剩下猜选项。

十二、本章要点

递推和动态规划都不是先背公式，而是先看“旧结果怎样推出新结果”。递推更关注计算顺序，动态规划还要进一步说清状态含义。看到 `dp` 数组时，第一步不是套模板，而是把 `dp[i]` 或 `dp[i][j]` 翻译成一句完整的话。

一维递推常见形式是当前位置由前一项或前两项推出。例如 `dp[i] = dp[i - 1] + dp[i - 2]`，可能表示数列，也可能表示方案数。判断它的意义，要看题目和初值，而不是只看公式长得像什么。

动态规划题通常按四步阅读：先看状态含义，再看初值，再看转移式，最后看返回值。初值和状态必须配套；转移式中的 `+`、`min`、`max` 分别对应不同含义。方案数问题常把来源相加，最大值问题常用 `max`，最小代价问题常用 `min`。

二维 DP 的关键是弄清两个下标。`dp[i][j]` 可能表示第 i 行第 j 列，也可能表示两个字符串前 i 个字符和前 j 个字符之间的关系。字符串题中尤其要注意，前 i 个字符的最后一个字符下标通常是 `i - 1`。

完善程序中，DP 的空缺常出现在初始化、判断条件、转移式和返回值处。补空时不要只看哪个选项像模板，要回到题意：最小规模的答案是什么，当前位置从哪些旧位置来，最后题目问的是哪一个状态。

阅读 DP 程序时，可以用一个很小的输入手算前几项。只要前几项能解释清楚，循环边界、数组下标、返回位置中的错误就更容易被发现。

0x1D 字符串处理

字符串处理是 CSP-J 第一轮中的常见内容。它可能出现在选择题中，考查字符串长度、下标、字符与字符串的区别、字典序比较；也可能出现在阅读程序和完善程序中，要求学生追踪字符串拼接、截取、查找、翻转或统计后的结果。

字符串题的难点通常不在代码长，而在细节密。一个长度为 n 的字符串，合法下标通常是 0 到 $n-1$；`substr(pos, len)` 的第二个参数是长度，不是结束下标；`cin >> s` 读不到空格后的内容；C 风格字符串还要考虑结尾的 `'\0'`。这些细节都很适合第一轮考试。

本章只讲 CSP-J 第一轮需要掌握的字符串处理。更复杂的字符串算法不在本章展开。

一、字符和字符串

字符串可以理解为一串字符。比如 `csp` 由三个字符组成：`'c'`、`'s'`、`'p'`。

在 C++ 中，单个字符通常使用 `char` 类型，用单引号表示，例如 `'a'`、`'0'`、`'A'`。字符串通常使用 `string` 类型，用双引号表示，例如 `"a"`、`"abc"`。

注意，`'a'` 和 `"a"` 不一样。`'a'` 是一个字符，`"a"` 是长度为 1 的字符串。选择题中如果问类型，这一点要分清。

字符串的长度是其中字符的个数。例如 `"hello"` 的长度是 5。空字符串 `""` 的长度是 0。

在 `string` 中，可以用 `s.size()` 或 `s.length()` 得到字符串长度。二者在常见使用中都表示字符串中字符的个数。

```
string s = "hello";
cout << s.size() << endl;
cout << s.length() << endl;
```

这段程序会输出两个 5。

字符串题中，长度和下标经常一起出现。长度是 n，最后一个字符的下标不是 n，而是 $n-1$。这是很多阅读程序题的第一个陷阱。

二、下标从 0 开始

C++ 的 `string` 使用从 0 开始的下标。若字符串 `s = "abcde"`，那么：

第 0 个下标处是字符 `'a'`；

第 1 个下标处是字符 `'b'`；

第 2 个下标处是字符 `'c'`；

第 3 个下标处是字符 `'d'`；

第 4 个下标处是字符 `'e'`。

字符串长度是 5，合法下标是 0 到 4。

可以用 `s[i]` 访问下标为 i 的字符。例如：

```
string s = "abcde";
for (int i = 0; i < s.size(); i++) {
    cout << s[i] << " ";
}
```

输出是：

```
a b c d e
```

这里循环条件是 `i < s.size()`，不是 `i <= s.size()`。如果写成 `i <= s.size()`，当 `i` 等于字符串长度时，就会访问越界位置。

纸面做题时，可以先在字符串下面标下标：

```
字符：a b c d e
下标：0 1 2 3 4
```

这样能避免把“第几个字符”和“下标是多少”混在一起。题目如果说“第 1 个字符”，通常是自然语言中的第一个字符；代码中的 `s[1]` 则表示下标为 1 的字符，也就是第二个字符。

三、读入字符串

字符串读入方式会影响是否能读到空格。

使用 `cin >> s` 读入字符串时，会跳过前面的空白字符，并从第一个非空白字符开始读，直到遇到空白字符停止。这里的空白字符包括空格、换行等。

例如输入为：

```
hello world
```

执行：

```
string s;
cin >> s;
```

读入的 `s` 是 `"hello"`，不是 `"hello world"`。

如果要读入整行，包括中间空格，可以使用 `getline(cin, s)`。它会读入一整行内容，直到换行结束。

需要注意的是，若前面刚用 `cin >> x` 读过数字，输入缓冲区中可能还留着换行。此时紧接着使用 `getline`，可能会先读到一个空行。CSP-J 第一轮通常不会把这个点考得很深，但阅读程序时如果看到 `cin >>` 和 `getline` 混用，要注意读入顺序。

选择题中常问“下面哪种写法能读入带空格的一整行字符串”。此时应想到 `getline`，而不是普通的 `cin >> s`。

四、拼接和修改

`string` 可以使用 `+` 拼接，也可以使用 `+=` 在原字符串后追加内容。

例如：

```
string a = "cs";
string b = "p";
string c = a + b;
a += b;
cout << c << " " << a << endl;
```

输出是：

```
csp csp
```

这里 `c = a + b` 会得到一个新字符串；`a += b` 会直接修改 `a`。

也可以修改某个位置的字符：

```
string s = "abc";
s[1] = 'x';
cout << s << endl;
```

输出是：

```
axc
```

因为 `s[1]` 原来是 `'b'`，被改成了 `'x'`。

阅读程序时，要区分“生成新字符串”和“修改原字符串”。例如 `t = s + "a"` 不会改变 `s`；而 `s += "a"` 会改变 `s`。如果后面继续使用 `s`，结果可能不同。

字符串也可以用循环逐个追加字符。例如把字符串反过来存入另一个字符串：

```
string s = "abc", t = "";
for (int i = s.size() - 1; i >= 0; i--) {
    t += s[i];
}
cout << t << endl;
```

输出是：

```
cba
```

这里 `t += s[i]` 表示把当前字符追加到 `t` 的末尾。

这段代码默认 `s` 非空。如果 `s` 可能为空，`s.size() - 1` 这类写法要格外小心。第一轮题目若保证字符串非空，就按题面保证处理。

五、子串

子串是字符串中连续的一段字符。例如 `"abcde"` 的子串包括 `"a"`、`"bc"`、`"cde"`、`"abcde"`，也包括空串。

在 C++ 的 `string` 中，常用 `substr(pos, len)` 取得子串。它表示从下标 `pos` 开始，取长度为 `len` 的一段。

例如：

```cpp
string s = "abcdef";
cout << s.substr(1, 3) << endl;
```

输出是：

```
bcd
```

因为下标 1 的字符是 `'b'`，从这里开始取长度为 3 的子串，得到 `"bcd"`。

这里第二个参数是长度，不是结束下标。`s.substr(1, 3)` 取的是下标 $1, 2, 3$ 三个字符；它不是“从下标 1 取到下标 3”这种写法。这个例子里看起来刚好对应，但换成其他数字就容易出错。

例如：

```cpp
string s = "abcdef";
cout << s.substr(2, 2) << endl;
```

输出是：

```
cd
```

不是 `"cde"`。因为从下标 2 开始，只取 2 个字符。

如果省略第二个参数，`substr(pos)` 会从 `pos` 一直取到字符串末尾。例如 `"abcdef"` 的 `substr(3)` 是 `"def"`。

完善程序中，若题目要求截取从第 l 个到第 r 个字符的一段，并且题目中的位置从 1 开始，那么转换到 C++ 下标时，起点应是 $l-1$，长度应是 $r-l+1$。这类题一定要把“位置”和“下标”分清。

六、查找字符和子串

string 可以使用 find 查找字符或子串第一次出现的位置。

例如：

```
string s = "ababc";
cout << s.find("ab") << endl;
cout << s.find("bc") << endl;
```

"ab" 第一次出现在下标 0，"bc" 第一次出现在下标 3。

如果找不到，find 会返回一个特殊值 string::npos。判断是否找到时，常写成：

```
if (s.find("abc") != string::npos) {
    cout << "YES";
}
```

不要把找不到时的返回值固定当成 −1。正确写法是与 string::npos 比较。

find 只返回第一次出现的位置。如果要统计某个字符出现多少次，通常用循环逐个检查更清楚。

例如统计字符 'a' 出现次数：

```
int cnt = 0;
for (int i = 0; i < s.size(); i++) {
    if (s[i] == 'a') cnt++;
}
```

这类统计程序在第一轮中很常见。关键是看清统计的是字符还是子串，是统计全部出现次数，还是只判断是否出现过。

七、字典序

字符串可以按字典序比较。简单理解，就是从左到右逐个字符比较，第一次不同的位置决定大小。

例如 "abc" 小于 "abd"，因为前两个字符相同，到第三个字符时 'c' 小于 'd'。

"abc" 小于 "abcd"，因为前三个字符都相同，但 "abc" 更短。较短的字符串是较长字符串的前缀时，短的更小。

在 C++ 中，可以直接用 <、>、== 比较 string：

```
string a = "abc";
string b = "abd";
cout << (a < b) << endl;
```

输出是 1，表示 a < b 为真。

字典序不是按字符串长度优先比较。比如 "9" 和 "10" 比较时，先比较第一个字符：'9' 大于 '1'，所以 "9" 的字典序大于 "10"。如果把它们当作数字，$9 < 10$；但当作字符串比较，结果不同。

这也是第一轮选择题的常见陷阱。看到字符串比较时，要按字符序列比较，不要自动转换成整数。

字符本身也有编码顺序。在常见字符集中，数字字符 '0' 到 '9' 连续排列，大写字母 'A' 到 'Z' 连续排列，小写字母 'a' 到 'z' 连续排列。第一轮通常只需要知道相同类别内部可以按顺序比较，不需要记复杂编码表。

八、C 风格字符串

除了 string，C++ 中还可能出现 C 风格字符串，也就是字符数组。

例如：

```
char s[10] = "abc";
```

这个字符数组中，不只存了 'a'、'b'、'c'，还会在后面存一个特殊字符 '\0'，表示字符串结束。

因此，字符串 "abc" 的长度是 3，但至少需要 4 个字符空间来存放：'a'、'b'、'c' 和 '\0'。

strlen(s) 返回的是 '\0' 之前的字符个数，不包括 '\0' 本身。

```
char s[10] = "abc";
cout << strlen(s) << endl;
```

输出是 3。

这点和数组大小不同。s 的数组大小是 10，但当前字符串长度是 3。选择题如果问 strlen(s)，不是问数组开了多大。

C 风格字符串常用函数包括 strlen、strcmp、strcpy 等。第一轮中不需要深入掌握每个函数的复杂细节，但要知道：

strlen 求字符串长度；

strcmp 比较两个字符串；

strcpy 把一个字符串复制到另一个字符数组中。

使用 C 风格字符串时，空间必须足够放下所有字符和结尾的 '\0'。若空间不够，就可能出错。纸面题中常用这一点考数组长度判断。

九、字符判断与转换

字符本质上可以参与比较和简单运算。常见题目会要求统计大写字母、小写字母、数字字符，或者把数字字符转成数字。

判断小写字母可以写：

```
if ('a' <= c && c <= 'z') {
    // c 是小写字母
}
```

判断大写字母可以写：

```
if ('A' <= c && c <= 'Z') {
    // c 是大写字母
}
```

判断数字字符可以写：

```
if ('0' <= c && c <= '9') {
    // c 是数字字符
}
```

数字字符和数字值不同。字符 `'7'` 不是整数 7。若要把数字字符转成对应整数，可以用：

```
int x = c - '0';
```

例如 `c` 为 `'7'` 时，`c - '0'` 的结果是 7。

小写字母转大写字母，常见写法是：

```
c = c - 'a' + 'A';
```

大写字母转小写字母，常见写法是：

```
c = c - 'A' + 'a';
```

这类写法依赖同类字母连续排列。CSP-J 第一轮中，通常默认在常见字符编码下处理这些基础字符。

阅读程序时，要分清字符和字符串。`c - '0'` 适用于字符；如果 `s` 是字符串，要访问某一位，如 `s[i] - '0'`。

十、例题

例题 阅读下面程序，若输入为 abacaba，输出是什么？

```
#include <iostream>
#include <string>
using namespace std;

int main() {
    string s, t = "";
    cin >> s;
    for (int i = 0; i < s.size(); i++) {
        if (i % 2 == 1) {
            t += s[i];
        }
    }
    cout << t << endl;
    return 0;
}
```

A. `aaa`

B. `bcb`

C. `aca`

D. `bab`

答案：B

解析：

输入字符串为 `"abacaba"`，下标如下：

```
下标：0 1 2 3 4 5 6
字符：a b a c a b a
```

程序在 `i % 2 == 1` 时追加字符，也就是取奇数下标处的字符。

下标 1 的字符是 `'b'`；

下标 3 的字符是 `'c'`；

下标 5 的字符是 `'b'`。

所以 `t` 最后是 `"bcb"`，选 B。

这道题的关键是下标从 0 开始。奇数下标不是“第奇数个字符”，而是代码中下标为 $1, 3, 5, \ldots$ 的字符。

十一、变式题

变式题 阅读下面程序，若输入为 `abcdef 2 3`，输出是什么？

```cpp
#include <iostream>
#include <string>
using namespace std;

int main() {
    string s;
    int p, len;
    cin >> s >> p >> len;
    cout << s.substr(p, len) << endl;
    return 0;
}
```

A. `bc`

B. `bcd`

C. `cd`

D. `cde`

答案：D

解析：

输入字符串为 `"abcdef"`，下标如下：

```
下标：0 1 2 3 4 5
字符：a b c d e f
```

p=2，len=3。s.substr(p, len) 表示从下标 2 开始，取长度为 3 的子串。

下标 2、3、4 对应的字符是 'c'、'd'、'e'，所以输出 "cde"。

答案选 D。

这道题的易错点是把 len 当成结束下标。substr(2, 3) 不是从下标 2 到下标 3，而是从下标 2 开始取 3 个字符。

十二、易错点

第一，字符和字符串不同。'a' 是字符，"a" 是字符串。单引号和双引号不能混用。

第二，string 的下标从 0 开始。长度为 n 的字符串，最后一个字符下标是 $n-1$。

第三，cin >> s 不能读入带空格的一整行。若要读入整行，通常使用 getline(cin, s)。

第四，substr(pos, len) 的第二个参数是长度，不是结束位置。若题目给的是从第 l 个到第 r 个字符，要转换成起点和长度。

第五，find 找不到时应与 string::npos 比较，不要把返回值固定理解为 -1。

第六，字符串比较按字典序，不等同于整数大小比较。"9" 和 "10" 作为字符串比较时，结果与数字 9 和 10 的大小不同。

第七，C 风格字符串以 '\0' 结束。strlen 统计的是 '\0' 之前的字符个数，不包括 '\0'，也不等于字符数组开辟的总空间。

十三、本章要点

字符串是字符序列。string 的长度可用 size() 或 length() 得到，下标从 0 开始，长度为 n 时合法下标为 0 到 $n-1$。

字符串可以拼接、修改、截取和查找。+ 和 += 都能拼接，但前者常生成新字符串，后者会修改原字符串。substr(pos, len) 从下标 pos 开始取长度为 len 的子串。find 返回第一次出现的位置，找不到时返回 string::npos。

字符串按字典序比较，从左到右找第一个不同字符；如果一个字符串是另一个的前缀，较短者更小。字符数字如 '7' 要转成整数 7，可以用 c - '0'。

C 风格字符串使用字符数组保存，并以 '\0' 表示结束。strlen 求的是实际字符串长度，不包括结尾的 '\0'，也不是数组容量。

做字符串阅读题时，先写出每个字符的下标，再追踪循环和修改。只要下标、长度、读入方式和子串参数不混，字符串处理题通常能稳定拿分。

0x1E 贪心

贪心是一类很适合 CSP-J 第一轮考查的算法思想。它不像动态规划那样需要完整保存很多状态，也不像搜索那样要展开大量分支，而是每一步都按照某个规则做一个当前选择，并希望这些选择合在一起得到最优答案。

第一轮中的贪心通常不会要求写出复杂证明。更常见的考法是：判断某个算法思想是不是贪心，理解某个排序规则为什么这样定，手算几个小规模过程，或者识别“每次选最大”“每次选最小”是否可靠。历年题目中，哈夫曼编码多次出现；四人过河这类小规模最短时间问题，也体现了贪心选择和分类比较的思想。

本章放在模拟、枚举、二分、递归、搜索、动态规划和字符串处理之后，是为了让学生能把几种常见算法思想分开：模拟按规则执行，枚举把可能情况试出来，动态规划保存子问题答案，贪心则依赖一个经过判断的局部选择规则。

一、贪心在想什么

贪心的核心不是“看见大的就选大”“看见小的就选小”，而是每一步只做一个局部选择，并且不回头修改。

例如有若干个活动，每个活动有开始时间和结束时间，要选出尽量多的互不冲突活动。一个常用贪心策略是：每次选结束时间最早的活动。因为结束得早，后面留下的时间更多，之后还能继续安排更多活动。

这个策略不是“选持续时间最短的活动”，也不是“选开始最早的活动”。如果只看开始时间，可能一开始选了一个很长的活动，把后面的很多活动都挡住；如果只看持续时间，可能短活动的位置不好，也不一定能留下更多选择。

贪心题最重要的问题是：当前这一步按什么规则选？

同样是“排序后选择”，排序关键字不同，结果可能完全不同。CSP-J 第一轮不常要求学生完整证明贪心正确性，但经常通过选项考查学生是否知道该按哪个规则排序。

判断贪心时，可以抓住三个问题：

第一，题目是否在求最大值、最小值或最优方案；

第二，是否存在一个可以反复执行的选择规则；

第三，做出当前选择后，剩下的问题是否仍然是同类问题。

如果这三个问题都能对应起来，才可能是贪心。若当前选择会影响后面很多状态，并且不能只靠一个规则决定，就要警惕：这可能是动态规划、搜索或枚举，而不是简单贪心。

二、局部最优不等于整体最优

贪心最容易出错的地方，是把“看起来最划算”当成“最后一定最优”。

看一个简单的换硬币问题。假设硬币面值为 $1, 3, 4$，要凑出 6，若每次都选不超过剩余金额的最大面值，那么过程是：

先选 4，剩余 2；

再选 1，剩余 1；

再选 1，剩余 0。

一共用了 3 枚硬币。但更好的方案是 $3 + 3$，只用 2 枚。这里“每次取最大面值”并不能保证最少硬币数。

这个例子说明：贪心规则必须有理由，不能只靠直觉。某些特殊面值体系下，“每次取最大面值”可以工作；换一组面值，就可能失败。第一轮选择题中，如果选项写成“所有最优化问题都可以用贪心解决”，应直接排除。

贪心和动态规划的区别也常在这里出现。动态规划会保存多个可能状态，用转移比较不同选择；贪心只保留当前规则得到的结果。如果题目必须比较很多种历史选择，贪心通常不够。

三、怎样判断一个贪心规则

正式证明贪心正确性不是 CSP-J 第一轮的重点，但学生至少要会用简单方式检查一个规则是否可信。

一种常见想法是“交换”。如果一个最优方案中没有采用我们想要的贪心选择，而把其中某一步换成贪心选择后，答案不会变差，那么说明这个贪心选择是安全的。

仍看活动选择问题。要选尽量多的互不冲突活动，按结束时间从小到大选。假设某个最优方案的第一个活动不是结束最早的活动，而是另一个结束更晚的活动。把它换成结束最早的活动后，选中的活动数量不会减少，因为结束更早只会给后面的活动留下更多时间。这样就能理解为什么“结束时间最早”是可信的规则。

这个思路不用写成严格证明，但可以帮助做选择题。若题目问“为什么要按结束时间排序”，答案通常不是“因为短”或“因为早开始”，而是“结束越早，给后续选择留下的空间越大”。

另一种检查方式是找反例。只要能找到一个小例子让规则失败，就说明该规则不是普遍正确的。比如前面的硬币面值 $1, 3, 4$ 和金额 6，就能反驳“每次取最大面值一定最少”。

第一轮做题时，可以快速测试选项中的贪心说法：

说法太绝对，先怀疑；

规则没有说明按什么排序，先怀疑；

只看当前收益，不管后续限制，先怀疑；

如果能用两三个小数据构造反例，就不要选它。

四、区间选择模型

区间问题是贪心中最常见的一类模型。CSP-J 第一轮不一定直接给出“区间选择”这个名称，但可能把它包装成活动安排、课程选择、会议安排、时间段选择等问题。

最基础的问题是：给出若干个时间段，选出尽量多的互不重叠时间段。

正确策略通常是：按结束时间从早到晚排序，然后依次选择能接在当前已选时间段之后的区间。

示例：

有 5 个活动，时间段分别为 $[1,3]$、$[2,5]$、$[4,6]$、$[6,7]$、$[5,9]$。若允许一个活动在另一个活动结束时立刻开始，则可以按结束时间排序后选择。

结束时间从早到晚为：

$[1,3]$，$[2,5]$，$[4,6]$，$[6,7]$，$[5,9]$。

先选 $[1,3]$，当前结束时间为 3；

$[2,5]$ 的开始时间小于 3，不能选；

$[4,6]$ 可以选，当前结束时间为 6；

$[6,7]$ 可以选，当前结束时间为 7；

$[5,9]$ 的开始时间小于 7，不能选。

最终选出 3 个活动。

对应代码片段可以写成：

```
sort(a + 1, a + n + 1, cmp);
int cnt = 0;
int last = -1;
for (int i = 1; i <= n; i++) {
    if (a[i].l >= last) {
        cnt++;
        last = a[i].r;
    }
}
```

这里 `cmp` 应该按右端点 `r` 从小到大排序。若右端点相同，再按左端点排序一般不影响最大数量，但为了程序稳定，可以补一个次关键字。

阅读这类程序时，要注意两个细节。

第一，条件是 `a[i].l >= last` 还是 `a[i].l > last`，取决于题目如何定义重叠。若一个活动在时刻 3 结束，另一个活动在时刻 3 开始，是否冲突，要看题面。

第二，`last` 表示当前已经选择的最后一个活动的结束时间，不是当前枚举到的活动编号。变量含义如果看错，后面的判断条件就会读错。

区间选择还有一个常见变形：用尽量少的区间覆盖一段目标区间。它的贪心策略通常不是按结束时间最早，而是在所有起点不超过当前已覆盖位置的区间中，选择右端点最远的一个。这个模型比活动选择更复杂，CSP-J 第一轮若出现，通常也只会考较小数据的手算或程序阅读。

两个模型的排序和选择规则不同，不能混在一起：

选最多互不冲突区间：看谁结束早；

用最少区间覆盖目标：看谁能覆盖得远。

五、哈夫曼编码

哈夫曼编码已经在第 21 章讲过。本章只从贪心角度再看一次它。

哈夫曼编码的目标是：出现频率高的字符用较短编码，出现频率低的字符可以用较长编码，从而使总编码长度尽量小。构造哈夫曼树时，每次选择权值最小的两个结点合并成一个新结点，新结点权值是二者之和。不断重复，直到只剩一棵树。

这个“每次选两个最小”就是典型贪心选择。它不是随便合并两个结点，也不是先合并最大权值。频率低的字符更适合放在树的深处，所以权值小的两个结点先合并，能让它们在后续结构中一起下沉。

示例：

权值为 $5, 9, 12, 13, 16, 45$。

第一次合并 5 和 9，得到 14；

当前权值为 $12, 13, 14, 16, 45$；

第二次合并 12 和 13，得到 25；

当前权值为 $14, 16, 25, 45$；

第三次合并 14 和 16，得到 30；

当前权值为 $25, 30, 45$；

第四次合并 25 和 30，得到 55；

当前权值为 $45, 55$；

第五次合并 45 和 55，得到 100。

每个字符的编码可能因为左右分支取 0 或 1 的方式不同而不同，但编码长度由树的深度决定。选择题中如果给出若干组编码，不能只看某个字符是不是 0，还要看是否满足前缀编码要求，以及高频字符是否对应短编码。

判断哈夫曼编码题时，可以按三个步骤做：

先按权值构造合并过程；

再得到每个字符的深度，也就是编码长度；

最后检查选项中每个字符的码长和前缀关系。

注意，左右子树标 0 或 1 不唯一。同一棵哈夫曼树可以产生多组不同的二进制编码。只要码长和前缀关系正确，左右互换通常不会影响最优性。因此，不能因为某个字符的编码不是自己算出的那串，就立刻判错。

六、排队与等待时间

另一类常见贪心模型是排队等待。问题通常是：有若干个人依次办理业务，每个人需要的时间不同，问怎样安排顺序能使总等待时间最小。

贪心策略是：用时短的人排在前面。

示例：

三个人办理业务的时间分别为 $8, 1, 4$。

如果按 $8, 1, 4$ 排队，三人的等待时间分别为 $0, 8, 9$，总等待时间为 17。

如果按 $1, 4, 8$ 排队，三人的等待时间分别为 $0, 1, 5$，总等待时间为 6。

为什么短的要放前面？因为排在前面的人会影响后面所有人。如果一个耗时很长的人放在前面，它会被加到许多人的等待时间里；把耗时短的人放前面，可以让更多人少等。

这个模型在第一轮中常以选择题出现。题目可能不要求写程序，只问“最小总等待时间”或“最佳顺序”。做题时，先把时间从小到大排序，再计算每个人等待了多少。

需要分清两个概念：总完成时间和总等待时间。若第 i 个人自己的办理时间也计入他的完成时间，那么总完成时间等于所有人的完成时刻之和；若只问等待时间，则不包括自己办理的时间。二者的最优顺序通常相同，但手算数值不同。

七、过河模型

过河问题常见设定是：若干个人从一岸到另一岸，船一次最多坐两人，船必须有人划回来，每个人过河时间不同，两人同行用较慢者的时间。目标是让所有人过河的总时间最短。

这类题数据通常很小，第一轮更看重纸面手算和策略比较。

以四个人过河为例，过河时间为 $1, 2, 4, 8$。船一次最多坐两人，两人同行用较慢者时间，求所有人到对岸的最短时间。

一个较优安排是：

1 和 2 过去，用时 2；

1 回来，用时 1；

4 和 8 过去，用时 8；

2 回来，用时 2；

1 和 2 过去，用时 2。

总时间为 $2+1+8+2+2=15$。

答案是 15。

这里不能简单地让最快的人反复送每个人过去。若按“最快的人一直摆渡”：

1 和 8 过去，1 回来；

1 和 4 过去，1 回来；

1 和 2 过去。

总时间是 $8+1+4+1+2=16$，比 15 多。

对于更多人的过河问题，常把时间从小到大记为 $a, b, \ldots, y, z$，其中 a 和 b 是最快的两个人，y 和 z 是当前最慢的两个人。把最慢两人送过去时，常比较两种方案：

方案一：a, b 先过去，a 回来，y, z 过去，b 回来，用时 $a+2b+z$；

方案二：a, z 过去，a 回来，a, y 过去，a 回来，用时 $2a+y+z$。

每次选择较小的方案，把两个最慢的人送到对岸，再处理剩下的人。CSP-J 第一轮一般不会让学生写完整程序，但这个比较式能帮助判断小规模题目。

做过河题时，不要只看“谁最快”。最快的人很适合回来，但第二快的人也很重要，因为有时让最快和第二快配合作为“摆渡员”，能减少最慢两个人带来的代价。

八、完善程序中的贪心

贪心若出现在完善程序中，空缺通常集中在三个位置。

第一，排序规则。

例如活动选择中，若程序后面是按顺序检查 `a[i].l >= last`，那么排序规则大多应是按结束时间从小到大。若空缺在 `cmp` 函数中，就要结合后面的选择条件判断它想实现什么贪心策略。

第二，选择条件。

排序之后，不是每个元素都要选。程序可能需要判断当前元素能否加入答案。活动选择看开始时间是否不早于 `last`；区间覆盖看左端点是否不超过当前覆盖位置；排队问题则可能排序后全部加入，只是计算方式不同。

第三，状态更新。

选择某个元素后，要更新当前状态。活动选择更新 `last`；区间覆盖更新已覆盖到的位置；哈夫曼合并更新当前权值集合；过河模型更新剩余人数。若只选择而不更新，后面的判断会失去依据。

看完善程序时，可以先把代码分成四段：

读入数据；

排序或准备数据；

循环选择；

输出答案。

贪心的关键通常在第二段和第三段。若选项中一个填法会让循环永远不前进，或让已经选择的状态不改变，多半不是正确答案。

九、贪心和其他算法的区别

贪心、枚举、动态规划很容易混淆。

枚举是把候选情况逐个试出来，再从中选最好的。例如枚举所有活动子集，再检查哪些互不冲突，这当然能求答案，但数量一大就不适合。贪心则不试所有子集，而是按规则直接选。

动态规划会保存多个状态。例如换硬币问题中，如果面值不是特殊体系，常设 `dp[i]` 表示凑出金额 i 的最少硬币数，再由小金额推出大金额。它会比较多种来源。贪心只选当前看好的一个来源，因此可能失败。

搜索会沿着不同分支深入尝试。比如迷宫路径、连通块填充、排列组合生成，都更像搜索。贪心一般不会回溯，也不会在失败后退回上一层换选择。

第一轮判断算法思想时，可以看程序特征：

若有排序，然后线性扫描选择，常可能是贪心；

若有 `dp` 数组和状态转移，通常是动态规划；

若有递归枚举多个分支，可能是搜索或递归；

若只是按题面规则一步一步维护变量，多半是模拟。

当然，程序可能同时使用多个思想。例如哈夫曼编码既用树结构，也体现贪心；区间题可能先排序再二分；最短路算法中也有贪心成分。本书只要求学生能在 CSP-J 第一轮范围内识别主要思想。

十、常见错误

第一，把 “贪心” 理解成 “每次选最大”。

很多贪心并不是选最大。活动选择选的是结束时间最早；哈夫曼编码每次合并两个最小；排队等待按耗时从小到大；过河问题还要比较两种送最慢者过河的方案。

第二，排序关键字看错。

贪心题常败在排序。区间选择按右端点，区间覆盖在可选范围内看最远右端点，排队等待按耗时。题目如果只问代码功能，要特别关注 `cmp` 函数。

第三，只算当前收益，不算后续影响。

当前收益最大不一定最好。把一个很长的活动先选走，后面可能少选很多活动；把最大面值硬币先选走，也可能导致硬币数变多。遇到 “每次都……” 的说法，要看是否存在反例。

第四，把哈夫曼编码的具体 0, 1 写法当成唯一答案。

哈夫曼树左右分支可以互换，编码本身可能不同。判断时应关注码长、前缀关系和总带权路径长度，而不是只盯着某个字符是不是写成某一串二进制。

第五，把贪心和动态规划混在一起。

贪心通常没有完整状态表，也不反复比较所有子问题；动态规划依靠状态和转移保存历史信息。如果程序中有二维数组 `dp[i][j]`，并且每个位置由前面的多个位置推出，一般不要硬说成贪心。

贪心题的纸面解法可以概括为三步：先找排序或选择规则，再手算前几步过程，最后检查这个规则有没有明显反例。CSP-J 第一轮中的贪心不会太深，但每个模型的规则必须分清，不能用一个“每次选最优”的空话代替具体判断。

十一、本章要点

贪心不是“每次选最大”或“每次选最小”，而是每一步按照一个明确规则做当前选择，并且不回头修改。一个贪心规则能不能用，要看它是否能保证后面的选择不会变差；如果能构造小反例，就说明这个规则不可靠。

区间选择模型中，若目标是选出尽量多的互不冲突区间，常按结束时间从早到晚排序；若目标是用尽量少的区间覆盖一段范围，则常在当前能接上的区间中选右端点最远的。两个模型都和区间有关，但选择规则不同，不能混用。

哈夫曼编码是 CSP-J 第一轮中很典型的贪心考点。构造时每次合并权值最小的两个结点。判断选项时，不要只看某个字符的编码是不是和自己写的一样，而要检查码长、前缀关系和高频字符是否更短。

排队等待类问题通常把耗时短的人放在前面，因为前面的人会影响后面所有人的等待时间。过河模型则不能只让最快的人反复摆渡，还要比较不同送最慢两人过河的方案，小规模题要把每一步的去回过程写清楚。

贪心和动态规划的区别在于：贪心按一个规则直接选择，通常不保存完整状态表；动态规划保存多个子问题答案，再从旧状态推出新状态。若程序中先排序再线性扫描选择，可能是贪心；若出现 `dp` 数组和状态转移，通常应按动态规划阅读。

做贪心题时，先找排序或选择规则，再手算前几步过程，最后检查有没有明显反例。规则说得越笼统，越要谨慎；只有能解释“为什么这样选不会影响后面更优结果”的规则，才适合作为贪心策略。

第六编 专项训练

0x1F 选择题分类训练

本章是选择题分类训练。前面各章已经讲过知识点，本章不再重新铺开讲概念，而是把常见选择题混合在一起，让学生练习“先分类，再判断”的过程。

一、题目

（一）概念与基础规则

1. 在 C++ 中，下列哪个声明能定义一个整型常量，并且之后不能通过变量名直接修改它的值？()

   A. `int x = 10;`
   B. `const int x = 10;`
   C. `static int x = 10;`
   D. `unsigned int x = 10;`

2. 下列关于 C++ 中数组的说法，正确的是（）。

   A. 若定义 `int a[10];`，则合法下标为 1 到 10
   B. 若定义 `int a[10];`，则 `a[10]` 表示数组最后一个元素
   C. 若定义 `int a[10];`，则数组中共有 10 个 `int` 类型元素
   D. C++ 数组下标必须从 1 开始

3. 以下哪个不是操作系统？()

   A. Linux
   B. Windows
   C. Android
   D. Dev-C++

4. 下面关于递归的说法，正确的是（）。

   A. 递归函数不能有返回值
   B. 递归函数必须直接或间接调用自身
   C. 递归函数一定比循环运行得更快
   D. 递归函数不能包含 `if` 语句

（二）进制、单位与计数

5. 二进制数 101101_2 与八进制数 27_8 的和，转换成十进制后为（）。

   A. 66

B. 68
C. 72
D. 74

6. 记 1KB = 1024B，1MB = 1024KB，1B = 8bit，则 2MB 等于多少二进制位？()

A. 2097152
B. 4194304
C. 8388608
D. 16777216

7. 从 6 名学生中选出 3 人参加活动，要求甲、乙两人中至少有一人被选中。不同的选择方式有 () 种。

A. 12
B. 14
C. 16
D. 18

8. 有 5 个互不相同的数字排成一行，要求数字 1 与数字 2 相邻。不同的排列方式有 () 种。

A. 24
B. 36
C. 48
D. 60

（三）数据结构与算法常识

9. 有一个空栈，元素按照 $1,2,3,4,5$ 的顺序依次入栈。下面哪一个序列不可能是出栈序列？()

A. $1,2,3,4,5$
B. $5,4,3,2,1$
C. $2,1,4,5,3$
D. $3,1,2,5,4$

10. 一棵完全二叉树用数组从下标 1 开始存储。若某结点存储在数组下标 14 的位置，则它的父结点下标和左孩子下标分别为 ()。

A. $6,28$
B. $7,28$
C. $7,29$
D. $28,7$

11. 已知一棵二叉树的前序遍历为 ABDCE，中序遍历为 DBAEC，则它的后序遍历为 ()。

A. DBECA
B. DBCEA
C. DBEAC
D. BDECA

12. 后缀表达式 `4 2 + 3 * 5 -` 的值为 ()。

A. 7
B. 13
C. 18
D. 21

13. 假设有 4 个字符的出现频率分别为 $2, 3, 7, 9$。若使用哈夫曼编码，则该哈夫曼树的带权路径长度为 ()。

A. 35
B. 37
C. 38
D. 42

14. 下列关于排序算法常见实现的说法，错误的是 ()。

A. 冒泡排序通常是稳定的
B. 插入排序通常是稳定的
C. 简单选择排序通常是稳定的
D. 归并排序可以实现为稳定排序

15. 在无向图中，若共有 12 条边，则所有顶点的度数之和为 ()。

A. 12
B. 24
C. 36
D. 不能确定

16. 广度优先搜索中，最常配合使用的数据结构是 ()。

A. 栈
B. 队列
C. 二叉树
D. 哈夫曼树

17. 下面哪种算法思想最符合“每次从当前可选对象中选择结束时间最早的活动，然后继续处理剩余活动”？()

A. 枚举
B. 递归
C. 贪心
D. 动态规划

（四）程序片段与复杂度

18. 阅读以下代码片段，输出结果为 ()。

```
int a[5] = {1, 2, 3, 4, 5};
int s = 0;
for (int i = 0; i < 5; i += 2) {
    s += a[i];
}
cout << s << endl;
```

A. 6
B. 8
C. 9
D. 15

19. 阅读以下代码片段，输出结果为 ()。

```
int x = 3 + 4 * 2;
int y = 5 % 2;
cout << ((x > 10) && (y == 1)) << endl;
```

A. 0
B. 1
C. 2
D. 11

20. 阅读以下代码片段，函数 `f(4)` 的返回值为 ()。

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

A. 7
B. 9
C. 10
D. 24

21. 下列程序片段的时间复杂度为 ()。

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

A. $O(n)$
B. $O(n \log n)$
C. $O(n^2)$

D. $O(\log n)$

22. 在一个有序数组中使用二分查找。若数组中有 33 个元素，则最坏情况下最多需要比较 () 次。

A. 5
B. 6
C. 7
D. 33

二、答案与解析

答案速查

题号	1	2	3	4	5	6	7	8	9	10	11
答案	B	C	D	B	B	D	C	C	D	B	A

题号	12	13	14	15	16	17	18	19	20	21	22
答案	B	C	C	B	B	C	C	B	C	B	B

1. 答案：B

`const int x = 10;` 定义的是整型常量，之后不能通过 `x = ...` 这样的赋值语句修改它的值。

A 只是普通整型变量。C 中的 `static` 影响变量的存储和作用范围，不表示常量。D 中的 `unsigned` 表示无符号整数，也不表示不能修改。

2. 答案：C

`int a[10];` 表示数组中有 10 个 `int` 类型元素。C++ 数组下标从 0 开始，因此合法下标是 0 到 9。`a[10]` 已经越过数组最后一个位置，不能把它当成最后一个元素。

这类题的关键是区分“元素个数”和“最大下标”。长度为 10 的数组，最后一个下标是 9。

3. 答案：D

Linux、Windows、Android 都是操作系统。Dev-C++ 是常见的 C++ 集成开发环境，不是操作系统。

做计算机常识题时，遇到“不是”一类问法，应先圈出否定词，再逐项判断。

4. 答案：B

递归的基本特征是函数直接或间接调用自身。递归函数可以有返回值，也可以没有返回值；递归中常常需要 `if` 判断递归出口；递归也不一定比循环更快。

A、C、D 都把某种写法说成了必然规则，范围过大。

5. 答案：B

先把两个数转换成十进制。$101101_2 = 32+8+4+1 = 45$，$27_8 = 2\times 8+7 = 23$。两者相加得到 $45+23 = 68$。

选项 B 正确。进制混合计算时，先统一到十进制通常最稳。

6. 答案：D

2MB $=$ 2 $\times$ 1024KB $=$ 2 $\times$ 1024 $\times$ 1024B。每个字节是 8 个二进制位，所以二进制位数为 $2\times 1024\times 1024\times 8 = 16777216$。

A 是 2MB 的字节数，没有乘 8。C 是 1MB 的二进制位数。

7. 答案：C

总共从 6 人中选 3 人，有 $\binom{6}{3} = 20$ 种。若甲、乙都没有被选中，就只能从另外 4 人中选 3 人，有 $\binom{4}{3} = 4$ 种。

因此至少选中甲、乙之一的方案数为 $20-4 = 16$。

这类“至少”问题常用反面计算：先算总数，再减去一个都不满足的情况。

8. 答案：C

把数字 1 和数字 2 看成一个整体。这个整体内部可以排列为 12 或 21，有 2 种。

此时共有 4 个对象：整体、另外 3 个数字。它们排成一行有 $4! = 24$ 种，再乘以整体内部的 2 种，得到 48 种。

答案为 C。

9. 答案：D

A 可以每入栈一个元素就立刻出栈。B 可以先全部入栈，再依次出栈。C 的过程可以是：入 $1,2$，出 $2,1$；入 $3,4$，出 4；入 5，出 $5,3$。

D 不可能。若先输出 3，说明 $1,2,3$ 已经入栈且 3 出栈。此时栈顶是 2，不能直接输出 1。所以 D 违反后进先出的规则。

栈题不要只看数字大小，要模拟栈顶元素。

10. 答案：B

完全二叉树采用 1 下标存储时，下标为 i 的结点：

父结点下标为 $\lfloor i/2\rfloor$；

左孩子下标为 $2i$；

右孩子下标为 $2i+1$。

当 $i = 14$ 时，父结点下标为 7，左孩子下标为 28。答案为 B。

11. 答案：A

前序遍历第一个字符 A 是根。中序遍历 DBAEC 中，A 左边是 DB，右边是 EC。

左子树的前序为 BD，中序为 DB，所以左子树根为 B，它的左孩子为 D。左子树后序为 DB。

右子树的前序为 CE，中序为 EC，所以右子树根为 C，它的左孩子为 E。右子树后序为 EC。

整棵树后序为左子树、右子树、根，即 DBECA。

12. 答案：B

后缀表达式从左到右处理，遇到数字入栈，遇到运算符取出栈顶两个数计算。

4 2 + 得到 6；

再与 3 做乘法，得到 18；

最后减去 5，得到 13。

答案为 B。

13. 答案：C

哈夫曼编码每次合并权值最小的两个结点。过程如下：

先合并 2 和 3，得到 5；

再合并 5 和 7，得到 12；

最后合并 9 和 12，得到根。

于是权值 9 的深度为 1，权值 7 的深度为 2，权值 2 和 3 的深度为 3。带权路径长度为 $9 \times 1 + 7 \times 2 + 2 \times 3 + 3 \times 3 = 38$。

答案为 C。

14. 答案：C

冒泡排序和插入排序在常见写法下可以保持相等元素的原有相对顺序，通常是稳定的。归并排序也可以写成稳定排序。

简单选择排序在常见实现中会把后面的最小元素与前面位置交换，可能改变相等元素的相对顺序，因此通常不是稳定排序。错误说法是 C。

15. 答案：B

无向图中，每条边会给两个端点各贡献 1 个度数。因此所有顶点度数之和等于边数的 2 倍。

有 12 条边时，度数之和为 24。

16. 答案：B

广度优先搜索通常按“先到先处理”的顺序扩展结点，最常配合使用队列。队列的特点是先进先出，符合逐层扩展的过程。

栈更常见于深度优先搜索的非递归实现。

17. 答案：C

每次选择结束时间最早的活动，然后继续处理剩余活动，是活动选择问题中的常见贪心策略。

它不是枚举，因为没有尝试所有活动子集；不是普通递归，因为没有展开多个分支；也不是动态规划，因为没有保存一张状态表。答案为 C。

18. 答案：C

数组下标和值分别为：

`a[0]=1`，`a[1]=2`，`a[2]=3`，`a[3]=4`，`a[4]=5`。

循环中 `i` 依次为 $0, 2, 4$，所以累加的是 `a[0]`、`a[2]`、`a[4]`，和为 $1 + 3 + 5 = 9$。

答案为 C。

19. 答案：B

先计算 `x`。乘法优先于加法，所以 `x = 3 + 4 * 2 = 11`。

再计算 `y`。`5 % 2 = 1`。

于是 `(x > 10)` 为真，`(y == 1)` 也为真，两个条件用 `&&` 连接后仍为真。`cout` 输出布尔值时，真通常输出 `1`，假输出 `0`。答案为 B。

20. 答案：C

逐层计算：

`f(1)=1`；

`f(2)=f(1)+2=3`；

`f(3)=f(2)+3=6`；

`f(4)=f(3)+4=10`。

因此答案为 C。

21. 答案：B

外层循环中，`i` 依次为 $1, 2, 4, 8, \ldots$，每次乘以 2，所以外层循环次数为 $O(\log n)$。

内层循环每次执行 n 次。总执行次数为 $O(n \log n)$。

答案为 B。

22. 答案：B

二分查找每比较一次，大致排除一半元素。最坏情况下，33 个元素需要的比较次数可以按“最多能区分多少种情况”判断。

5 次比较最多覆盖的规模不够，因为 $2^5 = 32$。6 次比较已经足够覆盖 33 个元素的查找过程。因此最坏情况下最多比较 6 次。

答案为 B。

0x20 程序阅读分类训练

本章是程序阅读分类训练。题面组织、代码长度、判断题与单选题混合方式尽量贴近 CSP-J 第一轮。

做本章题时，建议先遮住答案与解析，只看程序和题目。每段程序至少读两遍：第一遍看函数作用，第二遍追踪具体输入。遇到判断题时，不要只凭感觉判断；遇到单选题时，先算出自己的结果，再看选项。

一、题目

(1)

```cpp
#include <iostream>
using namespace std;

int a[105], b[105];

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

    int ans = 0;
    for (int l = 1; l <= n; l++) {
        for (int r = l; r <= n; r++) {
            if ((b[r] - b[l - 1]) % 2 == 0) {
                ans++;
            }
        }
    }
    return ans;
}

int main() {
    int n;
    cin >> n;
    for (int i = 1; i <= n; i++) {
        cin >> a[i];
    }
    cout << calc(n) << endl;
    return 0;
}
```

假设输入的 n 是不超过 100 的正整数，数组元素均为整数，完成下面的判断题和单选题。

判断题

1. 当输入为 `4 1 2 3 4` 时，程序输出为 `4`。()
2. 若输入数组中的所有元素都是偶数，则程序输出一定为 $n(n+1)/2$。()
3. 该程序统计的是所有元素和为奇数的连续子数组个数。()

单选题

4. 当输入为 `5 2 1 1 2 3` 时，程序输出为 ()。

   A. `5`
   B. `6`
   C. `7`
   D. `8`

5. 若将第 8 行附近的语句 `b[i] = b[i - 1] + a[i];` 改为 `b[i] = a[i];`，当输入为 `3 1 1 1` 时，程序输出为 ()。

   A. `1`
   B. `2`
   C. `3`
   D. `6`

6. 当 $n = 5$ 时，该程序可能输出的最大值为 ()。

   A. `5`
   B. `10`
   C. `15`
   D. `25`

(2)

```cpp
#include <iostream>
#include <string>
using namespace std;

string work(string s) {
    string t;
    for (int i = 0; i < s.size(); i++) {
        if (!t.empty() && t[t.size() - 1] == s[i]) {
            t.erase(t.size() - 1, 1);
        } else {
            t += s[i];
        }
    }
    return t;
}

int main() {
    string s;
```

```
    cin >> s;
    string t = work(s);
    cout << t << " " << t.size() << endl;
    return 0;
}
```

假设输入字符串只由小写英文字母组成，长度不超过 100，完成下面的判断题和单选题。

判断题

7. 当输入为 `abbac` 时，程序输出为 `c 1`。()
8. `work("aaaa")` 的返回值为空字符串。()
9. 该程序只会删除原字符串中第一次出现的一对相邻相同字符，然后立即停止处理。()

单选题

10. 当输入为 `abbacd` 时，程序输出为 ()。

    A. `acd 3`
    B. `cd 2`
    C. `bd 2`
    D. `abbacd 6`

11. 若将第 8 行附近的条件 `t[t.size() - 1] == s[i]` 改为 `t[t.size() - 1] != s[i]`，当输入为 `abbacd` 时，最终输出中第二个数为 ()。

    A. `0`
    B. `1`
    C. `2`
    D. `6`

12. 设输入字符串长度为 n，函数 `work` 的时间复杂度最接近 ()。

    A. $O(1)$
    B. $O(n)$
    C. $O(n^2)$
    D. $O(2^n)$

(3)

```
#include <iostream>
using namespace std;

int f(int n) {
    if (n == 0) {
        return 0;
    }
    if (n % 2 == 0) {
        return f(n / 2) + 1;
```

```
    }
    return f(n - 1) + 1;
}

int g(int n) {
    int cnt = 0;
    while (n > 0) {
        if (n % 2 == 0) {
            n /= 2;
        } else {
            n--;
        }
        cnt++;
    }
    return cnt;
}

int main() {
    int n;
    cin >> n;
    cout << f(n) << " " << g(n) << endl;
    return 0;
}
```

假设输入的 n 是不超过 10^6 的自然数，完成下面的判断题和单选题。

判断题

13. 当输入为 `10` 时，程序输出为 `5 5`。()
14. 对任意正整数 n，`f(n)` 与 `g(n)` 的返回值相同。()
15. 函数 `f(0)` 的返回值为 `1`。()

单选题

16. 当输入为 `23` 时，程序输出为 ()。

    A. `7 7`
    B. `8 8`
    C. `9 9`
    D. `10 10`

17. 若将函数 `f` 中的 `return f(n / 2) + 1;` 改为 `return f(n / 2);`，当输入为 `8` 时，程序输出为 ()。

    A. `1 4`
    B. `2 4`
    C. `4 4`
    D. `0 4`

18. 函数 `g` 的时间复杂度最接近 ()。

A. $O(1)$
B. $O(\log n)$
C. $O(n)$
D. $O(n^2)$

二、答案与解析

答案速查

题号	1	2	3	4	5	6
答案	✓	✓	✗	C	C	C

题号	7	8	9	10	11	12
答案	✓	✓	✗	B	A	B

题号	13	14	15	16	17	18
答案	✓	✓	✗	B	A	B

（1）解析

这段程序的核心是数组 `b`。第一个循环中，`b[i]` 表示 `a[1]` 到 `a[i]` 的和，因此 `b[r] - b[l - 1]` 表示连续子数组 `a[l]` 到 `a[r]` 的元素和。

第二重循环枚举所有满足 $1 \le l \le r \le n$ 的区间。如果区间和是偶数，就把 `ans` 加 1。所以程序统计的是“元素和为偶数的连续子数组个数”。

第 1 题，输入 `4 1 2 3 4` 时，数组为 $1,2,3,4$。所有连续子数组中，和为偶数的有：$1+2+3$、$1+2+3+4$、2、4，共 4 个，判断正确。

第 2 题，若所有 `a[i]` 都是偶数，则任意连续子数组的元素和都是偶数。长度为 n 的数组共有 $n(n+1)/2$ 个连续子数组，所以判断正确。

第 3 题，程序判断条件是 `(b[r] - b[l - 1]) % 2 == 0`，统计的是偶数和，不是奇数和，所以判断错误。

第 4 题，输入数组为 $2,1,1,2,3$。前缀和奇偶性依次为：

位置	0	1	2	3	4	5
前缀和	0	2	3	4	6	9
奇偶性	偶	偶	奇	偶	偶	奇

两个前缀和奇偶性相同，它们相减后才是偶数。偶前缀有 4 个，可组成 $\binom{4}{2}=6$ 个区间；奇前缀有 2 个，可组成 $\binom{2}{2}=1$ 个区间。总数为 7，选 C。

第 5 题，修改后 `b` 不再是前缀和。输入 `3 1 1 1` 时，`b[0]=0`，`b[1]=1`，`b[2]=1`，`b[3]=1`。程序仍然枚举 `b[r] - b[l - 1]` 是否为偶数。

逐个看所有区间：

- `l=1,r=1`：$b[1]-b[0]=1$，奇数；
- `l=1,r=2`：$b[2]-b[0]=1$，奇数；
- `l=1,r=3`：$b[3]-b[0]=1$，奇数；
- `l=2,r=2`：$b[2]-b[1]=0$，偶数；
- `l=2,r=3`：$b[3]-b[1]=0$，偶数；
- `l=3,r=3`：$b[3]-b[2]=0$，偶数。

共有 3 个，选 C。

第 6 题，长度为 5 的数组共有 $5\times 6/2=15$ 个连续子数组。如果所有元素都是偶数，则每个连续子数组的和都是偶数，程序输出可达到 15。答案为 C。

（2）解析

这段程序可以把字符串 `t` 看作一个栈。每读入一个字符 `s[i]`：

- 若 `t` 非空，且 `t` 的最后一个字符等于 `s[i]`，就删掉 `t` 的最后一个字符；
- 否则，把 `s[i]` 加到 `t` 的末尾。

注意，程序不是在原字符串上只删除一次，而是从左到右持续处理。删掉一对字符后，后面的字符还可能继续和新的末尾字符配对。

第 7 题，输入 `abbac` 的处理过程如下：

读入字符	操作后 `t`
a	a
b	ab
b	a
a	空
c	c

最终输出为 `c 1`，判断正确。

第 8 题，`aaaa` 的处理过程是：前两个 `a` 抵消，后两个 `a` 抵消，返回空字符串，判断正确。

第 9 题，程序会一直处理到字符串末尾，不是删除一次就停止。例如 `abbac` 中先删除 `bb`，之后新形成的 `aa` 还会继续被删除，所以判断错误。

第 10 题，输入 `abbacd` 的过程是：

读入字符	操作后 t
a	a
b	ab
b	a
a	空
c	c
d	cd

最终输出为 `cd 2`，选 B。

第 11 题，条件改成“不相等时删除末尾字符”。对输入 `abbacd` 重新追踪：

读入字符	操作后 t
a	a
b	空
b	b
a	空
c	c
d	空

最终 `t.size()` 为 0，所以第二个数为 `0`，选 A。

第 12 题，循环从左到右扫描输入字符串，每个字符处理一次。`t += s[i]` 和删除最后一个字符都只发生在字符串末尾，按本题范围可按一次基本操作理解。函数 `work` 的时间复杂度最接近 $O(n)$，选 B。

（3）解析

函数 `f` 和函数 `g` 做的是同一件事：把当前整数变成 0，每一步按下面规则操作：

- 如果当前数是偶数，就除以 2；
- 如果当前数是奇数，就减去 1；
- 每进行一步，计数加 1。

区别在于，`f` 用递归写法，`g` 用循环写法。

第 13 题，输入 10 时，变化过程为 $10 \to 5 \to 4 \to 2 \to 1 \to 0$，共 5 步，所以输出为 `5 5`，判断正确。

第 14 题，对任意正整数 n，两个函数的规则一致：偶数走到 $n/2$，奇数走到 $n-1$，并且都把步数加 1。到 0 时都停止。因此返回值相同，判断正确。

第 15 题，函数 `f` 的第一段代码是：

```
if (n == 0) {
    return 0;
}
```

所以 `f(0)` 返回 0，不是 1，判断错误。

第 16 题，输入 23 时，变化过程为：

$23 \to 22 \to 11 \to 10 \to 5 \to 4 \to 2 \to 1 \to 0$。

共 8 步，所以程序输出 `8 8`，选 B。

第 17 题，修改后，函数 `f` 在处理偶数时不再把当前这一步计入答案。输入 8 时，递归过程为：

`f(8)` 调用 `f(4)`，不加 1；

`f(4)` 调用 `f(2)`，不加 1；

`f(2)` 调用 `f(1)`，不加 1；

`f(1)` 调用 `f(0)` 并加 1。

因此修改后的 `f(8)` 返回 1。函数 `g` 没有修改，`g(8)` 的过程是 $8 \to 4 \to 2 \to 1 \to 0$，共 4 步。程序输出 `1 4`，选 A。

第 18 题，函数 `g` 中，每遇到偶数就除以 2，每遇到奇数就先减 1，之后如果还大于 0，通常会进入除以 2 的过程。整体步数与二进制位数同级，时间复杂度最接近 $O(\log n)$，选 B。

0x21 完善程序分类训练

本章是完善程序分类训练。前面章节已经讲过常见知识，本章重点放在“从题意和代码框架推出空缺处”的能力。

完善程序题不能只看某一个空。每个空都在一段程序流程中：有的负责初始化，有的负责循环边界，有的负责状态转移，有的负责更新答案。做题时，应先判断整段程序想完成什么，再判断每个空缺位置承担什么作用。

本章代码风格和设问方式尽量贴近 CSP-J 第一轮。题目与答案解析分开，建议先独立完成全部题目，再核对最后的解析。

一、题目

（1）（最长连续上升段）

给定 n 个整数，求其中最长连续上升段的长度。连续上升段指的是若干个相邻元素组成的一段，并且从第二个元素开始，每个元素都严格大于前一个元素。

例如，序列 $2,3,5,4,6,7,1$ 中，最长连续上升段是 $4,6,7$，长度为 3。

试补全程序。

```
#include <iostream>
using namespace std;

const int N = 1005;
int a[N];

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

    int ans = 1, len = 1;
    for (int i = 2; i <= n; i++) {
        if (①) {
            ②;
        } else {
            ③;
        }
        if (④) {
            ans = len;
```

```
        }
    }

    cout << ⑤ << endl;
    return 0;
}
```

1. ①处应填（）

   A. `a[i] > a[i - 1]`
   B. `a[i] >= a[i - 1]`
   C. `a[i] < a[i - 1]`
   D. `a[i] == a[i - 1]`

2. ②处应填（）

   A. `len++`
   B. `ans++`
   C. `len = 1`
   D. `i++`

3. ③处应填（）

   A. `len++`
   B. `len = 1`
   C. `ans = 1`
   D. `a[i] = a[i - 1]`

4. ④处应填（）

   A. `len < ans`
   B. `len == ans`
   C. `len > ans`
   D. `i > ans`

5. ⑤处应填（）

   A. `n`
   B. `len`
   C. `ans`
   D. `a[ans]`

（2）（网格最短步数）

给定一个 n 行 m 列的网格，其中 `.` 表示可以经过的位置，`#` 表示障碍，`S` 表示起点，`T` 表示终点。每一步可以向上、下、左、右四个方向移动一格，不能走出网格，也不能经过障碍。求从 `S` 到 `T` 的最少步数；若无法到达，则输出 `-1`。

试补全程序。

```cpp
#include <iostream>
#include <queue>
using namespace std;

const int N = 25;
int n, m;
char mp[N][N];
int dis[N][N];
int dx[4] = {1, -1, 0, 0};
int dy[4] = {0, 0, 1, -1};

struct Node {
    int x, y;
};

bool ok(int x, int y) {
    return ①;
}

int bfs(int sx, int sy, int tx, int ty) {
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            dis[i][j] = -1;
        }
    }

    queue<Node> q;
    q.push(Node{sx, sy});
    ②;

    while (!q.empty()) {
        Node u = q.front();
        q.pop();

        if (③) {
            return dis[u.x][u.y];
        }

        for (int k = 0; k < 4; k++) {
            int nx = u.x + dx[k];
            int ny = u.y + dy[k];
            if (ok(nx, ny) && ④) {
                dis[nx][ny] = dis[u.x][u.y] + 1;
                ⑤;
            }
        }
    }

    return -1;
}

int main() {
    cin >> n >> m;
```

```
    int sx = 0, sy = 0, tx = 0, ty = 0;

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            cin >> mp[i][j];
            if (mp[i][j] == 'S') {
                sx = i;
                sy = j;
            }
            if (mp[i][j] == 'T') {
                tx = i;
                ty = j;
            }
        }
    }

    cout << bfs(sx, sy, tx, ty) << endl;
    return 0;
}
```

6. ①处应填（）

   A. `x >= 1 && x <= n && y >= 1 && y <= m && mp[x][y] != '#'`
   B. `x >= 1 && x <= n && y >= 1 && y <= m && mp[x][y] == '#'`
   C. `x >= 0 && x < n && y >= 0 && y < m && mp[x][y] != '#'`
   D. `x >= 1 && x <= m && y >= 1 && y <= n && mp[x][y] != '#'`

7. ②处应填（）

   A. `dis[sx][sy] = -1`
   B. `dis[sx][sy] = 0`
   C. `dis[tx][ty] = 0`
   D. `dis[tx][ty] = 1`

8. ③处应填（）

   A. `u.x == sx && u.y == sy`
   B. `u.x == tx && u.y == ty`
   C. `mp[u.x][u.y] == '#'`
   D. `dis[u.x][u.y] == -1`

9. ④处应填（）

   A. `dis[nx][ny] == -1`
   B. `dis[nx][ny] != -1`
   C. `mp[nx][ny] == '#'`
   D. `nx == tx && ny == ty`

10. ⑤处应填（）

A. `q.pop()`

B. `q.push(u)`

C. `q.push(Node{nx, ny})`

D. `q.push(Node{sx, sy})`

（3）（最少硬币数）

有 n 种硬币，第 i 种硬币的面值为 c_i，每种硬币都可以使用任意多枚。给定目标金额 m，求凑出金额 m 所需的最少硬币数。若无法凑出，则输出 `-1`。

试补全程序。

```
#include <iostream>
using namespace std;

const int N = 105;
const int M = 10005;
const int INF = 1000000000;

int c[N];
int dp[M];

int main() {
    int n, m;
    cin >> n >> m;
    for (int i = 1; i <= n; i++) {
        cin >> c[i];
    }

    for (int i = 0; i <= m; i++) {
        dp[i] = ①;
    }
    dp[0] = ②;

    for (int i = 1; i <= n; i++) {
        for (int j = ③; j <= m; j++) {
            if (dp[j - c[i]] + 1 < dp[j]) {
                ④;
            }
        }
    }

    if (⑤) {
        cout << -1 << endl;
    } else {
        cout << dp[m] << endl;
    }

    return 0;
}
```

11. ①处应填（）

A. `0`
B. `1`
C. `INF`
D. `m`

12. ②处应填（）

A. `0`
B. `1`
C. `INF`
D. `m`

13. ③处应填（）

A. `0`
B. `1`
C. `c[i]`
D. `m - c[i]`

14. ④处应填（）

A. `dp[j] = dp[j - c[i]] + 1`
B. `dp[j - c[i]] = dp[j] + 1`
C. `dp[j]++`
D. `dp[i] = dp[j]`

15. ⑤处应填（）

A. `dp[m] == INF`
B. `dp[m] == 0`
C. `m == 0`
D. `dp[0] == INF`

二、答案与解析

答案速查

题号	1	2	3	4	5
答案	A	A	B	C	C

题号	6	7	8	9	10
答案	A	B	B	A	C

题号	11	12	13	14	15
答案	C	A	C	A	A

（1）解析

这段程序用 `len` 表示“以当前位置结尾的连续上升段长度”，用 `ans` 表示目前见过的最长长度。

初始化时，至少有一个元素，因此 `ans = 1`，`len = 1`。循环从 `i = 2` 开始，是因为要把 `a[i]` 与它前面的 `a[i - 1]` 进行比较。

第 1 题，连续上升要求当前元素严格大于前一个元素，因此条件应为 `a[i] > a[i - 1]`，选 A。这里不能使用 `>=`，因为相等不属于严格上升。

第 2 题，如果 `a[i] > a[i - 1]`，说明当前元素可以接在前一个连续上升段后面，长度增加 1，所以应填 `len++`，选 A。

第 3 题，如果当前元素不能接上前面的连续上升段，那么以当前位置结尾的新段只能从 `a[i]` 自己开始，长度重新变为 1，所以应填 `len = 1`，选 B。

第 4 题，更新答案时，应当在当前长度比历史答案更大时更新，所以应填 `len > ans`，选 C。

第 5 题，最终要求输出最长连续上升段的长度，对应变量是 `ans`，选 C。`len` 只是“以最后一个位置结尾”的段长，不一定是全局最长。

这一题的空缺可以按“判断是否延长当前段、延长或重置、更新答案、输出答案”的顺序理解。

（2）解析

这段程序使用广度优先搜索求网格中的最短步数。`dis[x][y]` 表示从起点走到位置 `(x,y)` 的最少步数。初始化为 `-1`，表示还没有访问过。

第 6 题，函数 `ok` 用来判断一个位置能否进入。合法位置必须满足：

- 行号在 1 到 n 之间；
- 列号在 1 到 m 之间；
- 该位置不是障碍 `#`。

因此应填 `x >= 1 && x <= n && y >= 1 && y <= m && mp[x][y] != '#'`，选 A。C 选项采用了 0 下标边界，但本程序读入网格时使用的是 1 到 n、1 到 m。

第 7 题，起点已经入队。起点到自身的距离应为 0，所以应填 `dis[sx][sy] = 0`，选 B。

第 8 题，广度优先搜索第一次取出终点时，得到的就是最少步数。因此在取出队首后，应判断当前位置是否为终点，即 `u.x == tx && u.y == ty`，选 B。

第 9 题，一个位置可以入队的前提是它合法，并且还没有访问过。`dis[nx][ny] == -1` 表示尚未访问，所以应填这一项，选 A。若不检查这一条件，已经访问过的位置可能反复入队。

第 10 题，确定新位置 `(nx,ny)` 可以访问后，需要把这个新位置加入队列，因此应填 `q.push(Node{nx, ny})`，选 C。

这一题做空时，先要找清楚三个变量的含义：

- `mp` 保存地图；
- `dis` 保存最少步数，同时兼作访问标记；
- `q` 保存还需要继续扩展的位置。

然后再判断每个空属于“合法性判断、起点初始化、终点判断、访问判重、入队扩展”中的哪一类。

（3）解析

这段程序求的是“无限次使用硬币”的最少硬币数。`dp[j]` 表示凑出金额 j 所需的最少硬币数。

第 11 题，开始时，除金额 0 外，其余金额都还不能确定能否凑出。为了表示“暂时不可达”，应把 `dp[i]` 初始化为一个很大的数 `INF`，选 C。

第 12 题，凑出金额 0 不需要任何硬币，所以 `dp[0] = 0`，选 A。

第 13 题，当前硬币面值为 `c[i]`。若要使用这枚硬币更新金额 `j`，必须保证 `j - c[i]` 不为负，因此内层循环应从 `c[i]` 开始，选 C。

第 14 题，若先凑出金额 `j - c[i]`，再加上一枚面值为 `c[i]` 的硬币，就可以凑出金额 `j`。对应转移是 `dp[j] = dp[j - c[i]] + 1`，选 A。

第 15 题，循环结束后，如果 `dp[m]` 仍然等于 `INF`，说明金额 m 无法凑出，应输出 `-1`。所以应填 `dp[m] == INF`，选 A。

这一题的核心是区分三个值：

- `INF` 表示暂时不可达；
- `dp[0]=0` 是动态规划的起点；
- `dp[j - c[i]] + 1` 表示在已有方案后再加一枚当前硬币。

内层循环从小到大枚举 `j`，表示同一种硬币可以重复使用。如果从大到小枚举，含义会变成每种硬币最多使用一次，与题意不符。

0x22 综合卷复盘

做综合卷时，建议按正式考试节奏完成。完成后不要只看答案，还要复盘错误来源：是概念不清，还是边界没看准；是程序没有追踪完，还是选项没有逐项排除。

一、综合训练卷

（一）单项选择题

1. 在 C++ 中，下列哪个语句可以定义一个值不能通过变量名直接修改的整型变量？()

   A. `int x = 5;`
   B. `const int x = 5;`
   C. `unsigned int x = 5;`
   D. `static int x = 5;`

2. 二进制数 10110_2 与八进制数 35_8 的和，转换成十进制后为 ()。

   A. 49
   B. 50
   C. 51
   D. 57

3. 一个空栈中，元素按照 $1,2,3,4,5$ 的顺序依次入栈。下面哪一个序列不可能是出栈序列？()

   A. $1,3,2,5,4$
   B. $4,3,2,1,5$
   C. $2,4,3,5,1$
   D. $3,1,2,5,4$

4. 根结点高度为 1。一棵包含 100 个结点的完全二叉树，其高度至少为 ()。

   A. 5
   B. 6
   C. 7
   D. 8

5. 一棵完全二叉树用数组从下标 1 开始存储。若某结点存储在数组下标 11 的位置，则它的父结点下标和右孩子下标分别为 ()。

   A. $5, 22$
   B. $5, 23$

C. $6, 22$

D. $6, 23$

6. 后缀表达式 `5 1 2 + 4 * + 3 -` 的值为 ()。

A. 11

B. 12

C. 14

D. 18

7. 下列程序片段的时间复杂度最接近 ()。

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

A. $O(n)$

B. $O(n \log n)$

C. $O(n^2)$

D. $O(\log n)$

8. 下列关于排序算法常见实现的说法，错误的是 ()。

A. 冒泡排序通常是稳定的

B. 插入排序通常是稳定的

C. 简单选择排序通常是稳定的

D. 归并排序可以实现为稳定排序

9. 从 8 名学生中选出 3 人参加活动，要求甲、乙两人中至少有一人被选中。不同的选择方式有 () 种。

A. 20

B. 28

C. 36

D. 42

10. 在一个有序数组中进行二分查找。若数组中有 100 个元素，则最坏情况下最多需要比较 () 次。

A. 6

B. 7

C. 8

D. 100

11. 有 4 个字符的出现频率分别为 $4, 5, 6, 7$。若使用哈夫曼编码，则对应哈夫曼树的带权路径长度为 ()。

A. 35
B. 39
C. 44
D. 52

12. 阅读以下递归函数，`f(5)` 的返回值为 ()。

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

A. 3
B. 5
C. 8
D. 13

13. 以下哪个不是操作系统？()

A. Linux
B. Windows
C. Android
D. HTML

14. 若定义 `int a[8];`，下列说法正确的是 ()。

A. 数组 `a` 中共有 8 个 `int` 类型元素
B. 数组 `a` 的合法下标为 1 到 8
C. `a[8]` 表示数组最后一个元素
D. 数组 `a` 中共有 9 个 `int` 类型元素

15. 在数据压缩中，哈夫曼编码方法在构造编码树时主要体现了哪种算法思想？()

A. 枚举
B. 贪心
C. 二分
D. 搜索

（二）阅读程序

程序 1

```
#include <iostream>
using namespace std;

int f(int x) {
    int s = 0;
```

```
    while (x > 0) {
        s += x % 2;
        x /= 2;
    }
    return s;
}

int main() {
    int n;
    cin >> n;

    int cnt = 0, last = 0;
    for (int i = 1; i <= n; i++) {
        if (f(i) == 2) {
            cnt++;
            last = i;
        }
    }

    cout << cnt << " " << last << endl;
    return 0;
}
```

假设输入的 n 是不超过 1000 的正整数，完成下面的判断题和单选题。

判断题

16. `f(7)` 的返回值为 3。()
17. 当输入为 `5` 时，程序输出为 `2 5`。()
18. 该程序统计的是 1 到 n 中所有偶数的个数。()

单选题

19. 当输入为 `10` 时，程序输出为 ()。

    A. `4 10`
    B. `5 10`
    C. `5 9`
    D. `6 10`

20. 若将 `if (f(i) == 2)` 改为 `if (f(i) <= 2)`，当输入为 `5` 时，程序输出为 ()。

    A. `2 5`
    B. `3 5`
    C. `4 5`
    D. `5 5`

21. 该程序的时间复杂度最接近 ()。

    A. $O(n)$

B. $O(n \log n)$

C. $O(n^2)$

D. $O(2^n)$

程序 2

```cpp
#include <iostream>
#include <string>
using namespace std;

string h(string s) {
    string t;
    for (int i = 0; i < s.size(); i++) {
        if (s[i] == '#' && !t.empty()) {
            t.erase(t.size() - 1, 1);
        } else if (s[i] != '#') {
            t += s[i];
        }
    }
    return t;
}

int main() {
    string a, b;
    cin >> a >> b;

    string x = h(a);
    string y = h(b);

    cout << (x == y) << " " << x.size() + y.size() << endl;
    return 0;
}
```

假设输入字符串只由小写英文字母和字符 `#` 组成，且长度不超过 100。字符 `#` 表示删除当前结果的最后一个字符；若当前结果为空，则这个 `#` 不产生影响。完成下面的判断题和单选题。

判断题

22. 当输入为 `ab#c ac` 时，程序输出为 `1 4`。()

23. `h("#a")` 的返回值为 `#a`。()

24. `h("a##")` 的返回值为空字符串。()

单选题

25. 当输入为 `a#bc b#bc` 时，程序输出为 ()。

A. `1 4`

B. `0 4`

C. `1 2`

D. `0 2`

26. 当输入为 `abc#d abdd#` 时，程序输出为 ()。

A. `0 6`
B. `1 6`
C. `1 4`
D. `0 4`

27. 若将 `t.erase(t.size() - 1, 1);` 改为 `t += s[i];`，当输入为 `ab#c ac` 时，程序输出为 ()。

A. `1 4`
B. `0 4`
C. `0 6`
D. `1 6`

程序 3

```
#include <iostream>
using namespace std;

const int N = 105;
int a[N], dp[N];

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

    if (n == 1) {
        cout << a[1] << endl;
        return 0;
    }

    dp[1] = a[1];
    dp[2] = max(a[1], a[2]);

    for (int i = 3; i <= n; i++) {
        dp[i] = max(dp[i - 1], dp[i - 2] + a[i]);
    }

    cout << dp[n] << endl;
    return 0;
}
```

假设输入的 n 是不超过 100 的正整数，数组元素均为正整数。完成下面的判断题和单选题。

判断题

28. 当输入为 `4 1 2 3 1` 时，程序输出为 `4`。()
29. 当输入的 $n = 1$ 时，程序会直接输出 `a[1]` 的值。()
30. `dp[i]` 表示从前 i 个数中选出若干个互不相邻的数时，可以得到的最大和。()

单选题

31. 当输入为 `5 2 7 9 3 1` 时，程序输出为（）。

A. `11`
B. `12`
C. `13`
D. `22`

32. 若将 `dp[2] = max(a[1], a[2]);` 改为 `dp[2] = a[1] + a[2];`，当输入为 `4 5 1 1 5` 时，程序输出为（）。

A. `6`
B. `10`
C. `11`
D. `12`

33. 该程序的时间复杂度最接近（）。

A. $O(n)$
B. $O(n \log n)$
C. $O(n^2)$
D. $O(2^n)$

（三）完善程序

程序 1（查找第一个不小于给定值的位置）

给定一个长度为 n 的升序数组 a 和一个整数 x，要求输出数组中第一个满足 $a_i \geq x$ 的位置。如果不存在这样的元素，输出 `-1`。数组下标从 1 开始。

试补全程序。

```
#include <iostream>
using namespace std;

const int N = 1005;
int a[N];

int main() {
    int n, x;
    cin >> n >> x;
    for (int i = 1; i <= n; i++) {
        cin >> a[i];
    }

    int l = 1, r = n, ans = -1;
    while (①) {
        int mid = ②;
        if (③) {
            ans = ④;
```

```
            r = mid - 1;
        } else {
            ⑤;
        }
    }

    cout << ans << endl;
    return 0;
}
```

34. ①处应填（）

A. `l <= r`
B. `l < 0`
C. `r < l`
D. `ans != -1`

35. ②处应填（）

A. `l + r`
B. `(l + r) / 2`
C. `r - l`
D. `(r - l) * 2`

36. ③处应填（）

A. `a[mid] < x`
B. `a[mid] == 0`
C. `a[mid] >= x`
D. `mid >= x`

37. ④处应填（）

A. `mid`
B. `a[mid]`
C. `x`
D. `l + r`

38. ⑤处应填（）

A. `r = mid - 1`
B. `l = mid - 1`
C. `r = mid + 1`
D. `l = mid + 1`

程序 2（网格路径计数）

给定一个 n 行 m 列的网格，`.` 表示可以经过的位置，`#` 表示障碍。现在从左上角 `(1,1)` 出发，每一步只能向下或向右走一格，不能经过障碍。求走到右下角 `(n,m)` 的不同路径条数。

试补全程序。

```
#include <iostream>
using namespace std;

const int N = 35;
char mp[N][N];
long long dp[N][N];

int main() {
    int n, m;
    cin >> n >> m;

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            cin >> mp[i][j];
            dp[i][j] = 0;
        }
    }

    if (①) {
        dp[1][1] = 1;
    } else {
        dp[1][1] = 0;
    }

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            if (②) {
                dp[i][j] = 0;
                continue;
            }

            if (③) {
                continue;
            }

            if (i > 1) {
                dp[i][j] += ④;
            }
            if (j > 1) {
                dp[i][j] += ⑤;
            }
        }
    }

    cout << dp[n][m] << endl;
    return 0;
}
```

39. ①处应填（）

A. `mp[1][1] == '.'`

B. `mp[1][1] == '#'`

C. `n == m`

D. `dp[1][1] == 1`

40. ②处应填（）

A. `mp[i][j] == '#'`

B. `mp[i][j] == '.'`

C. `dp[i][j] > 0`

D. `i == n && j == m`

41. ③处应填（）

A. `i == 1 && j == 1`

B. `i == n && j == m`

C. `i > 1`

D. `j > 1`

42. ④处应填（）

A. `dp[i - 1][j]`

B. `dp[i][j - 1]`

C. `dp[i + 1][j]`

D. `dp[i][j + 1]`

43. ⑤处应填（）

A. `dp[i - 1][j]`

B. `dp[i][j - 1]`

C. `dp[i + 1][j]`

D. `dp[i][j + 1]`

二、答案与解析

答案速查

题号	1	2	3	4	5	6	7	8	9	10	11
答案	B	C	D	C	B	C	A	C	C	B	C

题号	12	13	14	15	16	17	18	19	20	21	22
答案	C	D	A	B	✓	✓	✗	B	D	B	✓

题号	23	24	25	26	27	28	29	30	31	32	33
答案	✗	✓	A	B	C	✓	✓	✓	B	C	A

题号	34	35	36	37	38	39	40	41	42	43
答案	A	B	C	A	D	A	A	A	A	B

（一）单项选择题解析

第 1 题，`const int x = 5;` 定义的是整型常量，之后不能通过 `x = ...` 直接修改。`unsigned` 表示无符号，`static` 与变量存储特性有关，都不表示“不能修改”。

第 2 题，$10110_2 = 16 + 4 + 2 = 22$，$35_8 = 3 \times 8 + 5 = 29$。两者相加得到 $22 + 29 = 51$，选 C。

第 3 题，D 不可能。若先输出 3，说明 $1, 2, 3$ 已经入栈，并且 3 已出栈。此时栈顶是 2，不能越过 2 直接输出 1。栈题要盯住当前栈顶，而不是只看数字是否出现过。

第 4 题，高度为 6 的二叉树最多有 $2^6 - 1 = 63$ 个结点，不能容纳 100 个结点；高度为 7 的二叉树最多有 $2^7 - 1 = 127$ 个结点，可以容纳 100 个结点。因此高度至少为 7。

第 5 题，完全二叉树用 1 下标数组存储时，下标为 i 的结点，其父结点下标为 $\lfloor i/2 \rfloor$，右孩子下标为 $2i + 1$。当 $i = 11$ 时，父结点下标为 5，右孩子下标为 23。

第 6 题，按后缀表达式规则从左到右计算。`1 2 +` 得到 3，再与 4 相乘得到 12，再加上 5 得到 17，最后减去 3 得到 14。

第 7 题，外层循环中 i 依次为 $1, 2, 4, 8, \ldots$。内层执行次数分别约为 $1, 2, 4, 8, \ldots$，这些数相加不超过 $2n$ 的数量级，所以时间复杂度是 $O(n)$。这题不能只看到两层循环就选 $O(n^2)$。

第 8 题，简单选择排序在常见实现中会把后面的最小元素交换到前面，可能改变相等元素的相对顺序，所以通常不是稳定排序。冒泡排序、插入排序通常是稳定的，归并排序也可以实现为稳定排序。

第 9 题，总方案数为 $\binom{8}{3} = 56$。若甲、乙都不被选中，则只能从剩下 6 人中选 3 人，有 $\binom{6}{3} = 20$ 种。至少有一人被选中的方案数为 $56 - 20 = 36$。

第 10 题，二分查找每比较一次大约排除一半范围。$2^6 = 64$，$2^7 = 128$，因此 100 个元素在最坏情况下最多需要 7 次比较。

第 11 题，哈夫曼树每次合并权值最小的两个结点。先合并 4 和 5 得到 9，再合并 6 和 7 得到 13，最后合并 9 和 13 得到 22。带权路径长度等于每次合并产生的新权值之和，即 $9 + 13 + 22 = 44$。

第 12 题，由代码可知 `f(1)=1`，`f(2)=2`。继续计算得到 `f(3)=3`，`f(4)=5`，`f(5)=8`。答案为 C。

第 13 题，Linux、Windows、Android 都是操作系统。HTML 是超文本标记语言，不是操作系统。

第 14 题，`int a[8];` 表示数组中有 8 个 `int` 类型元素，合法下标为 0 到 7，最后一个元素是 `a[7]`。因此 A 正确。

第 15 题，哈夫曼编码构造过程中，每次选取当前权值最小的两个结点合并，体现的是贪心策略。

（二）阅读程序解析

程序 1 解析

函数 `f(x)` 每次取 `x % 2`，再令 `x /= 2`，这是按二进制从低位到高位统计 1 的个数。因此 `f(x)` 返回的是 x 的二进制表示中 1 的个数。

第 16 题，7 的二进制表示为 111_2，其中有 3 个 1，所以 `f(7)` 返回 3，判断正确。

第 17 题，1 到 5 中，二进制表示中恰好有两个 1 的数是 3 和 5。程序输出个数和最后一个符合条件的数，所以输出 `2 5`，判断正确。

第 18 题，程序统计的不是偶数个数，而是二进制中 1 的个数恰好为 2 的数的个数，所以判断错误。

第 19 题，1 到 10 中符合条件的数为 $3, 5, 6, 9, 10$，共有 5 个，最后一个是 10，所以输出 `5 10`，选 B。

第 20 题，修改后统计 `f(i) <= 2` 的数。1 到 5 中每个数的二进制 1 的个数都不超过 2，所以共有 5 个，最后一个是 5，输出 `5 5`。

第 21 题，外层枚举 1 到 n，每次调用 `f(i)`。函数 `f(i)` 的循环次数与 i 的二进制位数同级，最多为 $O(\log n)$。总时间复杂度最接近 $O(n \log n)$。

程序 2 解析

函数 `h` 从左到右扫描字符串，用 `t` 保存当前处理结果。遇到普通字符就追加到 `t` 末尾；遇到 `#` 且 `t` 非空时，就删去 `t` 的最后一个字符；遇到 `#` 且 `t` 为空时，不做处理。

第 22 题，`h("ab#c")` 的过程为：加入 `a`，加入 `b`，遇到 `#` 删除 `b`，再加入 `c`，结果为 `ac`。`h("ac")` 也是 `ac`。两个结果相同，总长度为 $2 + 2 = 4$，输出 `1 4`，判断正确。

第 23 题，`h("#a")` 中第一个 `#` 出现时 `t` 为空，因此不产生影响，之后加入 `a`，返回值是 `a`，不是 `#a`，判断错误。

第 24 题，`h("a##")` 中先加入 `a`，第一个 `#` 删除 `a`，第二个 `#` 不产生影响，最终为空字符串，判断正确。

第 25 题，`h("a#bc")` 的结果为 `bc`，`h("b#bc")` 的结果也是 `bc`。二者相同，总长度为 4，输出 `1 4`。

第 26 题，`h("abc#d")` 的结果为 `abd`，`h("abdd#")` 的结果也是 `abd`。二者相同，总长度为 $3 + 3 = 6$，输出 `1 6`。

第 27 题，修改后，遇到 `#` 且 `t` 非空时，不再删除最后一个字符，而是把 `#` 加入结果。输入 `ab#c ac` 时，第一个字符串处理后为 `ab#c`，第二个字符串处理后为 `ac`。二者不同，总长度为 $4 + 2 = 6$，输出 `0 6`。

程序 3 解析

这段程序求的是：从序列中选择若干个互不相邻的数，使得它们的和最大。数组元素均为正整数，因此每个位置只有两种关键选择：选当前位置，或者不选当前位置。

`dp[i]` 表示只考虑前 i 个数时，能得到的最大和。若不选第 i 个数，结果为 `dp[i - 1]`；若选第 i 个数，则第 $i-1$ 个数不能选，结果为 `dp[i - 2] + a[i]`。因此转移为 `dp[i] = max(dp[i - 1], dp[i`

- 2] + a[i])。

第 28 题，输入数组为 $1,2,3,1$。可选的最优方案是选第 1 个和第 3 个，和为 4，程序输出 4，判断正确。

第 29 题，代码中当 n == 1 时，会执行：

```
cout << a[1] << endl;
return 0;
```

所以判断正确。

第 30 题，dp[i] 的含义正是前 i 个数中选择互不相邻若干数时的最大和，判断正确。

第 31 题，输入数组为 $2,7,9,3,1$。可选第 $1,3,5$ 个数，和为 $2+9+1=12$；其他方案不会更大，所以输出 12。

第 32 题，修改后输入数组为 $5,1,1,5$。有 dp[1]=5，dp[2]=6。接着 dp[3]=max(6,5+1)=6，dp[4]=max(6,6+5)=11，输出 11。这个修改破坏了“不能选相邻位置”的含义。

第 33 题，程序读入数组后，只用一个从 3 到 n 的循环完成转移，因此时间复杂度为 $O(n)$。

（三）完善程序解析

程序 1 解析

这段程序使用二分查找，寻找升序数组中第一个不小于 x 的位置。变量 ans 保存当前已经找到的候选位置。

第 34 题，二分查找区间为 [l,r]。当 l <= r 时，区间中仍有元素需要判断，所以 ① 应填 l <= r。

第 35 题，mid 应为当前区间中点，② 应填 (l + r) / 2。

第 36 题，要找第一个满足 $a_i \geq x$ 的位置，因此当 a[mid] >= x 时，mid 是一个候选答案，③ 应填 a[mid] >= x。

第 37 题，题目要求输出位置，不是输出数组元素的值，所以找到候选位置时应执行 ans = mid，④ 应填 mid。

第 38 题，当 a[mid] < x 时，mid 及其左侧都不可能满足要求，应继续查找右半边，所以应执行 l = mid + 1，⑤ 应填这一项。

程序 2 解析

这段程序用动态规划统计从左上角到右下角的路径数。dp[i][j] 表示走到位置 (i,j) 的路径条数。

第 39 题，只有当起点 (1,1) 可以经过，即 mp[1][1] == '.' 时，起点路径数才初始化为 1，所以 ① 选 A。

第 40 题，若当前位置是障碍 #，则不能经过，路径数应为 0 并跳过后续转移，所以 ② 应填 mp[i][j] == '#'。

第 41 题，起点已经在循环前初始化过。在循环中再次处理起点时，应直接跳过，避免从上方或左方转移，所以 ③ 应填 `i == 1 && j == 1`。

第 42 题，如果可以从上方走到当前位置，那么应把上方位置的路径数加到当前位置，即 `dp[i][j] += dp[i - 1][j]`，所以 ④ 选 A。

第 43 题，如果可以从左方走到当前位置，那么应把左方位置的路径数加到当前位置，即 `dp[i][j] += dp[i][j - 1]`，所以 ⑤ 选 B。

这类完善程序题应先判断 `dp[i][j]` 的含义，再看每个空缺负责的是初始化、障碍处理、起点跳过，还是从相邻位置转移。

附录

附录 A 运算符优先级

一、常用优先级表

下表按优先级从高到低排列。CSP-J 第一轮不要求背下完整 C++ 运算符表，但下表中的常见层次应能判断。

优先级	运算符或形式	含义	结合方向	提醒
1	`()`、`[]`、`.`、`->`、`f(x)`	括号、下标、成员访问、函数调用	从左到右	括号、数组下标、函数调用优先级很高
2	`i++`、`i--`	后置自增、后置自减	从左到右	表达式中先使用旧值，再修改变量
3	`++i`、`--i`、`!`、`~`、`-x`、`+x`、`*p`、`&x`、`sizeof`	前置自增、自减、逻辑非、按位取反、单目正负、解引用、取地址、求大小	从右到左	前置自增先修改再使用；`!` 用于真假取反
4	`*`、`/`、`%`	乘、除、取余	从左到右	整数除法会舍去小数部分
5	`+`、`-`	加、减	从左到右	低于乘除取余，高于移位和比较
6	`<<`、`>>`	左移、右移；也可出现在输入输出语句中	从左到右	加减优先级高于移位
7	`<`、`<=`、`>`、`>=`	大小比较	从左到右	比较结果是真假值
8	`==`、`!=`	相等、不等比较	从左到右	`==` 是比较，`=` 是赋值
9	`&`	按位与	从左到右	和取地址符号相同，但用法不同
10	`^`	按位异或	从左到右	常出现在二进制相关程序中
11	`|`	按位或	从左到右	比按位异或低，比逻辑与高
12	`&&`	逻辑与	从左到右	两个条件都为真，整体才为真

优先级	运算符或形式	含义	结合方向	提醒
13	`||`	逻辑或	从左到右	至少一个条件为真，整体为真
14	`?:`、`=`、`+=`、`-=`、`*=`、`/=`、`%=`	条件运算、赋值、复合赋值	从右到左	`a = b = c` 应理解为 `a = (b = c)`
15	`,`	逗号运算符	从左到右	考得少，不要和函数参数分隔逗号混淆

二、最常用层次

考试中最常见的是下面这些层次。记不住完整表时，至少要能判断这些运算谁先谁后。

顺序	类型	例子	速查提醒
1	括号、下标、函数调用	`(a + b)`、`a[i]`、`f(x)`	先处理被括起来、被下标访问或被调用的部分
2	自增、自减、单目运算	`i++`、`++i`、`!ok`	注意前置和后置区别
3	乘除取余	`a * b`、`a / b`、`a % b`	高于加减
4	加减	`a + b`、`a - b`	高于移位和比较
5	移位	`x << 1`、`x >> 1`	低于加减，高于比较
6	大小比较	`a < b`、`a >= b`	结果是真假值
7	相等比较	`a == b`、`a != b`	低于大小比较，高于按位与
8	按位运算	`a & b`、`a ^ b`、`a | b`	从高到低为按位与、按位异或、按位或
9	逻辑与	`p && q`	两个条件都要满足
10	逻辑或	`p || q`	满足一个条件即可
11	赋值	`x = y`、`x += 1`	会修改左边变量，从右向左结合
12	逗号运算符	`a, b`	优先级最低，考试中较少单独考

三、常见表达式加括号

复杂表达式不要只凭直觉从左到右读。下面这些表达式可以在草稿上先改写成“等价理解”，再计算。

原表达式	等价理解	提醒
`a + b * c`	`a + (b * c)`	乘法高于加法
`a * b + c`	`(a * b) + c`	先乘后加
`x + 1 < y`	`(x + 1) < y`	加法高于大小比较
`x % 2 == 0`	`(x % 2) == 0`	先取余，再判断是否等于 0
`x == y && z > 0`	`(x == y) && (z > 0)`	比较运算先得到真假值，再做逻辑与
`x < 1 || x > 100`	`(x < 1) || (x > 100)`	两个比较结果再做逻辑或
`a = b = c`	`a = (b = c)`	赋值运算从右向左结合
`cout << a + b`	`cout << (a + b)`	加法高于 `<<`
`a + b << 1`	`(a + b) << 1`	加法高于移位
`!x == y`	`(!x) == y`	逻辑非高于相等比较
`i++ + 1`	`(i 的旧值) + 1`	后置自增先使用旧值，再让 `i` 增加 1
`++i + 1`	`(i 增加 1 后的新值) + 1`	前置自增先修改，再使用新值

四、易错写法

写法或情况	正确判断
`=` 和 `==` 混淆	`=` 是赋值，会修改左边变量；`==` 是比较，结果是真假值
`1 <= x <= 100`	不能表示 $1 \le x \le 100$，应写成 `1 <= x && x <= 100`
`i++` 和 `++i`	单独成句时最终都让 `i` 增加 1；放在表达式中，使用的值不同
`x / y` 两边都是整数	结果是整数，直接舍去小数部分
`x % y`	表示余数，常用于判断整除、奇偶和循环下标
`a | b == c`	应先判断优先级，不要把它误读成 `(a | b) == c`
`a && b || c`	`&&` 高于逻辑或，应理解为 `(a && b) || c`
复杂逻辑表达式	先分别算比较表达式，再算逻辑与，最后算逻辑或
不确定优先级	主动加括号，不要赌记忆
一条语句中多次修改同一变量	阅读题中要谨慎追踪；平时写程序应避免这种写法

五、考场检查

检查项	提醒
是否有括号	括号内先算，括号可以改变原来的优先级
是否有自增自减	看清前置还是后置
是否有整数除法	注意是否会舍去小数部分
是否有赋值	赋值会改变变量，不只是判断
是否有逻辑运算	先算每个条件真假，再合并判断
是否有按位运算	不熟悉时先把相关数写成二进制
是否有移位运算	注意加减高于移位
是否有输出语句	`cout << a + b` 中通常先算 `a + b`
是否看错题意	题目问“表达式的值”还是“执行后变量的值”
是否能加括号	能加括号时，先在草稿上补出等价括号

附录 B 进制与存储单位

一、常见进制

进制	可用数字	常见写法	例子
二进制	$0, 1$	下标 2	1010_2
八进制	0 到 7	下标 8，C++ 中整数前导 `0`	17_8，`017`
十进制	0 到 9	通常不写下标	123
十六进制	0 到 9，A 到 F	下标 16，C++ 中前缀 `0x`	$2F_{16}$，`0x2F`

二、十六进制字母

十六进制	十进制	十六进制	十进制
A	10	D	13
B	11	E	14
C	12	F	15

三、常用权值

位数	二进制权值	八进制权值	十六进制权值
第 0 位	1	1	1
第 1 位	2	8	16
第 2 位	4	64	256
第 3 位	8	512	4096
第 4 位	16	4096	65536
第 5 位	32	32768	1048576
第 6 位	64	262144	16777216
第 7 位	128	2097152	268435456

四、常见二进制数

二进制	十进制	二进制	十进制
1_2	1	10000_2	16
10_2	2	100000_2	32
100_2	4	1000000_2	64
1000_2	8	10000000_2	128

五、进制转换方法

任务	方法	速查提醒
其他进制转十进制	按权值展开	从右向左依次乘 1、进制、进制平方
十进制转其他进制	除进制取余，余数倒读	转十六进制时，10 到 15 写成 A 到 F
二进制转八进制	从右向左每 3 位一组	不足 3 位在高位补 0
二进制转十六进制	从右向左每 4 位一组	不足 4 位在高位补 0
八进制转二进制	每个八进制数字写成 3 位二进制	最前面多余的 0 可省略
十六进制转二进制	每个十六进制数字写成 4 位二进制	A 到 F 先换成 10 到 15

六、快速换算例子

表达式	结果	过程提醒
101101_2	45	$32+8+4+1$
32_8	26	$3\times 8+2$
$2F_{16}$	47	$2\times 16+15$
101.11_2	5.75	整数部分 5，小数部分 $\frac{1}{2}+\frac{1}{4}$
32.1_8	26.125	整数部分 26，小数部分 $\frac{1}{8}$
101101_2 转八进制	55_8	分组为 $101, 101$
101101_2 转十六进制	$2D_{16}$	补成 $0010, 1101$

七、存储单位

单位	含义	换算
bit	二进制位	一个 bit 只能表示 0 或 1
byte	字节	1byte = 8bit
KB	千字节	1KB = 1024byte
MB	兆字节	1MB = 1024KB
GB	吉字节	1GB = 1024MB

八、常用容量值

表达式	结果
2^{10}	1024
2^{20}	1048576
2^{30}	1073741824
1KB	1024byte
1MB	1048576byte
1MB	8388608bit
2MB	16777216bit

九、二进制位数与范围

位数	可表示状态数	无符号整数范围
1 位	2 种	0 到 1
2 位	4 种	0 到 3
4 位	16 种	0 到 15
8 位	256 种	0 到 255
16 位	65536 种	0 到 65535
32 位	2^{32} 种	0 到 $2^{32}-1$

十、原码、反码与补码

内容	速查
负数补码	绝对值写成二进制，按位取反，再加 1
n 位有符号整数范围	-2^{n-1} 到 $2^{n-1}-1$
8 位补码例子	`11111111` 表示 -1，`10000000` 表示 -128

十一、做题检查

检查项	说明
看下标	1010_2、1010_8、1010_{16} 不是同一个数
看前缀	C++ 中 `010` 是八进制，`0x10` 是十六进制
看单位	byte 是字节，bit 是二进制位
记得乘 8	字节换成 bit 时要乘 8
不把 1024 写成 1000	题目按计算机存储单位时通常用 1024
混合进制先统一	多个进制混合计算，先全部转成十进制

附录 C 数据结构速查

一、线性结构

数据结构	核心规则	常见操作	常见考法
数组	连续存储，可按下标访问	`a[i]`、遍历、修改	下标范围、长度、越界、前缀和、排序后访问
字符串	字符组成的序列	`s[i]`、`s.length()`、拼接、比较	字符下标、长度、子串、逐字符处理
链表	每个结点保存数据和指向后继的指针	插入、删除、遍历	指针修改顺序、头插、尾插、删除结点
栈	后进先出	`push`、`pop`、`top`、`empty`	出栈序列、括号匹配、表达式求值、递归过程
队列	先进先出	`push`、`pop`、`front`、`empty`	BFS、排队模拟、层次遍历

二、数组与字符串

项目	速查
长度为 n 的数组	若从 0 开始，下标为 0 到 $n-1$
`int a[10]`	有 10 个元素，最后一个合法下标是 9
`string s`	`s.length()` 或 `s.size()` 表示长度
字符串最后一个字符	若长度为 n，下标为 $n-1$
常见错误	把长度当最大下标，把 `a[n]` 当最后一个元素
阅读程序重点	循环边界、下标起点、是否访问了不存在的位置

三、链表

操作	关键语句	速查提醒
访问当前结点数据	`p->data`	p 必须指向一个有效结点
访问后继	`p->next`	若 p 是尾结点，后继可能为空
头插新结点	`s->next = head; head = s;`	先让新结点接上原头结点，再改头指针
在 p 后插入 s	`s->next = p->next; p->next = s;`	顺序不能反，否则原后继会丢失
删除 p 后的结点	`p->next = p->next->next;`	要确认 `p->next` 存在
双向链表插入	同时修改前驱和后继	四个指针关系都要接好

四、栈

项目	速查
访问规则	后进先出
入栈	把新元素放到栈顶
出栈	删除栈顶元素
取栈顶	只能查看当前栈顶
空栈风险	空栈不能执行 `top()` 或 `pop()`
出栈序列判断	按目标序列模拟，目标必须能到达栈顶
括号匹配	左括号入栈，右括号检查栈顶是否匹配
表达式求值	操作数入栈，遇到运算符取出需要的操作数计算

五、队列

项目	速查
访问规则	先进先出
入队	元素进入队尾
出队	队首元素离开
查看队首	`front()`
常见用途	BFS、排队模拟、树的层次遍历
BFS 过程	起点入队；每次取队首；扩展未访问邻点；新点入队
常见错误	忘记标记访问，导致重复入队

六、树与二叉树

概念	速查
根结点	没有父结点的结点
叶结点	没有孩子的结点
深度	从根到该结点经过的层数，具体从 0 还是 1 开始要看题面
高度	从该结点到最深叶结点的层数，具体从 0 还是 1 开始要看题面
二叉树	每个结点最多有两个孩子
满二叉树	每一层结点都达到最大数量
完全二叉树	除最后一层外都满，最后一层从左到右连续
二叉搜索树	左子树较小，右子树较大，具体是否允许相等要看题面

七、二叉树数组下标

下表默认根结点下标为 1。

结点位置	下标关系
当前结点	i
父结点	$\lfloor i/2 \rfloor$
左孩子	$2i$
右孩子	$2i+1$
左兄弟或右兄弟	相邻下标需结合奇偶判断
最后一层是否存在	要看下标是否不超过结点总数 n

八、二叉树遍历

遍历方式	顺序	速查提醒
前序遍历	根、左、右	第一个元素是当前子树根
中序遍历	左、根、右	可用根把左右子树分开
后序遍历	左、右、根	最后一个元素是当前子树根
层序遍历	从上到下、从左到右	常用队列实现

由前序和中序确定二叉树时，先用前序第一个元素找根，再到中序中分出左子树和右子树。由后序和中序确定二叉树时，先用后序最后一个元素找根。

九、图

概念	速查
顶点	图中的点
边	顶点之间的连接
无向图	边没有方向
有向图	边有方向
度数	无向图中与某点相连的边数
入度	有向图中指向该点的边数
出度	有向图中从该点指出的边数
连通图	任意两个顶点之间都有路径相连
树	连通且无环的无向图
有向无环图	没有有向环的有向图，常用于拓扑排序

十、图的常用结论

结论	速查
无向图度数和	等于边数的 2 倍
有向图入度和	等于边数
有向图出度和	等于边数
n 个点的树	有 $n-1$ 条边
邻接矩阵	适合点数较小、需要快速判断两点是否有边
邻接表	适合边数较少、需要遍历邻点
DFS	常配合递归或栈
BFS	常配合队列

十一、哈夫曼树与哈夫曼编码

项目	速查
构造规则	每次选权值最小的两个结点合并
新结点权值	两个被合并结点权值之和
编码长度	等于字符所在叶结点深度
高频字符	通常应对应较短编码
前缀编码	任意一个字符编码不能是另一个字符编码的前缀
编码不唯一	左右子树标 0 或 1 可以互换
判断选项	看码长、前缀关系和带权路径长度，不只看具体 $0,1$ 串

十二、排序速查

排序	常见时间复杂度	稳定性	速查提醒
冒泡排序	$O(n^2)$	通常稳定	相邻元素比较交换
选择排序	$O(n^2)$	通常不稳定	每轮选出最小或最大
插入排序	$O(n^2)$	通常稳定	把当前元素插入前面有序部分
归并排序	$O(n\log n)$	可稳定	分治合并
快速排序	平均 $O(n\log n)$	通常不稳定	选基准划分
堆排序	$O(n\log n)$	通常不稳定	利用堆维护最大或最小

十三、考前快速判断

题目关键词	优先想到
后进先出、括号匹配、出栈序列	栈
先进先出、最短步数、层次遍历	队列
递归遍历、根左右、左右根	二叉树遍历
度数和、连通、边数	图的基本性质
每次合并两个最小权值	哈夫曼编码
下标 $2i$、$2i+1$	完全二叉树数组存储
判断两点是否有边	邻接矩阵或邻接表
从一个点扩展到相邻点	DFS 或 BFS

附录 D 考前检查表

一、进考场前

检查项	做法
证件与文具	按考试要求准备准考证、身份证件、黑色签字笔、铅笔、橡皮等
时间	提前到达考场，避免临近开考才进场
心态	加油喵
草稿	进制换算、变量追踪、循环过程都应写在草稿上
答题习惯	题目问“正确”还是“错误”，先圈关键词

二、单项选择题检查

检查项	提醒
是否有否定词	“不正确”“错误”“不是”“不属于”要先圈出
进制是否看清	1010_2、1010_8、1010_{16} 含义不同
单位是否看清	bit 和 byte 不同，字节换二进制位要乘 8
下标是否从 0 开始	C++ 数组和字符串常从下标 0 开始
= 和 ==	一个是赋值，一个是比较
整数除法	两边都是整数时，结果舍去小数部分
复杂度	看循环次数，不只看循环层数
树和图	度数和、边数、遍历顺序要按定义判断
选项太绝对	“一定”“总是”“所有”要特别谨慎

三、阅读程序检查

步骤	要做什么
先看输入	输入有几个数、几行、是否有数组或字符串
再看输出	输出几个值，是否有空格、换行、固定小数
标变量含义	在草稿上写出 `n`、`i`、`cnt`、`ans`、`sum` 等变量含义
看初始化	初值错，后面过程都会错
看循环边界	注意 `<`、`<=`、起点和终点
看数组下标	是否访问 0 到 $n-1$，还是 1 到 n
看条件分支	先判断条件真假，再决定执行哪一段
看函数返回	函数返回值是什么，是否改变全局变量或引用参数
看递归出口	没有出口或出口看错，递归结果会错
看修改题	若题目问“把某行改成……”，只分析修改后的程序

四、变量追踪检查

检查项	提醒
表格不要太大	只追踪关键变量，不抄整段程序
每轮循环写一次	循环次数少时，可逐轮写 `i`、`ans`、`cnt`
先更新再判断还是先判断再更新	自增、自减和循环末尾更新要看清
后置自增	`i++` 在表达式中先使用旧值
前置自增	`++i` 在表达式中先修改再使用
赋值语句	执行后左边变量立即改变
数组变化	修改 `a[i]` 后，后面再访问的是新值
函数调用	每次调用参数可能不同，不要混用上一层变量

五、完善程序检查

空缺类型	判断方法
初始化	看后面第一次使用前需要什么初值
循环条件	看程序应在什么时候继续、什么时候停止
判断条件	看进入分支需要满足什么题意
状态转移	看当前值应由哪些旧值推出
更新语句	看本轮处理完后，哪个变量要改变
返回值	看函数名和调用处需要什么结果
输出	看题目要求输出编号、数值、次数还是真假
排序规则	看后面的扫描过程需要按什么顺序处理
边界情况	检查 $n=0$、$n=1$、空串、起点终点等特殊情况
排除选项	先排除会越界、死循环、类型不合或含义相反的选项

六、常见边界

场景	检查点
数组长度为 n	最后下标通常是 $n-1$，不是 n
循环 `i < n`	循环变量最后一次通常是 $n-1$
循环 `i <= n`	循环变量会取到 n
字符串为空	不能访问 `s[0]`
栈为空	不能直接 `top()` 或 `pop()`
队列为空	不能直接 `front()` 或 `pop()`
二分查找	检查 `l`、`r` 更新后区间是否变小
递归	检查参数是否逐渐接近出口
取余	除数不能为 0
开平方	题目是否保证被开方数非负

七、最后十分钟

检查项	做法
答题纸	确认所有答案都填在答题纸上
题号	检查是否错位，尤其是阅读程序和完善程序
空题	不留空；不会的题也要根据排除法选择
涂写或书写	保持清楚，避免阅卷无法识别
修改答案	修改要彻底，不要留下两个选项
特殊分值	阅读程序中有些小题分值不同，但题号不能漏
时间分配	不在一道题上耗尽剩余时间
复查顺序	先查容易错位的答题纸，再查计算题和程序追踪题

八、考场提醒

题型	最后提醒
选择题	先圈关键词，再排除明显错误选项
阅读程序	先看输入输出，再追踪关键变量
完善程序	先读题意和代码框架，再判断空缺作用
进制单位题	先统一单位或进制，再计算
数据结构题	先写出访问规则，例如栈后进先出、队列先进先出
递归题	写出前两三层调用和出口
二分题	检查左右边界和中点更新
动态规划题	先判断 `dp` 数组含义，再看初始化和转移

A GUIDE FOR CSP-J ROUND 1

0x22

关于本书

本书面向 CSP-J 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 **考试方法**

先建立选择题、程序阅读与完善程序的作答顺序，再进入知识点训练。

02 **高频考点**

围绕 C++ 行为、进制计数、数据结构、基础算法和字符串处理组织内容。

03 **考前收束**

附录速查、分类训练和检查表用于最后阶段查漏补缺。

J1