C++ 学习笔记

C++ 学习笔记 1
序言 13
使用方法 13
按学习目标选择必读路线 14
零基础起步：先写出程序，再认识常用工具 15
从 C 转到 C++：补齐写法与工具的依赖 15
ACM 入门：沿着题目需要逐步扩展算法 16
C++ 系统学习：把应用接回类型与对象 16
从一个阶段进入下一个阶段 17
分阶段练习索引 17
快速 I/O 与竞赛模板 18
标准流加速 18
基础竞赛模板：使用 ll 类型别名 18
竞赛宏模板：统一使用 long long 19
多组输入、EOF 与输出精度 20
C 与 C++ 的关系 21
一、C++ 语法基础 21
（一） 程序结构与编译 21
1. 第一份程序的结构与语句作用 21
2. 从固定输出到读入后计算 22
3. 源代码的编译与执行 23
（二） 变量、数据类型与初始化 24
1. 声明、初始化与赋值是不同动作 24
2. 从题目范围选择基本类型 25
3. 单字符、文本与逻辑值 26
（三） 常量、类型转换与整数溢出 27
1. 常量的声明与命名 27
2. 整数除法、浮点除法与转换位置 27
3. 乘法结果大于单项：在运算前扩宽 28
4. 自动转换、窄化与有符号混合 29
（四） 运算符与位运算 29
1. 算术、赋值与自增 30
2. 比较与逻辑：从两个条件得到一个结论 30
3. 逻辑短路与表达式求值顺序 31

4. 用二进制位记录选中位置 32
（五） 条件与循环 33
1. if、else if 与 else：选择恰好一条路径 33
2. switch：按离散的值选择动作 34
3. for：初始化、判断、执行、更新 34
4. while 与 do-while：重复次数由状态决定 36
5. break 与 continue 的控制流 37
6. 嵌套循环：内层完整执行后，外层才前进 38
（六） 原生数组与枚举 40
1. 元素、下标与初始化 40
2. 物理长度与当前使用的 n 项 40
3. 下标也能表示类别：计数数组 42
4. 二维数组的行列下标与访问 43
5. 字符数组与字符串结束标记 45
6. 枚举：给有限状态命名 46
（七） 函数、参数与作用域 46
1. 函数的返回类型、参数与执行过程 46
2. 声明让名字可见，定义提供具体操作 48
3. 重载：同名操作接收不同参数类型 48
4. 局部、外层与全局：同名不代表同一对象 49
5. static 局部变量：跨调用保留数据 49
6. 递归的最小例子：暂停调用者，返回后继续 50
（八） 指针、引用与对象生命周期 51
1. 把值传参改成引用或指针传参 52
2. auto 与范围 for：逐项绑定原数组元素 53
3. 动态申请与释放 55
4. 通过引用交换：先保存将被覆盖的旧值 55
5. 指针移动与数组下标的对应关系 56
6. 对象结束生命周期，地址不会自动变成有效新对象 57
（九） auto、lambda 与结构化绑定 58
1. auto 类型推导与引用绑定 58
2. lambda：保存一个小计算动作 59
3. 结构化绑定：副本与引用仍要分清 59
4. auto 推导的是表达式已有类型 60
5. 按值捕获与 mutable 状态更新 61
（十） 输入细节、文件与异常 62
1. 整行读取与残留换行 62
2. 把输入来源换成文件 63

3. 异常的抛出与捕获 64
4. 读取失败后：先清状态，再处理残留数据 64
5. scanf、printf 的格式化输入输出 66
6. throw 与 return 走的是不同路径 67
（十一） 命名空间、头文件与多文件程序 67
1. 命名空间与同名标识符 67
2. 多文件程序的声明、定义与链接 68
（十二） 常用数学函数与数值计算 69
1. 整数坐标的距离计算 70
2. 取整方向要沿数轴判断 71
3. 浮点误差与判断精度 71
（十三） 常用标准流与输出格式 72
1. 标准输入输出流 72
2. iomanip 格式化输出 72
2.1 保留小数位数（`fixed` + `setprecision`） 73
2.2 设置宽度（`setw`） 73
2.3 设置填充字符（`setfill`） 73
2.4 综合示例 73
3. 输出格式状态的保留与重置 74
4. 输出格式的逐字符核对 75
（十四） 模板入门：从相同操作到类型参数 T 75
1. 相同计算在不同类型上的复用 75
2. 推导、显式指定与实例化 77
3. 先做一条记录，再把记录里的类型参数化 77
4. 尖括号里也可能放一个编译期数值 78
5. 标准库的类型工具也按同一方式阅读 79
6. 两个类型参数：同一条记录保存不同类型的字段 80
（十五） STL：容器、迭代器与算法 80
1. string：字符串 81
1.1 用 string 保存和访问文本 81
1.2 string 操作的参数与返回值 81
1.3 完整演示：查找后修改一个片段 82
2. vector：连续动态数组 84
2.1 连续动态数组与初始化 84
2.2 用引用读入运行时长度的数组 84
2.3 size、capacity、reserve 与 resize 86
2.4 vector 常用操作的时间复杂度 87
2.5 完整例题：先压紧，再用区间位置删除尾部 88

2.6 vector 修改与迭代器失效 90
2.7 迭代器的用处：把同一个区间交给不同操作 90
3. pair：成对数据 92
3.1 pair 的两个字段与构造 92
3.2 完整演示：pair 的字典序比较 92
4. 迭代器：访问位置与区间 94
4.1 位置、元素值与半开区间 94
4.2 迭代器的移动、比较与距离运算 94
4.3 值拷贝、引用与只读遍历 94
4.4 完整演示：遍历中安全删除 95
5. `<algorithm>`：通过迭代器操作区间 96
5.1 算法处理区间，容器提供位置 96
5.2 完整例题：去重后统计等值边界 97
5.3 自定义比较规则 99
5.4 从手动压紧到 remove 与 erase 99
6. STL 头文件与包含方式 101
7. list：双向链表 101
7.1 list 的节点存储与双向连接 101
7.2 list 的插入、删除与节点移动 102
7.3 完整演示：定位、插入、删除和移动节点 103
8. stack：栈 104
8.1 栈顶、栈底与后进先出 104
8.2 完整例题：按出栈顺序输出 105
8.3 完整例题：用栈进行括号匹配 106
9. queue：队列 108
9.1 队头、队尾与先进先出 108
9.2 完整例题：循环报数出队 108
10. map：有序键值映射 110
10.1 键、值与有序查找 110
10.2 查询与写入的区别 111
10.3 完整例题：统计后查询单词次数 112
10.4 迭代器的用处：定位一次，修改值或沿有序区间前进 113
11. set：有序键集合 115
11.1 有序、不重复的键集合 115
11.2 set 的插入、删除与定位 115
11.3 完整例题：严格前驱与不小于查询值的后继 116
11.4 自定义记录的排序与唯一性 117
12. unordered_map：哈希键值映射 118

12.1 哈希先定位桶，再比较键 118
12.2 下标、负载与访问失效 119
12.3 完整例题：按查询顺序输出频次 119
13. unordered_set：哈希键集合 120
13.1 unordered_set 的存在性查询 120
13.2 完整例题：按首次出现顺序去重 121
13.3 自定义类型的相等与哈希要一致 122
14. priority_queue：优先队列 123
14.1 从堆顶读取优先级最高的一项 123
14.2 大根堆、小根堆与比较器 125
14.3 完整例题：优先级与平局顺序 126
14.4 完整例题：用小根堆保留最大的 k 项 127
15. deque：双端队列 129
15.1 双端操作与随机访问 129
15.2 完整例题：滑动窗口最小值 130
16. array：定长数组 132
17. multiset 与 multimap：保存重复键 134
17.1 重复是多个独立元素 134
17.2 完整演示：重复删除与按键分组 135
18. forward_list：单向链表 136
19. 根据操作选择容器 138
19.1 操作需求与容器选择 138
19.2 学完各容器后对照迭代器能力 138
19.3 从元素与迭代器理解修改后的访问 139
20. `<numeric>`：填值、求和与求公约数 139
21. tuple：多字段记录 140
22. `<sstream>`：从一行文本读取数据 141
（十六） std::function：保存、传递和更换计算动作 141
1. 从“保存数据”到“保存要执行的动作” 141
2. 把动作作为参数传给另一段处理过程 143
3. 空目标与已保存的动作 143
4. 递归 lambda：先准备名字，再让动作调用这个名字 144
二、面向对象编程 146
（一） 类与对象 146
（二） 封装与访问控制 147
（三） 构造函数、初始化列表与析构函数 148
（四） this、const 成员函数与 static 成员 150
（五） 继承与组合 151

（六） 虚函数、重写与多态 152
（七） 拷贝、移动与资源自动管理 154
（八） 运算符重载与比较规则 156
（九） 面向对象部分的练习 156
三、解题方法论：从题意到代码 157
（一） 先提取对象、变量、限制和目标 157
（二） 把关键词翻译成条件 158
（三） 从要处理的对象确定循环 159
（四） 必要条件、充分条件与等价条件 160
1. 完整示例：分成两个正偶数 160
（五） 从文字规则整理优先级 161
1. 完整示例：闰年判断 161
（六） 从计数要求推导向上取整 162
1. 完整示例：载客人数与最少车辆数 162
（七） 把最优化改成可行性判断 162
1. 完整示例：最小工作量上限 163
（八） 用不变量和反例检查推导 165
（九） 建模与推导中的典型反例 166
1. 连续片段被改成任意选择 166
2. 两个下标被误写成两次取值 167
3. 排序改变了不能重排的对象 168
4. 当前收益被当成全局最优 170
（十） 从样例猜想到证明与构造 171
1. 完整示例：相邻两盏灯的翻转 171
（十一） 从输出目标确定状态信息 173
1. 依据后续行为合并历史 173
2. 根据未来的选择检查遗漏信息 174
（十二） 一题多建模与最优方法的选择过程 174
1. 完整题面与下标对模型 174
2. 排序后按相同取值成批计数 175
3. 固定后来的位置，查询过去的频次 177
4. 从候选模型逐步筛选到最终方案 179
4.1 先核对合法方案和输出目标 179
4.2 把最大输入代入工作量 179
4.3 根据每一步需要的查询作出选择 180
4.4 沿着同一问题变化重新选择方法 181
5. 从所选模型组装提交程序 181
（十三） 复杂题面到事件模拟的完整推导 182

1. 完整示例：多个服务窗口的排队 182
2. 提取对象、时间与动作先后 183
3. 从逐时刻检查改成只处理事件 183
4. 三种调度模型的比较与选择 184
5. 用两份顺序保存不同的选择依据 185
6. 把动作拆成可组合的函数 186
7. 用完整样例逐项核对模型和代码 188
（十四） 从推导到程序职责与错误定位 188
1. 通过第一处不同定位更新次序 189
2. 分开核对模型、算法与实现 189
（十五） 反例构造、缩小输入与小规模对拍 190
1. 每份测试对应一项要验证的规则 190
2. 把失败输入缩成能看清规则的小例子 190
3. 用独立枚举核对优化程序 191
（十六） 从 WA 回到题面与模型 192
1. 先得到一份能解释的失败输入 192
2. 用不同现象区分可能的误读 192
3. 完整推演：从窗口编号错误找回选择规则 193
4. 把推测变成可以重复检查的证据 194
（十七） 解题复盘与模型迁移 194
四、批量输入、大整数运算与提交检查 195
（一） fread 整数输入模板 195
（二） 大整数的加减乘除 198
1. 完整示例：大整数四则计算器 198
2. 数字位的存储、比较与输出 199
3. 加法：逐位相加与进位 201
4. 减法：绝对值比较与逐位借位 203
5. 乘法：位权展开与错位累加 204
6. 除法：逐位确定商和余数 206
6.1 从具体前缀推导下一位商 206
6.2 大整数除以大整数的构建 208
6.3 大整数除以普通整数的简化 209
7. 把正负号接到四则运算 210
7.1 同号相加与异号相抵 210
7.2 乘除符号与整数余数 211
8. 组装计算器与核对结果 212
9. 按数字位数估算复杂度与选用模板 213
（三） 解题和自查顺序 214

五、数据结构与算法 214
（一） 时间复杂度与空间复杂度 215
1. 输入规模、基本操作与大 O 215
2. 增长曲线与数量对照表 216
3. 顺序、嵌套与变长循环的复杂度 219
4. 双指针、图遍历与 DP 的总工作量 220
5. 递归的时间与空间要分开算 221
6. 数据规模与算法选择 222
7. 多组数据、最坏情况与均摊 223
8. 空间复杂度与内存用量换算 223
9. 算法实现的复杂度核对 225
（二） 枚举与模拟 225
1. 枚举数对 225
2. 枚举子集 226
（三） 前缀和与差分 228
1. 一维前缀和 228
2. 二维前缀和 230
3. 差分 232
（四） 排序与离散化 234
1. 常见排序思路 234
2. 归并排序模板 235
3. 离散化 237
（五） 基础数论 238
1. 最大公约数与最小公倍数 239
2. 素数判断 240
3. 埃氏筛 241
4. 试除法分解质因数 243
（六） 快速幂 244
1. 把指数拆成二进制位 245
2. 普通整数快速幂 245
3. 取模快速幂 247
4. 正确性与应用 248
（七） 哈希（散列）表 249
1. 哈希值、桶下标与冲突 249
2. 线性探测：沿后续槽位找空位 250
3. 桶内存储：在同一桶中逐个比较 251
4. 装载因子、rehash 与哈希设计 253
（八） 二叉树 254

1. 二叉树的结构与性质 255
1.1 左右子树与五种结构形态 255
1.2 节点的度、深度与高度 256
1.3 常见二叉树类型与数量关系 257
1.3.1 满、完全、搜索与平衡二叉树 257
1.3.2 节点数、高度与叶子数的关系 259
1.3.3 用二叉树性质计算节点数与高度 260
1.4 二叉树的遍历 260
1.4.1 层序遍历 261
1.4.2 前序遍历 261
1.4.3 中序遍历 262
1.4.4 后序遍历 263
1.4.5 遍历序列与二叉树的唯一性 263
2. 按层序插入与删除的二叉树实现 264
2.1 二叉树节点类 264
2.2 BinaryTree 的接口与成员操作 265
2.2.1 先确定类的接口与所有权 265
2.2.2 层序寻找插入位置 266
2.2.3 队列驱动层序遍历 268
2.2.4 前序：根、左、右 269
2.2.5 中序：左、根、右 271
2.2.6 后序：左、右、根 271
2.2.7 用最后一个节点替换目标并删除叶子 271
2.2.8 组合调用与输出 274
【九】 双指针与滑动窗口（尺取法） 275
1. 头尾双指针 276
1.1 从两端判断回文字符串 276
1.2 两数之和：从两端排除候选 277
1.3 头尾双指针的移动模板 278
2. 快慢双指针 278
2.1 移除有序数组中的重复元素 279
2.2 快慢双指针的读写模板 280
3. 滑动窗口 280
3.1 区间和：扩张右端与移动左端 281
3.2 最长无重复子串 282
3.3 滑动窗口的扩张与收缩模板 283
【十】 二分与三分搜索 283
1. 二分搜索 284

1.1 整数二分 284
1.1.1 查找一个等于目标的元素 284
1.1.2 边界查找（lower_bound / upper_bound） 285
1.1.3 查找前驱与后继 287
1.1.4 两数之和：二分查找互补值 289
1.2 实数二分 290
1.2.1 求平方根 291
1.2.2 二分法求方程的根 292
1.2.3 与整数二分的对比 295
1.3 二分答案 299
1.3.1 最小吃香蕉速度（Koko，LeetCode 875） 300
1.3.2 木材切割：寻找最长可行长度 302
1.3.3 最小化最大值：寻找第一个可行上限 304
1.3.4 任务分段：最小化最大工作量 305
1.3.5 最大化最小值：寻找最后一个可行下限 307
1.3.6 进击的奶牛：最大化最小间距 307
1.3.7 分派：最大化每人的切片面积（POJ 3122） 310
1.3.8 二分答案的最大值与最小值模板 312
2. 三分搜索 315
2.1 从单峰与单谷寻找极值 315
2.2 从曲线形状理解取点 315
2.3 比较两个取点并舍弃一侧区间 315
2.4 实数与整数三分的实现 318
2.4.1 实数三分：缩小区间逼近峰顶 318
2.4.2 整数三分：比较取点后枚举剩余值 319
2.5 二分与三分的判定方式对照 323
2.6 从求根、判定与求极值选择搜索方式 324
（十一） 贪心算法 324
1. 逐步选择当前最有利的决策 324
2. 从题目结构寻找贪心策略 325
3. 贪心策略的例题与反例 325
3.1 最少硬币：贪心反例 325
3.2 最少加油次数：用堆选择已途经的站点 326
3.3 最少会议室：扫描开始与结束事件 328
4. 常见贪心问题与对应选择策略 330
5. 设计贪心策略步骤 331
5.1 明确目标（目标函数定义） 331
5.2 尝试局部最优策略（构造贪心规则） 331

5.3 证明正确性 332
5.3.1 交换论证法（Exchange Argument） 332
5.3.2 最优子结构证明法 333
5.4 验证样例与极端情况 333
（十二） 搜索：递归、回溯、DFS、BFS 与记忆化 334
1. 递归：把当前任务拆成更小的任务 334
1.1 汉诺塔问题 335
2. 回溯：选择、递归与撤销 337
2.1 n 皇后问题 337
3. 深度优先搜索（DFS） 342
3.1 完整例题：迷宫终点的可达性 344
4. 广度优先搜索（BFS） 347
4.1 完整例题：迷宫最少步数 348
5. 剪枝：提前结束无用分支 353
6. 记忆化搜索：把重复递归变成状态计算 355
（十三） 动态规划（DP）基础 359
1. 线性（一维）DP 362
1.1 斐波那契 362
1.2 跳楼梯 364
1.3 最大连续子数组和 366
1.3.1 保存每个结尾的最优和 368
1.3.2 只保存当前结尾和 369
1.3.3 同时恢复区间两端 369
1.4 最长递增子序列 371
1.4.1 枚举所有合法前驱 373
1.4.2 用最小末尾与二分优化 374
1.4.3 沿前驱恢复真实子序列 375
1.5 最长不下降子序列 376
1.5.1 推导允许相等的前驱 378
1.5.2 upper_bound 保留相等元素 379
1.5.3 不递增子序列的降序末尾 379
1.5.4 恢复不下降子序列 380
2. 二维 DP 382
2.1 最长公共子串 382
2.1.1 填写全部结尾状态 384
2.1.2 两行滚动与不匹配置零 385
2.1.3 根据结束位置取出内容 386
2.2 最长公共子序列 387

2.2.1 按末尾匹配结果推导二维表 390
2.2.2 分清当前行与上一行 391
2.2.3 沿最优来源恢复内容 392
2.3 编辑距离 393
2.3.1 三种操作对应三个前驱 395
2.3.2 两行滚动：先填写空前缀的代价 396
2.3.3 不同代价下比较总费用 398
2.3.4 填表时记录最优来源 399
2.3.5 从终点倒走并反转操作 401
2.3.6 用一个入口组合两个步骤 402
3. 背包问题 403
3.1 01 背包 403
3.1.1 按选或不选划分二维状态 406
3.1.2 用逆序容量保留上一行 406
3.1.3 通过相邻两行恢复物品 407
3.2 完全背包 408
3.3 最少硬币数量 411
4. 网格 DP：按依赖顺序填表 413
（十四） 并查集 416
（十五） 图论入门 419
1. 顶点、边、路径与连通性 419
1.1 先建立准确的图模型 419
1.2 度、自环、重边与简单图 420
1.3 走法、路径、环与距离 421
1.4 连通性、树与 DAG 421
2. 建图过程与存储方式选择 422
2.1 邻接表、邻接矩阵与边表 422
2.1.1 邻接表：每个顶点有一张邻居列表 422
2.1.2 邻接矩阵：每一对顶点预留一个格子 424
2.1.3 边表：每条输入边单独存一条记录 425
2.2 完整例题：读入并输出邻接表 426
2.3 完整例题：读入并输出邻接矩阵 427
2.4 带权图的终点、权值与重边存储 428
3. 用搜索标记无向图的连通块 429
4. 二分图：沿边给相邻顶点分组 431
5. Dijkstra：非负边权单源最短路 432
6. Floyd：所有点对最短路与距离矩阵 436
7. 拓扑排序 440

8. DAG 上的 DP：用拓扑序安排状态转移 442
9. 无向树的根、父节点与深度计算 444
10. 最小生成树：Kruskal 按权值选边 446
10.1 排序与并查集选边 447
10.2 读入与输出 448
11. 图算法的选择与题目对应 449
后记 449
从读懂题目到独立完成一道题 449
留给继续练习的你 450
按学习阶段选择推荐书籍 450
后续学习路线 451
牛客周赛与 Codeforces 分级训练 452
用牛客周赛练习定时解题 452
按 CF 题目 rating 逐步提高难度 453
把周赛、分级练习和复盘接成一轮 453
国内 ACM 与 OI 类竞赛 454
高校团队赛：ICPC 与 CCPC 454
各省大学生程序设计竞赛 455
青少年信息学：CSP-J/S、NOIP 与 NOI 456
ACM 式与 OI 式计分带来的训练重点 457
蓝桥杯与天梯赛等相关比赛 457
企业组织的算法与编程赛事 458
从日常练习接到参赛训练 459
阅读、做题与复盘的安排 460
附录 461
题解仓库入口 461
按章节与题号查阅 461
从提示回到独立解题 462

序言

这篇笔记从 C++ 语言基础出发，介绍常用标准库、面向对象和 ACM 入门算法。目标是让我们理解原理，逐渐学会独立读题、推导算法、写出程序，并用样例和手算核对结果。

使用方法

1. **先选路线，再逐段学习。** 根据下面的四条路线确定第一轮的必读内容；每完成一个阶段，再用对应练习检查理解。做题遇到新的工具或算法时，按目录回查。

2. **每节按“题面 → 分析 → 代码 → 解释”阅读。** 先尝试手算样例，再读实现；完整程序可以单独编译，函数片段要与本节头文件和调用片段组合使用。

3. **区分示例和练习。** 示例包含题面、分析、代码和样例解释，适合跟着手算、运行和修改；自拟题与外部题分别标明来源。练习题提供原题链接，读完一节后可以独立尝试。需要提示时，从附录进入题解仓库，先看建模与推导，再按需要读手算、图解和实现。

4. **做完讲解再做练习。** 每类算法附有洛谷或 Codeforces 的对应题目链接，只提供链接和必要的难度提示，不在练习区提前给答案。

5. **把模板与题目联系起来。** 手算它怎样处理当前数据，再解释每个变量、判断和循环的作用；遇到题目变化时，从这些具体步骤调整实现。

6. **记录自己的错误。** WA 后先找反例，TLE 后先算复杂度；修正代码后，把触发问题的输入保留下来。

正文默认使用 C++17。序言给出快速 I/O、竞赛模板、多组输入与 EOF；第一部分先学习原生数组、函数、引用与模板，再学习 STL 和 `std::function`，第二部分介绍面向对象，第三部分讲解题意建模、证明、方法选择与 WA 排查，第四部分介绍缓冲批量输入、大整数四则运算和提交检查，第五部分学习数据结构与算法；必读内容见下面按学习目标划分的路线，做题时也可随时查阅。全文采用从 `0` 开始的下标；个别题目或完全二叉树编号使用 `1` 时会单独说明。

通过目录可以先找到所属部分，再找到章节与具体用法。例如“一、C++ 语法基础 → （十五）STL → 1. string → 1.1 用 string 保存和访问文本”。

阅读过程图时，先认清数组、下标和状态的含义，再沿箭头手算一轮。蓝色表示基本结构，绿色表示当前状态或计算结果，橙色表示舍弃、冲突或本轮正在比较的位置。

按学习目标选择必读路线

第一遍学习不必把整本笔记从头读到尾。先选一条与自己当前目标相近的路线，完成其中的必读内容和练习，再根据做题时遇到的问题回查。下面的章号与目录一致；例如“五（十二）”表示“第五部分 → （十二）搜索”。

路线	适合的起点	第一轮的目标	阅读入口
零基础起步	还不能独立写出输入、判断和循环	把一段简单题意写成能运行的程序	先读语言主线，再接 ACM 入门主线
从 C 转到 C++	已经会用 C 写数组、循环和函数	用引用、模板与 STL 简化实现	先补 C++ 的写法，再按题目学习算法
ACM 入门与比赛练习	能独立写基本程序，开始接触算法	完成常见入门题，并参加周赛	先学基础技巧，再学搜索、DP 与图论
C++ 系统学习	希望理解语言、标准库与对象设计	能解释类型、对象和资源的使用过程	顺序学习第一、第二部分，再用算法检验理解

必读是当前路线的学习主线；**选读**可以接在相关练习之后；**回查**是在题目用到某个接口或模板时打开。表中的阶段按从上到下的顺序推进，每个阶段先读讲解、手算例子，再独立完成对应练习。

零基础起步：先写出程序，再认识常用工具

阶段	必读内容	本轮可以先放下的内容	完成标志
输入与计算	一（一）至（六）：程序结构、类型、转换、运算、判断循环、原生数组	位运算进阶、枚举类型的细节	独立完成 P1001、P5703、P5710、P1428，能手算每轮循环
把重复动作写成函数	一（七）函数；一（八）引用、值传参与生命周期；一（九）auto 与范围 for	动态分配、复杂指针、lambda 的递归写法	能解释实参与形参，能用引用修改一个变量
读入文本与组织数据	一（十）输入细节；一（十三）输出格式；一（十四）模板入门的类型参数 T；一（十五）STL 的 string、vector、pair、迭代器与 `<algorithm>`	文件操作、异常、多文件程序、类型工具及较少使用的容器	独立完成 P5733、P1427、P1059，解释排序与去重的步骤
从题意写出判断	三（一）至（六）、（八）（九）（十一）：提取信息、条件、循环、等价关系、优先级、向上取整、反例与状态	三（七）（十二）（十三）的综合模型	能给自己的判断找一个反例，并说明修正原因
开始算法练习	转入下面的 ACM 入门路线，从“基础技巧”开始	先不通读面向对象、二叉树长实现和全部二分变式	用已学知识独立完成一道枚举题，再按题目需要继续学

序言中的竞赛模板先作为提交时的查阅材料。学会函数与头文件以后，再回来解释 `solve`、测试组数和 EOF 循环；算法学习前理解标准流加速的两行设置即可。

从 C 转到 C++：补齐写法与工具的依赖

阶段	必读内容	选读与回查	完成标志
认识 C++ 对象的使用过程	一（二）（三）的初始化与类型转换；一（七）（八）的引用、const、传参与生命周期	用一（一）至（六）的练习检查熟悉的语法	能说明拷贝、引用与对象销毁分别发生在哪里
从类型参数到标准库	一（九）auto、lambda；一（十四）模板；一（十五）string、vector、pair、迭代器、algorithm、map、set、stack、queue、priority_queue	list、deque、unordered 容器在相应题目中补读	能用一个带注释的小程序演示插入、查找与遍历
传递动作与组织记录	一（十六）std::function；二（一）（三）（四）（八）的对象、构造、const 与比较规则	二（五）至（七）的继承、多态、拷贝移动在系统学习时回读	能给排序写出明确的比较规则，并解释一次回调
接上解题方法	三的必读小节与下面 ACM 主线；序言的快速 I/O、多组输入与 EOF	一（十一）的多文件组织、四（一）的 fread 在需要时回查	用 C++ 独立完成一道之前会用 C 做的题，再比较实现

ACM 入门：沿着题目需要逐步扩展算法

先补齐上一条路线中的常用语言工具。面向对象先读 `struct`、构造和比较规则；继承、多态、完整资源管理可穿插在后续语言学习中。

阶段	必读章节与小节	第一轮练习	后续选读
建模与工作量	三（一）至（六）、（八）（九）（十一）（十四）（十六）；五（一）复杂度；四（三）提交检查	P5711、P5709、P1085	三（十）（十二）（十三）（十五）（十七），用构造、方法比较、事件模拟与对拍扩展推导
基础技巧	五（二）枚举与模拟；五（三）前缀和与差分；五（四）排序；五（五）GCD、素数判断与筛法；五（六）快速幂	P1554、P8218、P2367、P1908、P1226	离散化、线性筛与较大规模数论题
有序性与局部选择	五（九）双指针；五（十）二分搜索的查找、单调判定与整数答案；五（十一）贪心	P1102、CF 279B、P1182、P1803、P1090	三分、实数二分、加油规划；无需先读完全部二分例题
搜索与重复计算	五（十二）递归 → 回溯 → DFS → BFS → 剪枝 → 记忆化	P1036、P1605、P1443、P1434	P1219 与新汉诺塔等状态较复杂的搜索题
从递归转成填表	紧接记忆化读五（十三）：线性 DP、二维状态、编辑距离、0/1 背包、完全背包、网格 DP	P1115、P1255、P2758、P1048、P2842、P1002	优化 LIS、排列转 LCS、P2679 子串状态扩展、多重背包
把关系建成图	五（十四）并查集；五（十五）概念与建图、连通块、二分图、Dijkstra、Floyd、拓扑序、DAG DP、树、Kruskal	P3367、B3643、P1330、P3371、B3647、P1807、P3366	五（八）二叉树的完整插入删除实现，用来理解结构维护
混合题与定时比赛	三（十二）（十三）（十五）（十七）；后记中的牛客周赛、CF rating 与复盘安排	把当前难度附近的题组成一次定时练习	按复盘结果补弱项，接入省赛、ICPC/CCPC 或 OI 训练

四（一）的 fread 用来处理较大的批量输入；四（二）的大整数四则运算在普通整数已经装不下答案时回查。遇到 P1255 这类精确计数题，再把大整数加法接到 DP 上，学习目标会更具体。

C++ 系统学习：把应用接回类型与对象

阶段	必读内容	配套任务
语言与标准库	第一部分按顺序阅读，包括指针、文件、异常、多文件组织、模板、STL 和 std::function	运行小程序，修改一次输入或调用方式，说明对象和结果的变化
对象与资源	第二部分按顺序阅读，从记录数据到封装、构造、继承、多态、拷贝移动、资源管理与比较规则	完成第二部分的四项练习，画出对象与所有者的关系
用题目组织代码	第三部分的建模、方法比较、完整事件模拟与职责划分；第四部分输入和大整数实现	把一个程序分成读入、建模、计算和输出，再解释每个部分保存的信息

阶段	必读内容	配套任务
用算法加深理解	第五部分先读复杂度、排序、搜索、记忆化与 DP，再按应用学习树和图	解释递归调用、容器存储和状态转移，回查相应语言机制

从一个阶段进入下一个阶段

读完讲解后，合上答案，用小数据走一遍，再独立写出本阶段的一道题。能解释变量、判断、更新顺序和复杂度，就继续下一阶段；若卡在某一步，回到对应小节重做这个步骤。每阶段留下一道已经独立完成的题和一道需要复习的题，下一次学习从这两道题开始。

正文练习旁保留原题链接。后记后的附录给出 GitHub 题解仓库入口，按章节与题号查阅完整推导。先看题解中的“建模与推导”，自己接着实现；仍有困难时再看手算、图解与代码。复习时可以先关闭答案，只看原题重新完成。

分阶段练习索引

阶段	先掌握	建议练习
语言入门	输入输出、循环、数组、函数、类型	数位统计 P1554
引用与模板	值传参、引用、auto、lambda、模板参数 T	用同一模板处理整数和小数，手算类型代入
语法基础中的 STL	`string`、`vector`、迭代器、排序、集合与映射	排序去重 P1059
传递计算动作	std::function 的签名、保存动作、回调与递归调用	手算规则替换与调用返回，再运行本章例题
建模与提交	提取变量与约束、反例与证明、一题多模型、WA 排查与小规模对拍、大整数运算、提交模板	数的性质 P5710、数列分段 P1182
基础技巧	枚举、复杂度、前缀和、差分、双指针	区间和 P8218、区间修改 P2367
数论与结构	GCD、筛法、快速幂、并查集	快速幂 P1226、并查集 P3367
算法策略与搜索	归并排序、二分、贪心、DFS、BFS、回溯、记忆化	迷宫 P1605、二分答案 P1182
动态规划与图论	从记忆化过渡到线性 DP、二维 DP、背包、最短路与生成树	背包 P1048、最短路 P3371

上表把各阶段的内容与练习接在一起，可配合自己选定的路线查阅。先用基本类型和原生数组写程序，再从引用、模板 T 走到 STL 和 std::function；算法部分从搜索与重复计算走到记忆化和 DP。

搜索章节最后学习记忆化搜索，随后接动态规划；两者的状态与前驱相同，计算顺序不同。

每学习一个模板，先手算一个小例子，再用自己的变量名实现一次。试着解释“这一轮处理什么、为什么这样更新、怎样得到答案”，比记住代码形状更有帮助。

快速 I/O 与竞赛模板

这部分作为提交时的查阅清单。初学者可先运行两整数求和的输入输出例子，再顺序学习正文；学过函数、类型和头文件后，回来理解通用模板中的每一行，不需要先掌握整份模板才能开始语言基础。

标准流加速

大多数入门题使用下面两行设置已经足够。把它们放在 `main` 开头、任何输入输出之前。

快在哪里？ C 与 C++ 标准流保持同步、读入前自动刷新关联输出，都可能产生额外工作。普通批量题只需把输入一次次读完、最后按规则输出，所以可以关闭这两项配合；它们不改变题目的计算逻辑。

模板在 main 开头先设置标准流，再用 cin、cout 完成输入输出。关闭同步后，C 与 C++ 的标准流各自使用缓冲；例如 scanf 已把一段输入读进自己的缓冲，cin 再读取时就可能接不到这段数据。因此，下面的标准流模板从读入到输出都使用 cin、cout，后面的 fread 模板则使用 C 输入输出。

`'\n'` 只写换行字符，endl 还会强制刷新。大量输出时，逐行刷新会增加开销；交互题中，程序输出询问后刷新，另一端收到询问才能继续回答，因此询问后可以使用 endl 或 cout.flush()。

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

ios::sync_with_stdio(false); // 关闭 C 与 C++ 标准流同步。
cin.tie(nullptr);            // 读入前不再自动刷新 cout。
```

- `sync_with_stdio(false)` 关闭 C++ 标准流与 C 标准流的同步。
- `cin.tie(nullptr)` 取消 `cin` 默认关联的输出刷新。
- `cout` 默认没有绑定输入流，通常不需要额外写 `cout.tie(nullptr)`。
- `endl` 会换行并刷新，普通批量输出使用 `'\n'` 即可。

> **注意：** 交互题需要在输出询问后主动刷新，例如 `cout << query << '\n' << flush;`。取消自动刷新以后，忘记刷新可能让双方一直等待。

基础竞赛模板：使用 ll 类型别名

下面适合 GNU C++17 等支持 `bits/stdc++.h` 的评测环境。每题先确认是否有多组输入，再决定是否读取 `T`。

```cpp
#include <bits/stdc++.h> // GNU 扩展，非标准头文件
using namespace std;
using ll = long long;

```

```cpp
void solve() {
    // 每组数据的变量尽量放在这里，自动重新初始化
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int T = 1;
    // cin >> T;  // 题目明确给出测试组数时开启
    while (T--) {
        solve();
    }
    return 0;
}
```

如果评测要求可移植的标准 C++，把第一行换为当前程序实际需要的标准头文件；只做整数输入输出时使用 `<iostream>` 即可。

竞赛宏模板：统一使用 long long

下面是一份常用的竞赛宏模板。可以与显式 `ll` 模板择一使用：

```cpp
#include <bits/stdc++.h> // 需要支持这个扩展头文件的 GNU 环境。
#define int long long    // 将后续 int 统一替换成 long long。
#define endl '\n'        // 仅换行，不刷新输出。
using namespace std;

void solve() {
    // 在这里实现一组数据的解法。
}

signed main() {
    ios::sync_with_stdio(false);
    cin.tie(0), cout.tie(0);
    int T = 1;
    // cin >> T;
    while (T--)
        solve();
}
```

说明：

- `#define int long long` 是预处理阶段的文本替换，不是类型别名。后续声明中的 `int` 会统一使用较宽整数，估算数组内存时按实际元素大小计算；学习 STL 后也按容器实际元素类型估算。
- `signed main()` 中的 `signed` 表示 `signed int`，没有显式的 `int` 记号，因此不会被宏替换；`main` 仍返回标准要求的 `int`。若写 `int main()`，返回类型也会被替换，因此模板使用 `signed main()`。
- 宏不会把字面量 `1` 自动变成 `1LL`。例如 `1 << 40` 的左操作数仍是普通 `int`，应根据需要写为 `1LL`

`<< 40` 或 `1ULL << 40`。

- `#define endl '\n'` 取消了刷新作用；后续写 `std::endl` 还会被替换成非法代码。需要刷新时使用 `flush`。
- 这个类型宏还会改变重载、接口匹配和类型组合。例如 `scanf("%d", &x)` 不再匹配替换后的 `long long x`。通用讲解示例使用 `ll` 或显式 `long long`，更容易看清类型。
- `cin.tie(0), cout.tie(0);` 使用逗号表达式执行两次调用，`0` 在这里表示空指针；现代写法可使用 `nullptr`。

这份模板适用于常见 GNU C++ 竞赛环境。统一整数类型可以减少累计和与乘法的类型遗漏；调试接口、移位和内存问题时，仍要按宏替换后的实际类型检查。

多组输入、EOF 与输出精度

有明确测试组数：

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

void solve() {
    // 示例每组读入两个整数，真正做题时替换为当前题目的解法。
    long long a, b;
    cin >> a >> b;
    cout << a + b << '\n'; // 输出这一组两数之和。
}

int main() {
    int T;
    cin >> T;
    while (T--) {
        solve(); // 每次调用处理恰好一组数据。
    }
}
```

没有组数，读到文件结束：

下面每组读入两个整数，用 long long 保存并输出它们的和。

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

int main() {
    long long a, b;
    while (cin >> a >> b) {
        cout << a + b << '\n'; // 每次成功读到一对数，处理恰好一组。
    }
}
```

`while(cin>>a>>b)` 把“本次完整读入成功”作为继续条件。EOF 表示输入结束，格式错误也会令读取失败。不要先判断 `!cin.eof()` 再读取；末尾标志通常在一次读取尝试后才设置，可能多处理一组旧数据。读到一组数据的中途就结束时，循环条件为假，这一组不会进入计算。

输出固定小数位：

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

double answer = 1.0 / 3.0;
// fixed 下的精度表示小数点后 6 位，输出 0.333333。
cout << fixed << setprecision(6) << answer << '\n';
```

> **注意：** 每组数据开始时重新初始化它需要的数组、计数器和答案，避免上一组结果影响下一组。后面学习容器后，也要按每组数据的实际长度重新准备存储。

C 与 C++ 的关系

C++ 在 C 的基础上发展而来，但不是 C 的严格超集。许多 C 程序只需少量修改就能作为 C++ 程序编译，也存在关键字、类型转换和语义不兼容的情况。

C 主要使用面向过程的方式组织程序；C++ 同时支持面向过程、面向对象和泛型等编程方式，并提供丰富的标准库。入门时可以先用函数和容器解决问题，再逐步理解类和模板。

一、C++ 语法基础

可以搭配 HDU OJ 第十一页开始的一百题来熟悉这些 C++ 基础语法，完成后可从 HDU OJ 第一页开始刷题。

（一） 程序结构与编译

先让程序输出一句话，再让它处理输入的两个数。运行这两个小程序时，分别观察“代码怎样被执行”和“答案怎样进入输出”。

1. 第一份程序的结构与语句作用

题目原文（自拟）： 程序无需输入，输出一行 `Hello, C++!`。

```cpp
#include <iostream> // 提供标准输入输出流的声明。

int main() {                        // main 是程序的入口；int 表示向运行环境返回整数状态。
    std::cout << "Hello, C++!\n"; // << 把右侧文本写入输出流，\n 换行。
    return 0;                       // 0 表示正常结束，不会自动显示在屏幕上。
}
```

输出：

```text
Hello, C++!
```

从 `main` 的左花括号进入，先执行输出，再执行 return，程序结束。`#include` 是预处理指令，不是运行时执行的一次输出；头文件使编译器认识 cout 等标准库名字。`std` 是标准库的命名空间，`std::cout` 明确指定名字属于哪里。

普通语句通常以分号结束，花括号把若干语句组成一个代码块。`//` 到行末是注释，`/* ... */` 是块注释。程序执行输出语句中的操作，注释则用来解释这些操作；main 的花括号标出函数体，输出语句后的分号标出这条语句的结束。

双引号中的文本是字符串字面量；这里直接输出它，不需要先创建 STL 的 string。`\n` 是一个换行字符，`\\` 表示反斜线，`\"` 表示双引号。源代码的换行便于阅读，输出是否换行则由输出内容决定。

2. 从固定输出到读入后计算

题目原文（自拟）： 输入两个整数 a、b，满足 $-10^6 \le a, b \le 10^6$，输出它们的和。

分析： 先准备两个保存整数的变量，用 cin 按顺序读入，最后把 `a+b` 的结果交给 cout。这里 `+` 表示整数加法，long long 用于保存整数；类型与范围在下一节展开。

```cpp
#include <iostream>

int main() {
    long long a = 0, b = 0;      // 先创建有名称的存储，并给出初值。
    std::cin >> a >> b;          // >> 按两个变量的类型读取两个数，空格或换行均可分隔。
    std::cout << a + b << '\n'; // 输出计算结果；不输出额外的输入提示。
    return 0;
}
```

输入：

```text
3 5
```

输出：

```text
8
```

读完后 a 为 3、b 为 5，表达式得到 8。cout 写出 8 与换行；`return 0` 向环境报告结束状态，与“输出整数 0”是两件事。竞赛评测给输入、比答案，除非题面要求，不添加“请输入”“结果为”等提示。

后面的学习程序常用 `using namespace std;`，于是可直接写 cin、cout。先掌握这两种写法，命名空间章节再解释名字冲突和多文件组织。

3. 源代码的编译与执行



预处理先处理 include、宏等指令；编译检查语言规则并生成目标代码；链接把需要的实现组合成可执行文件。不同工具可能合并展示这些步骤，错误仍可按发生阶段定位。

将程序保存为 main.cpp，以 GCC 为例执行：

```shell
g++ -std=c++17 -Wall -Wextra main.cpp -o main
```

`-std=c++17` 选择标准，`-Wall -Wextra` 开启常用警告，`-o main` 指定输出名称。Windows 上通常生成 main.exe，可在 PowerShell 中用 `./main.exe` 运行。修改源码后需要重新编译，旧可执行文件不会自动变成新代码。

观察到的情况	先检查什么	例子
编译不通过	语法、名字、类型、头文件	漏分号，cout 拼成 cuot
链接不通过	已声明的函数是否有实现、源文件是否都参与链接	声明了 add，却未提供定义
能运行但答案错误	题意、公式、边界与输出	输出了 a-b，题目要求 a+b
运行中异常结束	非法访问、除零、资源与递归深度	数组越界、除数为 0

读诊断时先处理靠前的错误，再重新编译；一个漏掉的符号可能引出很多后续诊断。警告也可能指出未初始化或类型不匹配的问题。先手算、运行样例，再用零、负数和题目中的最大规模核对运行结果。

练习题： 洛谷 P1001 - A+B Problem。

下一步把“两个整数变量”拆开理解：类型、名称、初值分别决定了什么。

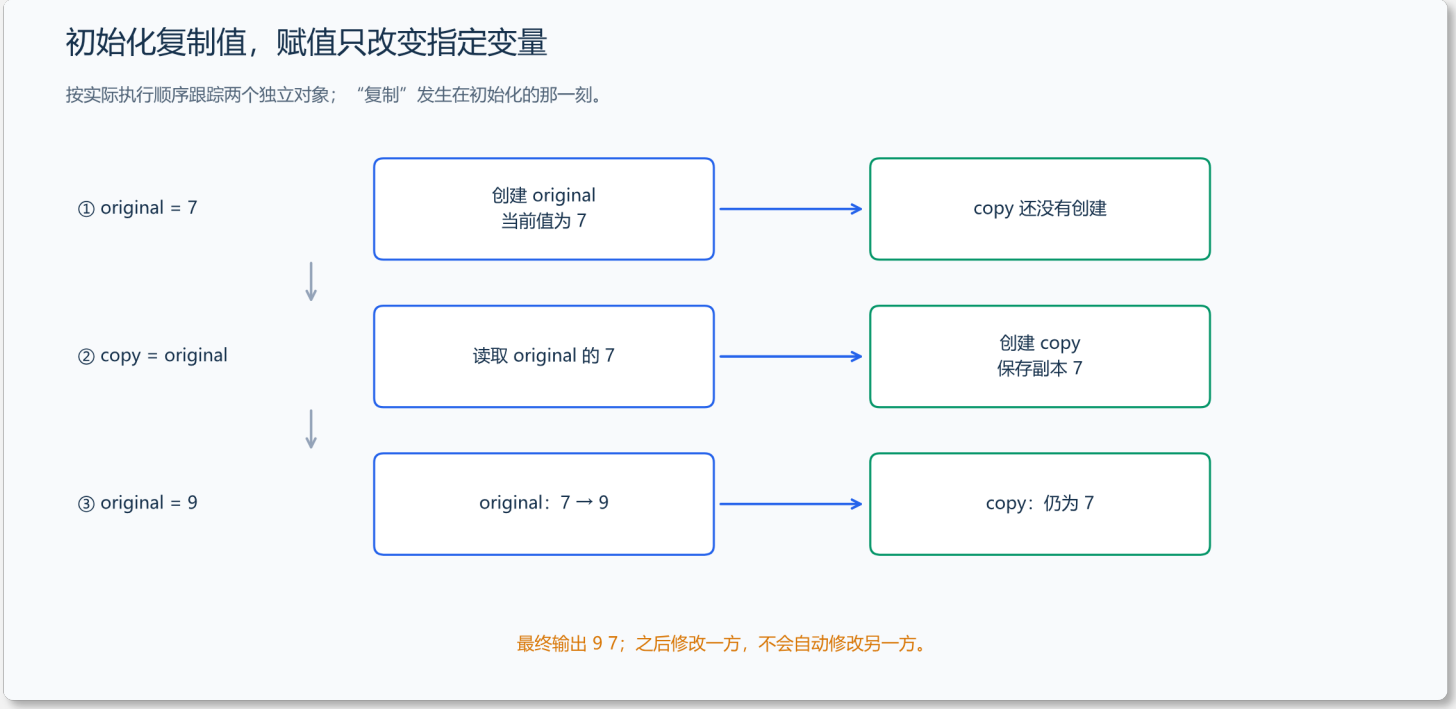
（二） 变量、数据类型与初始化

如果同一程序需要保存人数、总费用、平均分和成绩等级，就不能只问“用哪个变量名”，还要问“数据有没有小数、最大可能是多少”。类型决定存储和运算规则，变量名表示题目中的角色。

1. 声明、初始化与赋值是不同动作

`int count=3;` 创建名为 count 的 int 变量，并用 3 初始化。之后 `count=8;` 给已有变量赋新值，不创建第二个 count。初始化发生在创建时，赋值发生在对象已经存在后。

题目原文（自拟）： 程序无需输入。original 初值为 7，copy 初始化为 original 的值；把 original 改成 9，依次输出两个变量。




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

int main() {
    int original = 7;
    int copy = original; // 复制当前的整数值，不建立长期关联。
    original = 9;        // 只修改 original，copy 仍为 7。
    cout << original << ' ' << copy << '\n';
}
```

输出 `9 7`。把值复制到另一个变量后，修改来源不会自动同步；引用章节才会介绍让两名称访问同一个对象的方式。

写法	含义	结果
`int x=0;`	创建并初始化	x 的初值为 0
`int x(0);`	直接初始化	x 的初值为 0
`int x{};` 、`int x{0};`	花括号初始化	基本整数得到 0；有些窄化转换会被拒绝
`int x;`	创建普通局部整数，不提供初值	在成功赋值前不能读取
`x=5;`	修改已有变量	执行后 x 的值变为 5

普通局部 `int x;` 没有初值，成功输入或赋值后才有可用的数值。全局变量和 static 变量会先进行零初始化，static 的用法在函数章节继续讲。标识符区分大小写，score 与 Score 是两个名字；int、return 等关键字已经有语言规定的用途。

2. 从题目范围选择基本类型

类型	保存什么	常见比赛环境中的表示
`int`	整数、下标	常见比赛环境中为 32 位，约 -2.1×10^9 到 2.1×10^9
`long long`	更大的整数、累计和与乘积	至少 64 位；常见上界约 9.22×10^{18}，仍不能无限增大
`double`	带小数的近似值	小数通常不能精确表示；精度与范围是不同问题
`char`	单个字符或一个字节单元	`'A'` 、`'0'` 与数字 0 不同；一个中文字符通常占多个 UTF-8 字节
`bool`	逻辑值	只有 true、false；默认输出为 1、0
`unsigned int` 、`unsigned long long`	非负整数和位运算	没有负数；与有符号类型混合时要核对转换

先估算中间结果。 如果有 10^5 项，每项绝对值至多 10^9，累计和的绝对值至多 10^{14}，应选 long long；不能因为单项能放进 int，就把总和也放进 int。`1` 通常是 int 字面量，`1LL` 是 long long，`1u` 是 unsigned int，`1ULL` 是 unsigned long long。

`sizeof(int)` 返回 int 占用的字节数，`sizeof(value)` 查询对象大小。`sizeof(char)` 为 1，常见比赛环境中一个字节为 8 位。字节数描述存储大小，INT_MIN、INT_MAX 等常量则直接给出可保存的数值范围。

`<climits>` 提供整数范围常量，例如 INT_MIN、INT_MAX、LLONG_MAX，以及每字节的位数 CHAR_BIT。

题目原文（自拟演示）： 程序无需输入，分三行输出 int 与 long long 的字节数、int 的下界与上界、long long 的上界与每字节的位数。

分析： 用 sizeof 查询类型占用的字节数，用 INT_MIN、INT_MAX、LLONG_MAX 查询整数范围，用 CHAR_BIT 查询每字节的位数。

```cpp
#include <climits> // 声明范围常量；这里不需要模板知识。
#include <iostream>
using namespace std;

int main() {
    cout << sizeof(int) << ' ' << sizeof(long long) << '\n'; // 单位为字节。
    cout << INT_MIN << ' ' << INT_MAX << '\n';               // 当前实现的 int 范围。
    cout << LLONG_MAX << ' ' << CHAR_BIT << '\n';            // long long 上界和每字节位数。
}
```

在常见的 32 位 int、64 位 long long 环境中，输出如下：

```text
4 8
-2147483648 2147483647
9223372036854775807 8
```

这些数值说明了类型能保存多大的数。做题时，再将题目中可能得到的最大结果与类型范围比较，就能决定用 int 还是 long long。模板章节还会介绍另一种查询工具 numeric_limits。

3. 单字符、文本与逻辑值

单引号写一个字符，例如 `'A'`；双引号写文本，例如 `"ABC"`。字符 `'0'` 有自己的编码值，不是整数 0。十进制数字字符连续排列，因此已确认 ch 是数字字符后，可用 `ch-'0'` 得到对应数字。

题目原文（自拟演示）： 程序无需输入。输出字符 A、数字字符 7 对应的数值，以及 false、true 的默认输出。

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

int main() {
    char grade = 'A';
    char digit = '7';
    bool passed = true;
    cout << grade << ' ' << digit - '0' << '\n'; // 本例已保证 digit 是数字字符。
    cout << false << ' ' << passed << '\n';      // 默认得到 0 1。
}
```

输出为 `A 7`、`0 1`，各占一行。bool 的 false 表示假，true 表示真；条件判断中数字 0 转成假，非零转成真。需要输出 true/false 的文本时可用 boolalpha，输出格式章节会展示状态设置。

先区分整数、小数、字符与真假，再决定怎样计算。下一节重点解释“表达式先按什么类型算”，而不是只看接收结果的变量类型。

（三） 常量、类型转换与整数溢出

5 除以 2 为什么得到 2，明明用 double 保存了？两个 int 相乘为什么放进 long long 仍出错？两种问题都要沿“操作数 → 表达式结果 → 接收变量”的顺序检查。

1. 常量的声明与命名

每分钟 60 秒是固定换算关系，可以给它一个名称。`const` 声明只读对象，初始化后保持原值；`constexpr` 表示能在编译时确定的常量。比如 60 在编译时已经确定，而读入 n 后创建的 `const int count=n;` 要等运行时才知道数值。

```cpp
constexpr int secondsPerMinute = 60; // 编译时已知的常量，后续不能重新赋值。
const int limit = 100;               // 只读整数，初始化后保持 100。
// 在运行时先读得 n 后，也可写 const int count=n，但它不一定是编译期常量。
```

`using ll=long long;` 给已有类型起别名，ll 与 long long 是同一种类型；不会改写其他名字中的字符。序言的 `#define int long long` 则是预处理记号替换。两种写法用途不同，学习代码先用明确的类型观察计算。

2. 整数除法、浮点除法与转换位置

题目原文（自拟）： 输入整数 a、b，满足 $-10^6 \le a, b \le 10^6$ 且 b 非零。第一行输出整数商和余数；第二行输出按 double 计算的商，使用默认浮点格式。

推导： a、b 都是整数时，`a/b` 向 0 截断，例如 5/2 为 2，-5/2 为 -2。`static_cast<double>(a)` 是显式转换表达式，先得到 double 操作数，再相除；除法就按浮点规则计算。它不修改原来的 a。



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

int main() {
    int a, b;
    cin >> a >> b;                              // 题面保证 b 非零，且商、余数均可表示。
    cout << a / b << ' ' << a % b << '\n';      // % 求整数余数。
    double ratio = static_cast<double>(a) / b; // 先转换，避免整数除法丢掉小数。
    cout << ratio << '\n';
}
```

输入：

```text
5 2
```

输出：

```text
2 1
2.5
```

写成 `double ratio=a/b;` 时，整数商已经成为 2，赋值只会把这个 2 转成 2.0，不会恢复丢掉的小数。下一节继续用商与余数拆解整数，记住这一步的类型选择。

3. 乘法结果大于单项：在运算前扩宽

题目原文（自拟）： 输入非负整数 a、b，均不超过 10^9，输出它们的乘积，结果不超过 10^{18}。

分析： 两输入作为 long long 保存，乘法按 long long 计算；或者先将较窄操作数显式转成 long long。不能先执行可能溢出的 int 乘法，再转换结果。

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

int main() {
    long long a, b;
    cin >> a >> b;
    cout << a * b << '\n'; // 按题目上界，乘积可由 long long 表示。
}
```

输入 `100000 100000`，输出 `10000000000`。若 a、b 为 int，可用 `1LL*a*b` 或 `static_cast<long long>(a)*b`；乘法左结合，先有一个较宽操作数，后续才按较宽类型算。`static_cast<long long>(a*b)` 转得太晚。



有符号整数运算超出类型范围称为溢出，属于未定义行为：程序可能得到错误结果，也可能受编译优化影响。无符号整数按位宽取模，例如最大值加一会回到 0。求总和或乘积时，我们通常需要完整的数学结果，因此先选能容纳结果的类型。

4. 自动转换、窄化与有符号混合

情况	会发生什么	怎样检查
int 与 double 运算	常见算术转换使运算使用 double	转换前是否已发生整数运算
double 转整数	向 0 截断小数部分	例如 2.7 变为 2，-2.7 变为 -2
long long 转 int	结果受 int 的范围限制	只有 int 范围内的数值能保持原值
有符号与无符号混算	可能使负数转成很大的无符号值	比较与计算前先弄清两边的实际类型
花括号窄化	一些丢失精度的转换会被编译器拒绝	例如 `int x{2.7};` 不能这样初始化

`static_cast<int>(-2.7)` 得到 -2：去掉小数部分，向 0 靠近。floor 则向负无穷取整，会得到 -3，数学函数章节会用数轴对照这两种方向。转换后的整数仍要放得进目标类型；long long 中超出 int 范围的数转换成 int 时，C++17 的结果由实现规定，见 C++17 草案：整数转换。

练习题： 洛谷 P5703 - 苹果采购。先按题目给出的答案范围选类型，再写乘法。

类型选择之后，下一节把算术、比较、逻辑和位运算分别练清楚，避免把相似符号当成同一操作。

（四） 运算符与位运算

把“是偶数且位于范围内”写成代码，需要先得到两个真假结果，再合并。计算、赋值和判断有不同含义，下面每次只追踪一个表达式怎样得到结果。

1. 算术、赋值与自增

写法	含义	例子
`+ - * / %`	加减乘除、整数取余	`5/2` 为 2，`5%2` 为 1
`=`	给已有对象赋值	`x=3` 修改 x，不是判断 x 是否等于 3
`+= -= *= /= %=`	复合赋值	`sum+=x` 把 x 加入 sum
`++x`、`x++`	都令 x 增加一	前置表达式得到新值，后置表达式得到旧值
`--x`、`x--`	都令 x 减少一	同样区别表达式的结果值

题目原文（自拟演示）： x 初值为 3，先用 `x++` 初始化 before，再用 `++x` 初始化 after，输出 before、after 和 x。

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

int main() {
    int x = 3;
    int before = x++;                                   // before 得到旧值 3；语句结束后 x 为 4。
    int after = ++x;                                    // 先把 x 改成 5，after 得到 5。
    cout << before << ' ' << after << ' ' << x << '\n'; // 输出 3 5 5。
}
```

只写一条 `++x;` 或 `x++;` 时，增加的效果相同；区别在于是否还使用表达式返回的值。避免在同一复杂表达式中多次修改同一个变量；拆成独立语句更容易确定执行顺序。

整数除法向 0 截断，商与余数满足 $a=(a/b)b+a\%b$。例如 -5/2 为 -2，余数为 -1，因为 $(-2)\times 2+(-1)=-5$。在正模数 m 下，若需要非负余数，可先令 `r=x%m`，再在 r 为负时加 m；下一节的 if 会写出这个判断。

2. 比较与逻辑：从两个条件得到一个结论

写法	结果或含义	示例
`== ≠ < ≤ > ≥`	比较，得到 bool	`a==b` 判断相等
`&&`	两条件都真	`x≥0 && x<n`
`||`	至少一个条件真	`x<0 || x≥n`
`!`	真假取反	`!found` 表示未找到
`condition ? a : b`	条件真取 a，否则取 b	`a>b ? a:b` 取较大值

题目原文（自拟）： 输入 `0≤x≤1000`。判断 x 是否为偶数、是否满足 $4 < x \le 12$，再输出“两个条件都成立、至少一个成立、恰好一个成立、都不成立”的结果，以空格分隔。

推导： 先把两个性质各保存为 bool。都成立用逻辑与，至少一个用逻辑或，恰好一个表示两个 bool 不同，都不成立表示分别取反后再做与。不要写 `4<x≤12`：它会先算前一个比较得到 0 或 1，再把这个结果与 12 比较。

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

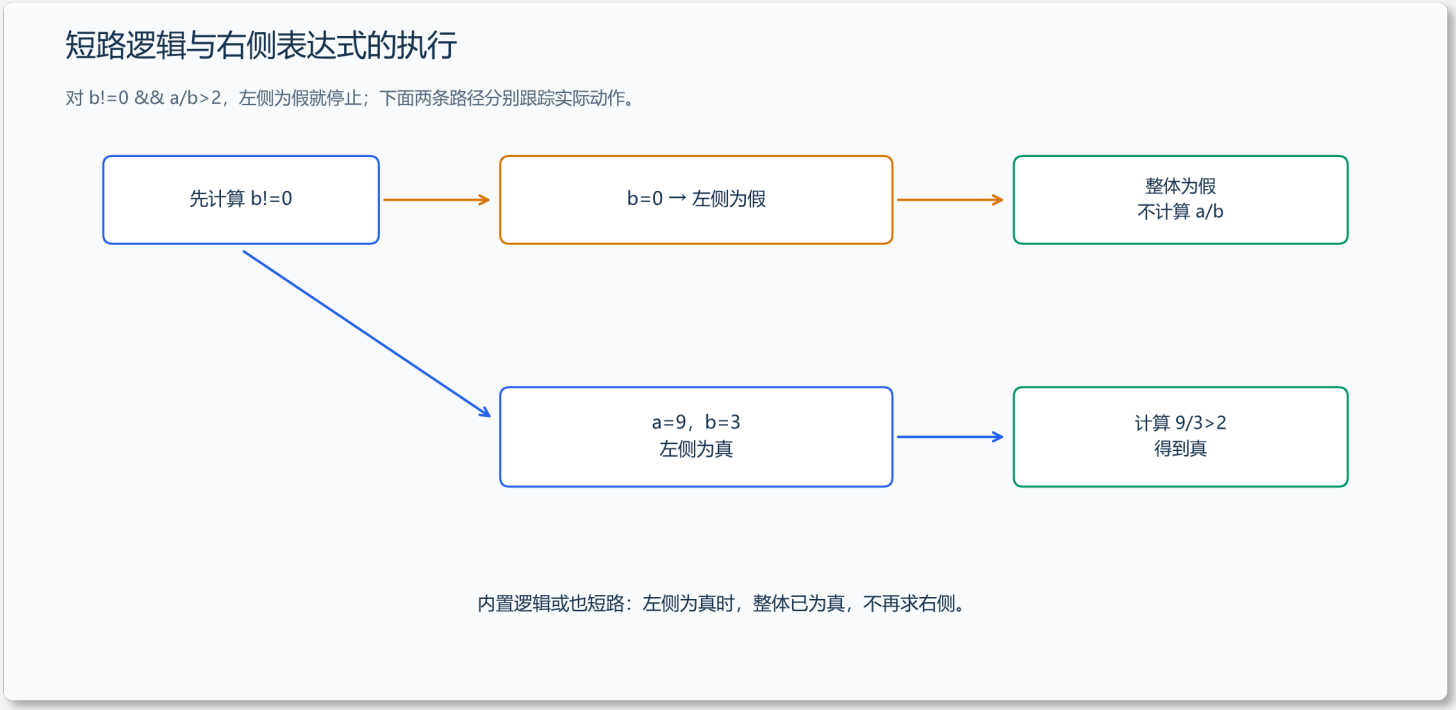
int main() {
    int x;
    cin >> x;
    bool even = x % 2 == 0;          // 先求余数，再比较它是否为 0。
    bool inRange = x > 4 && x ≤ 12; // 把数学连不等式拆成两个比较。
    cout << (even && inRange) << ' ' << (even || inRange) << ' ' << (even ≠ inRange) << '
'
         << (!even && !inRange) << '\n';
}
```

cpp

输入 `8`，输出 `1 1 0 0`。再分别代入 4、5、12、13：4 不满足严格的“大于 4”，5 满足；12 满足“不超过 12”，13 不满足。这样就能看出 > 与 ≥、< 与 ≤ 的区别。

3. 逻辑短路与表达式求值顺序

逻辑与 `&&` 的左侧为假时，整个结果已经确定为假，右侧就不再计算；逻辑或的左侧为真时，也不再计算右侧。比如 `b≠0 && a/b>2`，先检查 b：b 为 0 就直接得到假，只有 b 非零才计算除法。这个“得到结论后停止”的过程称为短路。



算术通常先乘除再加减；比较之后才合并逻辑结果。优先级决定如何分组，不等于所有子表达式都按书写顺序执行。`(a+b)*c` 用括号明确先求和，`(mask & bit)≠0` 用括号明确先按位与。逻辑判断中少写副作用，读到哪里就能知道检查了什么。

4. 用二进制位记录选中位置

按位运算处理每一个二进制位，和逻辑运算处理真假不同。`&` 是按位与，`|` 是按位或，`^` 是异或，`~` 是按位取反；`^` 不表示乘方。对本例的 unsigned int，右移一位相当于整除 2；左移一位把各位向高位移动，并在最低位补 0。移出类型位宽的高位会丢失。

题目原文（自拟演示）： mask 从 0 开始，将从 0 编号的第 1、3 位设为 1，检查第 3 位是否选中，再清除第 3 位，依次输出初次设置结果、判断结果和清除结果。




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

int main() {
    unsigned int mask = 0;
    mask |= 1u << 1; // 1u 左移一位得到 2，将第 1 位置为 1。
    mask |= 1u << 3; // 得到 2+8=10；按位或不影响已经选中的第 1 位。
    cout << mask << '\n';
    bool selected = (mask & (1u << 3)) != 0; // 只取出目标位，再转成真假。
    cout << selected << '\n';
    mask &= ~(1u << 3); // 目标位的掩码取反后，该位为 0，其余位为 1。
    cout << mask << '\n';
}
```

输出为 `10`、`1`、`2`，各占一行。图展示低四位，其余高位仍参与真实的无符号运算。`mask ^= bit` 会切换目标位，再切一次恢复；`mask |= bit` 重复置位不会切回 0。

位号从 0 开始。在常见的 32 位 unsigned int 中，可以标记第 0 到 31 位；需要标记第 40 位时，写 `1ULL<<40`，先用 unsigned long long 的 1 再移位。左侧字面量决定运算类型，接收结果的变量不会倒过来改变它。

练习题： 洛谷 P5710 — 数的性质；洛谷 P1100 — 高低位交换（位运算进阶，按题目的 32 位数理解拆分与组合）。

表达式能给出一个值，下一节再让这个值决定执行哪条路径、重复哪些步骤。

（五） 条件与循环

“成绩达到 60 就及格”“把 1 到 n 加起来”分别需要选择和重复。先列出可能的执行路径，再对照代码；循环最容易错的地方是边界、更新时机和什么时候退出。

1. if、else if 与 else：选择恰好一条路径

题目原文（自拟）： 输入 $0 \le score \le 100$。成绩至少 90 输出 A；否则至少 60 输出 B；其余输出 C。

推导： 先判断最高等级。如果没达到 90，再判断 60；这一支不必再写小于 90，因为前面已经排除了它。else 处理剩余情况。独立写两个 if 可能让 95 同时进入两个条件，不能表达“只输出一个等级”。



例如 75：第一问假、第二问真，输出 B；后续分支不再执行。

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

```cpp

int main() {
    int score;
    cin >> score;
    if (score ≥ 90) {
        cout << "A\n"; // 达到最高等级，不再执行后面的 else if。
    } else if (score ≥ 60) {
        cout << "B\n"; // 已知 score<90，所以此分支是 [60,90)。
    } else {
        cout << "C\n"; // 前两支均失败，score<60。
    }
}
```

输入 `75`，输出 `B`。用 59、60、89、90 检查两处分界。花括号内可以有多条语句；即使只有一条，也可保留花括号，降低后来添加语句时写错范围的机会。else 关联最近一个尚未匹配的 if，嵌套判断用缩进和花括号标清。

2. switch：按离散的值选择动作

如果不是比较范围，而是根据一个整数编号选择菜单，可以使用 switch。下面是完整的固定数据演示：day=2，输出 `Tue`。

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

int main() {
    int day = 2;
    switch (day) { // 本例按整数值选择分支。
    case 1:
        cout << "Mon\n";
        break; // 离开 switch，避免继续执行后面的 case。
    case 2:
        cout << "Tue\n";
        break;
    default:
        cout << "other\n"; // 没有 case 匹配时进入这里。
        break;
    }
}
```

case 后通常写互不重复的整数常量，switch 可用于整数或枚举值，不直接按 double 的区间选择。没有 break 时可能继续执行后面标签的语句，这叫贯穿；分支标签不会自动帮我们退出。两个 case 可以共用同一段语句，但要明确是有意分组。

3. for：初始化、判断、执行、更新

题目原文（自拟）： 输入 `0≤n≤1000000`，求 1 到 n 的和，先输出和的奇偶文字 even 或 odd，下一行输出

和。n=0 时和为 0。

推导： sum 先为 0，i 从 1 开始。每轮先判断 `i≤n`，成立才加当前 i；加完才执行 `++i`，然后回到判断。最后一次加 n 后 i 变成 n+1，条件失败结束。




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

int main() {
    int n;
    cin >> n;
    long long sum = 0; // 最坏和约为 5×10^11，用 long long 保存。
    for (int i = 1; i ≤ n; ++i) {
        sum += i; // 循环结束前，sum 等于 1 到当前 i 的和。
    }
    if (sum % 2 == 0) {
        cout << "even\n";
    } else {
        cout << "odd\n";
    }
    cout << sum << '\n';
}
```

输入 `3`，输出：

```text
even
6
```

检查时 i	条件	本轮动作	动作后的 sum
1	真	加 1，再令 i=2	1
2	真	加 2，再令 i=3	3
3	真	加 3，再令 i=4	6
4	假	不执行循环体	6

初始化只做一次，条件在每轮之前检查；n=0 时首次判断为假，循环体执行零次。求 1 到 n 的和，用 `i=1`、`i<=n`；访问 n 项数组，用 `i=0`、`i<n`，下标依次为 0…n-1。起点和条件随任务改变。循环中的 i 只在这个 for 的范围里可见。

4. while 与 do-while：重复次数由状态决定

while 先判断条件，可能零次执行；do-while 先执行一次再判断，末尾需要分号。比如统计非负整数的十进制位数，0 也应算一位，适合先处理一次。

题目原文（自拟）： 输入 $0 \le x \le 10^{18}$，输出 x 的十进制位数。

推导： 每次除以 10 去掉一位，同时计数加一；当剩余数为 0 时停止。使用 do-while，输入 0 也会经历一轮，得到 1。

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

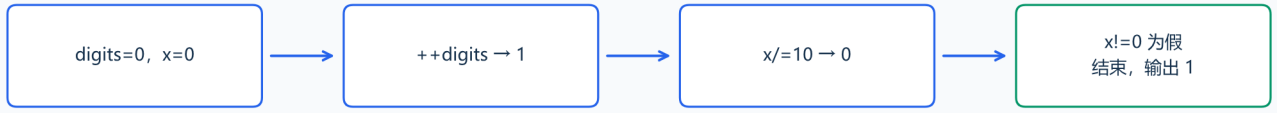
int main() {
    long long x;
    cin >> x;
    int digits = 0;
    do {
        ++digits; // 当前至少存在一位，包括输入 0 的情况。
        x /= 10;  // 整数除法去掉末尾一位。
    } while (x != 0); // do-while 在这里检查，末尾保留分号。
    cout << digits << '\n';
}
```

do-while：先执行，再判断循环条件

digits 从 0 开始；每轮先加一，再把 x 整除 10，最后判断 x!=0。

轮次	本轮开始 x	++digits 后	x/=10 后	最后判断
1	5080	1	508	真，继续
2	508	2	50	真，继续
3	50	3	5	真，继续
4	5	4	0	假，结束

输入 0 也先进入循环体：



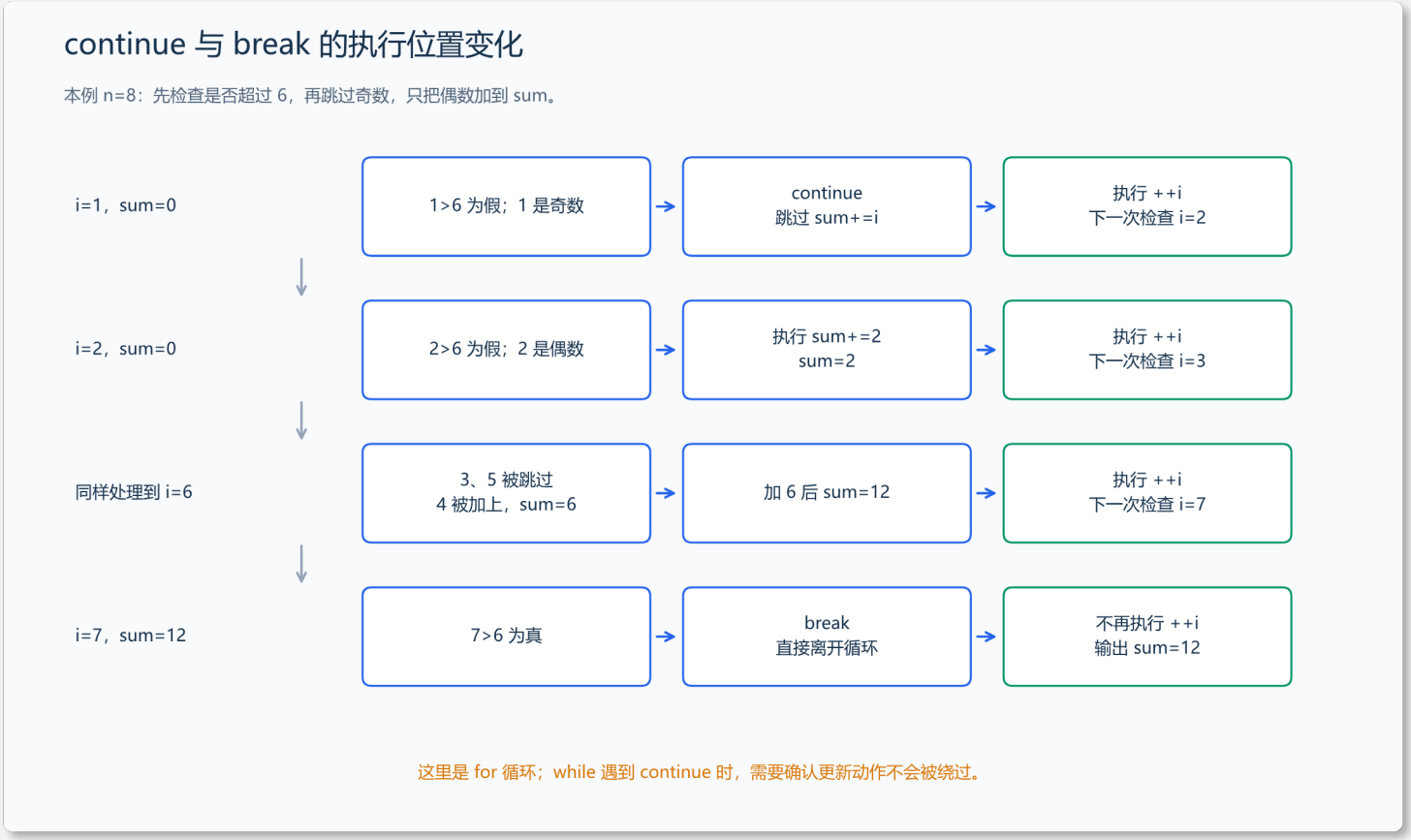
先判断的 while(x!=0) 在输入 0 时不会进入；do-while 保留了“0 也是一位”的要求。

输入 `5080`，输出 `4`。剩余数依次为 5080→508→50→5→0，恰好处理四轮。若写成先判断 `x≠0` 的 while，输入 0 会得到零次，需要另外处理。

题目有结束标记时，可把“本次成功读到数且它不是结束标记”作为 while 条件。例如 `while(cin>>x && x≠0)` 先读，成功后再判断数值；题面规定的 0 与 EOF 是不同的结束方式。先写清什么是一组输入，再选循环条件。

5. break 与 continue 的控制流

题目原文（自拟）： 输入 `0≤n≤20`，按从 1 到 n 的顺序检查整数，只累加偶数；一旦遇到大于 6 的数就结束，不再处理后面的数。输出累计和。




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

int main() {
    int n;
    cin >> n;
    int sum = 0;
    for (int i = 1; i <= n; ++i) {
        if (i > 6) {
            break; // 直接离开本层循环，不再执行 ++i。
        }
        if (i % 2 != 0) {
            continue; // 跳过本轮剩余语句，但仍执行 for 的 ++i。
        }
        sum += i; // 只有 i 为偶数且不大于 6 才能到达这里。
    }
    cout << sum << '\n';
}
```

输入 `8`，输出 `12`：1、3、5 跳过，2、4、6 累加，7 触发退出。break 只退出最内层循环或 switch，不能直接退出所有层；continue 在 while 中回到条件检查，不会自动执行我们写在循环体末尾的更新，更新被跳过可能导致死循环。for 的 continue 行为见 C++17 草案：for 语句。

6. 嵌套循环：内层完整执行后，外层才前进

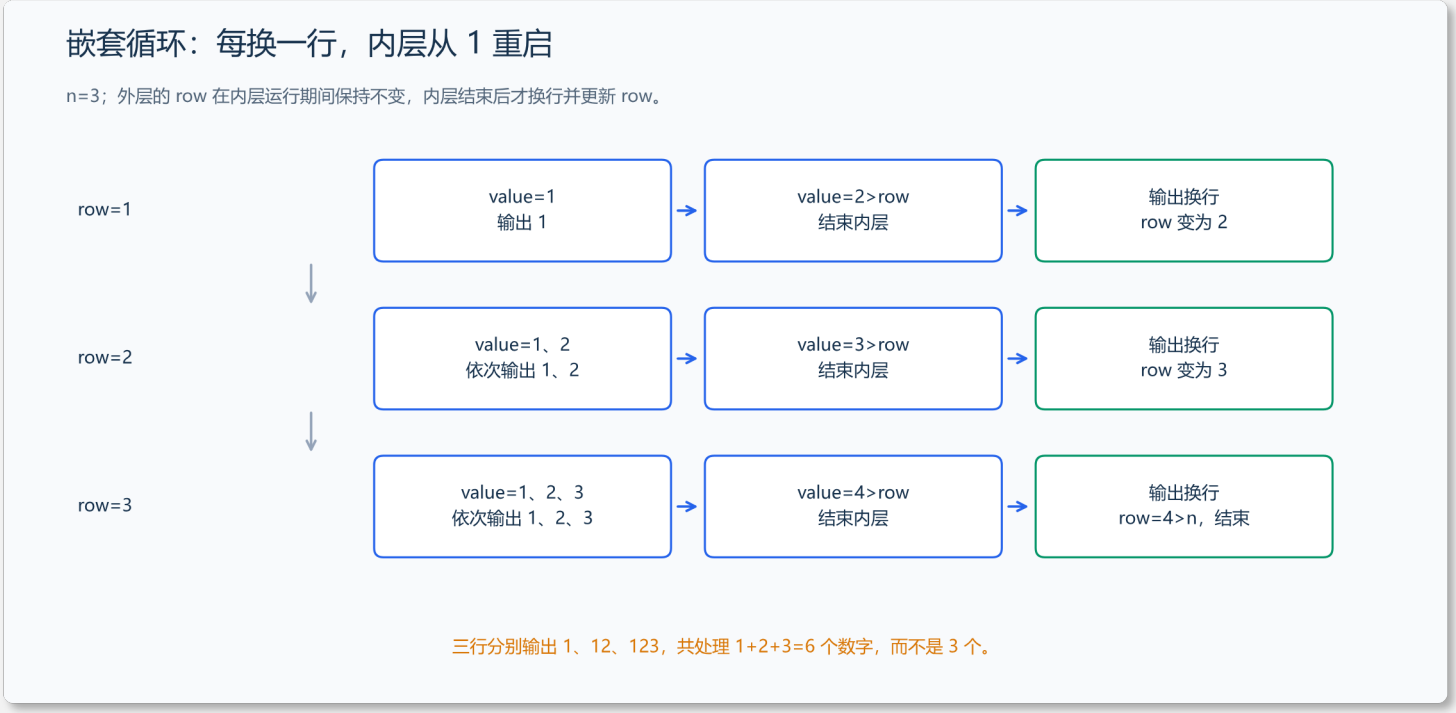
题目原文（自拟）： 输入 $1 \le n \le 9$，输出 n 行数字，第 i 行输出从 1 到 i 的数字，中间没有空格。

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

int main() {
    int n;
    cin >> n;
    for (int row = 1; row ≤ n; ++row) {
        for (int value = 1; value ≤ row; ++value) {
            cout << value; // 这一行的所有 value 输出完，内层才结束。
        }
        cout << '\n'; // 换行属于外层，每行恰好执行一次。
    }
}
```

输入 `3`，输出：

```text
1
12
123
```



实际访问顺序为 `(row,value)=(1,1)→(2,1)→(2,2)→(3,1)→(3,2)→(3,3)`。每次外层进入新行，内层都重新从 1 开始。总输出项数为 $1+2+\cdots+n$，不是只做 n 次；复杂度章节会继续用这种计数推导增长。

练习题：洛谷 P5712 – Apples（条件输出）；洛谷 P5721 – 数字直角三角形（嵌套循环，格式可结合输出格式章节）；洛谷 P5723 – 质数口袋（学完质数判断后练习循环和提前退出）。

能按顺序处理值以后，下一节让程序保存一整批值，并用循环访问它们的位置。

（六） 原生数组与枚举

如果需要先保存成绩，再统计或按另一种顺序输出，单个变量不够。原生数组创建一组同类型元素，下标指出要访问哪一项；先用数组和已学的循环解决任务，再理解长度、有效范围和二维布局。

1. 元素、下标与初始化

```cpp
int values[3] = {1, 2, 3}; // 三项，下标 0、1、2。
int zeroes[3]{};           // 三个元素都初始化为 0。
int partial[5] = {7, 8};   // 前两项为 7、8，剩余三项初始化为 0。
                           // int unknown[3]; 作为普通局部数组时，不会自动给整数元素清零。
```

`values` 是数组名称，`values[i]` 是下标 `i` 的一个元素。长度为 3 的数组有三项，不存在可以读取的 `values[3]`。`[]` 不检查越界，即使某次运行没有崩溃，越界也不能视为正确写法。

题目原文（自拟演示）： 三个整数为 1、2、3，把每项乘 2，再按顺序输出，以空格分隔。

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

int main() {
    int values[3] = {1, 2, 3};
    for (int i = 0; i < 3; ++i) {
        values[i] *= 2; // 第 i 项修改后，下一轮才访问第 i+1 项。
    }
    for (int i = 0; i < 3; ++i) {
        if (i > 0)
            cout << ' ';
        cout << values[i]; // 只读取已经创建的三项。
    }
    cout << '\n';
}
```

输出 `2 4 6`。不是修改一个“整个数组的值”，而是用下标逐项修改；引用章节会用范围 `for` 表达同样的逐项操作。

2. 物理长度与当前使用的 n 项

标准 C++17 的局部原生数组通常要求编译期常量长度，输入 `n` 后的 `int values[n];` 是部分编译器的扩展。可先按题目上限创建固定数组，再只读写前 `n` 项；数组的实际长度不会因此变成 `n`。

题目原文（自拟）： 输入 $0 \le n \le 1000$ 和 `n` 个绝对值不超过 10^6 的整数。n=0 输出 EMPTY；否则输出总和、最小值和最大值，以空格分隔。

分析： 数组准备 1000 项，但循环只到 n-1。n=0 时没有可用于初始化最值的实际输入；先输出 EMPTY 并结束。非空时从第一项建立最值，再逐项比较，不能用 0 当作所有数据的最小值或最大值。

数组长度与这次实际输入的数量

数组长度为 1000，n=4 时只有下标 0..3 是实际输入；空余空间不能当作额外数据。



有效输入：0 ≤ i < n；n=4，边界位置是 4

本步动作	sum	minimum	maximum
初始化首项	-5	-5	-5
加入 -2	-7	-5	-2
加入 -9	-16	-9	-2
加入 -1	-17	-9	-1

输入全为负数时，从第一项开始比较；maximum=0 则会保留一个没有出现过的值。

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

int main() {
    int n;
    cin >> n;
    int values[1000]{}; // 1000 是实际长度，n 是当前题目使用的数量。
    for (int i = 0; i < n; ++i)
        cin >> values[i];
    if (n == 0) {
        cout << "EMPTY\n";
        return 0; // 没有输入项，不拿补零的槽位冒充题目数据。
    }
    long long total = 0;
    int smallest = values[0], largest = values[0];
    for (int i = 0; i < n; ++i) {
        total += values[i];
        if (values[i] < smallest)
            smallest = values[i];
        if (values[i] > largest)
            largest = values[i];
    }
    cout << total << ' ' << smallest << ' ' << largest << '\n';
}
```

输入：

```text
4
-5 -2 -9 -1
```

输出：

```text
-17 -9 -1
```

设初值 largest=0 会在全负数时错误地保留 0。这里第一次循环读取，第二次循环处理，共两遍 n 项；数组剩余槽位虽初始化为 0，也不属于本次输入。

3. 下标也能表示类别：计数数组

题目原文（自拟）： 输入 $0 \le n \le 1000$，接着输入 n 个 0…9 的整数，输出各数的出现次数，从 0 到 9，空格分隔。

推导： `frequency[d]` 表示数字 d 的出现次数。读到 d 后，增加这一项；下标是类别编号，元素值是次数，不再是原输入值。题目把 d 限定在 0…9，所以十项数组足够。

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

int main() {
    int n;
    cin >> n;
    int frequency[10]{}; // 所有类别先出现零次。
    for (int i = 0; i < n; ++i) {
        int digit;
        cin >> digit;
        ++frequency[digit]; // 输入值决定要更新哪个下标。
    }
    for (int digit = 0; digit < 10; ++digit) {
        if (digit > 0)
            cout << ' ';
        cout << frequency[digit];
    }
    cout << '\n';
}
```

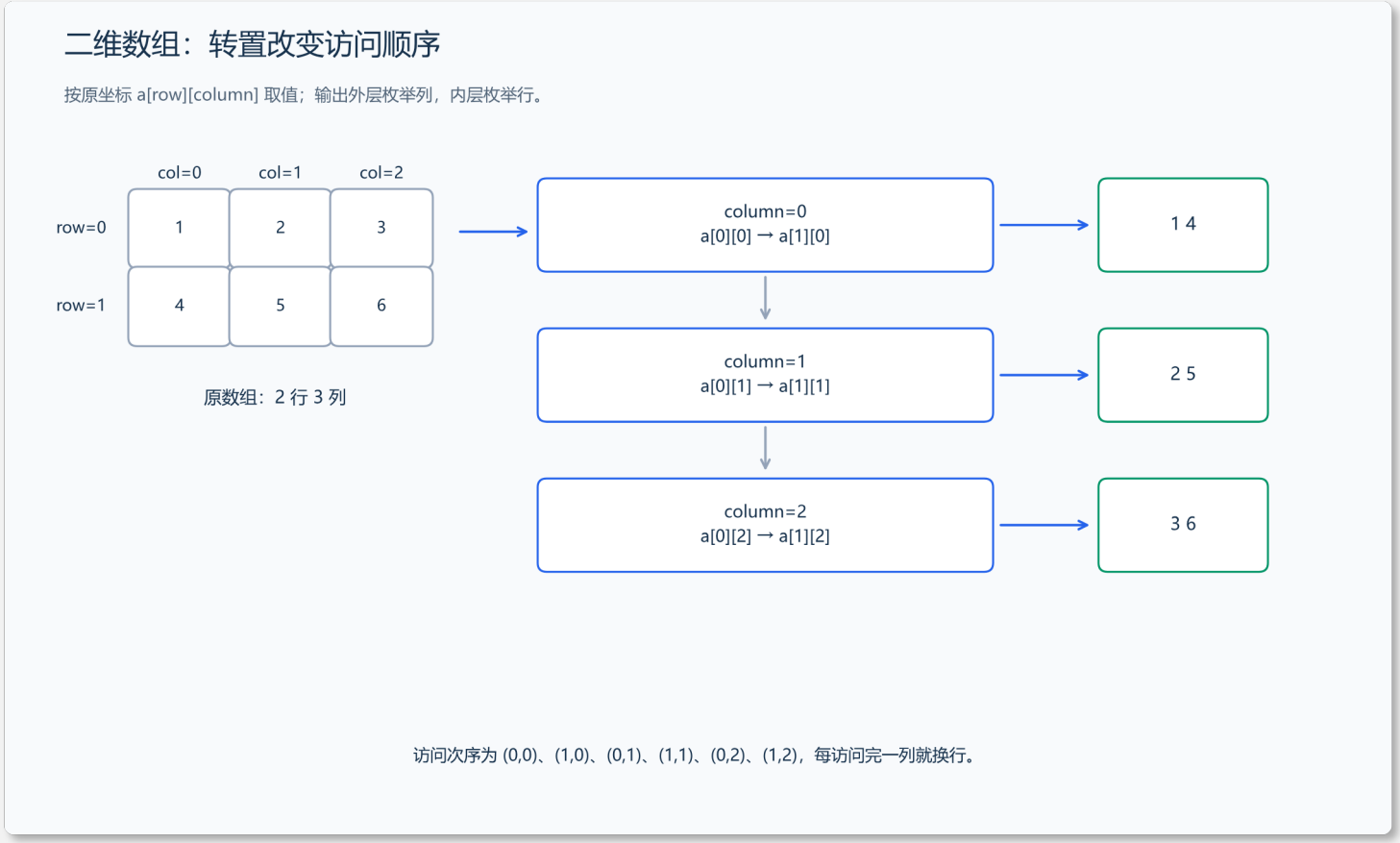


输入 `5` 与 `2 0 2 9 2`，输出 `1 0 3 0 0 0 0 0 0 1`。多次读到同一个数字，会累加同一项；n=0 时十项都为 0。这里数字 0…9 恰好对应十个类别。后面还会用映射和离散化给负数、较大的数建立类别编号。

4. 二维数组的行列下标与访问

题目原文（自拟）： 输入一个 2 行 3 列的整数矩阵，按行输入；输出它的转置，即 3 行 2 列，每行元素用空格分隔。

推导： `a[row][column]` 先用行号选择一行，再用列号选元素。转置把原第 column 列变成输出的一行，因此输出外层循环枚举 column，内层枚举 row；原数组仍用原坐标读取。




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

int main() {
    int a[2][3]{}; // 行号 0、1；列号 0、1、2。
    for (int row = 0; row < 2; ++row) {
        for (int column = 0; column < 3; ++column)
            cin >> a[row][column]; // 按行读入，每一行的列从 0 开始。
    }
    for (int column = 0; column < 3; ++column) {
        for (int row = 0; row < 2; ++row) {
            if (row > 0)
                cout << ' ';
            cout << a[row][column]; // 输出坐标 (column,row) 读取原坐标 (row,column)。
        }
        cout << '\n';
    }
}
```

输入：

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

输出：

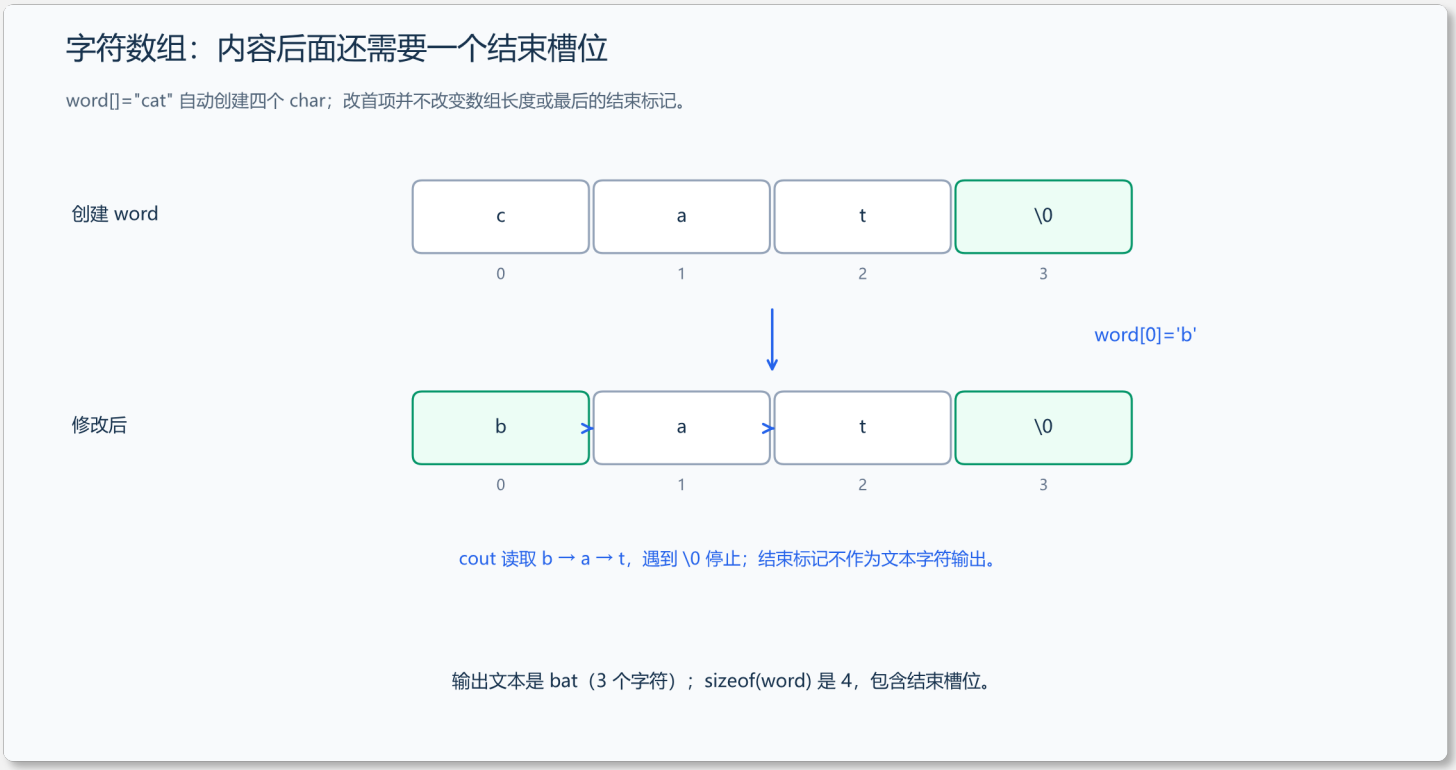
```text
1 4
2 5
3 6
```

原生 `int a[2][3]` 是“含两个三项数组的数组”，按行连续存储。第一下标选第 0 或第 1 行，第二下标选这一行的第 0、1、2 项；例如第二行第二项写成 `a[1][1]`。

5. 字符数组与字符串结束标记

题目原文（自拟演示）： 创建保存 cat 的字符数组，把首字符改成 b；先输出这段文本，再输出整个数组的字节数，各占一行。

推导： 文本的三个字符后还需要结束标记，数组因此有四项。修改首字符不改变最后的结束标记，输出仍能确定在哪里停止。




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

int main() {
    char word[] = "cat";          // 自动得到长度 4：c、a、t、结尾标记 '\0'。
    word[0] = 'b';                // 单项是 char，使用单引号赋值。
    cout << word << '\n';         // 按 C 风格字符串输出到 '\0' 为止，得到 bat。
    cout << sizeof(word) << '\n'; // 得到数组字节数 4，不是文本字符数 3。
}
```

输出为 `bat`、`4`，各占一行。文本包含三字符，但 sizeof 计入最后一个 char 槽位。`char word[3]={'c','a','t'};` 是三项字符数组，却没有结束标记，不能把它当成以 `\0` 结尾的字符串让 cout 持续读取。字符数组直接输出与 int 数组直接输出的行为不同；int 数组要按元素遍历，不能指望 cout 自动展示全部元素。整行读入时怎样准备容量，输入细节章节会继续解释。

6. 枚举：给有限状态命名

如果只需要表示等待、运行、结束三种状态，可以用枚举，避免数字 0、1、2 的含义分散在各处。下面片段放在 main 的函数体中，观察状态的声明与赋值：

```cpp
enum class State { Waiting, Running, Finished }; // 定义类型与三个有名称的枚举值。
State state = State::Waiting;                    // :: 指明枚举值属于 State。
state = State::Running;                          // 把已有状态从等待改为运行。
```

`enum class` 的枚举值具有自己的作用域，不会自动转成普通 int。可以用 `state==State::Running` 判断，也可以交给 switch；需要其整数表示时显式转换。枚举名称让状态更容易读懂，例如从 Waiting 改为 Running，就表达了从等待到运行的变化。

练习题： 洛谷 P1428 – 小鱼比可爱（数组与嵌套比较）；洛谷 P5732 – 杨辉三角（二维数组，先手算每行如何依赖上一行）。

数组与循环已经能组织一段处理过程；下一节把这段过程取名，重复调用时明确数据怎样传入、结果怎样返回。

（七） 函数、参数与作用域

在多个位置重复计算两数之和，每次复制代码会让修改容易遗漏。函数把一段操作取名，调用者给输入、函数完成处理、结果交回调用者；先分清这三个角色，再学习参数和生命周期。

1. 函数的返回类型、参数与执行过程

`int add(int a,int b)` 中，第一个 int 是返回值类型，add 是名称，括号里的两项声明参数，花括号里的语句是函数体。调用 `add(2,3)` 时，2、3 是实参，a、b 是接收它们的形参。

题目原文（自拟）： 程序无需输入。x 初值为 10。调用将值参数加一并返回结果的函数，第一次忽略结果，输出 x；第二次接收结果，输出结果与 x；最后输出 `add(x)` 和 `add(2,3)`，add 的第二参数默认是 0。

分析： 值参数得到 x 当前值的副本；改副本不会写回 x。返回 11 也不等于自动给 x 赋 11，调用者需要明确接收或赋值。默认参数只补充调用中省略的实参。




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

int add(int a, int b = 0) {
    return a + b; // 返回一个独立的整数结果。
}

int increaseCopy(int value) {
    ++value;      // 只修改函数自己的参数。
    return value; // 函数调用表达式的结果是这个整数。
}

int main() {
    int x = 10;
    increaseCopy(x); // 计算出 11，但没有接收，也没有修改原来的 x。
    cout << x << '\n';
    int result = increaseCopy(x); // 将返回值接收到另一个变量。
    cout << result << ' ' << x << '\n';
    cout << add(x) << ' ' << add(2, 3) << '\n'; // 第一处补上默认参数 0。
}
```

输出：

```text
10
11 10
10 5
```

如果写 `x=increaseCopy(x);`，才会用返回值更新 x。`void` 返回类型表示没有结果值，例如只执行输出的函数；`return;` 可以提前结束 void 函数。普通非 void 函数应在所有可达的正常返回路径提供结果；main 末尾隐式返回 0 是特殊规则。

2. 声明让名字可见，定义提供具体操作

固定数据演示： 下面 main 调用 add(2,3)，但把函数体写在 main 后面；先声明接口，输出 5。

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

int add(int a, int b); // 声明：编译 main 时已经知道参数与返回类型。

int main() {
    cout << add(2, 3) << '\n'; // 调用前有声明，函数体可在后面定义。
}

int add(int a, int b) {
    return a + b; // 定义给出实现：将两个参数相加并返回结果。
}
```

只写声明而没有定义，不会凭空生成实现；需要这个函数时，链接可能报错。默认参数通常从参数列表末尾给出，写 `add(3)` 时省略的是最后一项。声明处已经提供的默认参数，不应又在同一作用域的定义处重复指定。

3. 重载：同名操作接收不同参数类型

题目原文（自拟演示）： 用两个同名 larger 函数，分别比较整数 3、5，以及小数 2.5、1.0，输出两结果。

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

int larger(int a, int b) {
    return a > b ? a : b; // 整数版本。
}
double larger(double a, double b) {
    return a > b ? a : b; // 小数版本，逻辑相同，参数类型不同。
}

int main() {
    cout << larger(3, 5) << '\n';     // 选择整数参数版本，输出 5。
    cout << larger(2.5, 1.0) << '\n'; // 选择 double 参数版本，输出 2.5。
}
```

重载需要参数列表等方面满足相应区分规则，只改返回类型不能让同一调用知道选谁。`larger(3,2.5)` 可能让这两个版本都需要转换，无法选出更优者；明确按 double 处理时可先转换 3。模板章节会从“两段相同逻辑”的重复继续推导类型参数 T。

4. 局部、外层与全局：同名不代表同一对象

题目原文（自拟演示）： 全局 value=100，main 内 value=7，内层块又有 value=2。先输出内层名字，再离开该块输出 main 的名字与明确指定的全局名字。

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

int value = 100; // 全局变量，普通程序运行期间持续存在。

int main() {
    int value = 7; // main 的局部变量，在这个范围内遮蔽全局同名名字。
    {
        int value = 2; // 新对象，内层同名名字暂时遮蔽外层。
        cout << value << '\n';
    } // 内层局部对象结束生命周期，重新能按 value 找到 main 的变量。
    cout << value << ' ' << ::value << '\n'; // ::value 明确从全局作用域找名字。
}
```

输出为 `2` 、`7 100` 。遮蔽改变当前名字查找的结果，没有给外层变量重新赋值。普通局部变量通常离开块后销毁，函数返回后其普通局部对象也不再存在；后面指针、引用章节会解释为什么不能继续访问它们。

5. static 局部变量：跨调用保留数据

固定数据演示： 普通局部 count 每次调用重新从 0 开始，static 局部 count 的初始化只进行一次；各调用三次，观察结果。

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

int ordinaryNumber() {
    int count = 0;
    return ++count; // 每次新建 count，始终返回 1。
}
int nextNumber() {
    static int count = 0; // 第一次初始化后，后续调用继续使用原来的对象。
    return ++count;
}

int main() {
    for (int i = 0; i < 3; ++i) {
        if (i > 0)
            cout << ' ';
        cout << ordinaryNumber();
    }
    cout << '\n';
    for (int i = 0; i < 3; ++i) {
        if (i > 0)
            cout << ' ';
```

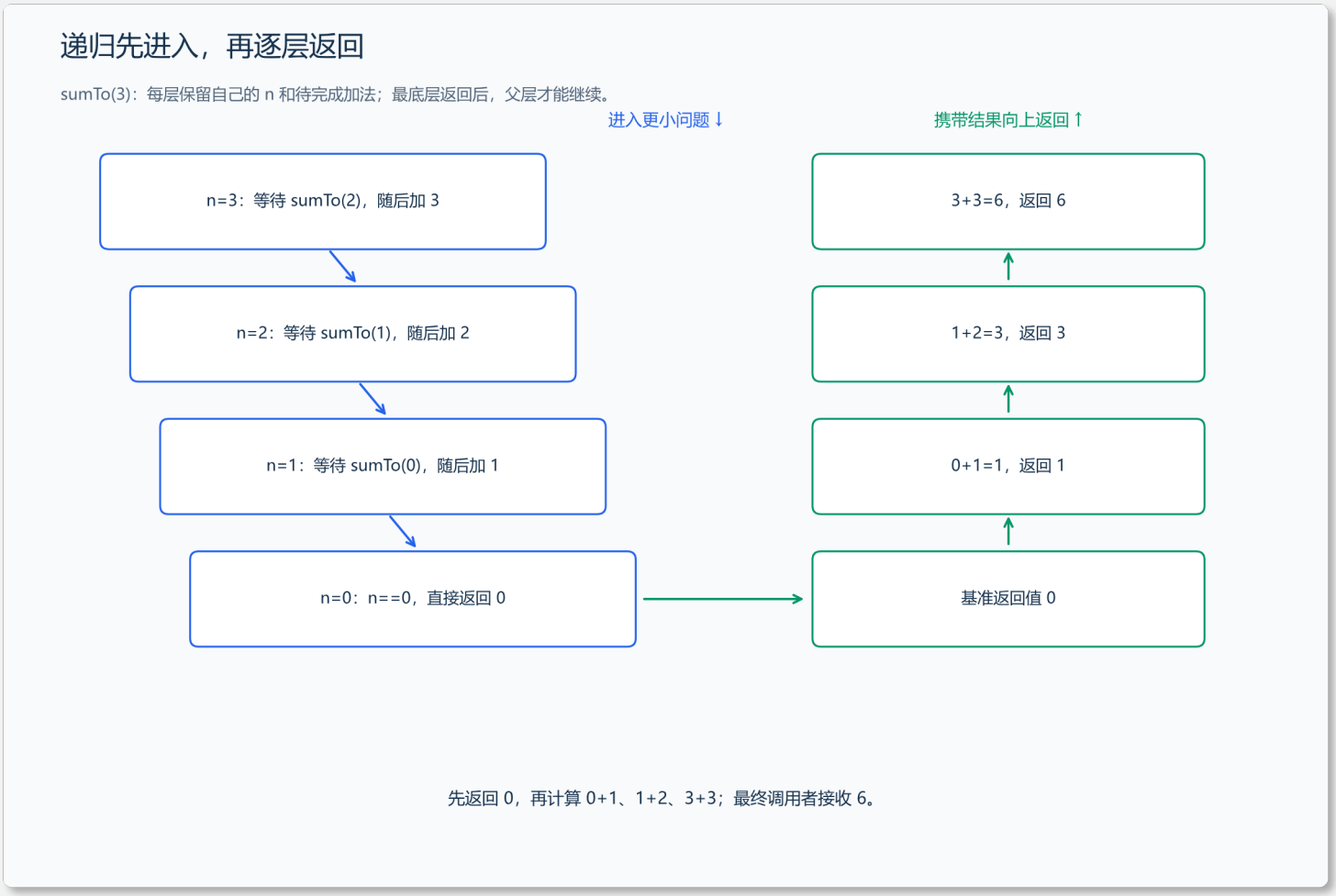
```cpp
        cout << nextNumber(); // 依次返回 1、2、3。
    }
    cout << '\n';
}
```

输出为 `1 1 1`、`1 2 3`。static 局部变量的名字仍局限于函数内部，生命周期却跨越调用；可见范围与存在时间不是同一概念。多组竞赛数据中，这种保留也可能让旧状态影响新一组，需要按题意重置或选择普通局部变量。

6. 递归的最小例子：暂停调用者，返回后继续

题目原文（自拟）： 输入 `0≤n≤1000`，用递归计算 1 到 n 的和；n=0 的和为 0。

推导： 求 n 项的和，可以先求前 n−1 项，再加 n；当 n=0 时不再分解，直接返回 0。每次调用都有自己的参数 n，因此“把 n 改小后再调用”没有把父层的 n 也改小。




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

long long sumTo(int n) {
    if (n == 0)
        return 0;                          // 终止条件；没有它就会继续调用负数。
    long long previous = sumTo(n - 1); // 当前层暂停，等待子调用的返回值。
    return previous + n;                   // 返回后恢复当前层，使用这一层自己的 n。
}
```

```cpp

int main() {
    int n;
    cin >> n;
    cout << sumTo(n) << '\n'; // n=3 时输出 6，n=0 时输出 0。
}
```

输入 `3`，输出 `6`。进入顺序是 3→2→1→0；返回时 0 交回 0，1 层算 0+1，2 层算 1+2，3 层算 3+3。加法发生在子调用返回之后，不是在第一次进入就知道完整答案。

这个任务的循环做法已经足够，递归用于观察调用机制。它会同时保留多层尚未结束的调用，n 很大时可能耗尽调用栈；搜索章节会继续讲递归选择、回溯和记忆化，不能仅凭公式递减就忽略空间。

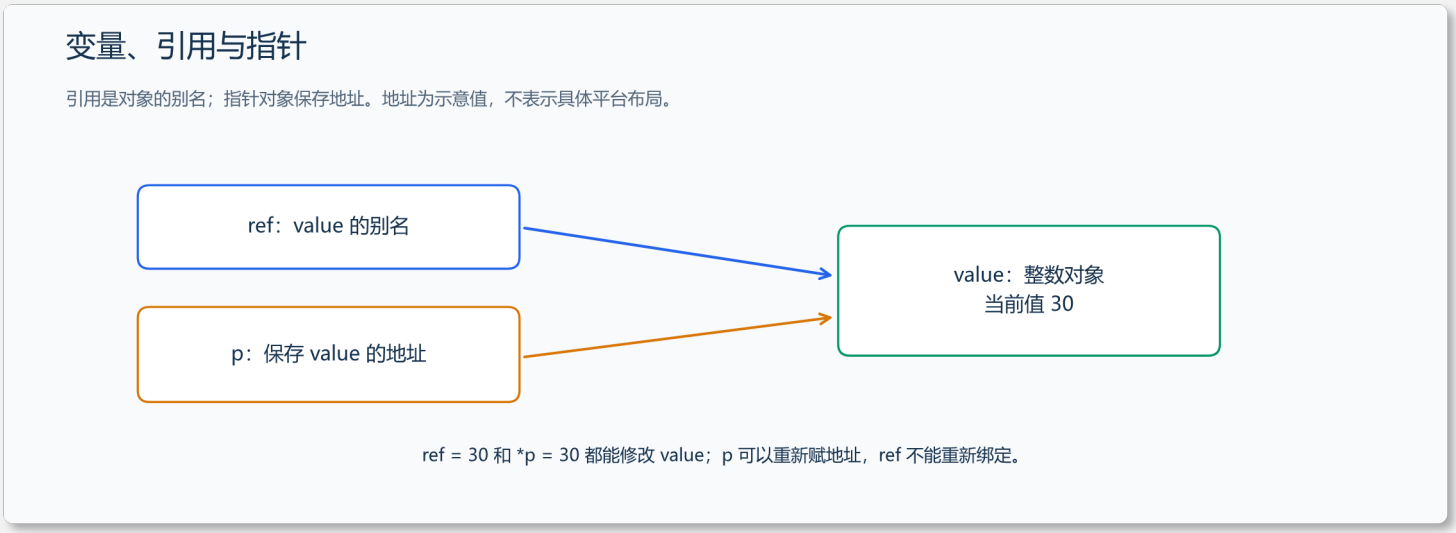
函数目前得到值的副本。若任务要求直接交换调用者的两个数，下一节就需要明确传入的是副本、别名还是地址。

（八） 指针、引用与对象生命周期

上一节给参数副本加一，没有改变调用者的 x。现在希望函数能操作原变量：**引用提供原对象的别名，指针保存可用于找到对象的地址。**

下面的局部操作片段放在 main 的函数体中执行：

```cpp
int value = 10;
int *p = &value;  // &value 取得 value 的地址，p 保存这个地址。
int &ref = value; // 定义中的 & 表示引用，ref 是 value 的别名。
*p = 20;          // *p 取得指针指向的对象，把 value 改为 20。
ref = 30;         // 通过别名修改同一个 value。
p = nullptr;      // nullptr 表示空指针，不能再通过 *p 访问对象。
```



同一个 `&` 在声明和表达式中的含义不同；图中的 ref 表示别名关系，不表示引用一定单独占用一块存储。指针可以改指向，引用一旦绑定就不能改绑；`ref=other` 是赋值给原对象，不是改绑到 other。

对比项	指针	引用
初始化	可以为空，推荐初始化	普通引用需要绑定一个对象

对比项	指针	引用
改变目标	可以重新赋地址	不能重新绑定
访问	`*p`，访问前保证指向有效对象	直接用引用名
生命周期	所指对象销毁后会悬空	所绑定对象销毁后也会悬空

1. 把值传参改成引用或指针传参

题目原文（自拟）： 程序无需输入。x 从 10 开始，依次通过值参数、引用参数和指针参数加一，每次输出 x；最后求数组 `1、2、3` 的总和。

分析： 值参数仍修改副本；引用参数直接绑定 x；指针参数得到 `&x` 的地址副本，再通过 `*p` 修改 x。参数 `const int*` 只允许通过它读数组，不能通过这个参数改元素。

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

void increaseCopy(int value) {
    ++value;
} // 只修改副本。
void increaseRef(int &value) {
    ++value;
} // 修改绑定的原变量。
void increasePointer(int *p) {
    if (p != nullptr)
        ++*p; // 先判空，再访问所指对象。
}

long long sum(const int *values, int n) {
    long long result = 0;
    for (int i = 0; i < n; ++i)
        result += values[i]; // 只读前 n 项。
    return result;           // 返回独立的整数，不返回局部变量的引用。
}

int main() {
    int x = 10;
    increaseCopy(x);
    cout << x << '\n';
    increaseRef(x);
    cout << x << '\n';
    increasePointer(&x);
    cout << x << '\n';
    int values[3] = {1, 2, 3};
    cout << sum(values, 3) << '\n'; // 数组在这个调用中转换为首元素指针。
}
```

cpp

输出：

```text
10
11
12
6
```

调用 `sum(values,3)` 时，values 转成首元素地址，3 告诉函数处理三项。函数中的 values 已是指针，sizeof 查询的是指针大小；原数组有几项则由传入的 n 表达。

参数形式	函数得到什么	是否能修改原对象
`int value`	整数副本	不能
`int& value`	原整数的别名	能
`const int& value`	只读别名	不能通过该引用修改
`int* p`	地址的副本	可通过有效的 `*p` 修改所指整数

`const int* p` 限制通过 p 修改所指整数；`int* const p` 限制 p 自己改指向；`const int* const p` 同时限制两者。离开函数后，普通局部变量已销毁，不能返回它的地址或引用给外部继续使用。

2. auto 与范围 for：逐项绑定原数组元素

先看 `auto x=10;`：初始化值是 int，编译器把 x 的类型推导为 int；`auto` 不表示类型能在运行中随意改变。范围 for（也叫 for-each）则让我们依次处理一个范围里的元素，不必自己写下标循环。

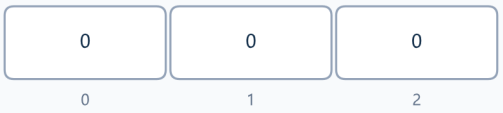
写法	本轮的 i	修改是否写回数组
`for (auto i : a)`	当前元素的副本	不能
`for (auto& i : a)`	当前元素的引用，这里为 int&	能
`for (const auto& i : a)`	当前元素的只读引用	不能修改，读取时避免复制

题目原文（自拟）： 输入恰好三个 int 范围内的整数，存进长度 3 的原生数组，再按输入顺序输出，以单个空格分隔。

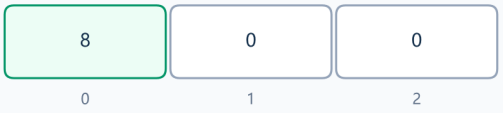
范围 for：引用逐轮绑定到真实元素

a 是长度为 3 的原生数组；auto& i 不复制元素，也不是元素下标。

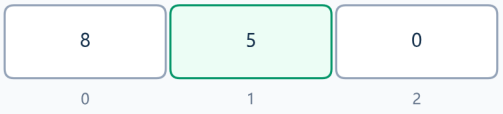
1. int a[3]{}：先有三个 0，等待输入



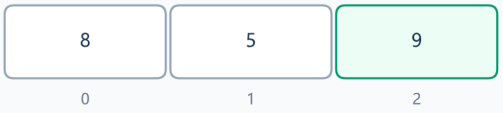
2. 第一轮 i 绑定 a[0]；cin>>i 读入 8，写回 a[0]



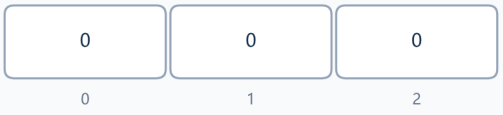
3. 第二轮 i 绑定 a[1]；读入 5，之前的 8 保留



4. 第三轮 i 绑定 a[2]；读入 9，输入完成



5. 若每轮用 auto i，则输入只写入副本，a 仍为三个 0



用 const auto& 读取这三个元素：8 5 9；引用身份由声明中的 & 决定。

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

int main() {
    int a[3]{}; // 已创建三个 0；范围 for 会遍历数组的完整长度 3。
    for (auto &i : a)
        cin >> i; // i 是当前元素的引用，不是下标。
    bool first = true;
    for (const auto &i : a) {
        if (!first)
            cout << ' ';
        cout << i; // 只读取数组元素，不修改它。
        first = false;
    }
    cout << '\n';
}
```

输入：

```text
8 5 9
```

输出：

```text
1  8 5 9
```

解释： 三轮的 i 分别绑定到 `a[0]`、`a[1]`、`a[2]`，cin 的输入写回对应元素。若改成 `auto i`，输入只写入三个临时副本，原数组仍为 `0、0、0`。原生数组的范围 for 遍历整个数组；只处理大数组的前 n 项时，`i<n` 的下标循环能明确控制实际访问数量。

3. 动态申请与释放

固定数组不能用输入 n 决定类型中的长度，但有时确实要在运行时申请 n 项。指针可以接收动态申请结果；先理解谁申请、谁释放，再学习自动管理内存的容器和对象。

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

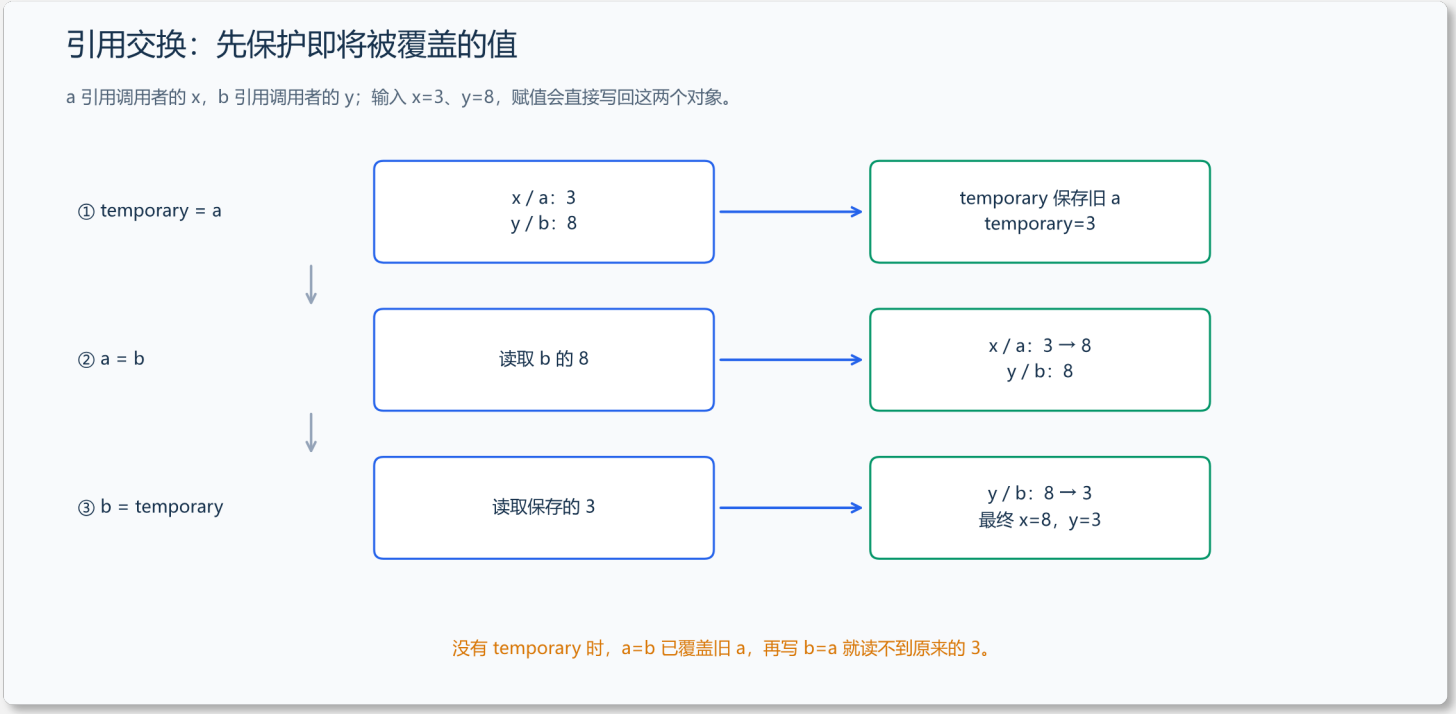
int main() {
    int *value = new int(42); // 申请一个整数，并初始化为 42。
    cout << *value << '\n';
    delete value;               // 单个对象的 new 与 delete 对应。
    value = nullptr;            // 释放后不能继续解引用旧地址。
    int *values = new int[3]{}; // 数组申请，三个元素初始化为 0。
    values[0] = 7;
    cout << values[0] << '\n';
    delete[] values; // 释放由 new[] 创建的整个数组。
    values = nullptr;
}
```

输出为 42、7，各占一行。`new int[n]` 可以使用运行时的正整数 n，申请成功后同样要按实际长度访问；它不会替我们记录有效长度或自动释放。重复释放、忘记释放和释放后继续访问都需要避免。后面的 STL 与资源管理章节会用这个问题说明自动管理的用途。

4. 通过引用交换：先保存将被覆盖的旧值

题目原文（自拟）： 输入两个绝对值不超过 10^9 的整数，通过引用参数交换它们，再输出交换后的值。

推导： 如果只执行 `a=b; b=a;`，第一步就覆盖了旧 a，第二步读到的也是原 b。先用 temporary 保存旧 a，再把旧 b 写入 a，最后把 temporary 写入 b。两个引用参数分别绑定调用者的变量，三次赋值会写回原对象。




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

void swapValues(long long &a, long long &b) {
    long long temporary = a; // 保存旧 a，避免下一步覆盖后丢失。
    a = b;
    b = temporary; // 把旧 a 写到 b，三个动作完成交换。
}

int main() {
    long long x, y;
    cin >> x >> y;
    swapValues(x, y); // a、b 分别成为 x、y 的别名。
    cout << x << ' ' << y << '\n';
}
```

输入 `3 8`，输出 `8 3`。若把参数改成不带 `&` 的值参数，只会交换两份副本；函数结束后 `x`、`y` 仍保持原值。两值相同或两参数绑定同一对象时，这种三步交换也能保持正确结果。

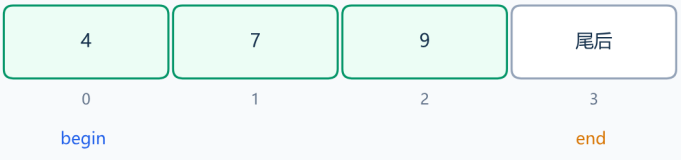
5. 指针移动与数组下标的对应关系

原生数组在很多表达式中转成首元素指针。对指向同一数组的指针，`p+i` 向后移动 `i` 个元素，而不是移动 `i` 个字节；`*(p+i)` 读取对应元素，与 `p[i]` 的访问含义相同。

固定数据演示： 数组为 4、7、9，输出中间项，再从首元素指针走到尾后边界，求三项总和。

指针按元素前进，尾后位置只作边界

values={4,7,9}；begin 指向第 0 项，end=begin+3 指向尾后，不能读取 *end。



p 的位置	p!=end	*p	sum
begin+0	真	4	0+4=4
begin+1	真	7	4+7=11
begin+2	真	9	11+9=20
begin+3	假	不解引用	仍为 20

每次 ++p 前进一个 int 元素；判断失败后停止，边界本身不作为元素访问。

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

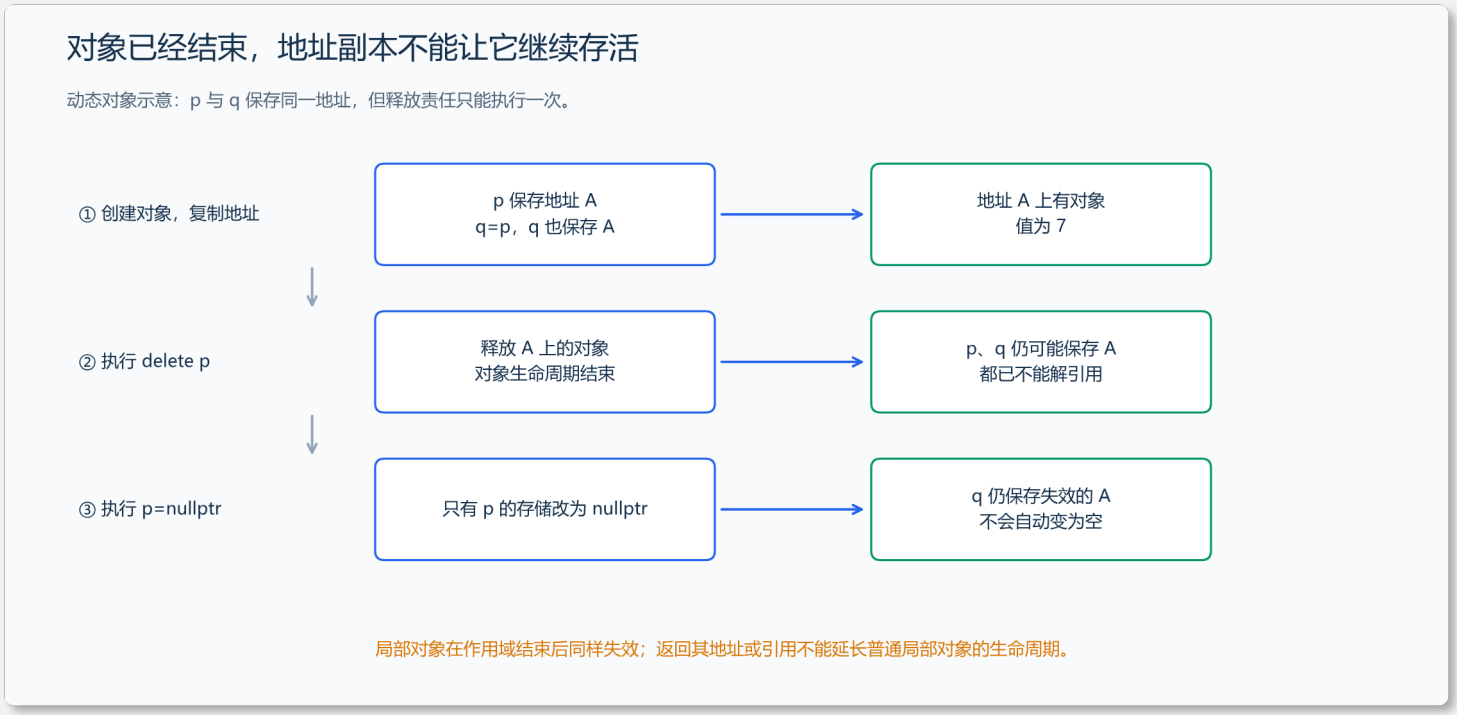
int main() {
    int values[3] = {4, 7, 9};
    int *first = values;           // 在此表达式中，数组转成首元素地址。
    int *last = values + 3;        // 合法的尾后指针，不能通过 *last 读取。
    cout << *(first + 1) << '\n'; // 对应 values[1]，输出 7。
    int total = 0;
    for (int *p = first; p != last; ++p) {
        total += *p; // 先读取当前项，再由 ++p 移到下一元素。
    }
    cout << total << '\n'; // 输出 20。
}
```

first 指向第 0 项，last 指向尾后位置。循环每次先比较位置，再读取并移动，所以不会读取尾后位置；合法的移动范围从首元素到尾后位置。`sizeof(values)/sizeof(values[0])` 用数组总大小除以单项大小，得到元素数；函数的指针参数只保存地址，sizeof 得到指针大小，因此长度要另外传入。

6. 对象结束生命周期，地址不会自动变成有效新对象

情况	对象何时消失	之后怎样处理访问
块内普通局部整数或数组	离开该块	不能保留地址或引用继续访问
new 创建的对象	匹配的 delete 执行后	旧地址失效；用 delete 释放所创建的对象
static 局部对象	通常持续到程序结束	名字仍局限于其声明范围

情况	对象何时消失	之后怎样处理访问
某对象的只读引用	取决于所引用的对象	const 不会让原对象永久存在



把某个指针设为 nullptr 只修改这一份指针，不会自动清除其他指向同一对象的指针。释放后另一份地址副本仍可能看起来是一个地址，实际已不能解引用。返回局部变量地址、重复 delete、用 delete 释放 new[] 得到的数组，都是生命周期或释放配对错误。

普通局部对象常由实现中的栈管理，动态对象通常从动态存储区取得；判断能否访问的依据是对象生命周期和语言规则，不是看到某个地址“好像还没变”。后面的容器与资源管理会用自动释放减少手工维护责任。

（九） auto、lambda 与结构化绑定

已经能用引用访问原数据，也见过 auto 推导整数类型。现在从三个小应用理解它们如何减少重复：推导变量类型、临时写一个计算动作、给数组中的两个字段取名字。

1. auto 类型推导与引用绑定

```cpp
auto count = 10;    // 初始化表达式为 int，所以 count 是 int。
auto price = 12.5; // price 是 double。
int value = 3;
auto copy = value;         // int 副本；修改 copy 不会改变 value。
auto &alias = value;       // int&，alias 是 value 的别名。
const auto &view = value; // const int&，只能通过 view 读取。
```

auto 的基本声明需要提供可推导类型的初始化表达式。普通的 `auto copy=value` 不会保留 value 的引用身份；需要绑定就显式写 `auto&`。这里没有“万能类型”，count 创建后仍是一个确定类型的变量。

2. lambda：保存一个小计算动作

题目原文（自拟）： 程序无需输入。x 初值为 3，分别建立按值和按引用捕获 x 的计算动作，每个动作接收 y，返回 x+y。再把 x 改为 5，传入 y=2，输出两个结果。




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

int main() {
    int x = 3;
    auto byValue = [x](int y) { return x + y; };       // 创建时保存 x 的副本 3。
    auto byReference = [&x](int y) { return x + y; }; // 绑定原 x，调用时读它。
    x = 5;
    cout << byValue(2) << ' ' << byReference(2) << '\n'; // 分别得到 3+2、5+2。
}
```

输出 `5 7`。lambda 可以按“小函数”阅读：捕获列表 `[]`、参数列表 `(int y)`、函数体 `{ ... }`。返回类型在这里由 return 表达式推导，也可显式写 `-> int`。创建 lambda 时不执行函数体，像调用函数一样写 `byValue(2)` 才执行它。

`[x]` 捕获副本，`[&x]` 捕获引用；`[=]`、`[&]` 则按相应方式捕获函数体用到的外部局部变量。引用捕获不自动延长 x 的生命周期，不能让这个动作在 x 销毁后继续访问它。后面会把同样的动作交给排序算法，或用 std::function 保存为可替换的规则。

3. 结构化绑定：副本与引用仍要分清

如果数组的两项表示行和列，每次写 `position[0]`、`position[1]` 不易看出含义。C++17 的结构化绑定可以一次给两项取名字。




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

int main() {
    int position[2] = {2, 3};
    auto [row, column] = position;              // 得到数组的独立副本中的两个字段。
    row = 9;                                    // 原 position[0] 不变。
    auto &[storedRow, storedColumn] = position; // 引用绑定到原数组。
    storedRow = 7;                              // 现在改动的是 position[0]。
    cout << row << ' ' << column << '\n';
    cout << position[0] << ' ' << storedColumn << '\n';
}
```

输出为 `9 3` 、 `7 3` ，各占一行。数组有两项，就用 row、column 两个名字分别绑定。后面的 pair、tuple 和映射记录也可使用相同写法，先理解这两个数组元素上的副本与引用即可。

4. auto 推导的是表达式已有类型

声明	推导结果	为什么
`auto value=5/2;`	int，值为 2	初始化表达式已经做了整数除法
`auto value=5.0/2;`	double，值为 2.5	操作数使除法按浮点进行
`const int source=7; auto copy=source;`	copy 为可修改的 int 副本	普通按值 auto 不保留来源的顶层 const
`auto& alias=source;`	const int&	引用仍受原对象只读类型约束

auto 不会推测“我们想要小数”或“以后可能放更大数”。先把表达式的类型算对，再让 auto 推导；同一声明中的多个变量也不能要求 auto 同时成为互相矛盾的类型。

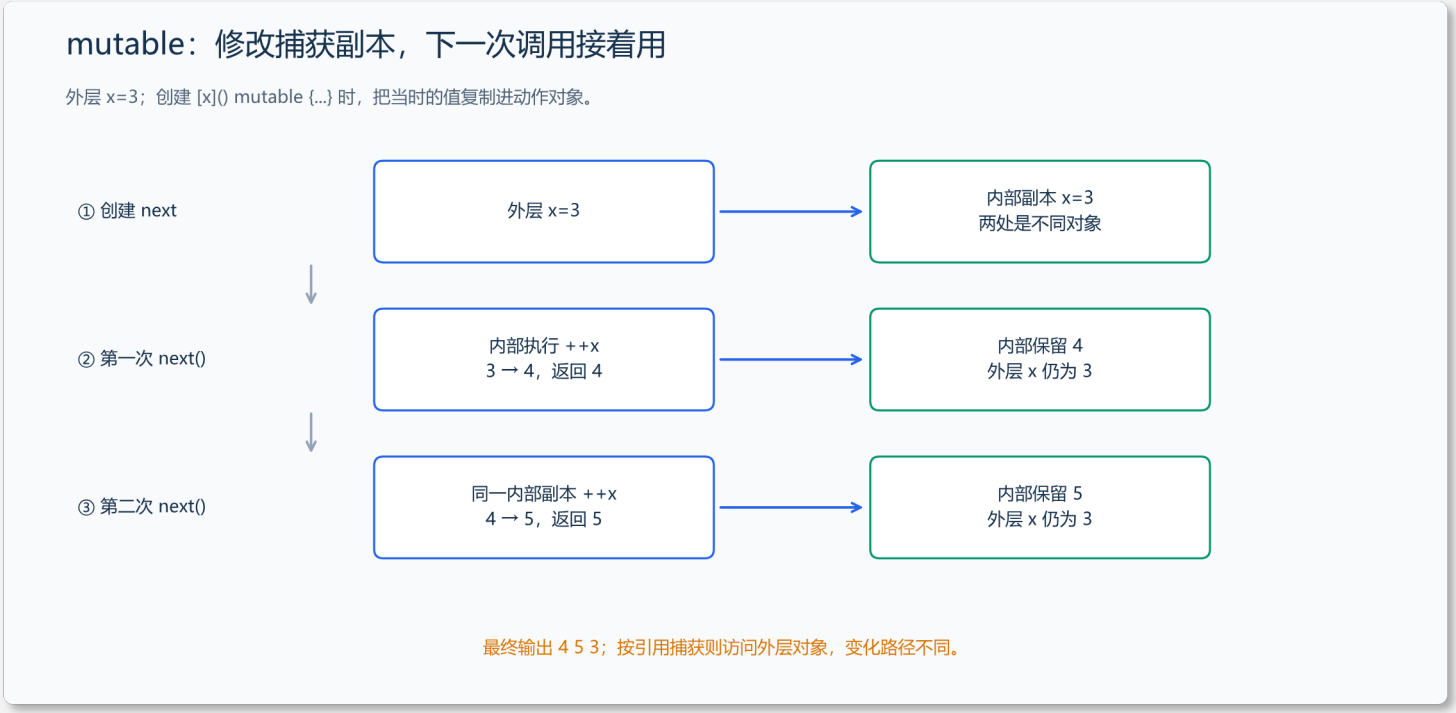
5. 按值捕获与 mutable 状态更新

题目原文（自拟演示）： x 初值为 3，lambda 按值捕获 x，每次调用给自己的副本加一并返回；连续调用两次，最后输出两结果与原 x。

推导： 普通按值捕获的 lambda 在函数体中默认不能直接修改这个捕获副本。添加 mutable 允许修改副本；副本是该 lambda 对象的状态，不会每次调用又从原 x 重新复制。

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

int main() {
    int x = 3;
    auto next = [x]() mutable {
        return ++x; // 改 lambda 自己保存的副本，不改外层 x。
    };
    int first = next();                                    // 副本从 3 变成 4。
    int second = next();                                   // 同一动作中的副本从 4 继续变成 5。
    cout << first << ' ' << second << ' ' << x << '\n'; // 输出 4 5 3。
}
```



对照按引用捕获：引用方式会修改外层 x，mutable 的按值方式只修改内部副本。需要稍后执行的动作时，也要检查被引用对象是否仍然存活；std::function 能保存动作，但不会替它延长外部变量的生命周期。

（十） 输入细节、文件与异常

如果先输入年龄，再读取下一行的姓名，为什么直接读整行会得到空行？原因在于：格式化读取整数后，行末换行还留在输入流里。先用已经学过的 char 数组观察输入流中还剩什么。

1. 整行读取与残留换行

题目原文（自拟）： 第一行输入一个 int 年龄，第二行输入至多 100 个英文字符组成的姓名，允许空格和空行。输出完整姓名、一个空格和年龄。




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

int main() {
    int age;
    if (!(cin >> age))
        return 0; // 先确认整数读取成功。
    char ch;
    while (cin.get(ch) && ch != '\n') {
    } // 按字符消耗本行余下内容，含换行。
    char name[101]{};        // 最多 100 个内容字符，另留一格给结尾的 '\0'。
    cin.getline(name, 101); // 把下一整行写入已有字符数组，并结束于 '\0'。
```

```cpp
    cout << name << ' ' << age << '\n';
}
```

输入：

```text
18
Alice Smith
```

输出：

```text
Alice Smith 18
```

`cin.get(ch)` 读取一个字符，不跳过空白；`cin.getline(name,101)` 接收字符数组和容量，读取一行、取走换行，但不把换行写入数组。C 风格字符串以 `\0` 标记结束，cout 输出 name 时读到该标记停止。100 个内容字符加上结尾的 `\0`，共需要 101 个槽位，所以声明 `char name[101]` 并传入容量 101。文本超过容量时，getline 会设置失败状态。

`cin >> ws` 跳过接下来的空白字符与空行；直接调用 getline 则从当前位置读取这一行，保留开头空白。学过 string 后，还会用它读取并保存整行文本。

2. 把输入来源换成文件

`<fstream>` 提供文件流，读取整数仍可使用已学的 `>>`。下面是一个无需标准输入的演示：同目录的 data.txt 保存 `1 2 3`，程序输出其总和 6。

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

int main() {
    ifstream input("data.txt"); // 尝试打开名为 data.txt 的文件用于读取。
    if (!input) {
        cerr << "文件打开失败\n"; // cerr 用于诊断输出，不是题目要求的答案。
        return 1;
    }
    long long total = 0;
    int value;
    while (input >> value)
        total += value; // 与 cin 相同，读取失败时退出。
    cout << total << '\n';
}
```

ifstream 用于读文件，ofstream 用于写文件，销毁时自动关闭。`ofstream output("result.txt");` 默认可能覆盖旧文件，`ofstream output("result.txt",ios::app);` 以追加方式写到末尾；选模式前先确定要保留哪些内容。竞赛评测通常读 cin、写 cout，文件演示用于理解输入输出对象。

3. 异常的抛出与捕获

除法要求除数不为 0。函数发现除数为 0 时，可以抛出异常，把错误交给调用者处理。下面抛出整数 0，外层用 catch 接住它，观察程序从计算路径转到错误处理路径。

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

int divide(int a, int b) {
    if (b == 0)
        throw b;  // 抛出整数 0，立即离开正常计算路径。
    return a / b; // 本演示仅使用 10 与 2，结果在 int 范围内。
}

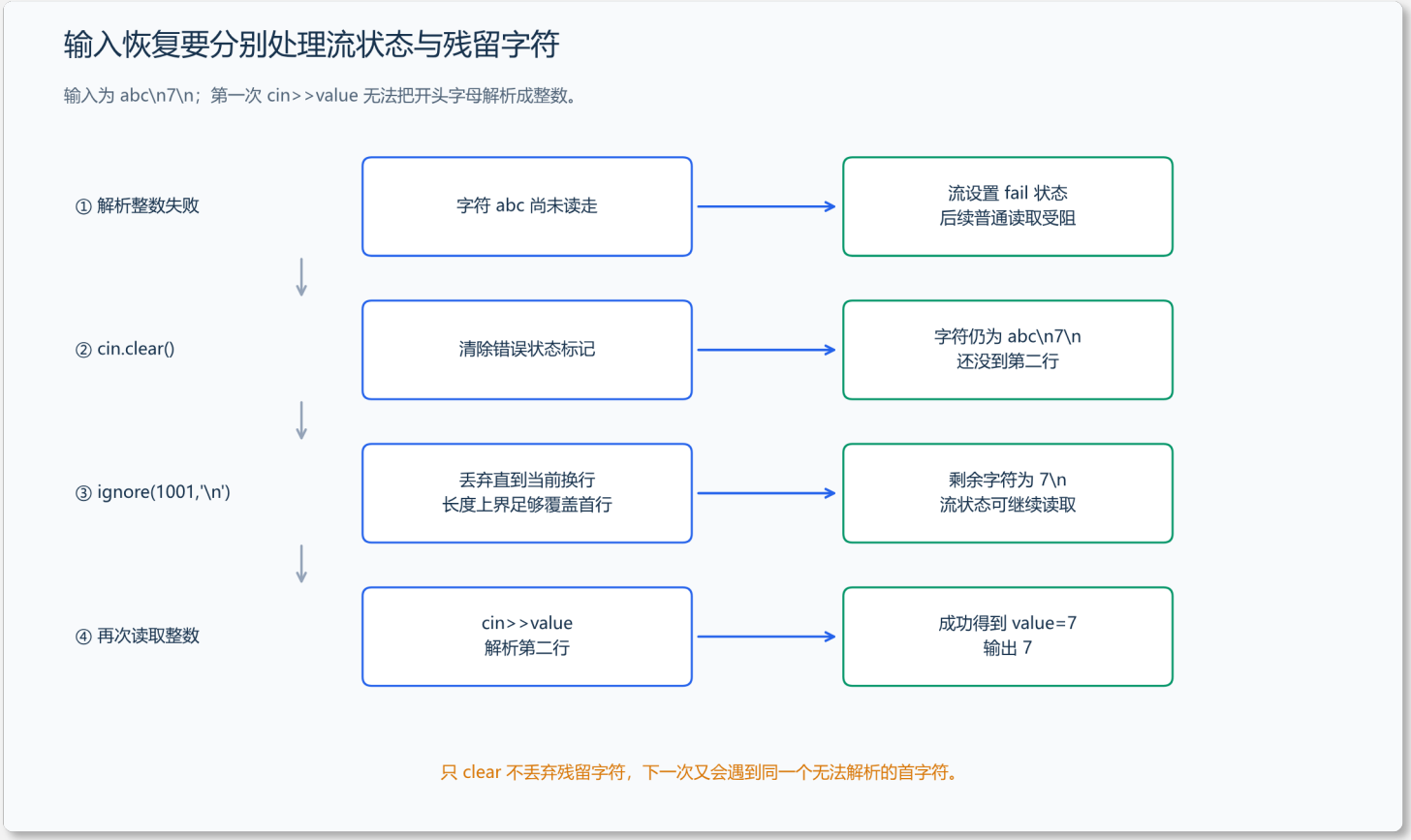
int main() {
    try {
        cout << divide(10, 2) << '\n';
        cout << divide(10, 0) << '\n'; // 此次调用抛出异常，不会输出商。
    } catch (int divisor) {
        cout << "invalid divisor: " << divisor << '\n'; // 接收并处理抛出的整数。
    }
}
```

输出为 `5`、`invalid divisor: 0`，各占一行。try 包住可能失败的操作，throw 交出失败信息，匹配的 catch 处理它；并不是所有错误都会自动抛异常。原生数组越界、悬空指针等错误仍要事先避免。后面遇到标准异常类型时，再解释其头文件、错误信息和捕获方式。

4. 读取失败后：先清状态，再处理残留数据

`cin>>value` 不只是取出字符，还会记录输入状态。逻辑判断 `if(cin>>value)` 表示这次提取有没有成功；eof 表示检测到输入末尾，fail 表示格式化读取等操作失败，bad 表示更严重的流故障。EOF 不是一个给普通整数自动赋入的数值。

题目原文（自拟演示）： 输入两行。第一行长度至多 1000，首字符是英文字母，不能解析为整数；第二行是一个合法的 int。第一次解析失败后输出 INVALID，恢复输入并丢弃第一行剩余内容，再输出第二行的整数。




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

int main() {
    int value = 0;
    if (!(cin >> value)) {
        cout << "INVALID\n";     // 演示题要求报告第一次失败。
        cin.clear();             // 重置失败状态；它不删除还没读走的错误字符。
        cin.ignore(1001, '\n'); // 至多丢弃 1001 个字符，包含遇到的换行。
    }
    if (cin >> value) {
        cout << value << '\n'; // 第二行读入成功后，输出这次得到的整数。
    }
}
```

输入为 `abc`、`7`，各占一行，输出为 `INVALID`、`7`。只 clear 而不丢弃错误字符，下一次仍会尝试读取同一个 a，再次失败；只 ignore 而不先 clear，在失败状态下也不能完成预期提取。首行至多 1000 个字符，再加一个换行，所以 ignore 的最大数量设为 1001，恰好能丢弃完整的这一行。

竞赛中通常将“读入一组成功”放进循环条件：读到数据就处理，输入结束就退出。上面的 clear 与 ignore 用来演示格式错误后的恢复；EOF 已表示数据结束，读到 EOF 后直接结束读取。

5. scanf、printf 的格式化输入输出

`<cstdio>` 提供 C 风格输入输出。scanf 根据格式字符串把结果写到给定地址，printf 根据格式说明读取参数值；这与流的 `>>` 、 `<<` 是另一套接口。指针章节已经解释 `&x` ，这里它表示 scanf 将写入的变量地址。

类型或格式	scanf 输入	printf 输出
int	`%d` 与 int*	`%d` 与 int
long long	`%lld` 与 long long*	`%lld` 与 long long
double	`%lf` 与 double*	`%f` ; `%.2f` 输出两位小数
char	`%c` 与 char*	`%c` 与字符值

题目原文（自拟）： 输入一个 int、一个 long long、一个 double 和一个英文字符，按同样顺序输出一行；double 保留两位小数，字段间一个空格。

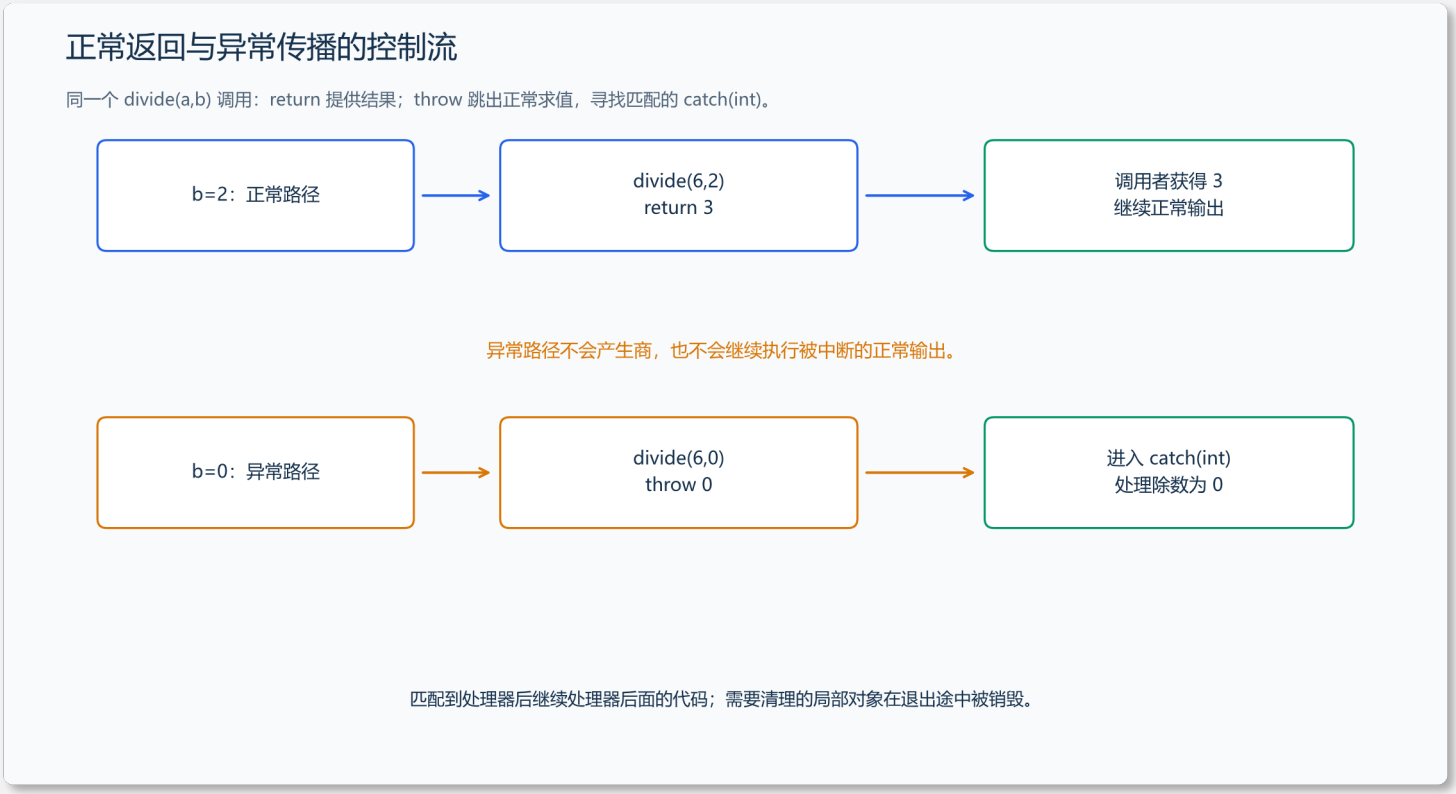
```cpp
#include <cstdio>

int main() {
    int number;
    long long total;
    double ratio;
    char grade;
    if (std::scanf("%d%lld%lf %c", &number, &total, &ratio, &grade) != 4) {
        return 0; // 返回值表示成功赋值的项目数；本例要求四项都读到。
    }
    std::printf("%d %lld %.2f %c\n", number, total, ratio, grade); // 输出传值，不传地址。
}
```

输入 `7 10000000000 2.5 A` ，输出 `7 10000000000 2.50 A` 。数值格式会跳过开头空白， `%c` 本身不跳过，所以在它前面放空白以消耗分隔符。用 C 风格字符串数组接收 `%s` 时要按容量限制宽度，不能任意长度写入已有数组。

scanf 把数据写入所传地址，printf 读取传入的值。使用序言中的整数宏后，int 会替换为 long long，读取格式相应写成 `%lld` 。这个程序使用 scanf、printf 完成输入输出，使用 cin、cout 的程序则按序言设置标准流加速。

6. throw 与 return 走的是不同路径



对应前面的 divide 演示，`return a/b` 返回一个结果，调用者继续正常输出；`throw b` 则中断这次正常求值，寻找外层匹配的 catch。传入除数 0 时，调用后的那次输出不会得到一个“默认商”，而是转到 `catch(int divisor)` 输出错误信息。

抛出异常会使离开的作用域按规则清理已构造的局部对象，这称为栈展开；面向对象部分会把它与析构、资源管理连接。return 处理正常结果，异常处理失败路径，数组越界等错误并不自动成为可捕获异常。

（十一） 命名空间、头文件与多文件程序

函数逐渐增多后，可以把加法工具与主程序放到不同文件；下面围绕一次跨文件调用，解释声明、头文件、命名空间和链接。

1. 命名空间与同名标识符

题目原文（自拟演示）： 为两套整数工具分别定义名为 calculate 的函数，一套求和、一套求差；对 2、3 调用两函数，输出 5、-1。

```
#include <iostream>

namespace addition {
int calculate(int a, int b) {
    return a + b;
} // 名字属于 addition。
} // namespace addition
```

```cpp
namespace subtraction {
int calculate(int a, int b) {
    return a - b;
} // 同名但属于另一个空间。
} // namespace subtraction

int main() {
    std::cout << addition::calculate(2, 3) << '\n'; // :: 明确选求和工具。
    std::cout << subtraction::calculate(2, 3) << '\n';
}
```

`using addition::calculate;` 可在当前作用域引入这个指定名字，`using namespace addition;` 则让该空间的名字参与查找。若把多个拥有同名函数的空间都引入，再只写 calculate，可能无法判断应该选择哪一个。学习程序用 using namespace std 减少重复前缀，项目头文件通常避免全局这样写。

2. 多文件程序的声明、定义与链接

头文件通常放声明，源文件放定义。自定义头文件可使用包含保护，避免同一份声明或类型定义被重复包含。`#ifndef` 检查标记是否尚未定义，`#define` 设置标记，`#endif` 结束这一段预处理判断；这些不是运行中的 if 分支。标准头文件常用尖括号，项目头文件常用双引号并按项目路径寻找。

`math_utils.h`：

```cpp
// 头文件保护避免重复定义；这里只声明函数，不提供函数体。
#ifndef MATH_UTILS_H
#define MATH_UTILS_H

namespace math_utils {
int add(int a, int b);
}

#endif
```

`math_utils.cpp`：

```cpp
// 在源文件中提供命名空间内函数的定义，与头文件声明一致。
#include "math_utils.h"

int math_utils::add(int a, int b) {
    return a + b;
}
```

`main.cpp`：

```cpp
// 调用另一个源文件实现的函数，链接时需要同时加入 math_utils.cpp。
#include <iostream>
#include "math_utils.h"

int main() {
    std::cout << math_utils::add(2, 3) << '\n';
}
```

main.cpp 看见声明就能编译调用，math_utils.cpp 提供函数体；链接阶段把这个调用与实现连接。修改头文件接口后，也要重新编译使用它的源文件。

三个文件放在同一目录后，执行：

```shell
g++ -std=c++17 main.cpp math_utils.cpp -o main
```

普通函数与全局变量通常在一个源文件中定义，头文件提供声明，让其他文件知道怎样使用。inline 函数可以按相应规则把定义放在头文件中，供多个源文件包含；至于机器代码是否把调用展开，则由编译器决定。模板定义为什么也常放在头文件中，下一章会结合实例化解释。

头文件不是运行时被“打开并执行”的脚本，也不是看到声明就已有实现。include 保护解决同一编译单元内重复包含，不能替代跨源文件的定义规则；项目扩大时，应检查声明、定义和参与链接的文件三者是否一致。

（十二） 常用数学函数与数值计算

计算距离需要平方根，计算需要多少辆车可能需要向上取整。先把这些工具看成已学过的函数：给定参数，得到返回值，再核对返回类型、取整方向和定义域。

头文件	函数	含义与结果
`<cstdlib>`	`abs(int)`	整数绝对值，例如 abs(-3) 得到 3
`<cmath>`	`sqrt(x)`	非负数 x 的平方根近似值
`<cmath>`	`fabs(x)`	浮点绝对值
`<cmath>`	`floor(x)`、`ceil(x)`	向负无穷、正无穷取整，返回值仍为浮点类型
`<cmath>`	`round(x)`	就近取整，恰好一半时向远离 0 的方向
`<cmath>`	`pow(x,y)`	浮点乘方，结果可能带有舍入误差
`<cmath>`	`isfinite(x)`	判断是否为有限值，排除无穷大与 NaN
`<cmath>`	`acos(x)`	反余弦，实数参数在 [-1,1]，返回弧度；`acos(-1.0)` 得到 π 的浮点近似值

题目原文（自拟演示）： 程序无需输入。求 9.0 的平方根，分别对 -2.7 向下、向上取整，再求 2.0 的三次方，分四行输出。

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

int main() {
    cout << sqrt(9.0) << '\n';      // 3；浮点输出不一定自动带小数点。
    cout << floor(-2.7) << '\n';    // -3，向下取整，向负无穷，不是截去小数部分。
    cout << ceil(-2.7) << '\n';     // -2，向上取整，向正无穷。
    cout << pow(2.0, 3.0) << '\n'; // 2.0 的三次方，这里输出 8。
}
```

把 -2.7 转成 int 得到 -2，是向 0 截去小数部分；floor(-2.7) 得到 -3，是向负无穷方向取整。普通 int 的最小值为 -2147483648 时，它的正绝对值是 2147483648，比 INT_MAX 大 1；先转成 long long 再求绝对值，就能保存这个结果。后面的快速幂用整数乘法逐步计算整数次幂。

1. 整数坐标的距离计算

题目原文（自拟）： 输入两点的四个整数坐标 x1、y1、x2、y2，绝对值不超过 10^6，输出两点距离的 double 近似值，使用默认浮点格式。

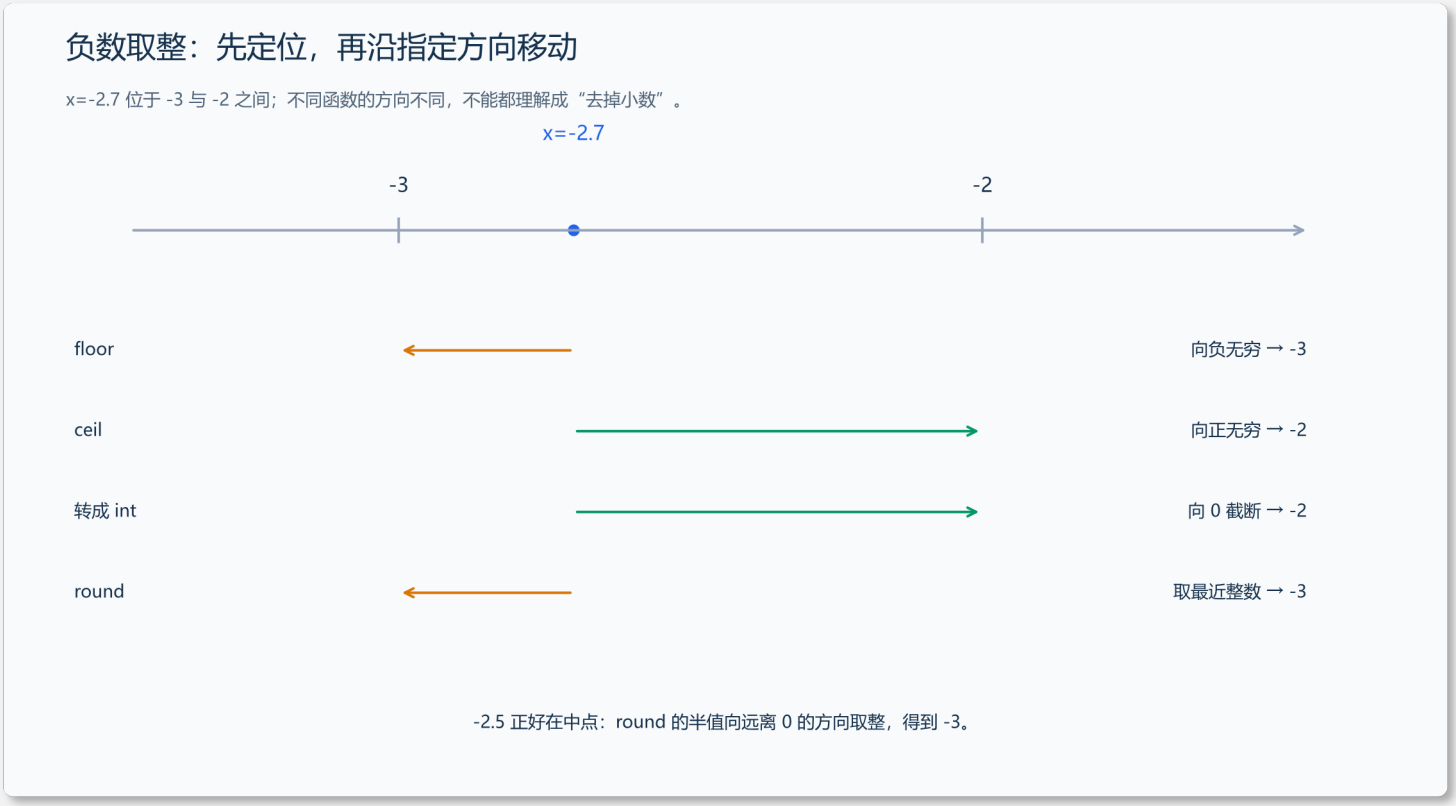
推导： 横向差 dx、纵向差 dy 是直角三角形两边，距离平方为 dx²+dy²。先让差和乘法使用 double，再调用 sqrt；若先用 int 平方，即使最后赋给 double，也可能在中间步骤越界。

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

int main() {
    int x1, y1, x2, y2;
    cin >> x1 >> y1 >> x2 >> y2;
    double dx = static_cast<double>(x1) - x2; // 在后续平方前选择浮点计算。
    double dy = static_cast<double>(y1) - y2;
    cout << sqrt(dx * dx + dy * dy) << '\n'; // sqrt 接收非负值，返回平方根近似。
}
```

输入 `0 0 3 4`，输出 `5`。表达式先得到 9+16=25，再开方；输出 5 不代表距离类型是 int，默认浮点格式可能省略小数点。题目要求小数位时，再使用下一节的格式设置。

2. 取整方向要沿数轴判断



对 -2.7，floor 向负无穷得到 -3，ceil 向正无穷得到 -2，转换成整数向 0 截断得到 -2；round 就近得到 -3。对于 -2.5，round 的一半规则是远离 0，得到 -3。负数更容易看出方向差别：向下是往数轴左侧走，向上是往右侧走，截断则朝 0 靠近。

3. 浮点误差与判断精度

double 在常见二进制浮点实现中用有限位表示数，0.1 等十进制小数通常不能精确保存。于是 `0.1+0.2` 与 0.3 的比较可能不相等；格式化显示都像 0.3，也不代表内部值完全相同。

题目原文（自拟演示）： 比较 a=0.1+0.2 与 b=0.3，输出它们的差是否在绝对容差 10^{-9} 内。

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

int main() {
    double a = 0.1 + 0.2, b = 0.3;
    constexpr double epsilon = 1e-9;      // 两值的绝对差不超过这个容差时，视为足够接近。
    bool close = fabs(a - b) <= epsilon; // fabs 求差的绝对值。
    cout << close << '\n';               // 差小于容差，输出 1。
}
```

绝对容差比较两数相差多少，相对容差比较这个差占数值的比例。例如题目允许绝对误差 10^{-6}，就按这个精度判

断；允许相对误差时，还要结合数值大小。求精确的整数幂时，后面会用整数乘法实现快速幂。

这些函数得到数值后，还要按题面要求输出，下一节先处理精度和格式。随后学习模板，解释同一段操作如何用于不同类型；进入 STL 时，尖括号中的类型就有明确含义了。

（十三） 常用标准流与输出格式

会读取整数与整行文本后，再核对题目要求怎样输出：多少位小数、是否需要补齐宽度、哪些内容属于答案。

1. 标准输入输出流

```cpp
#include <iostream>
using namespace std; // 使用标准命名空间

int main() {
    cout << "Hello, World!" << endl; // 输出并换行
    int x;
    cin >> x; // 输入
    cout << "x = " << x << endl;
}
```

也可以显式使用命名空间前缀：

```cpp
// std:: 前缀明确命名空间；endl 同时换行并刷新输出流。
#include <iostream>
int main() {
    std::cout << "Hello, World!" << std::endl;
}
```

说明：

`cin` 是标准输入流，通常使用 `>>` 按类型读取；`scanf` 按格式字符串读取，它们的用法和缓冲机制不同。

`cout` 是标准输出流，通常使用 `<<` 写出；`printf` 使用格式字符串，不能把两者的占位符规则混用。

`endl`：换行并刷新缓冲区。

2. iomanip 格式化输出

需要头文件：

```cpp
// 输出精度、宽度和填充等格式控制需要此头文件。
#include <iomanip>
```

2.1 保留小数位数（`fixed` + `setprecision`）

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

int main() {
    double pi = 3.1415926535;

    cout << fixed << setprecision(2) << pi << endl; // 3.14
    cout << fixed << setprecision(4) << pi << endl; // 3.1416
}
```

说明：

`fixed << setprecision(n)`：按定点格式显示小数点后 n 位。

默认浮点格式下，`setprecision(n)` 控制有效数字；`scientific` 格式下也控制小数点后的位数。

2.2 设置宽度（`setw`）

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

int main() {
    int a = 42, b = 1234;
    cout << setw(6) << a << endl; // 宽度为6，右对齐
    cout << setw(6) << b << endl; // 字段宽度不足时自动扩展，不截断数据
}
```

2.3 设置填充字符（`setfill`）

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

int main() {
    int x = 7;
    cout << setw(5) << setfill('0') << x << endl; // 00007
}
```

2.4 综合示例

```cpp
// fixed 后 setprecision 表示小数位数；setw 只作用于下一次格式化插入。
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
    double score = 95.6789;
```

```cpp

    cout << "默认输出: " << score << endl;
    cout << "保留两位小数: " << fixed << setprecision(2) << score << endl;

    cout << "宽度8，空格补齐: [" << setw(8) << 123 << "]" << endl;
    cout << "宽度8，0补齐:    [" << setw(8) << setfill('0') << 123 << "]" << endl;
}
```

输出：

```text
默认输出: 95.6789
保留两位小数: 95.68
宽度8，空格补齐: [     123]
宽度8，0补齐:    [00000123]
```

3. 输出格式状态的保留与重置

设置	影响	后续是否保留
fixed、scientific、defaultfloat	浮点显示形式	保留，直到换模式
setprecision(k)	精度；含义取决于显示模式	保留
setfill(ch)	字段不足时使用的填充字符	保留
setw(k)	下一次相关格式化输出的字段宽度	通常在该项输出后重置为 0
boolalpha、noboolalpha	bool 输出文字或 0/1	保留
hex、oct、dec	整数输出进制	保留，直到换进制

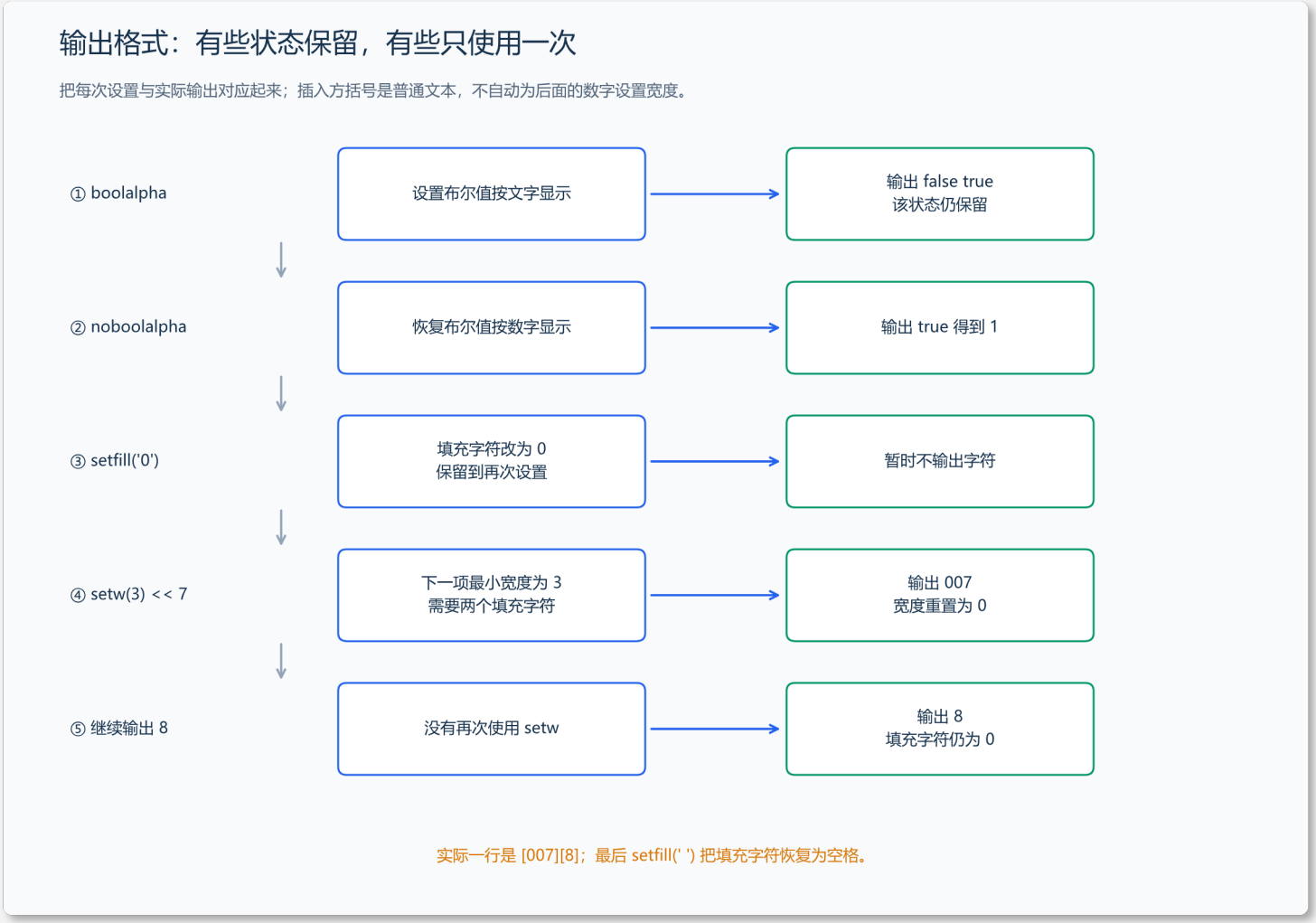
题目原文（自拟演示）： 程序无需输入。先输出 false、true 的文字，再恢复数字输出 true；最后把整数 7 用宽度 3 和字符 0 补齐，紧接着在另一对方括号内输出 8，不再设置宽度。

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

int main() {
    cout << boolalpha << false << ' ' << true << '\n';  // 输出 false true。
    cout << noboolalpha << true << '\n';                // 显式恢复，输出 1。
    cout << '[' << setfill('0') << setw(3) << 7 << ']'; // 本次整数按宽度补成 007。
    cout << '[' << 8 << "]\n";                          // setw 已重置，8 不会自动变成 008。
    cout << setfill(' ');                               // 填充字符会保留，需要时主动恢复为空格。
}
```

输出：

```text
false true
1
[007][8]
```



setw 是最小字段宽度，不会把较长结果截短。fixed 与 setprecision 只改变显示，不把变量的内部值改成四舍五入后的数；需要改变数值时，应明确执行相应计算。用 `defaultfloat << setprecision(6)` 可以恢复常见的默认浮点显示设置。

4. 输出格式的逐字符核对

答案要求一行两个数，就检查数之间是否只有规定的空格、末尾是否换行；输出表格时再分别核对宽度和对齐。先对照样例的结构，不把调试提示一起交给评测。大量普通批量输出用 `'\n'`，交互询问按协议主动刷新；设置显示精度与刷新是不同操作。

（十四） 模板入门：从相同操作到类型参数 T

1. 相同计算在不同类型上的复用

两整数取较大值与两小数取较大值，判断过程都是 `a>b ? a:b`。分别写 int 版和 double 版会重复同一段逻辑；现在希望保留逻辑，把参与计算的类型作为参数。

题目原文（自拟）： 程序无需输入。用同一个取较大值的函数模板，依次计算 `(3,5)`、`(2.5,1.0)`，以及显式按 double 处理的 `(3,2.5)`，各输出一行。

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

template <typename T> T larger(T a, T b) {
    return a > b ? a : b; // 比较两个 T 类型的值，返回较大的一项。
}

int main() {
    cout << larger(3, 5) << '\n';           // 两实参都是 int，推导 T=int，得到 5。
    cout << larger(2.5, 1.0) << '\n';       // 两实参都是 double，推导 T=double。
    cout << larger<double>(3, 2.5) << '\n'; // 明确 T=double，3 转成 3.0 后比较。
}
```

输出：

```text
5
2.5
3
```



沿第一行读：`template` 声明模板，`typename T` 声明一个**类型参数**；接着 `T larger(T a,T b)` 表示返回类型与两个参数类型都使用这个 T。T 不是一个存数字的变量，也不是固定的内置类型，只是这里选用的参数名，换成 Type 或 ValueType 也可以。普通竞赛模板里的 `int T` 是组数变量，与这里的类型参数角色不同。

2. 推导、显式指定与实例化

调用	怎样确定类型	按具体类型理解
`larger(3,5)`	两个参数都推导为 int	`int larger(int,int)`
`larger(2.5,1.0)`	两个参数都推导为 double	`double larger(double,double)`
`larger<double>(3,2.5)`	调用者直接指定 double	两个值先作为 double 参数传入
`larger(3,2.5)`	两个位置分别要求 int 与 double	这个模板的同一个 T 推导冲突，不能这样调用

编译器依据模板和确定的参数形成所需的具体函数，这个过程称为**实例化**。它发生在编译相关的阶段，不是在每次运行时把类型换一遍。模板内部支持哪些操作，要看代入后的类型；若类型没有 `>`，本函数也无法用于它。

`template<class T>` 与 `template<typename T>` 都声明类型参数，int、double 或我们定义的类都可以代入 T。函数模板、推导和实例化规则见 C++17 草案：模板参数 与 调用时的参数推导。

3. 先做一条记录，再把记录里的类型参数化

假设要把一个值装进名为 Box 的记录，整数盒子与小数盒子的区别只有 value 的类型。先认识最小的 struct：它定义一种自定义类型，花括号内声明成员；创建对象后用 `对象.成员` 访问，成员默认可以公开访问。

```cpp
struct IntBox {
    int value; // 每个 IntBox 对象各自保存一个整数。
};
// 在函数中写 IntBox box{7}; 后，box.value 就是 7。
```

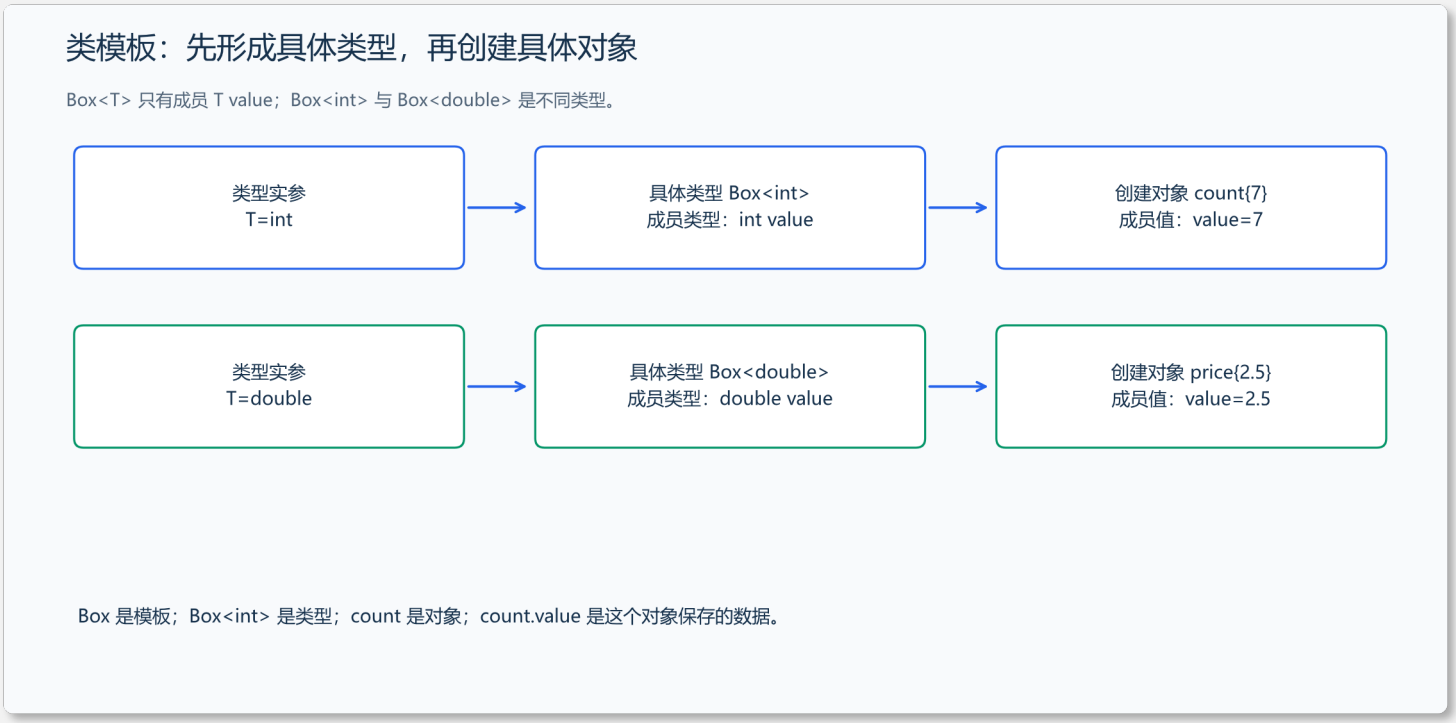
现在把 int 换成 T，仍然只保存一个值：

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

template <typename T> struct Box {
    T value; // 代入什么类型，这个成员就保存什么类型的值。
    T read() const {
        return value;
    } // 成员函数；末尾 const 表示只读该对象。
};

int main() {
    Box<int> count{7};                                  // value 的类型为 int。
    Box<double> price{2.5};                             // value 的类型为 double，是另一种具体类型。
    cout << count.value << ' ' << price.read() << '\n'; // 点号访问数据或调用成员函数。
}
```

输出 `7 2.5`。`Box<int>` 是一个具体类型，`Box<int> count` 才是在创建该类型的对象；Box 本身是模板名称。`Box<int>` 与 `Box<double>` 是不同类型，不能理解为一个对象在运行时改变类型。



struct 内的函数通过对象调用；例如 `price.read()` 读取 price 的 value。这样，一条记录既能保存数据，也能提供操作这些数据的函数。面向对象部分会继续用它讲访问控制、构造和继承。

4. 尖括号里也可能放一个编译期数值

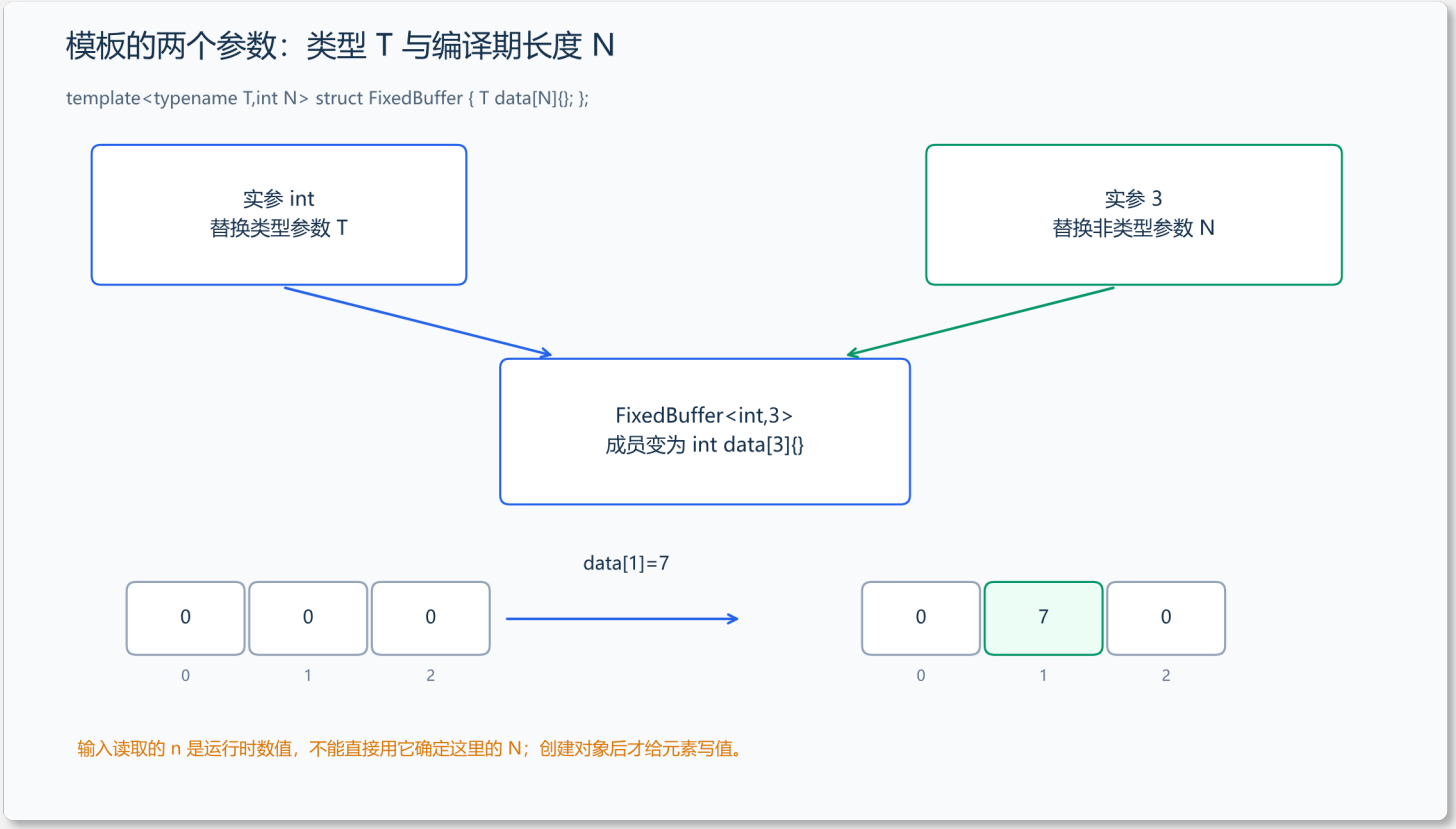
如果还想让记录保存固定长度的原生数组，就需要两个参数：元素类型 T，以及长度 N。N 表示值，不是另一种类型。

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

template <typename T, int N> struct FixedBuffer {
    T data[N]{}; // 本例要求 N>0；用编译期长度创建原生数组。
};

int main() {
    FixedBuffer<int, 3> values; // T=int，N=3，三个整数先初始化为 0。
    values.data[1] = 7;
    for (int i = 0; i < 3; ++i) {
        if (i > 0)
            cout << ' ';
        cout << values.data[i]; // 仍使用已学的成员访问和数组下标。
    }
    cout << '\n';
}
```

输出 `0 7 0`。这里的 3 是**非类型模板实参**，参与确定具体类型；输入读取的普通变量 n 不满足这个编译期长度要求。类型参数通常写 T、Key、Value，数值参数常写 N；判断它们的角色，应看声明，不应只凭字母大小写。



多个类型参数可以分别指定类型，例如 `template<typename A,typename B>` 中的 A、B 可以取不同类型。模板参数也可提供默认值：把 Box 声明为 `template<typename T=int>` 后，`Box<>` 就使用 int；写 `Box<double>` 则使用明确给出的 double。

5. 标准库的类型工具也按同一方式阅读

想知道当前平台 int 能保存的范围，可以使用 `<limits>` 的 `numeric_limits<int>`。`numeric_limits` 是类模板，int 是类型实参；`::min()`、`::max()` 是属于这个具体类型的查询函数，不必创建一个对象。

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

int main() {
    cout << numeric_limits<int>::min() << '\n';       // 当前平台 int 的最小值。
    cout << numeric_limits<int>::max() << '\n';       // 当前平台 int 的最大值。
    cout << numeric_limits<long long>::max() << '\n'; // 换实参，查另一种类型。
}
```

结果取决于实现。普通整数的 min 是最小值；浮点类型的 min 通常表示最小正正规值，若要其最低有限值应使用 lowest。模板定义通常放在头文件中，使调用或创建具体类型的源文件能看见完整定义。

6. 两个类型参数：同一条记录保存不同类型的字段

函数重载中，两份 larger 重复了判断逻辑；模板把变化的类型参数化。类模板同样可以让记录的多个字段分别选择类型，不必要求所有字段共用 T。

题目原文（自拟演示）： 建立一条“编号、等级”记录，编号类型 int、等级类型 char，保存 7 与 A，输出两个字段。

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

template <typename Key, typename Value> struct Record {
    Key key;      // 第一个类型实参决定编号字段的类型。
    Value value; // 第二个类型实参独立决定等级字段的类型。
};

int main() {
    Record<int, char> student{7, 'A'};                    // 参数是类型，花括号里的才是对象的实际值。
    cout << student.key << ' ' << student.value << '\n'; // 输出 7 A。
}
```

把声明、类型与对象分开读：Record 是模板名称，Record<int,char> 是具体类型，student 是该类型的对象。若改成 Record<int,double>，第二字段就换成 double；不是让已经创建的 student 在运行中随意换类型。

`typename T` 与这里的 `class T` 都声明类型参数；`int N` 则声明值参数。实例化要满足模板体中的操作要求，typename 不会自动让任意类型都能使用加法、比较或数组初始化。下一节的 vector、pair<A,B> 都可以按这套参数读法理解，再分别学习它们提供的行为。

到这里，已经能区分模板名称、类型实参、具体类型和对象，也能用两个参数分别指定字段类型。接下来面对运行时长度的序列、查找与排序等任务，用这套读法理解 STL，继续从用途推进。

（十五） STL：容器、迭代器与算法

读题得到 n 个数，要保存、排序或查询它们。手工管理动态数组需要记录长度并负责释放；每道题都重写查找和排序也会重复工作。STL（Standard Template Library，标准模板库）把这些操作组织成可组合的工具，以**容器保存数据、迭代器表示位置、算法处理区间**为基本配合方式。这里把常用标准容器、string 和相关工具一并放在 STL 章节学习；string 是标准字符串类型，pair 是成对数据工具，整个 C++ 标准库还包含输入输出、数学和其他设施。

先按保存的数据和需要的操作认识它们，不必一次背完所有接口。

先读懂操作费用的记号。 设有 n 项，只读一项的工作不随 n 增长，记作 $O(1)$；逐项走一遍记作 $O(n)$；每步把候选大致减半，步数随对数增长，记作 $O(\log n)$。排序常见的 $O(n \log n)$ 可先理解为接近 n 项乘以对数级工作，两层各走 n 项则是 $O(n^2)$。这些记号描述规模增大时的增长，不表示固定秒数；算法部分再配增长图、数值表和内存估算完整推导。

类别	本章内容	首先分清什么
字符串与成对数据	string、pair	字符下标、两个字段、比较顺序
顺序容器	vector、deque、list、array、forward_list	动态或固定大小；连续数组或节点连接；位置访问成本
有序关联容器	set、map、multiset、multimap	保存键还是键值；是否允许等价键重复
无序关联容器	unordered_set、unordered_map 及 multi 版本	按哈希判重；遍历没有有序保证
容器适配器	stack、queue、priority_queue	封装底层容器，限定后进先出、先进先出或优先取出
连接容器与算法的工具	迭代器、algorithm、numeric	半开区间、迭代器能力、比较器、修改后的位置

自学时先熟悉 string、vector 和 pair，再学迭代器、算法库、链表、栈、队列、map、set 与堆；deque 可结合滑动窗口学习，链表与 multi 版本按需要查阅。每节先手算图中的操作，再运行带注释的例题，最后按练习链接独立写程序。表中的复杂度通常假设单个元素的比较、复制或移动为常数；大字符串和复杂对象还需计算这些操作本身的费用。

1. string：字符串

1.1 用 string 保存和访问文本

`string` 把字符按顺序保存，支持下标、遍历、拼接和查找，头文件为 `<string>`。`string s="abc"` 有 3 个字符，有效下标为 `0`、`1`、`2`。`s.size()` 表示字符单元的数量；普通 `string` 保存中文 UTF-8 文本时，数量通常是字节数，不能直接当成汉字个数。

`cin >> s` 跳过开头空白，读到下一段空白停止；`getline(cin,s)` 读取一整行，可以读出空行。`getline(cin, s)` 是适用于 string 的整行读取函数；基础语法中的 `cin.getline(chars,capacity)` 则写入原生字符数组。混用数字输入与 getline 时，仍要先消耗前一行余下内容；例如使用本部分输入细节中的 get 循环，再调用 getline。空字符串是合法对象，但不能读取它的 `front()`、`back()` 或某个已有字符。

1.2 string 操作的参数与返回值

下面 n 表示字符串长度，k 表示这次处理的字符数。拼接、插入和删除会移动或复制字符，不能把所有操作都当成常数时间。

操作	含义与返回值	操作结果
`s.size()`、`s.empty()`	长度、是否为空，$O(1)$	长度类型是无符号的 `size_type`
`s[i]`、`s.at(i)`	访问一个字符	访问已有字符要求 `i<size()`；`at` 越界抛异常
`s += t`、`s.push_back(ch)`	尾部拼接、追加一个字符	扩容可能复制已有字符，旧访问位置可能失效
`s.substr(pos,len)`	从 pos 开始取最多 len 个字符，返回独立字符串	第二参数是长度；pos 可以等于 size，不能超过 size
`s.find(t,pos)`	从 pos 开始找第一次匹配，返回起点	找不到返回 `string::npos`；查找会逐步比较字符
`s.insert(pos,t)`	在 pos 前插入 t	尾部 pos=size 也可以插入
`s.erase(pos,len)`	删除从 pos 开始的最多 len 个字符	不是删除到下标 len；后续字符左移
`s.replace(pos,len,t)`	用 t 替换指定片段	t 的长度可与被删片段长度不同
`s.clear()`	清空字符	不承诺释放所有已分配容量
`s.c_str()`	获取以空字符结尾的 `const char*`	不拥有副本；不要通过它修改字符或在原对象销毁后使用

at 失败时抛出 `<stdexcept>` 中的标准异常 out_of_range，表示访问位置超出范围；可按已学的 try/catch 方式捕获 `const std::out_of_range& error`，`error.what()` 提供错误说明。异常类别不同，匹配的 catch 类型也不同。

`npos` 表示“没有找到位置”。例如 find 返回 npos 时，程序输出未找到；返回普通位置时，再从该位置截取或访问字符。字符串比较采用字典序，例如 `string("10")<string("2")` 为真，因为第一个不同字符是 1 与 2；它比较的是字符顺序。

1.3 完整演示：查找后修改一个片段

题目原文（自拟）： 程序无需输入。从字符串 `banana` 开始，找第一次出现的 `na`，第一行输出其下标和截出的两个字符；再将这两个字符换成 `NA`，删除下标 4 开始的两个字符，在下标 2 前插入 `-`，每次修改后各输出一行。



推导： 第一次 `na` 位于下标 2，`substr(2,2)` 取的是下标 2、3。替换后为 `baNAna`；删除位置 4、5 后为 `baNA`；在 2 前插入横线，后面的 `NA` 右移，得到 `ba-NA`。每次修改都会影响后续下标，应按当前字符串解释位置。

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

int main() {
    string s = "banana";
    auto pos = s.find("na");
    if (pos == string::npos)
        return 0; // 未找到时不能拿 npos 当下标。
    cout << pos << ' ' << s.substr(pos, 2) << '\n';
    s.replace(pos, 2, "NA"); // 第二参数是被替换的长度。
    cout << s << '\n';
    s.erase(4, 2); // 删除下标 4 开始的两个字符。
    cout << s << '\n';
    s.insert(2, "-"); // 在当前位置 2 之前插入，后缀整体右移。
    cout << s << '\n';
}
```

输出：

```text
2 na
baNAna
baNA
ba-NA
```

查找会扫描字符，截取会复制结果，修改片段可能移动后缀。处理长度不断增长的字符串时，反复向开头插入可能累计到平方量级。字符遍历若需要修改，用 `for(char &ch:s)`；只读取可用 `char ch`。

练习题： 洛谷 P5733 – 自动修正；洛谷 P1308 – 统计单词数（进阶，注意完整单词和大小写规则）。

2. vector：连续动态数组

2.1 连续动态数组与初始化

`vector<T>` 保存同类型元素，头文件为 `<vector>`。除 `vector<bool>` 的特殊表示外，元素连续存储，支持 $O(1)$ 下标访问。长度可在运行时改变，容器负责申请、搬移和释放存储，适合比赛中的数组、序列和邻接表。

begin() 表示第一项的位置，end() 表示最后一项之后的边界；它们是迭代器，`*位置` 才读取元素。`[begin(), end())` 表示整个半开区间，空数组时两个位置相等。对 vector，`begin()+k` 表示从起点向后移动 k 项；只有已有元素位置才能读取，end 不能读取。下面先用这些位置复制元素，迭代器小节再统一说明它们的操作。

下面是初始化片段；在函数中使用，还需 `<vector>` 和 `using namespace std;`。

```cpp
vector<int> a;                                  // 空数组，没有可以访问的元素。
vector<int> b(3);                               // 三个已有的 int，值都是 0。
vector<int> c(3, 7);                            // 三个 7，结果为 [7,7,7]。
vector<int> d{3, 7};                            // 两个元素 3、7，花括号是元素列表。
vector<int> e(d.begin(), d.end());              // 复制 d 的整个半开区间。
vector<vector<int>> grid(2, vector<int>(3, 0)); // 2 行，每行 3 个元素。
```

`vector<int>(3,7)` 与 `vector<int>{3,7}` 的长度不同。二维 vector 的外层保存行容器，每一行分别保存自己的元素，因此各行可以有不同长度；单行元素连续存储，不同行的元素区则分别管理。拷贝一个含 n 项的 vector 通常需 $O(n)$ 时间和独立存储；只读参数优先用 `const vector<int>&`。

2.2 用引用读入运行时长度的数组

现在已经知道 vector(n) 会建立 n 个元素，可以把引用小节的 `auto&` 输入方式用于它。

题目原文（自拟）： 输入 $0 \le n \le 100000$，接着输入 n 个 int 范围内的整数，用 vector 保存，再按输入顺序输出一行，元素间一个空格；n=0 时输出空行。

分析： `vector<int> a(n)` 先建立 n 个已有元素，初值为 0；输入循环中的 `i` 类型为 `int&`，每轮绑定到下一项。`cin >> i` 会把数据写入这一项。若写成 `auto i`，虽然也读掉了输入，a 中的元素仍是原来的 0。范围 for 会按已有长度遍历，不会自动增加元素；空的 `vector<int> a;` 不会进入循环。这里把引用小节在原生数组上的范围 for 用于已经创建的 vector 元素。



范围 for 的机制与原生数组相同；图以 n=3、输入 `8 5 9` 为例。绿色位置是本轮 i 的绑定目标：每轮修改一个真实元素，下一轮再绑定到下一项；i 表示值的别名，不表示下标 0、1、2。

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

int main() {
    int n;
    cin >> n;
    vector<int> a(n); // 先创建 n 个元素，范围 for 才能遍历它们。
    for (auto &i : a) {
        cin >> i; // i 是当前元素的引用，输入直接写回 a。
    }
    bool first = true;
    for (const auto &i : a) {
        if (!first)
            cout << ' ';
        cout << i; // 只读取元素，用 const 引用表达只读意图。
        first = false;
    }
    cout << '\n';
}
```

输入：

```text
3
8 5 9
```

输出：

```text
1  8 5 9
```

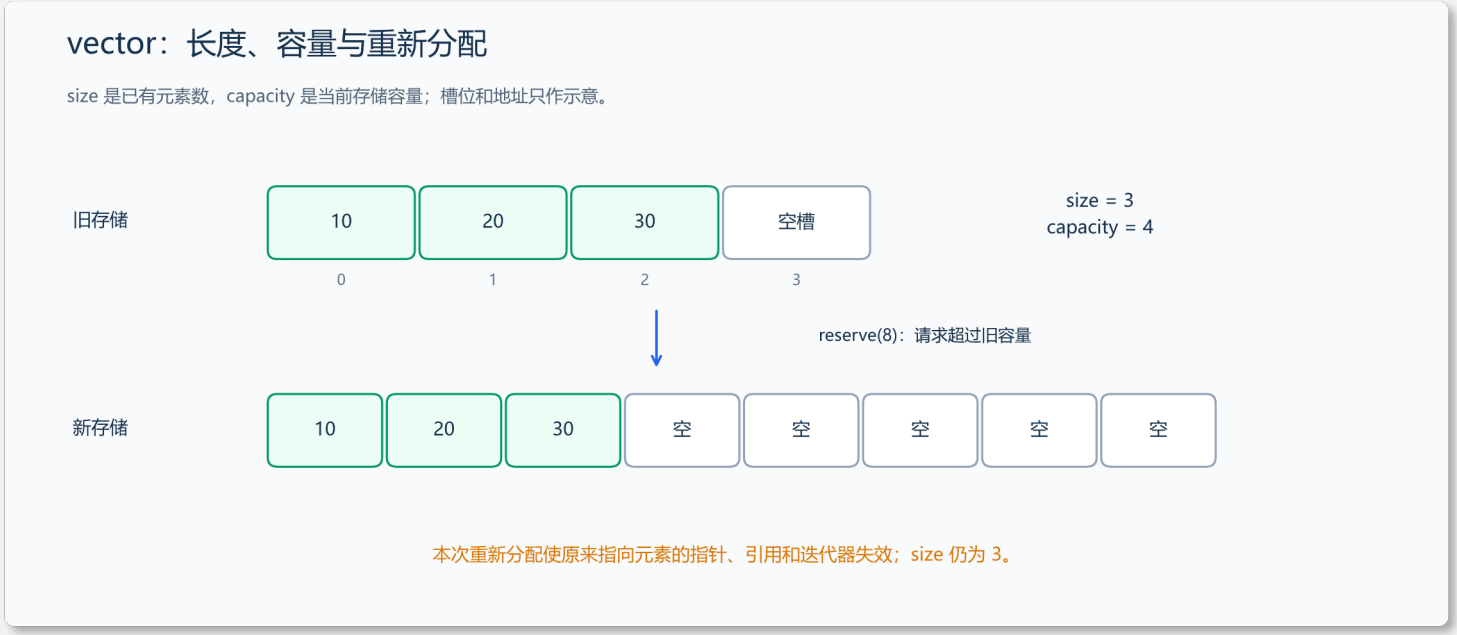
两次遍历共 $O(n)$ 时间，容器占 $O(n)$ 空间。对较小的 int，只读时按值遍历也很方便；对 string 等较大对象，`const auto&` 可以避免每轮复制。不要在这个范围 for 里同时 push_back 或 erase，让遍历依赖的位置失效；这些位置失效的规则在本节后面展开。

2.3 size、capacity、reserve 与 resize

size 是已有元素数，capacity 是当前存储可以容纳的元素数。 只有 `0…size-1` 是已有元素的位置。`reserve(k)` 请求容量至少为 k，不创建新元素；`resize(k,value)` 才会改变长度，增长部分填 value，缩小时删去尾部。



图从空数组开始预留 4 格，size 仍为 0；`resize(3,7)` 后才有三个 7；追加 9 后有四项；缩到 2 后只剩两个 7。图中的容量用于示意，实际 capacity 可以大于请求值，不能依赖每次扩容都恰好翻倍。



这张图的 `reserve(8)` 超过旧容量 4，因此发生重新分配。`clear()` 删除全部元素，但不降低 capacity；`shrink_to_fit()` 向实现请求缩减容量，实际容量可在调用后用 capacity 查看。容量、长度及重新分配规则见 C++17 草案：vector.capacity。

2.4 vector 常用操作的时间复杂度

以下 n 为当前元素数，元素复制、移动按常数代价估算。

操作	含义	时间复杂度
`v.size()`、`empty()`、`v[i]`、`at(i)`	状态或单项访问	$O(1)$；at 检查越界
`front()`、`back()`	访问首尾	$O(1)$，要求非空
`push_back(x)`、`emplace_back(args...)`	尾部追加；后者直接构造元素	均摊 $O(1)$，扩容的一次可为 $O(n)$
`pop_back()`	删除最后一项，没有返回值	$O(1)$，要求非空
`insert(it,x)`	在位置 it 前插入	中间通常 $O(n)$，需要腾出位置
`erase(it)`、`erase(first,last)`	删除一项或一个半开区间	中间通常 $O(n)$，后缀要左移
`resize(k)`、`assign(k,x)`	调整长度、重设内容	按实际新增、销毁或赋值数量计算
`clear()`	删除所有元素	$O(n)$，容量保留
`swap(other)`	交换两个 vector 的内容与容量	默认分配器下 $O(1)$，不逐项复制

“均摊”看的是一串操作的总费用：多次尾插合计为线性量级，平均每次常数；不是说每一次尾插都固定耗时。若每次都先 `reserve(size()+1)`，可能破坏正常的增长策略，反复搬移已有元素；知道总量时一次预留即可。



`[10,20,30,40]` 在下标 1 前插入 15，得到 `[10,15,20,30,40]`；再删 `[2,4)`，删掉的是 20、30，40 左移，得到 `[10,15,40]`。位置参数是迭代器，分别写成 `begin()+1` 和 `begin()+2`、`begin()+4`，不能把元素值当位置。

2.5 完整例题：先压紧，再用区间位置删除尾部

题目原文（自拟）： 第一行输入 `0≤n≤100000` 和 int 范围内的 x，随后输入 n 个 int。删除所有等于 x 的元素，其余保持输入顺序。输出一行剩余数量，下一行输出剩余元素，以单个空格分隔；没有剩余元素时第二行为空。

分析： 每次删除一项都会搬移后缀，不如先不改变长度，只在数组内部写出保留结果。read 逐项检查原数据，kept 表示已保留的数量，也就是下一次写入位置。遇到 x 跳过；遇到其他值，写入 kept 后将它加一。kept 始终不超过 read，因此只会写到当前或已经读过的位置，不会覆盖未来尚未检查的数据。



样例 `[3,1,3,2,3]` 删除 3：跳过下标 0，在读到 1 时写到下标 0，在读到 2 时写到下标 1。扫描结束后 kept=2，结果恰好位于半开区间 `[begin(),begin()+2)`。此时 size 仍为 5；`begin()+kept` 是结果之后的边界，用 `erase(begin()+kept,end())` 一次删掉尾部，才真正改变长度。这就是迭代器区间的用途：把“保留了几项”转成“要删除哪个区间”。

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

int main() {
    int n, x;
    cin >> n >> x;
    vector<int> values(n);
    for (int &value : values)
        cin >> value; // 引用读入已经创建的元素。
    int kept = 0;
    for (int read = 0; read < n; ++read) {
        if (values[read] != x) {
            values[kept] = values[read]; // 只写到当前或已检查的位置。
            ++kept;                      // 下一个保留元素将写到这里。
        }
    }
    values.erase(values.begin() + kept, values.end()); // 一次删除无用尾部；空区间也合法。
```

```cpp
    cout << values.size() << '\n';
    for (int i = 0; i < kept; ++i) {
        if (i > 0)
            cout << ' ';
        cout << values[i]; // 只输出有效前缀。
    }
    cout << '\n';
}
```

输入：

```text
5 3
3 1 3 2 3
```

输出：

```text
2
1 2
```

扫描和一次删除合计 $O(n)$ 时间，保存输入占 $O(n)$ 空间，筛选额外用 $O(1)$。n=0 时 begin 与 end 相等；全部删除时 kept=0；一个也不删时 kept=n，删除区间为空。算法库小节会在相同过程上解释 remove，先把这次读写想清楚。

2.6 vector 修改与迭代器失效

重新分配会使全部旧迭代器、指针、引用失效；插入未重新分配时，插入点之前的访问仍有效，插入点及之后的旧位置不能继续依赖；erase 会使删除点及之后的迭代器、引用失效。尾插即使没有扩容，旧 end 也不能继续当新 end。需要位置时可以先存下标，修改后重新获取迭代器，但插入删除仍可能改变这个下标代表的元素。见 C++17 草案：vector.modifiers。

`vector<bool>` 为节省空间可采用位压缩，单项访问可能是代理对象，不能套用普通 `bool&` 与 `bool*` 的假设。需要普通布尔标记数组时，比赛中也常用 `vector<char>`。

2.7 迭代器的用处：把同一个区间交给不同操作

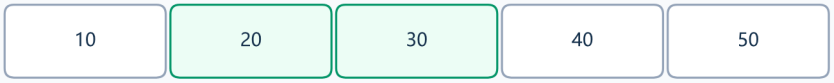
题目原文（自拟）： 程序无需输入。vector 初始为 `10、20、30、40、50`。复制下标半开区间 `[1,3)` 到新数组，然后从原数组删除这段区间。依次输出复制结果、原数组剩余结果和删除操作返回位置的元素，各占一行。

推导： begin()+1 指 20，begin()+3 指 40；区间包含 20、30，不包含 40。两个位置既可交给 vector 的区间构造，也可交给 erase。复制得到独立存储，不删除原数据；随后 erase 搬移后缀，旧 first、last 都不能继续使用，应接收它返回的新位置。本例返回指向左移后的 40。

同一对迭代器：先复制，再删除

vector=[10,20,30,40,50]；first=begin()+1，last=begin()+3。

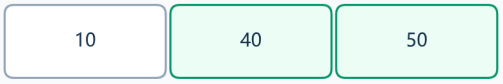
1. first 指 20，last 指 40；半开区间只包含 20、30



2. middle(first,last) 把这两项复制到独立存储，原数组不变



3. erase(first,last)：删除原数组的 20、30，后缀 40、50 左移



4. 旧 first、last 已失效；接收返回值 nextPosition，它现在指 40



先复制用有效旧位置；修改后只使用新获取或返回的位置，end 永远不能读取。

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

void print(const vector<int> &values) {
    bool first = true;
    for (int value : values) {
        if (!first)
            cout << ' ';
        cout << value;
        first = false;
    }
    cout << '\n';
}

int main() {
    vector<int> values{10, 20, 30, 40, 50};
    auto first = values.begin() + 1;
    auto last = values.begin() + 3;                // last 是边界，40 不属于 [first,last)。
    vector<int> middle(first, last);               // 复制到独立数组；values 此时没有改变。
    auto nextPosition = values.erase(first, last); // 接收删除后的有效位置。
    print(middle);
    print(values);
    if (nextPosition != values.end())
        cout << *nextPosition << '\n'; // 本例输出 40；若返回 end 则不能读取。
}
```

输出：

```text
20 30
10 40 50
40
```

复制 k 项需 $O(k)$ 时间和独立空间；删除还要搬移后缀，本例操作及输出总计 $O(n)$ 时间、额外 $O(k)$ 空间。vector 的 `it+k` 是常数时间的位置跳转；后面 list 的节点迭代器不能这样跳。删除后，从 erase 的返回位置继续处理即可。

练习题： 洛谷 P1427 – 小鱼的数字游戏（读到题面规定的 0，逆序输出）；洛谷 P5732 – 杨辉三角（二维数组，先推导相邻两项的关系）。

3. pair：成对数据

3.1 pair 的两个字段与构造

`pair<A,B>` 位于 `<utility>`，它是一项成对数据，不是容器。first 与 second 可以不同类型，例如 `pair<int,string>` 保存编号与名字；`vector<pair<int,int>>` 中的每项可以是坐标。后面的 map 会用这种成对记录保存键和值。

以下是初始化和拆分片段，需 `<utility>`、`<string>` 和 `using namespace std;`。

```cpp
pair<int, string> a;                    // int 初始化为 0，string 为空。
pair<int, string> b{2, "banana"};       // 直接给出两个成员。
auto c = make_pair(3, string("pear")); // 根据实参推导两个类型。
auto [id, name] = b;                    // C++17 结构化绑定得到副本，修改它们不改 b。
auto &[storedId, storedName] = b;       // 引用绑定可以写回 b 的对应成员。
```

`p.first`、`p.second` 访问成员，`swap(p,q)` 交换两项。对普通整数或字符串字段，`<` 按字典序：先比较 first，相同再比较 second；`==` 要求两个成员都相等。不能说所有比较符号都是“只看第一个字段”。

3.2 完整演示：pair 的字典序比较

题目原文（自拟）： 程序无需输入。建立 a=`(2,8)`、b=`(1,9)`、c=`(2,3)`、d=`(1,9)`，依次输出 `a<b`、`a<c`、`b==d` 的判断结果，一项一行；false 输出 0，true 输出 1。

推导： a、b 的第一字段不同，2 不小于 1，`a<b` 为 false，不再比较第二字段。a、c 的第一字段相同，转而比较 8、3，`a<c` 也为 false。b、d 两个字段都相同，`b==d` 为 true。把先决字段、下一字段和停止条件拆开，就能读懂字典序。




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

int main() {
    pair<int, int> a{2, 8}, b{1, 9}, c{2, 3}, d{1, 9};
    cout << (a < b) << '\n';  // first 不同，由 2<1 直接决定为 false。
    cout << (a < c) << '\n';  // first 相同，再判断 second: 8<3 为 false。
    cout << (b == d) << '\n'; // 两个字段都相同，结果为 true。
}
```

输出：

```text
0
0
1
```

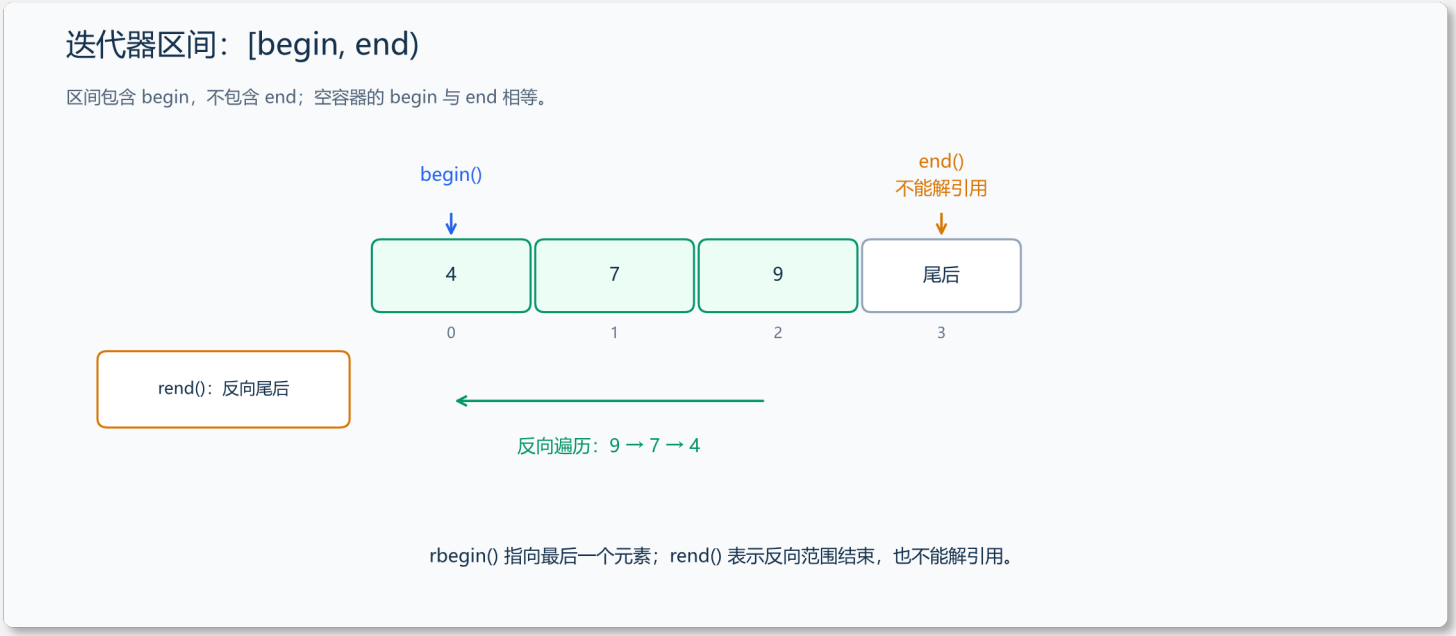
本例字段为 int，每次比较为 $O(1)$，额外空间 $O(1)$；字符串字段还要计入字符比较代价。算法库会把相同的比较规则用于排序，也会展示怎样指定第一字段降序、第二字段升序。pair 自身没有“去重”动作。

练习题：洛谷 P1093 — 奖学金（多个排序字段，可以继续学习 tuple 或 struct；按题面完整处理平局）。

4. 迭代器：访问位置与区间

4.1 位置、元素值与半开区间

迭代器表示容器中的一个访问位置，`*it` 才是该位置的元素；`it->first` 等价于 `(*it).first`。`begin()` 指向第一项，`end()` 表示尾后位置；空容器中二者相等。end 可以比较、作为区间边界，不能解引用。



`[first,last)` 包含 first，不包含 last，遍历条件是 `it≠last`。反向区间为 `[rbegin(),rend())`，同样不能读取 rend。反向迭代器的 `++it` 是沿反向序列前进，不是沿原下标增大。

4.2 迭代器的移动、比较与距离运算

先用已经学过的 vector 判断：`++it` 移到下一项，`--it` 移到上一项，`it+k` 跳过 k 项，同一 vector 中的合法位置可以相减。begin 不能再向前，end 可以作边界，不能读取它。

`<iterator>` 提供 next、prev 和 distance；`next(it,k)` 返回向前移动 k 项后的新位置，`prev(it,k)` 返回向后移动后的新位置，distance 求区间长度。对 vector，它们可利用随机访问；其他容器的能力会在学完各容器后统一对照。

4.3 值拷贝、引用与只读遍历

下面是函数片段，需 `<vector>`、`<iostream>` 和 `using namespace std;`。

```cpp
void changeValues(vector<int> &v) {
    for (int x : v)
        x += 1; // x 是副本，这一句不会修改 v。
    for (int &x : v)
        x += 1; // 引用指向原元素，每项真正增加 1。
    for (auto it = v.cbegin(); it != v.cend(); ++it) {
        cout << *it << ' '; // const_iterator 可以读取，不能通过它修改值。
    }
}
```

只读遍历大对象常用 `const auto&`，避免逐项拷贝。对 vector 中的值，const 迭代器只允许读取；它不承诺后续插入或删除后位置仍有效。

4.4 完整演示：遍历中安全删除

题目原文（自拟）： 程序无需输入。用 vector 保存 `1、2、3、4`，按从前到后的顺序删除偶数，输出剩余元素。删除后还要继续检查紧邻的下一项，不能漏项或使用已删除的位置。



推导： 在 2 处调用 erase 后，这个位置已失效，返回值指向 3。下一轮应直接检查 3；若再多加一次 `++it` 就会跳过它。在 4 处删除后返回 end，循环结束。

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

int main() {
```

```cpp
    vector<int> values{1, 2, 3, 4};
    for (auto it = values.begin(); it != values.end();) {
        if (*it % 2 == 0) {
            it = values.erase(it); // 删除时接收下一位置，此分支不再 ++。
        } else {
            ++it; // 保留当前项，才移动到下一位置。
        }
    }
    bool first = true;
    for (int value : values) {
        if (!first)
            cout << ' ';
        cout << value;
        first = false;
    }
    cout << '\n';
}
```

输出：

```text
1 3
```

vector 每次删除都可能搬移后缀，逐项删除最坏需要 $O(n^2)$ 时间。若目标是删掉全部匹配值，手动压紧后统一 erase 可以减少搬移；这里逐项删除，是为了观察返回位置怎样帮助遍历继续。范围 for 会自行维护遍历位置，删除导致位置失效时就难以继续，因此这个过程使用显式迭代器。

练习题： 洛谷 P1059 - 明明的随机数（用迭代器观察排序和压紧后的有效区间，最后统一删除尾部）。

5. <algorithm>：通过迭代器操作区间

5.1 算法处理区间，容器提供位置

STL 算法通常接收迭代器区间，而不是整个容器对象。算法是否可用，要看迭代器能力：本节先在已学的 vector 和 string 上用 sort，它需要随机访问迭代器。后面的链表和关联容器会说明为什么同一接口不能用于所有容器。

操作	作用	条件与复杂度
`min(a,b)`、`max(a,b)`	返回两项中的较小、较大值	同一模板通常要求一致的实参类型；按比较代价计算
`min({a,b,c})`、`max({a,b,c})`	对列表中的多个同类型值取最值	花括号提供候选列表，不是数组下标；需非空，按候选数计算
`sort(first,last)`	排序	$O(n\log n)$ 次比较；保留等价项顺序时用 `stable_sort`
`stable_sort(first,last)`	保留等价项相对顺序的排序	足够额外内存时 $O(n\log n)$ 次比较，否则至多 $O(n\log^2 n)$ 次比较
`reverse(first,last)`	原地逆序	$O(n)$，要求双向迭代器

操作	作用	条件与复杂度
`find(first,last,x)`	逐项找等于 x 的元素	最多 $O(n)$ 次比较，失败返回 last
`lower_bound(first,last,x)`	在升序区间找首个不小于 x 的位置	二分查找，找不到时返回 last
`upper_bound(first,last,x)`	在升序区间找首个大于 x 的位置	二分查找，找不到时返回 last
`unique(first,last)`	压紧相邻的重复项	$O(n)$；不改容器长度，不去掉不相邻重复
`remove(first,last,x)`	压紧不等于 x 的元素	$O(n)$；也需要 erase 才真正缩短序列
`min_element`、`max_element`	找最小、最大元素的位置	$O(n)$；空区间返回 last，不可解引用
`fill(first,last,x)`	把区间里的每项改为 x	$O(n)$，逐项赋值

求和 `accumulate` 位于 `<numeric>`。初值决定累计结果类型，例如整数总和可能很大时用 `0LL`，不能等计算完溢出的 int 后才转为 long long。

stable_sort 适合要求等价记录保持原先次序的场景，不能默认其额外空间为常数；比较次数与可用内存的关系见 C++17 草案：stable.sort。

5.2 完整例题：去重后统计等值边界

题目原文（自拟）： 输入 $0 \le n \le 100000$、int 范围内的查询值 x 和 n 个 int。将数据按升序排序，第一行输出不同值的数量；第二行输出这些不同值。第三行输出 x 在去重数组中的 lower_bound、upper_bound 下标，均从 0 开始；找不到不小于 x 或大于 x 的元素时，对应下标输出数组长度。

分析： 先排序使同值相邻；unique 留下每组第一项，返回新结尾；erase 删尾。lower_bound 和 upper_bound 针对当前去重数组查找，返回的是位置，不是元素值。



样例排序为 `[1,1,2,3,3]`，压紧后的有效前缀为 `[1,2,3]`。查询 2 时，首个不小于 2 在下标 1，首个大于 2 在下标 2。若想统计排序前数据中 2 的出现次数，应在删除重复项之前求上下界差。

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

int main() {
    int n, x;
    cin >> n >> x;
    vector<int> values(n);
    for (int &value : values)
        cin >> value;
    sort(values.begin(), values.end()); // unique 之前让相同值聚到一起。
    auto newEnd = unique(values.begin(), values.end());
    values.erase(newEnd, values.end()); // unique 自身不会减少 size。
    cout << values.size() << '\n';
    for (int i = 0; i < static_cast<int>(values.size()); ++i) {
        if (i > 0)
            cout << ' ';
        cout << values[i];
    }
    cout << '\n';
    auto left = lower_bound(values.begin(), values.end(), x);
    auto right = upper_bound(values.begin(), values.end(), x);
    cout << left - values.begin() << ' ' << right - values.begin() << '\n';
}
```

输入：

```text
5 2
3 1 3 2 1
```

输出：

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

排序主导时间 $O(n \log n)$，vector 中的每次边界查找为 $O(\log n)$。通用边界查找的比较次数可为对数级，但总费用还受迭代器移动能力影响；后面的 set、map 将使用成员查找接口。

5.3 自定义比较规则

对 vector<pair<int,int>> 直接调用 sort(begin,end) 时，会复用 pair 小节的字典序规则：先 first 升序，再 second 升序；重复项仍保留。若希望改变这个规则，再提供比较器。

比较器回答“a 是否排在 b 前面”。同一项不会排在自己前面，所以 `comp(a,a)` 为 false；`≤` 在两项相等时返回 true，与这个含义不同。排序和二分查找使用同一个顺序，就能按排序后的次序定位。下面函数使用 `<algorithm>`、`<utility>`、`<vector>`。lambda 参数中的 auto 会按实际元素分别推导；这里 a、b 都是 pair 的只读引用，第一项降序、第二项升序。

```cpp
void sortRecords(std::vector<std::pair<int, int>> &records) {
    std::sort(records.begin(), records.end(), [](const auto &a, const auto &b) {
        if (a.first != b.first)
            return a.first > b.first; // 第一项降序。
        return a.second < b.second;   // 相同第一项时，第二项升序。
    });
}
```

5.4 从手动压紧到 remove 与 erase

题目原文（自拟）： 第一行输入 `0≤n≤100000` 和 int 范围内的 x，随后输入 n 个 int。删除所有等于 x 的元素，其余保持输入顺序。输出一行剩余数量，下一行输出剩余元素，以单个空格分隔；没有剩余元素时第二行为空。

分析： vector 小节已经用读、写两个位置完成了同一任务。算法库 remove 封装了这次压紧：用 `remove` 一次扫描，把保留的元素按顺序压到前面，返回有效前缀的新结尾；再用一次区间 erase 真正缩短数组。

remove 与 erase 分工：保留前缀，再缩长度

输入 [3,1,3,2,3]，删除值 3；写位置 w 只在保留一个元素后增加。

1. 读下标 0 是 3，跳过；w=0



2. 读下标 1 是 1，写到下标 0；w=1



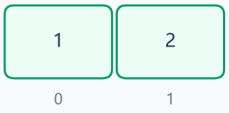
3. 跳过下标 2 的 3；保留下标 3 的 2，写到下标 1；w=2



4. remove 返回 begin()+2；size 仍为 5，尾部不应再当作结果



5. erase(newEnd,end())：真正删除尾部，size=2



问号表示结果区间外的值未指定；不能假定 remove 会清零尾部。

样例 `[3,1,3,2,3]` 删除 3，保留 1、2。`remove` 返回下标 2 对应的位置，但此时 size 仍为 5；尾部内容不作为结果，不承诺具体值。`erase(newEnd,end())` 后 size 才变为 2。

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

int main() {
    int n, x;
    cin >> n >> x;
    vector<int> values(n);
    for (int &value : values)
        cin >> value;                                   // 引用把输入写回已有元素。
    auto newEnd = remove(values.begin(), values.end(), x); // 只压紧，不改 size。
    values.erase(newEnd, values.end());                 // 删除整个无用尾部；空区间也合法。
    cout << values.size() << '\n';
    for (int i = 0; i < static_cast<int>(values.size()); ++i) {
        if (i > 0)
            cout << ' ';
        cout << values[i];
    }
    cout << '\n';
}
```

输入：

```text
5 3
3 1 3 2 3
```

输出：

```text
2
1 2
```

总时间 $O(n)$，保存输入占 $O(n)$ 空间，删除过程额外使用 $O(1)$。全部删除、一个也不删、n=0 都由同一段代码处理。

练习题： 洛谷 P1177 – 排序、洛谷 P1059 – 明明的随机数、洛谷 P2249 – 查找（原题找第一次出现且从 1 编号，不能先去重后直接交本节程序）。

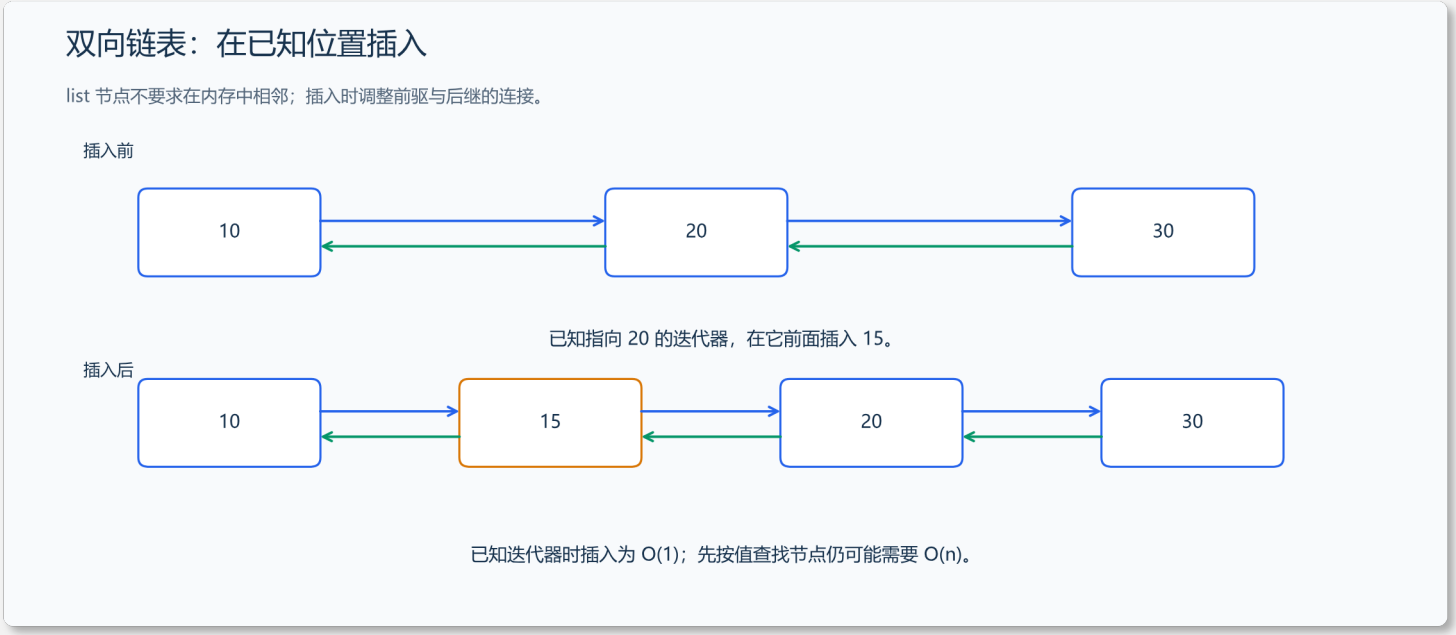
6. STL 头文件与包含方式

`#include <bits/stdc++.h>` 是 GNU libstdc++ 的实现扩展，在提供它的竞赛环境中很方便。通用 C++17 程序也可以按需包含 `<vector>`、`<string>`、`<algorithm>` 等标准头文件。它们只引入声明，不会替你创建容器；`using namespace std;` 只影响名字查找，也不会导入缺失的头文件。

7. list：双向链表

7.1 list 的节点存储与双向连接

`list<T>` 是双向链表，头文件为 `<list>`。每项在节点中保存，节点以连接关系形成顺序，元素不要求物理相邻。找到一个有效迭代器后，前后单项移动、插入或删除都为常数；但从开头找第 k 项要走 k 步。因此“频繁插入删除”还要问：是否已经知道位置。



插入只修改邻近连接，其他节点不用整体搬移；节点连接与分配也有额外内存代价。list 没有 `[]`，不能用 `begin()+k`，也不能调用通用 sort。

7.2 list 的插入、删除与节点移动

操作	含义	时间复杂度
`push_front/back`、`pop_front/back`	两端插入删除	单项 $O(1)$；pop 要求非空
`front()`、`back()`、`size()`、`empty()`	两端或状态访问	$O(1)$；读取两端要求非空
`insert(it,x)`	在 it 前插入，返回新节点位置	已有位置时 $O(1)$，it 可以是 end
`erase(it)`	删除该节点，返回下一位置	$O(1)$，it 不能是 end
`remove(x)`、`remove_if(pred)`	遍历并真正删除匹配项	$O(n)$；不是算法库 remove 的“仅压紧”
`sort()`	成员排序，保留等价项顺序	$O(n \log n)$ 次比较
`unique()`	真正删除相邻重复项	$O(n)$；全部去重通常先 sort
`splice(pos,other,it)`	将 other 中单个节点移到 pos 前	$O(1)$，默认分配器下可直接使用

`merge(other)` 可合并两张按同一规则排好序的 list，节点转移后 other 变空；它不会自动先排序。插入不使其他节点的迭代器、引用失效；删除只使被删节点相关访问失效。单节点 splice 的访问继续指向原节点，只是节点归属改变；其他 splice 重载的复杂度可能不同。见 C++17 草案: list.ops。

7.3 完整演示：定位、插入、删除和移动节点

题目原文（自拟）： 程序无需输入。A 初始为 `10、20、30`，B 初始为 `7、8`。在 A 的 20 前插入 15，删除 20，将 B 的第一个节点移到 A 的 30 前，最后分别输出 A、B 的元素，以空格分隔；再用保存的迭代器输出先前插入的 15 与移来的 7，验证它们仍指向原节点。



分析： 先用 find 确定 20 的位置；用 inserted 保存 insert 返回的 15 位置，同时原 target 仍指 20；erase 删除 20 后返回 30，改用这个返回位置。用 moved 保存 B 中 7 的位置，再 splice 到 A；moved 仍指同一个 7，但现在属于 A。图中移动的是节点，B 的 7 不再留在源列表。

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

void print(const list<int> &values) {
    bool first = true;
    for (int value : values) {
        if (!first)
            cout << ' ';
        cout << value;
        first = false;
    }
```

```cpp
    cout << '\n';
}

int main() {
    list<int> a{10, 20, 30}, b{7, 8};
    auto target = find(a.begin(), a.end(), 20); // 找位置本身要扫描。
    if (target == a.end())
        return 0;
    auto inserted = a.insert(target, 15); // 原 target 仍指向 20。
    auto nextPosition = a.erase(target);  // target 已失效，改用返回的 30 位置。
    auto moved = b.begin();               // 保存 7 的位置，之后不再从头寻找它。
    a.splice(nextPosition, b, moved);     // 转移一个节点，b 只剩 8。
    print(a);
    print(b);
    cout << *inserted << ' ' << *moved << '\n'; // 15、7 的位置仍有效；target 不再使用。
}
```

输出：

```text
10 15 7 30
8
15 7
```

定位和输出扫描节点，总时间 $O(n)$；已知位置的三个修改动作均为常数。保存位置的用处是把“找到节点”与“操作节点”分开：一次定位后，可以在它附近连续插入、删除或转移，不必每次重扫。target 所指节点被删除后，它的位置也随之失效；moved 所指节点转移到了 A，后续遍历从 A 中继续。做大量按值查找时，list 不会因为删除快就自动比 vector 或 map 更合适。

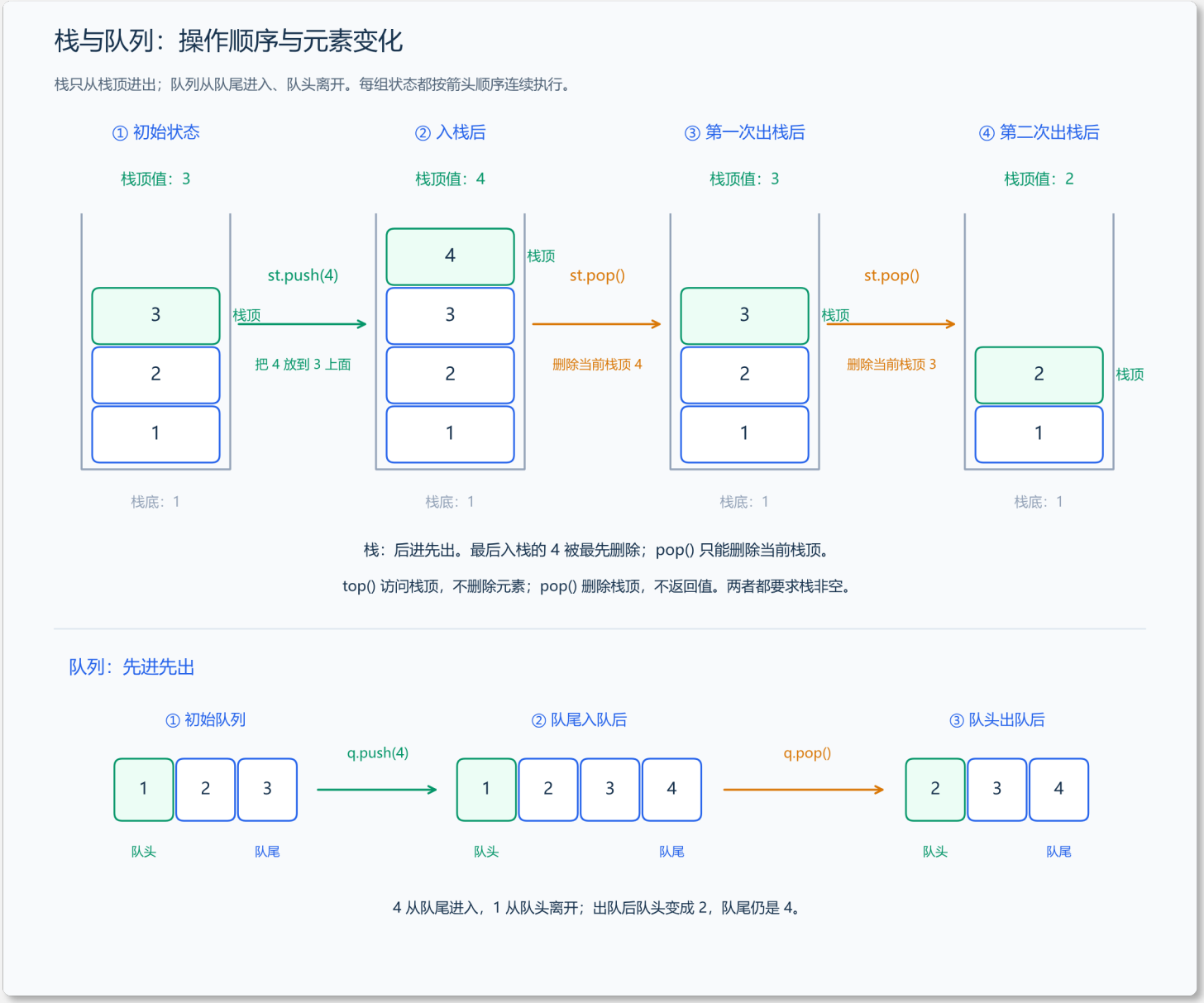
练习题： 洛谷 P1160 – 队列安排（按编号保存位置，可避免每次从头找；删除标记要防止重复删除）。

8. stack：栈

8.1 栈顶、栈底与后进先出

`stack<T>` 是容器适配器，头文件为 `<stack>`，默认用 deque 保存数据。适配器为底层容器提供受限接口：栈只允许操作栈顶，不允许下标或普通迭代器遍历。最后放入的元素最先移走，这叫后进先出（LIFO）。

可以把栈想成一叠盘子，栈顶是当前最上方的盘子：每次放入或取走，都在顶部进行。`push(x)` 把 x 放到顶部，`top()` 访问顶部，`pop()` 删除顶部。top 不会删除，pop 不返回值；要保存再删除，先读 top，再 pop。



栈的四个状态按顺序执行：初始顶为 3，压入 4 后顶为 4，第一次 pop 删除 4，第二次才删除 3。top 返回引用，非 const 栈可以修改顶部，但其他位置没有直接修改接口。若想遍历并保留栈，可以复制后逐项出栈，复制和扫描需 $O(n)$。

操作	功能	默认底层容器下的复杂度
`push(x)`、`emplace(args...)`	入栈或在顶部构造元素	$O(1)$
`top()`、`pop()`	读取或删除顶部	$O(1)$，通常配合 `empty` 判断后使用
`empty()`、`size()`	判空、数量	$O(1)$

8.2 完整例题：按出栈顺序输出

题目原文（自拟）： 输入 `0≤n≤100000`，随后 n 个 int 范围内的整数；按读入顺序入栈，输出全部出栈值，以单个空格分隔。n=0 时输出空行。

```
#include <iostream>
#include <stack>
```

```cpp
using namespace std;

int main() {
    int n;
    cin >> n;
    stack<int> pending;
    for (int i = 0; i < n; ++i) {
        int value;
        cin >> value;
        pending.push(value); // 后读入的数盖在前面的数上。
    }
    bool first = true;
    while (!pending.empty()) {
        if (!first)
            cout << ' ';
        cout << pending.top(); // 先访问，不会删除。
        pending.pop();         // 再删除，循环会清空整个栈。
        first = false;
    }
    cout << '\n';
}
```

输入：

```text
5
1 2 3 4 5
```

输出：

```text
5 4 3 2 1
```

每项入栈、出栈各一次，时间与最大存储量均为 $O(n)$。默认底层 deque 的尾部操作为常数时间；换成 vector 后，尾插偶尔会扩容并移动已有项，一连串尾插按均摊常数时间计算。

8.3 完整例题：用栈进行括号匹配

题目原文（自拟）： 输入一整行长度不超过 100000 的字符串，只含 `(`、`)`、`[`、`]`，可以为空。判断是否每个右括号都匹配到最近的未匹配左括号，且最终没有剩余左括号。满足输出 YES，否则输出 NO；空串满足要求。



推导： 对 `([])`，先压入左圆括号，再压入左方括号；读到右方括号时，栈顶恰好是左方括号，配对后弹出。接着读到右圆括号，栈顶已恢复为左圆括号，也能配对。对 `([)]`，读到右圆括号时栈顶却是左方括号，配对失败。栈中一直按出现顺序保存尚未匹配的左括号，最新的一项就在顶部。

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

bool balanced(const string &text) {
    stack<char> pending;
    for (char ch : text) {
        if (ch == '(' || ch == '[') {
            pending.push(ch); // 新的左括号比之前的更晚，应优先匹配。
        } else {
            if (pending.empty())
                return false; // 没有左括号，不能读 top。
            char expected = ch == ')' ? '(' : '[';
            if (pending.top() != expected)
                return false;
            pending.pop(); // 当前一对匹配完，恢复外层的待匹配状态。
        }
    }
    return pending.empty(); // 剩下的左括号没有右括号与它们匹配。
```

```cpp
}

int main() {
    string text;
    getline(cin, text); // 一整行输入，允许空串。
    cout << (balanced(text) ? "YES" : "NO") << '\n';
}
```

输入：

```text
([])
```

输出：

```text
YES
```

先配对内层 `[]`，再配对外层 `()`；输入 `([)]`、`]`、`((` 都输出 NO。总时间 $O(n)$，最多 n 个未匹配左括号，额外空间 $O(n)$。题目输入由这三种括号组成，每个字符都参与配对。

练习题： 洛谷 P1449 – 后缀表达式（遇运算符取栈顶两个操作数，注意减除的左右顺序）；洛谷 P1241 – 括号序列（进阶，原题还需给未配对括号补字符）。

9. queue：队列

9.1 队头、队尾与先进先出

`queue<T>` 位于 `<queue>`，也是容器适配器，默认底层为 deque。队尾接收新元素，队头移走最早进入的元素，这叫先进先出（FIFO）。它适合等待处理的任务和 BFS 的待扩展顶点。

`push(x)` 在尾部加 x；`front()`、`back()` 分别访问两端；`pop()` 删除队头，不返回值。这三种访问或删除要求队列非空。队列不提供 `[]`、迭代器或任意删除；遍历出队会清空内容，想保留时先复制。默认底层下，以上操作与 size、empty 均为 $O(1)$。

在前面栈与队列的图中，队列从 `[1,2,3]` 追加 4 后为 `[1,2,3,4]`，pop 删除 1，变成 `[2,3,4]`。队列中的先后取决于进入次序，与元素数值大小无关。

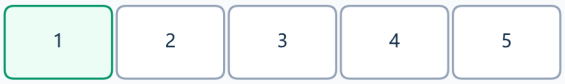
9.2 完整例题：循环报数出队

题目原文（自拟）： 输入 `1≤n≤100、1≤m≤1000`。编号 1…n 的人按顺序围成一圈，从队头开始报数；前 m−1 人各移动到队尾，第 m 人出圈，下一人重新从 1 报数。输出所有出圈编号，以单个空格分隔，直到队列为空。

循环报数：轮到的人从队头开始

n=5，m=3；从左到右为队头到队尾，每轮移动 m-1 次，再删除队头。

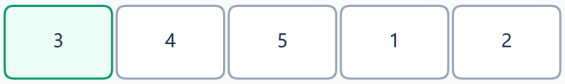
1. 初始队列；先把 1 从队头取出再放到队尾



2. 移动一次后；再把 2 移到队尾



3. 移动两次后：3 正好报到 3，输出并删除它



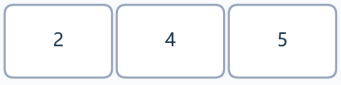
4. 剩 [4,5,1,2]；下一轮依次把 4、5 移到队尾



5. 队头变成 1，输出并删除它



6. 已输出 3、1；继续同样步骤，接着输出 5、2、4



每轮重新从当前队头报 1；pop 不返回元素，输出前先读 front。

分析： n=5、m=3 时，先将 1、2 依次移到尾部，队列为 `[3,4,5,1,2]`，删除 3；接着将 4、5 移到尾部，删除 1。队头始终就是下一轮报数的起点，不需要真的把末尾指向开头。

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

int main() {
    int n, m;
    cin >> n >> m;
    queue<int> pending;
    for (int id = 1; id <= n; ++id)
        pending.push(id);
    bool first = true;
    while (!pending.empty()) {
        for (int count = 1; count < m; ++count) {
            int id = pending.front(); // 先保存，pop 后不能依赖旧引用。
            pending.pop();
            pending.push(id); // 没有出圈，只移动到尾部。
        }
```

```cpp
        if (!first)
            cout << ' ';
        cout << pending.front(); // 当前是报到 m 的人。
        pending.pop();
        first = false;
    }
    cout << '\n';
}
```

输入：

```text
1  5 3
```

输出：

```text
1  3 1 5 2 4
```

每轮删一人，执行 m-1 次轮转，总时间 $O(nm)$，队列空间 $O(n)$。m=1 时不轮转，按原编号输出。m 特别大时这种逐次模拟就不够快，需要结合剩余人数取模或其他推导。

结构	下一个被处理的元素	典型用途
stack	最近进入者	括号匹配、递归的待返回状态
queue	最早进入者	分层搜索、按到达次序处理任务
priority_queue	当前比较规则下最优先者	最值调度、堆与最短路

练习题： 洛谷 P1996 - 约瑟夫问题、洛谷 P1540 - 机器翻译（先进先出的缓存淘汰，另需判断是否已缓存）。

10. map：有序键值映射

10.1 键、值与有序查找

`map<Key,Value>` 位于 `<map>`。每个键对应一个值，键按比较器有序且唯一；默认从小到大遍历键。可以把它想成“用名字查记录”的表，键用于定位，值保存统计结果或其他信息。常见实现使用平衡搜索树，让按键查找、插入和删除保持对数复杂度。

“键唯一”按比较规则判断：若 `comp(a,b)`、`comp(b,a)` 都为 false，说明两者在这个顺序中等价，map 将它们视为同一键。查询、按键插入删除和成员边界查找通常为 $O(\log n)$ 次比较；字符串键还要计入一次比较的字符代价。



图中只比较成绩时，同分的两条记录等价；加入编号的第二关键字，才能保存两条同分但不同编号的记录。这不仅影响排序，还影响“重复键”的判定。

10.2 查询与写入的区别

操作	返回或效果	关键区别
`mp.find(key)`	指向该键的迭代器，失败返回 `end`	不创建缺失键
`mp.count(key)`	`0` 或 `1`	判断键存在，不是读取 `value` 中的频次
`mp.at(key)`	已存在键的值引用	缺失抛 `out_of_range`，不插入
`mp[key]`	值引用；缺失时先创建默认值	`int` 初值为 `0`，`string` 初值为空
`mp.insert({key,value})`、`emplace(key,value)`	`(迭代器,是否插入)`	等价键已存在则不覆盖旧值
`mp.insert_or_assign(key,value)`	C++17：插入或更新	有键也会覆盖 `value`
`mp.try_emplace(key,args...)`	C++17：仅缺失时构造 `value`	已存在时保留 `value`
`mp.erase(key)`	删除数，`map` 为 `0` 或 `1`	删除键，而不是删除第 `key` 项
`mp.erase(it)`	删除迭代器所指项，返回下一位置	单位置删除为均摊常数
`lower_bound`、`upper_bound`	按键找边界	默认首个 ≥key、>key，不要求 key 已存在
`size()`、`empty()`、`clear()`	数量、判空、清空	前两者 $O(1)$，clear 为 $O(n)$



已有 `apple→3` 时，`find("pear")` 失败但大小仍为 1；读取 `mp["pear"]` 会建立 `pear→0`，大小变成 2；`insert({"apple",9})` 不改旧 3，`mp["apple"]=9` 才会更新。只想问是否存在时，不要用下标读取，尤其不能用 `mp[key]≠0` 代替“存在”：合法值可能就是 0。下标语义见 C++17 草案：map.access。

10.3 完整例题：统计后查询单词次数

题目原文（自拟）： 第一行输入 `0≤n,q≤100000`。接着 n 个长度 1…20 的小写英文单词，然后 q 个查询单词。先按字典序输出每个不同单词及出现次数，一项一行；再依查询顺序输出次数，未出现输出 0。查询不能创建新的单词记录。

分析： 统计时 `++frequency[word]` 正好需要“缺失时从 0 开始”；查询时用 find，成功读 second，失败直接输出 0。遍历中 `first` 是单词，`second` 是累计次数，`count(word)` 只有 0、1，不能代替次数。

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

int main() {
    int n, q;
    cin >> n >> q;
    map<string, int> frequency;
    for (int i = 0; i < n; ++i) {
        string word;
        cin >> word;
        ++frequency[word]; // 缺失单词创建为 0，再加到 1。
```

```cpp
    }
    for (const auto &[word, times] : frequency) {
        cout << word << ' ' << times << '\n'; // 按键字典序，不是出现先后。
    }
    while (q--) {
        string word;
        cin >> word;
        auto it = frequency.find(word);
        cout << (it == frequency.end() ? 0 : it->second) << '\n'; // 查询不插入。
    }
}
```

输入：

```text
5 2
pear apple pear banana apple
apple orange
```

输出：

```text
apple 2
banana 1
pear 2
2
0
```

设 u 为不同单词数，长度上限固定时，总时间 $O((n+q)\log(u+1)+u)$，存储 $O(u)$ 条记录。插入不使已有元素的迭代器、引用失效，删除只使被删项相关访问失效；不能通过迭代器改 first。需要修改值时可用 `auto &[key,value]`；`auto [key,value]` 是拷贝，改它不会写回。

10.4 迭代器的用处：定位一次，修改值或沿有序区间前进

题目原文（自拟）： 程序无需输入。map 中已有 `10→2、20→3、30→4、40→5`。找到键 20，将它的值加 7；输出键位于闭区间 `[15,35]` 内的记录，每项占一行。随后从这一区间中删除值为偶数的记录，再输出全部剩余记录；两个阶段之间输出一行 `--`。

分析： find 返回的不只是“找到了”，还是这个键值记录的位置，因此可直接修改 `it->second`，不用再次查找。键 first 参与有序组织，不能通过迭代器改它。lower_bound(15) 返回首个不小于 15 的键 20；upper_bound(35) 返回首个大于 35 的键 40。于是题意的闭区间 `[15,35]` 转成容器的半开迭代器区间 `[first,last)`，恰好覆盖 20、30。



继续删除怎样避免漏项？ 20 的值由 3 变成 10，是偶数；erase 返回 30，下一轮立即检查 30，再删除其值 4，返回 40。此时 `it==last`，停止。last 指向不在删除范围内的 40，因此这些删除不会使它失效；如果区间延伸到末尾，`last=end` 也作为有效边界。空区间则从开始就满足 `first==last`，不进入循环。

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

int main() {
    map<int, int> records{{10, 2}, {20, 3}, {30, 4}, {40, 5}};
    auto found = records.find(20);
    if (found != records.end())
        found->second += 7; // 修改已有值，不再使用下标二次查找；first 不能改。

    auto first = records.lower_bound(15); // 首个键 ≥15，即 20。
    auto last = records.upper_bound(35);  // 首个键 >35，即 40；40 不在区间中。
    for (auto it = first; it != last; ++it)
        cout << it->first << ' ' << it->second << '\n'; // ++ 按键顺序到下一条。
    cout << "--\n";

    for (auto it = first; it != last;) {
        if (it->second % 2 == 0)
            it = records.erase(it); // 当前位置已失效，接收下一位置，不再 ++。
        else
```

```cpp
            ++it; // 只在保留当前项时前进。
        }
        for (const auto &[key, value] : records)
            cout << key << ' ' << value << '\n'; // 输出未删除的 10、40。
    }
```

输出：

```text
20 10
30 4
--
10 2
40 5
```

设总记录数 n、选中记录数 k，find 和两次成员边界查找合计 $O(\log n)$，区间遍历与按位置删除合计 $O(k)$；最后输出全部剩余记录至多还要 $O(n)$，所以本程序总时间 $O(n+\log n)$，存储 $O(n)$、额外 $O(1)$。一般边界查询加区间处理为 $O(\log n+k)$，不能把访问 k 项也算成常数。map 迭代器支持 ++、--，没有 `it+3`；移动三项需三次前进，不能通过两位置相减求数量。边界与失效规则见 C++17 草案：关联容器要求。

练习题： CF 4C – Registration System、洛谷 P5266 – 学籍管理。

11. set：有序键集合

11.1 有序、不重复的键集合

`set<T>` 位于 `<set>`，只保存键，没有另外的 value。插入等价键时保留一项；默认升序遍历，`*it` 读取键。它适合动态去重、存在性查询和有序的前驱后继查找。与 map 一样，唯一性由严格弱序比较器决定，常见实现为平衡搜索树。

`set<int,greater<int>>` 按降序组织键，需要 `<functional>`。边界查找也随比较器改变，此时 lower_bound(x) 找首个“不排在 x 前”的键，即数值 ≤x，不能继续解释为 ≥x。

11.2 set 的插入、删除与定位

操作	功能	时间复杂度
`insert(x)`	返回 `(位置,是否新插入)`	$O(\log n)$
`find(x)`、`count(x)`	查存在；count 为 0 或 1	$O(\log n)$
`erase(x)`	删除等价键，返回 0 或 1	$O(\log n)$
`erase(it)`	删除一个有效位置，返回下一位置	均摊 $O(1)$
`lower_bound(x)`、`upper_bound(x)`	按比较规则找边界	$O(\log n)$
`begin()`、`rbegin()`	默认升序下定位最小、最大	$O(1)$，empty 可判断是否有元素
`size()`、`empty()`、`clear()`	数量、判空、清空	前两者 $O(1)$，clear 为 $O(n)$

set 没有下标，无法直接读“第 k 小”。`advance(it,k)` 会移动 k 次，反复用它查询排名可能很慢。要删除默认升序下闭区间 `[L,R]`，L≤R 时可用 `erase(lower_bound(L),upper_bound(R))`；用 find(L) 当左端会要求 L 恰好存在，若返回 end 而右端在前面，还可能构成非法区间。



11.3 完整例题：严格前驱与不小于查询值的后继

题目原文（自拟）： 第一行输入 `0≤n,q≤100000`，接着 n 个 int，再接着 q 个 int 查询。将数据作为升序去重集合；每次输出“小于 x 的最大值”和“不小于 x 的最小值”，中间一个空格。某侧不存在时输出 NA。查询不修改集合。

推导： lower_bound(x) 就是不小于 x 的首项；若它不是 begin，前一个位置便是严格前驱。x 比所有值大时 lower_bound 返回 end，仍可在非空集合中向前一步取最大值；空集合中 begin=end，两侧都不存在。

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

int main() {
    int n, q;
    cin >> n >> q;
    set<int> values;
    for (int i = 0; i < n; ++i) {
        int value;
        cin >> value;
        values.insert(value); // 重复输入只保存一次。
```

```cpp
    }
    while (q--) {
        int x;
        cin >> x;
        auto it = values.lower_bound(x);
        if (it == values.begin())
            cout << "NA";
        else
            cout << *prev(it); // end 可向前一步，但 begin 不能再向前。
        cout << ' ';
        if (it == values.end())
            cout << "NA";
        else
            cout << *it; // end 只作边界，不能解引用。
        cout << '\n';
    }
}
```

输入：

```text
4 3
10 20 30 40
25 10 50
```

输出：

```text
20 30
NA 10
40 NA
```

建集合与查询共 $O((n+q)\log(u+1))$，u 为不同值数，空间 $O(u)$。插入保留已有迭代器，删除只使被删项失效；通过迭代器直接改键会破坏排序，因此接口不允许这样写。

11.4 自定义记录的排序与唯一性

下面是类型片段，需 `<set>`、`<string>`。比较顺序为成绩降序、姓名升序、年龄升序；三项都等价时视为同一记录。struct 中的 `operator()` 定义“把对象当函数调用”时执行的操作，例如 `StudentLess compare; compare(a,b);`。末尾 const 表示比较不修改比较器对象；同一个元素的各字段都相等，因此与自身比较返回 false。

```cpp
struct Student {
    std::string name;
    int age;
    double score;
};

struct StudentLess {
    bool operator()(const Student &a, const Student &b) const {
        if (a.score != b.score)
            return a.score > b.score; // 成绩不同时按降序比较；相同时继续比较后面的字段。
        if (a.name != b.name)
            return a.name < b.name;
```

```cpp
        return a.age < b.age; // 继续比较第三字段，满足严格弱序。
    }
};
```

在函数中用 `std::set<Student,StudentLess> students;` 创建容器，再用 `students.insert({"Alice",18,90.0});` 插入记录。插入同分的 Bob 会保留，两次插入完全相同的 Alice 则只留一项。若希望同名同龄同分的两条记录也能共存，改用 multiset，或给每条记录增加独立编号。保存“名字→成绩”通常用 map；只保存唯一的名字用 set；保存全部学生再统一排序通常用 vector，按需求选结构。

练习题： 洛谷 P1059 – 明明的随机数、洛谷 P2234 – 营业额统计（进阶，边插入边找邻近值）。

12. unordered_map：哈希键值映射

12.1 哈希先定位桶，再比较键

`unordered_map<Key,Value>` 位于 `<unordered_map>`，保存唯一键到值的映射，按哈希值组织桶，遍历时依次访问桶中的元素。查找先由 hash 定位候选桶，再用键相等规则核对实际键；不同键可以发生哈希冲突，哈希值相等不等于键相等。



图用教学哈希 `h(x)=x`、桶下标 `h(x)%10` 说明冲突，12、22、32 同在桶 2；查 22 时比较 12 失败、22 成功，查 42 要检查全部候选。标准库不承诺采用取模 10 或这组实际桶编号。

查找、单项插入和按键删除平均为 $O(1)$，最坏可为 $O(n)$，还需计入哈希和比较键的代价。它适合不依赖键顺序的统计与查询；没有 lower_bound、upper_bound，也不能从 begin 得到数值最小键。

12.2 下标、负载与访问失效

find、count、at、下标、insert、emplace、insert_or_assign、try_emplace 的“查询、默认插入、是否覆盖”含义与 map 相同，只是组织方式和复杂度不同。清空为 $O(n)$，size、empty 为常数。

`load_factor()` 为元素数除以桶数；`max_load_factor(x)` 设负载上限，`reserve(k)` 按预计元素数安排桶，`rehash(b)` 请求桶数。它们不会创建 k 个键值对，也不能保证消除冲突。已知要插入很多记录时，可先设负载上限，再 reserve，减少增长过程中的重建。



发生 rehash 会使旧迭代器失效，但指向已有元素的引用、指针仍有效；删除只使被删项相关访问失效。插入若触发 rehash，就不能继续使用保存的旧迭代器。见 C++17 草案：无序关联容器要求。不要在遍历中不断插入新键，又默认旧迭代器一直有效。

12.3 完整例题：按查询顺序输出频次

题目原文（自拟）： 第一行输入 $0 \le n,q \le 100000$，接着 n 个 int，再接着 q 个 int 查询。每次输出这个数在前 n 项中的出现次数，没有则输出 0。只需按查询顺序输出，不要求遍历键或按键排序。

分析： 统计使用下标累加，查询使用 find。输出次序由输入查询决定，不依赖哈希容器遍历，因此答案稳定。reserve 只是容量规划，n=0 时表仍为空。

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

int main() {
    int n, q;
    cin >> n >> q;
    unordered_map<int, int> frequency;
```

```cpp
    frequency.max_load_factor(0.7f); // 先设置负载，再按预计元素数预留桶。
    frequency.reserve(n);
    for (int i = 0; i < n; ++i) {
        int value;
        cin >> value;
        ++frequency[value]; // 已存在则累加，不存在从 0 起。
    }
    while (q--) {
        int value;
        cin >> value;
        auto it = frequency.find(value);
        cout << (it == frequency.end() ? 0 : it->second) << '\n';
    }
}
```

输入：

```text
5 3
3 1 3 2 1
3 9 1
```

输出：

```text
2
0
2
```

整数键下，平均总时间 $O(n+q)$，若冲突集中，最坏可能退化到 $O((n+q)(u+1))$，u 是不同键数。存储计入元素与桶；代码按 n 预留，空间上界 $O(n+1)$。需要有序邻近查询、稳定的对数查找界或有序输出时，应考虑 map，不能只按“哈希一定更快”选择。

练习题： 洛谷 P5266 – 学籍管理（可以分别用 map 与 unordered_map 比较）；CF 4C – Registration System。

13. unordered_set：哈希键集合

13.1 unordered_set 的存在性查询

`unordered_set<T>` 位于 `<unordered_set>`，只保存唯一键，没有关联 value。insert 返回 `(位置,是否新插入)`，find 失败返回 end，count 为 0 或 1。平均单项插入、查找、按键删除为 $O(1)$，最坏 $O(n)$。需要统计次数时用 unordered_map，不能把集合的 count 当输入中的重复次数。

哈希集合按桶组织元素，遍历次序通常与输入顺序、数值大小都不同。它适合维护“已经见过哪些值”；若还需要首次出现顺序，可以把新值同时记入 vector。reserve、rehash 和访问失效规则与上一节一致。

13.2 完整例题：按首次出现顺序去重

题目原文（自拟）： 输入 $0 \le n \le 100000$ 和 n 个 int，按第一次出现的先后输出各个不同值，中间单个空格。n=0 时输出空行。不能按数值排序。



推导： [3,1,3,2,1] 的插入结果依次为成功、成功、重复、成功、重复；只有成功时追加到输出 vector，所以输出为 3 1 2。哈希集合用于判重，vector 用于保存题面要求的顺序。

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

int main() {
    int n;
    cin >> n;
    unordered_set<int> seen;
    seen.reserve(n);
    vector<int> order;
    for (int i = 0; i < n; ++i) {
        int value;
        cin >> value;
        auto [it, inserted] = seen.insert(value);
        if (inserted)
```

```cpp
            order.push_back(value); // 重复键不改变首次出现的顺序。
        }
        for (int i = 0; i < static_cast<int>(order.size()); ++i) {
            if (i > 0)
                cout << ' ';
            cout << order[i]; // 这里遍历有序记录，不是遍历 seen。
        }
        cout << '\n';
    }
```

输入：

```text
5
3 1 3 2 1
```

输出：

```text
3 1 2
```

保留的是首次出现顺序，不是数值顺序。平均时间 $O(n)$，最坏可能 $O(n^2)$，预留桶、判重集合和输出序列合计 $O(n)$ 空间。

13.3 自定义类型的相等与哈希要一致

例如学生按编号 id 判重，名字仅是附加信息。下面是类型片段，需 `<string>`、`<functional>`、`<unordered_set>`、`<cstddef>`。`operator==` 定义 Student 之间的相等判断。`std::hash<int>{}` 创建整数哈希函数对象，再用 `(student.id)` 调用它；返回的 std::size_t 是表示大小的无符号整数类型，声明由 `<cstddef>` 提供。

```cpp
struct Student {
    std::string name;
    int id;
    bool operator==(const Student &other) const {
        return id == other.id; // 同一编号就是同一键，即使名字不同。
    }
};

struct StudentHash {
    std::size_t operator()(const Student &student) const {
        return std::hash<int>{}(student.id); // 按用于判等的学号计算哈希值。
    }
};
```

在函数中创建 `std::unordered_set<Student,StudentHash> students;`，依次用 insert 插入 `{"Alice", 1}`、`{"Bob",2}`、`{"Ann",1}` 后只有两个键，编号 1 保留先插入的 Alice，不会自动换成 Ann。若 equality 只比较 id，hash 却还依赖 name，同 id 不同名可能被分到不同桶，破坏容器要求；相反，不同键得到相同 hash 是允许的，交给相等比较继续区分。

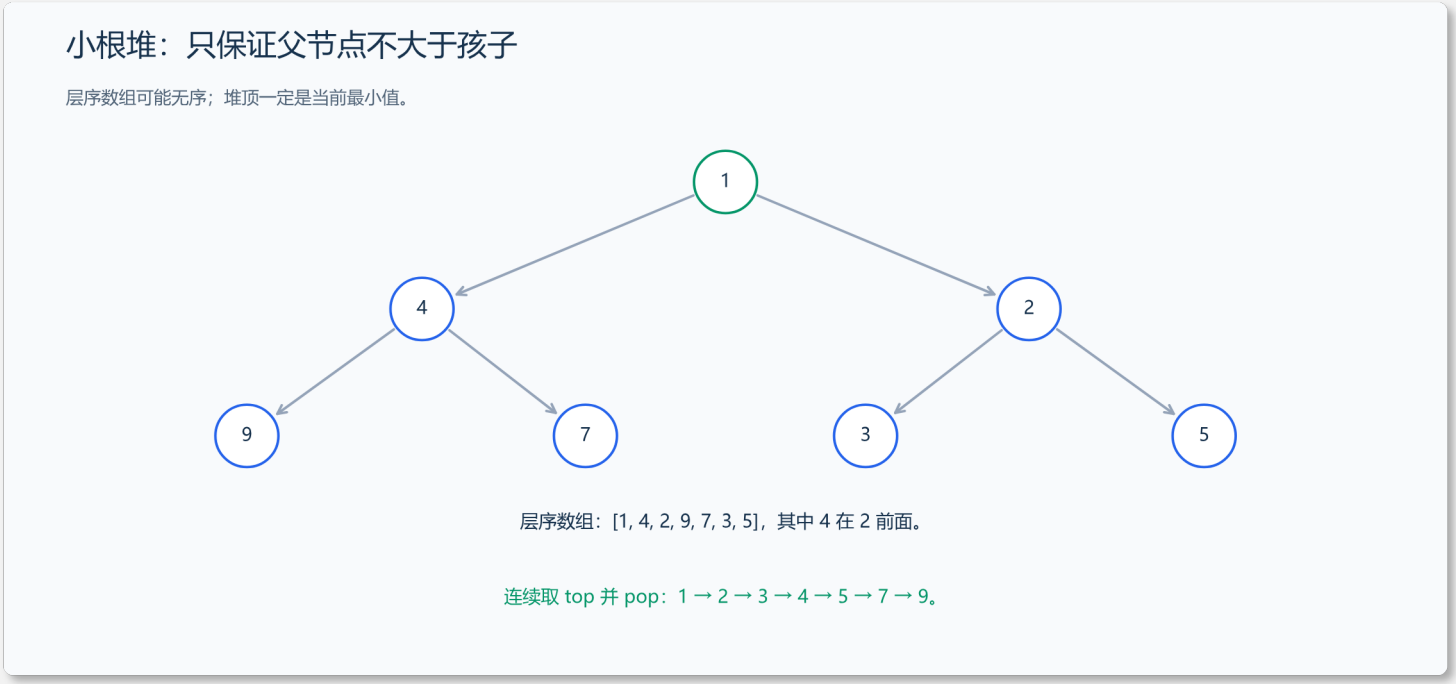
set 判断唯一性靠比较器的等价关系，unordered_set 靠相等规则；两者都不允许通过迭代器随意改键。只需判存在选集合，要携带并修改附加数据选映射，要重复保存同键选 multi 版本。

练习题： 洛谷 P3370 — 字符串哈希（可先用 `unordered_set<string>` 按完整字符串判重，再学习自行哈希时的碰撞问题）；洛谷 P1059 — 明明的随机数（原题需排序，判重后还要 sort，不能直接输出哈希遍历结果）。

14. priority_queue：优先队列

14.1 从堆顶读取优先级最高的一项

`priority_queue<T,Container,Compare>` 位于 `<queue>`，默认底层为 `vector<T>`、比较器为 `less<T>`。默认堆顶是最大值；取出后重新调整，保证下一次 top 仍是当前最大。底层序列只保持堆序，不能当成完整排序数组，也没有普通迭代器接口。



本图使用小根堆解释结构，`[1,4,2,9,7,3,5]` 中 4 在 2 前面也合法；只有根 1 保证是最小值。从 0 编号时，父位置为 `(i-1)/2`，孩子为 `2i+1`、`2i+2`。这描述常见二叉堆实现，不承诺标准库内部采用某个固定数组布局。



对小根堆 `[2,4,3,9,7,5]` 追加 1，先与父值 3 交换，再与根 2 交换，得到 `[1,4,2,9,7,5,3]`。只沿祖先链移动，层数为 $O(\log n)$。



从 `[1,4,2,9,7,3,5]` 删除根 1，末尾 5 补到根；先选较小孩子 2 交换，再与 3 交换，得到 `[2,4,3,9,7,5]`。选择错误孩子可能仍违反堆序，所以不能随便向一边下移。

操作	含义	默认 vector 底层下的费用
`top()`	当前最优先元素的 const 引用	$O(1)$；配合 empty 判断后读取
`push(x)`、`emplace(args ...)`	插入后向上调整	$O(\log n)$ 次堆比较；单次扩容还可能搬移 $O(n)$ 项
`pop()`	删除当前堆顶，不返回值	$O(\log n)$，要求非空
`size()`、`empty()`	数量、判空	$O(1)$

若要改变已有任务的优先级，标准接口没有直接定位修改的操作；可另存最新状态，插入新记录，取出过期记录时跳过，Dijkstra 章节就使用这个思路。不能修改比较器所依赖的外部状态，又默认旧堆已经自动重排。

14.2 大根堆、小根堆与比较器

`<functional>` 的 less、greater 是标准比较函数对象类型：`less<int>{}(a,b)` 判断 a<b，`greater<int>{}(a,b)` 判断 a>b。第三个模板参数填写比较器类型，容器创建后用它进行比较。以下是定义片段，需要 `<queue>`、`<vector>`、`<functional>`。

```cpp
std::priority_queue<int> maxHeap;                                  // less<int>，较大值优先。
std::priority_queue<int, std::vector<int>, std::greater<int>> minHeap; // 较小值优先。
```



对整数，less 使最大值在 top，greater 使最小值在 top。理解自定义规则时，可问“a 是否比 b 更晚处理”；它与 sort 中“a 排在 b 前面”的使用直觉不同。比较器按同一个先后关系安排元素，堆才能持续找到优先级最高的一项。接口与默认底层见 C++17 草案：priority.queue。

14.3 完整例题：优先级与平局顺序

题目原文（自拟）： 输入 $0 \le n \le 100000$，随后 n 行各含唯一编号 id 和 int 范围内的优先级 priority。按优先级从大到小取出；同优先级先取编号较小者。每次输出一行 id，直至全部取完。



分析： a 优先级更小时应更晚；优先级相同时，a 编号更大也应更晚。样例任务 $(1,3)、(2,5)、(3,5)$ 依次取 2、3、1，不能只按 priority 比较后期望自动按编号稳定排序。

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

struct Task {
    int id;
    int priority;
};

struct Later {
    bool operator()(const Task &a, const Task &b) const {
        if (a.priority != b.priority)
            return a.priority < b.priority; // 小者更晚。
        return a.id > b.id;                 // 同优先级，大编号更晚。
    }
};

int main() {
    int n;
```

```cpp
    cin >> n;
    priority_queue<Task, vector<Task>, Later> pending;
    for (int i = 0; i < n; ++i) {
        Task task;
        cin >> task.id >> task.priority;
        pending.push(task);
    }
    while (!pending.empty()) {
        cout << pending.top().id << '\n'; // 先读堆顶，不能修改其比较字段。
        pending.pop();
    }
}
```

输入：

```text
3
1 3
2 5
3 5
```

输出：

```text
2
3
1
```

共 n 次插入、删除，总时间 $O(n\log(n+1))$，空间 $O(n)$。若元素是成绩记录，可以同样按 score 降序、名字等字段处理平局；不要让相等项同时满足两个方向的比较。

14.4 完整例题：用小根堆保留最大的 k 项

题目原文（自拟）： 输入 $1\le k\le n\le 100000$ 和 n 个 int，保留按数值最大的 k 个元素，重复值按不同输入位置计入。最后按降序输出这 k 个值，以空格分隔。



推导： 保留集合中最小的一项是入选门槛。小根堆不足 k 项时直接放入；已经满时，若新值大于 top，删掉门槛并插入它，否则跳过。对 `5、1、9、3、9、2`、k=3，最终保留 `5、9、9`。图按升序列出入选值便于比较，不表示真实堆数组已排序。

```cpp
#include <functional>
#include <iostream>
#include <queue>
#include <vector>
using namespace std;

int main() {
    int n, k;
    cin >> n >> k;
    priority_queue<int, vector<int>, greater<int>> kept;
    for (int i = 0; i < n; ++i) {
        int value;
        cin >> value;
        if (static_cast<int>(kept.size()) < k) {
            kept.push(value); // 尚有名额，先保存。
```

```cpp
        } else if (value > kept.top()) {
            kept.pop(); // 移走入选集合中最小的那项。
            kept.push(value);
        }
    }
    vector<int> answer;
    while (!kept.empty()) {
        answer.push_back(kept.top()); // 小根堆弹出顺序为升序。
        kept.pop();
    }
    for (int i = static_cast<int>(answer.size()) - 1; i ≥ 0; --i) {
        if (i + 1 < static_cast<int>(answer.size()))
            cout << ' ';
        cout << answer[i]; // 反向输出，才符合题面的降序。
    }
    cout << '\n';
}
```

输入：

```text
6 3
5 1 9 3 9 2
```

输出：

```text
9 9 5
```

不变量是“处理前 i 项后，堆中保存其中最大的 min(i,k) 项”；新项不够门槛便不能改善集合，超过门槛则只需替换最小者。输出时另有 k 次弹堆，因 k≤n，总时间仍为 $O(n\log(k+1))$，空间 $O(k)$。重复值保留，负数也正常参与比较。

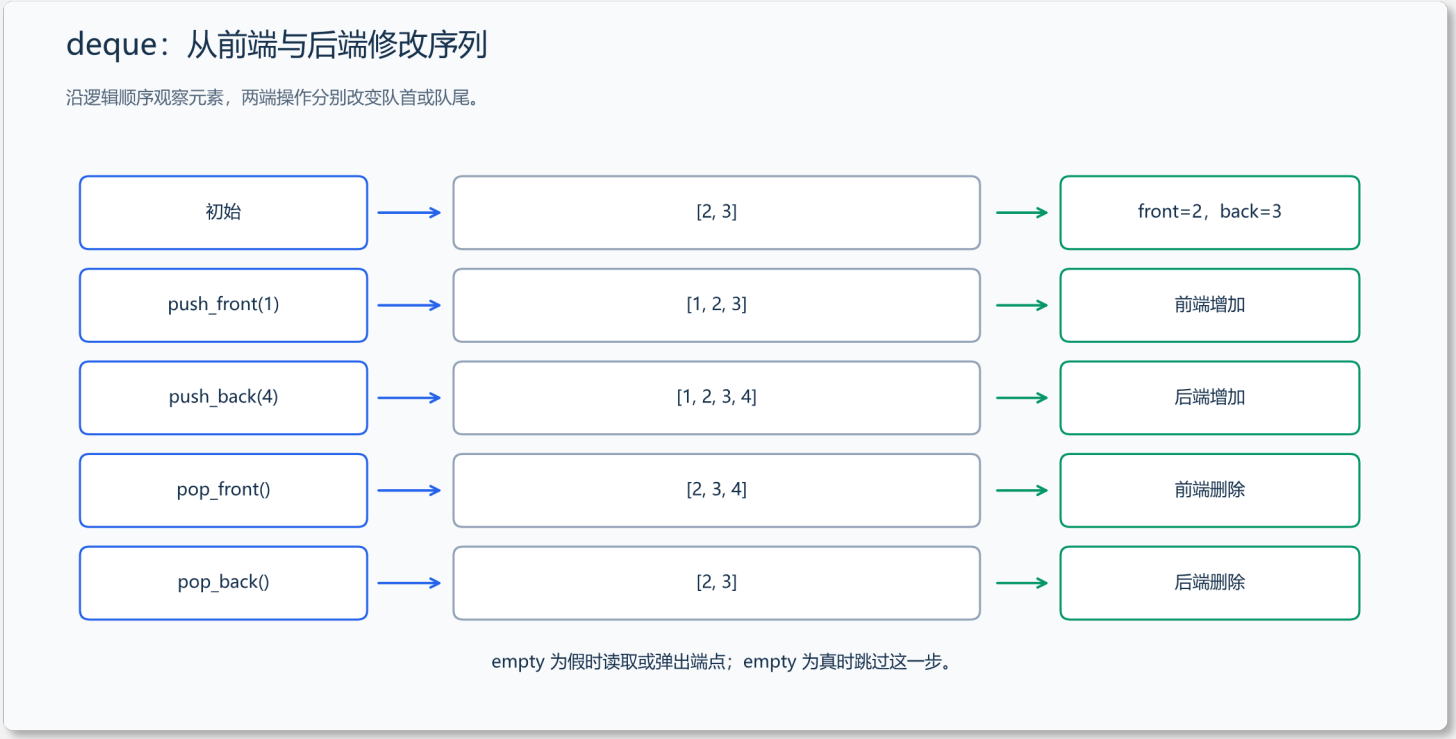
练习题： 洛谷 P3378 – 堆、洛谷 P1090 – 合并果子；洛谷 P1631 – 序列合并（进阶，堆中还需保存来源位置）。

15. deque：双端队列

15.1 双端操作与随机访问

`deque<T>` 位于 `<deque>`，是可以直接遍历和下标访问的双端队列，和只暴露队头、队尾接口的 queue 不同。两端单项插入、删除为常数级，随机访问为 $O(1)$，中间插入、删除通常为 $O(n)$。

deque 常用多个存储块组织元素，通过下标访问时由容器定位对应位置，因此 `dq[i]` 可以快速找到元素。整个序列没有像 vector 那样的连续存储要求，也没有 reserve、capacity 接口。



从 `[2,3]` 开始，先前插 1 得到 `[1,2,3]`，后插 4 得到 `[1,2,3,4]`，再前删、后删回到 `[2,3]`。push_front/back、pop_front/back 和 front/back 要分清“改变内容”与“只读取”，pop 与端点读取都要求非空。

两端插入会使旧迭代器失效，但已有元素的引用仍有效；中间插入会使全部旧迭代器、引用失效。pop_front 删除的不是最后一项时，只使被删项相关访问失效；删最后一项还使旧 end 失效。中间删除应重新获取访问位置，不能照搬 list 的规则。见 C++17 草案: deque.modifiers。

15.2 完整例题：滑动窗口最小值

题目原文（自拟）： 输入 $1 \le k \le n \le 100000$ 和 n 个 int。对每个连续长度为 k 的窗口输出最小值，从最左窗口向右移动，每次一步，输出一行 `n-k+1` 个整数，以空格分隔。

先分析普通做法： 每个窗口重新扫 k 项，时间为 $O(nk)$。窗口右移后，最左旧项过期，右端新项加入，deque 可以同时在两端处理候选，但还需要知道哪些候选以后已不可能成为最小值。

为什么可以从后面删除候选？ 若较早下标 j 的值不小于刚到的值，新的值更小或相等，而且过期更晚。以后只要旧 j 仍在窗口，新的下标也在窗口，旧 j 就没有独有优势，可以删除。于是 deque 保存下标，按时间递增、对应值严格递增；前端便是当前最小值。保存下标才能判定过期，不能只存数值。



图中 k=3，读到下标 2 的 −1 时，队尾值 3、1 都被更小的新值取代；读到下标 5 的 3 时，队尾值 5 被取代，但前面的 −3 仍保留；到下标 6 时，−3 的下标 3 已离开 `[4,6]`，从前端弹出这个过期下标后，新的队首对应 3，它就是当前窗口的最小值。

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

int main() {
```

```cpp
    int n, k;
    cin >> n >> k;
    vector<int> values(n);
    for (int &value : values)
        cin >> value;
    deque<int> candidates; // 保存下标，队内值严格递增。
    bool first = true;
    for (int i = 0; i < n; ++i) {
        while (!candidates.empty() && candidates.front() ≤ i - k) {
            candidates.pop_front(); // 当前窗口左端是 i-k+1，旧下标过期。
        }
        while (!candidates.empty() && values[candidates.back()] ≥ values[i]) {
            candidates.pop_back(); // 新值不大且过期更晚，旧候选没有优势。
        }
        candidates.push_back(i);
        if (i ≥ k - 1) {
            if (!first)
                cout << ' ';
            cout << values[candidates.front()]; // 当前最小值在前端。
            first = false;
        }
    }
    cout << '\n';
}
```

输入：

```text
8 3
1 3 -1 -3 5 3 6 7
```

输出：

```text
-1 -3 -3 -3 3 3
```

每个下标只加入一次，删除后不会重新加入，虽然有两个 while，总处理次数仍为 $O(n)$。候选队列最多 k 项，占 $O(k)$，本程序另保存整个输入数组，占 $O(n)$。k=1 时每个元素自己就是答案；重复值保留更晚出现的下标，不改变最小值。若改求最大值，队尾淘汰条件改为旧值 ≤ 新值，维护递减队列。

练习题： 洛谷 P1886 - 单调队列 / 滑动窗口（同时输出最小值和最大值）；洛谷 P1440 - 求 m 区间内的最小值（窗口是当前项之前的若干项，输出时机与本例不同）。

16. array：定长数组

`array<T,N>` 位于 `<array>`，长度 N 写在类型中，适合大小编译时已知的序列。它连续存储，支持下标、at、迭代器、size 与 empty，也能使用 sort、fill、reverse。它没有 push_back、resize 或 reserve，复制时复制全部 N 项；大数组作局部变量还要考虑栈空间。

`array<int,5> a{};` 的五项为 0；`array<int,5> a;` 中基本整数元素没有自动清零，不能在赋值前读取。需要运行时决定长度时用 vector，不能把输入的普通变量 n 直接写成标准 C++ 的 `array<int,n>`。

题目原文（自拟演示）： 程序无需输入。建立长度 5 的全零 array，将每项设为 7，再将下标 2 设为 3，只对下标半开区间 [1,4) 升序排序，输出全部元素。

分析： 花括号负责初始清零，fill 对所有已有元素赋值，最后的下标赋值只改变第三项；这些动作都不会改变长度。之后 begin()+1 指下标 1，end()-1 指下标 4；sort 处理两位置之间的 [1,4)，不动下标 0、4。array 的位置也支持随机跳转，因此可以直接复用算法库接口。




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

int main() {
    array<int, 5> values{}; // 花括号使整数元素先初始化为 0。
    values.fill(7);         // 对五个已有位置赋值，不改变长度。
    values[2] = 3;
    sort(values.begin() + 1, values.end() - 1); // 只排序 [1,4)，两端保持 7。
    for (int i = 0; i < static_cast<int>(values.size()); ++i) {
        if (i > 0)
            cout << ' ';
        cout << values[i];
    }
    cout << '\n';
}
```

输出：

```text
1  7 3 7 7 7
```

这里排序区间的三项由 [7,3,7] 变成 [3,7,7]，迭代器把“处理哪几项”交给算法。成员 fill(x) 填全部元素，算法库 fill(first,last,x) 也可只填选定区间。访问单项为 $O(1)$，填充、复制、完整遍历均为 $O(N)$；本例加上排序总时间为 $O(N + k\log(k+1))$，k 是选中区间长度。空的 array<T,0> 允许存在，但不能读取首尾。

练习题： 洛谷 P1428 – 小鱼比可爱（可按题面最大规模准备固定数组，再用实际 n 控制遍历）。

17. multiset 与 multimap：保存重复键

17.1 重复是多个独立元素

multiset<T> 与 set 都位于 <set>，multimap<Key,Value> 与 map 都位于 <map>。它们按比较器有序，但允许多个等价键；重复项仍各占一个元素，size 包含全部记录。multimap 没有下标运算符：同一个键可能对应多个值，不能用 mp[key] 唯一确定其中一个。

count(key) 返回等价项数量，费用为 $O(\log n + c)$，c 为匹配数；equal_range(key) 返回 [lower_bound(key),upper_bound(key))，用于遍历全部等价项。find 找到某个等价项或 end，不代表得到完整分组。

操作	set / map	multiset / multimap
insert	返回 (位置,是否成功)，等价键不再插入	返回新项位置，等价键也插入
count(key)	0 或 1	可能大于 1
erase(key)	删除至多一项	删除全部等价项，$O(\log n + c)$
erase(it)	删除指定一项	仍只删除该一项，均摊常数
下标访问	map 有，set 没有	两者都没有



17.2 完整演示：重复删除与按键分组

题目原文（自拟）： 程序无需输入。multiset 初始保存 `2、2、3`，输出 2 的个数；删除找到的一项 2，再输出 2 的个数；随后按键删除所有 2，输出删除数量及剩余总量。multimap 插入 `A→10、A→20、B→30`，输出所有键 A 对应的值，保持它们的插入先后。

分析： 第一次 erase 传迭代器，只删一个 2；第二次 erase 传整数键，删剩下的全部 2。multimap 的等价键元素按插入次序排列，通过 equal_range 取出整个 A 分组，而不是只调用一次 find。

```
#include <iostream>
#include <map>
#include <set>
#include <string>
using namespace std;

int main() {
    multiset<int> values{2, 2, 3};
    cout << values.count(2) << '\n';
    auto it = values.find(2);
    if (it != values.end())
        values.erase(it); // 只删这一项，不能 erase(end())。
    cout << values.count(2) << '\n';
    auto removed = values.erase(2); // 按键删除所有等价项。
    cout << removed << ' ' << values.size() << '\n';

    multimap<string, int> records;
    records.emplace("A", 10);
    records.emplace("A", 20);
    records.emplace("B", 30);
    auto [first, last] = records.equal_range("A");
    bool separator = false;
    for (auto position = first; position != last; ++position) {
```

```cpp
        if (separator)
            cout << ' ';
        cout << position->second; // 同键可以有多个独立值。
        separator = true;
    }
    cout << '\n';
}
```

输出：

```text
1 2
2 1
3 1 1
4 10 20
```

unordered_multiset、unordered_multimap 分别位于 `<unordered_set>`、`<unordered_map>`，也允许重复键，但无全局排序；要统计大量重复频次，映射到计数往往比保存每份重复记录更合适。不要为了“需要重复”一律选择 multi，先看是否要保留每项附加信息。

练习题： 洛谷 P2234 - 营业额统计（有序邻近查找）；洛谷 P2073 - 送花（按价格维护记录，重复价格如何处理以题面为准）。

18. forward_list：单向链表

`forward_list<T>` 位于 `<forward_list>`，是单向链表，只支持向前迭代，没有下标、back、push_back 或 size。它提供 `before_begin()` 表示第一个节点之前的位置，便于在头节点前插入或删除；该位置不能解引用。

`insert_after(it,x)` 在 it 后插入；`erase_after(it)` 删除 it 后的一项，返回后续位置。这与 list 的“在 it 前插入、删除 it 本身”不同。已知前驱时单项修改为 $O(1)$，找前驱仍需扫描。也有 sort、unique、remove 等成员操作，普通比赛代码更常用 vector 或 list，了解接口差异即可。

题目原文（自拟演示）： 程序无需输入。单向链表初始为 `1、3`，在头部插入 2，再删除 2 后面的一项，输出剩余序列。

分析： before_begin 位于首项之前，在它后面插入就得到 `2、1、3`。此时 begin 指向 2，erase_after(begin()) 删除的是下一项 1，结果为 `2、3`，不是删除 2 本身。




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

int main() {
    forward_list<int> values{1, 3};
    values.insert_after(values.before_begin(), 2); // 变为 [2,1,3]。
    values.erase_after(values.begin());            // begin 指向 2，实际删除它后面的 1。
    bool first = true;
    for (int value : values) {
        if (!first)
            cout << ' ';
        cout << value;
        first = false;
    }
    cout << '\n';
}
```

输出：

```text
2 3
```

单链表迭代器沿 next 向后走，所以只有 ++it。erase_after 删除前驱后面的那一项；图中先找到 20 的前驱 10，再由这个位置删除 20。频繁操作尾部时，可以保存尾部位置，或选择提供尾部接口的容器。

练习题：洛谷 P1160 – 队列安排（需要左右插入时对比单、双向连接的差异，不能直接照搬本节的两次操

作）。

19. 根据操作选择容器

19.1 操作需求与容器选择

需求	常见选择	判断依据
大小固定、编译时已知	array	固定长度，连续存储
按下标访问，主要尾部追加	vector	连续动态数组，尾插均摊常数
两端都频繁加入或移除	deque	两端常数操作，可随机访问
已知节点位置后频繁插入、删除、移动	list	不需搬移整个后缀；定位成本另计
只需后进先出	stack	只暴露栈顶接口，状态约束清楚
只需先进先出	queue	队头出、队尾入
动态取当前最优先项	priority_queue	不维护全序，只保证堆顶
动态有序去重、前驱后继	set	有序键，成员边界查找对数时间
有序键到值的映射	map	保存键值关系，可按键遍历
只需判重、查频次，不依赖键顺序	unordered_set / unordered_map	平均常数查找，最坏退化另计
相同键的多项都要分别保存	multi 版本	明确重复项的删除与分组语义

静态数据一次排序、多次二分查询，通常 vector 就足够；反复向有序序列中间插入则要计入移动费用。set 不提供高效的第 k 小接口，priority_queue 不支持任意定位删除，unordered 容器不提供有序边界，不能因一个接口方便就假设其他操作也很快。

19.2 学完各容器后对照迭代器能力

所属容器	可以做什么	不支持什么
vector、string、deque、array	`++it`、`--it`、`it+k`、合法同序列位置相减	越过合法边界再读元素
list、set、map 及其 multi 版本	前后逐项移动；`next(it,k)` 需走 k 步	`it+k`、`it2-it1`、通用 sort
forward_list、unordered 容器	向前逐项移动	`--it`、随机跳转；unordered 不提供有序边界查找
stack、queue、priority_queue	通过各自接口访问指定端	普通迭代器和下标遍历

`distance(first,last)` 对随机访问迭代器可直接相减，对其他迭代器一般要逐项移动。`next`、`prev` 位于 `<iterator>`；它们不会让链表突然获得 $O(1)$ 的下标访问。

19.3 从元素与迭代器理解修改后的访问

1. **访问是否存在：** 下标小于 size；front、back、top、pop 前非空；find 结果与 end 比较；区间方向和开闭正确。
2. **修改是否改变题意：** 下标查询 map 是否意外插入；set 是否丢失本应保留的重复项；unordered 遍历是否被误当输入顺序；pair 的平局规则是否满足要求。
3. **旧位置是否还能用：** vector 扩容和后缀移动、deque 两端或中间修改、哈希 rehash 各有失效规则；使用 erase 返回值继续遍历，不能继续读被删项。

`clear()` 后元素不存在，即使容量还在也不能按旧下标访问。size 为无符号类型，空容器上的 `size()-1` 会先下溢；倒序遍历可以用反向迭代器，或在范围已知可放进 int 时先转为有符号下标。竞赛宏把 int 改成 long long 后，元素存储也会变宽，应按实际类型与容量估算内存。

这些规则的容器类型区分见 C++17 草案：关联容器要求。学习时先独立运行各节的小例题，再把容器与后面的搜索、二分、贪心和图算法结合起来。

20. `<numeric>`：填值、求和与求公约数

已经能遍历数组，每次写“从起点依次加一”和“把所有元素相加”仍会重复。`<numeric>` 提供这些常用操作，可以用已经认识的迭代器区间调用。

接口	参数含义	返回值或效果
`iota(first,last,start)`	对已有的半开区间从 start 连续赋值	不创建元素，依次写 start、start+1…
`accumulate(first,last,initial)`	从 initial 开始累计	初值的类型决定累计类型；求整数总和常用 0LL
`gcd(a,b)`	两个整数	最大公约数，`gcd(0,0)` 为 0
`lcm(a,b)`	两个整数	最小公倍数，任一参数为 0 时结果为 0

题目原文（自拟）： 程序无需输入。建立四项整数数组，连续填为 1、2、3、4；第一行输出全部元素，第二行输出总和、12 与 18 的最大公约数和最小公倍数，以空格分隔。

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

int main() {
    vector<int> values(4);                  // 先建立已有区间，iota 不会增长长度。
    iota(values.begin(), values.end(), 1); // 依次填入 1、2、3、4。
    for (int i = 0; i < 4; ++i) {
        if (i > 0)
            cout << ' ';
```

```cpp
        cout << values[i];
    }
    cout << '\n';
    long long total = accumulate(values.begin(), values.end(), 0LL); // 按 long long 累计。
    cout << total << ' ' << gcd(12, 18) << ' ' << lcm(12, 18) << '\n';
}
```

输出：

```text
1 2 3 4
10 6 36
```

填值、求和各遍历一次，共 $O(n)$ 时间。gcd、lcm 从 C++17 起提供，分别返回最大公约数与最小公倍数；结果类型由传入的整数类型决定。数论章节会通过辗转相除继续推导公约数算法。

练习题： 洛谷 P1029 — 最大公约数和最小公倍数问题（先检查给定两数能否形成所需关系，再枚举符合条件的数对；可在数论章节学完后回做）。

21. tuple：多字段记录

pair 能保存两个字段。若一条记录要同时保存行、列和代价，可以用 `<tuple>` 的 `tuple<int,int,long long>`；各类型按字段顺序列在尖括号里。

题目原文（自拟）： 程序无需输入。保存记录 `(2,3,5)`，先输出全部字段；再通过结构化绑定得到副本，把副本中的行改成 9，输出副本。原记录不能改变。

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

int main() {
    tuple<int, int, long long> record{2, 3, 5}; // 三个类型分别对应三个字段。
    cout << get<0>(record) << ' ' << get<1>(record) << ' ' << get<2>(record) << '\n';
    auto [row, column, cost] = record; // 值绑定得到副本。
    row = 9;
    cout << row << ' ' << column << ' ' << cost << '\n';
}
```

输出为 `2 3 5`、`9 3 5`，各占一行。`get<0>(record)` 是函数模板调用：0 是编译期字段编号，从 0 开始，不是运行时下标运算。要修改原记录可用 `get<0>(record)=9` 或引用绑定；成员名较多且有明确含义时，自己定义 struct 通常更易读。

tuple 默认按字段依次进行字典序比较，与 pair 的思路相同。DP 的区间恢复和图论的边记录会用到它；遇到 `auto [left,right,sum]` 时，先核对返回 tuple 的字段排列。

22. <sstream>：从一行文本读取数据

有时题目一行包含若干整数，但没有告诉这一行有几个。先把整行读进 string，再像读取 cin 一样从这段文本里解析整数，可以把“行的边界”与“数值的读取”分开。

题目原文（自拟）： 输入一行，只含合法的整数和空白，允许空行；整数及累计总和都在 long long 范围内。输出这一行整数的总和，空行输出 0。

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

int main() {
    string line;
    getline(cin, line);         // 首次读取就是整行，没有前一次数字输入残留的换行。
    istringstream input(line); // <sstream> 把这段字符串作为输入来源。
    long long value, total = 0;
    while (input >> value)
        total += value; // 读完整行或解析失败时停止。
    cout << total << '\n';
}
```

输入：

```text
12 -5 7
```

输出：

```text
14
```

istringstream 只读取已经保存的文本，不会越过这一行去取 cin 中下一行的数据。它仍遵守整数解析规则；若题面允许非法字符，要另外检查，不应把遇到非法字符后的部分误认为已经全部处理。

（十六） std::function：保存、传递和更换计算动作

1. 从“保存数据”到“保存要执行的动作”

STL 已能保存数和记录。如果希望先选择“加一”或“加上偏移量”，之后再统一调用一个名字执行它，能不能把计算动作也保存起来？普通函数与 lambda 都能被调用，std::function 用一个约定的调用形式接住它们。

题目原文（自拟）： 程序无需输入。先令规则为“加一”，作用于 3；再换成捕获偏移量 5 的“加上偏移量”，作用于 3；最后换成“平方”，作用于 3。每次输出结果。

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

```cpp

int addOne(int x) {
    return x + 1;
} // 一个普通函数。

int main() {
    function<int(int)> rule = addOne; // 接收一个 int，返回一个 int。
    cout << rule(3) << '\n';
    int offset = 5;
    rule = [offset](int x) { return x + offset; }; // 改存带捕获的 lambda。
    cout << rule(3) << '\n';
    rule = [](int x) { return x * x; }; // 再次更换规则，调用形式仍相同。
    cout << rule(3) << '\n';
}
```

输出：

```text
4
8
9
```



`<functional>` 提供 function。`function<int(int)>` 的外层是模板；尖括号里的 `int(int)` 是**函数类型**：前一个 int 是返回类型，括号里的 int 是参数类型。不是两个整数值，更不是把输入 3 写在类型里。`rule(3)` 才是实际调用。

写法	调用约定	例子
`function<int(int)>`	一个 int 参数，返回 int	整数变换

写法	调用约定	例子
`function<bool(int,int)>`	两个 int 参数，返回 bool	判断两个数的关系
`function<double(double)>`	一个 double 参数，返回 double	后面二分求根使用的函数 f
`function<void(int)>`	一个 int 参数，不返回结果值	输出动作、搜索中的调用

不同 lambda 表达式具有不同的类型，auto 变量由初始化时的表达式确定类型；std::function 则把能按指定形式调用的目标包装成同一个类型，因此可以更换保存的动作。包装后仍按签名传入参数、接收返回值，这种统一调用接口的方式也称为类型擦除。

2. 把动作作为参数传给另一段处理过程

题目原文（自拟）： 程序无需输入。数组为 `1、2、3`，分别按“乘 2”和“加 10”的规则转换并输出，每个规则的结果占一行，以空格分隔。

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

void printTransformed(const vector<int> &values, const function<int(int)> &rule) {
    for (int i = 0; i < static_cast<int>(values.size()); ++i) {
        if (i > 0)
            cout << ' ';
        cout << rule(values[i]); // 过程不决定公式，只负责把每项交给规则。
    }
    cout << '\n';
}

int main() {
    vector<int> values{1, 2, 3};
    printTransformed(values, [](int x) { return x * 2; });  // 传入第一个动作。
    printTransformed(values, [](int x) { return x + 10; }); // 数据相同，动作不同。
}
```

输出为 `2 4 6`、`11 12 13`，各占一行。只读引用避免在 printTransformed 的参数处再复制已有容器与函数对象；这里传入的临时 function 在整个调用期间有效。容器的遍历和计算规则分别负责自己的部分，后面的求根代码就会把 f 作为这样的参数。

3. 空目标与已保存的动作

```cpp
std::function<int(int)> rule; // 默认构造时为空，没有目标可以执行。
if (rule) {                   // 转成逻辑判断：检查是否已保存一个可调用目标。
    rule(3);                  // 只有存在目标时才调用；本片段需 <functional>。
}
```

直接调用空 function 会抛出 `std::bad_function_call`，不是默认返回 0。改用 `rule=nullptr` 可以清空目标。保存引用捕获的 lambda 时，function 也不会延长它所引用变量的生命周期。

4. 递归 lambda：先准备名字，再让动作调用这个名字

题目原文（自拟）： 程序无需输入。从 3 开始依次输出 `3、2、1`，到 0 时停止；用一个能够调用自己的 lambda 完成，元素之间一个空格。

推导： 输出当前剩余数 left 后，下一个动作与当前完全相同，只是参数变成 left-1；left=0 时直接返回，不再调用。先声明 countDown，再给它赋值，lambda 才能引用一个已声明的名字。

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

int main() {
    function<void(int)> countDown; // 先声明用于调用的名字。
    countDown = [&countDown](int left) {
        if (left == 0)
            return; // 这一支结束递归，不再调用下一层。
        cout << left;
        if (left > 1)
            cout << ' ';
        countDown(left - 1); // 调用同一动作，参数逐次减小。
    };
    countDown(3);
    cout << '\n';
}
```



输出 `3 2 1`。先进入 3，再进入 2、1、0；0 返回后，依次回到 1、2、3 的调用现场，每层从调用后的语句继续执行。本例之后没有其他操作，所以直接继续返回。搜索章节会在这个基础上加入选择、回溯和记忆化，并画出返回后还要做什么。

`std::function` 把不同的可调用对象包装成同一个调用接口。包装可能分配存储，调用还会经过包装层，所以计算费用时既要计入包装与调用，也要计入目标函数本身的工作。固定动作可以直接使用普通函数或 `auto lambda`；把动作作为参数传递、保存或替换时，`function` 提供统一的类型。C++17 的 `function` 会保存可复制的目标，例如前面按值捕获整数的 `lambda` 可以随包装一起复制。相关接口见 C++17 草案：function。

现在能够把数据组成容器，也能把动作传给处理过程。下一部分用自定义类把相关的数据、约束和操作放到一起，先从学生记录这样的具体需求开始。

二、面向对象编程

前面已用 struct 组织简单记录，用容器保存数据，用 function 传递动作。现在从学生姓名、成绩及合法范围出发，把相关数据和操作放进同一类型，再逐步处理初始化、生命周期与不同对象的共同接口。

（一） 类与对象

想让每个学生同时拥有姓名、成绩和及格判断，可以从模板章的小记录继续扩展，而不是用互不相关的变量维护同一个人。

可以把**类（class）**理解为一种自定义类型，把**对象（object）**理解为该类型的具体实例。类把相关数据和操作放在一起，例如学生的姓名、成绩和判断是否及格的方法。

区分类本身和每个对象的数据。 `Student` 定义了学生记录的组成和可执行操作；`Student alice; Student bob;` 创建两个实例。修改 `alice.score` 不会修改 `bob.score`，因为普通非静态数据成员分别属于各自对象。调用 `alice.passed()` 时，函数读取的是 `alice` 的成绩。

类的作用是让“属于同一个对象的数据和操作”有明确位置。竞赛中把边的终点、权值放进 `Edge`，比用两个独立数组再手工维持相同下标更容易理解。是否需要隐藏数据、继承或虚函数，要由问题决定；一个只有公开数据的 `struct` 也可以是合适的设计。

这里先把 class 当成带数据和成员函数的自定义类型阅读；public: 之后的成员允许类外访问，passed() 通过当前对象的 score 判断。下一节再收紧成绩的修改权限。

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

class Student {
  public:
    string name;
    int score = 0;

    bool passed() const { // const 表示判断成绩时不修改当前对象。
        return score ≥ 60;
    }
};

int main() {
    Student student;
    student.name = "Alice";
    student.score = 85;
    cout << student.name << ' ' << student.passed() << '\n'; // Alice 1
```

```
20  }                                                                    cpp
```

说明：

- `name`、`score` 是数据成员；`passed()` 是成员函数。
- `object.member` 访问对象成员；`pointer->member` 通过有效指针访问成员。
- `struct` 和 `class` 都可以拥有成员函数、构造函数和继承关系。它们的主要语法差异是默认成员访问权限和默认继承权限：`struct` 为 `public`，`class` 为 `private`。

竞赛中常用 `struct Edge { int to; long long weight; };` 表示一条边，不必把所有结构都写成复杂的类。

（二） 封装与访问控制

学生记录能保存成绩后，还要阻止外部写入 -1 或 101。先用公开方法检查范围，再说明哪些成员应允许外部直接访问。

封装是把数据及其操作组织起来，通过接口维护对象的约束。例如成绩只能在 `0` 到 `100` 之间，就让修改成绩的方法统一检查范围。

权限	类内部	派生类成员	类外普通代码
`public`	可访问	可访问	可访问
`protected`	可访问	可以按保护成员规则访问	不可直接访问
`private`	可访问	不可直接访问	不可直接访问

如果希望某个外部函数或类也能访问非公开成员，可以把它声明为友元。普通成员访问仍按上表中的权限进行。

这里维护的对象不变量是 `0 ≤ score_ ≤ 100`。成员先用已学的默认初值设为 0；每次修改只允许通过 setScore 检查范围。因此，只经公开接口操作的对象始终保持合法成绩。

例如当前成绩为 `85`，调用 `setScore(90)` 会变成 `90`；调用 `setScore(-1)` 则先抛出异常，没有执行后面的赋值，成绩保持修改前的值。调用者需要处理异常，否则异常继续向外传播。`private` 限制的是访问途径，不会自动判断成绩是否合法，setScore 中的判断负责检查成绩数值。

本例用到的异常接口： `<stdexcept>` 的 invalid_argument 表示参数不满足约定；`throw invalid_argument("...")` 抛出带文字说明的异常，可用 `catch(const invalid_argument& error)` 捕获，再用 `error.what()` 读取说明。判断在赋值之前，因此抛出异常不会写入非法成绩。

```cpp
// 数据成员设为 private，修改成绩统一经过 setScore 的范围检查。
#include <iostream>
#include <stdexcept>
#include <string>
using namespace std;

class Student {
```

```cpp
  private:
    string name_;
    int score_ = 0; // 对象创建时先给合法初值。

  public:
    void setName(const string &name) {
        name_ = name; // 先用已经学过的赋值，不要求先掌握构造函数。
    }

    void setScore(int score) {
        if (score < 0 || score > 100) {
            throw invalid_argument("成绩必须在 0 到 100 之间");
        }
        score_ = score;
    }

    int score() const {
        return score_;
    }

    const string &name() const {
        return name_;
    }
};

int main() {
    Student student;
    student.setName("Alice");
    student.setScore(85);
    cout << student.name() << ' ' << student.score() << '\n';
}
```

外部不能直接写 `student.score_ = -1`，只能调用公开接口。`name()` 返回的引用仍依赖 `student` 的生命周期，不能在对象销毁后继续使用。

（三） 构造函数、初始化列表与析构函数

上一节先创建对象再调用设置方法；现在希望创建时就准备好数据，并在离开作用域时完成清理。沿一个对象从创建到销毁观察这些操作。

构造函数在对象创建时初始化它；**析构函数**在对象生命周期结束时执行清理。

- 构造函数与类同名，没有返回类型，可以重载。
- 析构函数写为 `~类名()`，没有返回类型和参数。
- `explicit` 可以阻止构造函数参与某些隐式转换。
- 成员初始化列表写在函数体之前，如 `: name_(name), score_(score)`。

> **注意：** 成员初始化顺序取决于它们在类中的**声明顺序**，不取决于初始化列表的书写顺序。`const` 成员和引用成员通常需要在初始化阶段给出值。

初始化与赋值发生在不同阶段。 初始化列表直接决定成员如何创建；在构造函数体里写赋值时，成员已经完成初始化，再执行赋值。例如 `string` 成员在进入函数体前就已经是一个对象，所以优先在初始化列表中提供初值。引用在初始化列表中绑定到对象，const 成员也在这里获得初值；进入函数体时，它们已经完成创建。

下面按执行顺序观察：创建 `first` 时输出“构造 A”；进入内层作用域创建 `second`，输出“构造 B”；遇到内层右花括号先销毁 `second`，输出“析构 B”；之后才执行“离开内层作用域”；`main` 结束时销毁 `first`。同一作用域中的普通局部对象按完成构造的相反顺序销毁，这让资源能随作用域自动释放。

```cpp
// 构造与析构打印对象名；内层 B 先离开作用域，因此先析构 B。
#include <iostream>
#include <string>
using namespace std;

class Trace {
  private:
    string name_;

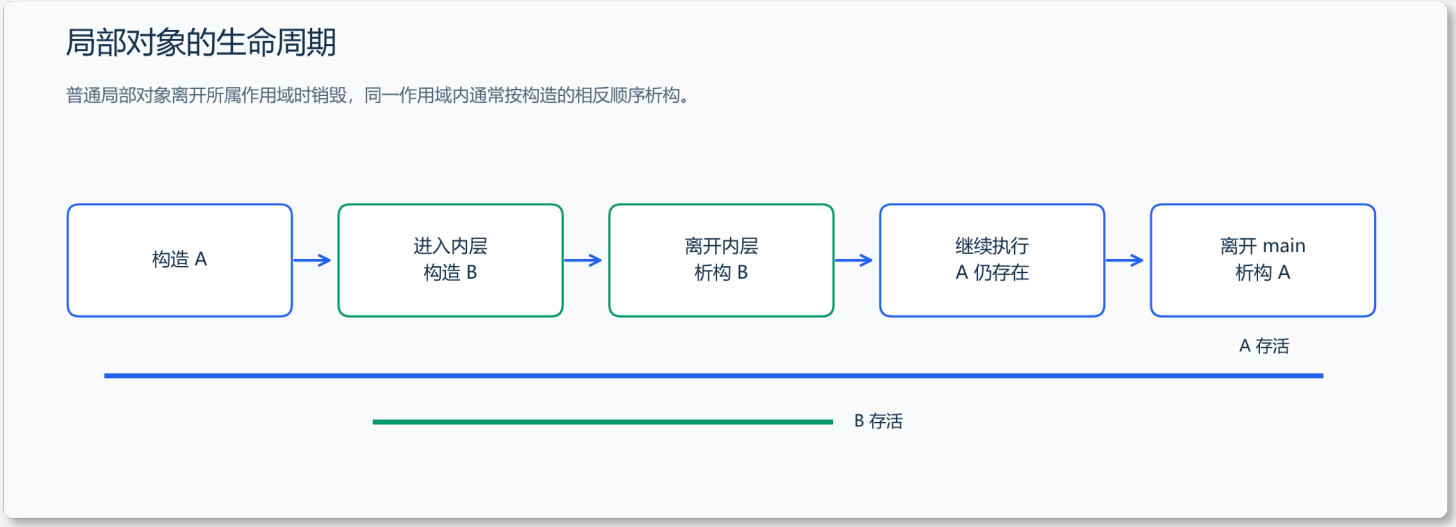
  public:
    explicit Trace(string name) : name_(name) {
        cout << "构造 " << name_ << '\n';
    }

    ~Trace() {
        cout << "析构 " << name_ << '\n';
    }
};

int main() {
    Trace first("A");
    {
        Trace second("B");
    }
    cout << "离开内层作用域\n";
}
```

输出：

```text
构造 A
构造 B
析构 B
离开内层作用域
析构 A
```



普通局部对象在离开作用域时销毁；使用 `new` 创建的对象需要由相应所有者负责释放。

（四） this、const 成员函数与 static 成员

当有多个计数器时，需要区分“修改当前这一个”的状态与“所有对象共享”的统计，再判断哪些操作只是读取。

`this` 是非静态成员函数中指向当前对象的指针。例如 `this->score_` 就表示当前对象的成绩成员。

`this` 指定操作的是哪一个实例。 `counter.add(2)` 中，`this` 指向 `counter`；`return *this` 返回当前对象的引用，所以紧接着的 `.add(3)` 仍在修改同一个对象，最终值为 `5`。若返回独立副本，后续操作可能只改动副本，链式调用的含义就不同了。

`value_` 每个对象各有一份，`created_` 则全体对象共享一份。这里的 `created_` 只在默认构造函数中增加，没有在析构时减少，因此不是“当前活着的对象数”；编译器生成的拷贝操作也不会执行这个默认构造函数。静态函数没有某个默认的当前对象，所以不能直接读取 `value_`。

```cpp
class Counter {
  private:
    int value_ = 0;
    inline static int created_ = 0; // C++17: 所有对象共享

  public:
    Counter() {
        ++created_;
    }

    Counter &add(int amount) {
        this->value_ += amount;
        return *this; // 返回当前对象，便于继续调用
    }

    int value() const {
        return value_;
    }

    static int created() {
        return created_;
```

```cpp
    }
};
```

调用片段：

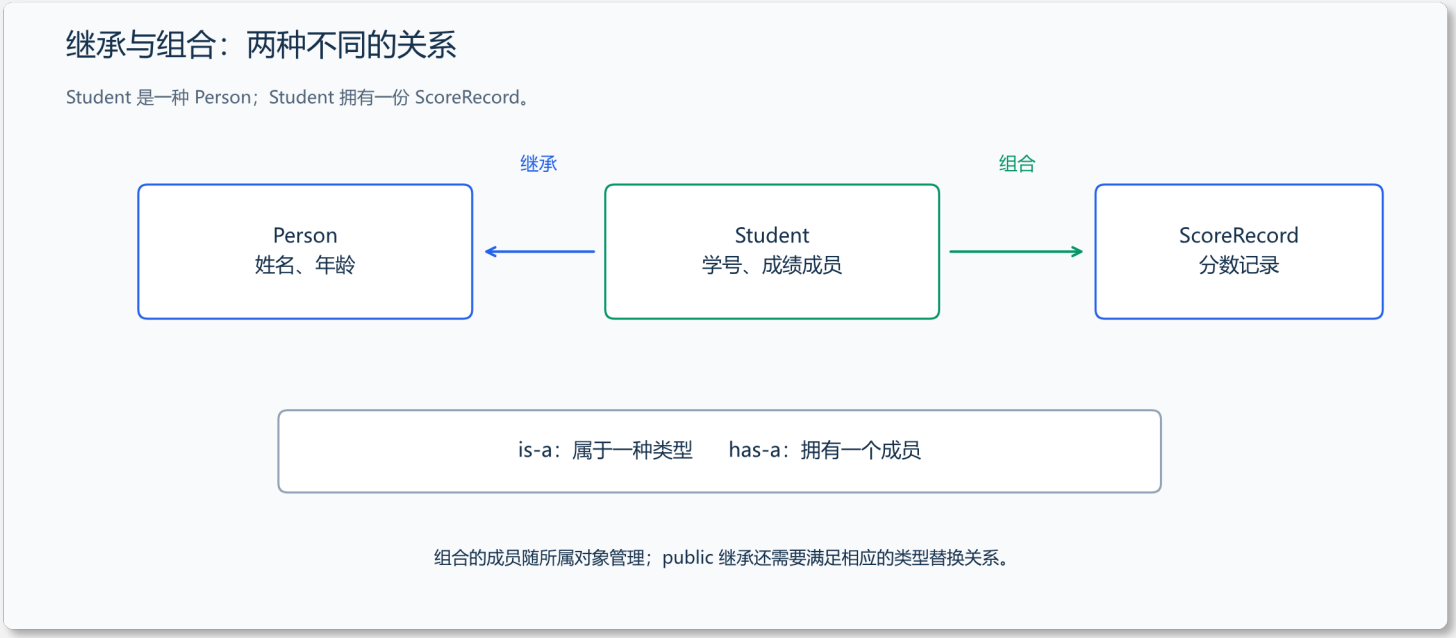
```cpp
Counter counter;
counter.add(2).add(3);
const Counter &view = counter;
// view.value() 为 5；Counter::created() 为已默认构造的对象数
```

`const` 成员函数承诺不通过该函数修改普通非静态数据成员，可由 `const` 对象调用。静态成员函数没有 `this`，不能直接访问某个对象的非静态成员。

（五） 继承与组合

对象的成员和操作已经明确后，再面对两种关系：学生属于人的一种，学生同时拥有一份成绩记录。先分清需求，再选继承或组合。

继承用来表达“派生类型属于基类类型”，例如“圆形是一种图形”；**组合**用来表达“对象拥有某种成员”，例如“学生拥有成绩记录”。



图中 Student 属于 Person，ScoreRecord 则是 Student 的成员，两种箭头表示不同关系。

```cpp
// 派生类先构造基类部分；protected 成员可由派生类访问。
#include <string>
using namespace std;

class Person {
  protected:
    string name_;

  public:
    explicit Person(string name) : name_(name) {}
};
```

```cpp

// 成绩记录作为 Student 的成员，表示组合关系。
struct ScoreRecord {
    int value = 0;
};

class Student : public Person {
  private:
    ScoreRecord score_;

  public:
    Student(string name, int score) : Person(name), score_{score} {}
};
```

派生对象先构造基类部分，再构造成员，最后执行派生类构造函数体；析构时大体按相反顺序进行。基类的 `private` 成员仍存在于对象中，只是派生类不能直接访问。

对于新手，优先使用清晰的 `public` 继承关系；如果只是想复用一个容器或辅助对象，直接把它作为成员通常更容易理解。

（六） 虚函数、重写与多态

把矩形和圆放到同一列表后，希望统一调用 area，却按实际图形采用不同公式。先沿这一调用观察，再解释 virtual 与 override。

运行时多态可以理解为：通过同一种基类接口操作不同派生对象，调用的实现由实际对象类型决定。

逐步看 `shape->area()` 怎样选择实现。 先创建具体的矩形和圆，再在 vector 中保存它们的 Shape* 地址，所以循环中的接口统一为 Shape。这些地址实际指向的对象分别是 `Rectangle` 和 `Circle`；`area` 被声明为虚函数，调用时会选择实际对象对应的实现。因此矩形 `3×4` 返回 `12`，半径为 `1` 的圆返回约 `3.14159`。

`Shape::area() const = 0` 说明基类只提出“图形应能求面积”这个接口，具体公式交给派生类。`virtual ~Shape() = default;` 让编译器提供这个虚析构函数的默认实现，并非“什么都不析构”。`override` 会检查派生类是否真正重写了该接口，例如漏写 `const` 就会报错。循环不需要再逐个判断“这是圆还是矩形”；新增一种正确实现接口的图形后，仍可使用同一个调用过程。

circle、rectangle 是 main 中的两个对象，shapes 保存它们的地址。循环取出地址后调用 area，执行的就是对应对象的面积计算；main 结束时，这两个局部对象随之销毁。

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

class Shape {
  public:
    virtual double area() const = 0; // 纯虚函数
    virtual ~Shape() = default;
};

```

```cpp
class Rectangle : public Shape {
  private:
    double width_;
    double height_;

  public:
    Rectangle(double width, double height) : width_(width), height_(height) {}

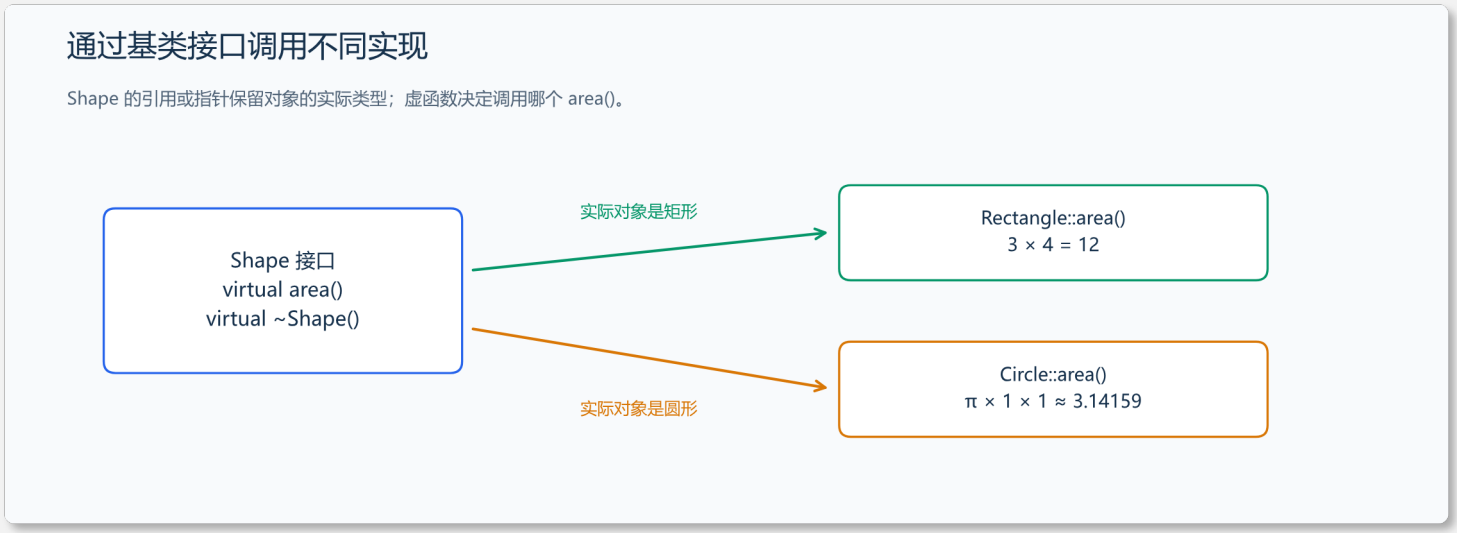
    double area() const override {
        return width_ * height_;
    }
};

class Circle : public Shape {
  private:
    double radius_;

  public:
    explicit Circle(double radius) : radius_(radius) {}

    double area() const override {
        constexpr double pi = 3.14159265358979323846;
        return pi * radius_ * radius_;
    }
};

int main() {
    Rectangle rectangle(3, 4);
    Circle circle(1);
    vector<Shape *> shapes{&rectangle, &circle}; // 只保存有效地址，不拥有对象。
    for (const auto &shape : shapes) {
        cout << shape->area() << '\n';
    }
}
```



说明：

- `virtual` 声明虚函数，派生类提供对应实现。

- `override` 让编译器检查重写是否正确，能发现参数或 `const` 不一致等问题。
- 存在纯虚函数尚未实现的类是抽象类，不能直接创建该类对象。
- 通过基类指针或引用调用虚函数，可以保留运行时多态。
- 把派生对象按值复制成基类对象会发生**对象切片**，派生部分不会被保留。

基类的虚析构函数让销毁过程也能按实际对象类型进行：通过基类接口释放对象时，先执行派生类析构，再执行基类析构。上面的局部对象由所在作用域管理；动态对象的释放在下一节结合资源管理说明。参见 C++ 标准草案：delete 表达式。

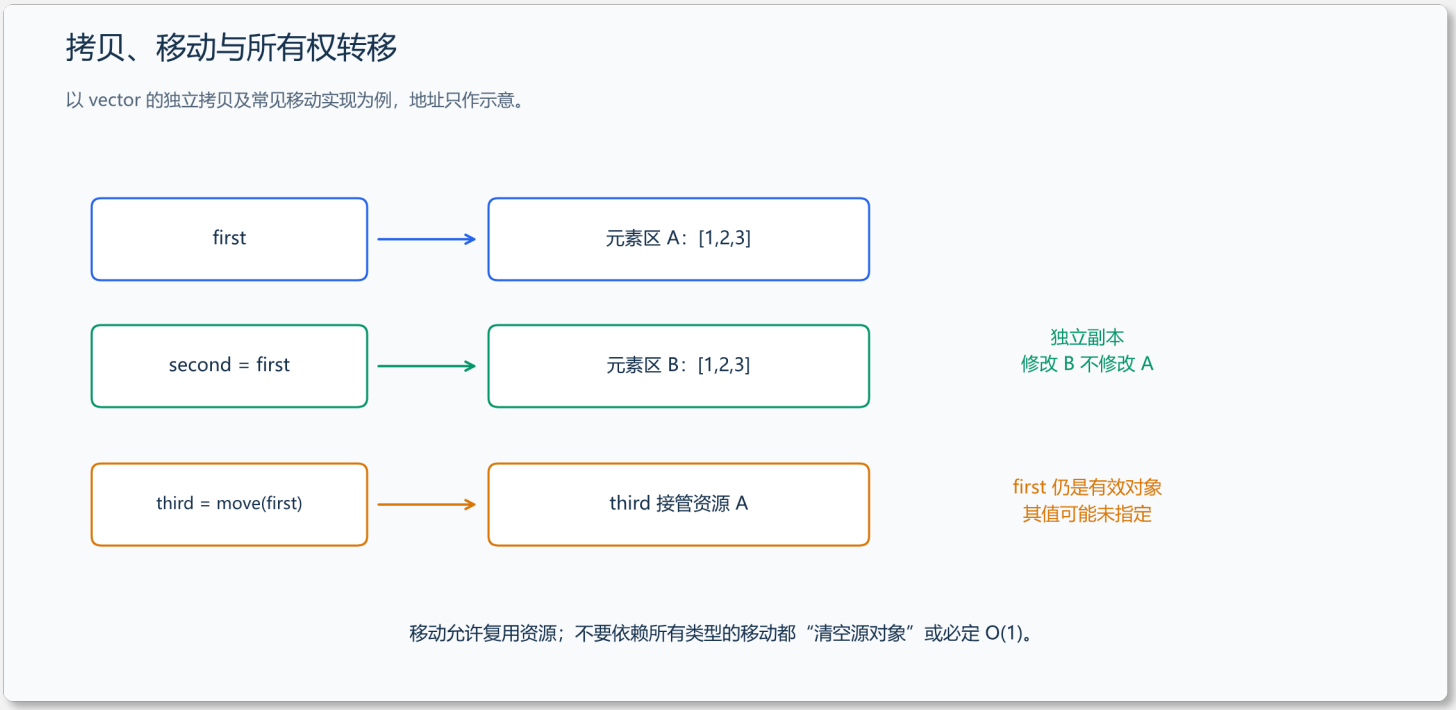
函数重载依靠不同参数列表区分；虚函数重写依靠继承关系和匹配的函数签名。二者不是同一个概念。

（七） 拷贝、移动与资源自动管理

已经能让对象随作用域销毁，接下来检查复制或转交拥有资源的对象时，数据与释放责任应落在哪一份对象上。

拷贝关注独立对象，移动关注资源转交。 对下面的 `vector`，拷贝后修改 `second[0]` 不会影响 `first[0]`，因为两个容器各自拥有元素。移动允许新对象接管资源；移动后的原容器仍能销毁、重新赋值，但在检查其当前状态之前，不能继续假定它还保留原来的三个元素。

如果类中直接拥有一个 `new` 得到的裸指针，而拷贝只是复制地址，两个对象可能都把同一地址当成自己的资源，析构时便会重复释放。使用 `vector` 保存数组，或用下面将说明的独占指针表达一个所有者，可以让成员类型处理相应生命周期。这就是零规则适合新手的原因：先选好资源成员，再尽量依赖编译器和成员的默认操作。



拷贝得到独立数据，移动则允许把已有资源交给另一个对象。移动后的原对象仍可销毁或重新赋值；图中的箭头表示资源转交，原容器是否为空取决于具体类型的移动行为。

`<utility>` 的 `std::move` 把对象转换成可参与移动操作的形式，实际转交由相应类型的移动操作完成。这里用它给 `third` 提供移动初始化的来源；它本身不执行元素搬运。

```cpp
#include <utility>
#include <vector>
using namespace std;

vector<int> first{1, 2, 3};
vector<int> second = first;            // 拷贝：两个 vector 各自管理元素
vector<int> third = std::move(first); // 移动：允许转移内部资源
```

`std::move` 本身不搬运数据，它把表达式转换成可以参与移动的形式。真正的行为由移动构造函数或移动赋值决定。移动后的标准库对象通常仍有效，但状态未指定；不要直接假定 `first` 一定为空。

RAII（资源获取即初始化）把资源的生命周期交给对象管理：构造时取得资源，析构时释放资源。例如 `vector` 管理动态内存，文件流管理打开的文件，`unique_ptr` 管理独占对象。

从手工 delete 到自动释放： `<memory>` 提供 unique_ptr 等智能指针；`unique_ptr<int>` 是独占一个动态整数的类型，`make_unique<int>(42)` 创建该整数并返回它的所有者。对象退出作用域时自动释放所拥有的资源，移动会转交所有权；不能直接拷贝出两个独占所有者。

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

int main() {
    auto owner = make_unique<int>(42); // 创建整数与独占所有者，失败不会留下裸资源。
    auto receiver = move(owner);       // 转交所有权，unique_ptr 的来源明确变为空。
    cout << (owner == nullptr) << ' ' << *receiver << '\n'; // 输出 1 42。
} // receiver 销毁时自动释放整数；不再手工 delete。
```

这时可以把多态例子中的局部对象访问扩展为自动拥有动态图形：`make_unique<Rectangle>(3,4)` 创建具体图形，`vector<unique_ptr<Shape>>` 保存独占所有者。与仅存 Shape* 相比，现在容器销毁也会释放它拥有的图形；通过基类销毁派生对象仍要求正确的虚析构接口。

工具	所有权含义	常见用途
`unique_ptr<T>`	一个所有者	独占动态对象；可移动，不可拷贝
`shared_ptr<T>`	多个所有者	确实需要共享对象生命周期
`weak_ptr<T>`	不增加共享所有者数量	观察共享对象，辅助避免循环引用
裸指针 / 引用	本身不表达自动释放责任	访问已有对象，需明确生命周期

如果类只使用 `string`、`vector`、智能指针等成员，通常可以依赖它们管理资源，这就是常说的**零规则（Rule of Zero）**。如果手工拥有裸指针资源，就需要认真设计析构、拷贝和移动，避免浅拷贝后重复释放。参见 C++ Core Guidelines: C.20。

（八） 运算符重载与比较规则

整数已有加法与大小比较。给自定义坐标实现同样的操作时，可以把加法写成横、纵坐标分别相加，把大小比较写成先比较横坐标、再比较纵坐标。

运算符重载让自定义类型支持自然的操作。例如坐标点相加：

比较规则要让等价和先后关系保持一致。 下面的点先比较 `x`，`x` 相同再比较 `y`，相当于按 `(x,y)` 的字典序排列。例如 `(1,9) < (2,0)`，而 `(1,3) < (1,9)`。两点都为 `(1,3)` 时，两个方向的比较都为 `false`。

沿这个字典序，若 a 排在 b 前、b 排在 c 前，就能推出 a 排在 c 前；两个方向都不排在前面的点则视为等价。这样的先后与等价关系称为严格弱序。比较学生成绩时，也可以先按成绩，再按编号决定同分记录的次序。operator+ 则把横、纵坐标分别相加，返回一个新点。

```cpp
// 加法逐坐标计算；小于比较采用字典序，保证严格弱序。
struct Point {
    long long x;
    long long y;

    Point operator+(const Point &other) const {
        return {x + other.x, y + other.y};
    }

    bool operator<(const Point &other) const {
        if (x != other.x) {
            return x < other.x;
        }
        return y < other.y;
    }
};
```

在 sort 中，比较函数回答“a 是否排在 b 前”。a 与自身比较时，答案是 false；若写成 `a≤b`，相等时会得到 true，与这个含义相反。上面的字典序使用 <，两字段都相等时自然返回 false。

对 `set` 来说，“两个方向比较都为 `false`”的元素视为等价。如果仅按成绩比较，同分学生可能被当成同一个元素；需要保留他们时，可以再按姓名或唯一编号比较。参见 C++ 标准草案：关联容器要求。

（九） 面向对象部分的练习

1. 用 `struct` 保存学生信息，按成绩降序、编号升序排序。
2. 给成绩类增加合法范围检查，观察非法输入时的行为。
3. 给上面的 `Shape` 增加 `Triangle` 派生类，继续使用同一个遍历循环输出面积。

4. 用 `vector` 或 `unique_ptr` 管理二叉树节点，并说明节点的所有者是谁。

ACM 入门阶段先掌握结构体、构造函数、比较规则和资源管理即可；继承与多态帮助理解 C++ 的类型设计，不需要在每道题中使用。

三、解题方法论：从题意到代码

学过语言和容器后，拿到题目仍可能不知道从哪一步开始。可以先用小数据手算，把故事中的对象、规则和输出写成变量与关系；再提出计算方法，解释每一步为什么能得到需要的结果，最后把它写成代码并逐步核对。

同一题可以有不同模型，也可能有某个看似合理的步骤改写了题意。本章从简单判断推进到计数与事件模拟，沿着具体例题练习反例、证明、状态选择、代码拆分和 WA 排查。



出现差异时，先停在第一处不同的状态，再构造小输入回看对应规则；做完后留下推导和失败例子，方便迁移到下一道题。

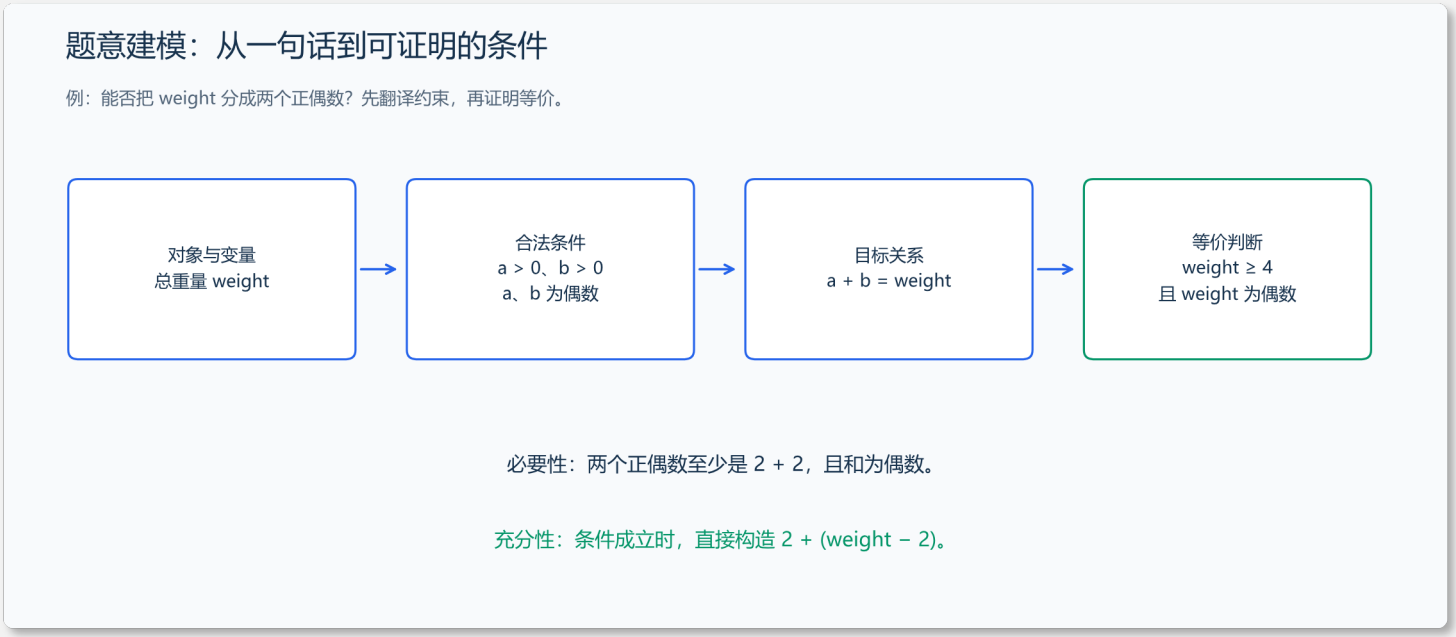
（一） 先提取对象、变量、限制和目标

读题时可以先写下面四项：

项目	要回答的问题	示例
对象	要处理哪些东西？	n 件物品、m 个区间、一个字符串
变量	哪些值已知，哪些值需要求？	已知高度 `h[i]`，求符合条件的数量
限制	哪些操作或状态合法？	每件只能选一次、两人不能用同一个位置
目标	最终输出什么？	是否存在、方案数、最小花费、具体方案

不要把故事中的无关信息都存入程序。例如“仓库有不同颜色的盒子”，如果颜色不影响规则，通常就不需要颜色变量。

数据范围也是题意的一部分。 正数与任意整数会影响滑动窗口的单调性；边权是否非负会影响最短路算法；能否重复选择会影响背包转移方向。



把原要求翻译成约束之后，还要分别证明必要性和充分性；后面会给出完整程序。

（二） 把关键词翻译成条件

自然语言	形式化表达	对代码的影响
至少 k 个	`count ≥ k`	等于 k 也合法
至多 k 个	`count ≤ k`	不能错误地写成 `< k`
恰好 k 个	`count == k`	不能只检查上下界之一
A 且 B	`A && B`	两个条件同时满足
A 或 B，至少一个成立	`A || B`	包括两个都成立
A、B 恰好一个成立	`A ≠ B`，其中 A、B 是 bool	区别于普通“或”
不满足 A	`!A`	对复杂条件注意括号
任意 / 所有元素满足	对每个元素检查	发现一个反例就失败
存在一个元素满足	找到一个即可	不能要求全部成立
两个不同的下标	`i ≠ j`	排序后也不能重复使用同一元素

“至少一个满足”与“全部满足”是不同的量词。一个常见错误是看到某个元素不满足就返回失败，而题目实际只要求存在一个满足的元素。

德摩根律常用于改写不合法条件：

$$\neg(A \wedge B) = (\neg A) \vee (\neg B), \qquad \neg(A \vee B) = (\neg A) \wedge (\neg B)$$

例如 `!(x ≥ 0 && x < n)` 等价于 `x < 0 || x ≥ n`。



同一数组上，“存在一个正数”与“所有元素都是正数”会得到不同的真假结果。

（三） 从要处理的对象确定循环

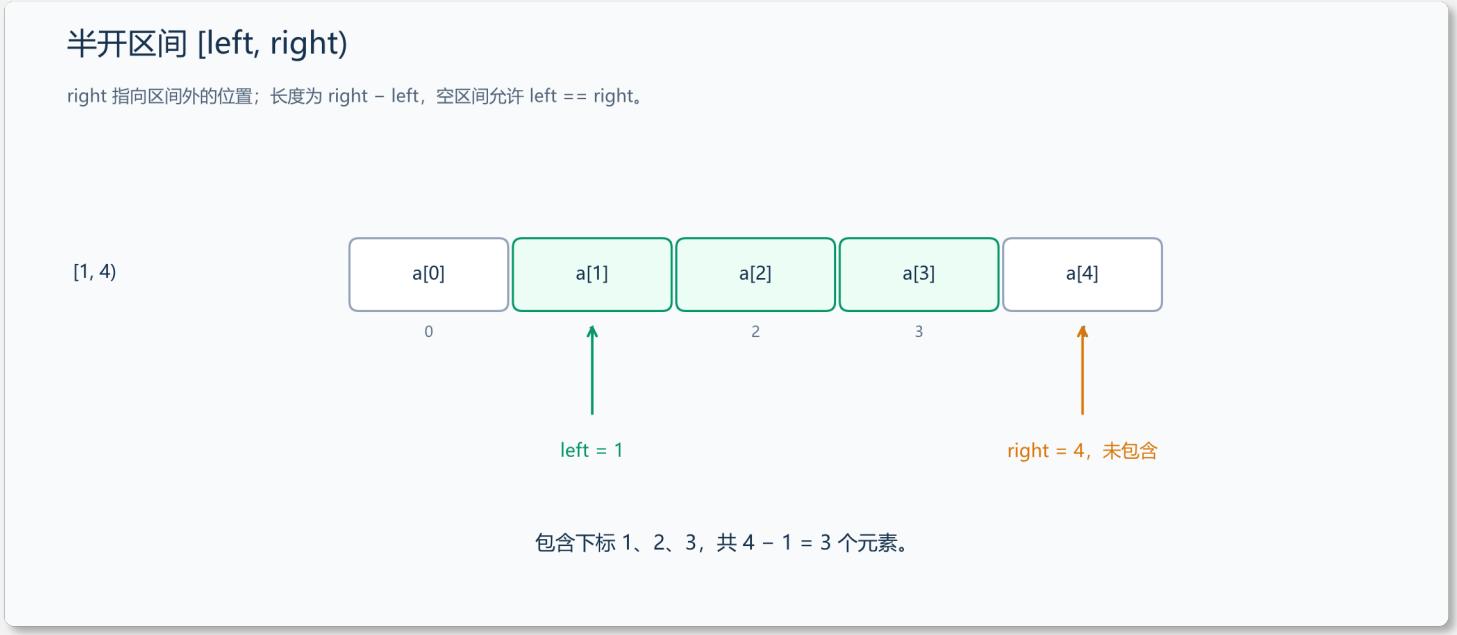
长度为 `n` 的数组，下标集合是：

$$\{0, 1, \dots, n-1\}$$

因此遍历使用 `i < n`。区间 `[left, right)` 长度为 `right - left`，可以为空；闭区间 `[left, right]` 的非空长度为 `right - left + 1`。

枚举两个不同元素时，`0 ≤ i < j < n` 已经同时处理了“不同下标”和“避免重复计数”。如果顺序有意义，例如两人担任不同职务，则 `(i, j)` 和 `(j, i)` 可能是不同方案，不能擅自合并。

> **注意：** 题面使用 `1` 到 `n` 的编号时，读入后可以统一减一；输出下标时再按要求加回去。转换规则应贯穿整个程序。



`right` 表示区间外的第一个位置。先在图上标出包含的下标，再确定循环条件。

（四） 必要条件、充分条件与等价条件

必要条件是“答案成立时一定满足”，但满足它还不一定有答案。**充分条件**是“满足它就可以保证答案成立”。两者同时成立，才能把原题要求等价替换成这个判断。

例：正整数 `a + b` 是偶数，说明 `a`、`b` 同奇偶；但如果要求 `a`、`b` 都是正偶数，只检查总和为偶数还不够，因为 `1 + 3 = 4` 是反例。

1. 完整示例：分成两个正偶数

题目原文（自拟）： 给定整数 `weight`，判断能否写成两个正偶数之和。只输出 `YES` 或 `NO`。输入范围为 `1 ≤ weight ≤ 1000000000`。

分析推导：

1. 两个偶数之和一定是偶数，所以 `weight % 2 == 0` 是必要条件。
2. 两个正偶数各至少为 `2`，所以 `weight ≥ 4` 也是必要条件。
3. 反过来，如果 `weight` 为偶数且至少为 `4`，可以选 `2` 和 `weight - 2`；两者都是正偶数，因此两个条件合起来也是充分条件。

所以原要求等价于：

$$weight \geq 4 \quad \wedge \quad weight \bmod 2 = 0$$

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

int main() {
    int weight;
    cin >> weight;
    // 两个正偶数的和至少为 4；偶数 weight 可拆成 2 与 weight-2。
    bool possible = weight ≥ 4 && weight % 2 == 0;
    cout << (possible ? "YES" : "NO") << '\n';
}
```

输入 `2` 输出 `NO`，输入 `8` 输出 `YES`，输入 `9` 输出 `NO`。

代码解释： 程序只计算一个已经证明充要的条件。时间、额外空间均为 $O(1)$；不需要枚举两个数。

练习： CF 4A - Watermelon。

（五） 从文字规则整理优先级

如果题面给出一般规则和例外，先找出例外覆盖的范围，而不是把所有条件直接用“或”连接。

1. 完整示例：闰年判断

题目原文（自拟）： 输入 `1 ≤ year ≤ 9999` 的年份。按公历规则判断是否为闰年：能被 `4` 整除通常是闰年；能被 `100` 整除的年份只有同时能被 `400` 整除才是闰年。闰年输出 `YES`，否则输出 `NO`。

分析推导： 先拆成两种互相兼容的合法情况：能被 `400` 整除，或者能被 `4` 整除且不能被 `100` 整除。

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

int main() {
    int year;
    cin >> year;
    // 整百年份由 400 的条件决定，普通年份检查 4 的条件。
    bool leap = year % 400 == 0 || (year % 4 == 0 && year % 100 ≠ 0);
    cout << (leap ? "YES" : "NO") << '\n';
}
```

样例与解释： `2000` 输出 `YES`，`1900` 输出 `NO`，`2024` 输出 `YES`。`1900` 是只检查 `% 4 == 0` 时会答错的反例。时间、额外空间均为 $O(1)$。

练习： 洛谷 P5711 - 闰年判断、洛谷 P5710 - 数的性质。

（六） 从计数要求推导向上取整

1. 完整示例：载客人数与最少车辆数

题目原文（自拟）： 有 `people` 个人，每辆车最多坐 `capacity` 人，可以有空座。求至少需要多少辆车。输入两个整数，`0 ≤ people ≤ 10^18`，`1 ≤ capacity ≤ 10^18`。

分析推导： 需要最小整数 `cars` 满足 `cars * capacity ≥ people`，因此数学答案是：

$$cars = \left\lceil \frac{people}{capacity} \right\rceil$$

不要先进行整数除法再调用 `ceil`，因为小数部分已经被丢掉。直接用商和余数：

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

int main() {
    long long people, capacity;
    cin >> people >> capacity;
    // 完整装满的车辆数，再加上“有余数时”的一辆。
    // 这种写法避免 people + capacity - 1 的中间值溢出。
    long long cars = people / capacity + (people % capacity != 0);
    cout << cars << '\n';
}
```

输入 `10 3` 输出 `4`；输入 `0 5` 输出 `0`。

代码解释： 商表示整车数量，非零余数意味着还要一辆车。时间、额外空间均为 $O(1)$。另一种写法是 `(a+b-1)/b`，但它多计算了一次加法。a 很大时，商加“是否有余数”的写法能避开这个中间结果。

练习： 洛谷 P5709 - Apples Prologue。

（七） 把最优化改成可行性判断

有时直接求“最小值”很难，但判断候选值是否足够更容易。先写：

$$check(x) = \text{是否存在一个满足所有限制、目标值不超过 } x \text{ 的方案}$$

如果 `check(x)` 从假到真只切换一次，就可以寻找第一个可行的 `x`。如果题目要最大可行值，并且从真到假只切换一次，则寻找最后一个可行值。

1. 完整示例：最小工作量上限

题目原文（自拟）： 有工作量依次为 `[3, 2, 4]` 的三个任务。要按原顺序分成至多 `2` 个连续非空组，使最大组内工作量尽量小。求这个最小值。

分析推导： 给定上限 `limit`，从左到右尽量向当前组添加任务；再添加会超过上限时，就开新组。工作量非负，因此当前组多容纳一个任务不会增加所需组数。

先看分组判定为什么正确，再看上限增大后发生什么。

对于固定上限，贪心把第一组取到“再加一个就超限”为止。因此，任何合法划分的第一组都不能比它更靠后结束。贪心用相同组数覆盖了更长前缀后，下一组又尽量向后延伸；工作量非负，从更靠后的起点接到另一划分的组末端，不会使组内和增大。按组数归纳，贪心不会比其他合法划分需要更多组。因此，用它算出的组数是否超过 `2`，确实能判断上限是否可行。

单调性来自同一个方案的复用：若某个划分的每组和都不超过 `limit`，把上限放大后，这个划分仍然合法。于是可行状态只会从假变真，不会再变回假。

上限	从左到右得到的组	是否至多两组
4	`[3]`、`[2]`、`[4]`	否
5	`[3,2]`、`[4]`	是
6	`[3,2]`、`[4]`	是
9	`[3,2,4]`	是

搜索范围为什么是 `[max(work), sum(work)]`？最大任务所在的组包含这个任务，所以答案至少为最大元素；把所有任务放在一组即可达到总和，所以总和一定是可行上界。每个任务贡献非负工作量，所以组内再加入一个任务，累计工作量就会增加或保持不变；这正是上面按顺序扩展分组的依据。



[3,2,4] 在上限 4 时需要三组，上限 5 时只需两组；上限继续增大后，已有的两组仍能使用。因此，要找的就是第一个能分成至多两组的上限。



以 [3,2,4] 的分组判定展示第一个 true 的搜索边界。

```cpp
#include <algorithm>
#include <iostream>
#include <numeric>
#include <vector>
using namespace std;

bool feasible(const vector<int> &work, long long limit, int maxGroups) {
    int groups = 1;
    long long current = 0;
    for (int task : work) {
        if (task > limit) {
            return false; // 一个任务已经超过上限，怎样分组都不行
        }
        if (current + task > limit) {
            ++groups; // 当前组放不下，开新组
            current = task;
        } else {
            current += task;
        }
    }
    return groups <= maxGroups;
}

int main() {
    vector<int> work{3, 2, 4};
    long long left = *max_element(work.begin(), work.end());
    long long right = accumulate(work.begin(), work.end(), 0LL);
    // 上限越大越容易分组；搜索第一个可行的上限。
    while (left < right) {
```

```cpp
        long long mid = left + (right - left) / 2;
        if (feasible(work, mid, 2)) {
            right = mid;
        } else {
            left = mid + 1;
        }
    }
    cout << left << '\n'; // 5，分成 [3, 2] 和 [4]
}
```

代码解释： `limit = 4` 时要分三组，失败；`limit = 5` 时可以分两组，成功。如果上限为 `5` 的方案可行，更大的上限也能使用这个方案，所以真假具有单调性。

这里先学习“把目标改成判断”的过程；二分更新边界的详细原理见后面的算法章节。若任务数为 `n`、工作量总和为 `S`，这个方法的时间为 $O(n\log(S+1))$，额外空间为 $O(1)$，不计输入数组。

练习： 洛谷 P1182 - 数列分段 Section II、洛谷 P1873 - 砍树。

（八） 用不变量和反例检查推导

循环不变量是循环开始或结束时始终成立的关系。它帮助我们检查“为什么当前步骤不会破坏已经得到的信息”。

- 累计和：处理完前 `i` 个元素后，累计量等于这 `i` 个元素的和。
- 连续分组：已完成的组与当前组覆盖了已经读到的全部任务，组内和都没有超过候选上限。
- 原地去重：写位置之前恰好保存已读元素中的不同值，后面的位置仍可用于继续读写。
- 检查所有元素：处理完一个前缀后，记录的真假值表示这个前缀是否每项都满足要求。

除了运行样例，还可以给自己的判断找反例。下面把几种看似合理的说法代入小数据，看看计算结果怎样：

要检查的结论	应主动尝试
“偶数一定能分成两个正偶数”	`2`
“凑钱先拿最大面值一定最优”	面值 `[1,3,4]`，金额 `6`
“两数之和可以两次使用同一个下标”	只有一个元素 `[3]`，目标 `6`
“删掉负数仍能得到最大连续区间”	`[5,-4,5]`
“统计下标对时，同值只保留一项”	`[1,1,5,5]`，目标和 `6`

可以把推导整理成四句话：**我维护什么状态；这一步为什么合法；为什么不会漏解；什么时候结束并得到答案。** 下面继续把反例、证明和实现步骤放到同一道题里，逐项检查这些关系。

练习： 洛谷 P1046 - 陶陶摘苹果、洛谷 P5717 - 三角形分类、洛谷 P1085 - 不高兴的津津。

（九） 建模与推导中的典型反例

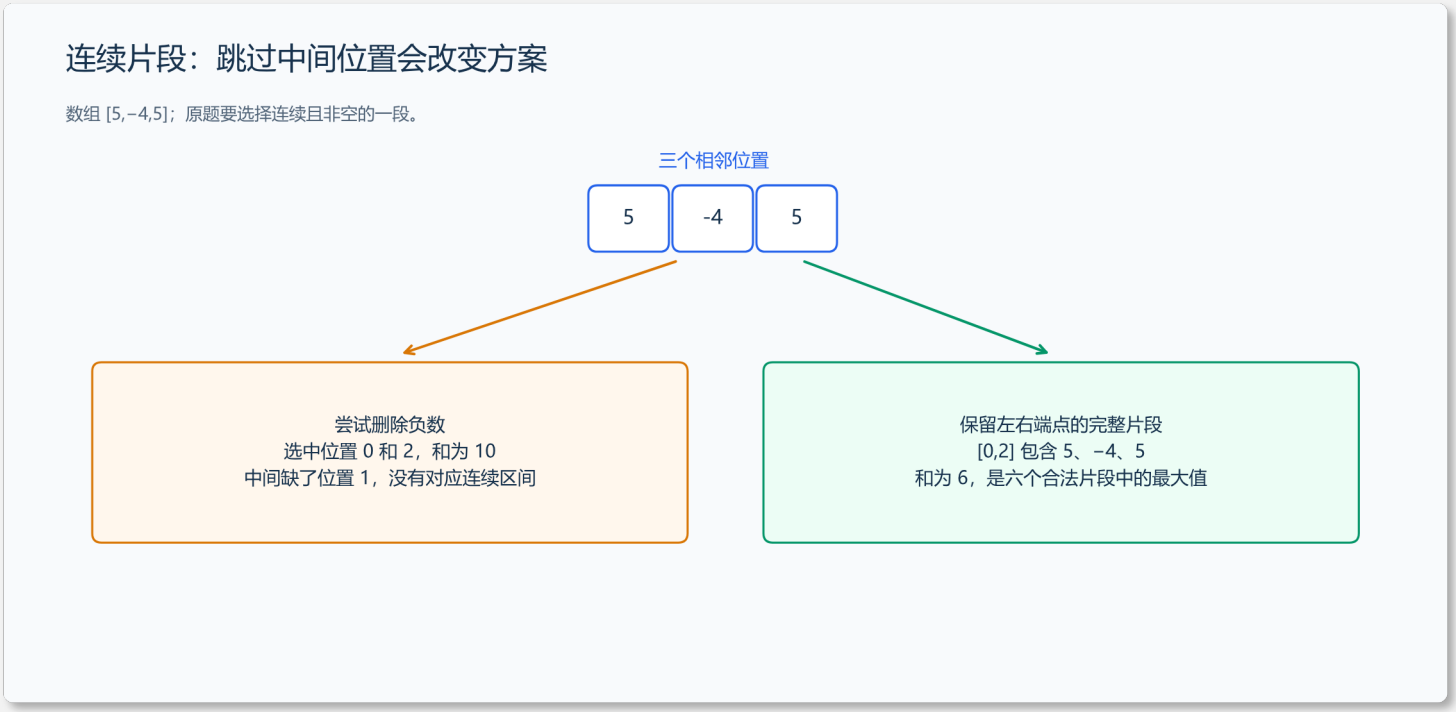
程序可以正确执行每一行，却仍然回答了另一道题。检查模型时，先把它允许的方案与题目允许的方案放在一起，找出被多算、漏算或改写的部分。

1. 连续片段被改成任意选择

题目原文（自拟）： 数组为 `[5,-4,5]`。选择连续且非空的一段，输出最大的区间和。程序无需输入。

看到负数就删除，会得到 `5+5=10`。但两个 5 中间隔着 -4，这个选择没有对应的连续区间。错误出在“允许选择哪些位置”，还没有进入算法快慢的比较。

连续区间的下标	实际选中的数	和
`[0,0]`	`5`	5
`[0,1]`	`5,-4`	1
`[0,2]`	`5,-4,5`	6
`[1,1]`	`-4`	-4
`[1,2]`	`-4,5`	1
`[2,2]`	`5`	5



用两个端点描述一个连续片段，就不会把中间位置跳过去。固定左端点后，让右端点逐个增加，每次把新元素加入当前和。

```
#include <algorithm>
#include <iostream>
```

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

int main() {
    vector<int> a{5, -4, 5};
    int answer = a[0]; // 从一个非空片段开始记录答案。
    for (int left = 0; left < static_cast<int>(a.size()); ++left) {
        int sum = 0;
        for (int right = left; right < static_cast<int>(a.size()); ++right) {
            sum += a[right]; // 此时 sum 是 a[left..right] 的和。
            answer = max(answer, sum);
        }
    }
    cout << answer << '\n'; // 6，对应整个数组。
}
```

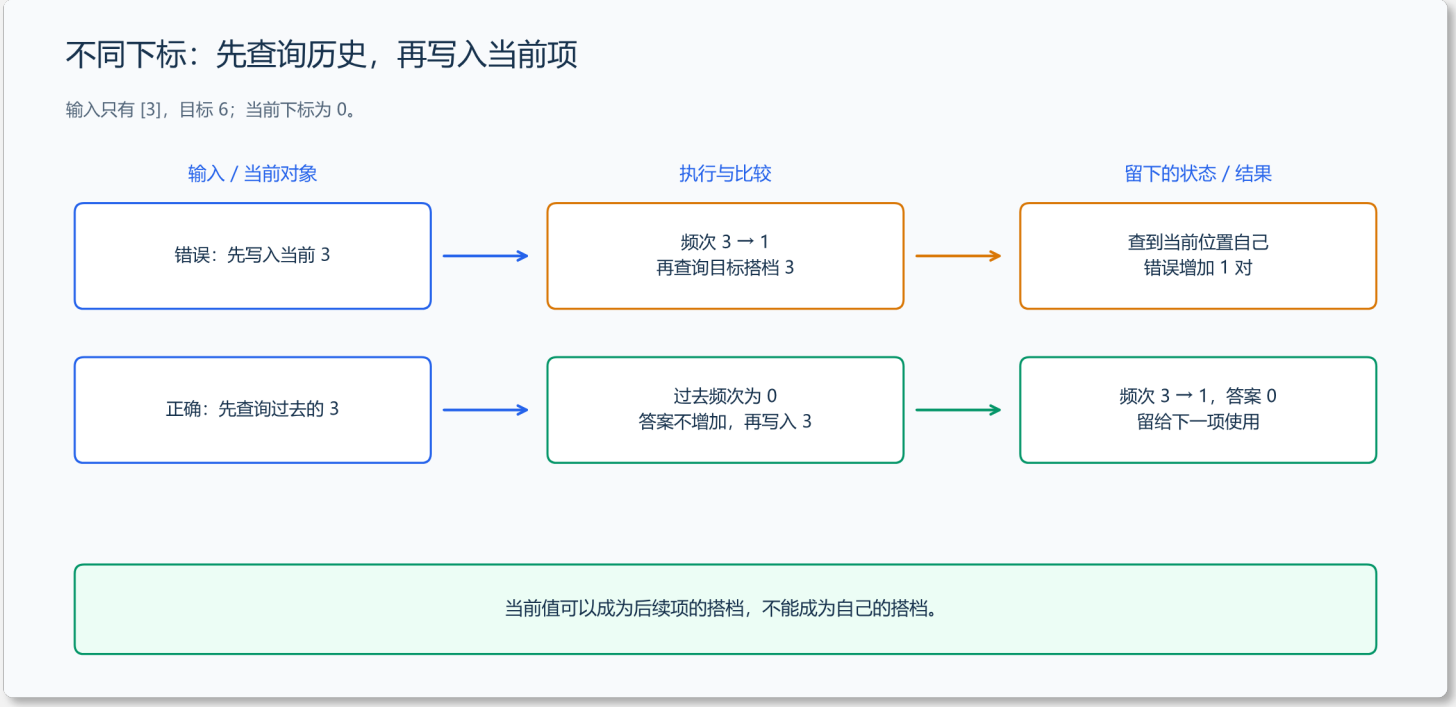
代码解释： 每个连续非空片段有唯一的一对左右端点，两个循环恰好检查全部片段。推广到长度 n 时，检查次数为 `n+(n-1)+…+1`，时间为 $O(n^2)$，额外空间为 $O(1)$。后面的线性 DP 会继续优化同一个模型。

练习： 洛谷 P1115 – 最大子段和（学习线性 DP 后完成）。

2. 两个下标被误写成两次取值

题目原文（自拟）： 数组为 `[3]`，统计满足 `i<j` 且 `a[i]+a[j]=6` 的下标对数量。程序无需输入。

“数组里有 3，另一个 3 也能找到”会让同一个位置参与两次。题目需要两个不同的位置；这个数组只有一个位置，所以答案是 0。




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

```cpp

int main() {
    vector<int> a{3};
    int count = 0;
    for (int i = 0; i < static_cast<int>(a.size()); ++i) {
        // j 从 i+1 开始，同一位置不会出现，也不会反向重复统计。
        for (int j = i + 1; j < static_cast<int>(a.size()); ++j) {
            if (a[i] + a[j] == 6) {
                ++count;
            }
        }
    }
    cout << count << '\n'; // 0，内层循环没有候选位置。
}
```

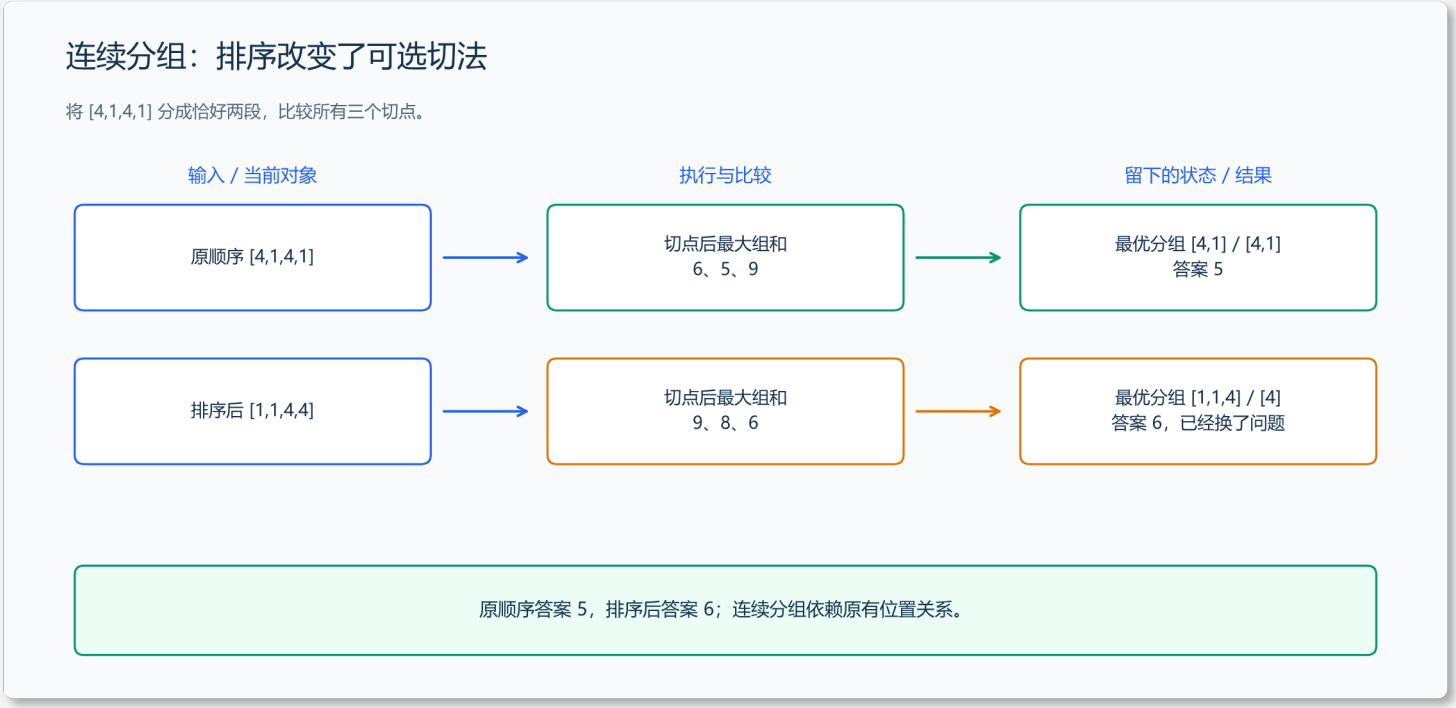
代码解释： 值相同与位置相同是两件事。`[3,3]` 有一个合法下标对，`[3,3,3]` 有三个。枚举长度 n 的数组需要 $O(n^2)$ 时间、$O(1)$ 额外空间；后面的一题多模型会保留同一个下标要求，再减少检查次数。

3. 排序改变了不能重排的对象

题目原文（自拟）： 四个任务的工作量依次为 `[4,1,4,1]`。按给定顺序切成恰好两个连续非空组，输出最大组内工作量的最小值。程序无需输入。

为了让大小相近的数放在一起而先排序，会把任务顺序改成 `[1,1,4,4]`。两个数组的可选切法不同：

切点	原顺序的两组与较大组和	排序后的两组与较大组和
第 1 项后	`[4]`、`[1,4,1]`：6	`[1]`、`[1,4,4]`：9
第 2 项后	`[4,1]`、`[4,1]`：5	`[1,1]`、`[4,4]`：8
第 3 项后	`[4,1,4]`、`[1]`：9	`[1,1,4]`、`[4]`：6



排序后的最优值 6 不能回答原题。原题只需枚举三个切点，得到答案 5。

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

int bestSplit(const vector<int> &a) {
    int total = 0;
    for (int x : a) {
        total += x;
    }
    int leftSum = 0;
    int answer = total;
    for (int cut = 1; cut < static_cast<int>(a.size()); ++cut) {
        leftSum += a[cut - 1]; // 切点前的数保持原顺序，构成第一组。
        answer = min(answer, max(leftSum, total - leftSum));
    }
    return answer;
}

int main() {
    vector<int> a{4, 1, 4, 1};
    cout << bestSplit(a) << '\n'; // 5，是原题的答案。
    sort(a.begin(), a.end());
    cout << bestSplit(a) << '\n'; // 6，展示重排后变成了另一道问题。
}
```

代码解释： 选定切点后，左组和为 `leftSum`，右组和为 `total-leftSum`。逐个比较较大的组和，再取最小值。一次分组计算需要 $O(n)$ 时间、$O(1)$ 额外空间；示例中的排序用来展示模型被改变的结果。

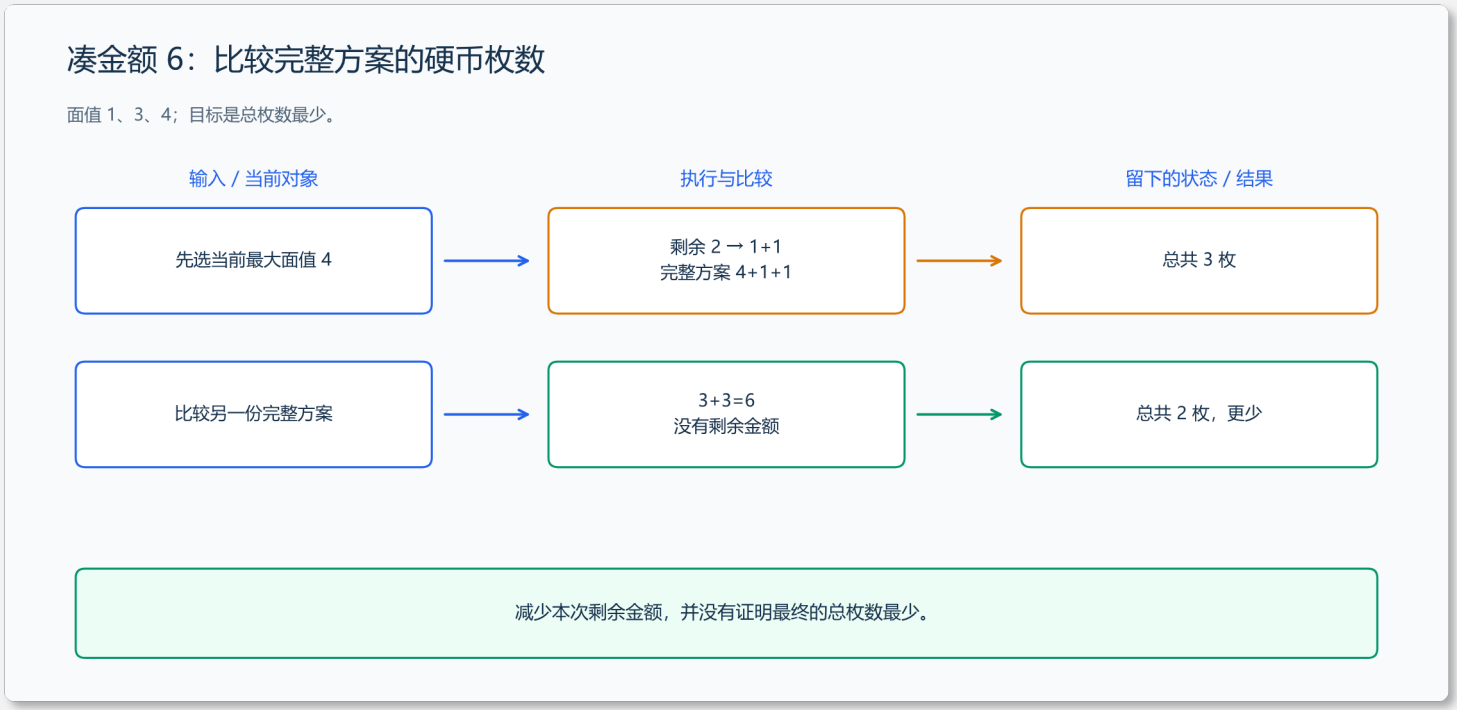
练习： 洛谷 P1182 - 数列分段 Section II（学过二分答案后完成）。

4. 当前收益被当成全局最优

题目原文（自拟）： 面值为 1、3、4 的硬币各有足够多枚。凑出金额 6，输出最少硬币数。程序无需输入。

“先拿最大面值”得到 `4+1+1`，需要三枚；`3+3` 只需两枚。选择最大硬币确实让本次剩余金额减少最多，但题目的目标是所有选择结束后的总枚数。

4 元枚数	3 元枚数	1 元枚数	总枚数
0	0	6	6
0	1	3	4
0	2	0	2
1	0	2	3



先写清模型：枚数 x、y、z 是非负整数，满足 `4*x+3*y+z=6`，要最小化 `x+y+z`。枚举 x、y 后，z 由剩余金额唯一确定。

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

int main() {
    int amount = 6;
    int answer = amount; // 全用 1 元硬币是一份可直接写出的方案。
    for (int fours = 0; fours * 4 <= amount; ++fours) {
        for (int threes = 0; fours * 4 + threes * 3 <= amount; ++threes) {
            int ones = amount - fours * 4 - threes * 3;
            // 每种面值的数量确定后，计算完整方案的枚数。
            answer = min(answer, fours + threes + ones);
```

```cpp
            }
        }
        cout << answer << '\n'; // 2，对应两个 3 元硬币。
    }
```

代码解释： 一份硬币方案只有一组对应的三种枚数，枚举没有重复。推广到金额 A 时，两个枚数的候选数量随 A 增长，时间为 $O(A^2)$，额外空间为 $O(1)$。学习完全背包后，可以把重复的剩余金额计算合并起来。

练习： CF 996A – Hit the Lottery（学习贪心后完成）。

前三个反例分别改写了合法位置、对象身份和原有顺序；硬币例题则把本次剩余金额减少最多当成了总枚数最少的理由。出现 WA 时，把失败输入代回这些具体要求，再检查每一步选择的依据。

（十） 从样例猜想到证明与构造

样例可以启发猜想，但样例中没有出现的情况也需要解释。可以先观察操作改变了什么，再寻找没有改变的量，最后尝试构造一份完整方案。

1. 完整示例：相邻两盏灯的翻转

题目原文（自拟）： 有 n 盏排成一行的灯，`0` 表示关闭，`1` 表示点亮。一次操作选择相邻两盏灯，同时翻转它们的状态。判断能否把所有灯关闭。能做到时，输出 `YES`，下一行输出操作次数，第三行输出每次操作的左侧灯编号；做不到时只输出 `NO`。灯从 1 编号。第一行输入整数 n，$1 \le n \le 200000$；第二行输入一个长度为 n 的 01 字符串。

样例输入：

```text
5
10111
```

样例输出：

```text
YES
3
1 2 4
```

先试 `11→00`，再试 `10→01`。一次翻转让亮灯数量减少 2、增加 2 或保持不变，因此亮灯数量的奇偶性始终不变。全部关闭时亮灯数量为 0，奇数盏亮灯无法变成这个结果。

亮灯数为偶数就一定能做到吗？从左向右处理：当前灯已经关闭就前进；当前灯亮着，就翻转它与右邻灯。这样当前灯变成 0，以后只操作右边的位置，不再碰它。



样例依次变为 `10111→01111→00011→00000`，操作左侧编号为 1、2、4。处理完前 n−1 盏后，只剩最后一盏可能亮着；此时亮灯数仍为偶数，因此最后一盏也只能是 0。这同时给出了充分性证明和实际操作方案。

我们得到等价关系：能全部关闭，当且仅当初始亮灯数为偶数。 奇偶性解释做不到的原因，逐项关闭的构造解释做得到的原因，两部分合起来才覆盖所有输入。

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

int main() {
    int n;
    string lights;
    cin >> n >> lights;
    int ones = 0;
    for (char state : lights) {
        ones += state == '1'; // 每盏亮灯贡献 1，先检查奇偶性。
    }
    if (ones % 2 != 0) {
        cout << "NO\n";
        return 0;
    }
    vector<int> operations;
    for (int i = 0; i + 1 < n; ++i) {
        if (lights[i] == '1') {
            lights[i] = '0'; // 翻转后锁定当前位置，后续不再碰它。
            lights[i + 1] = lights[i + 1] == '0' ? '1' : '0';
            operations.push_back(i + 1); // 输出使用题目的 1 起始编号。
        }
```

```cpp
    }
    cout << "YES\n" << operations.size() << '\n';
    for (int i = 0; i < static_cast<int>(operations.size()); ++i) {
        if (i > 0) cout << ' ';
        cout << operations[i];
    }
    cout << '\n'; // 没有操作时，这一行为空。
}
```

代码解释： 前面的计数得到判定，后面的循环把充分性证明直接写成程序。`0` 得到 `YES`、0 次操作和空行；`1` 得到 `NO`；`10` 得到 `NO`，而不是因为“至少有两盏”就判定成功。每个位置只处理一次，时间为 $O(n)$，保存方案需要 $O(n)$ 空间。

这里用到两种证明方法：从操作前后的变化得到**不变量**，从目标反推可执行步骤得到**构造**。后面遇到交换、翻转或移动题，也可以先列出一次操作改变了哪些量。

练习： CF 4A - Watermelon（分别写出必要性与一个实际构造）；洛谷 P5717 - 三角形分类（区分存在三角形与三角形的类型）。

（十一） 从输出目标确定状态信息

看到输入后，不必把全部历史都搬进状态。先看未来的判断和最终输出会读到哪些信息。

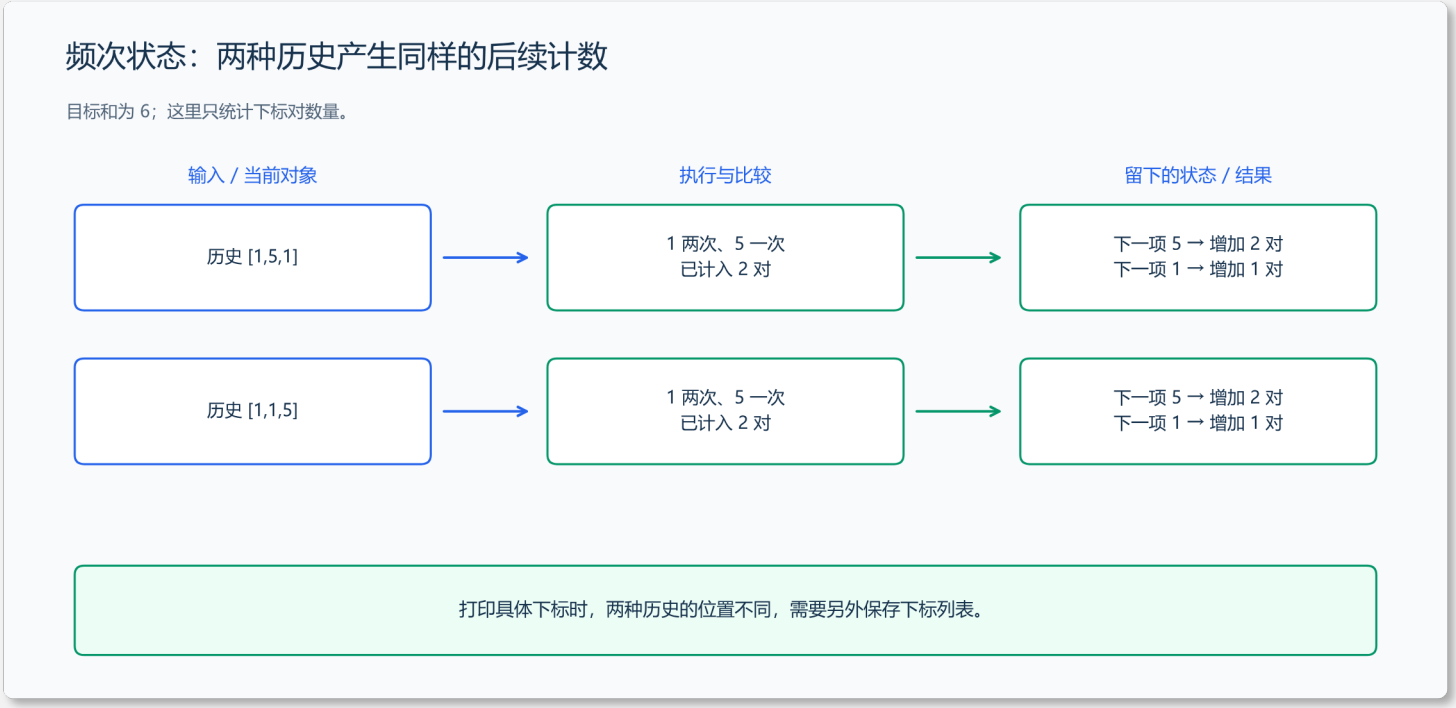
以 `[3,3,3]`、目标和 6 为例，同一组数据可以提出四种问题：

输出要求	结果	计算时需要保留的信息
是否存在下标对	YES	过去是否出现过需要的值
下标对数量	3	过去每个值出现多少次，以及已累计的答案
不同取值组合数量	1	哪些取值组合已经计入
输出全部下标对	`(0,1)`、`(0,2)`、`(1,2)`	每个值对应的具体下标

把频次换成一个 bool，就能判断存在性，却会丢失计数信息；只保留频次，也无法还原具体下标。同样，求最短距离与输出实际路线，对状态的要求不同，后面的 BFS 会在距离之外保存父节点。

1. 依据后续行为合并历史

目标和为 6 时，已经读过 `[1,5,1]` 或 `[1,1,5]`，都会得到“1 出现两次、5 出现一次，已找到两对”。下一个数若为 5，两个历史都增加两对；若为 1，都增加一对。后续计数只会查询值的频次，历史内部的排列不再影响更新。



如果目标改为打印下标对，这两种历史的下标列表不同，就要继续保存具体位置。**状态能否合并，取决于题目要保留哪些结果。**

2. 根据未来的选择检查遗漏信息

在后面的服务窗口例题中，只知道“有两个窗口”无法决定把顾客交给谁，还需要知道它们是否空闲、何时结束，以及窗口编号。两份安排即使当前时间相同，窗口的结束时间不同，下一位顾客的开始时间也可能不同。

检查状态时可以做一个实际比较：找两段会被你记成同一个状态的历史，接上同一份后续输入。如果它们需要采取不同动作，或者最终答案不同，就还有信息被遗漏。若后续只读取同一份统计量，就可以围绕这份统计量实现程序。

这也是搜索逐渐走向记忆化和 DP 的起点：先确认一个子问题需要哪些信息，再让具有同样信息的计算共享结果。具体的缓存和状态转移会在搜索与 DP 中展开。

（十二） 一题多建模与最优方法的选择过程

面对同一道题，先提出几种表示和计算方法，再判断它们是否计算同一份合法方案。下面保留一个数对计数目标，依次构建枚举下标、排序分组、过去频次三种模型，随后代入规模逐项筛选，说明最后采用哪种方法。

1. 完整题面与下标对模型

题目原文（自拟）： 给定 n 个整数和目标 K，统计满足 $0 \le i < j < n$ 且 `a[i]+a[j]=K` 的下标对数量。不同下标对分别计数，即使取值相同。第一行输入 n、K，第二行输入 n 个整数；$1 \le n \le 200000$，$-10^9 \le a[i] \le 10^9$，$-2 \times 10^9 \le K \le 2 \times 10^9$。输出一行计数结果。

样例输入：

```text
6 6
1 5 3 3 5 1
```

样例输出：

```text
5
```

样例解释： 五个下标对为 `(0,1)、(0,4)、(1,5)、(2,3)、(4,5)`。其中四对由 1 与 5 组成，一对由两个 3 组成。

先把所有 `i<j` 的候选都列出来，再检查和。下面的函数需要 `<vector>`，参数是输入数组和目标值；可以单独与本节最后的调用程序组合。

```cpp
long long countByEnumeration(const vector<long long> &a, long long target) {
    long long answer = 0;
    int n = static_cast<int>(a.size());
    for (int i = 0; i < n; ++i) {
        for (int j = i + 1; j < n; ++j) {
            // 每个不同位置的无序下标对恰好检查一次。
            answer += a[i] + a[j] == target;
        }
    }
    return answer;
}
```

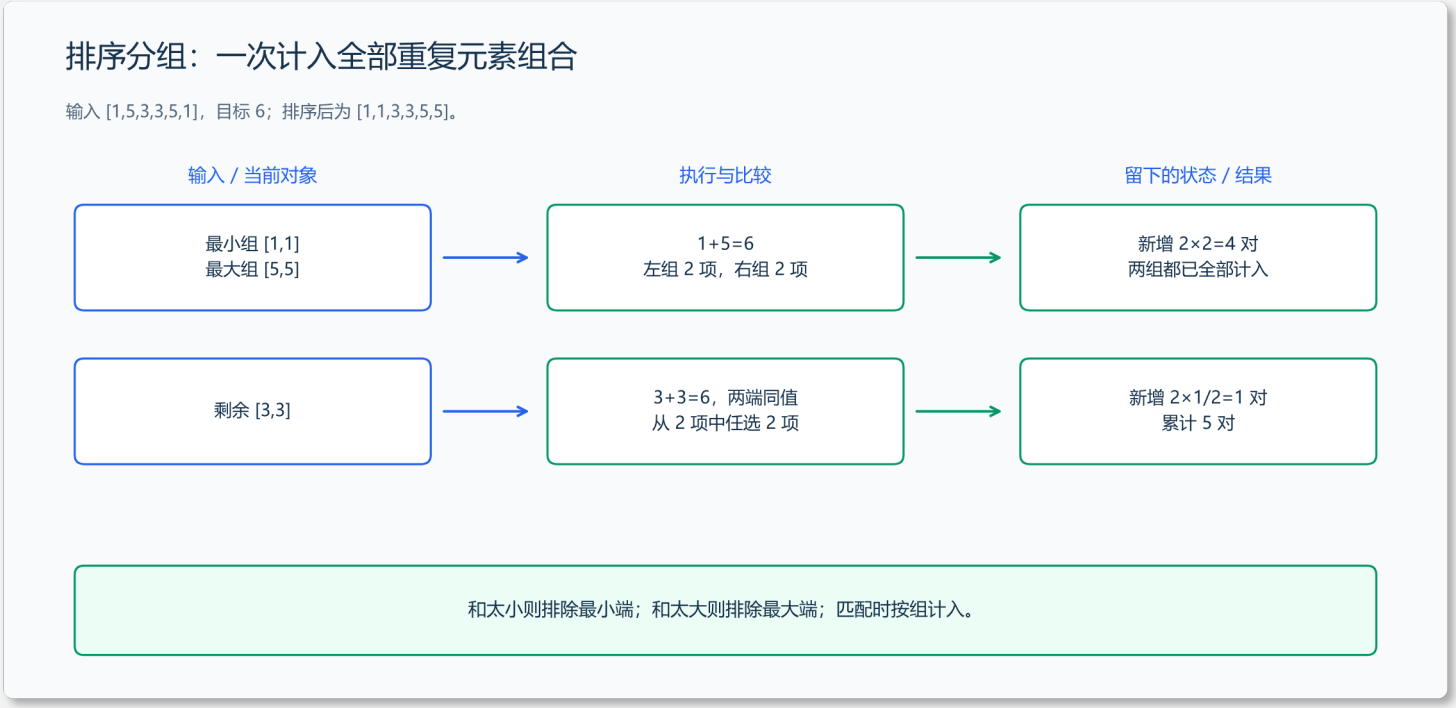
分析： n 个位置有 `n*(n-1)/2` 对，时间为 $O(n^2)$，额外空间为 $O(1)$。n 为 200000 时，要检查 19999900000 对。模型清楚，适合手算和核对小数据，但这里仍有大量重复的取值比较。

2. 排序后按相同取值成批计数

这道题只求数量，不输出原下标。排序把位置重新排列，但每个元素仍作为一个独立元素保留；排序前后的两元素组合存在一一对应，和也没有变化，因此数量不变。这与“原顺序连续分组”的反例不同：那里排序改变了哪些组合法。

排序后得到 `[1,1,3,3,5,5]`。让 left 指向最小值，right 指向最大值：

- 和小于 K：当前最小值与剩余最大的值相加仍不够，换其他更小的搭档也不够，因此跳过这个最小值。
- 和大于 K：当前最大值与剩余最小的值相加仍偏大，换其他更大的搭档也偏大，因此跳过这个最大值。
- 和等于 K：两端取值不同，就统计两端各有多少项，将数量相乘；两端取值相同，就从剩余项中任选两项。



样例中两项 1 与两项 5 贡献 `2×2=4` 对；剩下两项 3 贡献 `2×1/2=1` 对，总计 5。这里的 `m*(m-1)/2` 来自：第一项搭配 m-1 项，第二项搭配 m-2 项，直到最后没有后续项。

下面的函数需要 `<algorithm>` 、 `<vector>` 。数组按值传参，因此排序的是副本，调用者的数组可以继续用于其他方法。

```cpp
long long countBySorting(vector<long long> a, long long target) {
    sort(a.begin(), a.end()); // 将同值元素放到一起，便于成批计算。
    int left = 0, right = static_cast<int>(a.size()) - 1;
    long long answer = 0;
    while (left < right) {
        long long sum = a[left] + a[right];
        if (sum < target) {
            ++left; // 当前最小值找不到足够大的搭档。
        } else if (sum > target) {
            --right; // 当前最大值找不到足够小的搭档。
        } else if (a[left] == a[right]) {
            long long length = right - left + 1;
            answer += length * (length - 1) / 2; // 全部剩余项任选两项。
            break;
        } else {
            long long leftCount = 1, rightCount = 1;
            while (left + leftCount ≤ right && a[left + leftCount] == a[left]) {
                ++leftCount;
            }
            while (right - rightCount ≥ left && a[right - rightCount] == a[right]) {
                ++rightCount;
            }
            answer += leftCount * rightCount; // 每项左值都能搭配每项右值。
            left += static_cast<int>(leftCount);
            right -= static_cast<int>(rightCount);
        }
    }
    return answer;
```

```
29  }                                                                    cpp
```

代码解释： 每次移动排除的是已经证明不可能配成目标和的一端，匹配时则一次计入整组，不会丢掉重复元素。排序需要 $O(n \log n)$ 时间，扫描时每个位置只被跳过一次，合起来仍为 $O(n \log n)$。本实现复制数组，额外空间为 $O(n)$。

3. 固定后来的位置，查询过去的频次

还可以从公式 `a[i]+a[j]=K` 得到 `a[i]=K-a[j]`。固定后来的位置 `j`，就只需知道过去有多少项等于 `K-a[j]`，不必逐个检查过去的下标。

用 `frequency[value]` 保存已经处理过的值的次数。先查询当前值需要的搭档，再把当前值加入表中；这样表里的位置都小于 `j`，自动满足 `i<j`。

j	当前值	需要的值	查询到的过去次数	新增对数	累计答案
0	1	5	0	0	0
1	5	1	1	1	1
2	3	3	0	0	1
3	3	3	1	1	2
4	5	1	1	1	3
5	1	5	2	2	5



正确性推导： 读到 `j` 时，每个已出现的搭档位置都小于 `j`，且值满足目标和，因此新增的配对全部合法。任一合法对 `(i,j)` 会在处理 `j` 时找到 `i`，不会漏掉；它也不会在其他步骤再次出现，因为较后的那个位置只有一个。

下面的函数需要 `<unordered_map>`、`<vector>`。`find` 查询频次而不创建新键；`++frequency[x]` 将当前值的次数加一。`reserve(a.size())` 为预计的不同键数量预留桶，元素仍由后面的写入创建。

```cpp
long long countByFrequency(const vector<long long> &a, long long target) {
    unordered_map<long long, long long> frequency;
    frequency.reserve(a.size()); // 提前准备桶，减少逐步扩容的次数。
    long long answer = 0;
    for (long long x : a) {
        auto it = frequency.find(target - x);
        if (it != frequency.end()) {
            answer += it->second; // 每个过去的同值位置贡献一对。
        }
        ++frequency[x]; // 查询之后才写入，当前项不会与自己配对。
    }
    return answer;
}
```

代码解释： 处理完前 j 项后，表中恰好是这 j 项的频次，answer 恰好是两端都在这 j 项里的合法下标对数。一次查询与一次写入就把这个关系推进到下一项。

哈希把键分散到桶中，平均查询与写入时间为 $O(1)$，因此这个方法的平均总时间为 $O(n)$，额外空间为 $O(n)$。多个键落入同一桶时，需要继续比较这些键，碰撞集中会增加查找工作；使用 map 保存同一份频次时，总时间为 $O(n \log n)$。

4. 从候选模型逐步筛选到最终方案

同一题的几个正确做法，不是把复杂度排个名就完成选择。要先核对它们是否仍在计算题目要求的对象，再把题面的规模代入，最后比较留下的信息和每步的工作。

4.1 先核对合法方案和输出目标

本题计算的是**不同位置组成的下标对数量**。用样例中的两个 1、两个 5 检查候选模型：它们组成四对，而不是只组成一个取值组合。

候选模型	保留的信息	核对结果
枚举所有 `i<j`	每个元素的位置和值	每对检查一次，得到 5
只用 set 保存出现过的值	出现过 1、3、5	丢掉重复次数，不能算出四对 1 与 5，排除
排序后按组计数	每项元素和同值组大小	保留元素个数，用组大小相乘，得到 5
保存过去频次	每个值在过去出现的次数	每对在较后位置处计入，得到 5

排序与频次都去掉了部分位置信息，但没有去掉计数需要的信息。这里每一对的较早、较后位置只用来避免重复计数；输出不要求列出它们的具体编号。

4.2 把最大输入代入工作量

取题面中的 `n=200000`，不要只写一个 O 符号：

方法	主要工作量的估算	本实现的额外空间
枚举下标对	$n(n-1)/2 = 19999900000$ 次配对检查	$O(1)$
排序分组	$n \log_2 n \approx 3521928$ 的排序比较量级，再扫描 n 项	$O(n)$，用于数组副本
map 频次查询	n 次查询、n 次写入，每次沿树查找约 $\log_2 n \approx 18$ 层	$O(n)$，保存实际出现的键
哈希频次查询	n 次查询、n 次写入；平均一次只需少量桶内比较	$O(n)$，保存实际出现的键

这些数表示工作量，不能直接当成秒数。枚举要重复检查约 200 亿对；后三种方法把重复配对合并了。枚举函数仍可用于小数据对拍，但提交时不采用它。

再比较频次的存储方法。把值平移后直接用作数组下标，需要为 -10^9 到 10^9 的 2000000001 个值准备位置。每项计数用 8 字节，合计 16000000008 字节，约 14.9 GiB；输入却最多只有 200000 项。map 和哈希表只保存实际出现的键，避免给大量没有出现的值分配计数位置。因此，直接按完整值域开数组也被排除。

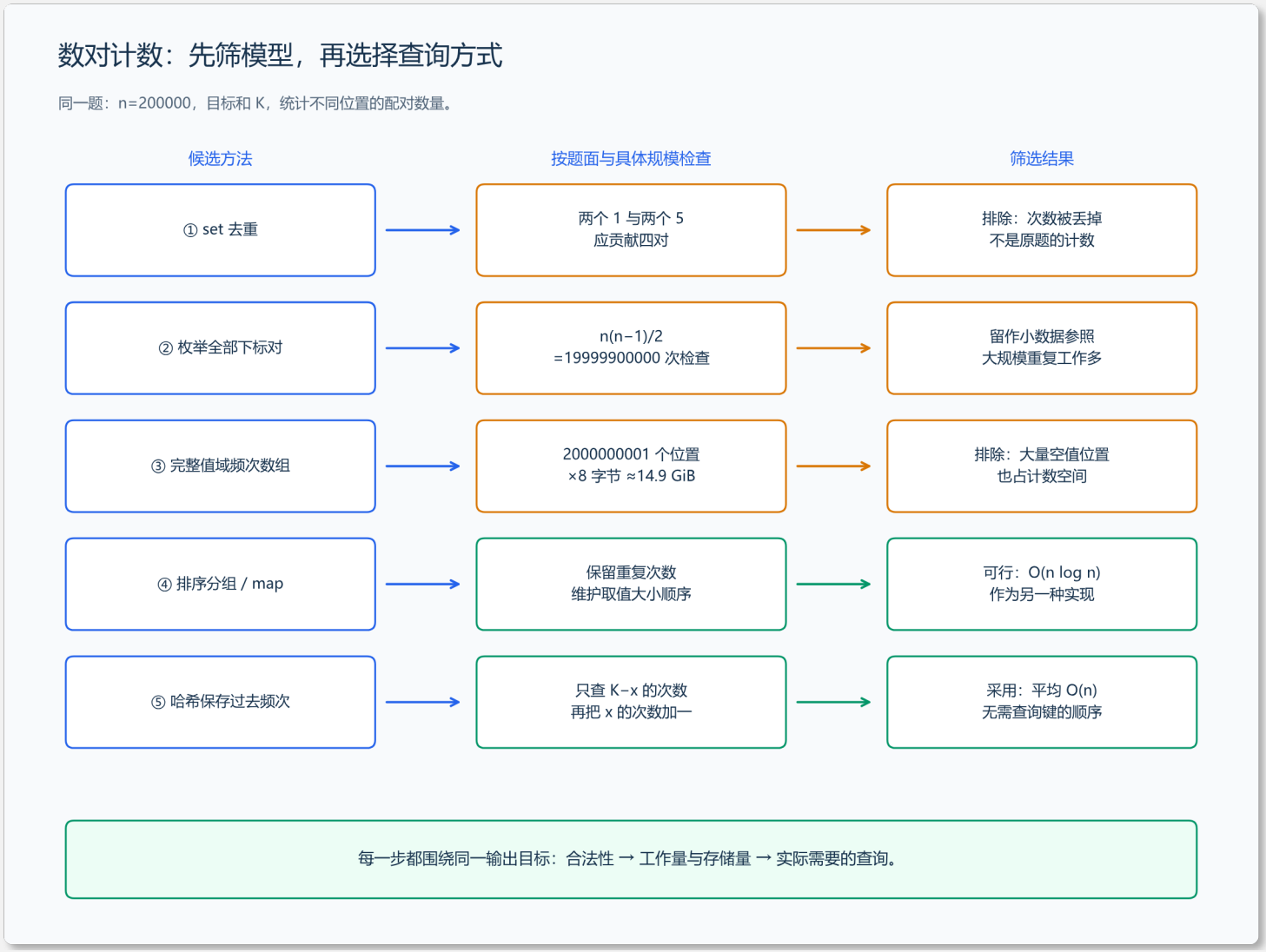
4.3 根据每一步需要的查询作出选择

剩下排序分组、map 频次和哈希频次，再回到每步的任务：读到 x，只查询“过去有多少个 `K-x`”，随后把 x 的频次加一。

1. 不查询最小值、最大值或某个取值区间，也不按键顺序输出，因此不用维护键的大小顺序。
2. 不必等全部输入排好序；读到一项就能确定它与过去的配对数，因此可以逐项查询频次。
3. 输出只有一个总数，不恢复下标列表，因此每个值保存一个次数就够了。

据此选择 **unordered_map 保存过去频次**：平均时间为 $O(n)$，空间为 $O(n)$。这里从“需要的查询”得到了容器选择，而不是看到计数题就先套哈希表。

哈希查询要在桶内核对键，碰撞集中时会增加工作。排序分组和 map 频次提供 $O(n \log n)$ 的另一种实现；排序分组已经写在前面，可以与哈希结果互相核对。



本题的选择结论： set 丢掉重复次数，首先排除；枚举保留全部信息但约有 200 亿次检查，留作小数据参照；完整值域数组约需 14.9 GiB，排除；排序分组和 map 可行，但维护了本题不查询的取值顺序，最后采用哈希频次。

4.4 沿着同一问题变化重新选择方法

方法选择依赖当前要做的事。把这道题稍作变化，选择过程也要重新走一遍：

具体任务	要增加或减少的信息	相应选择
只有 30 项，用来检查另一个程序	全部只有 435 个候选对	枚举最直接，适合作为参照答案
输入逐项到来，每读一项就输出当前配对总数	需要立刻查询过去，不能等待最后排序	逐项维护频次
取值为 0 到 1000	频次只需 1001 个位置，8 字节计数约占 7.82 KiB	直接用频次数组，免去键查找
输入已经有序，只输出最后总数	已经有取值顺序，不必再排序	双指针按组计数，时间 $O(n)$
要输出每一对的原下标	单个次数不能恢复具体位置	保存每个值对应的过去下标列表；输出 P 对本身就需要 $O(P)$ 时间

最后一行也说明：“最优方法”要围绕输出目标讨论。若题目要打印约 200 亿对，光输出就有这么多工作；不能把只统计数量的程序直接拿来回答。

5. 从所选模型组装提交程序

计数结果可能达到 `n*(n-1)/2`；200000 项全能互相配对时，结果为 19999900000，所以 answer 使用 long long。每项输入能放进 int，累计答案仍可能需要更大的类型。

完整调用程序如下，与 `countByFrequency` 放在同一个文件中；头文件置于该函数之前。替换调用的函数名，就能比较其他两个模型，使用它们时同时加入 `<algorithm>`。

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

// 在这里放置上面的 countByFrequency 函数定义。

int main() {
    int n;
    long long target;
    cin >> n >> target;
    vector<long long> a(n);
    for (auto &x : a) {
        cin >> x; // 每个位置保留一个输入元素，重复值仍分别计数。
    }
    cout << countByFrequency(a, target) << '\n';
```

```cpp
17  }
```

练习： 洛谷 P1102 - A-B 数对、CF 702B - Powers of Two。

（十三） 复杂题面到事件模拟的完整推导

复杂题目常把“谁先处理、当前能做什么、同一时刻先执行哪个动作”藏在几段文字中。先把规则逐项落实，再决定需要哪些容器。

1. 完整示例：多个服务窗口的排队

题目原文（自拟）： 服务大厅有 `m` 个窗口，编号为 `1` 到 `m`，时刻 `0` 都空闲。`n` 位顾客按输入顺序排队，第 `i` 位顾客在时刻 `arrival[i]` 到达，办理业务需要 `duration[i]` 个时间单位。

大厅按以下规则安排：

1. 到达时间按非递减顺序给出；同时到达的顾客按输入顺序排队。后面的顾客不能越过仍在等待的前面顾客。
2. 排在最前面的顾客一旦有空闲窗口就立即开始；多个窗口空闲时，选择编号最小的窗口。
3. 没有空闲窗口时，顾客继续等待。某窗口在时刻 `t` 结束办理，就能在同一时刻 `t` 接待下一位顾客。
4. 一位顾客办理期间占用一个窗口，不能中断或更换窗口。结束时刻等于开始时刻加办理时长。

第一行输入 `m`、`n`，接下来 `n` 行各输入到达时刻和办理时长。$1 \le m, n \le 200000$，$0 \le arrival[i] \le 10^9$，$1 \le duration[i] \le 10^6$。按输入顺序，每位顾客输出一行“窗口编号 开始时刻 结束时刻”；最后输出一行“总等待时间 最大等待时间”。

样例输入：

```text
2 7
0 6
0 2
1 1
1 1
2 1
7 2
7 1
```

样例输出：

```text
1  1 0 6
2  2 0 2
3  2 2 3
4  2 3 4
5  2 4 5
6  1 7 9
7  2 7 8
8  5 2
```

2. 提取对象、时间与动作先后

题目中的大厅装饰和业务名称不影响安排；每位顾客需要到达时刻、办理时长和排队次序。每个窗口需要编号、是否空闲，以及忙碌时的结束时刻。

文字规则	形式化关系	要执行的动作
先到先服务，同时到达按输入顺序	顾客依次处理，当前顾客开始后再安排下一位	用一个顺序循环处理输入
顾客已经到达才能开始	$start \geq arrival$	当前安排时刻至少推进到 arrival
结束时刻就能接待下一人	$finish \leq now$ 的窗口已经空闲	选择窗口之前，释放全部已完成窗口
空闲时选择编号最小的窗口	在空闲编号中取最小值	按编号维护小根堆
全部忙碌时等待下一次完成	推进到最小的 finish	按结束时刻维护忙碌小根堆
总等待与最大等待	$wait=start-arrival$	累加 wait，并更新最大值

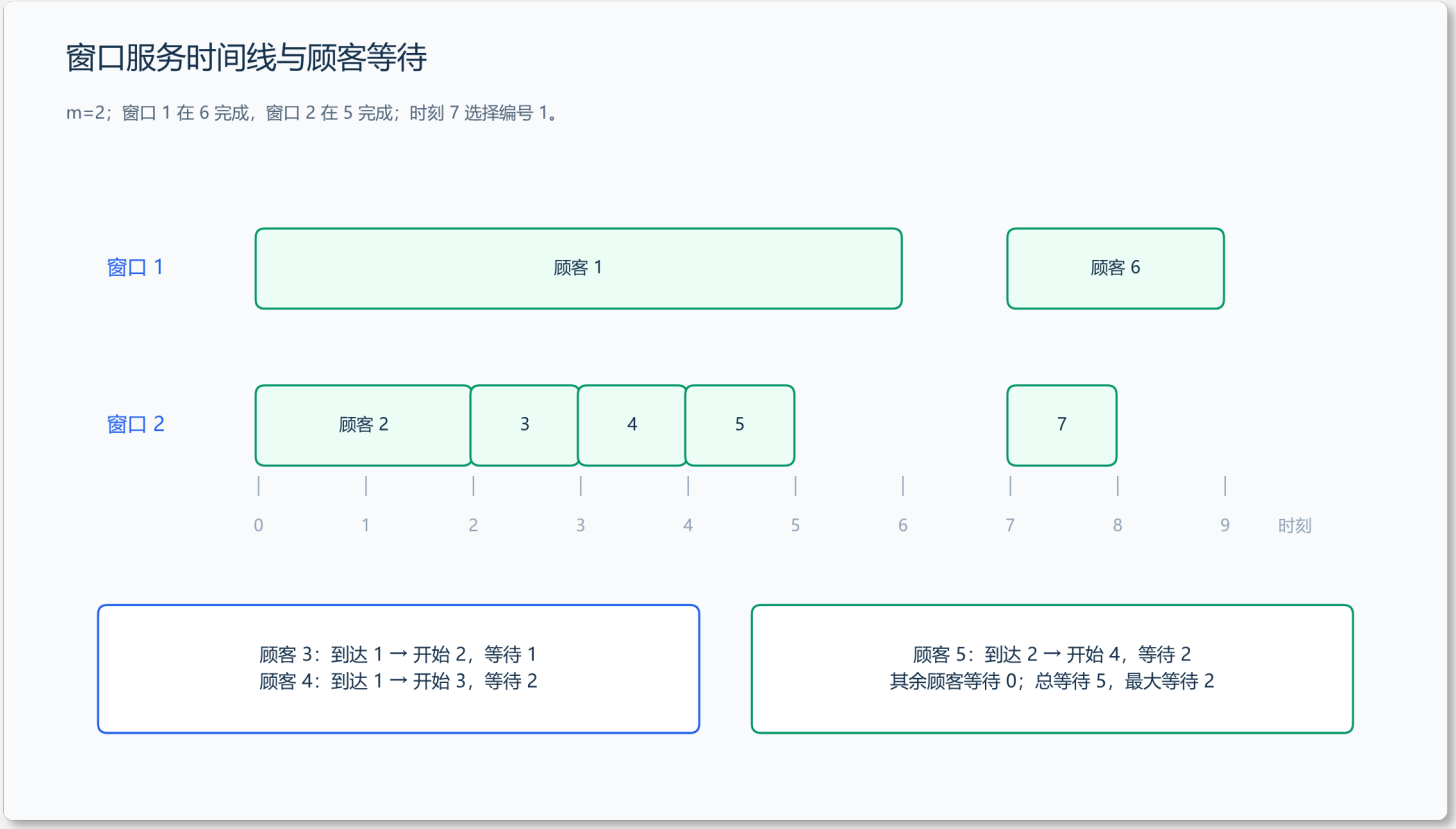
不能先排序办理时长来“减少等待”，因为那会改变题目已经指定的排队次序。本题让我们执行给定安排规则，两个等待统计量是安排完成后的结果。

3. 从逐时刻检查改成只处理事件

最直观的程序每个时间单位检查窗口。但两次到达或完成之间，窗口状态没有变化，反复检查会得到相同结果。一次业务持续 10^6 个时间单位时，就会有大量这样的重复检查。

因此只在顾客到达或业务完成时处理变化。用 now 保存“正在给当前顾客安排业务的时刻”。它至少是当前顾客的到达时刻；前一位已经等待到更晚才开始时，后一位也不能倒回过去抢先开始，所以取 $now=max(now,arrival)$。

先释放所有 $finish \leq now$ 的窗口，再选择空闲窗口。如果全部忙碌，把 now 推进到最早的结束时刻，重新释放，随后就能安排当前顾客。无须逐秒走过中间的空白时间。



顾客 3 到达时两个窗口都忙，等到时刻 2；顾客 4 仍排在后面，等到时刻 3。到时刻 7，两个窗口都已空闲，顾客 6 选择编号 1，即使窗口 2 更早在时刻 5 就结束了。

4. 三种调度模型的比较与选择

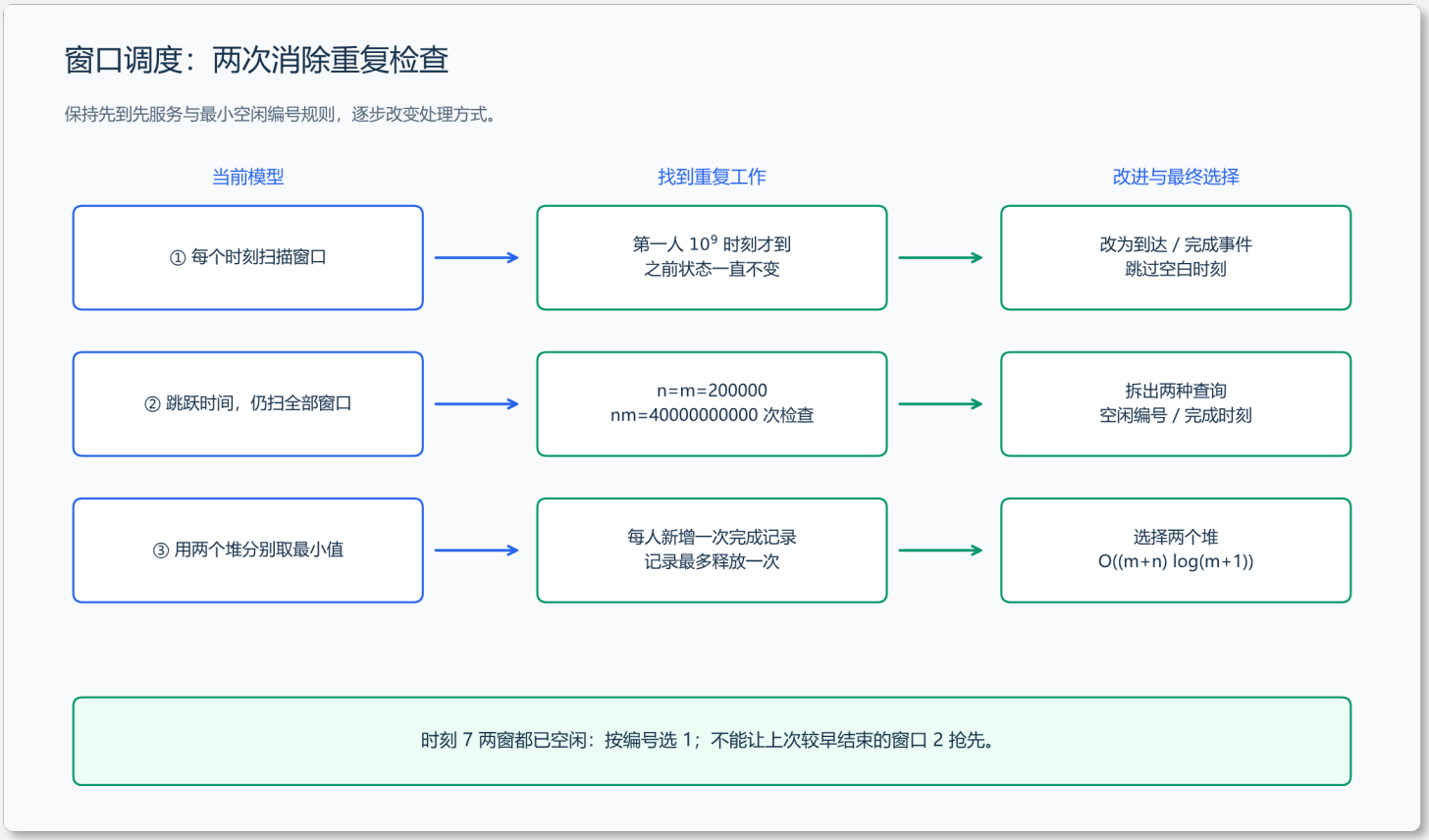
窗口题也能用几种方法完成。沿用同一题面和同一份顾客数据，逐步减少查找工作，不能把“选最小空闲编号”改成“选上次最早结束的窗口”。

候选模型	时间怎样前进	每次怎样选窗口	主要工作
逐时刻扫描	每次加一	扫描 m 个窗口，按编号找空闲窗口	到最后安排时刻 T，约 $O(Tm+n)$
事件跳跃加数组扫描	没有空闲窗口时跳到最早完成时刻	扫描全部窗口，找最小空闲编号或最早完成时刻	每位顾客扫描窗口，$O(nm)$
事件跳跃加两个堆	同样跳到到达或完成时刻	空闲堆找编号，忙碌堆找完成时刻	$O((m+n)\log(m+1))$

第一步，去掉无变化的时刻。 假如第一位顾客在时刻 10^9 到达，之前逐秒检查不会产生安排；m 为 200000 时，约有 2×10^{14} 次窗口检查。直接令 now 到达顾客的 arrival，就能跳过这些重复判断。

第二步，去掉每位顾客对全部窗口的扫描。 跳跃时间后，若仍对每位顾客扫描 m 个窗口，n、m 同为 200000 时就是 4×10^{10} 次窗口检查。真正需要的只有两个极小值：空闲编号的最小值、忙碌结束时刻的最小值。

第三步，分别维护这两种顺序。 两个堆让每次取最小值都发生在堆顶；插入和删除沿约 $\log_2 m$ 层更新。m 为 200000 时，这个量级约为 18 层。每位顾客增加一份完成记录，每份记录最多释放一次，没有按时间长度或顾客数重复扫描全体窗口。



因此本题选择“事件跳跃 + 空闲编号堆 + 忙碌完成堆”。逐时刻模拟可以核对很短的手算数据；数组扫描可以作为小数据参照；两个堆用于完整规模。它们沿同一规则得到同一安排，优化的是重复检查次数。

5. 用两份顺序保存不同的选择依据

空闲时比较窗口编号，等待时比较结束时刻，这是两种不同的顺序。只用“上次结束时刻最小”的一个堆，会在样例时刻 7 错选窗口 2：它上次在 5 结束，窗口 1 在 6 结束，但现在两者都空闲，题目要求选编号 1。

因此维护两份集合：

- `freeWindows`：保存空闲窗口编号，堆顶是最小编号。
- `busyWindows`：保存 `(结束时刻,窗口编号)`，堆顶是最早结束的记录。

`pair` 先比较 first，再比较 second；`greater` 让 priority_queue 成为小根堆。完成时间相同时会先取出编号较小的记录，但释放过程会取出所有已完成窗口，然后才由空闲堆决定接待编号。



不能只释放一条记录再选窗口。例如两个窗口都在时刻 5 完成，当前顾客就在时刻 5 开始；全部释放后才能确定当前空闲的最小编号。

6. 把动作拆成可组合的函数

下面三段代码依次放入同一个源文件。第一段包含头文件、类型别名和释放函数；第二段实现一位顾客的安排；第三段初始化窗口、读取输入并输出结果，只有第三段包含 main。

`<queue>` 提供 priority_queue，`<functional>` 提供 greater，`<utility>` 提供 pair；`<vector>` 是堆的底层容器，`<algorithm>` 提供 max，`<iostream>` 提供输入输出。这些都是语法基础中已经使用过的接口。

```
#include <algorithm>
#include <functional>
#include <iostream>
#include <queue>
#include <utility>
#include <vector>
using namespace std;

using FinishEvent = pair<long long, int>;
using BusyHeap = priority_queue<FinishEvent, vector<FinishEvent>, greater<FinishEvent>>;
using FreeHeap = priority_queue<int, vector<int>, greater<int>>;

void releaseFinished(long long now, BusyHeap &busy, FreeHeap &freeWindows) {
```

```cpp
    while (!busy.empty() && busy.top().first ≤ now) {
        // 先把本时刻全部已完成窗口变为空闲，再决定接待顺序。
        freeWindows.push(busy.top().second);
        busy.pop();
    }
}
```

释放函数解释： 每弹出一个忙碌记录，就把对应编号加入空闲堆。这是一份窗口状态从“忙碌”到“空闲”的转移，不是新增窗口。函数结束后，忙碌堆里的结束时刻全部大于 now。

```cpp
pair<long long, int> serveCustomer(long long arrival, long long duration,
                                   long long &now, BusyHeap &busy, FreeHeap &freeWindows) {
    now = max(now, arrival); // 顾客到达且前面的顾客已经开始，才轮到本顾客。
    releaseFinished(now, busy, freeWindows);
    if (freeWindows.empty()) {
        now = busy.top().first; // 全部忙碌，直接跳到下一次完成事件。
        releaseFinished(now, busy, freeWindows);
    }
    int window = freeWindows.top(); // 此时按编号选择，而不是按旧结束时刻。
    freeWindows.pop();
    busy.push({now + duration, window}); // 接待后转入忙碌集合。
    return {now, window}; // first 是开始时刻，second 是窗口编号。
}
```

安排函数解释： 当前顾客能开始前，只可能被两件事阻止：自己还没到达，或所有窗口仍忙。前一个由 max 处理，后一个由最早完成事件处理。获得空闲窗口后立即开始，正好对应题目的安排规则。

```cpp
int main() {
    int m, n;
    cin >> m >> n;
    BusyHeap busy;
    FreeHeap freeWindows;
    for (int window = 1; window ≤ m; ++window) {
        freeWindows.push(window); // 时刻 0 所有窗口空闲。
    }
    long long now = 0, totalWait = 0, maxWait = 0;
    for (int i = 0; i < n; ++i) {
        long long arrival, duration;
        cin >> arrival >> duration;
        auto [start, window] = serveCustomer(arrival, duration, now, busy, freeWindows);
        long long wait = start - arrival;
        totalWait += wait; // 等待不是办理时长，不计服务开始后的时间。
        maxWait = max(maxWait, wait);
        cout << window << ' ' << start << ' ' << start + duration << '\n';
    }
    cout << totalWait << ' ' << maxWait << '\n';
}
```

7. 用完整样例逐项核对模型和代码

顾客	到达 / 时长	安排时刻的变化	选择窗口	开始 / 结束	等待
1	0 / 6	0，已有空闲窗口	1	0 / 6	0
2	0 / 2	0，窗口 2 空闲	2	0 / 2	0
3	1 / 1	0→1，全部忙；1→2，窗口 2 完成	2	2 / 3	1
4	1 / 1	保持 2，全部忙；2→3	2	3 / 4	2
5	2 / 1	保持 3，全部忙；3→4	2	4 / 5	2
6	7 / 2	4→7，窗口 2、1 都已完成	1	7 / 9	0
7	7 / 1	保持 7，窗口 2 空闲	2	7 / 8	0

等待时间为 `0、0、1、2、2、0、0`，总和为 5，最大值为 2。窗口 1 的业务区间为 `[0,6)、[7,9)`；窗口 2 为 `[0,2)、[2,3)、[3,4)、[4,5)、[7,8)`。结束时刻不再占用窗口，所以相邻区间可以首尾相接。

正确性核对： 处理下一位顾客前，已处理顾客的安排与规则一致，每个窗口恰好处于空闲或忙碌集合之一。释放只转移已完成窗口，不会提前空出窗口；没有空闲窗口时，最早完成之前无处办理，跳到该时刻不会跳过可安排的时刻；全部释放后选择最小空闲编号。安排当前顾客后，两个集合再次完整表示窗口状态。

复杂度计算： m 个窗口逐个压入空闲堆；每位顾客让一个窗口进入忙碌堆，它的完成记录最多弹出一次，再进入空闲堆。看似内外嵌套的释放循环，总共最多处理 n 次完成事件。每次堆操作的元素数最多为 m，总时间为 $O((m+n)\log(m+1))$，空间为 $O(m)$；顾客边读边处理，输出也逐行进行。

now 可能超过单个到达时刻：一个窗口连续办理 200000 份、每份耗时 10^6 的业务，会累计到 2×10^11。若这些顾客同时到达，等待总和为 `(0+1+…+199999)×10^6=19999900000000000`。把到达时刻、开始时刻、结束时刻和等待统计都保存为 long long，就能完整表示这些累计结果。

练习： 洛谷 P2058 — 海港、洛谷 P1065 — 作业调度方案（进阶，重新按它自己的机器与工序规则建模）。

（十四） 从推导到程序职责与错误定位

写代码前，把每项规则对应到一份状态、一个判断或一次更新。这样出错时能沿着对应关系定位，而不是对整个程序重新猜算法。

推导中的一句话	数对程序中的职责	窗口程序中的职责
已经处理过的对象是什么	frequency 只保存过去位置	两个堆保存窗口状态
当前对象需要什么信息	查 `K-x` 的过去次数	查空闲编号或最早完成时刻
当前步骤计算了什么	加入以当前位置为较后端的新数对	确定当前顾客的开始时刻与窗口
更新后留给下一步什么	把 x 的频次增加 1	把窗口转为忙碌，now 保存本次开始时刻

推导中的一句话	数对程序中的职责	窗口程序中的职责
最终怎样输出	累计数对数量	每位安排、总等待、最大等待

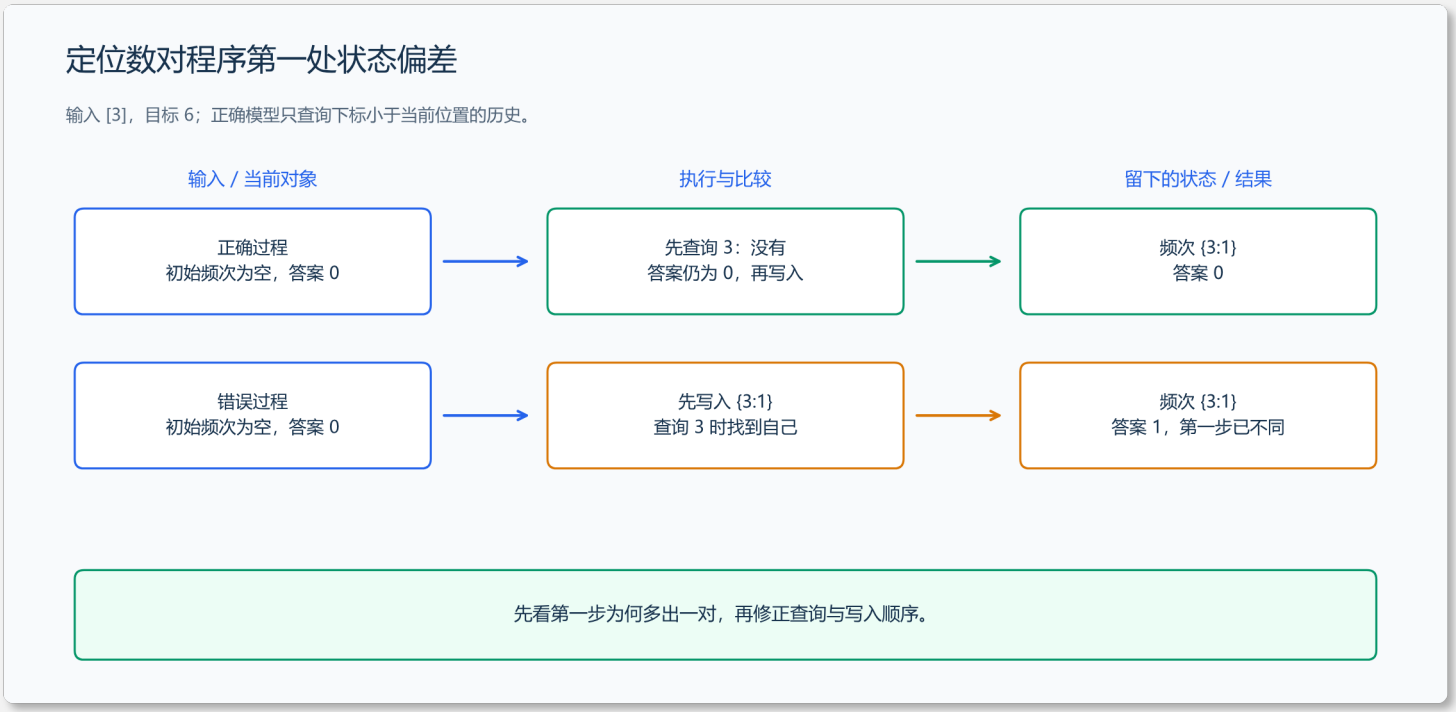
1. 通过第一处不同定位更新次序

数对题读到 `[3]`、目标 6 时，正确过程为“查询 3 的次数 0，答案不变，再写入 3”。如果先写入，再查询，就会得到一对，第一步已经与合法下标要求不同。

下面是**错误写法**，用来定位问题；它是循环内部的演示片段，不是答案程序：

```cpp
++frequency[x]; // 错误：先写入，查询时已经能看见当前位置。
auto it = frequency.find(target - x);
if (it != frequency.end()) {
    answer += it->second; // x+x==target 时，把当前项与自己配成一对。
}
```

同样的数据下，窗口题若把 `finish ≤ now` 写成 `finish<now`，时刻 2 完成的窗口就不能在时刻 2 接待顾客 3；偏差出在“结束时刻即可接待”的翻译。先比较时刻 2 的窗口集合，就能定位这个判断，而不必等到最后的总等待时间。



2. 分开核对模型、算法与实现

“把任意选择当成连续区间”改写了合法方案，是模型错误；硬币题模型仍是最少枚数，但“先拿大硬币”漏掉更好的完整方案，是算法推导错误；数对的频次模型正确，却先写后查，是实现没有执行已经推导的顺序。

先手算失败输入的正确结果，再逐步记录自己的状态。若模型已经允许题目禁止的选择，应先改模型；若步骤理由不能解释某个合法方案，应补推导；若推导与手算一致而运行过程不同，就对照变量、分支和更新顺序修代码。

（十五） 反例构造、缩小输入与小规模对拍

随机数据并不总能碰到关键错误。可以先围绕某项规则构造数据，再把失败输入删到能手算的大小。

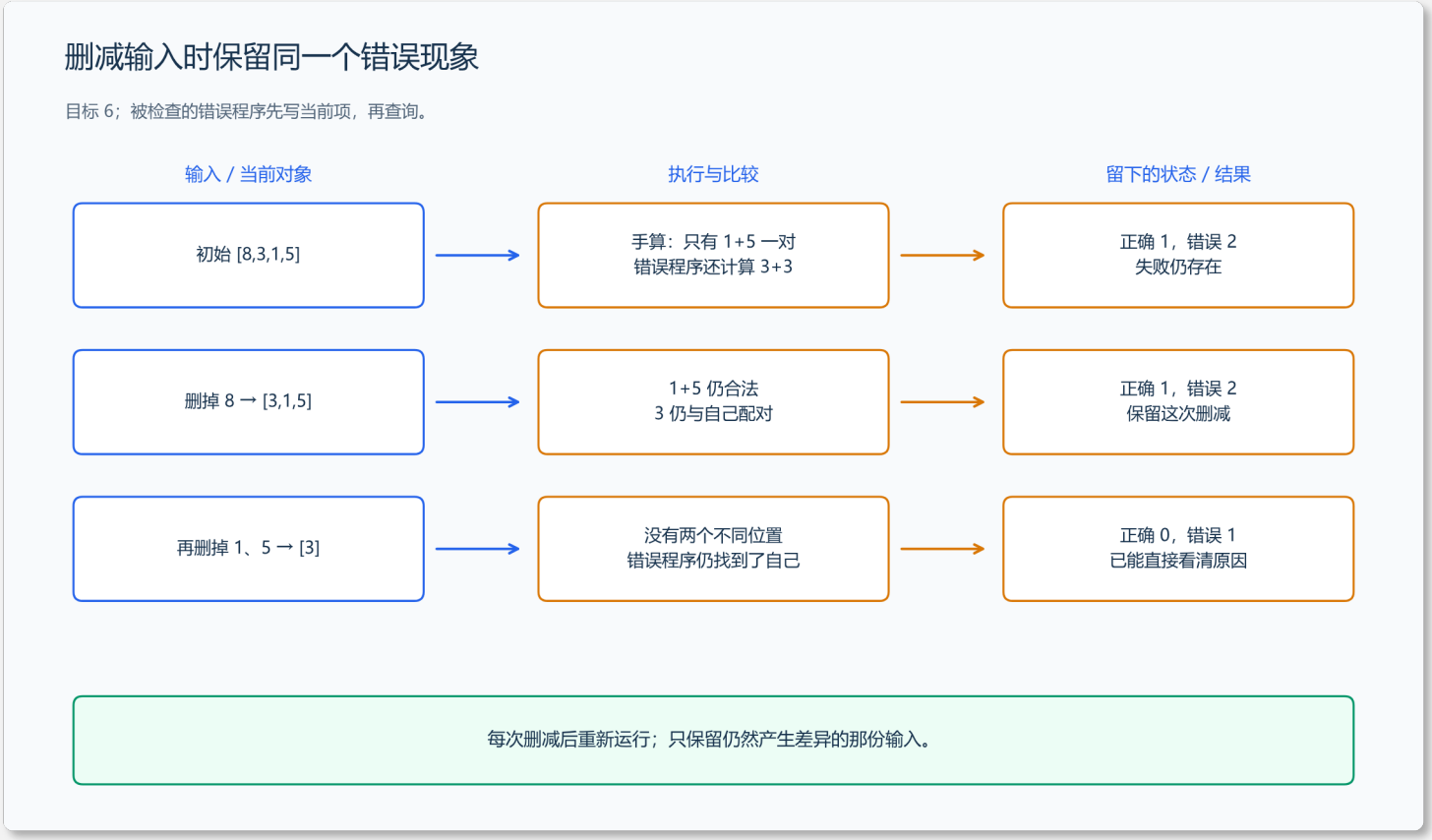
1. 每份测试对应一项要验证的规则

数对输入与目标	正确结果	要核对的规则
`[3]`，6	0	不能和当前位置配对
`[3,3]`，6	1	两个相同值可以来自不同位置
`[3,3,3]`，6	3	次数不是存在性
`[1,1,5,5]`，6	4	两组重复值贡献数量的乘积
`[-2,5,-2]`，3	2	负数同样可以成为键与搭档
`[1,2,3]`，100	0	未找到搭档时不增加答案

窗口例题也可以逐项试：一个窗口连续办理检查等待；三个顾客同刻到达检查输入顺序；两个窗口同刻完成检查全部释放；窗口 2 更早空闲但两者都已空闲检查编号优先。每份数据都有具体要核对的动作。

2. 把失败输入缩成能看清规则的小例子

若一份很长的数对输入失败，先尝试删掉一部分元素，并分别运行枚举与频次程序。仍有差异就保留删减结果；差异消失就撤回这次删减，换另一部分。接着可以尝试缩小数值，但每次都重新运行，确认错误仍存在。



例如“先写入再查询”的程序在 `[8,3,1,5]` 、目标 6 上多算了一对。删掉 8、1、5 后，`[3]` 仍然错误；它直接显示出“同一位置被使用两次”。缩小输入是为了看清原因，不是把所有数字机械地改成 0。

3. 用独立枚举核对优化程序

前面的枚举函数逐对检查，频次函数按历史统计，排序函数按同值组统计，三者经过不同的计算过程。用小输入逐一比较结果，就能发现某种实现漏掉重复值、重复计数或错误更新。

题目原文（自拟验证程序）： 程序无需输入。枚举长度 1 到 7、元素只取 -1、0、1 的全部数组；每份数组分别检查目标 -2 到 2。用枚举、排序、频次三种方法计算数对数量。第一次不一致时输出 `FAIL` 、数组长度、目标、数组和三份结果并结束；全部一致时输出 `OK` 。

将前面的 `countByEnumeration` 、`countBySorting` 、`countByFrequency` 函数放在同一文件中，文件开头加入 `<algorithm>` 、`<iostream>` 、`<unordered_map>` 、`<vector>` 和 `using namespace std;` 。用下面的 main 替换数对输入程序的 main，不需要新的头文件。

```
int main() {
    for (int n = 1; n ≤ 7; ++n) {
        int total = 1;
        for (int i = 0; i < n; ++i) total *= 3; // n 个位置共有 3^n 种填写方式。
        for (int code = 0; code < total; ++code) {
            vector<long long> a(n);
            int remaining = code;
            for (auto &x : a) {
                x = remaining % 3 - 1; // 三进制位 0、1、2 对应 -1、0、1。
```

```cpp
                remaining /= 3;
            }
            for (long long target = -2; target <= 2; ++target) {
                long long brute = countByEnumeration(a, target);
                long long sorted = countBySorting(a, target);
                long long fast = countByFrequency(a, target);
                if (brute != sorted || brute != fast) {
                    cout << "FAIL\n" << n << ' ' << target << '\n';
                    for (long long x : a) cout << x << ' ';
                    cout << '\n' << brute << ' ' << sorted << ' ' << fast << '\n';
                    return 0; // 保留第一份差异输入，再按步骤手算。
                }
            }
        }
    }
    cout << "OK\n"; // 这些小数组的三份计算结果全部相同。
}
```

cpp

样例输出与解释： 输出 `OK`。总共生成 `3+9+27+81+243+729+2187=3279` 份数组，比较 16395 组“数组、目标”。三进制枚举逐位确定每个位置的取值，覆盖了小值域中的重复值、负数和不同排列。

验证程序的复杂度： 最大长度记为 L，目标共有 5 个；长度 n 的数组有 3^n 份，每份的枚举计算需要 $O(n^2)$，因此总时间为 $O(\sum_{n=1}^{L} 3^n n^2)$，同时保存数组和频次表需要 $O(L)$ 空间。这里 L 为 7，用较慢但直观的枚举来检查较快的实现。

对拍得到的是这些输入上的一致结果。正确性理由仍来自“每一对在较后的那个位置恰好计入一次”的证明；类型检查还要代入 200000 项的计数，窗口时间也要代入较长的办理过程。证明、手算、对拍分别检查不同的部分。

（十六） 从 WA 回到题面与模型

WA 表示程序输出与题目要求的结果不同。评测给出的 WA 通常没有写出原因，我们可以把它转成几个可核对的猜想，再通过小数据判断哪项规则被理解错了。

1. 先得到一份能解释的失败输入

保留当前提交，按题目的输入格式构造小数据，先手算正确结果，再运行提交。已经有失败输入时，按前一节的方法逐步缩小。记录输入、正确结果、程序结果和第一处差异，之后每次修改都重新运行这份数据。

例如数对程序对 `[1,1,5,5]`、目标 6 输出 1，手算得到 4。四个下标对是 `(0,2)`、`(0,3)`、`(1,2)`、`(1,3)`，差异集中在重复值的贡献，先检查“统计下标对”是否被写成“统计不同取值组合”。

2. 用不同现象区分可能的误读

失败现象	优先回看的题意	能区分猜想的小输入
一个元素也被算成一对	“两个不同下标”	`[3]`、目标 6，应为 0

失败现象	优先回看的题意	能区分猜想的小输入
重复元素时数量偏少	“不同位置分别计数”	`[1,1,5,5]`、目标 6，应为 4
每对都被算了两遍	下标对是否有顺序	`[1,5]`、目标 6，应为 1
结束与到达同刻时安排变晚	“结束时刻即可接待”	一个窗口，顾客为 `(0,2)`、`(2,1)`，第二位在 2 开始
多个窗口都空闲时编号不对	“选择最小空闲编号”	样例顾客 6，在时刻 7 应选窗口 1
YES/NO 正确，但方案被判错	输出编号与操作执行规则	灯为 `11` 时，翻转左侧编号应为 1
小数据相同，大计数不同	答案表示的是累计量还是单项值	200000 项 3、目标 6，答案为 19999900000

这些现象给出排查顺序。接着仍要对照运行过程：数量偏少既可能来自错误去重，也可能来自正确模型中漏掉某次频次更新。逐步看状态，才能区分二者。

3. 完整推演：从窗口编号错误找回选择规则

假设一个程序只用 `（上次结束时刻,窗口编号）` 的小根堆。它在前五位顾客上与样例一致，却把顾客 6 安排给窗口 2。先停在第一处差异，不继续比较后面的总等待时间。

此时顾客在 7 到达，窗口 1 于 6 结束，窗口 2 于 5 结束。两者都空闲。列出程序可能使用的规则及各自预测：

可能的理解	在时刻 7 的预测	与题面对应的文字
选择最早变空闲的窗口	窗口 2	程序把旧结束时刻继续用于当前选择
选择当前空闲窗口中的最小编号	窗口 1	题面明确要求多个空闲时选最小编号
每次轮流选择窗口	取决于上一位编号	题面没有安排轮转动作



再去掉前面的等待过程，只保留两位已经完成的顾客与后来的一位：两个窗口，输入 `(0,6)、(0,2)、(7,1)`。前两位分别使用窗口 1、2；第三位仍应使用窗口 1。这个短例子保留了错误，却去掉了可能干扰判断的排队细节。

问题出在：**完成时间用于确定窗口何时空闲，编号用于决定当前选哪个空闲窗口。** 把全部已完成记录转到按编号排序的空闲堆，就让两项规则各自对应到一个动作。修正后，重新核对短例子、完整样例和同刻完成的例子。

4. 把推测变成可以重复检查的证据

每次只针对一个猜想构造数据。例如怀疑同刻动作顺序，就让结束与到达相等；怀疑重复计数，就只保留两三个相同值。先写下这个猜想预测的输出，再运行代码，避免看到结果之后再随意解释。

若手算已经违背题面的明确规则，回到文字重新建模；若手算符合规则而程序不同，继续比较状态和更新顺序。对拍发现差异时也沿着同一条路线处理：保留数据、定位第一处变化、解释为什么该动作错误，再修改实现。

把修正后的失败输入留在验证程序或练习记录中。下一次修改代码时先运行它，既能确认问题没有再次出现，也能提醒自己当时漏读的是哪一句题意。

（十七） 解题复盘与模型迁移

做完一道题后，可以留下题目的对象和目标、关键等价关系、步骤正确的理由，以及曾让程序失败的那份小输入。代码很长时，这几项能帮助我们重新得到实现。

题目中保留下来的关系	可以再次识别的任务	要重新落实的题意
固定一个对象，查询所需搭档	和、差或多个候选目标的配对计数	下标是否有序、同值是否分开计数、能否重复使用当前位置
历史只影响某份统计量	频次、最值、累计量的逐项维护	未来是否还需要具体位置或方案
状态只在到达与完成时改变	窗口、机器或资源的事件模拟	队列顺序、同刻动作顺序、资源选择规则
操作保持一个量不变	翻转、交换与可达性判断	是否还需要构造证明，能否写出完整操作序列
候选值可用同一方案继续满足	最小可行值、最大可行值	判断函数是否真的代表原题全部合法方案

数对题改成差值时，要重新写出谁减谁；改成统计不同取值组合时，要重新解释重复值的贡献。窗口题改成“选择最早空闲窗口”时，选择依据也变了。相似故事可以提示一个模型，但每项计算仍要来自当前题面。

阶段练习： 洛谷 P1102 — A-B 数对、CF 702B — Powers of Two、洛谷 P2058 — 海港、洛谷 P1065 — 作业调度方案（进阶）。

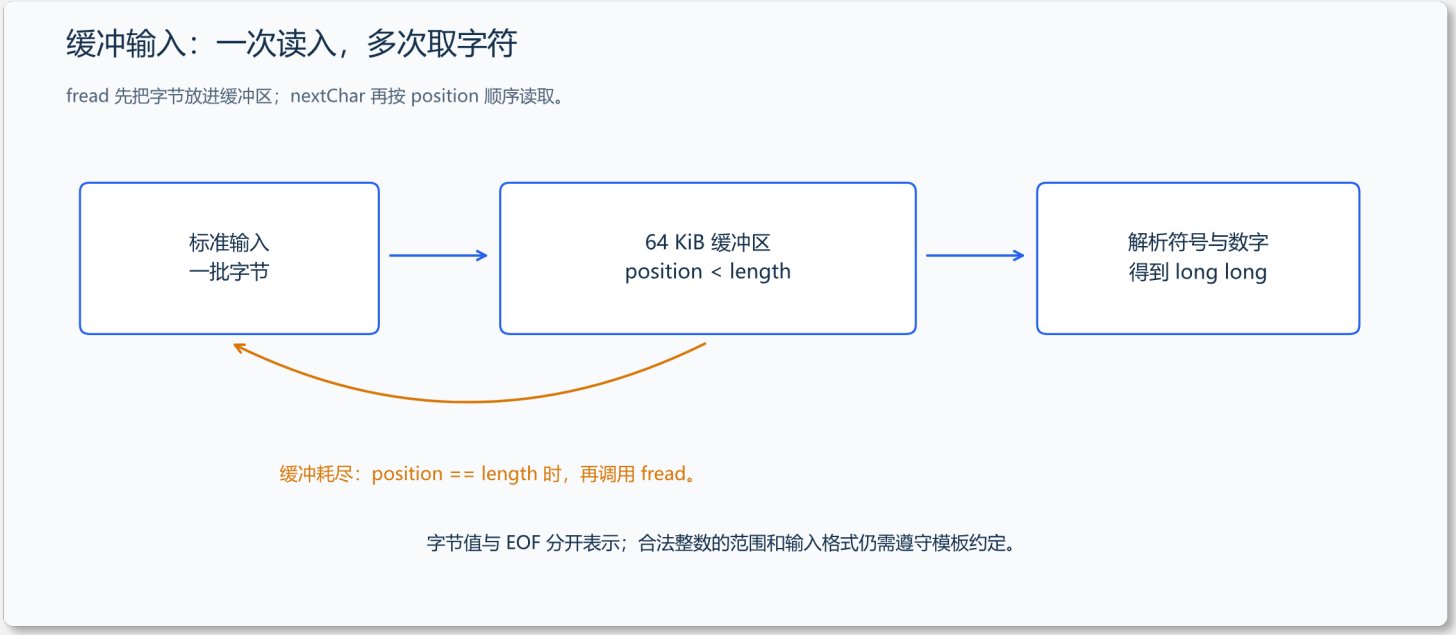
建模得到的是变量、合法方案和计算步骤，程序还要把输入完整读入、把计算结果保存下来，并按题目要求输出。下一部分先解决批量输入，再把几百位整数的运算拆成逐位动作，最后核对可提交的程序。

四、批量输入、大整数运算与提交检查

题意已经能转成变量与计算步骤，还要把输入完整读进程序，让运算结果按要求保存和输出。先沿字节解析一批整数，再从超长数字的竖式构建大整数四则运算，最后用具体数据核对提交程序。

（一） fread 整数输入模板

一次读入大量整数时，可以先把一批字符读进缓冲区，再逐个解析。下面按空格或换行分隔整数，用 `long long` 保存解析结果；先看字符怎样组成一个数，再看缓冲区怎样补充数据。



数据先读进缓冲区，解析函数再逐字节取用；缓冲耗尽时重新填充，而不是每个字符都发起一次读取。



图把缓冲容量缩小为 4 字节来手算，实际代码使用 64 KiB。输入 `-12 7` 后换行：先读出负号，再用 `value=value*10+digit` 得到 1、12；空格结束第一个整数，乘符号后得到 -12。下一次读数时缓冲已耗尽，重新填入 `7` 与换行，得到 7；再次填充读到 0 字节才返回 EOF。EOF 是读取状态，不能当成数字或普通空白继续处理。

本例用到的输入接口： **`<cstdio>` 提供 fread、stdin、EOF 和 printf。`fread(buffer,1,count,stdin)` 最多读取 count 个一字节单元，返回实际读到的单元数；stdin 是标准输入来源，EOF 表示读不到下一个字符。`<cctype>` 的 isspace 判断空白，代码先识别 EOF，再把读到的字节转成 unsigned char 交给它。std::size_t 是无符号的大小类型；字符用 int 接收，可以同时保存字节值和 EOF。`printf("%lld\n",value)` 输出 long long 整数。本章从读入到输出都使用 C 输入输出接口。**

```cpp
#include <cctype>
#include <cstdio>
#include <limits>

class FastInput {
private:
    static constexpr std::size_t bufferSize = 1 << 16;
    unsigned char buffer_[bufferSize];
    std::size_t position_ = 0;
    std::size_t length_ = 0;

    int nextChar() {
        if (position_ == length_) {
            // 当前缓冲区耗尽时，一次读入最多 64 KiB 字节。
            length_ = std::fread(buffer_, 1, bufferSize, stdin);
            position_ = 0;
            if (length_ == 0) {
                return EOF;
            }
        }
        return buffer_[position_++];
    }

public:
    bool read(long long &result) {
        int ch = nextChar();
        while (ch ≠ EOF && std::isspace(static_cast<unsigned char>(ch))) {
            ch = nextChar();
        }
        if (ch == EOF) {
            return false;
        }
        bool negative = ch == '-';
        if (ch == '-' || ch == '+') {
            ch = nextChar();
        }
        if (ch < '0' || ch > '9') {
            return false;
        }
        unsigned long long value = 0;
        // 先累计绝对值，才能表示 long long 最小值对应的幅度。
        while (ch ≥ '0' && ch ≤ '9') {
            value = value * 10 + static_cast<unsigned int>(ch - '0');
            ch = nextChar();
        }
        auto limit = static_cast<unsigned long long>(std::numeric_limits<long
long>::max());
        if (negative && value == limit + 1) {
            // 最小负数不能通过“先转为正的 long long 再取负”获得。
            result = std::numeric_limits<long long>::min();
        } else {
            result = static_cast<long long>(value);
            if (negative) {
                result = -result;
            }
```

```cpp
        }
        return true;
    }
};
```

调用片段（与上面的类放在同一个源文件中）：

```cpp
int main() {
    FastInput input;
    long long value;
    while (input.read(value)) {
        // 一直读到 EOF；输入由 FastInput 统一接管。
        std::printf("%lld\n", value);
    }
}
```

`read` 跳过空白，读取负号和数字，并把结果写入参数；没有下一个整数时返回 `false`，外层循环随即结束。之后的读取继续从该对象保存的缓冲位置开始。

（二） 大整数的加减乘除

缓冲输入解决“一次读很多数”的问题，数本身有几百位时，还需要改变保存和计算的方式。例如 `999999999999999999999+1`，答案是 `10000000000000000000`；每一位都要保留下来。把输入读成 string，再用已学的数组、循环和函数模拟竖式，就能逐步构建**高精度整数运算**。

这里先让四种运算处理数的绝对值，再单独处理正负号。这样，加法的进位、减法的借位和除法的余数都能分别看清楚。

1. 完整示例：大整数四则计算器

题目原文（自拟）： 第一行输入操作次数 T，接下来 T 行，每行输入一个运算符 op 和两个十进制整数 A、B。op 是 `+`、`-`、`*`、`/` 之一。$1 \le T \le 20$；A、B 的数字部分各有 1 到 2000 位，可以带一个正号或负号，也可以包含前导零。

加、减、乘操作分别输出和、差、积。除法输出整数商与余数，用一个空格分隔；商向零截断，余数随被除数的符号。例如 `-17/5` 输出 `-3 -2`，因为 $-17 = (-3) \times 5 + (-2)$。B 为零时，除法输出 `DIV0`。每个操作的结果占一行，数字按通常的十进制形式输出。

样例输入：

```text
8
+ 999 87
- 1000 728
* 123 45
/ 12045 37
- 728 1000
+ -999 87
/ -17 5
/ 0012 -000
```

样例输出：

```text
1086
272
5535
325 20
-272
-912
-3 -2
DIV0
```

分析路线： 先保存数字位；加减从个位向高位处理；乘法展开每一对位的贡献；除法从最高位向低位读入前缀，逐步得到商和余数；最后用绝对值大小和符号决定调用哪种运算。

各段代码按本节顺序放进同一个源文件，最后接上计算器的 `main`。下面不会把几种运算塞进一个长代码块，而是逐段构建它们。

2. 数字位的存储、比较与输出

把 `1203` 存成 `[3,0,2,1]`：`a[0]` 是个位，`a[1]` 是十位，`a[2]` 是百位。每个位置保存一个 `0` 到 `9` 的数字，数值为

$$A = a[0] + 10a[1] + 100a[2] + \cdots = \sum_{i=0}^{n-1} a[i]10^i.$$

加法产生的进位总是去下一格，减法也是从这一格向下一格借位；让低位在前，便于从下标 `0` 开始循环。输出时再反向读取。数字 `0001203` 和 `1203` 表示同一个数，数组末尾多余的高位零会被删除；零统一保存成 `[0]`。



先放入头文件和下面的辅助函数。`<string>` 保存输入文本，`<vector>` 保存数字位，`<algorithm>` 提供 max，`<utility>` 提供 pair，`<iostream>` 负责本例的标准流输入输出。`using Digits = vector<int>` 是类型别名；与前面的 fread 示例各自组成一个程序。

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

using Digits = vector<int>; // 下标 i 保存十进制的 10^i 位。

void trim(Digits &a) {
    while (a.size() > 1 && a.back() == 0) {
        a.pop_back(); // 删除多余高位零，但给数字 0 留下一位。
    }
}

Digits readDigits(const string &s) {
    Digits a;
    for (int i = static_cast<int>(s.size()) - 1; i >= 0; --i) {
        a.push_back(s[i] - '0'); // 字符 '7' 转成数字 7，低位先存。
    }
    trim(a);
    return a;
}

string writeDigits(const Digits &a) {
    string s;
```

```cpp
    for (int i = static_cast<int>(a.size()) - 1; i ≥ 0; --i) {
        s.push_back(static_cast<char>('0' + a[i])); // 高位先输出。
    }
    return s;
}

int compare(const Digits &a, const Digits &b) {
    if (a.size() ≠ b.size()) {
        return a.size() > b.size() ? 1 : -1; // 位数多的绝对值大。
    }
    for (int i = static_cast<int>(a.size()) - 1; i ≥ 0; --i) {
        if (a[i] ≠ b[i]) {
            return a[i] > b[i] ? 1 : -1; // 第一处不同的高位决定大小。
        }
    }
    return 0; // 每一位都相同。
}
```

比较过程： `1000` 有四位，`728` 有三位，所以前者更大；`1203` 与 `1290` 都有四位，千位、百位相同，十位的 0 小于 9，因此前者更小。不能从个位开始比较大小，`1203` 的个位虽较大，整个数仍较小。

读入、输出和同位数比较都最多检查每一位一次，时间为 $O(n)$。读入得到 n 个数字位，存储为 $O(n)$。

3. 加法：逐位相加与进位

先手算 `999+87`。个位相加得到 16，个位留下 6，把一个十送到十位；十位再把这个进位一起加上。

当前位	A 的数字	B 的数字	进入本位的进位	本位总和 s	写入的数字 s%10	送到下一位的进位 s/10
个位	9	7	0	16	6	1
十位	9	8	1	18	8	1
百位	9	0	1	10	0	1
千位	0	0	1	1	1	0



数组按顺序得到 `[6,8,0,1]`，反向输出为 `1086`。较短的数没有百位时，这一位按 `0` 参与相加；两个数都处理完后，千位仍收到一个进位，所以还要再走一次循环。

把每行手算换成变量：本位总和为 $s = a_i + b_i + c$。十进制分解 $s = 10\lfloor s/10 \rfloor + (s \bmod 10)$，因此留下余数，传递商。

```cpp
Digits addMagnitude(const Digits &a, const Digits &b) {
    Digits result;
    int n = static_cast<int>(max(a.size(), b.size()));
    int carry = 0;
    for (int i = 0; i < n || carry != 0; ++i) {
        int sum = carry;
        if (i < static_cast<int>(a.size())) sum += a[i];
        if (i < static_cast<int>(b.size())) sum += b[i];
        result.push_back(sum % 10); // 当前位留下个位部分。
        carry = sum / 10; // 十位部分成为下一格的进位。
    }
    return result;
}
```

代码解释： `carry` 在一次循环开始时来自上一位，在循环结束时改为送给下一位的值。每格最多是 `9+9+1=19`，进位最多为 `1`；再长的输入也没有把整个大整数塞进 `int`。时间为 $O(\max(n, m))$，结果空间为 $O(\max(n, m))$，`n`、`m` 分别是两个数的位数。

练习： 洛谷 P1601 – 高精度加法。

4. 减法：绝对值比较与逐位借位

先计算 `1000-728`。个位 `0-8` 不够减，就从下一位借一个十，得到 2；下一位除了减去 B 的数字，还要扣掉刚才借走的 1。

当前位	A 的数字	B 的数字	上一位借走的 1	初步差 t	本位写入	从下一位借的 1
个位	0	8	0	-8	2	1
十位	0	2	1	-3	7	1
百位	0	7	1	-8	2	1
千位	1	0	1	0	0	0



得到 `[2,7,2,0]`，去掉最高位的 0，输出 `272`。写入数字时，初步差为负就加 10，并记录借位；下一格再扣掉这个借位。

```cpp
Digits subtractMagnitude(const Digits &a, const Digits &b) {
    Digits result;
    int borrow = 0;
    for (int i = 0; i < static_cast<int>(a.size()); ++i) {
        int difference = a[i] - borrow;
        if (i < static_cast<int>(b.size())) difference -= b[i];
        borrow = difference < 0 ? 1 : 0;
        if (borrow != 0) difference += 10; // 从下一格借来一个十。
        result.push_back(difference);
    }
    trim(result); // 1000-728 的末尾零、相等两数相减的高位零都在这里整理。
    return result;
}
```

调用顺序： `1000-728` 比较后直接减；`728-1000` 则计算 `1000-728` 的幅度，再给结果加负号，得到 `-272`。所以符号处理会先用 compare 比较，随后把较大的幅度放在第一个参数。相等时逐位得到 0，最后留下 `[0]`。

借位不是把下一格永久改成负数，而是用 borrow 保存“下一格要扣掉 1”。每步的差在 -10 到 9 之间，加 10 后就是一个数字位。时间为 $O(\max(n, m))$，结果空间同阶。

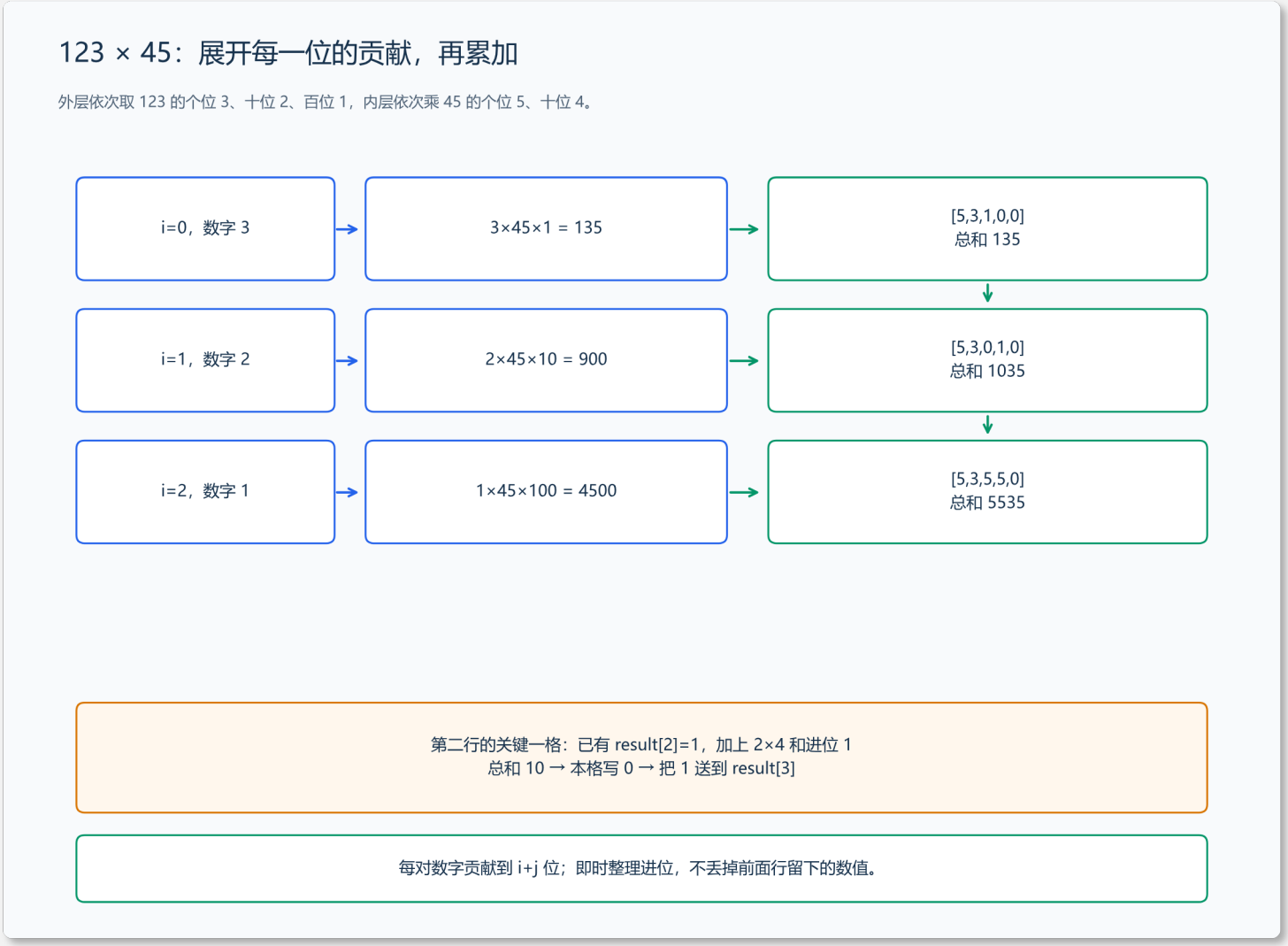
练习： 洛谷 P2142 – 高精度减法。

5. 乘法：位权展开与错位累加

计算 `123×45` 时，把 123 展开成 $3 + 2 \times 10 + 1 \times 100$：

$$123 \times 45 = 3 \times 45 + 2 \times 45 \times 10 + 1 \times 45 \times 100 = 135 + 900 + 4500 = 5535.$$

每一行都在已有结果上继续累加。个位 3 的贡献从下标 0 开始，十位 2 的贡献从下标 1 开始，百位 1 的贡献从下标 2 开始。



再拆开一行：`a[i]` 表示 $a_i 10^i$，`b[j]` 表示 $b_j 10^j$，两者相乘贡献 $a_i b_j 10^{i+j}$，所以要累加到 `result[i+j]`。但这格可能已有前一行留下的数字，也可能收到本行上一格的进位，必须三项一起算。

处理 A 的位 i	处理 B 的位 j	本格原值	位乘积	本行进位	总和	写入 / 新进位
0，数字 3	0，数字 5	0	15	0	15	5 / 1
0，数字 3	1，数字 4	0	12	1	13	3 / 1，行末写下标 2 为 1
1，数字 2	0，数字 5	3	10	0	13	3 / 1
1，数字 2	1，数字 4	1	8	1	10	0 / 1，行末写下标 3 为 1
2，数字 1	0，数字 5	0	5	0	5	5 / 0
2，数字 1	1，数字 4	1	4	0	5	5 / 0

```cpp
Digits multiplyMagnitude(const Digits &a, const Digits &b) {
    Digits result(a.size() + b.size(), 0);
    for (int i = 0; i < static_cast<int>(a.size()); ++i) {
        int carry = 0; // 每一行从自己的起点开始进位。
        for (int j = 0; j < static_cast<int>(b.size()); ++j) {
            int sum = result[i + j] + a[i] * b[j] + carry;
            result[i + j] = sum % 10; // 边累加边整理为一个数字位。
            carry = sum / 10;
```

```cpp
            }
            result[i + b.size()] = carry; // 本行多出的最高位还没有被前面的行写入。
        }
        trim(result);
        return result;
    }
```

代码解释： 外层选 A 的一位，内层让它乘 B 的每一位；`i+j` 实现竖式中的错位。算完一行，已经得到这一行与前面各行的总和，下一行在这个结果上继续加。

每步已存数字最多为 9，单个位乘积最多为 81，进位最多为 9，三项总和最多为 99。及时整理每个数字位，让中间值始终很小。若先把所有位乘积堆在一个位置、最后才进位，就要另外分析那个累计值的大小。

两个数有 n、m 位时，共做 nm 次位乘法，时间为 $O(nm)$；结果最多 n+m 位，结果空间为 $O(n+m)$。2000 位乘 2000 位约做 4000000 次位乘法。

练习： 洛谷 P1303 – A*B Problem。

6. 除法：逐位确定商和余数

加、减、乘从低位开始，是因为低位产生的进位和借位要传给高位。除法要先决定高位商，因此从被除数最高位开始。手算 `12045÷37`，不是一次求出整个商，而是把数字 1、2、0、4、5 依次接到当前余数后面。

6.1 从具体前缀推导下一位商

开始时，已处理前缀为 0，商为 0，余数为 0。读入数字 d 后，把原来的余数 r 扩大十倍，再接上 d，得到 $u = 10r + d$。

读入数字 d	上一步余数 r	本步待除数 u=10*r+d	逐次减去 37	本位商	新余数
1	0	1	1 小于 37，不减	0	1
2	1	12	12 小于 37，不减	0	12
0	12	120	`120→83→46→9`	3	9
4	9	94	`94→57→20`	2	20
5	20	205	`205→168→131→94→57→20`	5	20



读出的商位依次为 `0、0、3、2、5`，整理成 `325`，余数为 `20`。可以反算 $37 \times 325 + 20 = 12045$。

为什么旧商不用重新计算？设已读前缀 `P` 的关系为 $P = BQ + r$，接下一位 `d` 后：

$$P' = 10P + d = B(10Q) + (10r + d).$$

再把 $10r + d$ 分成 $Bq + r'$，得到

$$P' = B(10Q + q) + r'.$$

因此原来的商 `Q` 扩大十倍，再补一个数字 `q`；新余数只需由 `10*r+d` 继续计算。这就是长除法每次只处理余数的原因。



每步减到不足一个 B 就停止，所以旧余数满足 $0 \le r < B$；接入一个 0 到 9 的数字后，$0 \le 10r + d < 10B$。因此最多减 9 次，本位商就在 0 到 9 之间。不是把整个被除数反复减到零，而是每读一位只确定一位商。

6.2 大整数除以大整数的构建

商的数组也按低位在前存储。处理被除数的下标 i 时，得到的商位就写在 `quotient[i]`；最高位不足相除会写 0，末尾统一整理。

余数数组 `[9]` 接上数字 4 后应表示 94，即 `[4,9]`。`remainder.insert(remainder.begin(),a[i])` 把新数字放在最低位，让原来的数字向高位移动一格，正好完成乘 10 再加新数字。余数为 `[0]` 时接入 1 得到 `[1,0]`，trim 会整理成 `[1]`。

```
pair<Digits, Digits> divideMagnitude(const Digits &a, const Digits &b) {
    Digits quotient(a.size(), 0);
    Digits remainder{0};
    for (int i = static_cast<int>(a.size()) - 1; i ≥ 0; --i) {
        remainder.insert(remainder.begin(), a[i]); // 余数乘 10，再接入下一位。
        trim(remainder);
        int digit = 0;
        while (compare(remainder, b) ≥ 0) {
            remainder = subtractMagnitude(remainder, b);
            ++digit; // 每减去一次除数，本位商加一。
        }
```

```cpp
        quotient[i] = digit; // 该商位对应被除数当前的十进制位权。
    }
    trim(quotient);
    return {quotient, remainder}; // 同一次长除法同时得到商和余数。
}
```

代码解释： 每次循环结束，都保存“已读前缀 = 除数×已得商 + 当前余数”。下一轮只把余数接入一位，再继续减；前面已经决定的高位商不受影响。

`5÷12` 的第一位不减，商为 0，余数为 5；`1005÷5` 的商位依次为 `0、2、0、1`，整理成 `201`。其中中间的 0 必须保留：已经得到商 2 后，下一位待除数是 0，本位商就要写 0，不能把 `201` 压成 `21`。

n 是被除数位数，m 是除数位数。每步接位、比较和减法处理至多 m+1 位的余数，本位最多做 9 次减法；因此时间为 $O(nm)$，商和余数空间合计为 $O(n+m)$。如果 n 小于 m，余数也只会增长到 n 位，实际工作更少。

这个实现重在直接表现长除法。需要计算几十万位的乘除时，nm 次逐位工作会迅速增多；此时再学习更大的进制、快速乘法及相应除法，先把本节的每一步算清楚。

练习： 洛谷 P1932 – A+B A-B A*B A/B A%B Problem（综合练习；20000 位数据要结合工作量继续优化）。

6.3 大整数除以普通整数的简化

若除数是 `37` 这样的普通整数，余数也能直接放进普通整数，不必调用数字数组的比较和减法。此时直接算出 `current/b` 和 `current%b`。

例如被除数有 5000 位，除数为 10^9：旧余数最多为 999999999，接一位后最多为 9999999999，用 long long 可以保存；商的每一位仍在 0 到 9 之间。

下面的函数是另一种独立的除法实现，复用 Digits 与 trim。它返回商数组和 long long 余数；前面的计算器采用完整的 `divideMagnitude`，不需要把这个函数接入 main。

```cpp
pair<Digits, long long> divideByInt(const Digits &a, int b) {
    Digits quotient(a.size(), 0);
    long long remainder = 0;
    for (int i = static_cast<int>(a.size()) - 1; i ≥ 0; --i) {
        long long current = remainder * 10 + a[i];
        quotient[i] = static_cast<int>(current / b); // 直接确定本位商。
        remainder = current % b; // 留给下一位的余数。
    }
    trim(quotient);
    return {quotient, remainder};
}
```

每一位只做一次普通整数除法和取余，时间为 $O(n)$；商数组占 $O(n)$，另有一个余数变量。**练习：** 洛谷 P1480 – A/B Problem（高精度除法 I），题面的除数为 1 到 10^9，可以用这一版逐位除法。

7. 把正负号接到四则运算

数组保存幅度，再增加 bool negative 保存是否带负号。`-000` 的幅度整理成 `[0]`，随后去掉负号；输出得到 `0`，不是 `-0`。

```cpp
struct BigInteger {
    Digits digits;
    bool negative = false; // true 表示负数，digits 只保存绝对值。
};

bool isZero(const Digits &a) {
    return a.size() == 1 && a[0] == 0;
}

BigInteger makeInteger(Digits digits, bool negative) {
    if (isZero(digits)) negative = false; // 所有运算得到的零都去掉负号。
    return {digits, negative};
}

BigInteger readInteger(const string &s) {
    bool negative = s[0] == '-';
    int firstDigit = (s[0] == '-' || s[0] == '+') ? 1 : 0;
    return makeInteger(readDigits(s.substr(firstDigit)), negative);
}

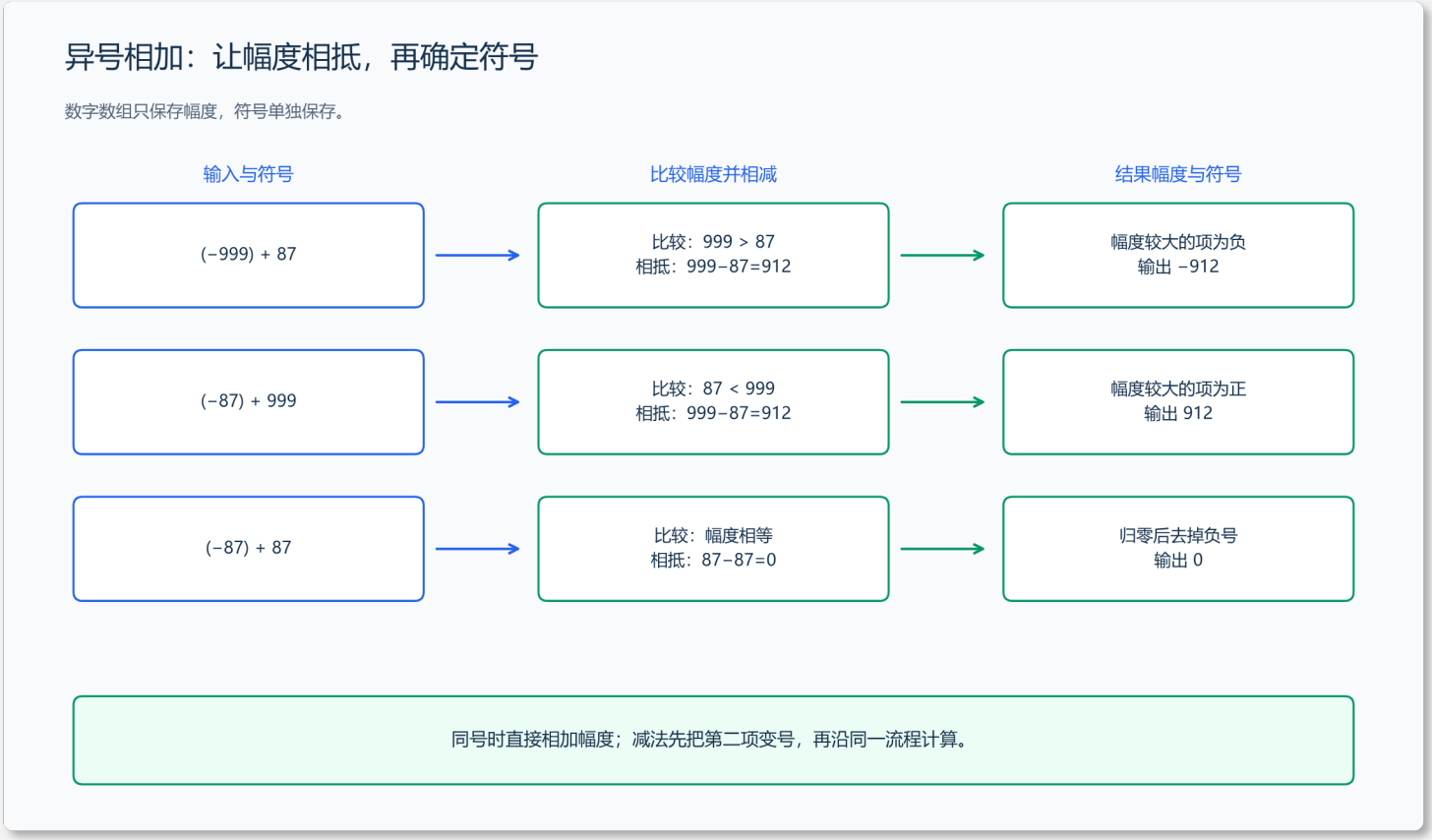
string writeInteger(const BigInteger &a) {
    return (a.negative ? "-" : "") + writeDigits(a.digits);
}
```

`s.substr(firstDigit)` 取数字部分，跳过开头的符号；makeInteger 把幅度和符号合成一个结果，同时统一零的写法。

7.1 同号相加与异号相抵

`(-999)+(-87)` 的两份幅度相加，结果带负号，得到 `-1086`；`(-999)+87` 则让两份幅度抵消，计算 `999-87=912`，符号来自幅度更大的 -999，得到 `-912`。

输入	幅度计算	结果符号	结果
`999+87`	`999+87`	正	`1086`
`(-999)+(-87)`	`999+87`	负	`-1086`
`(-999)+87`	`999-87`	跟随幅度 999	`-912`
`(-87)+999`	`999-87`	跟随幅度 999	`912`
`(-87)+87`	`87-87`	归零后去掉负号	`0`




```cpp
BigInteger addInteger(const BigInteger &a, const BigInteger &b) {
    if (a.negative == b.negative) {
        return makeInteger(addMagnitude(a.digits, b.digits), a.negative);
    }
    int order = compare(a.digits, b.digits);
    if (order ≥ 0) {
        // 异号相抵，幅度较大的 a 决定结果符号。
        return makeInteger(subtractMagnitude(a.digits, b.digits), a.negative);
    }
    return makeInteger(subtractMagnitude(b.digits, a.digits), b.negative);
}

BigInteger subtractInteger(const BigInteger &a, BigInteger b) {
    if (!isZero(b.digits)) b.negative = !b.negative;
    return addInteger(a, b); // A-B 等于 A+(-B)，复用已推导的加法。
}
```

减法解释： `728-1000` 变成 `728+(-1000)`，幅度 1000 较大，计算 272 并取负号；`728-(-1000)` 变成 `728+1000`，两份正幅度相加，得到 1728。

7.2 乘除符号与整数余数

乘法和除法的商，同号为正、异号为负。因此用 `a.negative ≠ b.negative` 判断结果是否为负。除法先对两份幅度求商和余数，再为商设置乘除符号，为余数设置被除数符号。

A 与 B	幅度除法	带符号商 / 余数	反算
17 与 5	商 3，余 2	`3 / 2`	$17 = 5 \times 3 + 2$
-17 与 5	商 3，余 2	`-3 / -2`	$-17 = 5 \times (-3) + (-2)$
17 与 -5	商 3，余 2	`-3 / 2`	$17 = (-5) \times (-3) + 2$
-17 与 -5	商 3，余 2	`3 / -2`	$-17 = (-5) \times 3 + (-2)$

```cpp
BigInteger multiplyInteger(const BigInteger &a, const BigInteger &b) {
    return makeInteger(multiplyMagnitude(a.digits, b.digits),
                       a.negative != b.negative); // 异号相乘得到负数。
}

pair<BigInteger, BigInteger> divideInteger(const BigInteger &a,
                                           const BigInteger &b) {
    auto [quotient, remainder] = divideMagnitude(a.digits, b.digits);
    return {makeInteger(quotient, a.negative != b.negative),
            makeInteger(remainder, a.negative)}; // 余数随被除数符号，零由统一函数整理。
}
```

这种除法得到整数商，不输出小数部分；符号处理与 C++ 整数 `/`、`%` 的向零截断结果一致。`(-2)/5` 的幅度商为 0，整理后输出 `0 -2`。

8. 组装计算器与核对结果

依次放入数字辅助函数、幅度加减乘除、BigInteger 与符号函数，最后加入下面的 main。每个函数都有一个职责：解析、计算幅度、处理符号或输出。独立的 `divideByInt` 可以另存为模板，也可以放在同一文件中。

```cpp
int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int t;
    cin >> t;
    while (t--) {
        char operation;
        string first, second;
        cin >> operation >> first >> second; // 超长整数先按文本读入。
        BigInteger a = readInteger(first);
        BigInteger b = readInteger(second);
        if (operation == '+') {
            cout << writeInteger(addInteger(a, b)) << '\n';
        } else if (operation == '-') {
            cout << writeInteger(subtractInteger(a, b)) << '\n';
        } else if (operation == '*') {
            cout << writeInteger(multiplyInteger(a, b)) << '\n';
        } else if (isZero(b.digits)) {
            cout << "DIV0\n"; // 除数 0 在进入长除法前单独输出。
        } else {
            auto [quotient, remainder] = divideInteger(a, b);
            cout << writeInteger(quotient) << ' '
```

```cpp
                    << writeInteger(remainder) << '\n';
            }
        }
    }
```

样例逐项解释： 前四项对应进位、借位、位乘积累加和长除法的过程图；第五项先比较幅度再相减，得到负数；第六项两份幅度相抵后留下 912，结果带负号；第七项给幅度商 3 与余数 2 设置符号；第八项把 `-000` 整理为零，再输出 `DIV0`。

再用下面的输入检查前导零、相消、连续进位、连续借位和商中间的零：

```text
6
+ 0000 -000
- -87 -87
+ 99999999999999999999 1
- 10000000000000000000 1
* -00012 +003
/ 1005 5
```

输出为：

```text
0
0
10000000000000000000
99999999999999999999
-36
201 0
```

检查除法时，还可以把输出商 Q、余数 R 代回 $A = BQ + R$，再检查余数的幅度小于除数幅度。例如 `12045=37×325+20`，且 20 小于 37。只反算等式还不够：`12045=37×324+57` 也成立，但 57 还够再减一次 37，因此商 324 尚未算完。

9. 按数字位数估算复杂度与选用模板

这里的规模是**数字位数**，不是整数本身的数值。两份 2000 位整数，用 n、m 表示位数；`n=2000`，而不是令 n 等于输入的大整数。

操作	时间	结果与临时存储	工作量来源
解析、输出	$O(n)$	$O(n)$	每位字符处理一次
比较、加法、减法	$O(\max(n, m))$	比较 $O(1)$；加减 $O(\max(n, m))$	一趟逐位扫描与进位、借位
大整数乘大整数	$O(nm)$	$O(n+m)$	两层循环，共 nm 个位乘积
大整数除以大整数	$O(nm)$	$O(n+m)$	n 次接位，每位至多 9 次 m 位减法
大整数除以普通整数	$O(n)$	商 $O(n)$、余数 $O(1)$	余数直接用普通整数计算

若两个整数都有 2000 位，乘法做约 400 万次位乘法；若都有 20000 位，则变成约 4 亿次。提高输入速度不会减少这些位运算，选择方法时仍要把位数代入估算。

数字数组的每个 int 通常占 4 字节，2000 位约用 8000 字节；两个输入和乘积的数字存储合计约 32000 字节，另有文本、vector 本身和计算时的临时对象。用 string 保存输入，不代表把 string 转成 long long 后再计算；正是逐位数组让全部数字都能参与运算。

接下来遇到阶乘、组合计数或递推结果越来越长时，可以复用高精度加乘；遇到除数较小的题目，则选逐位普通整数除法。对应练习分别见加、减、乘、除各节，综合练习可以先计算短数据核对四种结果，再逐步增加数字位数。

（三） 解题和自查顺序

1. **读题：** 写清输入、输出、约束和是否有多组数据。
2. **估算复杂度：** 根据数据规模选算法，再决定是否需要优化 I/O。
3. **先构造样例：** 包括最小输入、重复值、负数、全相同、递增或递减数据。
4. **明确不变量：** 例如二分区间、滑动窗口的有效条件、DP 数组的含义。
5. **检查类型：** 累计和、乘法、坐标差和计数是否需要 `long long`。
6. **核对输出：** 按题目格式输出空格、换行、小数精度、大小写和无解标记。

评测结果	含义	常见检查方向
AC	Accepted，通过	仍可回顾正确性和复杂度
WA	Wrong Answer，答案错误	条件判断、状态定义、漏掉的情况
TLE	Time Limit Exceeded，超时	算法复杂度、重复计算、输入输出
MLE	Memory Limit Exceeded，内存超限	大数组、类型宏、重复存储
RE	Runtime Error，运行错误	越界、除零、栈深度、非法访问
CE	Compilation Error，编译错误	语法、头文件、标准版本和接口

在本地开启编译警告；需要排查容器越界时，可在支持的 GCC / libstdc++ 环境下使用 `-D_GLIBCXX_DEBUG`。本地调试配置不一定适合直接提交，提交前再按评测环境编译。

五、数据结构与算法

先用已学的数组、循环和函数写一个能得到答案的做法，再数它需要多少工作。之后每种算法都从“哪里重复、哪些候选可排除、哪些状态可复用”的具体问题推进。

（一） 时间复杂度与空间复杂度

同一个题目可以用一次遍历，也可以用两层枚举；先把输入规模代入，比较工作量与同时占用的存储，再决定哪种做法能用于最大数据。

先判断一个方法能不能处理题目的最大数据，再决定是否值得实现。复杂度描述输入规模增大时，计算量和内存需求如何增长；它不是程序实际用了多少秒、多少 MiB 的直接测量值。

1. 输入规模、基本操作与大 O

输入规模要先定义清楚。数组常用元素数 `n`，字符串匹配可能需要两个长度 `n、m`，图算法需要顶点数 `V` 和边数 `E`，背包问题还需要容量 `C`。不能把所有题目的规模都简化成一个 `n`。

估算时间时，可以选出反复执行的**基本操作**，例如一次比较、一次加法或一次状态转移，统计它随规模增长的次数。我们通常假设机器整数运算、数组下标访问的单次代价为常数；若一次操作需要遍历字符串、复制数组或计算大整数，就要继续展开它的代价。

例如某段程序的操作次数是 $3n^2+5n+7$。随着 `n` 增大，平方项主导增长，常数系数不影响增长阶，所以通常记作 $O(n^2)$。忽略常数是为了比较增长趋势，实际判断能否通过时仍要考虑常数。

严格地说，$O(f(n))$ 表示一个渐近上界：存在正常数 `c` 和起始规模 `n₀`，使足够大的 `n` 上有 $T(n) \le cf(n)$。若上下界都与同一个函数成常数比例，则可以写 $\Theta(f(n))$。本笔记通常用大 O 表示分析得到的常用增长量级；遇到仅有上界、最坏情况或均摊情况时会另作说明。

对数的底数只差一个常数，所以复杂度中通常写 $O(\log n)$。**手算时统一用** $\log_2 n$，因为“每轮减半”正好对应二进制位数。

> 注：根据对数换底公式
>
> $$\log_a x = \frac{\log_p x}{\log_p a}$$
>
> 可知，对于任意固定底数 a 的对数，要换底为 p，相差的常数为：
>
> $$\frac{1}{\log_p a}.$$
>
> 也就是说，
>
> $$\log_a x = \frac{1}{\log_p a}\log_p x.$$

反过来，

$$\log_p x = \log_p a \cdot \log_a x,$$

此时相差的常数为 $\log_p a$。这些常数都与 x 无关，所以不会改变增长量级。

特别地，换成以 2 为底时：

$$\log_a n = \frac{\log_2 n}{\log_2 a} = \frac{1}{\log_2 a}\log_2 n.$$

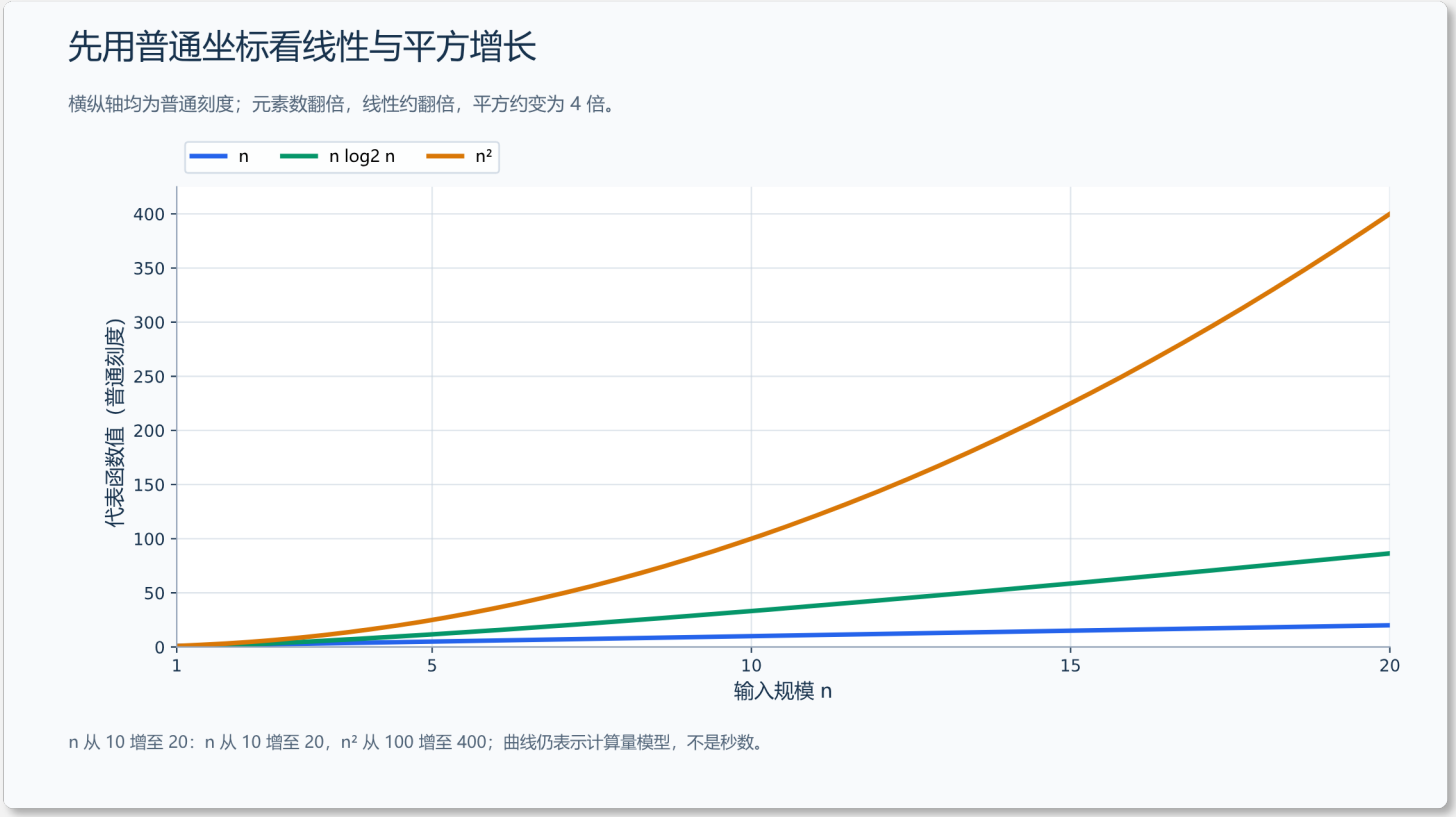
因此

$$O(\log_a n) = O(\log_2 n).$$

2. 增长曲线与数量对照表

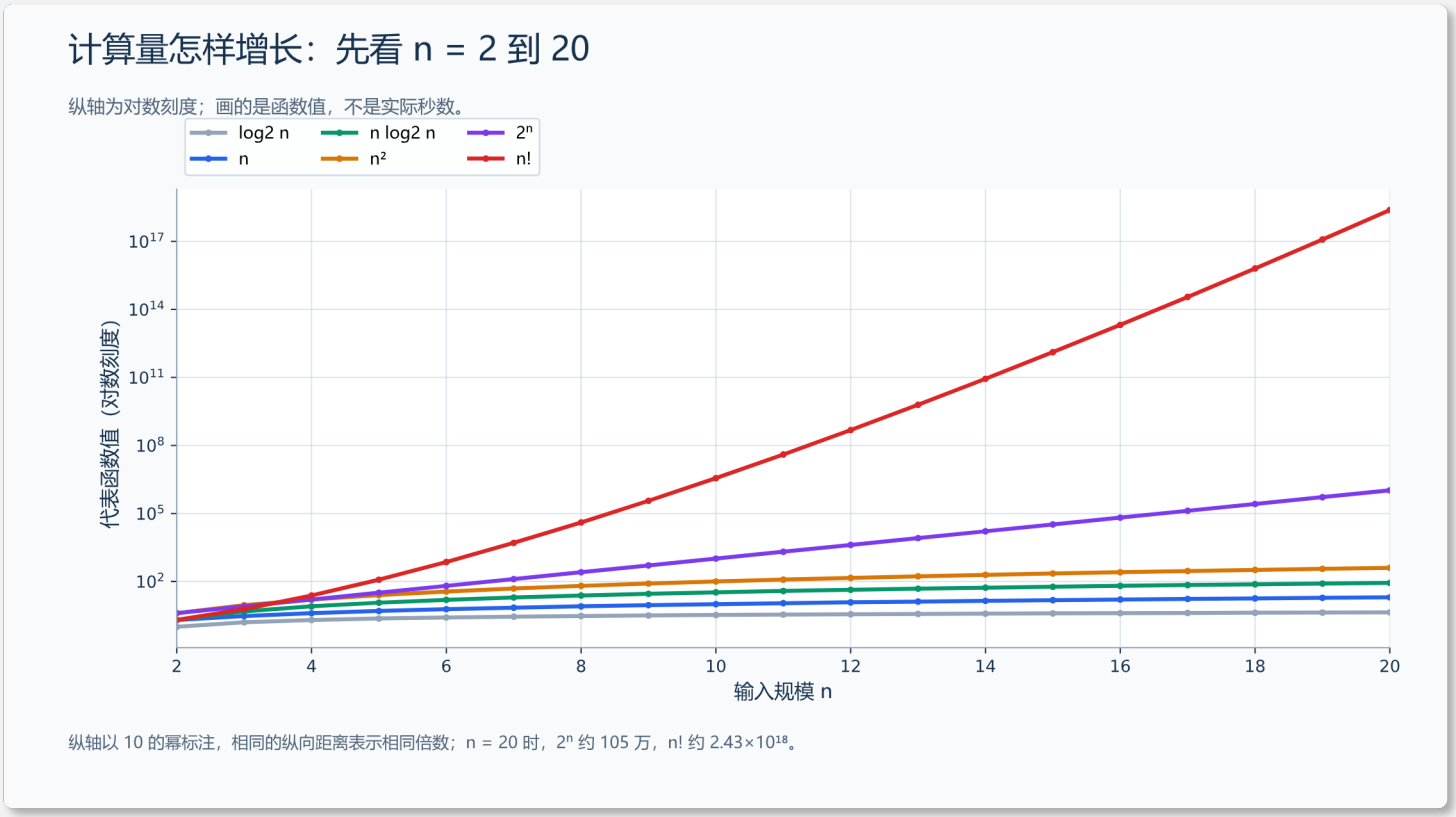
下面用函数值比较不同增长速度，统一采用 `log₂ n`，小数取近似值。曲线纵轴表示工作量随 n 的增长趋势，实际耗时还取决于每步操作与运行环境。

增长量级	常见场景	n 翻倍时的变化
$O(1)$	已知下标访问一个元素	基本不变
$O(\log n)$	二分查找、按位快速幂	对数值增加约 1
$O(n)$	一次遍历、建立前缀和	约 2 倍
$O(n \log n)$	归并排序、常用比较排序	略多于 2 倍
$O(n^2)$	枚举所有数对、一些二维 DP	约 4 倍
$O(n^3)$	Floyd、三重全范围枚举	约 8 倍
$O(2^n)$	每个位置选或不选	比原来再乘 2^n；n 增加 1 时才是 2 倍
$O(n!)$	无重复地枚举全部排列	n 增加 1 时乘 n+1



先看普通刻度图：`n` 从 10 增到 20，线性函数从 10 增到 20，平方函数从 100 增到 400。平方曲线越来越陡，说明相同的输入增量会带来越来越多的计算。

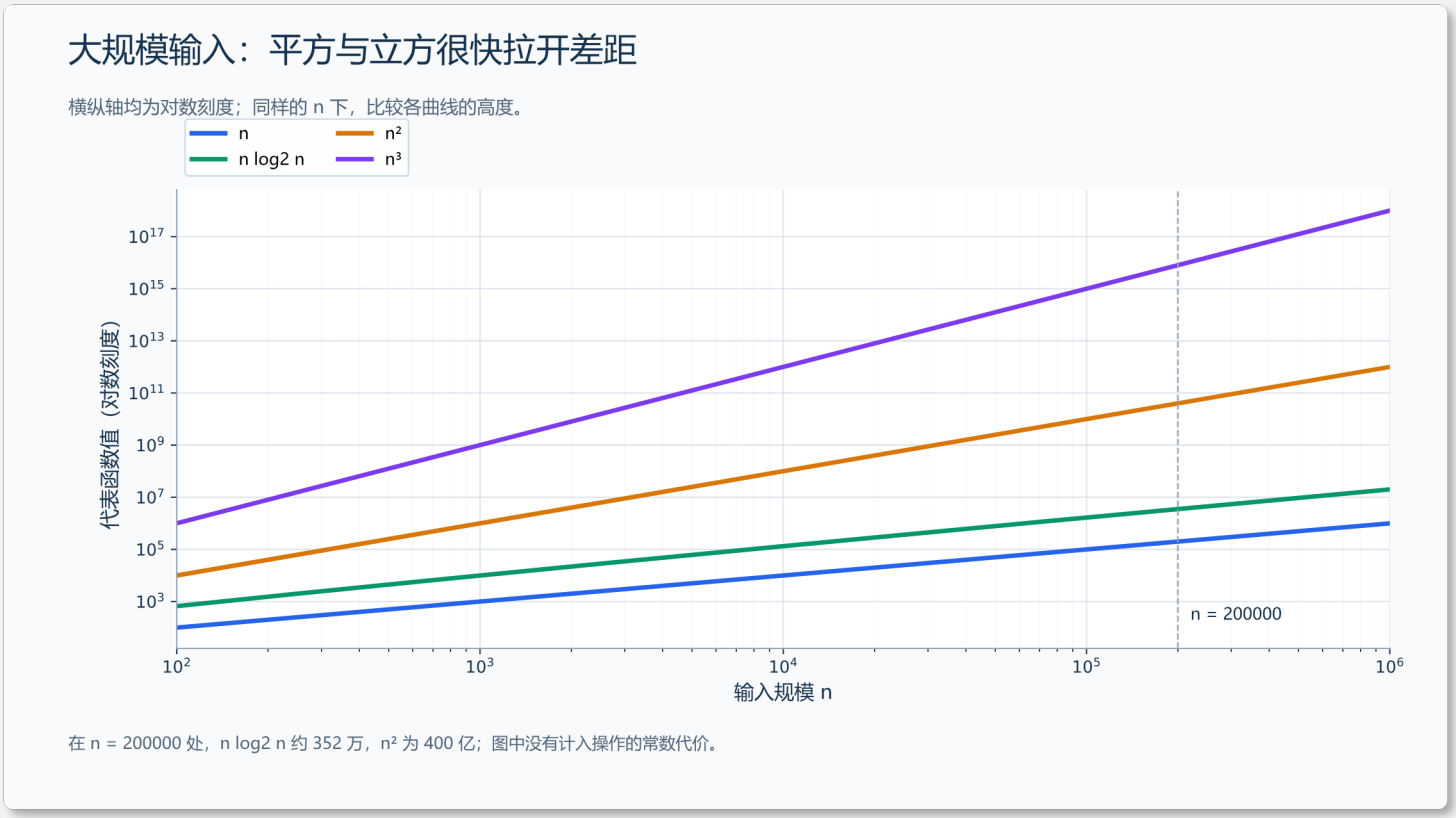
加入指数和阶乘后，各函数差距太大，普通刻度会把较小曲线挤在底部，所以接下来使用对数刻度。



图中横轴是元素数，纵轴使用对数刻度：刻度以 10 的幂标注，相同的纵向距离表示相同的倍数变化。指数和阶乘很快拉开差距；阶乘曲线在对数纵轴上也仍明显加速上升。

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3
10	3.32	10	33.2	100	1,000
100	6.64	100	664	10,000	1,000,000
1,000	9.97	1,000	9,966	1,000,000	1,000,000,000
10,000	13.29	10,000	132,877	100,000,000	10^{12}
200,000	17.61	200,000	3,521,928	40,000,000,000	8×10^{15}

这些数只是代表函数值。例如枚举 `i < j` 的次数是 $n(n-1)/2$，不是恰好 `n²`；排序比较次数也不必恰好等于 `n log₂ n`。增长表用于先分辨数量级，再按具体代码细算。



这张图的两个轴都使用对数刻度，适合同时展示跨越多个数量级的数据。图上线性函数也会向上增长；曲线看上去近似直线不代表它们的复杂度都是线性。

n	子集数 2^n	排列数 $n!$
10	1,024	3,628,800
15	32,768	1,307,674,368,000
20	1,048,576	2,432,902,008,176,640,000
25	33,554,432	约 1.55×10^{25}

不要漏算“每种情况的处理成本”。 枚举全部子集共有 `2ⁿ` 种，如果每次再扫描 `n` 位，时间是 $O(n2^n)$；输出全部排列时，每个排列要输出 `n` 个数，时间至少与 $n \cdot n!$ 个输出元素相对应。

3. 顺序、嵌套与变长循环的复杂度

先数循环体真正执行了多少次，再化简。不要仅凭“有几层循环”给出结论。

顺序执行取总和。 一次长度 `n` 的遍历后，再做一次长度 `m` 的遍历，总时间是 $O(n+m)$。如果 `m=n`，才可以化简成 $O(n)$。排序后再遍历一遍是 $O(n\log n+n)=O(n\log n)$；不是把两个代价相乘。

独立的嵌套范围相乘。 外层跑 `n` 次，每次内层跑 `m` 次，而且每次循环体是常数代价，合计 `nm` 次，时间为 $O(nm)$。当两个长度都为 `n` 时才是 $O(n^2)$。

内层长度变化时求和。 枚举 `0 ≤ i < j < n` 时，内层次数依次是 `n-1、n-2、…、0`，因此：

$$T(n)=(n-1)+(n-2)+\cdots+1+0=\frac{n(n-1)}{2}=O(n^2)$$

虽然只检查了所有有序数对的一半，增长阶仍是平方级。如果内层每次从 `0` 跑到 `i`，同样要计算 $1+2+\cdots+n$，不能因为内层不是固定 `n` 次就判为线性。

每轮减半时算轮数。 从正整数 `n` 开始，反复执行整数除法 `x /= 2`，直到 `x=0`，共执行 $\lfloor\log_2 n\rfloor+1$ 次，所以是 $O(\log n)$。例如 `20→10→5→2→1→0`，执行 5 轮。若每轮还扫描整个长度 `n` 的数组，则总时间变成 $O(n\log n)$。

工作规模减半时求几何和。 若各轮扫描的长度分别是 `n、n/2、n/4、…`，总扫描量小于 `2n`，所以是 $O(n)$。它与“每轮都扫描完整数组”的情况不同。

下面用一个完整例子实际数次数。

题目原文（自拟）： 输入 `1 ≤ n ≤ 1000`。分别统计一次遍历、枚举所有 `i < j`、将 `n` 反复整除 2 直到 0、外层 `i=1…n` 且内层执行 `i` 次这四个过程的循环体执行次数。按上述顺序输出四个整数，每个占一行。

分析： 为每个过程设置单独计数器；只统计循环体，不把循环条件检查和计数器更新拆成更多操作。这样能对照推导验证循环次数，但不等于测出了程序耗时。

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

int main() {
    int n;
    cin >> n;
    long long linear = 0, pairs = 0, halving = 0, triangle = 0;

    for (int i = 0; i < n; ++i) {
        ++linear; // 每个元素对应一次循环体执行。
    }
    for (int i = 0; i < n; ++i) {
        for (int j = i + 1; j < n; ++j) {
            ++pairs; // 只枚举不同下标，每个无序数对出现一次。
```

```cpp
        }
    }
    for (int remaining = n; remaining > 0; remaining /= 2) {
        ++halving; // 每轮减半，轮数等于正整数 n 的二进制位数。
    }
    for (int i = 1; i <= n; ++i) {
        for (int j = 0; j < i; ++j) {
            ++triangle; // 内层次数随 i 增加，累加得到 1+2+...+n。
        }
    }
    cout << linear << '\n' << pairs << '\n' << halving << '\n' << triangle << '\n';
}
```

输入：

```text
5
```

输出：

```text
5
10
3
15
```

样例与代码解释： 数对次数为 `4+3+2+1+0=10`；减半过程为 `5→2→1→0`，执行 3 轮；最后一段为 `1+2+3+4+5=15`。四段依次运行，总时间是 $O(n+n^2+\log n+n^2)=O(n^2)$，计数器的额外空间为 $O(1)$。

4. 双指针、图遍历与 DP 的总工作量

双指针看总移动次数。 例如滑动窗口中，外层让右端从左到右前进，内层在必要时收缩左端。只要两个指针都不后退，每个指针最多移动 `n` 次，总移动次数不超过 `2n`。因此，即使存在嵌套的 `while`，每次只做常数次加减或赋值，总时间就是 $O(n)$。如果每轮把左端重置后再重新扫描，就失去了这个结论。

邻接表遍历看所有边的总量。 DFS / BFS 对每个顶点最多进入或入队一次，合计 $O(V)$；各顶点的邻接表长度相加为 `E`，无向图通常是 `2E`。因此总时间为 $O(V+E)$。只有用邻接矩阵并对每个顶点扫描全部 `V` 列时，才会出现 $O(V^2)$ 的扫描量。

DP 看“状态数量 × 每个状态的转移数量”。 网格路径有 `R×C` 个状态，每个只检查上方、左方，时间为 $O(RC)$。最少硬币有 `M+1` 个金额状态，每个尝试 `k` 种硬币，时间为 $O(kM)$。背包的 `C` 是容量数值，不是物品个数；即使 `n` 很小，`C=10^9` 也不能直接开表逐容量更新。

库函数也要算进去。 `sort` 需要 $O(n\log n)$ 量级的比较；在 `n` 轮循环中每轮重新排序完整数组，便会得到 $O(n^2\log n)$ 量级的比较次数。`string` 的比较在最坏情况下需要检查多个字符；比较很长的字符串时，不能继续把每次比较当成一个固定大小整数比较。

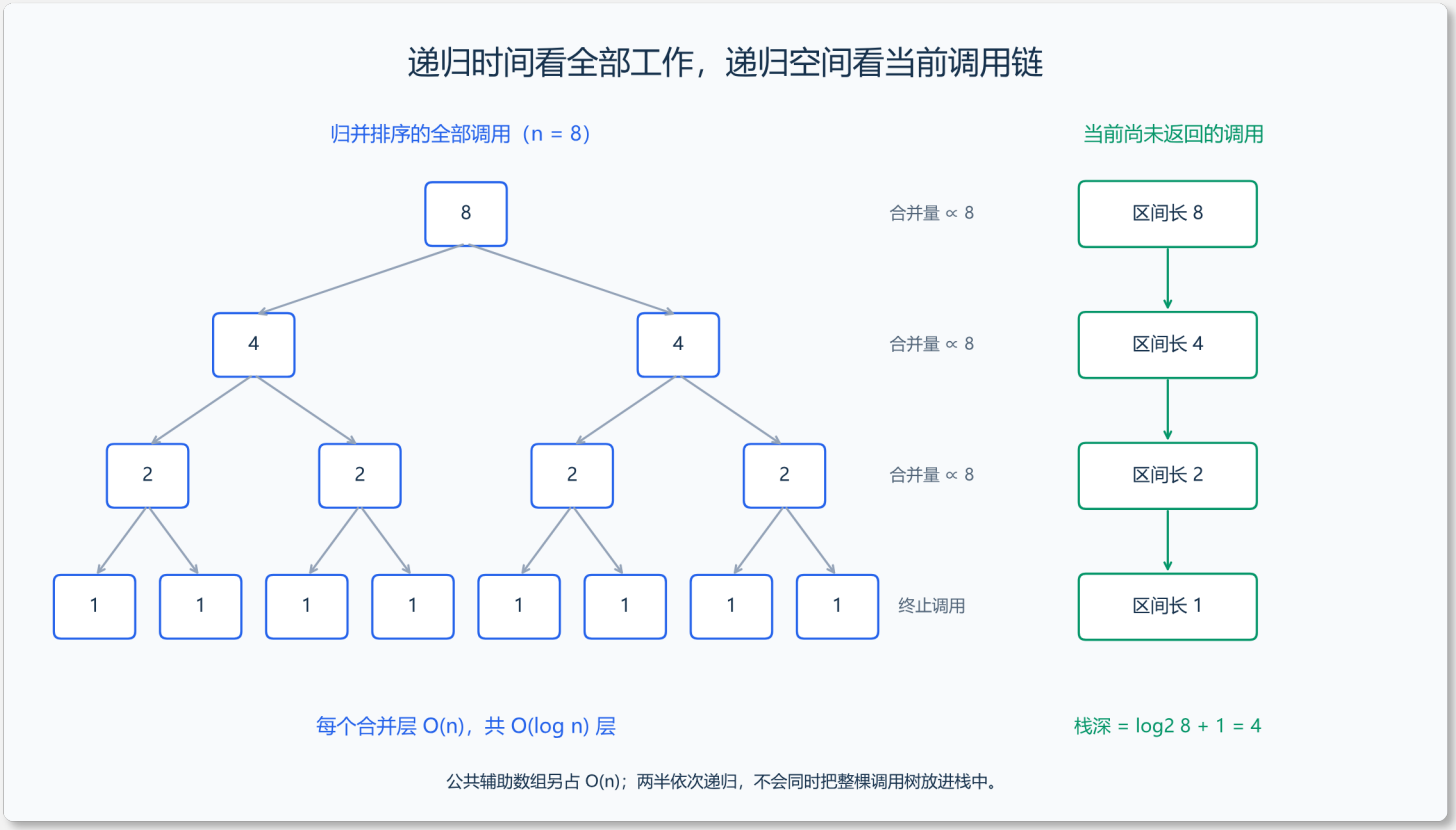
5. 递归的时间与空间要分开算

递归时间要统计所有调用的工作，递归空间要统计同时还没返回的调用。总调用数和最大调用深度通常不同。

归并排序每层把数组分成更小的区间，但一层中所有区间的长度加起来仍是 `n`。合并一层共 $O(n)$ 工作，层数为 $O(\log n)$，所以：

$$T(n) = 2T(n/2) + O(n) = O(n \log n)$$

递归时先做左半，再做右半，并不同时保留整棵调用树。因此单个调用只保留常数个局部变量时，递归栈为 $O(\log n)$。本笔记的归并排序另有长度 `n` 的公共辅助数组，额外空间合计为 $O(n + \log n) = O(n)$。



左图把全部调用按层画出，用来计算时间；右图只画当前尚未返回的一条调用链，用来计算递归栈。图以 `n=8` 为例，包含长度为 1 的终止调用。

朴素递归斐波那契会反复计算相同参数，调用树有指数级数量的节点，常写上界 $O(2^n)$，但单条调用链深度仍为 $O(n)$。加入记忆化后，每个参数只展开计算一次，时间降为 $O(n)$；缓存和递归栈各为 $O(n)$，总额外空间仍为 $O(n)$。

> **注意：** 不能看到“递归参数每次减半”就直接写 $O(\log n)$。还要检查每次递归调用几个子问题、每个节点额外做了多少工作。若每个调用还保存长度与当前问题相同的数组，栈中的这些数组也要计入空间。

6. 数据规模与算法选择

做题时可以优先寻找 $O(n \log n)$、$O(n)$ 的方法；规模较小时，$O(n^2)$ 方法常常足够，而且更容易写对。**需要控制的是总运算量和内存，而不是固定把每道题压到某一种复杂度。** 读取 `n` 个数本身通常就要线性时间；要求输出所有数对的题目，其输出量已是平方级。

把主要规模代入增长式，就能先比较候选方案需要多少工作，再结合每次操作的内容估时：

主要规模	可以先考虑什么	仍需计算什么
`n ≤ 10`	全排列、回溯	`10!` 已约 363 万种，每种还有处理或输出代价
`n ≤ 20`	子集枚举、状态压缩	`2²⁰` 约 105 万；再扫描 20 位约 2,097 万次检查
`n` 为几百	平方方法；简单立方方法也可估算	`500³` 为 1.25 亿次，未必轻松，需要看具体操作与时限
`n` 为几千	枚举数对、平方 DP	`2000²` 为 400 万；但有大常数或多组数据时仍可能过慢
`n` 为十万量级	线性、排序、二分、堆等	`200000²` 为 400 亿，通常要改进平方枚举
`n` 为百万量级	先考虑线性；简单排序也要具体估算	还要检查输入输出、元素大小和辅助数组

手算一个选择过程： 对 `n=200000` 的两数之和，暴力枚举约有

$$\frac{200000 \times 199999}{2} = 19,999,900,000$$

次数对检查。排序后使用双指针，代表量约为

$$n \log_2 n + n \approx 3,521,928 + 200,000 \approx 3.72 \times 10^6$$

两种方案的主要工作量差了约几千倍，应优先考虑排序或合适的哈希方案。若要求原下标，排序时要把原下标和数值一起保存；不能只优化复杂度却丢掉题目要求。若数组本来已经有序，扫描部分只需 $O(n)$。

对 `n=2000`，暴力需要 `1,999,000` 次数对检查。将这个工作量与题目时限、单次检查的成本结合起来，就能判断暴力是否够用，以及是否值得换成更复杂的方法。

实际估时可以按下面的方法做：

1. 用最大合法输入计算主要操作次数 `N`，把排序、预处理、所有询问都加进去。
2. 测量同类操作的处理速率 R，再粗估 $t \approx N/R$。例如每秒处理约 `2×10⁷` 次，`4×10⁶` 次约需 0.2 秒；换一种操作后，再按它的处理速率计算。
3. 给内存访问、分配、输入输出和环境差异留下余量。整数加法、哈希查找、树节点分配的代价不同，不能共用一个“每秒操作次数”。
4. 如果估算已经明显超限，先改算法；快速 I/O 和编译优化通常只能改善常数，不能把平方级增长变成线性对

数级。

给定一个估算的操作预算 B，还可反解候选规模。平方方法从 $n^2 \le B$ 得到 $n \le \sqrt{B}$；线性对数方法把候选 n 代入 $n \log_2 n \le B$ 验算。取 `B=10⁷`，平方模型约得到 3162；`n=500000` 时，`n log₂ n` 约为 `9.47×10⁶`。数对枚举只访问每对一次，就改用实际的 `n(n-1)/2` 计算。

7. 多组数据、最坏情况与均摊

多组数据总耗时是各组代价相加。若第 `i` 组长度为 `nᵢ`，排序总时间为 $O(\sum_i n_i \log n_i)$，平方方法为 $O(\sum_i n_i^2)$。先读清题面是否限制“所有测试的 `n` 之和”，不能总把单组最大值机械乘测试组数。

例如 `T=100`，每组都允许 `n=2000` 且没有总长度限制，数对检查合计可达约 `2×10⁸` 次。若改为 $\sum_i n_i \le 2000$，则因为 $\sum_i n_i^2 \le (\sum_i n_i)^2$，平方量合计不会超过 `4×10⁶`。两种限制会改变算法选择。

表述	含义	例子
最坏复杂度	规模给定时最慢的合法情况	试除判素数遇到素数时要检查到平方根附近
平均 / 期望复杂度	依赖输入分布或随机选择的分析	随机枢轴快速排序的期望时间为 $O(n \log n)$
均摊复杂度	一串操作的总成本除以操作数，不要求随机性	`vector::push_back` 为均摊常数；扩容的某一次可以是线性

哈希容器平均查找常用 $O(1)$，但严重冲突时一次查找可能退化到 $O(n)$；剪枝可能减少实际分支，却不自动改变最坏复杂度。平均表现与最坏表现描述不同的问题：前者观察通常需要多少工作，后者计算最费力的数据会产生多少工作。

8. 空间复杂度与内存用量换算

总空间包括输入保存、答案、辅助结构和调用栈；**额外空间**常指不计已有输入存储的辅助空间。写复杂度时明确口径。例如对已有数组进行双指针扫描，额外空间是 $O(1)$，但数组本身仍占 $O(n)$ 空间。

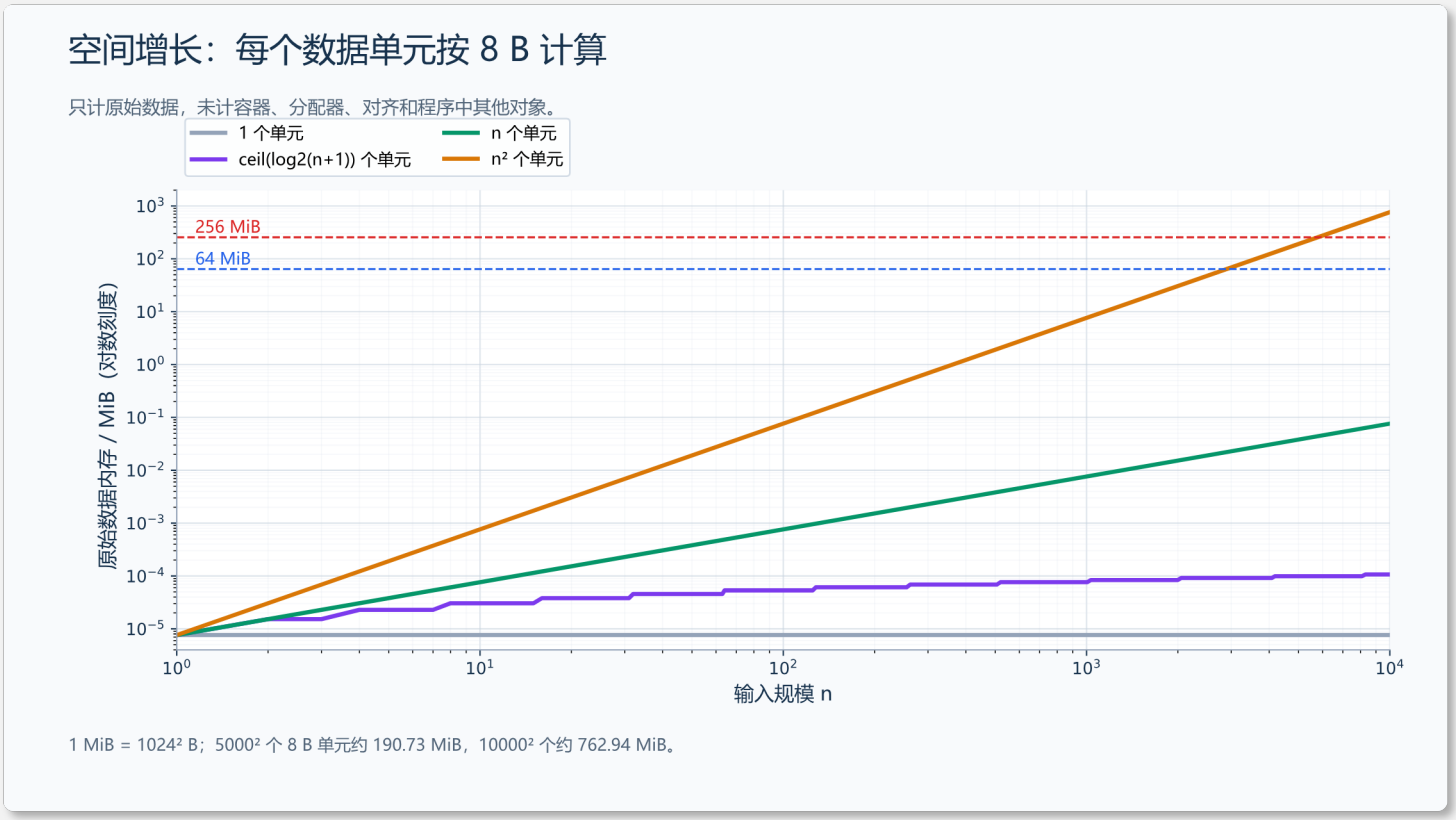
内存预算要计算程序运行时**同时存在的最大存储量**。两个临时数组若确实先后创建并释放，取峰值；若它们同时存活，就相加。函数按值传入一个 `vector` 可能创建副本，递归调用中复制容器可能使实际峰值远大于预期。

先从元素大小计算：

$$\text{数组字节数} \approx \text{元素数量} \times \text{sizeof}(\text{元素类型})$$

`1 KiB = 1024 B`，`1 MiB = 1024² B`。下面假设每个 `int` 为 4 B、每个 `long long` 为 8 B；这是常见竞赛环境的情况，具体环境用 `sizeof` 检查，不作为所有 C++ 实现的规定。

存储内容	数据部分字节数	换算结果
10^6 个 4 B 整数	4×10^6	约 3.81 MiB
10^6 个 8 B 整数	8×10^6	约 7.63 MiB
2000×2000 的 4 B 表	16×10^6	约 15.26 MiB
5000×5000 的 8 B 表	200×10^6	约 190.73 MiB
10000×10000 的 8 B 表	800×10^6	约 762.94 MiB



图中每个存储单元统一假设为 8 B；$\lceil \log_2(n+1) \rceil$ 个单元用来展示对数增长。纵轴为对数刻度，虚线标出 64 MiB、256 MiB。曲线只计算数据单元，尚未计容器对象、分配器、对齐和其他结构的开销。

算一次完整预算： 若内存限制为 256 MiB，5000×5000 的 long long DP 数据就约为 190.73 MiB，再保存同尺寸 4 B 输入约 95.37 MiB，两份数据同时存在已约为 286.10 MiB，超过限制。不能看到“DP 单独不到 256 MiB”就判断足够。

vector<vector<T>> 还有每行 vector 对象和单独分配的开销；vector 的容量可能大于实际元素数；map、set、哈希表和 priority_queue 也有额外存储。vector<bool> 的元素常按位压缩，不能简单用 n*sizeof(bool) 推断它的实际占用。

递归栈按“同时保留的层数 × 每层栈帧大小”估算。例如 DFS 沿十万个顶点组成的链递归，会同时保留十万层调用，每层的参数、局部变量和返回位置都占栈空间。改用显式栈后，这些待处理状态保存在容器中；改成迭代 DP 后，则按依赖顺序逐项计算。

从依赖关系减少存储。 二维 DP 的当前行只读上一行时，保留两行就能把表空间从 $O(RC)$ 降到 $O(C)$。恢复完整路径时可以另存前驱，读取更多历史行时则保留对应的行。背包还能压成一行：01 背包逆序更新，完全背包正序更新；后文会从“这一轮读到哪个值”推导这两种方向。

9. 算法实现的复杂度核对

按下面顺序计算一遍，最后把结论写在算法旁边：

1. 定义所有规模变量：`n`、`m`、`q`、`V`、`E`、`C` 分别表示什么。
2. 找出主要操作，检查它本身是否真是常数代价。
3. 用求和、乘法或递归树统计总次数，补上预处理和每个询问。
4. 代入最大数据，计算具体数量级；有多组时算总和。
5. 列出同时存活的数组、容器、副本和递归栈，估算内存峰值。
6. 分清最坏、期望与均摊复杂度，再对照题目的最大规模估算总工作量。

例如前缀和处理长度 `n` 的数组和 `q` 次询问，总时间为 $O(n+q)$、额外空间 $O(n)$；不是只写“每次查询 $O(1)$”就结束。二分答案若每次判定扫描 `n` 个元素、候选范围宽度为 `W`，应写 $O(n\log(W+1))$，不能漏掉判定函数的成本。

练习题： 洛谷 P1177 – 排序、洛谷 P1102 – A-B 数对。先写出各方法的时间、空间和最大规模估算，再选择实现；比较时要遵守原题对计数和重复元素的要求。

（二） 枚举与模拟

先从能够逐项检查的候选开始写正确解，确认没有漏掉，再计算枚举范围；后面的技巧会围绕这种基准做法减少重复工作。

枚举是把可能的情况依次检查；**模拟**是按题目规则直接执行过程。它们是构造解法和验证复杂算法的起点。

1. 枚举数对

题目原文（自拟）： 给定整数数组 `values` 和目标 `target`，返回两个不同的从 `0` 开始的下标，使对应元素之和等于目标。任意一组合法下标均可；无解返回 `(-1, -1)`。数组长度不超过 `2000`，元素是 `int`，目标是 `long long`。

分析： 枚举所有 `i < j` 的数对，计算和并比较目标。先写最直接的 $O(n^2)$ 方法：

为什么这样枚举不会漏解？ 任意两个不同下标，总能把较小的记为 `i`、较大的记为 `j`，所以它一定落在 `i < j` 的范围内；反过来，每次枚举出的数对下标确实不同。这样建立了“合法候选与枚举范围”的对应关系。

对 `[3,1,5]`、目标 `6`，依次检查 `(0,1)` 的和 `4`、`(0,2)` 的和 `8`、`(1,2)` 的和 `6`，此时返回即可，因为题目只要求任意一组。若题目改成“有多少对”或“输出全部数对”，就不能第一次找到后直接返回。最坏情况下无解，需要检查全部 $n(n-1)/2$ 对。



按 `j>i` 枚举不同位置，不会遗漏无序数对。

```cpp
#include <utility>
#include <vector>
using namespace std;

pair<int, int> twoSumBrute(const vector<int> &values, long long target) {
    int n = static_cast<int>(values.size());
    for (int i = 0; i < n; ++i) {
        for (int j = i + 1; j < n; ++j) {
            // j 从 i 后面开始，保证下标不同；先提升类型再相加。
            if (1LL * values[i] + values[j] == target) {
                return {i, j};
            }
        }
    }
    return {-1, -1};
}
```

`j` 从 `i + 1` 开始，保证不重复使用同一个元素，也不重复检查 `(i, j)` 与 `(j, i)`。数组有序时，可以使用后面的双指针降到 $O(n)$；无序数组也可用哈希记录已经出现的数。

样例与解释： `values = [3, 1, 5]`、`target = 6`，返回 `(1, 2)`；`values = [3]`、`target = 6` 无解，同一个元素不能使用两次。额外空间为 $O(1)$。

2. 枚举子集

题目原文（自拟）： 给定不超过 20 个整数，每个位置最多选择一次，输出所有子集的元素和，包含空集；按子集的二进制编号递增输出。所有求和结果在 `long long` 范围内。下面用 `[1, 2, 4]` 演示。

分析： 用二进制的一位表示“是否选择对应元素”，位置 `i` 对应第 `i` 位。



每个位置对应一位，重复数值也仍是不同位置；空集的编号和元素和都是 0。



以 `[1,2,4]`、`mask=5` 为例，二进制为 101，最低位对应位置 0。检查第 0 位为 1，加入 1；第 1 位为 0，跳过 2；第 2 位为 1，加入 4，最终和为 5。图中的“选择”来自 `mask & (1ULL<<i)` 的判断，不是看到某个目标和后反推子集。

把每个决定对应到一位，才知道枚举是否完整。 对长度 `n` 的数组，每个位置只有选、不选两种决定，合计 2^n 种。范围 $0 \le mask < 2^n$ 恰好覆盖所有 `n` 位二进制串；同一个串与同一组位置一一对应，不会漏掉空集或重复某组位置。

`mask & (1u << i)` 只保留第 `i` 位：结果非零就选择 `values[i]`。例如 `mask=5` 对应二进制 `101`，从低位起分别为选、不选、选，因此取第 0、2 个元素。即使两个元素数值相同，不同位置仍代表不同选择；若题目要求按数值去重，不能直接把位置枚举的个数当成答案。

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

vector<int> values{1, 2, 4};
int n = static_cast<int>(values.size());
for (unsigned int mask = 0; mask < (1u << n); ++mask) {
    long long sum = 0;
    for (int i = 0; i < n; ++i) {
        if ((mask & (1u << i)) != 0) {
            sum += values[i];
        }
    }
    cout << sum << ' '; // 空集编号为 0，和也为 0。
}
cout << '\n';
```

共有 2^n 个子集，上面每个子集再扫描 `n` 位，总时间为 $O(n2^n)$。模拟题则先明确状态，例如当前位置、方向、时间，再严格按照规则更新。

样例与解释： 输出 `0 1 2 3 4 5 6 7`。编号 `5 = (101)₂` 选择第 0、2 个元素，和为 `1 + 4 = 5`。本片段放入 `main` 中运行。

练习题： 洛谷 P2089 - 烤鸡、洛谷 P1554 - 梦中的统计、洛谷 P1036 - 选数。

（三） 前缀和与差分

若每次区间查询都重新累加，会反复访问相同元素。先保存从起点到各位置的累计结果；若问题改成很多次区间修改，再反过来记录边界变化。

1. 一维前缀和

如果需要多次查询区间和，可以先累计数组前缀。

题目原文（自拟）： 给定整数数组和若干合法半开区间 `[left, right)`，求各区间的元素和；空区间的和为 0。所有前缀与区间和可放入 `long long`。

定义 `prefix[i]` 为前 `i` 个元素的和，`prefix[0] = 0`：

$$prefix[i+1] = prefix[i] + a[i]$$

那么半开区间 `[left, right)` 的和为：

$$sum(left, right) = prefix[right] - prefix[left]$$

从重复计算推导前缀状态。 若每个询问都扫描区间，`q` 次询问最坏需要 $O(nq)$。而这些区间包含许多重复的前缀，可以把“前 `i` 个元素的累计和”保存下来。

`prefix[right]` 包含下标 `0…right-1`，`prefix[left]` 包含 `0…left-1`。两者相减，前面相同的部分抵消，恰好留下 `left…right-1`。这也解释了为什么半开区间直接用端点，而闭区间 `[l,r]` 应改成 `prefix[r+1]-prefix[l]`。

多留 `prefix[0]=0` 让左端为 0 时也能使用同一公式。`left=right` 时两项相等，空区间自然得到 0。数组有负数也不影响相减公式；前缀和查询并不要求数组元素为正。

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

vector<long long> buildPrefix(const vector<int> &values) {
    // prefix[i] 表示前 i 个元素的和，多保留一个初始 0。
    vector<long long> prefix(values.size() + 1, 0);
    for (size_t i = 0; i < values.size(); ++i) {
        prefix[i + 1] = prefix[i] + values[i];
    }
    return prefix;
}
```



例：`a = [2, 1, 3, 4]`，`prefix = [0, 2, 3, 6, 10]`，区间 `[1, 4)` 的和为 `10 - 2 = 8`。



从 `prefix[0]=0` 开始，每读入一个元素，就把它接到此前的累计和上，依次写出 2、3、6、10。查询 `[1,4)` 时，`prefix[4]` 包含位置 0…3，`prefix[1]` 恰好包含不需要的位置 0，相减得到 8；这也解释了前缀数组为何比原数组多一格。

预处理 $O(n)$，每次查询 $O(1)$，额外空间 $O(n)$。原数组频繁修改时，这份前缀和不会自动更新。

练习题： 洛谷 P8218 – 求区间和↗。

2. 二维前缀和

题目原文（自拟）： 给定一个矩形整数矩阵及若干合法子矩形，求每个子矩形的元素和。矩形用半开范围 `[r1, r2) × [c1, c2)` 表示，所有累计值和计算中间值均在 `long long` 范围内。

约定原矩阵从下标 `0` 开始；`prefix[i][j]` 表示左上角 `i` 行、`j` 列的元素和：

$$prefix[i+1][j+1] = a[i][j] + prefix[i][j+1] + prefix[i+1][j] - prefix[i][j]$$

矩形 `[r1, r2) × [c1, c2)` 的和为：

$$prefix[r2][c2] - prefix[r1][c2] - prefix[r2][c1] + prefix[r1][c1]$$

加上两个前缀时左上区域被重复计算，所以要减掉一次；查询时则使用同样的容斥思想。



对值为 1…9 的 3×3 数组查询 `[1,3)×[1,3)`：先取大矩形和 45，减上方 6 得 39，再减左方 12 得 27；左上角的 1 被减了两次，加回后为 28。四幅图标出每个格子的计数从 1、0、-1 再修正到 0，目标四格始终各保留一次。

把查询拆成几块，检查每个格子被计了几次。 先取 `[0,r2)×[0,c2)` 的大矩形，再减掉上方 `[0,r1)×[0,c2)` 与左方 `[0,r2)×[0,c1)`。目标外的左上角同时属于上方和左方，被减了两次，需要加回一次。

对矩阵 `[[1,2,3],[4,5,6]]`，查询 `[0,2)×[1,3)`：先取全部 `21`，上方高度为 0 所以减 `0`，左方第一列为 `1+4=5`，左上角也是 `0`，结果为 `21-0-5+0=16`。额外的第 0 行、第 0 列都是 0，使靠边与空矩形也能使用同一个式子。

建表时先加上方、加左方，再减掉重复的左上角，最后加入当前格子，与查询的容斥过程相同。这里预处理与查询都用半开区间，r2、c2 表示矩形之后的位置，因此直接代入图中的四个前缀值。

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

vector<vector<long long>> buildPrefix2D(const vector<vector<int>> &a) {
    int rows = static_cast<int>(a.size());
    int columns = rows == 0 ? 0 : static_cast<int>(a[0].size());
    vector<vector<long long>> prefix(rows + 1, vector<long long>(columns + 1, 0));
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < columns; ++j) {
```

```cpp
            // 上方加左方会重复计算左上角，减去一次后再加入当前格。
            prefix[i + 1][j + 1] = a[i][j] + prefix[i][j + 1] + prefix[i + 1][j] -
prefix[i][j];
        }
    }
    return prefix;
}

long long rectangleSum(const vector<vector<long long>> &prefix, int r1, int c1, int r2, int
c2) {
    // 两次减去多余区域后，左上角被多减一次，因此加回来。
    return prefix[r2][c2] - prefix[r1][c2] - prefix[r2][c1] + prefix[r1][c1];
}
```

样例与解释： 矩阵 `[[1,2,3],[4,5,6]]`，调用 `rectangleSum(buildPrefix2D(a), 0, 1, 2, 3)` 返回 `16`，对应 `2+3+5+6`。预处理时间与空间均为 $O(RC)$，一次查询为 $O(1)$。

练习题： 洛谷 P1387 - 最大正方形（可用二维前缀和检查候选正方形，也可进一步学习 DP）、洛谷 P3397 - 地毯（练习二维差分，并用二维前缀恢复）。

3. 差分

差分适合“多次区间加，最后统一查看结果”。定义 `diff[0] = a[0]`，`diff[i] = a[i] - a[i-1]`。给 `[left, right)` 的所有元素加上 `delta`，只需修改两端：

题目原文（自拟）： 给定数组和若干合法区间加操作，把每个区间内的元素都加上指定增量，最后输出修改后的数组。累计结果在 `long long` 范围内。下面先演示差分的端点修改，再给完整程序。

两端修改为什么能影响整个区间？ 还原下标 `i` 的值时，计算的是 `diff[0]+…+diff[i]`。在 `diff[left]` 加 `delta` 后，从 `left` 开始的每个前缀都会多出这份增量；到 `right` 再加上 `-delta`，增量被抵消，所以只有 `[left,right)` 受影响。

例如原差分为 `[2,-1,2,1,0]`，给 `[1,3)` 加 5 后变成 `[2,4,2,-4,0]`。累计得到 `2、6、8、4`：下标 1 开始多出 5，下标 3 又取消它。长度 `n+1` 的数组保存右端取消标记，使 `right=n` 的操作也无需特殊分支；最后只恢复前 `n` 个原数组元素。

若题目要求每次修改后立刻回答区间和，这种“最后一次性还原”的模板不能直接完成任务，需要进一步考虑树状数组或线段树等结构。



闭区间增加由左端开始、右端后一格抵消。



原数组 `[2,1,3,4]` 的差分为 `[2,-1,2,1,0]`，最后一格是边界备用位置。给闭区间 `[1,2]` 加 5，只改 `diff[1]+=5` 与 `diff[3]-=5`，得到 `[2,4,2,-4,0]`。从左向右累计：2、6、8、4；加法从位置 1 开始影响后缀，到位置 3 被抵消，因此区间外的元素恢复原值。

```cpp
#include <vector>
using namespace std;
vector<long long> diff{2, -1, 2, 1, 0}; // [2,1,3,4] 的差分，多留一个尾部位置。
int left = 1, right = 3;
long long delta = 5;
diff[left] += delta;
diff[right] -= delta; // diff 多分配一个位置，允许 right == n
```

最终对 `diff` 做前缀和，还原数组。

```cpp
#include <iostream>
#include <vector>
```

```cpp
using namespace std;

int main() {
    vector<long long> values{2, 1, 3, 4};
    int n = static_cast<int>(values.size());
    vector<long long> diff(n + 1, 0);
    for (int i = 0; i < n; ++i) {
        diff[i] = values[i] - (i == 0 ? 0 : values[i - 1]);
    }
    diff[1] += 5;
    diff[3] -= 5; // 给 [1, 3) 加 5

    long long current = 0;
    for (int i = 0; i < n; ++i) {
        current += diff[i];
        cout << current << ' ';
    }
    cout << '\n'; // 2 6 8 4
}
```

> **注意：** 前缀和常用于快速查询，差分常用于批量修改；它们解决的问题不同。所有累计结果都要检查数值范围。

样例与解释： 原数组 `[2,1,3,4]`，操作 `[1,3)` 加 5，输出 `2 6 8 4`。两端操作启动和取消增量，还原时只影响下标 1、2。处理 n 个元素、q 次操作共 $O(n+q)$ 时间，额外空间 $O(n)$。

练习题： 洛谷 P2367 - 语文成绩、洛谷 P3397 - 地毯。

（四） 排序与离散化

前缀和按位置累积，遇到按数值查找、比较大小或给稀疏数值编号时，先建立顺序会更方便。这里继续从 STL 的排序接口追问它如何减少查找工作。

1. 常见排序思路

方法	思路	时间复杂度	特点
选择排序	每轮找剩余元素的最小值	$O(n^2)$	入门直观，通常不稳定
插入排序	将当前元素插入已有序部分	最坏 $O(n^2)$	有序时可达 $O(n)$，可稳定实现
归并排序	分成两半排序后合并	$O(n \log n)$	可稳定实现，常需 $O(n)$ 辅助空间
快速排序	按枢轴划分后递归	平均 $O(n \log n)$，最坏 $O(n^2)$	划分策略影响性能

竞赛通常直接使用 sort，比较次数为 $O(n \log n)$。如果两条同分记录还要保持输入时的先后顺序，就使用 stable_sort，它会保留等价元素的相对顺序。

稳定是指排序键相同的元素，排序后相对顺序不变。`vector` 使用 `sort` 按位置移动元素；`list` 使用成员 `list::sort()` 调整链表中的节点连接。

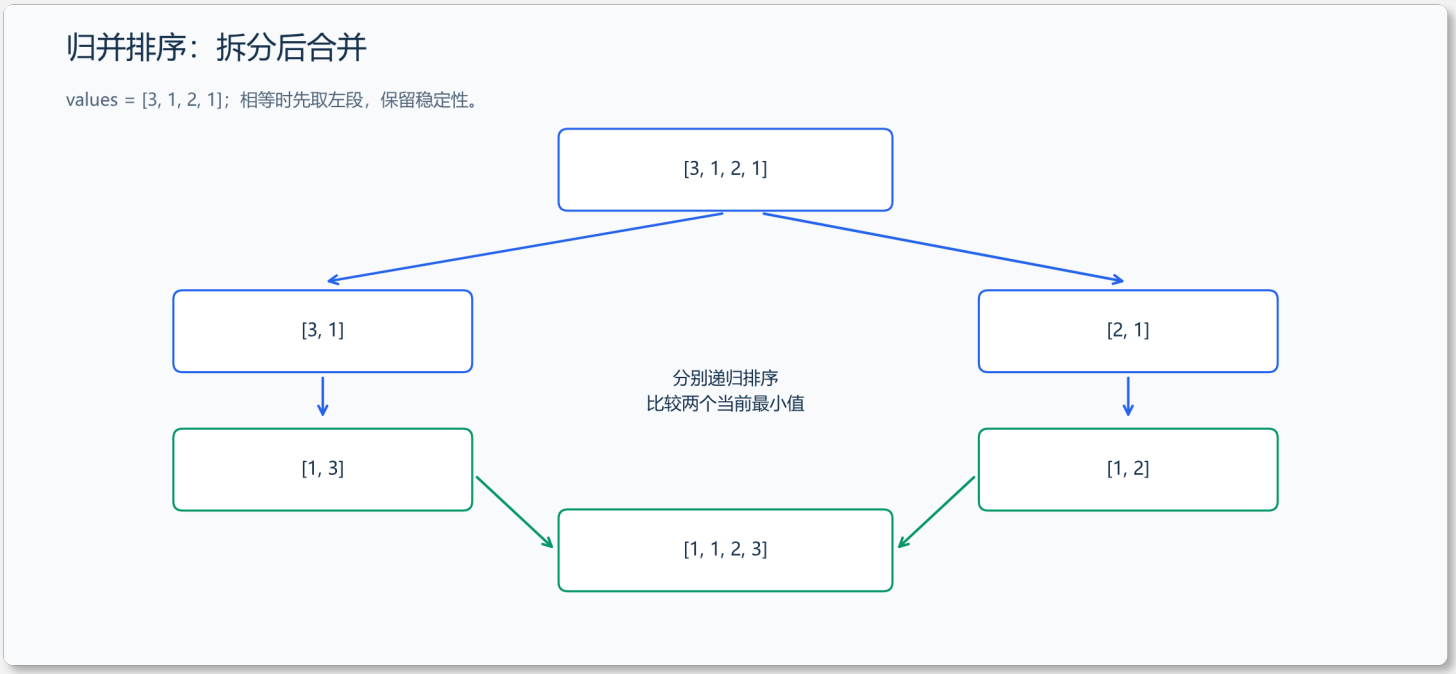
练习题： 洛谷 P1177 - 排序。

2. 归并排序模板

下面使用半开区间 `[left, right)`：

题目原文（自拟）： 给定整数数组，将元素按非递减顺序排列，保留所有重复元素。用归并排序实现。

分析： 把区间拆成两半，递归排序，再按两段的当前最小值合并。区间长度为 `0` 或 `1` 时已经有序。



两段排好序后，只比较各自尚未取出的第一个值；相等时先取左段。



样例 `[3,1,2,1]` 的两半排好后为 `[1左,3]` 与 `[1右,2]`。第一轮相等，按代码先取左侧 1；随后依次比较 3 与 1、3 与 2，取右侧的 1、2；右段耗尽，复制左段剩余的 3。左右标签只用于追踪相等元素来源，最终输出仍是 `[1,1,2,3]`，相等时先取左段使归并保持稳定。

合并为什么只看两段的第一个剩余元素？ 两段内部已经有序，所以各自的第一个剩余元素都是该段剩余部分的最小值。比较这两个值，把较小者写入 `temp`，它就一定是全部剩余元素中的最小值。重复这一步，已写部分始终有序，且不会漏掉元素。

对 `[3,1,2,1]`，左右分别排成 `[1,3]` 和 `[1,2]`。合并时先取左边的 `1`，再取右边的 `1`，然后取 `2`，最后补上 `3`。相等时先取左段，配合各段内部也保持原相对次序，便得到稳定排序；若只排序整数，稳定性的差别不明显，排序带编号记录时就能看出来。

`temp` 只作为当前合并结果，完成后要复制回 `values[left…right)`，因为上层递归合并读取的是 `values`。长度 0 或 1 的区间不用再拆分，这既是递归终点，也保证最终能够结束。

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

void mergeSort(vector<int> &values, vector<int> &temp, int left, int right) {
    if (right - left <= 1) {
        return;
    }
    // 把半开区间拆为两半，分别排序后再合并。
    int mid = left + (right - left) / 2;
    mergeSort(values, temp, left, mid);
    mergeSort(values, temp, mid, right);

    // i、j 读取左右有序段，k 写入辅助数组。
    int i = left, j = mid, k = left;
    while (i < mid && j < right) {
        if (values[i] <= values[j]) {
            temp[k++] = values[i++]; // 相等时先取左边，保持稳定
```

```cpp
        } else {
            temp[k++] = values[j++];
        }
    }
    while (i < mid) {
        temp[k++] = values[i++];
    }
    while (j < right) {
        temp[k++] = values[j++];
    }
    for (int index = left; index < right; ++index) {
        values[index] = temp[index];
    }
}
```

调用时先创建同样长度的辅助数组：`vector<int> temp(values.size());`，再调用 `mergeSort(values, temp, 0, static_cast<int>(values.size()));`。

归并排序体现“分解、解决子问题、合并”的分治思想。若合并时右侧元素小于当前左侧元素，左侧未取出的 `mid - i` 个元素都会与它形成逆序对，可以据此扩展为逆序对计数。

样例与解释： `[3,1,2,1]` 排序后为 `[1,1,2,3]`；重复的两个 1 都保留。时间 $O(n \log n)$，辅助数组空间 $O(n)$，递归栈深 $O(\log n)$。

练习题： 洛谷 P1908 - 逆序对。

3. 离散化

数据值很大，但我们只关心大小关系时，可以把它们映射成紧凑下标：

题目原文（自拟）： 给定整数数组，将不同值按升序依次编号为 `0,1,...`，输出原数组每个元素的编号。相同原值必须得到相同编号。

分析： 先在副本中排序去重，再用 `lower_bound` 找每个值在去重数组中的位置。

保留需要的关系，去掉无用的数值跨度。 `[100,-5,100,20]` 如果按数值当数组下标，会遇到负数与很大的范围；按值排序去重后只剩 `[-5,20,100]`，用它们的位置 `0、1、2` 就能表示大小次序。回到原顺序便得到 `[2,0,2,1]`。

`unique` 只把连续相等元素压到前面并返回新的逻辑末尾，不直接缩短容器，所以先排序让相同值相邻，再用 `erase` 删除尾部。`lower_bound` 找原值的第一处位置；原值都来自这份副本，所以一定能找到对应排名。

排名之间的差不能当坐标距离：原值 `-5` 到 `20` 距离为 `25`，新排名只差 `1`。处理区间长度、面积等问题时，需要保留排序后的原坐标，用相邻原坐标的差计算长度。



相同原值映射到相同排名，大小关系保留；排名差不等于原坐标距离。

```cpp
#include <algorithm>
#include <vector>
using namespace std;

vector<int> compress(const vector<int> &values) {
    // 复制后排序去重，得到每种原值对应的有序排名。
    vector<int> sorted = values;
    sort(sorted.begin(), sorted.end());
    sorted.erase(unique(sorted.begin(), sorted.end()), sorted.end());

    vector<int> rank;
    for (int x : values) {
        rank.push_back(
            static_cast<int>(lower_bound(sorted.begin(), sorted.end(), x) - sorted.begin()));
    }
    return rank;
}
```

例： `[100, -5, 100, 20]` **映射为** `[2, 0, 2, 1]`**。离散化保留大小关系和相等关系，不保留数值差距，不能直接用新下标计算原坐标距离。时间** $O(n \log n)$**，空间** $O(n)$**。**

练习题： **洛谷 P1496 — 火烧赤壁**（排序、区间合并或离散化练习）。

（五） 基础数论

接下来处理整数本身的结构：哪些数能整除、怎样找公约数、哪些数是质数。先用枚举写出判断，再利用整除性质缩小检查范围。

1. 最大公约数与最小公倍数

欧几里得算法利用：

$$\gcd(a, b) = \gcd(b, a \bmod b)$$

下面要求 `a ≥ 0`、`b ≥ 0`：

题目原文（自拟）： 给定两个非负整数，求最大公约数；额外约定两个数都为 0 时返回 0。

分析： 每次用 `(b,a%b)` 替换 `(a,b)`，公约数集合不变，直到余数为 0。



每一步都不改变最大公约数，而第二个参数不断缩小，直到变成 0。

证明公约数集合没有变化。 写成带余除法 $a = qb + r$。若 `d` 同时整除 `a、b`，就整除 $r = a - qb$；若 `d` 同时整除 `b、r`，就整除 $a = qb + r$。因此两对数拥有相同公约数，也就拥有相同最大公约数。

每步余数满足 `0 ≤ r < b`，第二个参数严格变小，最终到达 0；此时 `gcd(a,0)=a`。这同时给出了正确性和终止理由。交换顺序或一开始 `a<b` 都不影响算法，第一步会把较大值转到合适位置。

最小公倍数公式也可以从质因数指数理解：最大公约数取两数每种质因子的较小指数，最小公倍数取较大指数，两者相乘恰好是 `a*b`。代码先用 a 除以 gcd(a,b)，再与 b 相乘，让中间值比先计算 a*b 更小。

```cpp
long long gcdEuclid(long long a, long long b) {
    // gcd(a,b) 与 gcd(b,a%b) 相同，余数逐步减小。
    while (b != 0) {
        long long remainder = a % b;
        a = b;
        b = remainder;
    }
    return a;
}
```

时间复杂度为对数级；本模板约定 `gcd(0, 0) = 0`。C++17 可以直接使用 `<numeric>` 中的 `std::gcd`。

样例与解释： `gcdEuclid(48,18)` 的过程为 `(48,18)→(18,12)→(12,6)→(6,0)`，返回 6；`gcdEuclid(0,5)` 返回 5。额外空间 $O(1)$。

当 `a`、`b` 为正整数时：

$$\operatorname{lcm}(a,b) = \frac{a}{\gcd(a,b)} \times b$$

计算 `a/gcd(a,b)*b` 时，先除去公因子，再相乘，中间数比直接计算 `a*b` 小。结果较大时，和前面的累计和一样，用 `long long` 保存。

练习题： 洛谷 P1029 — 最大公约数和最小公倍数问题。

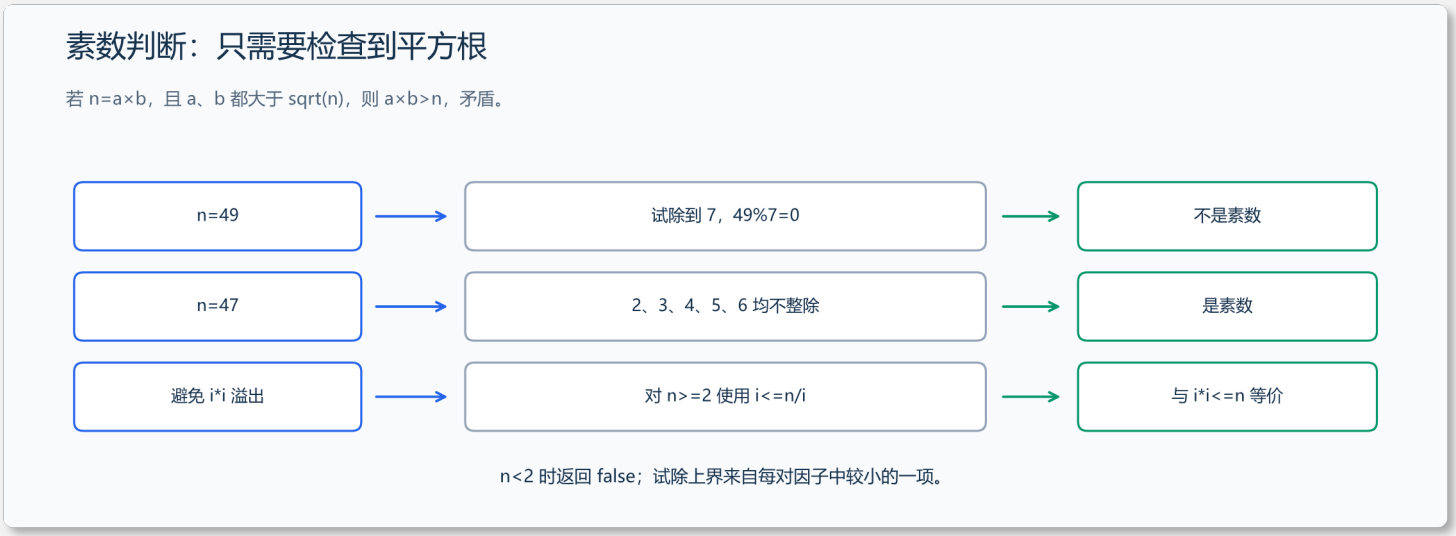
2. 素数判断

如果整数 `n ≥ 2` 有非平凡因数，至少有一个因数不超过 $\sqrt{n}$：

题目原文（自拟）： 给定整数 n，判断它是否为素数，返回 bool。素数是大于 1 且只有 1、自己两个正因数的整数。

为什么试到平方根就够了？ 若合数 `n=a*b`，且两个因子都大于 $\sqrt{n}$，乘积就会大于 `n`，矛盾。因此至少有一因子不超过平方根。在这个范围找不到非平凡因子，就能排除所有合数分解。

例如 `49=7×7`，因子恰好落在平方根上，所以检查到 `i=7` 时仍要试除，循环使用 `≤`；只写 `<` 就会在检查 7 之前停止。`1` 的正因数只有一个，不符合素数定义；负数和 0 也在进入循环前排除。遇到因子可以立即返回失败，而素数会执行到范围结束，所以用它估计最坏次数。



每对因子至少有一个不超过平方根。

```cpp
bool isPrime(long long n) {
    if (n < 2) {
        return false;
    }
    // 有合数因子时，至少一个不超过平方根；用除法避免平方溢出。
    for (long long factor = 2; factor ≤ n / factor; ++factor) {
        if (n % factor == 0) {
            return false;
        }
    }
    return true;
}
```

使用 `factor ≤ n / factor` 避免 `factor * factor` 溢出。时间 $O(\sqrt{n})$；要判断很多小范围整数时，筛法更合适。

样例与解释： `isPrime(1)` 为 false；`isPrime(17)` 为 true；`isPrime(49)` 在检查因子 7 时得到 false。额外空间 $O(1)$。

练习题： 洛谷 P1036 – 选数（结合枚举或 DFS）。

3. 埃氏筛

如果 `p` 是素数，就把它的倍数标为合数。小于 `p * p` 的倍数已经会被更小的素因子处理。

题目原文（自拟）： 给定非负上限 n，建立下标 0 到 n 的素数标记表；位置 x 为 true 当且仅当 x 是素数。上限应符合题目的内存与时间限制。



上限 30 时，p=2 从 4 开始划去倍数，p=3 从 9 开始继续划去倍数；轮到 p=4 时发现它已经不是素数，整轮跳过；p=5 从 25 开始，只新排除 25。到 p=6 已有 $p^2>30$，停止外层循环。图同时列出“这一轮新增排除”与“之前已排除”，不能把后者重复算成新的合数。

为什么从 p^2 开始，而不是从 $2p$？ 小于 p^2 的倍数可以写成 $p*k$，其中 $2 \le k < p$。k 必有一个比 p 小的素因子，这个素因子先前已经把 $p*k$ 标成合数。因此，从 p^2 开始不会遗漏合数，却能减少重复标记。

以筛到 30 为例，先用 2 标记 4、6、8、…、30；再用 3 从 9 开始标记，6 已被 2 处理；4 已是合数，不再用它启动新一轮；到 5 从 25 开始。大于平方根的合数仍然有一个不超过平方根的素因子，因此外层也只需到平方根。

筛法一次计算 0 到 n 的素数标记，之后访问 prime[x] 就能查到这一范围里 x 的答案。它适合反复查询同一批数；只判断一个数时，前面的试除法会省去整张标记表。

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

vector<bool> primeSieve(int n) { // n ≥ 0
    vector<bool> prime(static_cast<size_t>(n) + 1, true);
    prime[0] = false;
    if (n ≥ 1) {
        prime[1] = false;
    }
    for (int p = 2; p ≤ n / p; ++p) {
        if (prime[p]) {
            // 更小的倍数已有更小素因子，从 p² 开始标记。
            for (long long multiple = 1LL * p * p; multiple ≤ n; multiple += p) {
                prime[static_cast<size_t>(multiple)] = false;
            }
        }
    }
    return prime;
}
```

得到 `prime[x]` 后可以 $O(1)$ 查询 `x` 是否为素数。筛的时间为 $O(n \log \log n)$，空间为 $O(n)$。

样例与解释： n=10，表中为 true 的位置是 2、3、5、7；0、1 从初始化时就排除。

练习题： 洛谷 P3383 — 线性筛素数（规模较大，可继续学习线性筛）。

4. 试除法分解质因数

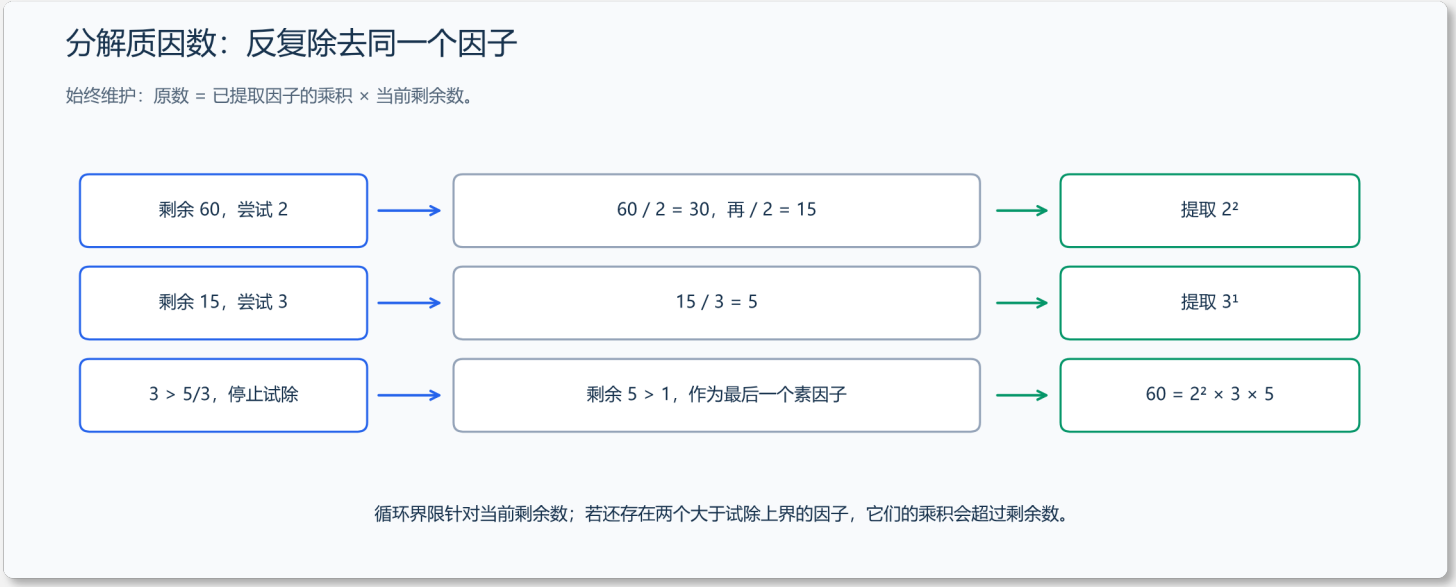
题目原文（自拟）： 给定正整数 n，按素因子从小到大返回 `(素因子,指数)` 列表；n=1 时返回空表。

分析： 从 2 开始尝试除数，遇到因子就连续除尽并统计指数。试除结束时若仍剩余大于 1 的数，它就是最后的素因子。

为什么试除的合数不会被记为素因子？ 到除数 `p` 时，比它小的因子都已经被除尽。如果 `p` 是合数，它含有更小的素因子；剩余数已不含这些因子，就不可能再被 `p` 整除。因此真正被加入结果的除数一定是素数，虽然循环没有单独调用判素函数。

手算 `60`：除以 `2` 两次，得到剩余数 `15`，记录 `(2,2)`；再除以 `3` 一次，剩余数 `5`，记录 `(3,1)`。此时后续除数的平方已大于剩余数，退出循环，把剩余 `5` 记为 `(5,1)`。

最后的剩余数若大于 1 就一定是素数：若它还能分解成两个非平凡因子，至少一个不超过它的平方根，应当已经被前面的试除过程找出，形成矛盾。这里循环上界随剩余 `n` 缩小，但最坏复杂度仍按最初输入值的平方根估计。



60 先除去两次 2，再除去一次 3，最后剩余的 5 是素数。循环检查的是不断缩小的剩余数。

```cpp
#include <utility>
#include <vector>
using namespace std;

vector<pair<long long, int>> factorize(long long n) { // n ≥ 1
    vector<pair<long long, int>> result;
    for (long long p = 2; p ≤ n / p; ++p) {
        if (n % p == 0) {
            // 连续除去同一个素因子，记录它在分解中的指数。
            int exponent = 0;
            do {
                n /= p;
                ++exponent;
            } while (n % p == 0);
            result.push_back({p, exponent});
        }
    }
    if (n > 1) {
        result.push_back({n, 1});
    }
    return result;
}
```

例：`60 = 2² × 3 × 5`，结果为 `[(2, 2), (3, 1), (5, 1)]`。`1` 没有质因数，返回空表；最坏时间 $O(\sqrt{n})$。

练习题： 洛谷 P1075 – 质因数分解。

（六） 快速幂

计算 a 的 b 次方，直接连续乘 b 次已经能得到答案；当 b 很大时，把指数拆成二进制，观察哪些重复乘法可以合并成平方。

1. 把指数拆成二进制位

直接计算 a^b，逐次乘 `a` 需要 $O(b)$ 次乘法。快速幂利用指数的二进制表示，把乘法次数降到 $O(\log b)$。

例如：

$$13 = (1101)_2 = 8 + 4 + 1, \qquad a^{13} = a^8 \cdot a^4 \cdot a$$



每轮检查最低位：若为 `1`，把当前底数乘进答案；随后底数平方，指数右移一位。

2. 普通整数快速幂

题目原文（自拟）： 给定整数底数和非负指数，计算整数幂，用 `long long` 保存结果。指数为 `0` 时返回 `1`，包括底数为 `0` 的情况。

把指数的一位与当前底数配起来。 初始底数是 `a`；每次平方后依次变成 a^2、a^4、a^8、…。指数每次右移，依次检查原来的第 0、1、2、3 位；该位为 1 才把对应的幂乘进结果。

以 2^{13} 为例，表中记录每轮入口状态和处理结果：

剩余指数	当前底数	最低位	处理后的 result
13	2	1	1×2=2
6	4	0	2
3	16	1	2×16=32
1	256	1	32×256=8192

最低位为 0 时只是不累乘，仍需推进到下一位。每轮开始可维护 $result \times base^{exponent} = a^b$：取走最低位对应的因子后，剩余指数变成偶数，再把底数平方、指数减半，这个乘积不变。指数为 0 时只剩 `result`，就是答案。

普通整数版处理完最后一位后，指数已经变为 0，直接返回累乘结果。以 2^{13} 为例，这时答案是 8192，再继续平方不会参与答案计算，所以代码跳过这次平方。



指数按二进制拆分，仅在当前位为 1 时乘入结果。

```cpp
long long fastPower(long long base, unsigned long long exponent) {
    long long result = 1;
    while (exponent != 0) {
        // 最低位为 1 时，把当前底数对应的这一项乘入答案。
        if ((exponent & 1ULL) != 0) {
            result *= base;
        }
        exponent >>= 1;
        if (exponent != 0) {
            base *= base; // 还有后续位时才需要继续平方
        }
    }
    return result;
}
```

`fastPower(2,13)` 得到 8192。代码用 unsigned long long 保存指数，用 long long 保存底数与结果；每次乘法仍按“常量、类型转换与整数溢出”一节介绍的整数运算类型计算。指数为 0 时循环执行零次，返回初值 1，因此这里的 `fastPower(0,0)` 也返回 1。

3. 取模快速幂

题目原文（自拟）： 给定整数 a、非负整数 b 和模数 m，其中 `1≤m≤2^31-1`，计算 $a^b \bmod m$，输出 `[0, m)` 中的余数。

当答案很大，而题目只要求 $a^b \bmod m$ 时，可以在每次乘法后取模。

取模后两因子最大为 `m-1`。题目中的最大模数是 `2^31-1`，所以乘积最大为 `(2^31-2)²`，小于 `2^63-1`，可以用 long long 先算乘积、再取模。这与“常量、类型转换与整数溢出”一节先推导数值大小、再选择类型的过程相同。

取模为什么不会破坏最后的余数？ 若 `x` 与 `r` 模 `m` 同余，`y` 与 `s` 模 `m` 同余，则 `xy` 与 `rs` 也模 `m` 同余，所以每次相乘后立刻取模，最后得到的余数不变。这是在中间过程压小数值，不是在最后才对一个已溢出的结果补救。

对 `2¹³ mod 1000`，累乘结果依次是 `2、2、32、192`；最后一步计算 `32×256=8192`，取模得到 `192`。所有底数和结果保持在 `[0,m)`，负底数先规范到这个区间。`result=1%m` 还覆盖了模数为 1 的情况：它的所有余数都为 0。



图按 `2¹³ mod 1000` 逐轮列出当前指数、最低位、底数和累乘余数：指数 6 的最低位为 0，结果保持 2；指数 3、1 的最低位为 1，结果先变为 32，最后由 8192 取模得到 192。普通版的最后一轮不再平方，取模版则始终保持底数、结果在合法余数范围。

```cpp
#include <limits>
#include <stdexcept>

long long modPower(long long base, unsigned long long exponent, long long mod) {
    if (mod ≤ 0 || mod > 2147483647LL) {
        throw std::invalid_argument("mod 必须在 [1, 2^31-1] 内");
```

```cpp
    }
    base %= mod;
    if (base < 0) {
        base += mod;
    }
    // mod 为 1 时结果应为 0，包括指数为 0 的情况。
    long long result = 1 % mod;
    while (exponent != 0) {
        // 最低位为 1 时，把当前底数对应的这一项乘入答案。
        if ((exponent & 1ULL) != 0) {
            result = result * base % mod;
        }
        // 底数平方对应指数下一位；乘法安全依赖前面声明的模数范围。
        base = base * base % mod;
        exponent >>= 1;
    }
    return result;
}
```

cpp

调用	结果	检查点
`modPower(2, 13, 1000)`	`192`	普通情况
`modPower(5, 0, 7)`	`1`	指数为 0
`modPower(5, 0, 1)`	`0`	模数为 1
`modPower(-2, 3, 5)`	`2`	负底数规范化

例如模数为 `1000000007` 时，两个余数的乘积最多为 `1000000006²`，long long 可以保存这个中间结果。计算更大的模乘时，还可以学习 GNU C++ 的 `__int128` 宽整数，或把乘法拆成逐位加法与取模；取模操作始终发生在乘法计算之后。

4. 正确性与应用

循环中保持“已累计的答案乘上剩余底数的剩余次幂，等于原来的幂”这一关系。指数最低位为 `1` 时取出一个底数因子，再把剩余偶数指数除以 `2`，同时将底数平方，所以结果不变。

时间 $O(\log b)$（`b = 0` 时直接返回），额外空间 $O(1)$。常见用途是大指数取模、矩阵幂和模逆元。

如果 `p` 是素数，且 `a` 不是 `p` 的倍数，那么费马小定理给出：

$$a^{-1} \equiv a^{p-2} \pmod{p}$$

例如 a=3、p=7 时，`modPower(3,5,7)=5`，而 `3×5 mod 7=1`，所以 5 是 3 的模逆元。逆元让模意义下的除法变成乘法：乘 a 的逆元，就抵消了乘 a 的效果。一般模数 m 下，a 与 m 互质时存在逆元；扩展欧几里得算法可以求它。这里素数模数的 `p-2` 指数来自费马小定理。

练习题： 洛谷 P1226 - 快速幂、CF 742A - Arpa's hard exam and Mehrdad's naive cheat。

（七） 哈希（散列）表

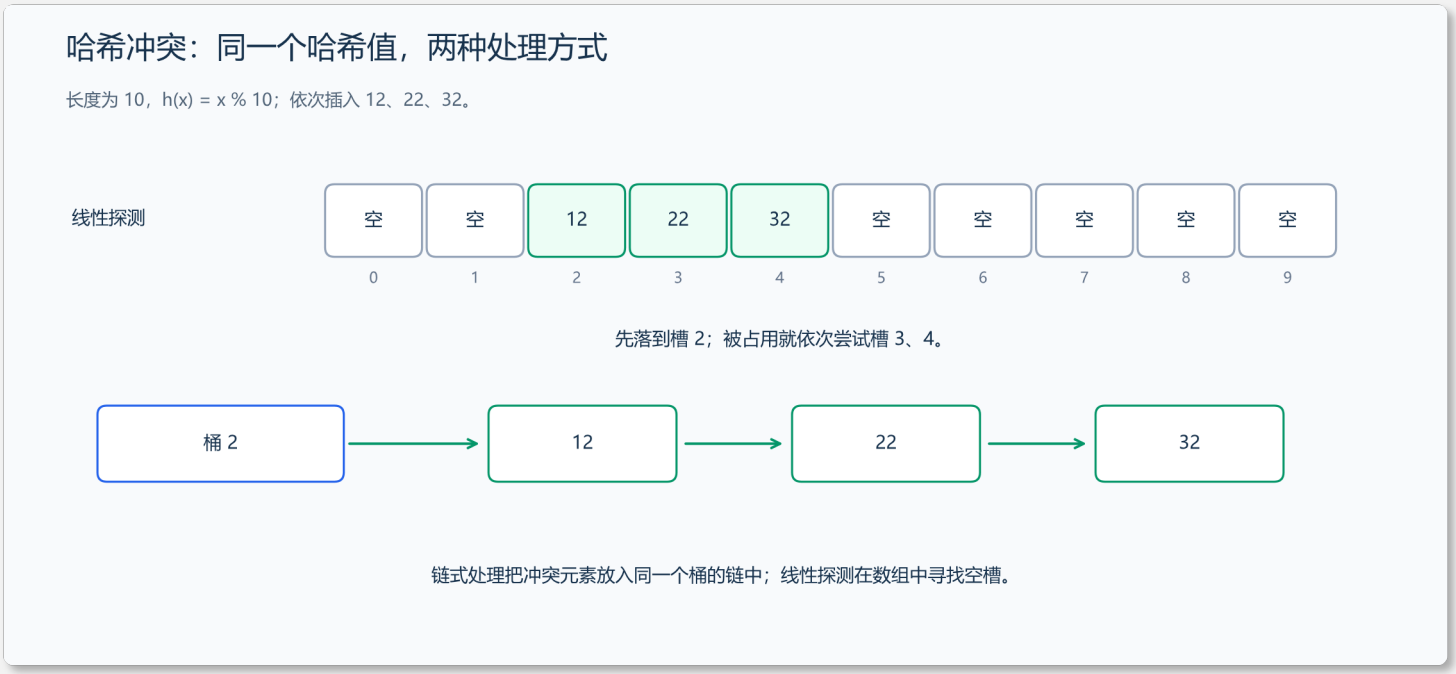
STL 的 unordered 容器已经能判重和统计。现在用小表手工插入、查找几个键，解释为何先定位桶仍要比较真实键，以及冲突怎样影响工作量。

哈希表（Hash Table）先对键计算哈希值，再把元素分配到桶或槽位中。键分散到各桶时，一次查询通常只比较少量候选，平均时间为 $O(1)$；n 个键集中在同一桶时，一次查询可能逐个比较到最后，最坏时间为 $O(n)$。unordered_map 与 unordered_set 是标准库提供的无序关联容器。

1. 哈希值、桶下标与冲突

哈希值是对键计算出的结果，桶下标是进一步映射到存储位置的编号，两者不能总是当作同一个数。教学示例直接使用 `index=((value%m)+m)%m`，把整数分配到 `0..m-1`；实际代码只在余数为负时加一次 `m`，避免先加产生溢出。

`m=10` 时，7 和 17 都映射到桶 7，这叫**冲突**。键仍然不同，不能因为桶相同就把它们当成同一个元素。查找 17 时，定位到桶 7 后还要比较实际保存的是 7 还是 17。下面介绍两种不同的处理方法。



图以 12、22、32 为例：它们的初始桶下标都为 2。左图寻找后续空槽，右图把元素保存在同一桶内；两幅图代表不同方案。

2. 线性探测：沿后续槽位找空位

题目原文（自拟）： 建立长度为 10 的整数表。输入非负整数 `n`，再输入 `n` 个在 `int` 范围内的整数，按输入顺序插入。起始槽位取非负余数；槽位已占用时循环检查下一槽，直到遇到空位。若插入某个数时整表已满，输出 `FULL` 并结束；否则按槽位 `0..9` 输出表，空槽输出 `EMPTY`，以单个空格分隔。重复数也各占一个槽。

本例的空槽表示： `<optional>` 提供 C++17 的 optional。`optional<int>` 可以处于“没有值”或“保存一个 int”两种状态；默认构造没有值，赋入整数后有值。has_value 或逻辑判断检查是否有值，has_value 为真时，`*slot` 读取所保存的整数；为假时，程序输出 EMPTY。它没有拿 0 或 -1 充当空标记，所以这些整数也可正常存入。

思路分析： 检查下标按 `(index+1)%length` 循环，确保到末尾后能回到开头。只检查一整圈：若检查了 10 个占用槽还没找到空位，就不能插入。空状态用 `optional` 表示，因而 `0`、`-1` 等数仍可合法保存。



插入遇到冲突后逐槽检查，直到找到空槽。

```cpp
// optional 的空状态表示空槽；探测最多一整圈，避免表满时死循环。
#include <iostream>
#include <optional>
#include <vector>
using namespace std;

int main() {
    constexpr int length = 10;
    vector<optional<int>> table(length);
    int n;
    cin >> n;
    for (int i = 0; i < n; ++i) {
        int value;
        cin >> value;
        int index = value % length;
        if (index < 0)
```

```cpp
            index += length;
        int checked = 0;
        while (checked < length && table[index].has_value()) {
            index = (index + 1) % length;
            ++checked;
        }
        if (checked == length) {
            cout << "FULL\n";
            return 0;
        }
        table[index] = value;
    }
    for (int i = 0; i < length; ++i) {
        if (i != 0)
            cout << ' ';
        if (table[i])
            cout << *table[i];
        else
            cout << "EMPTY";
    }
    cout << '\n';
}
```

样例输入：

```text
7
0 1 2 7 17 8 6
```

样例输出：

```text
0 1 2 EMPTY EMPTY EMPTY 6 7 17 8
```

代码解释： 17 从槽 7 探测到槽 8；随后 8 从槽 8 探测到槽 9。6 的初始槽为空，所以放入槽 6。checked 记录已经检查几项，绕过十槽还未找到空位时，程序输出 FULL。查找 17 时也从槽 7 开始，跳过 7 后在槽 8 找到它。删除槽 7 中的数时，可以留下“已删除”标记，让查找继续到槽 8；若直接变成空槽，查找就在 7 停下，遗漏后面的 17。

推广到 m 个槽时，一次探测最坏检查 m 项，额外存储为 $O(m)$。表中空槽较多、起始位置较分散时，探测往往很快遇到空位；连续占满一段槽位后，查找和插入都会沿这段位置继续扫描。固定十槽的程序可以直接观察这两种过程。

3. 桶内存储：在同一桶中逐个比较

链地址法让一个桶保存所有落入该桶的元素，常用链表连接它们。下面用 `vector<vector<int>>` 表示桶与桶内元素：外层下标是桶号，内层 vector 是这个桶里的列表，便于观察冲突后怎样保存和查找。

题目原文（自拟）： 输入非负整数 `n`、`n` 个整数，再输入查询值 `target`，所有值在 `int` 范围内。把数据按非负的模 10 余数放入相应桶。先输出 `target` 是否存在，存在输出 1，否则输出 0；再依次输出十个桶，每行格式为 `编号:` 后跟该桶按插入顺序保存的数。重复输入不去重。

思路分析： 先找到 `target` 应在的桶，再检查桶内是否有相等的数。例如桶 7 中有 7、17，查找 17 时先比较 7，再比较 17；其他桶不用访问。



桶下标相同的键仍需逐个比较。

```cpp
#include <iostream>
#include <vector>

using namespace std;
const int len = 10;

int main() {
    // 每个桶用 vector 保存冲突元素，模拟桶式存储；它并不是链表
    vector<vector<int>> hash(len);

    int n;
    cin >> n;
    for (int i = 0; i < n; ++i) {
        int val;
        cin >> val;

        int key = val % len;
        if (key < 0)
            key += len;
        hash[key].push_back(val);
    }

    // 查找操作示例
    int target;
    cin >> target;
    bool flag = 0;

    int index = target % len;
    if (index < 0)
```

```cpp
        index += len;
    const auto &h = hash[index];
    for (int i = 0; i < h.size(); ++i)
        if (h[i] == target)
            flag = 1;

    cout << flag << endl;

    // 输出结构
    for (int i = 0; i < len; ++i) {
        cout << i << ": ";
        for (auto v : hash[i])
            cout << v << " ";
        cout << endl;
    }
}
```

样例输入：

```text
7
0 1 2 7 17 8 6
17
```

样例输出：

```text
1
0: 0
1: 1
2: 2
3:
4:
5:
6: 6
7: 7 17
8: 8
9:
```

代码解释： 查询 17 时只检查桶 7，其中先遇到 7，再遇到 17，最后输出 1。展示用的桶输出包含行末空格，不影响图示内容；解题输出仍要遵守题面格式。若有 `m` 个桶、`n` 个元素且分布均匀，平均桶长约为 `n/m`，查找期望代价为 $O(1+n/m)$；最坏所有元素集中在一桶，查找为 $O(n)$。这个程序还要输出全部桶，所以整段演示的时间为 $O(n+m)$，空间为 $O(n+m)$，不能把“单次平均查找”直接当成整个程序的复杂度。

4. 装载因子、rehash 与哈希设计

装载因子为 $\alpha = n/m$，这里 `n` 是元素数，`m` 是槽位数或桶数。开放地址法越接近满表，探测越可能变长；链式桶即使 `alpha>1` 也能存储，但平均桶长会增加。`0.7` 可以作为某些线性探测实现的经验阈值，不能称为所有哈希表的统一要求。

`rehash` 改变桶数，并按新的桶数重新安排元素；它改变的是组织方式，冲突仍由桶内存储或探测过程处理。标准无序容器还可用 `reserve` 规划预计元素数、用 `max_load_factor` 设置负载上限，从而减少增长时的重建。

方案	存储方式	需要关注
开放地址	元素存于同一槽位数组，冲突时继续探测	满表、聚集、删除标记
桶内存储	同一桶内保存多个元素	桶长、相等比较、附加存储
rehash 操作	改变桶组织并重新分配元素	重建成本、遍历迭代器失效

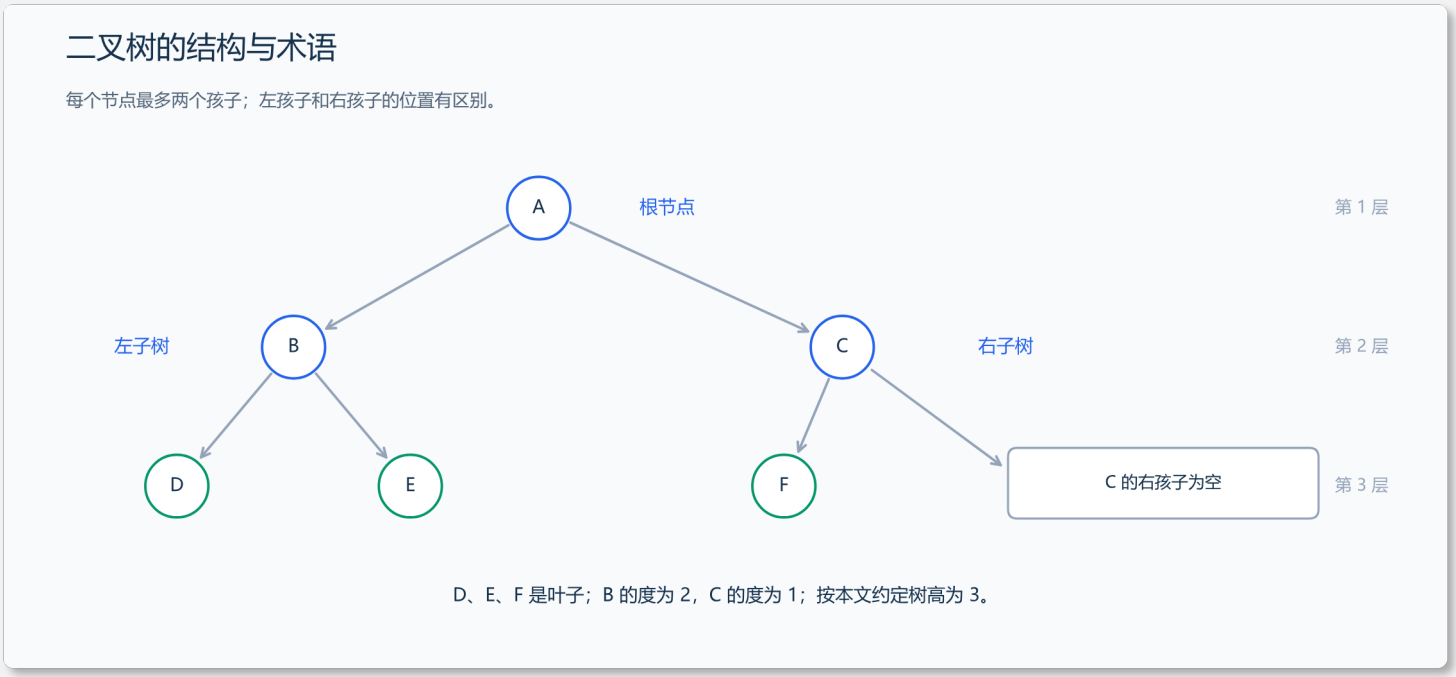
标准无序容器以桶组织元素，冲突时仍在候选桶中比较键；各实现可以选择不同的桶内存储方式。`rehash` 会使迭代器失效，但不会仅因重分配桶而使元素指针与引用失效。相等的键经同一个哈希函数计算得到相同位置，查找因此能回到插入时的桶；不同键也可能落在同一桶，如 7 与 17，这时靠键比较区分。参见 C++ 标准草案：无序关联容器要求。

哈希函数应兼顾计算成本与分布。整数可以用适当的混合函数；字符串可逐字节进行多项式乘加；自定义结构需要结合参与相等比较的字段。字符串哈希中的 31、131 等基数是常用选择之一，不同字符串仍可能得到相同哈希值，因此找到哈希相同的候选后，还要核对实际字符串。精确判重时可直接用保存原字符串的 `set` 或 `unordered_set`；后者仍会比较完整键。

练习题： 洛谷 P3370 — 字符串哈希（可先用 set / unordered_set 精确判重）、CF 4C — Registration System。

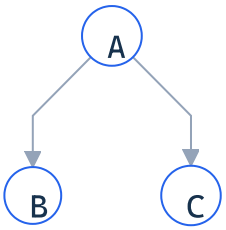
（八） 二叉树

除了顺序数组与按键查找，有些数据需要表达“谁是谁的孩子”。从父节点与左右孩子的记录开始，再观察不同遍历为什么产生不同访问次序。



每个节点最多两个孩子，空孩子不计作节点；这里沿用本文根为第 1 层的约定。

1. 二叉树的结构与性质



如上图，每个节点最多有两个有序子节点的树结构称为**二叉树**。

左边和右边的子节点又被分别被称为：

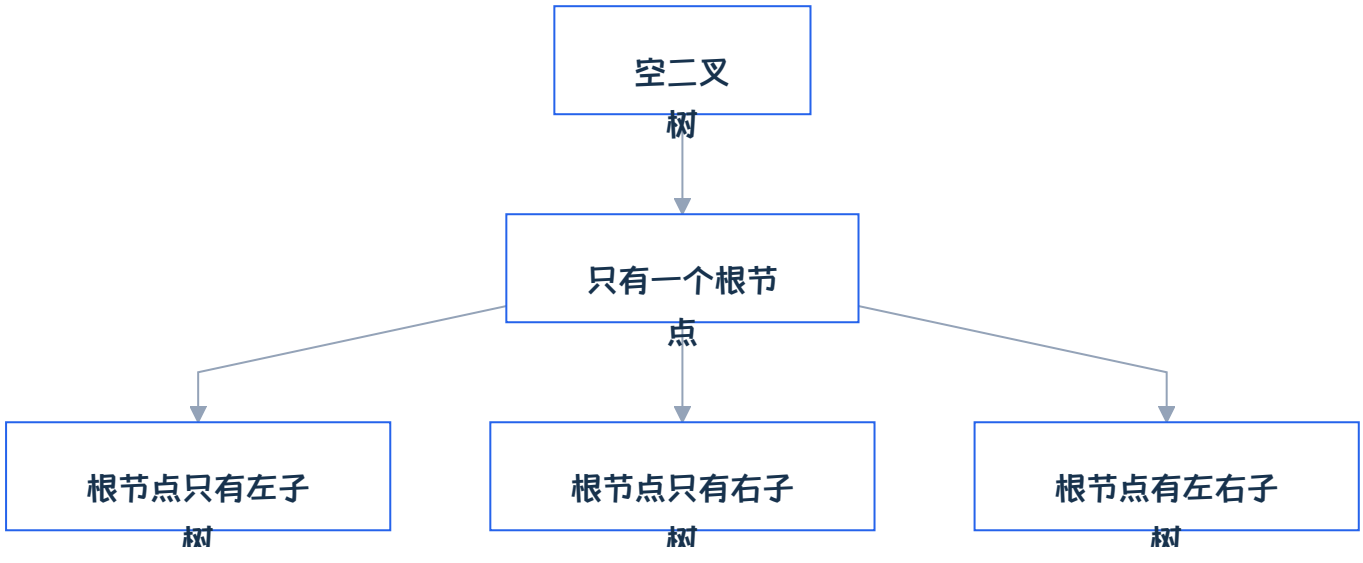
- 左子节点（left child）
- 右子节点（right child）

严格定义：**二叉树** 是 n(n≥0)个节点的有限集合，该集合或者为空集（称为空二叉树），或者由一个根节点和两棵互不相交的、分别称为根节点的左子树和右子树的二叉树组成。

1.1 左右子树与五种结构形态

- 每个节点最多有两棵子树（即度不超过 2）
- 左子树和右子树有顺序，不能颠倒
- 即使只有一棵子树，也要区分是左子树还是右子树

二叉树的五种形态：



1.2 节点的度、深度与高度

（1）节点的度

- **节点的度**：节点拥有的子树数目
- 在二叉树中，节点的度只能是 0、1 或 2

（2）叶子节点

- **叶子节点**：度为 0 的节点（没有子节点的节点）

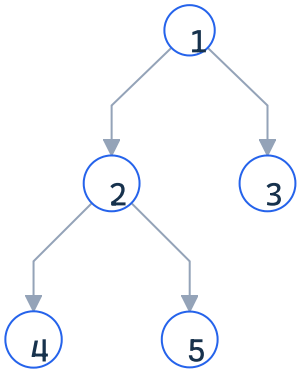
（3） 节点的层次和深度

- **根节点的层次**：第 1 层
- **节点的深度**：本文统一按节点所在层数计算，根的深度为 1；若按边数计算根深度为 0，公式应相应调整。

（4）树的高度

- **树的高度**：树中节点的最大层次

示例：



对于上述二叉树，根节点有两个子树，五个节点的度数分别为 2，2，0，0，0，其中叶子节点，即度为 0 的节点为 3，4，5

根节点 1 的深度为 1，

子节点 2，3 的深度为 2

子节点 4，5 的深度为 3

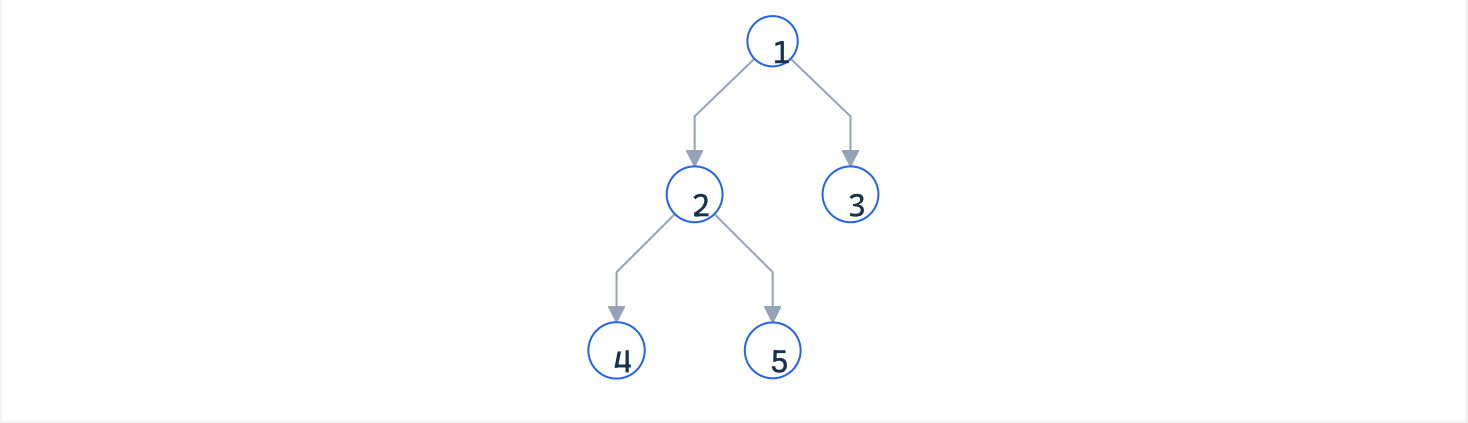
树的最大高度为 3

1.3 常见二叉树类型与数量关系

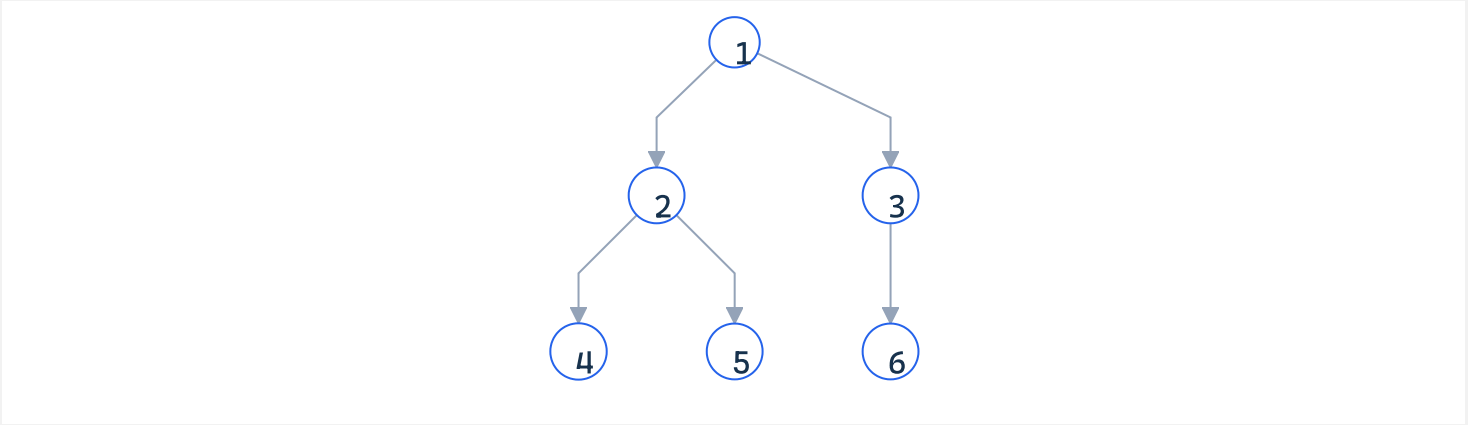
1.3.1 满、完全、搜索与平衡二叉树

满二叉树（本文采用国内教材常见定义，对应 perfect binary tree）：每层都达到最多节点数，高度为 h 时共有 $2^h - 1$ 个节点。

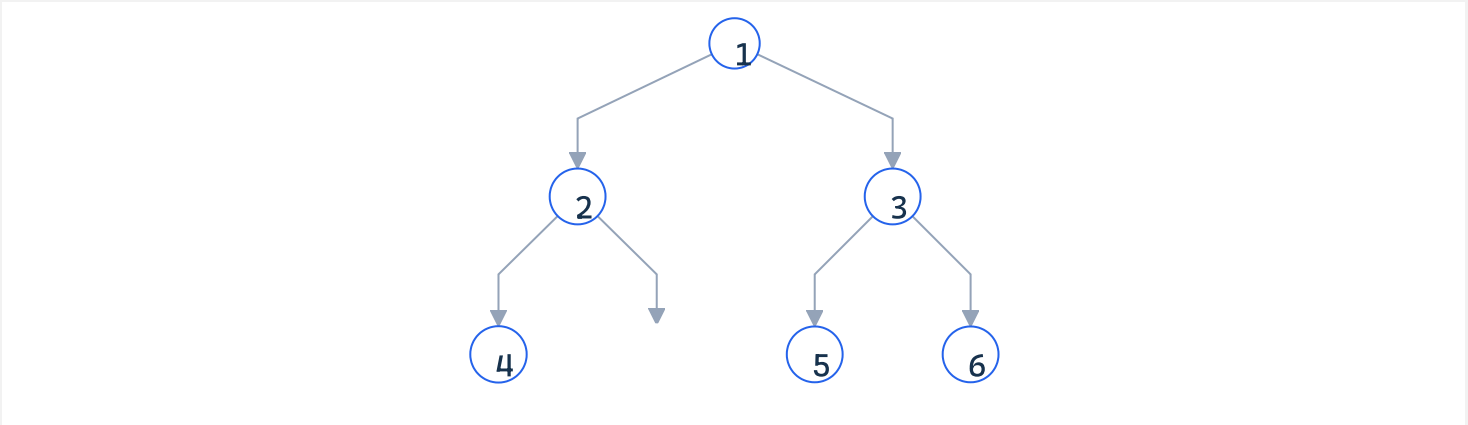
英文 full / proper binary tree 通常只要求每个节点有 0 或 2 个孩子，不要求所有叶子处于同一层。下面五个节点的图符合 full 的英文定义，但不是本文定义的满二叉树。



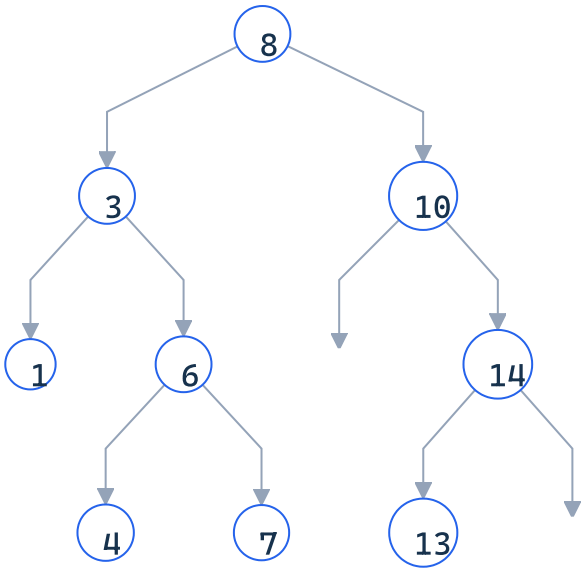
完全二叉树（Complete Binary Tree）：除最后一层外，其他层节点都满，且最后一层节点从左到右连续排列。



由于是从左到右连续排列，因此下列这种**不是**完全二叉树：



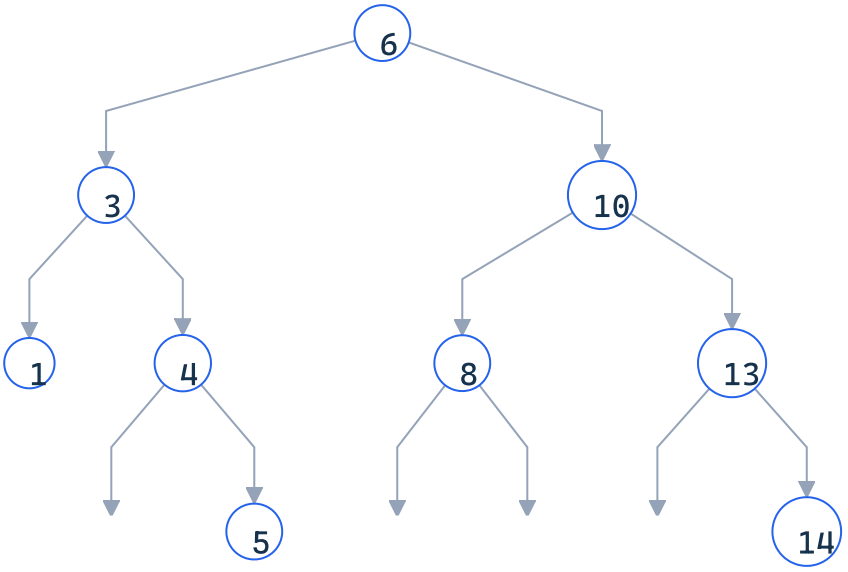
二叉搜索树（BST, Binary Search Tree）：左子树的所有节点值 < 当前节点值 < 右子树所有节点值。



说明：

- 以节点 8 为根：左子树(3,1,6,4,7)都 <8，右子树(10,14,13)都> 8
- 以节点 3 为根：左子树(1)<3，右子树(6,4,7)> 3
- 以节点 6 为根：左子树(4)<6，右子树(7)> 6

AVL 树：二叉搜索树，并且每个节点的左右子树高度差不超过 1。红黑树使用颜色等约束维持对数级高度，不要求每个节点的左右高度差都不超过 1。



自平衡二叉搜索树通过相应的旋转或调整规则控制树高，保持对数级的查找、插入和删除代价。AVL 与红黑树使用的平衡约束不同。

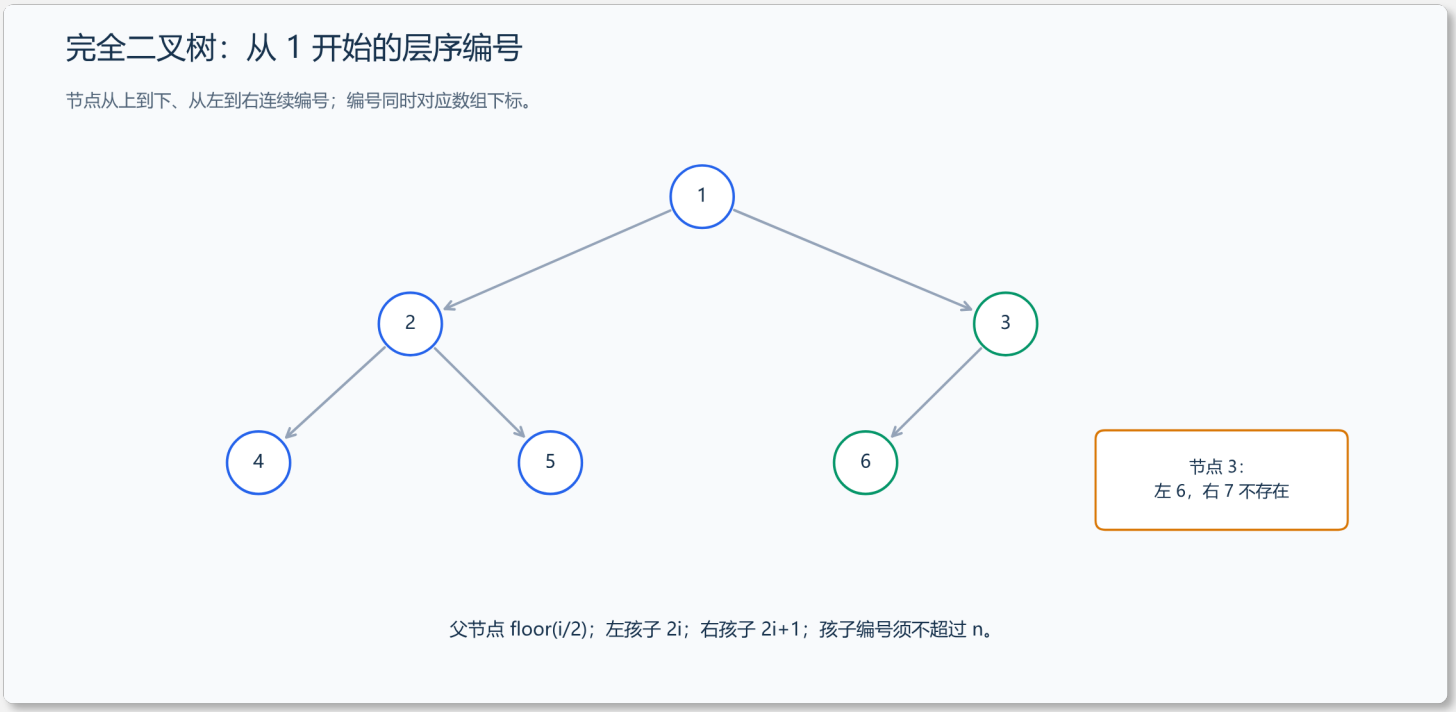
平衡因子：某节点的左子树高度 − 右子树高度

AVL 平衡条件：所有节点的平衡因子绝对值 ≤ 1

常见自平衡二叉搜索树：AVL 树和红黑树使用不同约束与调整规则。

图中的 AVL 示例保持搜索树的大小关系，并且每个节点的左右子树高度差不超过 1。

1.3.2 节点数、高度与叶子数的关系



此图采用从 1 开始的层序编号，父节点为 i/2，孩子为 2i、2i+1；编号超出节点数表示孩子不存在。

性质 1：在二叉树的第 i 层上至多有 $2^{(i-1)}$ 个节点($i \geq 1$)

性质 2：深度为 k 的二叉树至多有 $2^k - 1$ 个节点($k \geq 1$)

性质 3：对任何一棵非空二叉树 T，如果其叶子节点数为 n_0，度为 2 的节点数为 n_2，则：
$n_0 = n_2 + 1$

性质 4：具有 n 个节点的完全二叉树的深度为 $\lfloor \log_2 n \rfloor + 1$ (n ≥ 1)

性质 5：

对完全二叉树按层编号，则对任意节点 i 有：

- 如果 $i = 1$，则是根节点
- 如果 $i > 1$，则其父节点编号为 $\lfloor i/2 \rfloor$
- 左孩子编号为 $2i$ (如果 $2i \leq n$)
- 右孩子编号为 $2i + 1$ (如果 $2i + 1 \leq n$)

性质 6：对于高度为 h 的二叉树：

- **最少节点数**：h（每层只有一个节点）
- **最多节点数**：$2^h - 1$（满二叉树）

性质 7：对于有 n 个节点的完全二叉树：

- **高度**：$\lfloor \log_2 n \rfloor + 1$（n ≥ 1）
- **叶子节点数**：$\lceil \frac{n}{2} \rceil$
- **非叶子节点数**：$\lfloor \frac{n}{2} \rfloor$

1.3.3 用二叉树性质计算节点数与高度

题目：一棵完全二叉树有 1000 个节点，求：

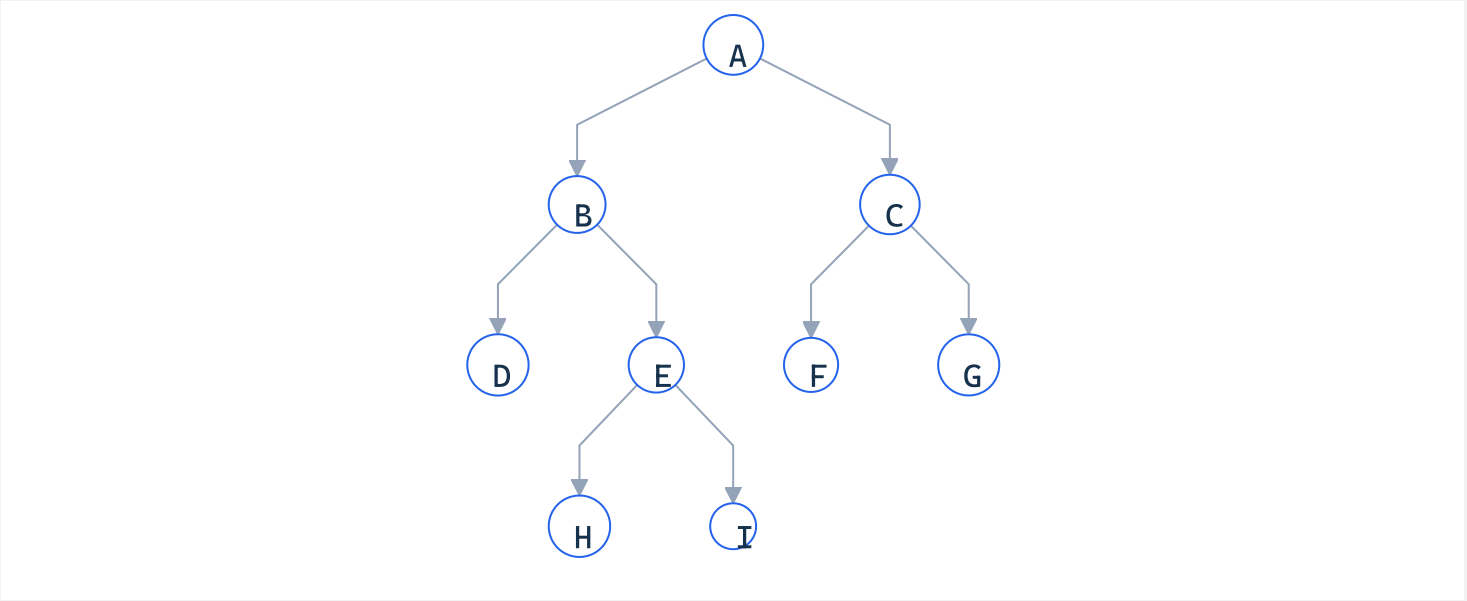
1. 树的高度
2. 叶子节点数
3. 度为 1 的节点数

解答：

1. **高度** = $\lfloor log_2 1000 \rfloor + 1 = \lfloor 9.966 \rfloor + 1 = 9 + 1 = 10$
2. **叶子节点数** = $ [\frac{1000}{2}] = 500$
3. **度为 1 的节点数**：
   - $n_0 = 500$
   - $n = n_0 + n_1 + n_2 = 1000$
   - 又因为 $n_0 = n_2 + 1$，所以 $n_2 = 499$
   - 因此 $n_1 = 1000 - 500 - 499 = 1$

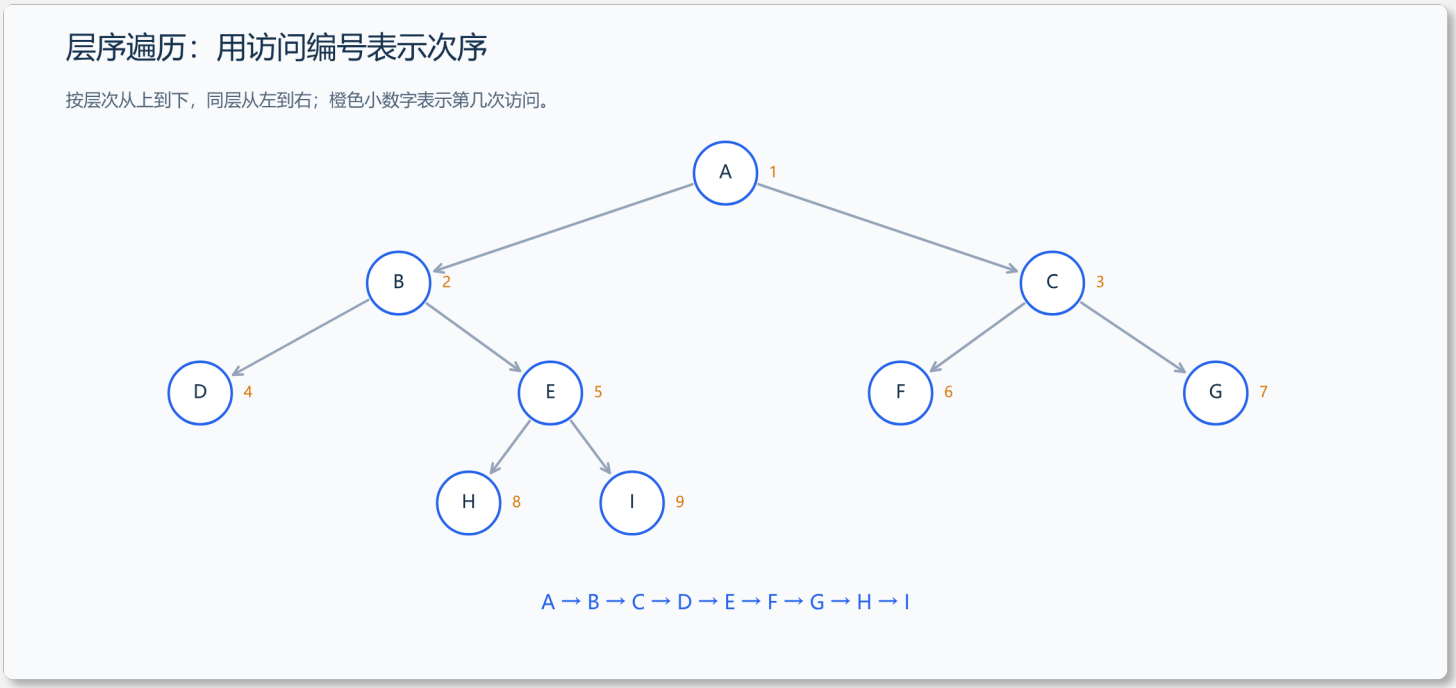
1.4 二叉树的遍历

示例二叉树：



1.4.1 层序遍历

访问顺序：按层次从上到下，每层从左到右

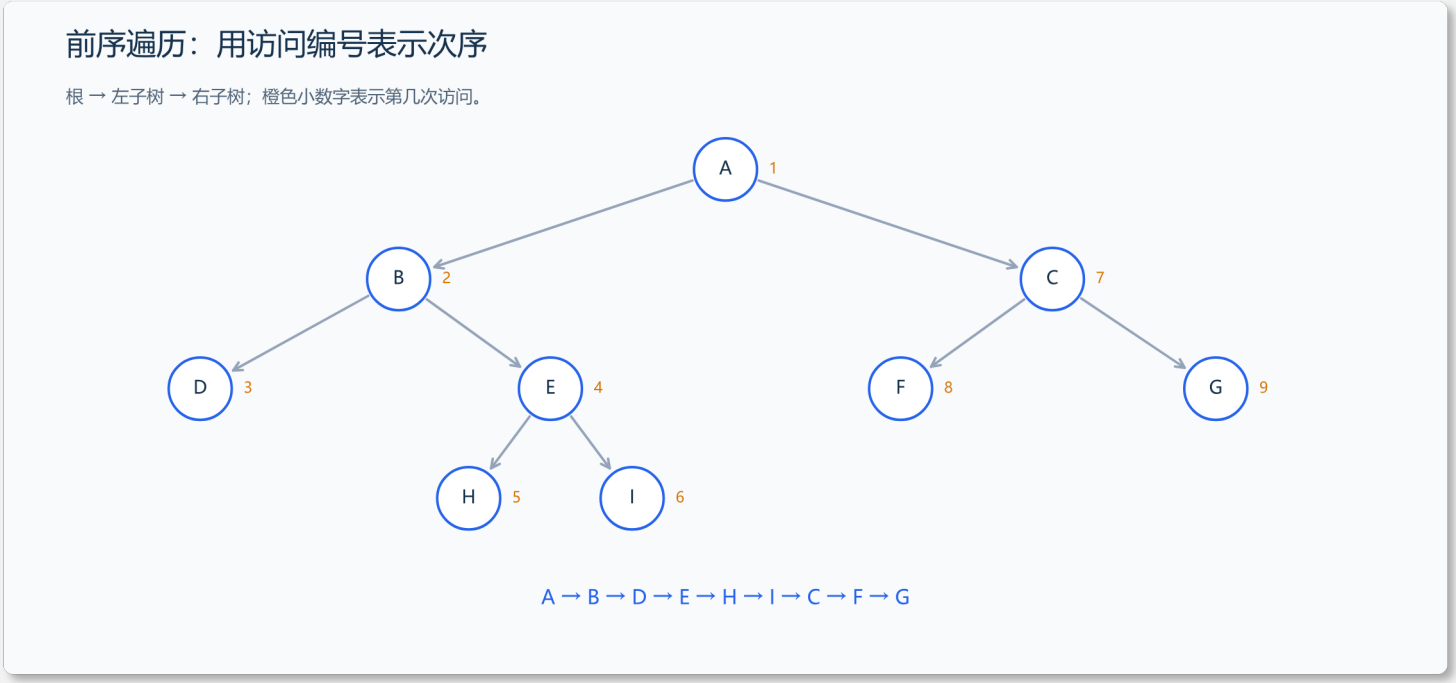


橙色小数字表示访问次序；树边表示父子关系，依次按层读取。

所以层序遍历为：A - B - C - D - E - F - G - H - I

1.4.2 前序遍历

访问顺序：根节点 → 左子树 → 右子树

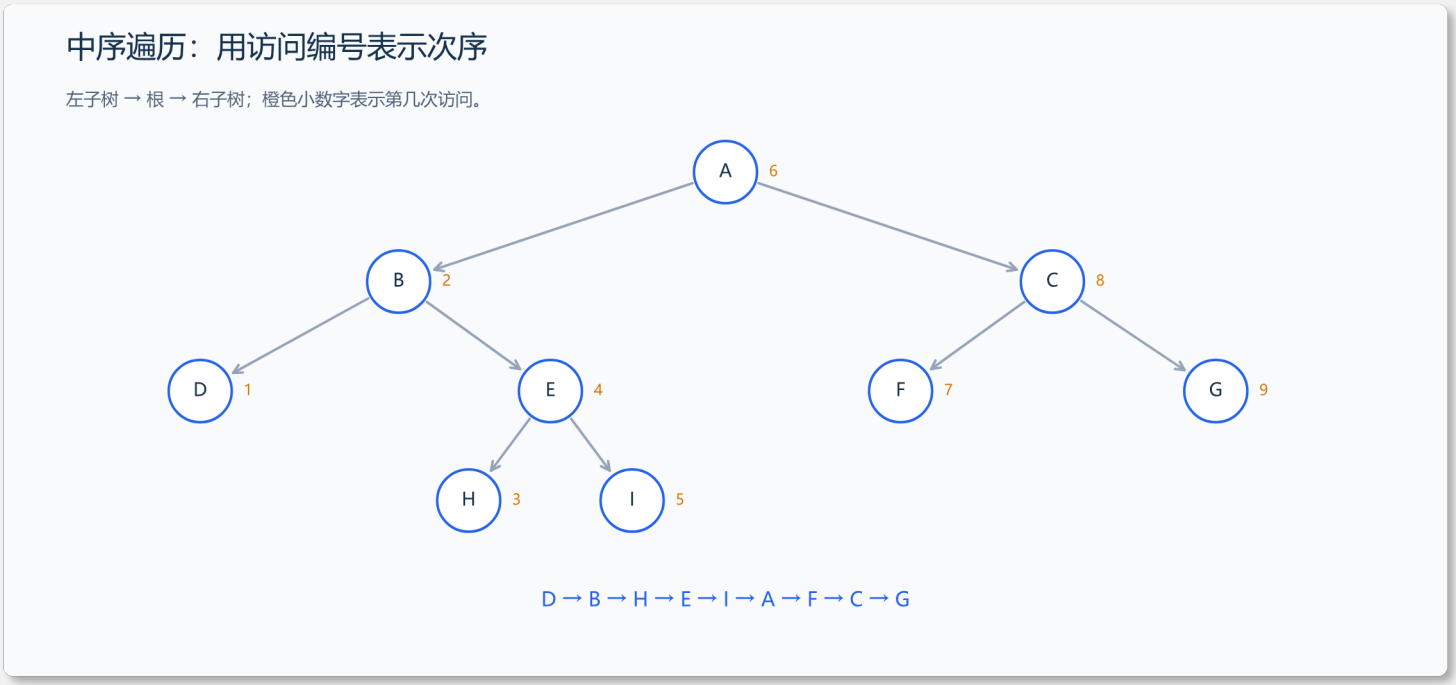


先根、再左、再右；用编号和下方序列表示访问顺序，树边只表示父子关系。

所以前序遍历为：A - B - D - E - H - I - C - F - G

1.4.3 中序遍历

访问顺序：左子树 → 根节点 → 右子树

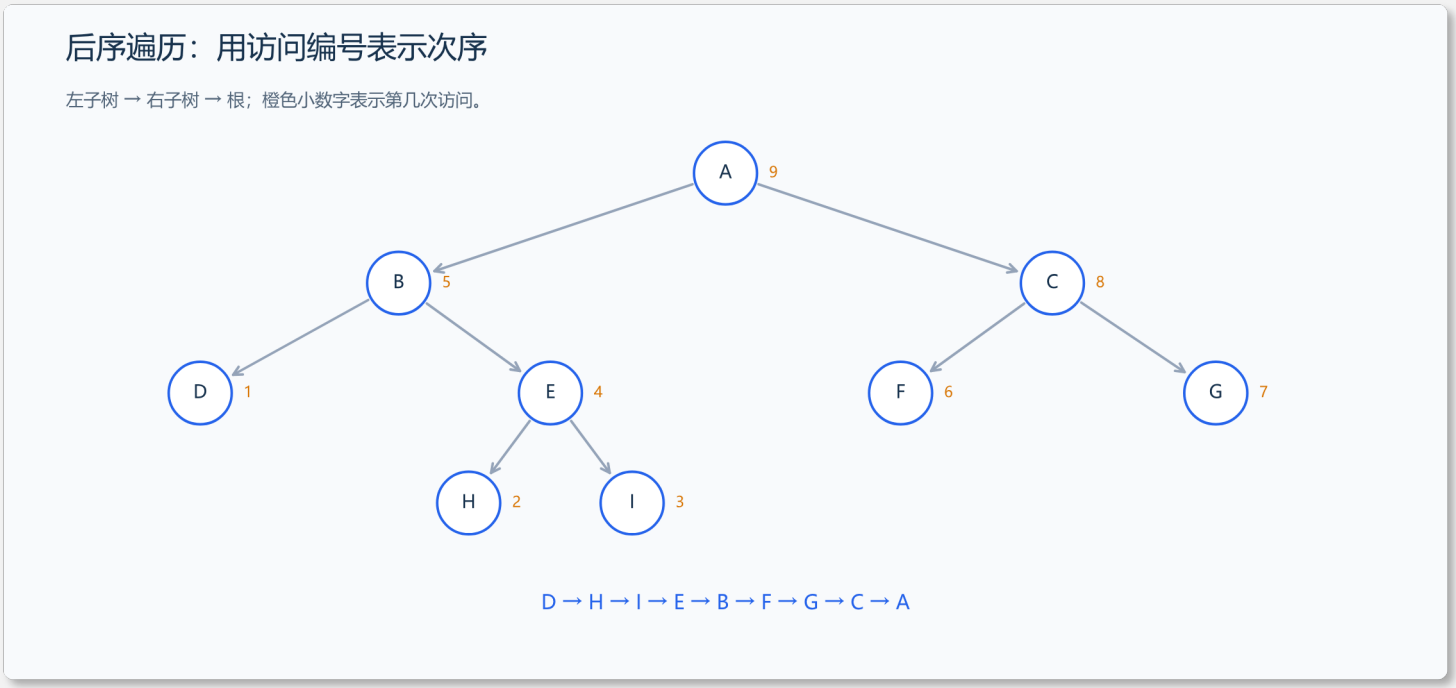


先左、再根、再右；当前节点要等左子树全部访问后才访问。

所以中序遍历：D - B - H - E - I - A - F - C - G

1.4.4 后序遍历

访问顺序：左子树 → 右子树 → 根节点



先左、再右、再根；子树处理完成以后才访问其根节点。

所以后序遍历：D - H - I - E - B - F - G - C - A

遍历方式	应用场景
前序遍历	复制二叉树、前缀表达式
中序遍历	二叉搜索树的有序输出、中缀表达式
后序遍历	释放二叉树内存、后缀表达式、计算目录大小
层序遍历	查找最短路径、按层次处理节点

这四张图用于核对同一棵树的完整访问次序。执行时怎样暂停在子调用、返回后何时访问根、队列如何进出，会在后面的二叉树实现中用 5 个节点逐步展开；先理解这些动作，再按相同规则推到这棵 9 节点树。

1.4.5 遍历序列与二叉树的唯一性

- 节点标识互不相同时，**前序 + 中序**或**后序 + 中序**可以唯一确定二叉树。
- 一般情况下只有前序 + 后序不能唯一确定；重复值需要额外的节点身份或约束。

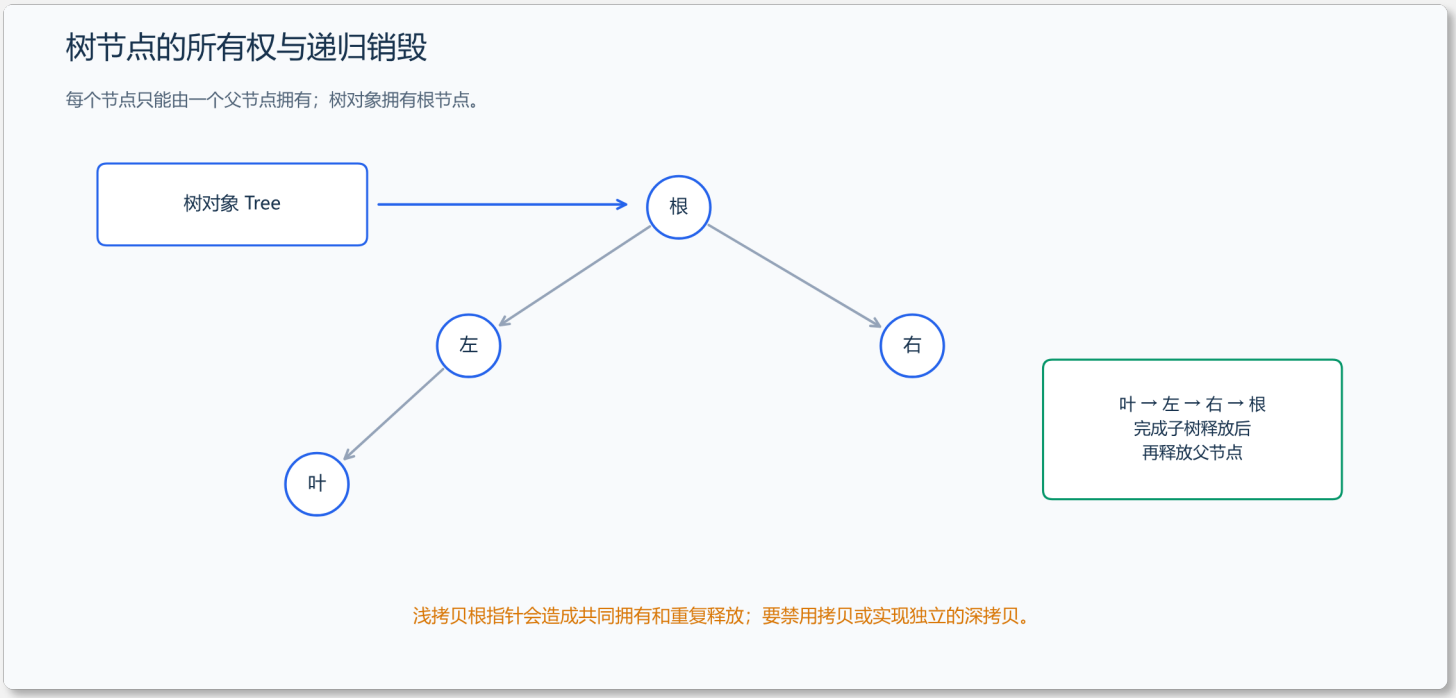
2. 按层序插入与删除的二叉树实现

2.1 二叉树节点类

题目原文（自拟示例）： 输入 n 个互不相同的整数，按层序放入普通二叉树，再输入一个待删除值。输出四种遍历结果，并删除目标后输出层序结果；允许空树，找不到目标时树不变。

思路分析： 层序插入寻找第一个空位；删除时把最后一个节点的值复制到目标位置，再摘除最后节点。这个操作维护普通树结构，不维护 BST 的大小关系。

节点类拥有左右子树，二叉树对象拥有根节点；所有节点由 `new` 创建，不允许共享同一个子节点或拷贝整个所有权结构。



每个孩子只由一个父节点拥有，根由树对象拥有；析构根会递归销毁左右子树，不能把同一节点交给两个父节点。

这里的 `TreeNode(const TreeNode&) = delete;` 与赋值函数末尾的 `= delete` 禁止这两种拷贝操作；它们不是释放内存的 delete 表达式。节点拥有左右子树的释放责任，若只复制地址，就可能让两份对象重复释放同一子树。后面的 BinaryTree 也按同样理由禁止直接拷贝。

```
/*
 * 二叉树节点类
 *
 * 定义了二叉树的基本结构，包含节点值和左右子节点指针
 */
class TreeNode {

public:
    int val;          // 节点值
    TreeNode *left;   // 左子节点指针
    TreeNode *right;  // 右子节点指针

```

```cpp
    /*
     * 默认构造函数
     *
     * 初始化节点值为0，左右子节点指针为空
     */
    TreeNode() : val(0), left(nullptr), right(nullptr) {}
    TreeNode(const TreeNode &) = delete;
    TreeNode &operator=(const TreeNode &) = delete;

    /*
     * 带参数的构造函数
     *
     * 参数 x: 用于初始化节点的值
     */
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

    ~TreeNode() {
        delete left;
        delete right;
    }
};
```

2.2 BinaryTree 的接口与成员操作

题目原文（自拟）： 输入 n（1≤n≤10000）和 n 个 int 范围的整数，按输入顺序层序插入一棵普通二叉树；重复值不插入。输出层序、前序、中序、后序遍历。再读一个待删除值，用最深最右叶子的值替换目标节点并移除该叶子，最后输出层序遍历；目标不存在则结构不变。遍历同层或同父节点时先左后右，输出格式见后面样例。

思路分析： 普通二叉树没有搜索树的大小约束。插入依据层序的首个空位，遍历依据访问顺序，删除则根据末尾叶子的替换规则；三个过程分别实现，不能以 BST 删除规则代替。

下面代码与上一段节点类合用，还需要 `<iostream>`、`<queue>` 和 `using namespace std;`。

代码组合方式： 在同一个 `.cpp` 中依次放置 TreeNode 定义、下面的 BinaryTree 类定义、各成员函数实现和唯一的 main。成员实现不能单独运行；`BinaryTree::` 表示它属于这个类。

2.2.1 先确定类的接口与所有权

TreeNode 析构时递归释放孩子，BinaryTree 只需释放根。禁止浅拷贝，避免两个对象重复释放同一棵树。成员声明先列出接口，下面再分开实现。

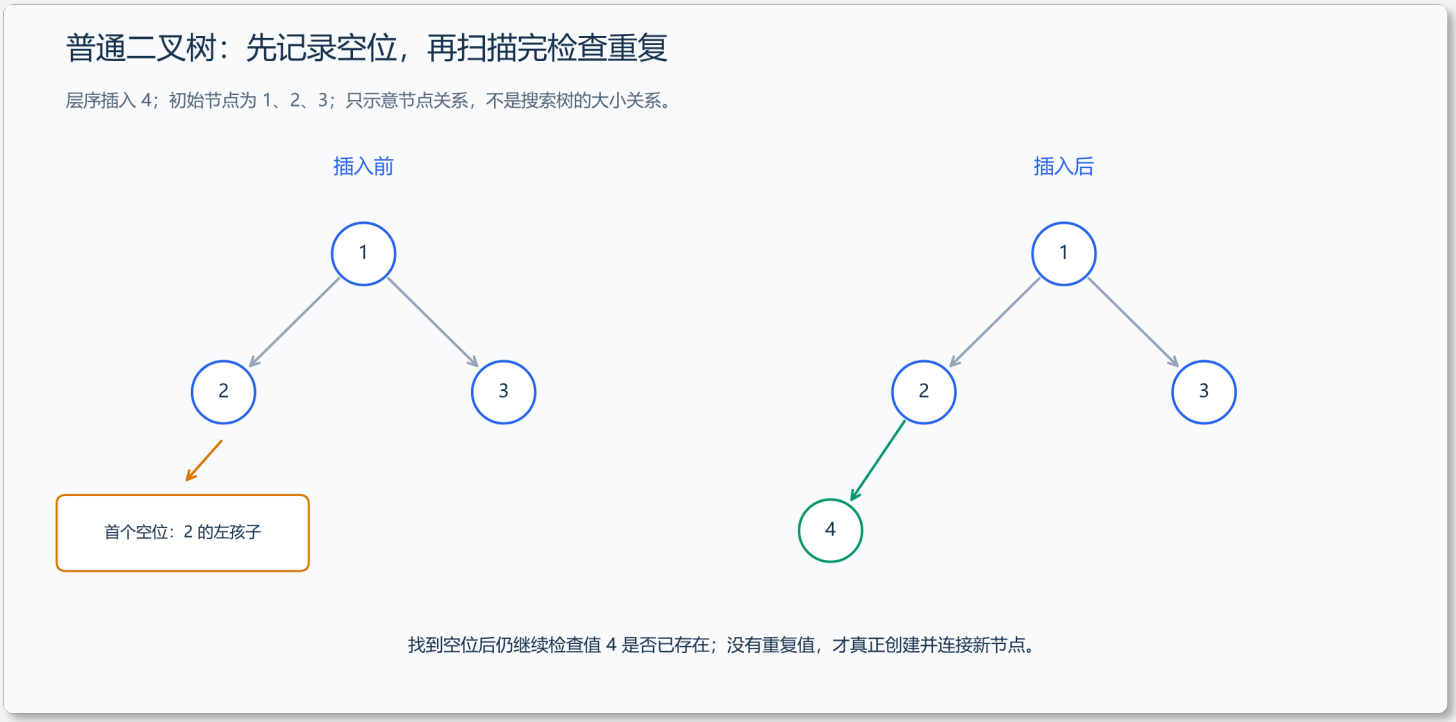
```cpp
#include <iostream>
#include <queue>
using namespace std;

// 先放入上一小节的 TreeNode 定义；root 拥有整棵树。
class BinaryTree {
public:
    TreeNode *root;
    BinaryTree() : root(nullptr) {}
    BinaryTree(const BinaryTree &) = delete;
```

```cpp
    BinaryTree &operator=(const BinaryTree &) = delete;
    ~BinaryTree() { delete root; }
    bool insert(int value);
    void levelOrder();
    void preOrder(TreeNode *node);
    void inOrder(TreeNode *node);
    void postOrder(TreeNode *node);
    TreeNode *deleteNode(TreeNode *root, int key);
};
```

2.2.2 层序寻找插入位置

队列按层扫描，记下第一个缺孩子的节点。即使已找到空位，也继续扫描全树，检查值是否重复；不存在重复才真正插入。单次插入 O(n)，连续插入 n 个不同值的总量可能为 O(n²)。



队列按层扫描，记下第一个缺孩子的节点。



在根 1、孩子 2 和 3 的树中插入 4：队列先处理 1，再处理 2，此时记下 2 的左空位；随后仍处理 3，确认没有值 4，才创建并连接新节点。若插入的是 3，处理到现有节点 3 就返回 false，之前记下的空位不发生写入。记录位置和实际插入是两个时机，不能混在一起。

```
// 队列按层扫描，记下第一个缺孩子的节点。
bool BinaryTree::insert(int value) {
    if (!root) {
        root = new TreeNode(value);
        return true;
    }
    queue<TreeNode *> pending;
    pending.push(root);
    TreeNode *vacancy = nullptr;
    bool leftVacancy = false;
    while (!pending.empty()) {
        TreeNode *node = pending.front();
        pending.pop();
        if (node->val == value)
            return false;
        if (node->left)
            pending.push(node->left);
        else if (!vacancy) {
            vacancy = node;
            leftVacancy = true;
        }
        if (node->right)
            pending.push(node->right);
        else if (!vacancy) {
            vacancy = node;
            leftVacancy = false;
        }
    }
```

```cpp
    TreeNode *added = new TreeNode(value);
    if (leftVacancy)
        vacancy->left = added;
    else
        vacancy->right = added;
    return true;
}
```

2.2.3 队列驱动层序遍历

取出队头并输出，再把左、右孩子依次入队。队列始终保存尚未访问的节点；每个节点处理一次，时间 O(n)，队列空间 O(w)，w 为最大层宽。



取出队头并输出，再把左、右孩子依次入队。

```cpp
// 取出队头并输出，再把左、右孩子依次入队。
void BinaryTree::levelOrder() {
    // 检查根节点是否为空
    if (!root)
        return;
    queue<TreeNode *> q;
    // 将根节点入队
    q.push(root);
    // 当队列不为空时继续遍历
    while (!q.empty()) {
        // 取出队首节点并出队
        TreeNode *current = q.front();
        q.pop();
        // 输出当前节点值
        cout << current->val << " ";
        // 将左右子节点依次入队
        if (current->left)
            q.push(current->left);
        if (current->right)
            q.push(current->right);
    }
```

```cpp
    // 遍历完成后换行
    cout << endl;
}
```

2.2.4 前序：根、左、右

空指针立即返回。先访问当前根，再递归左子树和右子树；递归深度为树高 h，时间 O(n)，调用栈 O(h)。



绿色标出根节点的访问位置，子树内部递归采用相同规则。

递归树遍历：看返回位置，再决定何时输出根

树：1 的孩子为 2、3，2 的孩子为 4、5；三种模式的递归调用顺序相同。

当前执行 / 返回位置	前序输出	中序输出	后序输出
1 进入 1	1	—	—
2 进入 2	2	—	—
3 进入叶子 4，空孩子返回	4	4	4
4 4 返回 2，左子树完成	—	2	—
5 进入叶子 5，空孩子返回	5	5	5
6 5 返回 2，右子树完成	—	—	2
7 2 返回 1，左子树完成	—	1	—
8 进入叶子 3，空孩子返回	3	3	3
9 3 返回 1，右子树完成	—	—	1
只有访问语句的位置变化	1,2,4,5,3	4,2,5,1,3	4,5,2,3,1

用根 1、孩子 2 和 3、节点 2 的孩子 4 和 5 来跟踪调用。前序在刚进入 2 时输出 2；中序等左侧 4 整棵子树处理完、返回 2 后才输出 2；后序还要等右侧 5 返回后才输出 2。子调用运行时，父调用停在调用语句处，返回后接着下一句，不能把“返回父节点”理解为重新执行整个父函数。三种输出依次为 `1 2 4 5 3`、`4 2 5 1 3`、`4 5 2 3 1`。

```cpp
// 空指针立即返回。
void BinaryTree::preOrder(TreeNode *node) {
    // 递归终止条件：如果当前节点为空，则直接返回
    if (node == nullptr)
        return;

    // 访问当前节点：打印节点值
    cout << node->val << " ";

    // 递归遍历左子树
    preOrder(node->left);

    // 递归遍历右子树
    preOrder(node->right);
}
```

2.2.5 中序：左、根、右

先完整处理左子树，返回后输出根，最后处理右子树。二叉搜索树把较小值放在左侧、较大值放在右侧，所以它的中序遍历恰好从小到大输出；普通二叉树的中序则按结构给出左、根、右的访问次序。

```cpp
// 先完整处理左子树，返回后输出根，最后处理右子树。
void BinaryTree::inOrder(TreeNode *node) {
    // 递归终止条件：如果当前节点为空，则直接返回
    if (node == nullptr)
        return;

    // 递归遍历左子树
    inOrder(node->left);

    // 访问当前节点：打印节点值
    cout << node->val << " ";

    // 递归遍历右子树
    inOrder(node->right);
}
```

2.2.6 后序：左、右、根

孩子处理完才访问根。删除整棵树也沿这个顺序：先通过父节点的 left、right 找到并释放两棵子树，最后释放父节点。这样读取孩子指针时，父节点仍然存在。

```cpp
// 孩子处理完才访问根。
void BinaryTree::postOrder(TreeNode *node) {
    // 递归终止条件：如果当前节点为空，则直接返回
    if (node == nullptr)
        return;

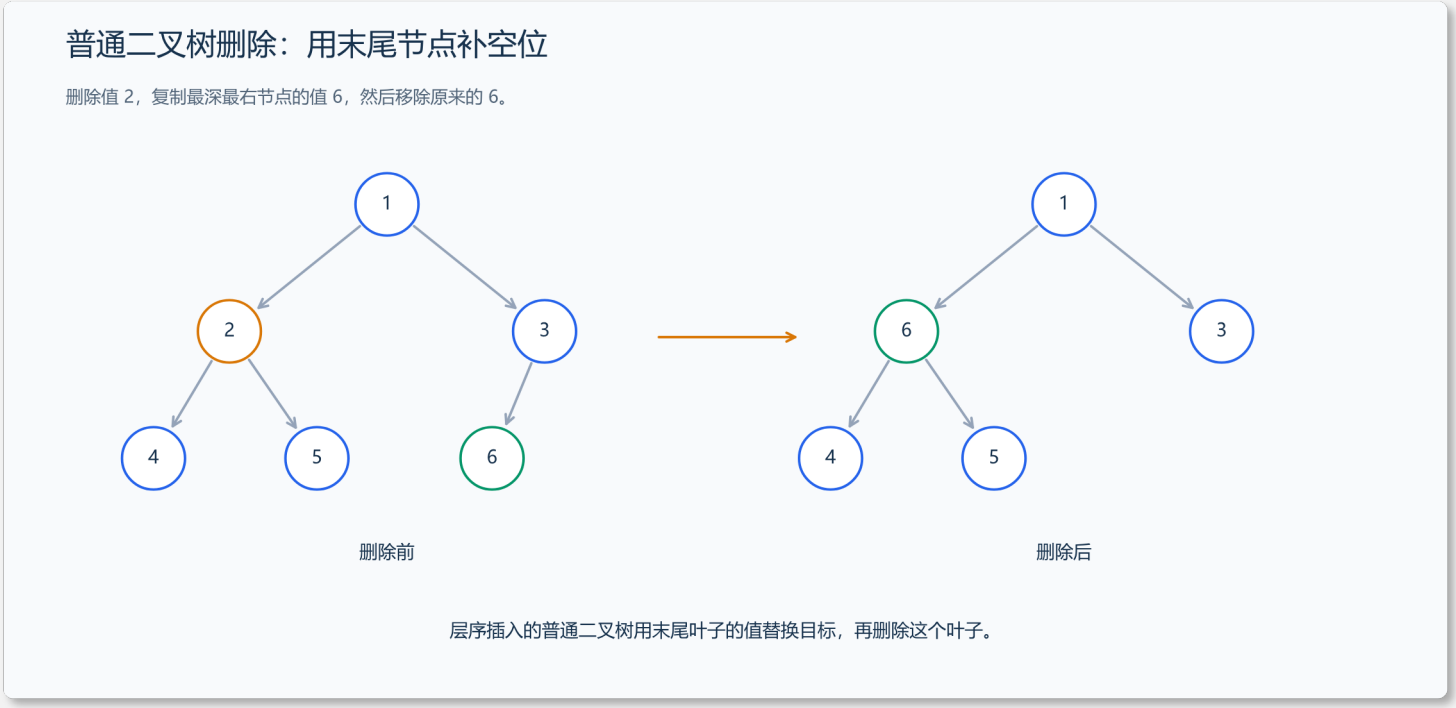
    // 递归遍历左子树
    postOrder(node->left);

    // 递归遍历右子树
    postOrder(node->right);

    // 访问当前节点：打印节点值
    cout << node->val << " ";
}
```

2.2.7 用最后一个节点替换目标并删除叶子

层序扫描同时找到目标、最后访问的节点及其父节点。最后一个节点若有孩子，就不可能最后出队，因此一定是叶子；把它的值复制到目标后，解除父节点的对应孩子指针并删除叶子。空树、单节点和目标不存在分别处理，最后把返回的根指针赋给 tree.root，树便指向删除后的根。



层序扫描同时找到目标、最后访问的节点及其父节点。



层序值为 `[1,2,3,4,5,6]`、删除值 2 时，扫描 2 只记下目标，不能立刻释放；扫描到最后的 6，才知道替换值及其父节点 3。把 6 复制到目标节点，再将原节点 6 的父指针 `3->left` 置空，最后释放该叶子。结果层序为 `[1,6,3,4,5]`；只复制值而不解除指针，会留下指向已释放内存的连接。

```
// 层序扫描同时找到目标、最后访问的节点及其父节点。
TreeNode *BinaryTree::deleteNode(TreeNode *root, int key) {
    if (root == nullptr)
        return nullptr;
    if (!root->left && !root->right) {
```

```
        if (root->val == key) {
            delete root;
            return nullptr;
        }
        return root;
    }

    // 用于查找目标节点和最后节点
    TreeNode *target = nullptr; // 要删除的节点（只删值）
    TreeNode *last = nullptr;   // 最深最右节点
    TreeNode *parent = nullptr; // 最深最右节点的父节点

    queue<TreeNode *> q;
    q.push(root);

    while (!q.empty()) {
        TreeNode *node = q.front();
        q.pop();

        // 记录最后一个节点
        last = node;

        // 找到目标节点
        if (node->val == key) {
            target = node;
        }

        // 入队孩子，并记录父节点
        if (node->left) {
            parent = node;
            q.push(node->left);
        }
        if (node->right) {
            parent = node;
            q.push(node->right);
        }
    }

    // 若未找到目标节点，直接返回
    if (target == nullptr) {
        return root;
    }

    // 将最深节点值放到目标位置
    target->val = last->val;

    // 删除最深节点
    if (parent->right == last) {
        delete parent->right;
        parent->right = nullptr;
    } else if (parent->left == last) {
        delete parent->left;
        parent->left = nullptr;
    }

```

```cpp
    return root;
}
```

2.2.8 组合调用与输出

先读入节点值，调用四种遍历，再读入要删除的值并写回返回的根。下面的输出对应后面的样例；重复输入值会被 insert 拒绝。

```cpp
// 将类定义和上面各成员实现放在 main 之前。
int main() {
    BinaryTree tree;
    int n;
    cin >> n;
    while (n--) {
        int x;
        cin >> x;
        tree.insert(x);
    }
    cout << "层序遍历: ";
    tree.levelOrder();
    cout << "前序遍历: ";
    tree.preOrder(tree.root);
    cout << endl;
    cout << "中序遍历: ";
    tree.inOrder(tree.root);
    cout << endl;
    cout << "后序遍历: ";
    tree.postOrder(tree.root);
    cout << endl;
    int value;
    cin >> value;
    tree.root = tree.deleteNode(tree.root, value);
    cout << "删除节点后，层序遍历: ";
    tree.levelOrder();
}
```

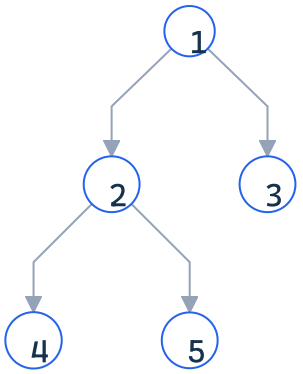
输入：

```text
5
1 2 3 4 5
2
```

输出：

```text
层序遍历: 1 2 3 4 5
前序遍历: 1 2 4 5 3
中序遍历: 4 2 5 1 3
后序遍历: 4 5 2 3 1
删除节点后，层序遍历: 1 5 3 4
```

层序插入构建的是如下二叉树



因此

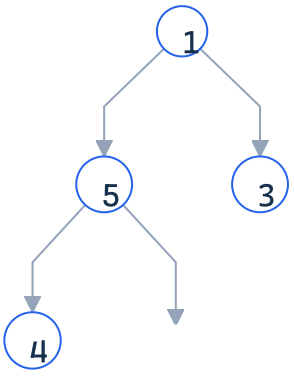
层序遍历为：1 2 3 4 5

前序遍历为：1 2 4 5 3

中序遍历为：4 2 5 1 3

后序遍历为：4 5 2 3 1

删除后二叉树：



练习题： 洛谷 P4913 – 二叉树深度、洛谷 P3884 – 二叉树问题。

（九） 双指针与滑动窗口（尺取法）

对有序数据或连续区间，不一定要重新枚举每一对端点。先跟踪两个位置怎样移动，证明每次移动保留了哪些可能答案。

双指针是高效遍历与查找序列关系的一种技巧，通过两个下标在同一或不同数组上同时移动，避免重复计算。

> **注意：** 这里的“双指针”通常指两个数组或字符串下标；它们不必是保存内存地址的 C++ 指针变量。

1. 头尾双指针

两个下标分别从序列两端开始，按题目的排除规则向中间移动。

例如，头尾双指针的两个经典应用：

1.1 从两端判断回文字符串

题目原文（自拟）： 给定按单字节字符处理的字符串 `s`，判断从左到右和从右到左读到的字符是否完全相同。空串和单字符都算回文，返回 `bool`。大小写、空格和标点都参与比较，不自动忽略。

思路分析： 回文要求每个位置 `i` 都满足 `s[i]==s[n-1-i]`。从两端开始检查一对字符，相等后一起向中间移动。已检查的外侧已经满足条件，剩下只需检查内部；一旦有一对不同就可以返回 `false`。指针相遇时，中央字符与自己相等；交错时所有对应位置已检查完。



已匹配的两端不再检查，未验证区间逐步缩小。

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

// 两端字符逐一比较；空串时右下标为 -1，循环直接跳过。
bool isPalindrome(const string &s) {
    int left = 0, right = static_cast<int>(s.size()) - 1;
    while (left ≤ right) {
        if (s[left] ≠ s[right])
            return 0;
        left++;
        right--;
    }
    return 1;
}
```

样例与代码解释： `abba` 先比较两个 `a`，再比较两个 `b`，返回 `true`；`abc` 第一轮的 `a` 与 `c` 不同，返回 `false`。空串令 `right=-1`，不进入循环而返回 `true`；不是先把无符号的 `size()` 减一再转换，所以没有无符号减法下溢。

复杂度： 每轮比较两端，再向中间移动，最多比较约一半字符，时间 $O(n)$，额外空间 $O(1)$。例如 racecar 的比较顺序是 r/r、a/a、c/c，剩下的 e 位于正中间。

练习题： 洛谷 P1217 – 回文质数（综合回文构造与素数判断）。

1.2 两数之和：从两端排除候选

题目原文（自拟）： 给定非递减整数数组 `arr` 和目标和 `target`，找到两个不同下标 `i<j`，满足 `arr[i]+arr[j]==target`。存在时返回任意一组下标，否则返回 `(-1,-1)`。数组长度可用 `int` 下标表示，元素为 `int`，目标为 `long long`。

思路分析： 初始令 `left=0`、`right=n-1`。从数组的有序关系，可以看出两端之和怎样决定下一步移动：

1. 和偏小：对当前范围内任意 `j≤right`，都有 `arr[left]+arr[j]≤arr[left]+arr[right]<target`。当前左端与谁配对都不够，因此排除左端。
2. 和偏大：对任意 `i≥left`，都有 `arr[i]+arr[right]≥arr[left]+arr[right]>target`。当前右端与谁配对都超出，因此排除右端。
3. 和相等：直接返回。循环要求 `left<right`，防止同一位置被使用两次。



图以 `[1,2,4,7,11]`、目标 9 为例：先排除 11，再排除 1，最后得到下标 `(1,3)`。每一步排除都依赖排序带来的不等式。

```cpp
// 数组有序时：和太大缩右端，和太小移左端；left < right 防止复用。
#include <utility>
#include <vector>
using namespace std;

```

```cpp
pair<int, int> twoSum(const vector<int> &arr, long long target) {
    int left = 0, right = static_cast<int>(arr.size()) - 1;
    while (left < right) {
        long long sum = 1LL * arr[left] + arr[right];
        if (sum == target)
            return {left, right};
        if (sum > target)
            --right;
        else
            ++left;
    }
    return {-1, -1};
}
```

样例与代码解释： `[1,2,4,7]`、目标 9 返回 `(1,3)`，对应 2+7；`[3]`、目标 6 返回 `(-1,-1)`，因为不能使用同一个 3 两次。计算和时先转为较宽整数，避免两个 `int` 相加后才赋值造成溢出。

复杂度分析： 每轮至少移动一个端点，端点总移动次数为 $O(n)$，额外空间 $O(1)$。若输入未排序，可带着原下标排序，或使用哈希表查互补值，得到另一种查找过程。后面的整数二分章节会给出逐个查互补值的实现。

练习题： 洛谷 P1102 - A-B 数对（变式：求差且统计方案，注意相同值的数量）。

1.3 头尾双指针的移动模板

```c++
// 标准头尾双指针模板
int functionName(vector<int>& arr,int target){
    int left = 0, right = static_cast<int>(arr.size()) - 1;
    while (left ≤ right) {  // 或 left < right，根据具体需求
        // 处理逻辑
        if (condition) {
            // 找到目标
         return result;
        }
        else if (condition1)
            left++;    // 左指针右移
        else
            right--;   // 右指针左移
    }
}
```

练习题： 洛谷 P1102 - A-B 数对。

2. 快慢双指针

两个下标从同一侧开始，同向扫描，分别负责读取和写入；链表判环中则常使用不同步长。

2.1 移除有序数组中的重复元素

题目原文（自拟）： 给定非递减整数数组 `nums`，在原数组中保留每种值一次，且仍按升序排列。返回有效前缀长度 `k`，要求前 `k` 项恰好为去重结果；无须改变 `vector` 的实际长度，末尾内容可以保留。空数组返回 `0`。

思路分析： 第一个元素直接保留，令 `slow=0`。`fast` 从下标 1 开始读取；若新值等于最后保留值，就跳过；否则先把 `slow` 加一，再把新值写入该位置。排序保证相同值相邻，只需与最后保留值比较。

循环不变量是：`nums[0..slow]` 已是扫描过部分的正确去重结果，`slow≤fast`。写入位置不会超过读取位置，因而不会破坏尚未扫描的后缀。



图以 `[1,2,3,3,4,5]` 为例：第二个 3 被跳过，随后 4、5 依次写到有效前缀末尾。绿色区域表示结果范围。

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

int removeDuplicates(vector<int> &nums) {
    if (nums.empty())
        return 0;

    int slow = 0; // 慢指针 - 指向当前唯一元素的位置
    for (int fast = 1; fast < nums.size(); fast++) {
        // 快指针寻找不同的元素
        if (nums[fast] != nums[slow]) {
            slow++;
            nums[slow] = nums[fast];
        }
    }
    return slow + 1; // 返回新长度
}
```

样例与代码解释： 最后 `slow=4`，返回 5；前五项为 `[1,2,3,4,5]`。`slow` 是下标，所以长度需要再加一。若确实要缩短容器，可在返回后执行 `nums.resize(k)`；若只按题目要求返回长度，则不必这样做。

复杂度： 只扫描一次，时间 $O(n)$、额外空间 $O(1)$。排序后相同值紧挨在一起，只需和上一个保留值比较，就能识别这一轮是否出现新值。

快慢指针也可用于链表判环：慢指针走一步，快指针走两步；一旦二者进入同一环，相对位置每轮推进一步，最终相遇。此时指针表示节点位置，与本例的数组读写职责不同。参见 链表环检测。

练习题： 洛谷 P1059 - 明明的随机数（先排序，再去重）。

2.2 快慢双指针的读写模板

```c++
// 伪代码：把“满足保留条件”换成题目要求的判断。
int slow = 0;
for (int fast = 0; fast < n; fast++) {
    if (/* 满足保留条件 */) {
        nums[slow] = nums[fast];
        slow++;
    }
}
// slow 是保留元素数量，也是下一次写入的位置，不是最后一个有效下标
```

练习题： 洛谷 P1059 - 明明的随机数（排序去重练习）。

3. 滑动窗口

滑动窗口是一种高效解决 **子区间问题（subarray / substring）** 的算法技巧，通过动态维护一个满足条件的区间，避免暴力枚举所有区间带来的重复计算。滑动窗口使用两个边界下标维护连续区间；它与原地去重中的读写下标含义不同。

核心思想：

通过两个指针（通常是 `left` 与 `right`）在序列上移动，维护一个“窗口区间” `[left, right)`。

- 右指针向右移动（扩大窗口） → **加入新元素**
- 左指针向右移动（收缩窗口） → **移除旧元素**
- 在此过程中实时更新窗口内的信息（如和、长度、字符频率等）

> **注意：** 滑动窗口本质上是一种 **动态区间维护算法**，但常用快慢指针来维护。

3.1 区间和：扩张右端与移动左端

题目原文（自拟）： 给定正整数数组 `arr` 和正目标 `s`，找一个非空连续区间，使区间和恰好为 `s`。存在时返回半开区间 `[left,right)`，否则返回 `(-1,-1)`；空数组无解。数组长度可用 `int` 表示，整个数组的和不超出 `long long` 范围。

思路分析： 维护窗口和 `sum`，初始窗口为空。向右加入一个新数后，若和太大，就不断删去左端；缩小到不再大于目标后，检查是否相等。

为什么这样不会漏解？正数保证加入元素只能增大和，删去左端只能减小和。窗口已经超过目标时，保留当前左端并继续往右只会更大，所以可以排除这个左端；窗口不足目标时，则继续尝试向右扩张。若有负数，这两个单调变化都可能失效，应考虑前缀和与映射等其他方法。



图中加入前三项后 `[0,3)` 的和为 6，超出目标 5；删除左端 1 后，`[1,3)` 的和变成 5，对应 2、3。

```cpp
#include <vector>
#include <utility>
using namespace std;

pair<int, int> findSum(const vector<int> &arr, long long s) {
    if (s ≤ 0)
        return {-1, -1}; // 寻找非空区间，数组元素为正数
    int left = 0, right = 0;
    long long sum = 0;
    while (right < arr.size()) {
        sum += arr[right++];            // 扩大窗口
        while (sum > s && left < right) // 收缩窗口
            sum -= arr[left++];
        if (sum == s)
            return {left, right};
    }
```

```cpp
    return {-1, -1};
}
```

样例与代码解释： `[1,2,3,4]`、`s=5` 返回 `(1,3)`。`right++` 出现在加入元素的语句中，因此 `right` 始终是尾后下标，返回前不再加一。函数把空区间与找不到的情况统一返回无解，找到时则返回包含实际元素的起止位置。

复杂度： 内层收缩循环虽然嵌套在外层中，但左端只向右走，每个元素最多加入、删去各一次，所以总时间 $O(n)$，额外空间 $O(1)$。

3.2 最长无重复子串

题目原文（自拟）： 给定按单字节字符处理的字符串 `s`，求不含重复字符的最长连续子串长度。空串返回 `0`；长度可用 `int` 表示。大小写视为不同字符。

思路分析： 集合保存当前半开窗口 `[left,right)` 中的字符。若 `s[right]` 不在集合中，就加入它并扩大右端；若已经存在，就先删除左端字符、右移左端，再检查同一个右端字符，直到能够合法加入。每次扩大右端后更新最大长度。

移动左端时，先从集合中删除对应字符，再将 `left` 加一；扩展右端时，把新字符加入集合，再将 `right` 加一。集合因此始终保存窗口里的字符。对一个固定右端，从左侧逐项删到冲突字符离开，就得到这个右端下尽可能长的不重复窗口。



图以 `abca` 为例：扫描最后一个 `a` 时，已有窗口为 `abc`；先删除左端 `a`，再加入新 `a`，得到 `bca`。字符只有真正进入窗口后才右移 `right`。

```cpp
#include <string>
#include <algorithm>
#include <unordered_set>
using namespace std;

// 集合保存窗口中的字符；重复时移除左端，直到可以加入右端字符。
```

```cpp
int lengthOfLongestSubstring(string s) {
    unordered_set<char> window;
    int left = 0, right = 0, res = 0;

    while (right < s.size()) {
        if (!window.count(s[right])) {
            window.insert(s[right]);
            right++;
            res = max(res, right - left);
        } else {
            window.erase(s[left]);
            left++;
        }
    }
    return res;
}
```

样例与代码解释： `abcabcbb` 返回 3，`bbbbb` 返回 1，空串返回 0。`res` 保存的是历史最大长度，不是最终窗口长度；例如 `abcabcbb` 最后的窗口很短，但答案仍为 3。

复杂度： 每个字符最多进、出窗口各一次，哈希操作平均常数级时，扫描时间 $O(n)$。集合最多保存当前出现的不同字符，空间 $O(min(n, \text{字符集大小}))$。本函数按值传入字符串，还会产生一份 $O(n)$ 输入副本，因此整个调用的额外空间为 $O(n)$；改为 `const string&` 可以省去副本。

3.3 滑动窗口的扩张与收缩模板

```c++
// 伪代码：add/remove 表示维护窗口状态的操作。
int left = 0, right = 0;
while (right < n) {
    // 扩大窗口
    window.add(s[right]);
    right++;

    while (window 不满足条件) {
        // 收缩窗口
        window.remove(s[left]);
        left++;
    }
    // 在这里更新答案（窗口满足条件时）
}
```

练习题： CF 279B – Books、洛谷 P1638 – 逛画展。

（十） 二分与三分搜索

前面的归并排序把两段分别排好，再合并成完整序列。本章换成查找任务：比较中间位置后排除一段候选，将注意力留在仍可能包含答案的部分；先学二分怎样舍弃一半，再看三分怎样沿曲线缩短区间。

将一个大问题拆成同类的小问题，分别求解，再合并结果，叫作分治。归并排序展示了“拆分、求解、合并”；二分查找则在比较后只保留一半继续查找，每轮都缩小同一个任务的规模。

1. 二分搜索

在升序数组中比较中间项与目标：中间项小于目标时，它左侧也都太小；中间项大于目标时，它右侧也都太大。因此一次比较就能排除一半位置，这就是二分查找。

这个思路还可以用来寻找第一个 true 或最后一个 true。例如速度越大越容易按时完成任务，判定结果就像排好的 false、true；不用真的建立数组，也能二分定位变化的位置。

1.1 整数二分

整数二分用于在 **有序的整数数组** 中查找某个元素，或者寻找一个满足某种性质的边界点。这是最经典的二分应用。

1.1.1 查找一个等于目标的元素

题目原文（自拟）： 输入一个升序（允许相等）的整数 vector arr 与整数 target，数组长度和下标均在 int 范围内。返回某个满足 `arr[index]` =target 的下标；不存在返回 -1。允许空数组；有重复值时不要求最左位置。

思路分析： 维护闭区间 [left,right]，只有其中可能存在答案。中点等于 target 就返回；中点较大时，升序保证中点及其右侧都不可能匹配，令 right=mid-1；中点较小时对称地令 left=mid+1。每轮保留全部可能答案且至少删除中点，区间最终为空则无解。

手算与复杂度： arr=[1,2,3,4,5,6,7,8,9,10]，target=7。候选下标 [0,9] 的中点 4 对应 5，保留 [5,9]；中点 7 对应 8，保留 [5,6]；中点 5 对应 6，保留 [6,6]，找到下标 6。每轮候选约减半，时间 O(log(n+1))，额外空间 O(1)。

练习题： 洛谷 P2249 - 查找（要求第一次出现，应使用接下来的下界方法）。

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

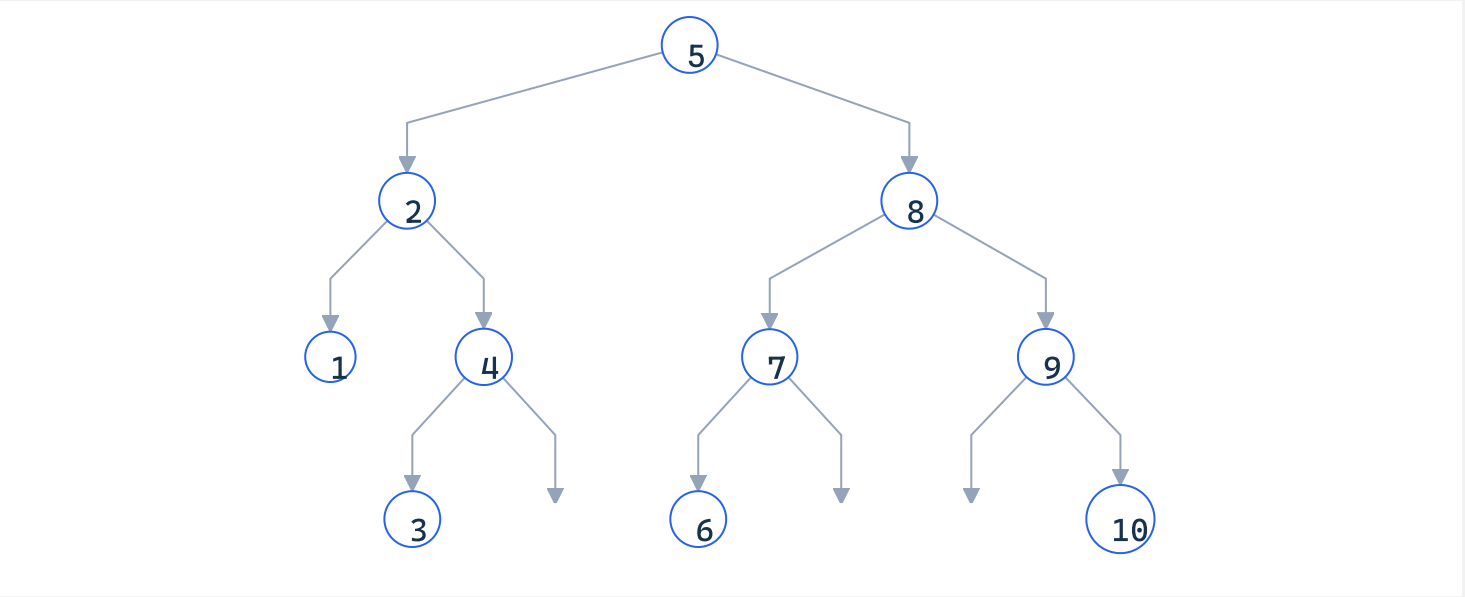
int Binary_Search(int target, vector<int> &arr) {
    int left = 0, right = static_cast<int>(arr.size()) - 1;
    while (left ≤ right) {
        int mid = left + (right - left) / 2; // 在非负下标区间中避免 left+right 溢出
        if (arr[mid] == target)
            return mid;
        else if (arr[mid] > target)
            right = mid - 1;
        else
            left = mid + 1;
    }
    return -1;
}
```

cpp

示例：

传入数组：1 2 3 4 5 6 7 8 9 10

可以把每次选择的中点画成下面的判定树；二分算法本身在数组上操作，并不实际构造这棵树。



平均查找长度 $ASL = \frac{\sum \text{查找路径长度}}{\text{节点数}} = \frac{1+2\times2+3\times4+4\times3}{10} = 2.9$

1.1.2 边界查找（lower_bound / upper_bound）

题目原文（自拟）： 给定升序整数 vector arr 与 target，返回第一个 ≥target 的下标 lower，以及第一个 >target 的下标 upper。若某种位置不存在，返回数组长度 n，空数组返回 0；两个函数均不直接读取返回位置。

思路分析： 使用半开区间 [left,right)。下界将“`arr[mid]` ≥target”作为真，否则在右侧查找；上界仅将“`arr[mid]` >target”作为真。真时令 right=mid 保留当前候选，假时令 left=mid+1 排除当前位置。目标条件沿升序数组从假到真，循环结束位置就是第一个真，也可能为 n。

样例与解释： arr=[1,2,2,2,4]、target=2 时 lower=1、upper=4，重复段是 [1,4)；target=3 时两者均为 4；target=5 时均为 5，不能读 `arr[5]`。每个函数 O(log(n+1)) 时间、O(1) 空间。

练习题： 洛谷 P2249 – 查找。

在 STL 的 `<algorithm>` 小节中使用的 `lower_bound` 与 `upper_bound`，就是用二分定位升序区间里的位置。

它们的原型是：

寻找第一个 ≥ target 的位置（lower_bound）



对 `[1,2,2,2,4]` 查 target=2，左边界每轮判断 `a[mid]≥2`，得到区间 `[0,5)→[0,2)→[0,1)→[1,1)`；右边界判断 `a[mid]>2`，得到 `[0,5)→[3,5)→[3,4)→[4,4)`。相等时两种判断保留的方向不同，最终等值位置为 `[1,4)`。返回 n 是合法的边界结果，但不是可读取的元素下标。

```cpp
#include <vector>
#include <algorithm>
using namespace std;

int lower_bound(const vector<int> &arr, int target) {
    int left = 0, right = arr.size(); // 注意右边界是 n
    while (left < right) {
        int mid = left + (right - left) / 2; // 防止整数溢出
        if (arr[mid] ≥ target)
            right = mid; // 收紧右边界，继续在左半部分查找
        else
            left = mid + 1; // 收紧左边界，在右半部分查找
    }
    return left; // 返回第一个 ≥ target 的下标
}
```

寻找第一个 > target 的位置（upper_bound）

```cpp
#include <vector>
```

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

int upper_bound(const vector<int> &arr, int target) {
    int left = 0, right = arr.size();
    while (left < right) {
        int mid = left + (right - left) / 2; // 防止整数溢出
        if (arr[mid] > target)
            right = mid; // 收紧右边界，继续在左半部分查找
        else
            left = mid + 1; // 收紧左边界，在右半部分查找
    }
    return left; // 返回第一个 > target 的下标
}
```

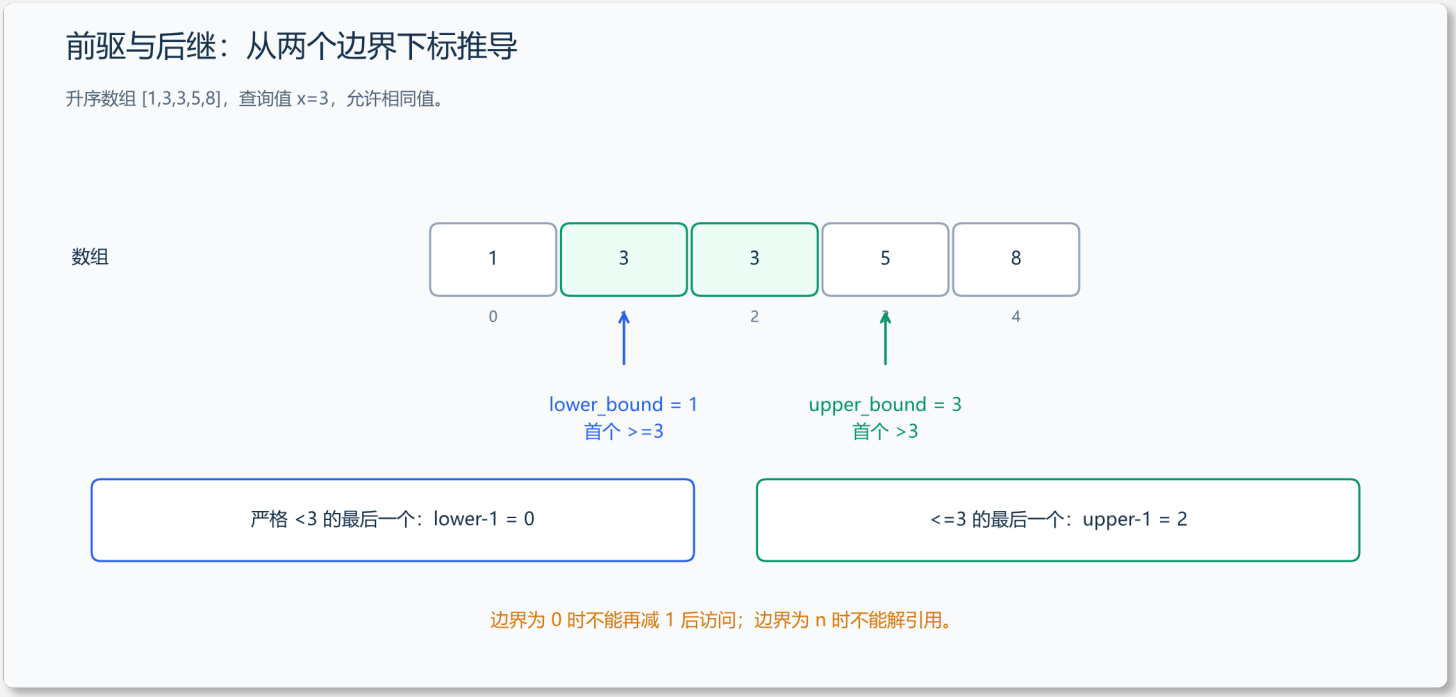
1.1.3 查找前驱与后继

题目原文（自拟）： 给定升序整数 vector arr 与 target。如果存在相等值，函数 bin_search 返回其中一个相等下标；若不存在，则返回最后一个小于 target 的位置；所有值都大于 target 或数组为空时返回 -1。遇到相等值立即返回，因此重复值中找到哪一项取决于二分访问次序；查找最后一个不大于 target 的位置时，使用前面的 upper_bound 再向前一项。

分析与样例： 除维护候选区间外，result 记目前见到的最大合法前驱下标。中点较小时记录它并向右查找，尝试更靠右的前驱；较大时向左。arr=[1,3,3,7]、target=4 时返回 2，target=0 时返回 -1，target=3 时可以返回 1 或 2。若严格要求“最后一个 ≤target”，应使用 upper_bound-1，而不能提前返回任意相等位置。时间 O(log(n+1))、空间 O(1)。

练习题： 洛谷 P1102 — A-B 数对（上下界统计等值段，区分重复下标数量）。

寻找目标值或目标值的前驱



以 `[1,3,3,7]` 为例，找 0 的严格前驱时，lower_bound 返回 0，左侧没有元素；找 8 的后继时，lower_bound 返回 4，它是数组之后的位置。代码用 -1 表示未找到，用 0…3 表示实际元素位置。

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

int bin_search(const vector<int> &arr, int target) {
    int left = 0, right = static_cast<int>(arr.size()) - 1;
    int result = -1; // 记录前驱位置

    while (left ≤ right) {
        int mid = left + (right - left) / 2;

        if (arr[mid] == target) {
            return mid; // 找到目标值，直接返回
        } else if (arr[mid] < target) {
            result = mid;   // 记录可能的前驱位置
            left = mid + 1; // 继续在右半部分查找目标值
        } else {
            right = mid - 1; // 在左半部分查找
        }
    }

    return result; // 返回前驱位置或-1
}
```

整数二分在离散下标或离散答案上搜索。首先确定区间约定和循环不变量，再选择中点与更新方式；不是只凭函数名字决定向上还是向下取整。

目标 / 写法	区间与中点	满足条件时	不满足时	无解表示
精确查找	闭区间，`mid = l + (r-l)/2`	相等直接返回	按大小更新 `l = mid+1` 或 `r = mid-1`	`-1`
第一个 `≥ x`	`[l,r)`，向下中点	`r = mid`	`l = mid+1`	返回 `n`
第一个 `> x`	`[l,r)`，向下中点	`r = mid`	`l = mid+1`	返回 `n`
最后一个可行点	闭区间，`mid = l + (r-l+1)/2`	`l = mid`	`r = mid-1`	需先保证至少一个可行点

这里的上取整写法要求 `r-l+1` 可表示。两端相邻时，`l = mid` 需要上中点避免停在原地；`r = mid` 的对称写法则使用下中点。`upper_bound` 返回第一个严格大于目标的位置，并不是“最右位置”。

不严格前驱（最后一个 `≤ x`）可以用 `upper_bound` 返回下标减一；严格前驱（最后一个 `< x`）用 `lower_bound` 返回下标减一；严格后继就是 `upper_bound`，返回 `n` 时表示不存在。

例题：

1.1.4 两数之和：二分查找互补值

题目原文（自拟示例）： 给定升序数组和目标和，返回两个不同下标；不存在返回 (-1, -1)。元素与目标在所用整数类型范围内。

思路分析： 双指针以 left < right 保证不同下标；二分版本只在 i 后面的区间找互补值。分别为 O(n) 和 O(n log n) 时间。

样例与解释： [1, 2, 4, 7]、目标 9，返回 (1, 3)；[3]、目标 6，返回 (-1, -1)。

给定一个有序数组和目标值 $target$，寻找数组中是否有两个数之和为目标值，找到则返回这两个数的下标

前面我们已经使用头尾双指针法解决了这个问题，现在我们尝试用整数二分



固定一个位置后，只在其右侧查找互补值，保证下标不同。

```cpp
#include <vector>
#include <utility>
using namespace std;

pair<int, int> twoSumBinary(const vector<int> &arr, int target) {
    int n = arr.size();
    for (int i = 0; i < n; i++) {
        long long need = 1LL * target - arr[i]; // 计算需要的互补值

        // 在 (i+1 ... n-1) 范围内二分查找 need
        int left = i + 1, right = n - 1;
        while (left ≤ right) {
            int mid = left + (right - left) / 2; // 防止整数溢出
            if (arr[mid] == need)
                return {i, mid}; // 找到匹配对
            else if (arr[mid] < need)
                left = mid + 1; // 在右半部分继续查找
            else
                right = mid - 1; // 在左半部分继续查找
        }
    }
    return {-1, -1}; // 无解
```

```
23  }                                                                    cpp
```

可以看到，时间复杂度相较于双指针法的 $O(n)$ 还是慢了些，但是逻辑也比较清晰

1.2 实数二分

实数二分用于在 **连续的实数区间** 上求解问题，例如求方程的根、函数的零点等。常见于：

- 求开方、开立方
- 求最短时间、最小距离
- 几何问题（切蛋糕、切绳子）
- 求方程的根

与整数二分处理的“离散化数据”不同，实数二分面对的是数学意义上的连续区间。在连续区间中不存在跳点，因此 **mid 不需要取整，也不需要考虑向上或向下的问题**，直接使用：`double mid = (left + right) / 2.0` 即可。

不过需要注意：**计算机并不具有真正的连续性**。
`double` 和 `float` 只能表示有限精度的浮点数，因此实数二分本质上是在一个“有限精度的浮点集合”内不断逼近答案，而不可能无限细分。

因此，实数二分的停止条件不是“区间缩为一个点”，而是：

- 区间长度达到给定精度：`right - left < eps`
- 或者迭代次数足够多：固定迭代次数，例如 100 次；次数应结合初始区间和所需误差选择

浮点舍入可能让 `mid` 等于端点，区间不再缩小。使用精度条件时，应加最大迭代次数或检查是否停滞；固定次数也不能突破浮点表示精度。

实数二分的基本形式如下：

```cpp
// 伪代码：保存左右端点，用 check(mid) 决定保留哪一半。
/*
 * 实数二分模板
 *
 * 该模板用于在浮点数范围内进行二分查找，解决连续函数求根、最优值查找等问题。
 * 通过设置精度阈值或固定迭代次数来控制搜索精度，避免无限循环。
 *
 * 参数 left: 搜索区间左边界
 * 参数 right: 搜索区间右边界
 * 参数 check: 判断函数，确定mid满足条件的哪一侧
 *
 * 注意: 时间复杂度: O(log(n))，其中n = (right-left)/eps
 * 注意: 空间复杂度: O(1)，仅使用常数级别的额外空间
 * 注意: 精度控制: 使用eps控制精度或固定迭代次数保证收敛
 * 注意: 应用场景: 连续函数求根、最优值搜索、数值计算等
 * 注意: 与整数二分区别: 边界处理不同，直接赋值mid而非mid±1
```

```
 */
const double eps = 1e-7;                    // 精度控制，根据问题要求调整
for (int iter = 0; iter < 200 && right - left > eps; ++iter) {                  // 或者使用
for(int i = 0; i < 100; ++i)
    double mid = left / 2 + right / 2; // 计算中点，并避免异号大端点的差溢出
    if (mid == left || mid == right) break; // 浮点中点不再变化时停止。
    if (check(mid))
        right = mid; // mid满足条件，在左半部分继续搜索
    else
        left = mid; // mid不满足条件，在右半部分继续搜索
}
```
text

> **注意：** 理想实数计算中，迭代 k 次后区间长度为初始长度的 2^{-k}。这不是浮点答案实际具有同样精度的保证。停止条件要同时考虑区间尺度、误差要求和浮点舍入。

使用示例：

1.2.1 求平方根

题目原文（自拟示例）： 给定有限非负 double x，返回其平方根的浮点近似值；负数或非有限输入抛出异常。

思路分析： 在覆盖答案的非负区间二分，用 mid > x/mid 代替平方比较来避免溢出。有限次迭代及中点停滞检查保证结束，实际精度受浮点表示限制。

样例与解释： x=0 返回 0；x=2 约为 1.4142135623730951；x=0.25 约为 0.5。



以 x=2 为例，在非负区间比较 mid 与 x/mid，保留包含平方根的一侧；用除法避免 mid*mid 溢出。

```
#include <algorithm>
#include <cmath>
#include <stdexcept>
using namespace std;

```

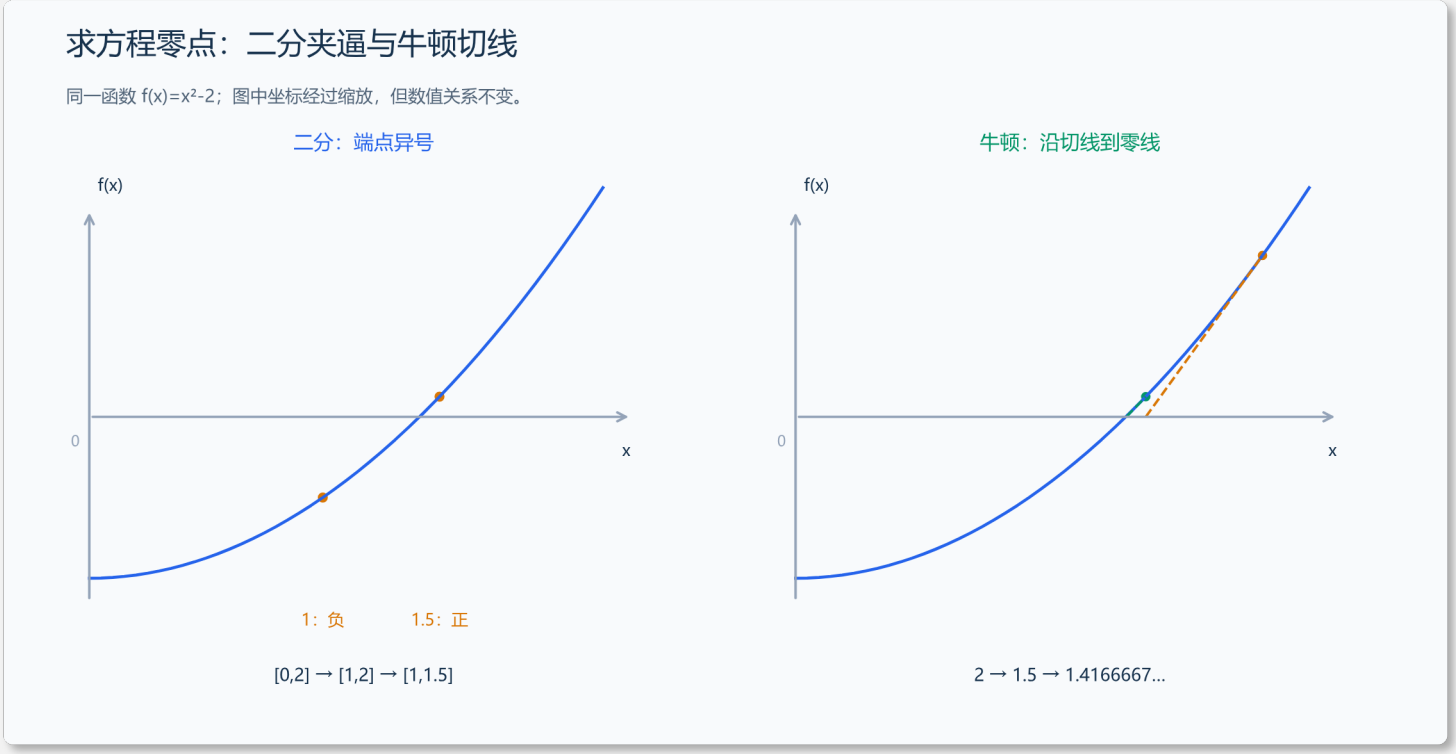
```cpp
double sqrt_binary(double x) {
    if (x < 0 || !isfinite(x))
        throw invalid_argument("x 必须是有限非负数");
    if (x == 0)
        return 0;
    double left = 0, right = max(1.0, x);
    for (int i = 0; i < 1100; ++i) {
        double mid = left + (right - left) / 2;
        if (mid == left || mid == right)
            break;
        if (mid > x / mid)
            right = mid; // 避免 mid*mid 溢出
        else
            left = mid;
    }
    return left;
}
```

1.2.2 二分法求方程的根

题目原文（自拟示例）： 给定有限区间 [a,b] 和连续函数 f，要求端点已为零或两端函数值异号，返回区间内一个根的近似值。计算过程中出现非有限函数值时抛出异常。

思路分析： 连续曲线的一端在横轴下方、一端在横轴上方，中间就会经过横轴。计算中点后，保留两端仍在横轴两侧的那一半，零点就在这个越来越短的区间中。

样例与解释： f(x)=x²−2、区间 [0,2]，返回约 1.4142135623730951；f(x)=x、区间 [0,1]，直接返回端点 0。



左图让零点留在两个端点之间，再不断缩短区间；右图画当前点的切线，取它与横轴的交点作为下一次位置。下文分

别手算这两种更新。

接口衔接： 这里的 `function<double(double)> f` 用前面的 std::function 章节同样的读法：传入一个 double，得到一个 double；调用 `f(mid)` 交给调用者提供的公式。`<cmath>` 的 isfinite 检查结果是否有限，后面的 fabs 求浮点绝对值；`<stdexcept>` 的 runtime_error 表示运行过程中未能满足条件，与 invalid_argument 的参数错误类别不同，均可通过异常对象的 what 获取说明。

```cpp
#include <functional>
#include <cmath>
#include <stdexcept>
using namespace std;

double solve_root_binary(double a, double b, function<double(double)> f) {
    if (!isfinite(a) || !isfinite(b) || a > b)
        throw invalid_argument("区间无效");
    // 先处理端点就是零点的情况，再检查夹逼条件。
    double fa = f(a), fb = f(b);
    if (!isfinite(fa) || !isfinite(fb))
        throw invalid_argument("函数值必须有限");
    if (fa == 0)
        return a;
    if (fb == 0)
        return b;
    if ((fa < 0) == (fb < 0))
        throw invalid_argument("端点须异号或已经为零");
    for (int i = 0; i < 100; ++i) {
        double mid = a / 2 + b / 2;
        if (mid == a || mid == b)
            break;
        double fm = f(mid);
        if (!isfinite(fm))
            throw invalid_argument("函数值必须有限");
        if (fm == 0)
            return mid;
        // 中点与左端同号时舍弃左半段，否则舍弃右半段。
        if ((fm < 0) == (fa < 0)) {
            a = mid;
            fa = fm;
        } else
            b = mid;
    }
    return a / 2 + b / 2;
}
```

二分代码解释： 端点为零时直接返回；否则检查异号条件。中点与左端同号时，左端移动到中点；异号时，右端移动到中点。这样每次仍保留含有零点的夹逼区间。连续曲线从负值连到正值时会经过 0，这就是介值定理的直观含义。曲线即使途中上下起伏，我们仍能沿符号变化找到其中一个零点。

固定 100 次是本实现的迭代上限。若把允许误差记为 `eps`、初始宽度记为 `L`，理论上需约 $\log_2(L/eps)$ 轮，每轮另需一次函数求值。浮点数能表示的位置有限，中点与端点相同时，代码结束迭代；此时继续计算同一个中点也不会让区间再缩小。函数对象本身的求值时间和存储也要另计。

牛顿迭代的推导： 当前点为 `(x,f(x))`，切线方程为 $y = f(x) + f'(x)(t - x)$。令 `y=0`，得到下一次估计 $t = x - f(x)/f'(x)$。这一步使用局部切线逼近曲线；它不像二分那样始终持有夹逼区间，所以初值和导数会影响结果。

本节仍以 `f(x)=x²-2` 为例，导数为 2x。从 x=2 开始，先得到 1.5，再得到约 1.4166667，逐渐接近 sqrt(2)。代码先计算 f(x)：它与 0 的差已不超过 eps 时返回 x；否则计算导数和下一个点。若从 x=0 开始，导数 2x 为 0，代码在执行除法前抛出异常。

代码示例：

```cpp
#include <functional>
#include <cmath>
#include <stdexcept>
using namespace std;

double newton_method(double x0, function<double(double)> f, function<double(double)> df,
                     double eps = 1e-7, int max_iter = 100) {
    if (!isfinite(x0) || !isfinite(eps) || eps ≤ 0 || max_iter ≤ 0)
        throw invalid_argument("初值、误差或迭代次数无效");
    double x = x0;
    for (int i = 0; i < max_iter; ++i) {
        double fx = f(x);
        if (!isfinite(fx))
            throw runtime_error("函数值不有限");
        if (fabs(fx) ≤ eps)
            return x; // 返回前检查残差，而非只检查步长。
        double dfx = df(x);
        if (!isfinite(dfx) || dfx == 0)
            throw runtime_error("导数无效或为零");
        double next = x - fx / dfx;
        if (!isfinite(next) || next == x)
            throw runtime_error("迭代溢出或停滞");
        x = next;
    }
    if (isfinite(f(x)) && fabs(f(x)) ≤ eps)
        return x;
    throw runtime_error("达到迭代上限但残差仍超出阈值");
}
```

每轮用当前 x 计算函数值与切线。上面的程序以 `|f(x)|≤eps` 表示函数值已足够接近 0；导数为零、计算结果非有限、迭代位置不再变化或次数用完时，走到对应的异常分支。它比较的是函数值与 0 的差；若要衡量位置与真实根的距离，还可以在这个平方根例子中计算 `|x-sqrt(2)|`。

两种方法对比：

项目	牛顿迭代法 Newton	二分法 Binary Search
怎样推进	计算当前点的切线，用切线与横轴的交点得到下一点	在横轴两侧取端点，计算中点并保留异号的一半
是否需要导数	需要 f'(x)	不需要
从哪里开始	本例从 x=2 开始画第一条切线	本例从 [0,2] 开始，f(0)<0、f(2)>0
每轮的变化	本例为 2→1.5→1.4166667→1.4142157	本例区间宽度为 2→1→0.5→0.25
保留哪些信息	保存当前点，重新计算函数值与导数	保存两端，使零点继续留在中间
代码复杂度	中等：需要传入导数函数	简单：只有区间取 mid
适合问题类型	可计算导数的方程零点近似	连续函数的零点夹逼；整数二分另见单调判定问题
每次迭代成本	计算 f 和 f'，成本由具体函数决定	计算一次 f(mid)，成本由具体函数决定
后续联系	用导数确定更新方向；还可学习求极值的迭代	用判断舍弃一半；整数二分与二分答案也沿这个思路
怎样结束	函数值足够接近 0 时返回；程序另外处理无法继续迭代的分支	按迭代次数缩短区间，中点与端点相同时结束

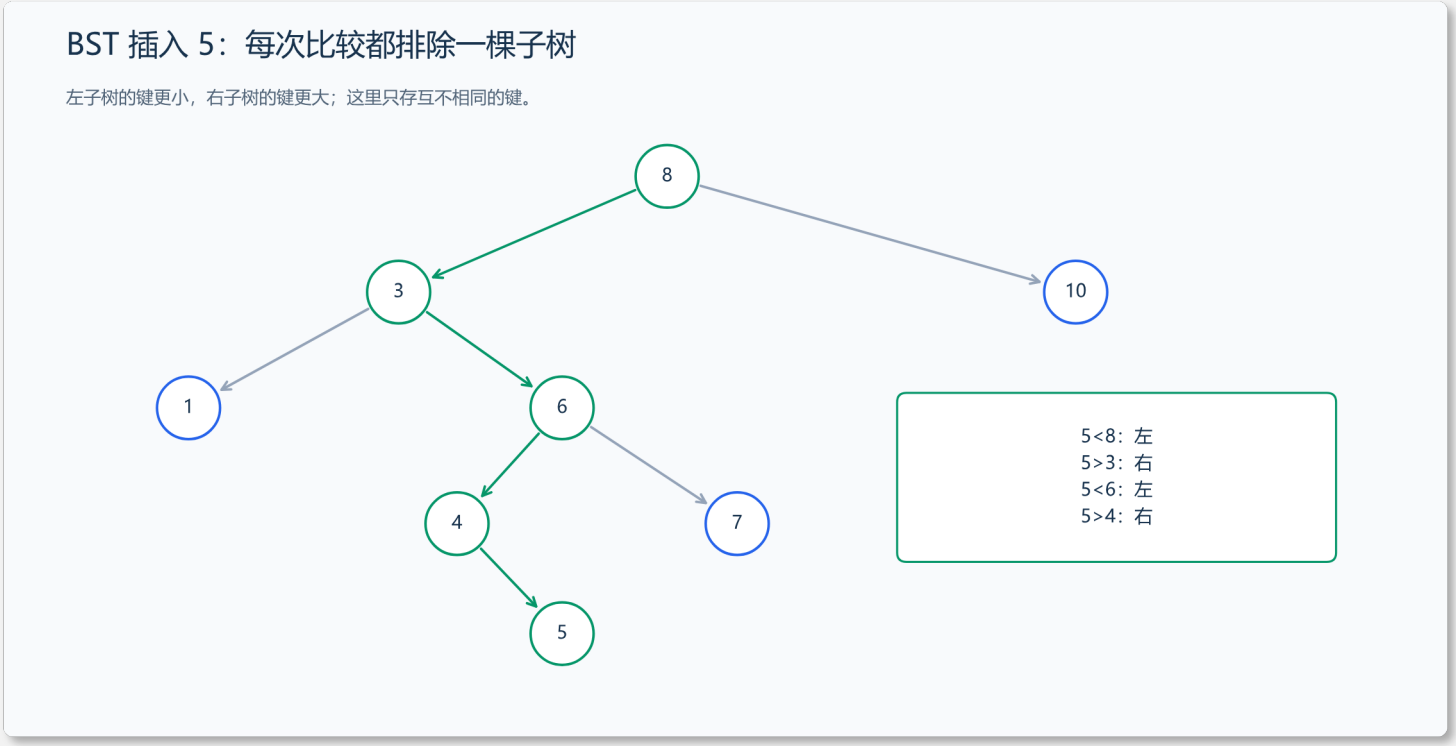
1.2.3 与整数二分的对比

特性	整数二分	实数二分
数据结构	数组 / 离散值	连续区间
终止条件	`left > right` 或 `left == right`	区间长度 < 精度
mid 的计算	整数	double/long double
判断依据	比较中间值，或观察判定结果的真假变化	比较函数值，或观察函数值的正负变化
场景	查找值、边界	求根、求最优值、几何问题

> **注意：** 根据二分思想和二叉搜索树的性质，我们可以写出二叉搜索树的插入算法

下列 `insert_BST` 是类内成员片段，需放入前面 `BinaryTree` 的 `public` 区域，并从空树开始使用 BST 插入；不能与层序插入或普通树删除混用。

首先，二叉搜索树的性质：左子树的所有节点值 < 当前节点值 < 右子树所有节点值。



插入 5 时依次与 8、3、6、4 比较，最后放到 4 的右孩子。普通 BST 的代价取决于树高。

```cpp
bool insert_BST(int value) {
    // 处理空树情况：直接创建根节点
    if (!root) {
        root = new TreeNode(value);
        return true;
    }

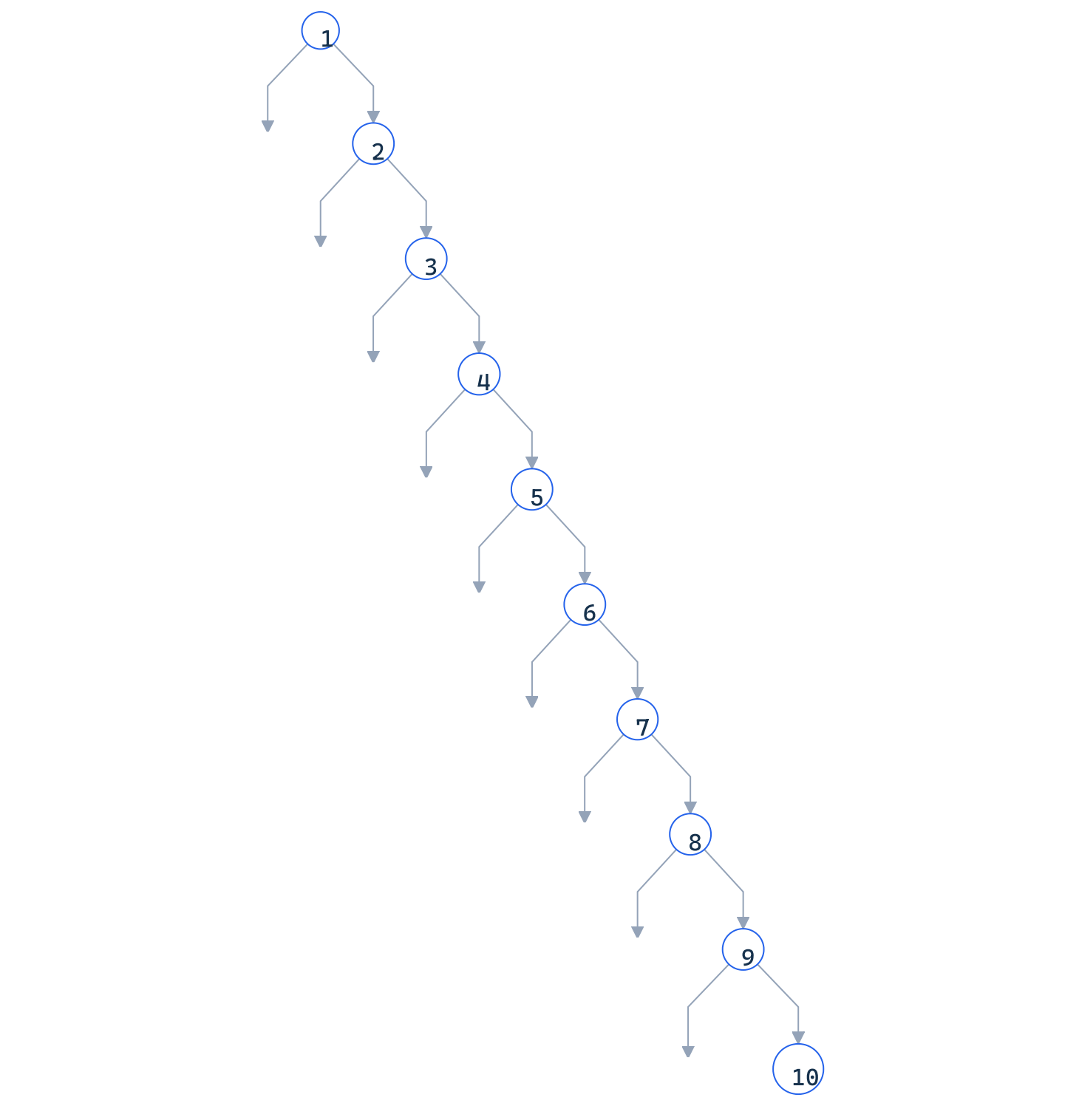
    TreeNode *cur = root;
    // 迭代查找合适的插入位置
    while (true) {
        if (value < cur->val) {
            // 值小于当前节点，向左子树查找
            if (cur->left)
                cur = cur->left; // 左子树非空，继续向下查找
            else {
                cur->left = new TreeNode(value); // 找到空位置，插入新节点
                return true;
            }
        } else if (value == cur->val) {
            return false; // 本笔记的 BST 不存重复值
        } else {
            // 值大于当前节点，向右子树查找
            if (cur->right)
                cur = cur->right; // 右子树非空，继续向下查找
            else {
                cur->right = new TreeNode(value); // 找到空位置，插入新节点
                return true;
            }
        }
    }
}
```

当然，因为这是普通 BST，而不是 AVL 树或者 Treap 树，输入严格有序的话，该二叉树会退化（若升序则退化为只有右子树的链表，降序则退化为只有左子树的链表）

例如：

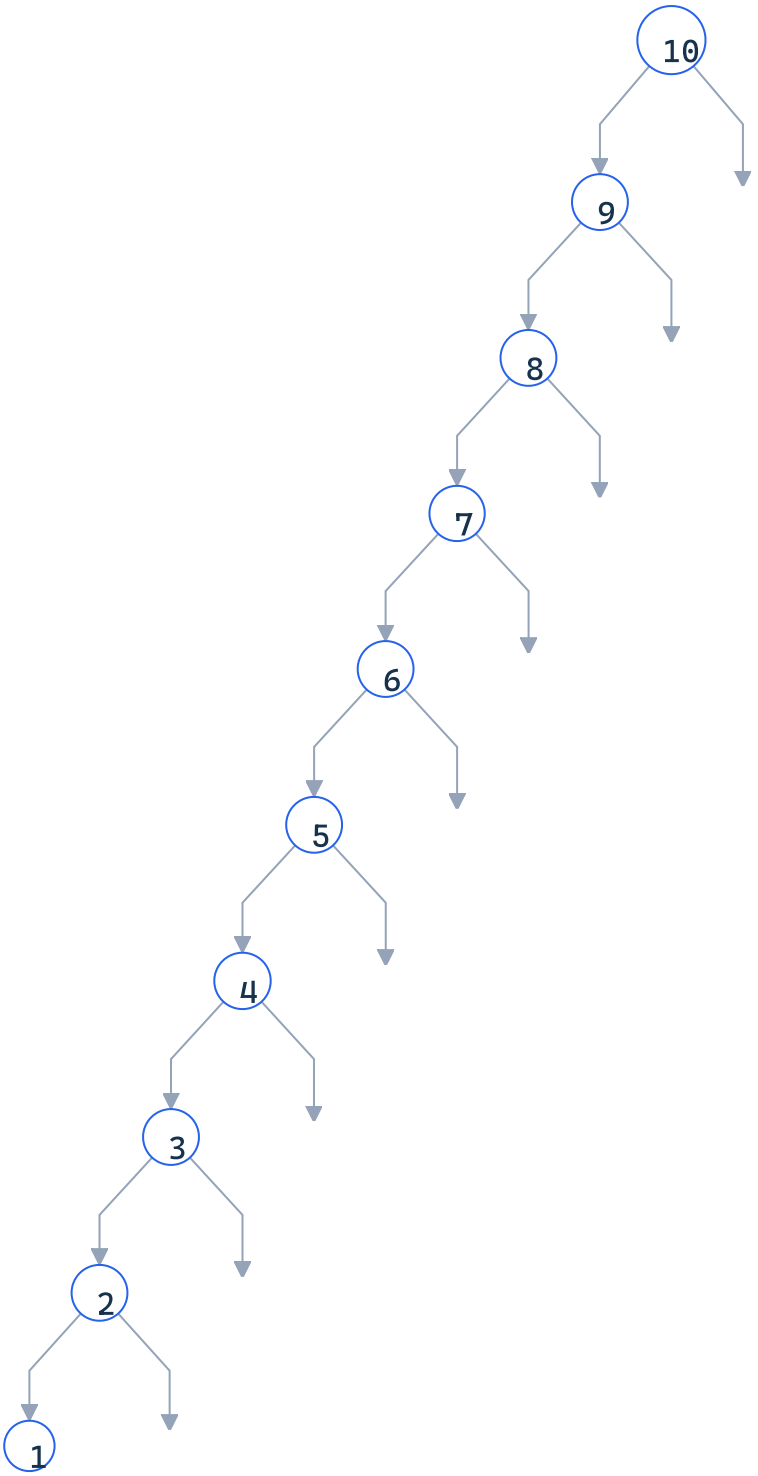
```text
输入:
1 2 3 4 5 6 7 8 9 10
```

构建的二叉树为：



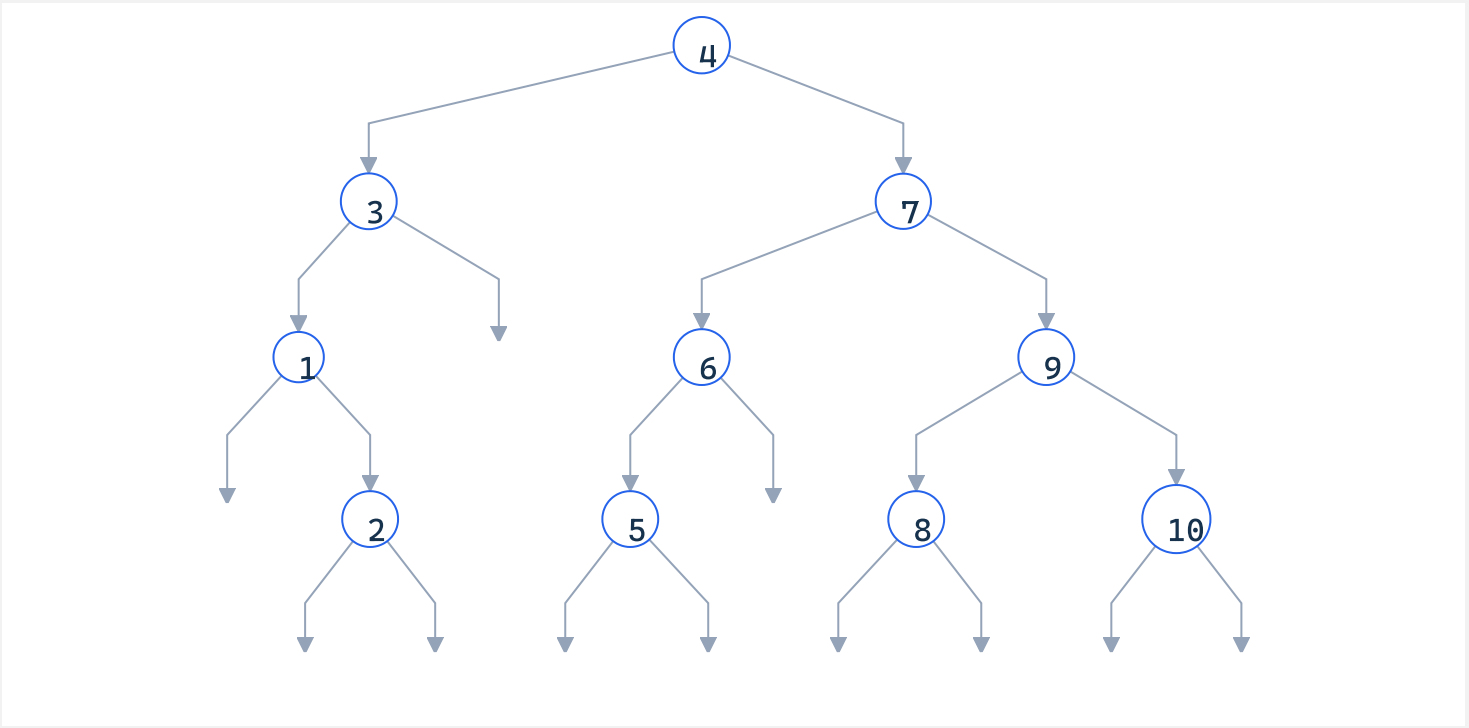
```text
输入:
10 9 8 7 6 5 4 3 2 1
```

构建的二叉树为:




```text
输入
4 7 9 6 5 3 1 8 2 10
```

构建如下二叉树:



1.3 二分答案

二分答案是一种 **解题策略**，它本身使用的是二分查找的技术。当一个问题直接求解很困难，但我们可以 **轻易地判断一个候选答案 `x` 是“偏大”还是“偏小”** 时，就可以使用二分答案。

适用问题特征：

- 问题的解在某个确定的范围内 `[min, max]`。
- 这个范围内的值相对于问题的条件存在单调性。例如，如果 `x` 可行，那么所有 `≤ x`（或 `≥ x`）的值也可能可行。
- 存在一个 `check(x)` 函数，可以高效地判断给定的 `x` 是否是一个可行的解。

常见题型：

- 最小化最大值 / 最大化最小值
- Koko 吃香蕉
- 分配工作、分配书本
- 木棒切割、绳子切割
- 最小花费 / 最大利润可行性
- 图论中的最小可行路径（比如最大边最小、最小边最大）

示例：

1.3.1 最小吃香蕉速度（Koko，LeetCode 875）

接口与范围： 给定非空正整数堆数组 piles 和小时 h，保证 h 至少等于堆数。每小时只能从一堆吃至多 k 根，求按时吃完的最小正整数 k。

思路分析： 用 ceil(pile/k) 统计小时，速度越大越容易完成。二分最小可行速度，时间 O(n log maxPile)。

样例与解释： piles=[3,6,7,11]、h=8，返回 4；各堆用时 1、2、2、3 小时。

珂珂喜欢吃香蕉。这里有 `n` 堆香蕉，第 `i` 堆中有 `piles[i]` 根香蕉。警卫已经离开了，将在 `h` 小时后回来。

珂珂可以决定她吃香蕉的速度 `k` （单位：根/小时）。每个小时，她将会选择一堆香蕉，从中吃掉 `k` 根。如果这堆香蕉少于 `k` 根，她将吃掉这堆的所有香蕉，然后这一小时内不会再吃更多的香蕉。

珂珂喜欢慢慢吃，但仍然想在警卫回来前吃掉所有的香蕉。

返回她可以在 `h` 小时内吃掉所有香蕉的最小速度 `k`（`k` 为整数）。

题目分析：对于一个给定的速度 `k`，我们可以计算吃完所有香蕉需要多少小时。如果所需小时数 `≤ h`，则速度 `k` 是可行的；否则速度太慢。

判断 k 是否可行的方法：

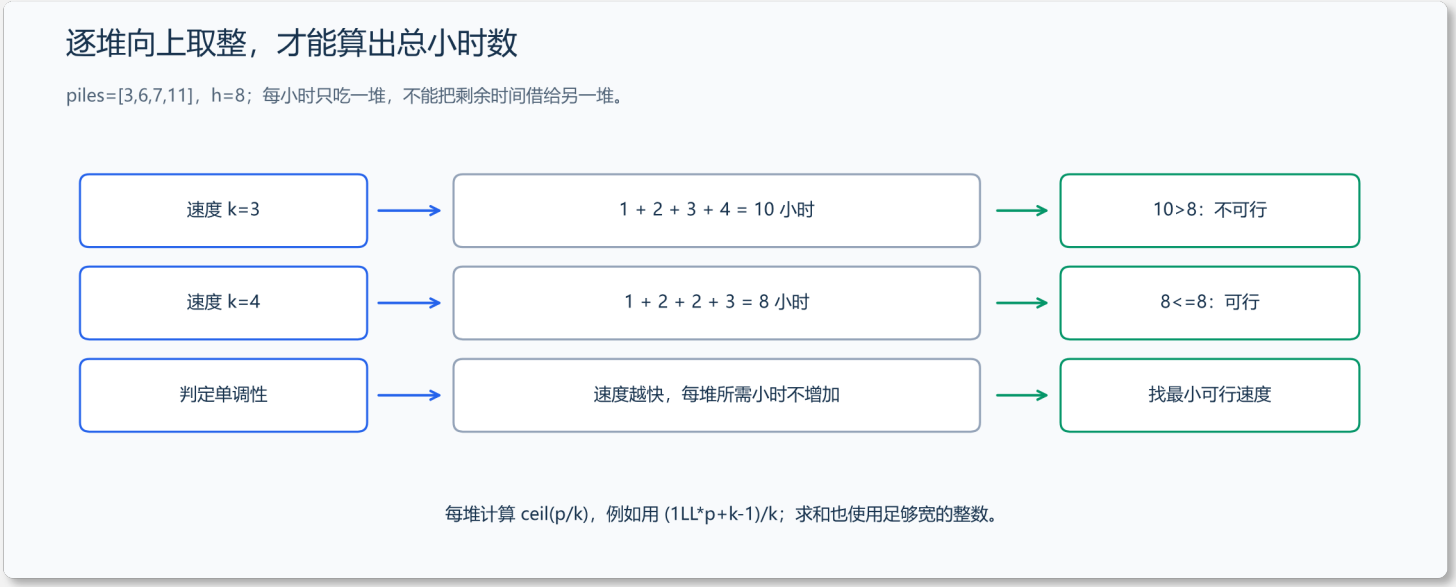
对每一堆香蕉 `piles[i]`：

吃完这一堆需要的小时数为：$hours = \lceil piles[i]/k \rceil$

因为每小时只能吃一个堆，剩余不足 k 时，这个堆仍占用当前一小时；下一小时再吃另一个堆。

将所有堆的小时数加和：$total_hours = \sum \lceil piles[i]/k \rceil$

题解：



每堆分别占完整小时；速度 3 共需 10 小时，不满足 h=8，速度 4 共需 8 小时，恰好可行。



对 `[3,6,7,11]`、h=8，初始速度范围 `[1,11]`。中点 6 需要 6 小时，可行，保留 `[1,6]`；中点 3 需要 10 小时，失败，排除 1…3；随后 5、4 都恰好需要 8 小时，依次收缩到 `[4,4]`。所求是第一个可行速度，失败时移到 `mid+1`，成功时仍保留 mid。

```cpp
#include <vector>
#include <algorithm>
using namespace std;

using ll = long long;

bool check(const vector<int> &piles, int k, int h) {
    ll hours = 0; // 用 long long 防止大数相加溢出
    for (int p : piles)
        hours += (1LL * p + k - 1) / k; // 等价于 ceil(p / k)，耗时累加
    return hours ≤ h;                   // 若总时间不超过 h，则速度可行
}
```

```cpp

int minEatingSpeed(vector<int> &piles, int h) {
    int left = 1, right = *max_element(piles.begin(), piles.end());
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (check(piles, mid, h))
            right = mid; // mid 足够快，尝试更小的速度
        else
            left = mid + 1; // mid 对应的耗时过长，继续查找更高速度
    }
    return left; // 返回最小可行速度
}
```

1.3.2 木材切割：寻找最长可行长度

题目原文（自拟教学题）： 函数输入非空、有限非负木条长度 vector a 和整数 k≥1，允许舍弃余料但不允许拼接，求能切出至少 k 段等长木条的最大长度；全为 0 时返回 0。返回近似 double，按调用者输出精度格式化。

接口与范围： 给定非空有限非负木条长度数组和 k≥1，切出至少 k 段等长木条，允许丢弃余料，求最大长度；所有长度为 0 时返回 0。

思路分析： 试探长度越大，可切段数越少；floor(length/guess) 累加达到 k 就停止，以避免计数溢出。二分返回近似值。

样例与解释： 长度 [5,8]、k=3，答案为 4，可从两根分别切出 1、2 段。

给定若干木棒，它们的长度分别为 `a[1] , a[2] , ... , a[n]`，需要从中切出 **至少** `k` **根** 等长的小木棒。

每次切割不会损失木材（也就是不能粘合，只能切割）。

问：**能够切出的所有小木棒长度的最大值是多少？（保留两位小数）**

题目分析：

对于一个给定测试长度 len，则可切出数量为 $total = \sum_{i=1}^{n} \lfloor \frac{a[i]}{len} \rfloor$

如果 $total >= k$ 说明 len 可行，可以尝试更长长度，否则太长，需要缩短

由于题目要求保留两位小数，因此这是个实数二分的二分答案

题解：



长度 4 时，[5,8] 分别提供 1、2 段，共 3 段；长度 5 时只剩 2 段，不能拼接余料。



对 `[5,8]`、k=3，从 `[0,8]` 开始。长度 4 提供 3 段，左端升到 4；长度 6、5、4.5 都只有 2 段，右端依次下降。每次只使用各根木材的完整段，余料不能跨木材拼接。程序继续固定轮数，最终返回可行一侧的 left，图中前几轮展示的就是这个更新规则。

```cpp
#include <vector>
#include <algorithm>
#include <cmath>
#include <stdexcept>
using namespace std;

using ll = long long;

bool check(const vector<double> &a, double len, int k) {
    ll cnt = 0;
    for (double x : a)
        if (len ≤ 0)
```

```cpp
            return false;
        else if (floor(x / len) ≥ k - cnt)
            return true;
        else
            cnt += static_cast<ll>(floor(x / len));
    return cnt ≥ k; // 总段数是否满足要求
}

double maxCutLength(const vector<double> &a, int k) {
    if (a.empty() || k ≤ 0)
        throw invalid_argument("木棒非空且 k > 0");
    double left = 0, right = *max_element(a.begin(), a.end());
    for (int i = 0; i < 100; ++i) {
        double mid = left + (right - left) / 2;
        if (mid == left || mid == right)
            break;
        long long count = 0;
        for (double x : a) {
            double pieces = floor(x / mid);
            if (pieces ≥ k - count) {
                count = k;
                break;
            }
            count += static_cast<long long>(pieces);
        }
        if (count ≥ k)
            left = mid;
        else
            right = mid;
    }
    return left;
}
```
cpp

1.3.3 最小化最大值：寻找第一个可行上限

定义：在一个集合或可行域内，找到一个点或策略，使得在所有可能的最坏情况（最大值）中，这个最坏情况是最好的（最小化）。

表达式： $\min_{x \in X} (\max_{y \in Y} f(x, y))$

X 是决策变量 x 的可行集（我们能控制的变量）。

Y 是对 x 产生影响的干扰变量 y 的可行集（我们不能控制的变量，代表最坏情况）。

$f(x, y)$ 是目标函数。

在二分法下：

目标： $\min(\max\{\text{集合中的值}\})$

单调性： **如果存在一个方案使得最大值 $\leq K$，那么对于任何 $K' > K$，也一定存在方案使得最大值 $\leq K'$。**

- 我们便可以用二分法来搜索这个最小的 K。

分摊工作的目标是最小化最大工作量；上面的木材切割则是在可行长度中求最大值，应寻找最后一个可行点。

示例：

1.3.4 任务分段：最小化最大工作量

题目原文（自拟教学题）： 输入非空正整数工作量数组 `W` 和 `M≥1`，将任务分为不超过 `M` 个连续非空段，最小化所有段和的最大值；返回 `long long`，所有总和可表示。

接口与范围： 给定非空、正整数工作量数组 `W` 和 `M≥1`，把它分成不超过 `M` 个连续非空段，求最大段和的最小值；总和在 `long long` 范围内。

思路分析： 猜一个上限，依次装入当前段，超过就另开一段；所需段数不超过 `M` 即可行。上限越大越容易可行，二分最小可行值。

样例与解释： `W=[7,2,5,10,8]`、`M=2`，返回 `18`，对应 `[7,2,5]` 与 `[10,8]`。

有 N 个任务，每个任务的工作量为 W_i。现在要将这些任务分配给 M 个人，要求分配给每个人的任务是 **连续** 的一段。目标是最小化 M 个人中工作量 **最大** 的那个人所承担的总工作量。

问题分析：

如果我们设一个最大允许工作量 `X`，那么我们能否按照不超过 `X` 的原则把任务分给 **不多于 `M` 个人**？

只要判断这个 `X` 是否可行，就能对答案进行 **二分搜索**。

题解：



图中的 17、18 分别位于不可行与可行一侧；二分要定位它们之间的边界。



对 `[7,2,5,10,8]`、M=2，范围先取 `[10,32]`。上限 21 分两组，可行；上限 15 分三组，不可行；上限 18 分两组，可行；上限 17 分三组，不可行，最后收缩到 18。图中的分组由从左向右累计、放不下才开新组产生，不能任意把非连续任务挪到同一组。

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

using ll = long long;
bool check(const vector<int> &W, ll X, int M) {
    ll load = 0; // 当前工人的累计工作量
    int cnt = 1; // 已使用的工人数量（至少1个）
    for (int w : W) {
        if (w > X)
            return false; // 单个任务超过 X，直接不可能
        if (load + w > X) {
            cnt++;     // 新开一个工人
            load = w; // 新工人从当前任务开始
        } else
            load += w; // 当前工人继续承担任务
    }
    return cnt ≤ M; // 使用的工人数是否在限制内
}

ll minimizeMaxWorkload(vector<int> &W, int M) {
    ll left = 0, right = 0;
    for (int w : W) {
        if (w > left) left = w; // 上限至少要容纳最大的单个任务。
        right += w;             // 全部任务放一组，得到可行上界。
    }

    while (left < right) {
        ll mid = left + (right - left) / 2;
```

```cpp
        if (check(W, mid, M))
            right = mid; // mid 可行，尝试更小的上限
        else
            left = mid + 1; // mid 对应的方案不可行，继续尝试更大上限
    }
    return left; // 返回最小可行最大工作量
}
```

1.3.5 最大化最小值：寻找最后一个可行下限

在一个集合或可行域内，找到一个点或策略，使得在所有可能的最优情况（最小值）中，这个最优情况是最好的（最大化）。（）找到一个策略，使得在所有可能的 **最差结果** 中，这个最差结果是 **最好的**。

数学表达式： $\max_{x \in X} \left(\min_{y \in Y} f(x, y) \right)$

在二分答案下：

目标：$\max(\min\{\text{集合中的值}\})$

单调性： 如果存在一个方案使得最小值 $\geq K$，那么对于任何 $K' < K$，也一定存在方案使得最小值 $\geq K'$。

- 我们便可以用二分法来搜索这个最大的 K。

1.3.6 进击的奶牛：最大化最小间距

教学接口题面（自拟）： 给定至少两个牛舍的整数坐标，范围 [0,10⁹]，以及 2≤C≤牛舍数。每个牛舍最多放一头牛，放入 C 头，求最近两头间距的最大可能值；返回整数。坐标数组可被函数排序。

接口与范围： 给定至少两个牛舍坐标，2≤C≤牛舍数，每个位置在 [0,1000000000]。每个牛舍至多放一头牛，求最大化最近两头牛距离的整数答案。

思路分析： 给定间距后总是选择最靠左的可行牛舍；可行间距呈先 true 后 false，二分最后的 true。排序 O(n log n)，二分检查 O(n log D)。

样例与解释： 位置 [1,2,8,4,9]、C=3，返回 3，可选 1、4、8。

农夫约翰建造了一座有 n 间牛舍的小屋，牛舍排在一条直线上，第 i 间牛舍在 x_i 的位置，但是约翰的 m 头牛对小屋很不满意，因此经常互相攻击。约翰为了防止牛之间互相伤害，因此决定把每头牛都放在离其它牛尽可能远的牛舍。也就是要最大化最近的两头牛之间的距离。

牛们并不喜欢这种布局，而且几头牛放在一个隔间里，它们就要发生争斗。为了不让牛互相伤害。约翰决定自己给牛分配隔间，使任意两头牛之间的最小距离尽可能的大，那么，这个最大的最小距离是多少呢？

题目分析：

如果我们猜一个最小距离 `d`，要判断能否把 `C` 头奶牛放置下去：

- 第一头奶牛放在最左的牛棚；

- 后续每次尽量放在距离上一次 ≥ `d` 的最靠前的牛棚；
- 数到能否放满 `C` 头。

若能放下，则说明距离 `d` 可行，可以再增大；
否则 `d` 太大，需要减小。因此可以使用向上取整的整数二分的二分答案

题解：



排序位置为 `[1,2,4,8,9]`；距离 `3` 可在 `1`、`4`、`8` 放三头，距离 `4` 只能放两头。



距离 `3` 时先放位置 `1`；位置 `2` 与上一次间隔为 `1`，跳过；位置 `4` 间隔为 `3`，可以放；位置 `8` 间隔为 `4`，再放一头便达到 `C=3`，可以返回成功。尽早放置不会挤占右侧空间：将某个方案的下一头左移到最早合法牛舍，前一段间隔仍够，后一段间隔只会增大。



初始距离范围 `[0,8]`，取向上中点 4，只能放在 1、8 两处，失败，保留 `[0,3]`；中点 2、3 都能在 1、4、8 放三头，左端依次升高，最后为 3。这里寻找最后一个可行值：成功就把左端移到 mid，失败就把右端移到 mid-1；前面的最小速度题寻找第一个可行值，成功时缩右端，两种目标对应不同的移动方向。

```cpp
#include <vector>
#include <algorithm>
using namespace std;

bool check(const vector<int> &x, int d, int C) {
    int cnt = 1;     // 已安置奶牛数量（第一头在x[0]）
    int last = x[0]; // 最后一头奶牛的安置位置
    for (int i = 1; i < x.size(); i++) {
        if (x[i] - last ≥ d) { // 满足最小间距要求
            cnt++;
            last = x[i]; // 更新最后安置位置
            if (cnt ≥ C)
                return true; // 已安置足够数量，提前返回
        }
    }
    return false; // 无法安置所有奶牛
}

int maxMinDistance(vector<int> &x, int C) {
    sort(x.begin(), x.end());                    // 预处理：排序牛栏位置
    int left = 0, right = x.back() - x.front(); // 最大可能间距

    while (left < right) {
        int mid = left + (right - left) / 2 + (right - left) % 2; // 取上中位数避免死循环
        if (check(x, mid, C))
            left = mid; // mid 可行，尝试更大的间距
        else
            right = mid - 1; // mid 不可行，缩小间距
    }
    return left; // 返回最大可行最小间距
}
```

1.3.7 分派：最大化每人的切片面积（POJ 3122）

接口与范围： 给定非空非负整数半径数组 `radii` 和朋友数 `F≥0`。主人也分一份，总人数为 `F+1`；每人得到一块面积相等、来自同一个派的切片，允许余料，求最大切片面积。半径转换成面积后，再按总人数计算每份面积。

思路分析： 用 floor(πr²/area) 判断能否凑够人数，人数达到目标就停止。面积越大越难分配，二分最大可行面积。

样例与解释： 半径 [1,1]、朋友数 F=2，总人数 3，答案为 π/2；每个派可切两块。

我的生日快到了，依据传统，我准备给大家分馅饼。我不只有一个馅饼，我有 N 个馅饼，口味和大小各不相同。有 F 个朋友会来参加我的派对，他们每个人都会得到一块馅饼。这一块必须是一整块馅饼，而不是几个小块拼凑起来的，并且大小应该一样。当然，我自己也想得到一块馅饼。那一块也应该是完整的一块，并且和其他人的大小相同。

我们每个人能得到多大的一块馅饼？所有的馅饼都是圆柱形的，高度都一样，所以馅饼的大小只取决于它的半径。

题目分析：

有 N 个派和 F 个朋友，加上自己一共 F + 1 个人。
每个派都是高为 1 的圆柱体，但是半径不同。
要求：

- 每个人分到 **体积相同** 的一块派。
- 只能从 **一个派** 中切出一块（不能从多个派拼凑）。
- 每个人只能得到 **一块** 派（不能要多个小块）。

问：每个人能分到的 **最大体积** 是多少。

每个派面积：$area[i] = \pi \cdot R_i^2$

如果我们猜测每个人能分得面积为 `A`，那么每个派可提供多少份：

若：$\sum cnt[i] \geq F + 1$

则说明面积 `A` 可行，还能尝试更大；
否则说明 `A` 太大，需要缩小。

因此可以使用实数二分求解

题解：



两个半径 1 的派面积各为 π；每份 π/2 时能提供四份，总人数为 3 时可行，余下的一份不必分配。



两个半径 1 的派、总人数 3 时，把面积按 π 归一化以便手算。每份 0.5π 可得 4 份，提升左端；0.75π、0.625π 都只能得 2 份，降低右端；不断逼近 0.5π。实际代码使用面积单位，`friends+1` 才是总人数。每个派分别向下取整，剩余碎料不能拼成一份。

```cpp
#include <vector>
#include <algorithm>
#include <cmath>
#include <stdexcept>
using namespace std;

bool check(const vector<double> &area, double A, int people) {
    int cnt = 0;
    for (double x : area)
        if (A ≤ 0)
            return false;
        else if (floor(x / A) ≥ people - cnt)
```

```cpp
            return true;
        else
            cnt += static_cast<int>(floor(x / A));
    return cnt ≥ people; // 总份数是否满足要求
}

double maxPieArea(const vector<int> &radii, int friends) {
    if (radii.empty() || friends < 0)
        throw invalid_argument("派非空且朋友数非负");
    long long people = 1LL * friends + 1;
    vector<double> area;
    for (int radius : radii)
        area.push_back(acos(-1.0) * radius * radius);
    double left = 0, right = *max_element(area.begin(), area.end());
    for (int i = 0; i < 100; ++i) {
        double mid = left + (right - left) / 2;
        if (mid == left || mid == right)
            break;
        long long count = 0;
        for (double x : area) {
            double pieces = floor(x / mid);
            if (pieces ≥ people - count) {
                count = people;
                break;
            }
            count += static_cast<long long>(pieces);
        }
        if (count ≥ people)
            left = mid;
        else
            right = mid;
    }
    return left;
}
```

cpp

1.3.8 二分答案的最大值与最小值模板

至此，二分答案的一般模板：

```cpp
// 伪代码：L/R 保存候选范围，check 计算当前候选值的判定结果。
/*
 * 二分答案通用模板
 *
 * 二分答案是一种基于单调性的高效搜索技术，通过二分查找确定满足条件的最优解。
 * 适用于求解最优化问题，如最大化最小值、最小化最大值等。
 */

using ll = long long;

/*
 * 检查函数 - 二分答案的核心
 *
 * 把候选值 x 代入判断，观察判定结果怎样随 x 变化：
 * - 对于最大化问题：若x满足条件，则所有≤x的值都满足
 * - 对于最小化问题：若x满足条件，则所有≥x的值都满足
```

```
 *
 * 参数 x: 待检查的值
 * 返回: bool 检查结果
 *
 * 从一个可行方案出发，改变 x 后仍能复用该方案，就能推导可行方向。
 * 注意: 实现要点: 根据具体问题设计高效的检查逻辑
 */
bool check(ll x) {
    // 判断 x 是否满足题目条件
    // 根据题目推导可行方向，再决定二分时缩短哪一侧。
    /*
     实际应用中除了上述示例的贪心遍历，可能更为复杂:

     二分 + BFS/DFS: 判断连通性或最短路是否满足限制。

     二分 + 动态规划: 判断是否存在满足条件的 DP 路径。

     二分 + 最大流: 网络流问题中的容量判定。
     */
}

/*
 * 整数二分答案模板 - 最大化最小值类问题
 *
 * 适用于求解满足条件的最大值，如: 最大最小间距、最大可行值等。
 * 单调性: 若mid满足条件，则所有≤mid的值都满足，目标是找到最大的满足条件的值。
 *
 * 返回: ll 满足条件的最大值
 *
 * 注意: 时间复杂度: O(log(R-L))
 * 注意: 适用场景: 最大化最小值、寻找右边界
 * 注意: 模板特点: 记录ans, l=mid+1, r=mid-1
 */
ll binary_search_maximize_min() {
    ll l = L, r = R, ans = -1;
    while (l ≤ r) {
        long long mid = l + (r - l) / 2;  // 防止溢出
        if (check(mid)) {                 // mid 能否满足要求
            ans = mid;                    // 记录可行答案
            l = mid + 1;                  // 继续尝试更大的值
        } else {
            r = mid - 1;                  // mid 不行，尝试更小的值
        }
    }
    return ans;
}

/*
 * 整数二分答案模板 - 最小化最大值类问题
 *
 * 适用于求解满足条件的最小值，如: 最小最大工作量、最小可行值等。
 * 单调性: 若mid满足条件，则所有≥mid的值都满足，目标是找到最小的满足条件的值。
 *
 * 返回: ll 满足条件的最小值
 *
```

```
 * 注意：时间复杂度：O(log(R-L))
 * 注意：适用场景：最小化最大值、寻找左边界
 * 注意：模板特点：不记录ans，r=mid，l=mid+1
 */
ll binary_search_minimize_max() {
    ll l = L, r = R;
    while (l < r) {
        long long mid = l + (r - l) / 2;  // 防止溢出
        if (check(mid)) {
            r = mid;                      // mid 可行，尝试更小的值
        } else {
            l = mid + 1;                  // mid 不可行，继续尝试更大的值
        }
    }
    return l;
}

/*
 * 浮点数二分答案模板
 *
 * 适用于实数范围内的二分答案，通过固定迭代次数或精度阈值控制搜索精度。
 * 常用于连续函数求根、数值计算等问题。
 *
 * 返回：double 满足条件的近似解
 *
 * 注意：时间复杂度：O(100) 或 O(log((R-L)/eps))
 * 注意：适用场景：实数范围的最优化、连续函数求根
 * 注意：模板特点：直接赋值mid，无±1操作
 */
double binary_search_real() {
    double l = L, r = R;
    // 方法1：固定迭代次数（避免无限循环，仍须检查精度要求）
    for (int i = 0; i < 100; i++) {
        double mid = (l + r) / 2;
        if (check(mid)) {
            r = mid;       // check 可行 → 在左半部分继续搜索
        } else {
            l = mid;       // check 不行 → 在右半部分继续搜索
        }
    }

    // 方法2：精度控制
    // const double eps = 1e-7;
    // while (r - l > eps) {
    //     double mid = (l + r) / 2;
    //     if (check(mid)) {
    //         r = mid;
    //     } else {
    //         l = mid;
    //     }
    // }

    return l; // 或 (l + r) / 2
}

```

```c++
/*
 * 二分答案解题步骤
 *
 * 1. 确定搜索范围 [L, R]
 * 2. 设计检查函数 check(x)，确保单调性
 * 3. 根据问题类型选择合适的模板：
 *    - 最大化最小值 → binary_search_maximize_min
 *    - 最小化最大值 → binary_search_minimize_max
 *    - 实数范围问题 → binary_search_real
 * 4. 从可行方案的复用关系推导判定方向
 *
 * 注意：核心思想：将最优化问题转化为判定性问题
 * 改变候选值后，已有方案仍能复用的方向就是可行方向
 * 注意：常见应用：分配问题、调度问题、数值计算等
 */
```

练习题： 洛谷 P1182 - 数列分段 Section II、洛谷 P1873 - 砍树、洛谷 P2678 - 跳石头。

2. 三分搜索

2.1 从单峰与单谷寻找极值

三分算法是一种用于在 **单峰 / 单谷函数** 上寻找 **极值（最大值或最小值）** 的算法。

单峰函数：函数值先单调增后单调减（求最大值）

单谷函数：函数值先单调减后单调增（求最小值）

核心思想：

每次将区间分成三段，通过比较中间两点函数值，丢弃不可能包含最优解的那一段

2.2 从曲线形状理解取点

先观察一个具体函数：$f(x) = -(x-2)^2 + 4$。在 [0,4] 中，它从 x=0 的 0 上升到 x=2 的 4，再下降到 x=4 的 0，形成一个峰。

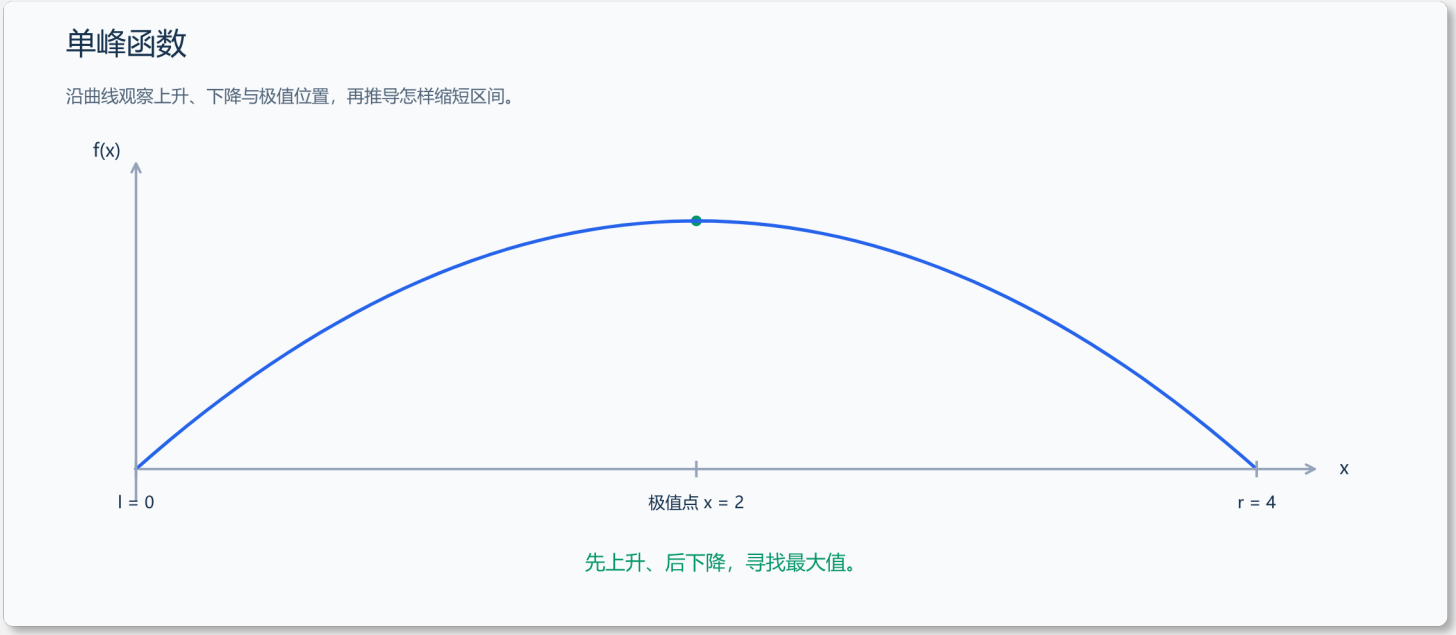
如果两个取点都在峰的左侧，右点更高，继续向右能接近峰；都在峰的右侧时，左点更高，继续向左能接近峰。两点分居峰的两侧时，峰就在它们之间。三分就是把这些位置关系转成区间更新，下一节沿图逐步推导。

反过来看两座峰之间的谷底：谷底两点的高低只能说明附近曲线的走向，远处还可能有另一座更高的峰。这时从局部比较中删去一整段，就可能把那座峰也删掉。这个例子说明，舍弃区间的理由来自曲线怎样变化。

2.3 比较两个取点并舍弃一侧区间

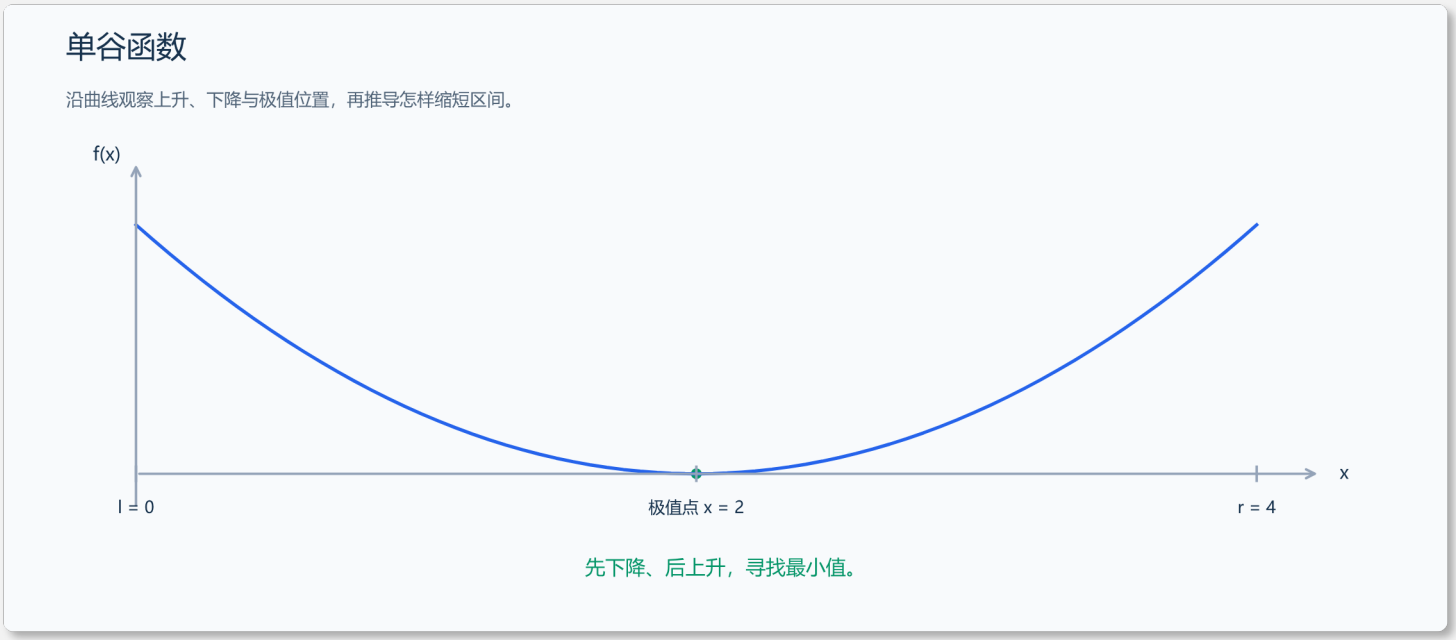
在单峰或单谷函数中

单峰函数：



图中的绿色点是最大值位置，区间内只有一个峰。

单谷函数：



图中的绿色点是最小值位置；求最小值时需要相应调整比较方向。

下面以单峰函数为例

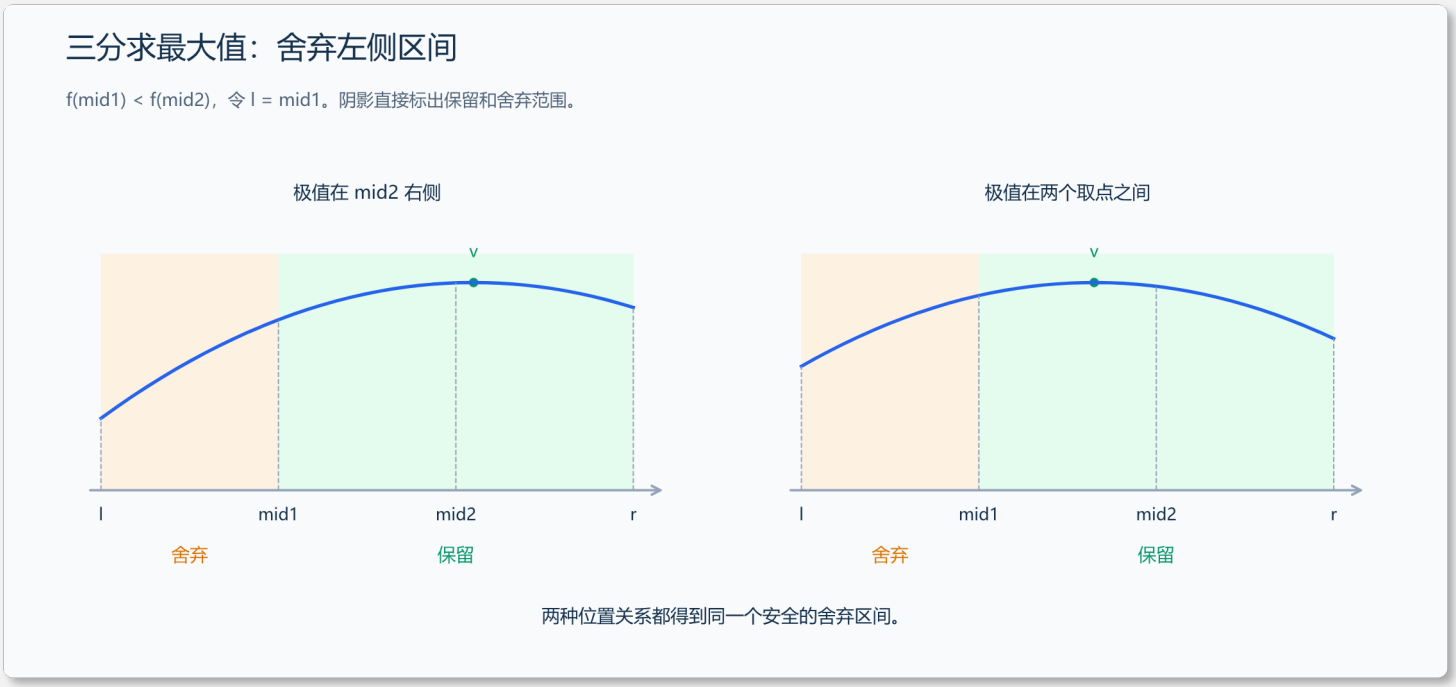
在区间 $[l,r]$ **中取两点** $mid1$ **，** $mid2$

且

$$mid1 = l + \frac{r-l}{3}, mid2 = r - \frac{r-l}{3}$$

则对于 $f(mid1)$ **和** $f(mid2)$ **的关系有：**

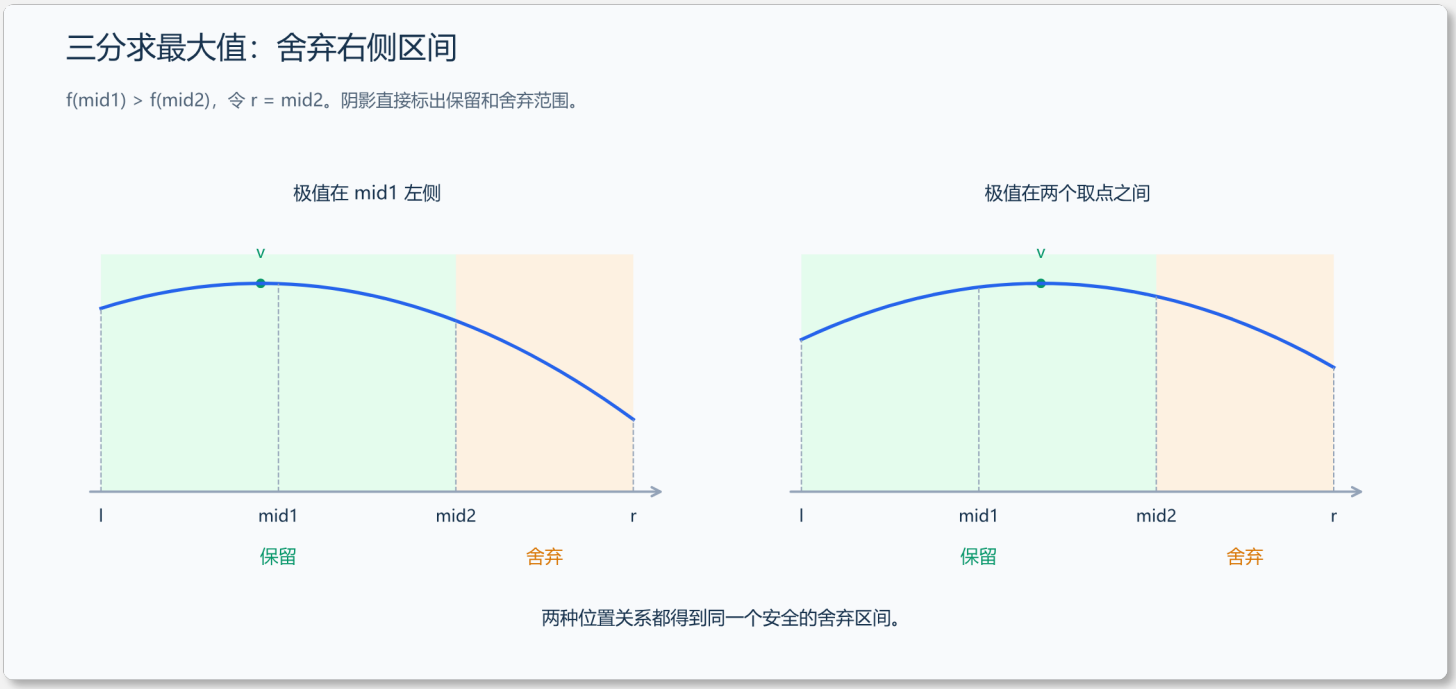
1. $f(mid2) > f(mid1)$ **时，有两种情况：**



两种极值位置都满足 `f(mid1) < f(mid2)`：橙色左段可舍弃，绿色范围继续搜索。

以上两种情况都有，极值点 v 在 $mid1$ 的右边，因此每次迭代使得 $l = mid1$ ，即使区间缩减为 $[mid1, r]$ ，并再次取两点分出三段逐步逼近即可

2. $f(mid1) > f(mid2)$ **时，也有两种情况：**



两种极值位置都满足 `f(mid1) > f(mid2)`：橙色右段可舍弃，绿色范围继续搜索。

以上两种情况都有，极值点 v 在 $mid2$ 的左边，因此每次迭代使得 $r = mid2$ ，即使区间缩减为 $[l, mid2]$ ，并再次取两点分出三段逐步逼近即可

2.4 实数与整数三分的实现

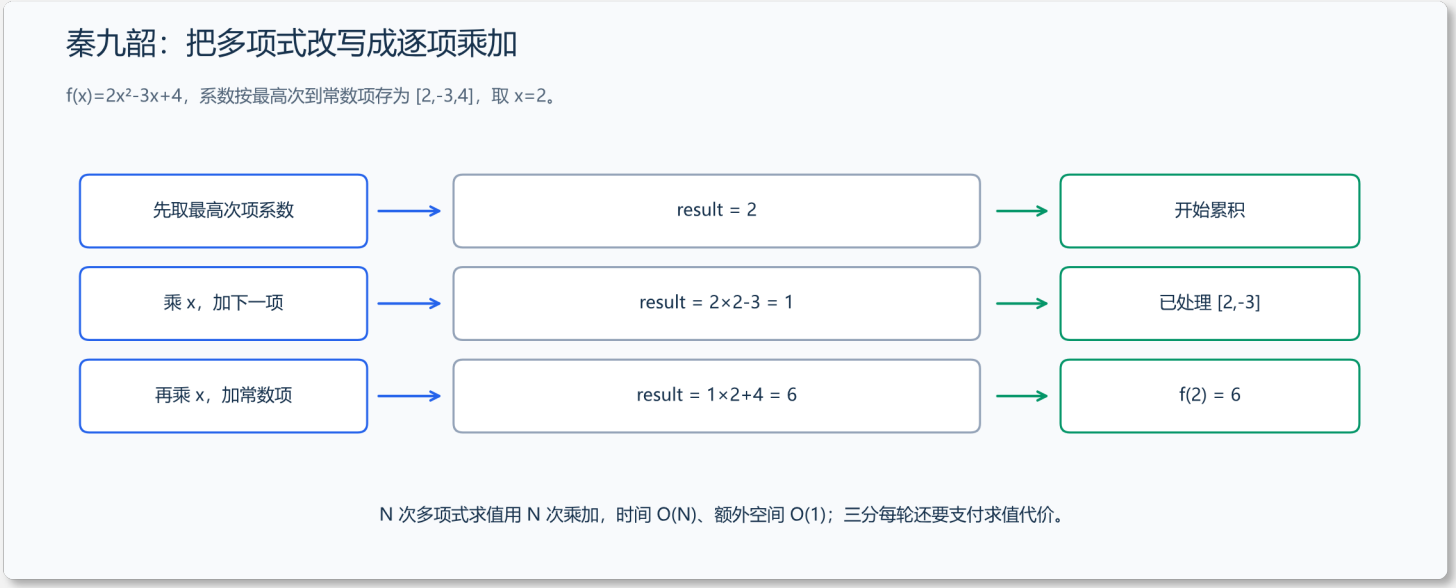
与二分算法类似，由于除法的取整方向的问题，也需要划分出整数三分或实数三分

2.4.1 实数三分：缩小区间逼近峰顶

题目原文（自拟）： 输入一个 `N` 次多项式的系数，从最高次项到常数项依次保存为 `a[0..N]`，以及有限区间 `[l,r]`。保证多项式在区间内先严格上升、后严格下降，只有一个最大值点；使用 double 计算函数值。求最大值点 `x` 的浮点近似。教学样例使用 `N=2`、系数 `[-1,4,0]`、区间 `[0,4]`，要求绝对误差不超过 `1e-6`。

思路分析： 每轮取两个三等分点。左点的函数值较小时，峰顶不在最左的三分之一，因此移动左边界；否则移动右边界。比较的是函数值，最终返回的是自变量 `x`，两者不能混淆。求最小值的模板将比较方向反过来，对应的曲线则先下降、后上升。

多项式不要每次分别计算所有幂。按 $(((a_0x+a_1)x+a_2)x+\cdots)x+a_N$ 逐项乘加，这就是**秦九韶算法**。`poly_func(x,a,N)` 要求系数恰好有 `N+1` 项；它每次求值花 $O(N)$ 时间。



图用 `2x²-3x+4` 在 `x=2` 的求值过程说明乘加顺序；三分样例的系数不同，但求值方法相同。

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

const double eps = 1e-6; // 精度阈值

double poly_func(double x, vector<double> const &a, int n) {
    double result = a[0]; // 初始化为最高次项系数
    for (int i = 1; i ≤ n; ++i) {
        result = result * x + a[i]; // 秦九韶算法迭代
    }
    return result;
}

template <typename Func> double ternary_search_max(Func f, double l, double r) {
    // 方法1: 固定迭代次数（避免无限循环，仍须检查精度要求）
```

```cpp
    for (int i = 0; i < 100; ++i) {
        double k = (r - l) / 3.0;
        double mid1 = l + k; // 左三等分点
        double mid2 = r - k; // 右三等分点

        if (f(mid1) < f(mid2)) {
            l = mid1; // 最大值在[mid1, r]区间
        } else {
            r = mid2; // 最大值在[l, mid2]区间
        }
    }

    return (l + r) / 2.0; // 返回区间中点作为近似最优解
}

template <typename Func> double ternary_search_min(Func f, double l, double r) {
    for (int i = 0; i < 100; ++i) {
        double k = (r - l) / 3.0;
        double mid1 = l + k;
        double mid2 = r - k;

        if (f(mid1) > f(mid2)) { // 注意：这里比较符号与求最大值相反
            l = mid1;            // 最小值在[mid1, r]区间
        } else {
            r = mid2; // 最小值在[l, mid2]区间
        }
    }

    return (l + r) / 2.0;
}
```

样例与解释： 对 `f(x)=-x²+4x`，在 `[0,4]` 上先上升后下降，最大值点是 `x=2`，函数最大值为 4。调用 `ternary_search_max` 时，将系数数组捕获到一个 lambda 中，再用 `poly_func` 计算传入位置的函数值。返回结果约为 2；若题目还要最大函数值，再对返回点求一次值。

复杂度与精度： 上面的三分固定迭代 100 次，每轮两次多项式求值，所以总计算量约为 `200N`，时间 $O(100N)$，除系数存储外额外空间 $O(1)$。若用误差作为停止条件，理论轮数为 $O(\log((r-l)/eps))$，总时间还要乘上求值代价。每轮区间宽度变为原来的 `2/3`，因此初始宽度为 L 时，t 轮后约剩 `L*(2/3)^t`。把这个宽度与题目允许的位置误差相比，就能估算需要多少轮。

练习题： 洛谷 P3382 - 三分法。

2.4.2 整数三分：比较取点后枚举剩余值

整数三分 用于在 **离散整数区间** $[l,r]$ 上，寻找一个函数的极值点（最大值或最小值）

题目原文（自拟）： 给定整数 l、r、k、c，l≤r，l、r、k 的绝对值不超过 10^6，c 的绝对值不超过 10^{12}。在整数闭区间 [l,r] 内寻找 f(x)=-(x-k)²+c 的最大值位置，若有多个最优位置返回一个即可；最大值可由 f(返回位置) 计算。下面用全局参数 k、c 与函数接口演示。

推导与样例： 抛物线在顶点 k 左侧上升、右侧下降。取两个三等分点，左值小于右值时可以排除更左侧，反之保留左侧与中间。剩余区间太短时枚举，不依赖浮点精度。l=-3、r=9、k=3、c=10 时返回位置 3，最大值为 10；k 不在区间时最优点是较靠近 k 的端点。

复杂度与练习： 假设 f 为 O(1)，三分时间 O(log(r-l+1))、空间 O(1)；固定抛物线的解析解为 O(1)。洛谷 P3382 - 三分法（多项式实数单峰题，使用实数版本）。

项目	整数三分	实数三分
区间	离散整数	连续实数
终止条件	区间长度很小	精度误差
结尾处理	暴力枚举剩余点	取中点
易错点	死循环	精度不足

例题：

给定一个定义在整数区间 $[l, r]$ 上的函数：

$$f(x) = -(x-k)^2 + c$$

其中 $l \le x \le r$，l, r, k, c 均为整数。请你找出 **使函数值最大的整数** x，并输出这个最大值。



图示 f(x)=-(x-3)²+10，在 [1,4] 中逐个比较 1、2、3、4，避免整数取整使区间不再缩小。

代码组合方式： 前三段共享第一段的头文件、ll、k、c 和 f；模板也使用 ll。函数名各不相同，可以放进同一文件。

1. 明确函数和返回值

以 f(x)=-(x-k)²+c 为例，越靠近 k，函数值越大。现在从整数区间 [l,r] 中寻找最大值的位置；函数返回位置 best，再计算 f(best) 就得到最大函数值。

```cpp
// 示例函数为 f(x)=-(x-k)²+c。
#include <algorithm>
using namespace std;

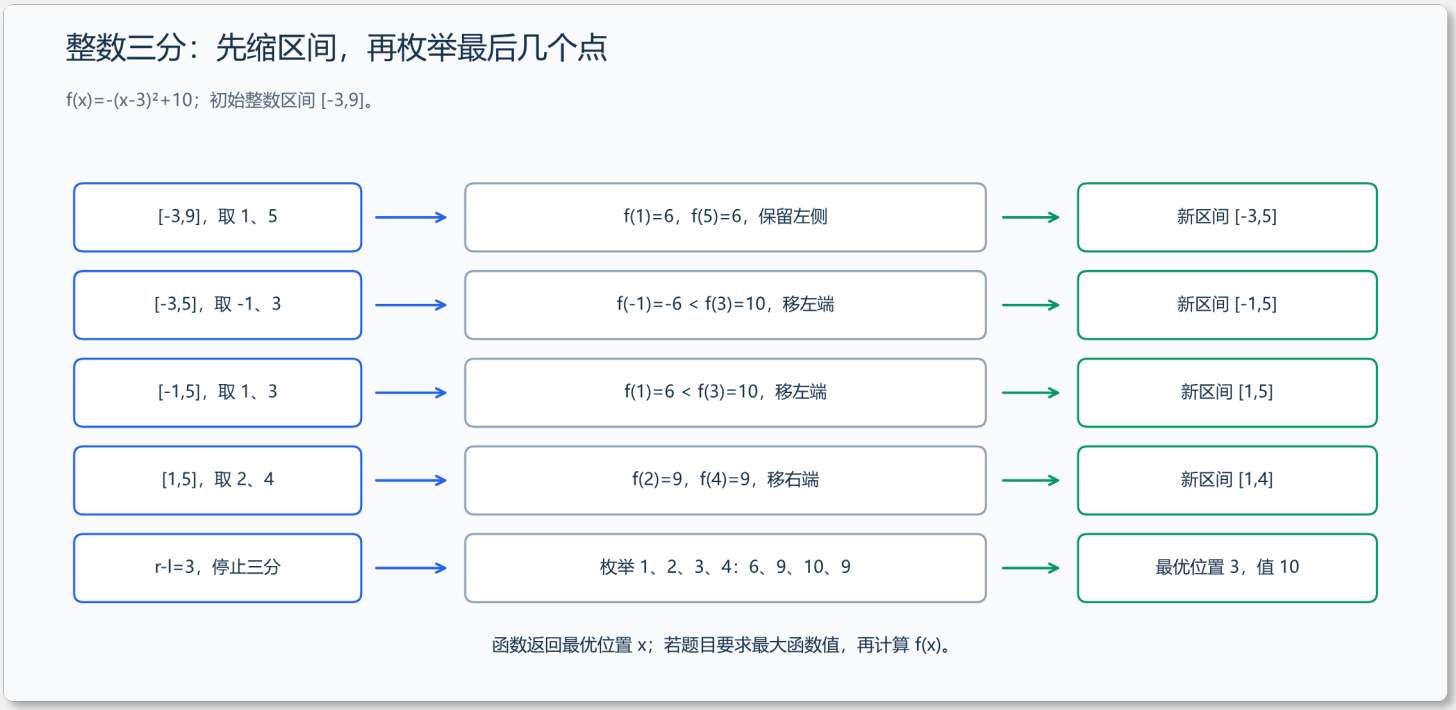
using ll = long long;

ll k, c; // 抛物线参数：顶点在(k, c)

ll f(ll x) {
    ll diff = x - k;
    return -(diff * diff) + c; // 避免使用pow，提高效率
}
```

2. 三分缩区间，再枚举剩余整数

整数区间不能依赖无限缩小。当前代码在 r-l≤3 时直接枚举所有剩余点，比较值后保存最优位置，避免取整导致区间停滞。



逐步缩小整数区间并枚举最后四个候选。

```cpp
// 整数区间不能依赖无限缩小。
ll ternary_search(ll l, ll r) {
    // 边界检查
    if (l > r)
        return -1; // 无效区间

    // 特殊情况处理：区间长度很小直接暴力
    if (r - l + 1 ≤ 3) {
        ll best = l;
        for (ll x = l + 1; x ≤ r; x++) {
```

```cpp
            if (f(x) > f(best))
                best = x;
        }
        return best;
    }

    // 三分阶段：不断缩小搜索区间
    while (r - l > 3) {             // 区间长度>3时继续三分
        ll mid1 = l + (r - l) / 3; // 左三等分点
        ll mid2 = r - (r - l) / 3; // 右三等分点

        // 比较两个三等分点的函数值
        if (f(mid1) < f(mid2))
            l = mid1; // 峰值在[mid1, r]区间
        else
            r = mid2; // 峰值在[l, mid2]区间
    }

    // 暴力阶段：枚举剩余的所有点
    ll best = l;
    for (ll x = l + 1; x ≤ r; x++) {
        if (f(x) > f(best))
            best = x;
    }

    return best;
}
```

cpp

3. 已知顶点时使用解析解

这条抛物线在 k 处最高。k 在区间内就取 k；在左侧就取 l，在右侧就取 r，因此答案是把 k 限制到 [l,r]。这是该函数的特定性质，不能当成任意单峰函数的解法。

```cpp
// 这条抛物线在 k 处最高。
ll optimal_solution(ll l, ll r) {
    if (l > r)
        return -1;
    return max(l, min(r, k));
}
```

cpp

4. 抽出可复用的整数三分模板

这里继续使用 ll 表示 long long。以峰形的整数函数为例，区间较长时比较两个三等分点，沿上一节推导的方向缩短区间；只剩少量整数时逐个计算并选出最大值。谷形函数则反过来比较，最终逐个取最小值。

```cpp
// ll 表示 long long；本函数比较峰形曲线上的两个取点，最后逐个检查剩余整数。
template <typename Func> ll integer_ternary_search(Func f, ll l, ll r) {
    while (r - l > 3) {
        ll mid1 = l + (r - l) / 3;
        ll mid2 = r - (r - l) / 3;

        if (f(mid1) < f(mid2))
            l = mid1;
```

```cpp
        else
            r = mid2;
    }

    ll best = l;
    for (ll x = l + 1; x ≤ r; x++) {
        if (f(x) > f(best))
            best = x;
    }
    return best;
}
```

使用暴力搜索的原因，在整数域中：

- `mid1` 和 `mid2` **可能相等**
- 区间长度小的时候三分 **不再缩小区间**
- 容易死循环或漏最优点

所以统一策略是：

当 `r - l ≤ 3`，直接枚举

以下情况 **不能用** 整数三分：

- 函数 **不单峰**
- 有多个局部极值
- 函数是“阶梯型”或“锯齿型”
- 题目本质是 **二分答案 + 判定**

若目标能转换成单调可行性边界，就分析二分答案；若直接优化单峰函数，再考虑三分。选择方法需要依据题目的结构。

2.5 二分与三分的判定方式对照

对比点	二分	三分
目标	找边界 / 判定条件	找极值
函数要求	单调	单峰 / 单谷
每轮比较	1 次	2 次
时间复杂度	$O(\log_2 n)$	$O(\log n)$（每轮约保留 2/3）
常见用途	判定、最小可行解	最大/最小函数值

2.6 从求根、判定与求极值选择搜索方式

(1). **物理问题**

- 最短距离
- 最小能量
- 最优角度

(2).**几何问题**

- 点到曲线最短距离
- 最大面积 / 最小周长

(3).**算法竞赛常见**

- 函数最值优化
- 与 DP / 贪心 结合

练习题： 洛谷 P3382 - 三分。

（十一） 贪心算法

有些任务可以逐次选择当前最合适的候选，但“看起来最好”还不足以证明正确。先尝试具体选择，再用替换、交换或反例判断它能否推广。

1. 逐步选择当前最有利的决策

贪心算法是指在求解问题时，每一步都 **选择当前看起来最优的策略**，从而期望得到全局最优解。

核心思想：

局部最优策略 → 全局最优解

但贪心并不总是正确，使用前需证明：

- **贪心选择性质**（每次局部最优可构成全局最优解）
- **最优子结构**（问题可分解成子问题，每个子问题也可使用贪心）

2. 从题目结构寻找贪心策略

场景	特征
每一步做局部最优能得到全局最优	如区间问题、最短作业时间、最小钱币数量
解可以逐步构造并可修改	动态增量策略
具有单调结构	解越走越接近最终答案

3. 贪心策略的例题与反例

3.1 最少硬币：贪心反例

题目原文（自拟实验）： 给定正整数面值数组和非负金额 M，按“每次尽量取不超过剩余金额的最大面值”运行贪心策略，返回这一策略选择的硬币数量；走到无法继续时返回失败。用 [1,3,4]、M=6，以及 [2,3,5]、M=9 检验它是否总能得到最少硬币，并与后面 DP 的正确答案比较。这个实验代码展示策略的失效，不是通用最优解。

给定不同面额的硬币 $[c1, c2, ..., ck]$ ，以及一个总金额 M 。每种硬币数量无限，问用最少的硬币数量凑出金额 M 。



同样凑出 6，局部选择最大面值会多用一枚硬币。后面的最少硬币 DP 会求得全局最优值。

```cpp
#include <vector>
#include <algorithm>
using namespace std;

int count(int amount, vector<int> &coins) {
    sort(coins.begin(), coins.end());
    int used = 0;
    for (auto it = coins.rbegin(); it != coins.rend(); ++it) {
        used += amount / *it; // 当前面值可取 amount / *it 枚
        amount %= *it;
    }
    return amount == 0 ? used : -1;
}
```

这个题解对随意面值贪心策略无法求出最优解甚至会导致无解

例如面值 1 3 4 求 6，4+1+1，但最优解为 3+3

面值 2 3 5 求 9，5+3+! 无解，但其实有解 5+2+2

3.2 最少加油次数：用堆选择已途经的站点

题目原文（自拟）： 车辆从位置 0 出发，终点在 `d ≥ 0`，初始油量为 `m ≥ 0`。每行驶一单位距离消耗一单位油，油箱容量不受限制。若干站点以 `(位置,油量)` 给出，位置和油量均为非负 `int`，每站最多加油一次。求到达终点的最少加油次数，无法到达返回 `-1`。

思路分析： 用 `maxReach` 表示已经选定的油量能够覆盖的最远位置。把所有位置不超过 `maxReach` 的未使用站点放入大根堆；如果还到不了终点，从这些站点中选油量最大的一个，把可达范围扩大。这里计算的是“应在已路过的哪些站加油”，没有要求车辆倒车。

为什么选最多油？ 当前候选站都是不增加次数就能路过的站。若某个方案下一次选了油量较小的候选，把这一次替换成候选中的最大油量，次数不增加，可达范围也不会变小，后续可选择的站只会更多。油箱无限使提前获得更多油不会浪费或挤占容量。连续应用这个交换理由，便能保持最少次数。



样例先从位置 10 的站选 60 单位油，可达范围变成 70；再从已经可达的站中选位置 60 的 40 单位油，范围变成 110，覆盖终点 100。

代码示例：

```cpp
#include <vector>
#include <utility>
#include <algorithm>
#include <queue>
using namespace std;

int minRefuelStops(vector<pair<int, int>> stations, int m, int d) {
    // 预处理：按位置对加油站排序（升序）
    sort(stations.begin(), stations.end());

    priority_queue<int> pq; // 最大堆保存可选加油量
    long long maxReach = m; // 当前最远可达位置（起始油量覆盖范围）
    int ans = 0;            // 加油次数
    int i = 0;              // 遍历加油站指针
    int n = stations.size();

    // 循环直到能够到达终点
    while (maxReach < d) {
        // 将所有当前可到达的加油站加入优先队列
        while (i < n && stations[i].first ≤ maxReach) {
            pq.push(stations[i].second); // 只记录油量，位置已排序
            i++;
        }

        // 没有可用的加油站 → 无法继续前进
        if (pq.empty())
            return -1;

        // 贪心选择：加油量最大的加油站
        maxReach += pq.top(); // 增加最大行驶距离
        pq.pop();             // 移除已使用的加油站
        ans++;                // 增加加油次数
    }
```

```cpp
    return ans;
}
```

样例与解释： `m=10、d=100`，站点为 `[(10,60),(20,30),(30,30),(60,40)]`，答案为 `2`。初始油足够时返回 `0`；仍未到终点而堆为空时，说明无法获取任何新的补给，返回 `-1`。同位置的多个站也应分别入堆。

复杂度分析： 排序需要 $O(n \log n)$，每站至多入堆、出堆各一次，合计仍为 $O(n \log n)$；参数副本与堆占 $O(n)$ 空间。这里一次拿走已经经过站点的整份油量，优化的是停靠次数；油箱容量或购买价格进入题目后，决策还会包含装多少油、在哪买油，可以从这些新增决策建立状态。另一种思路是用 DP 记录“加油 `k` 次能到的最远位置”，常见实现为 $O(n^2)$。

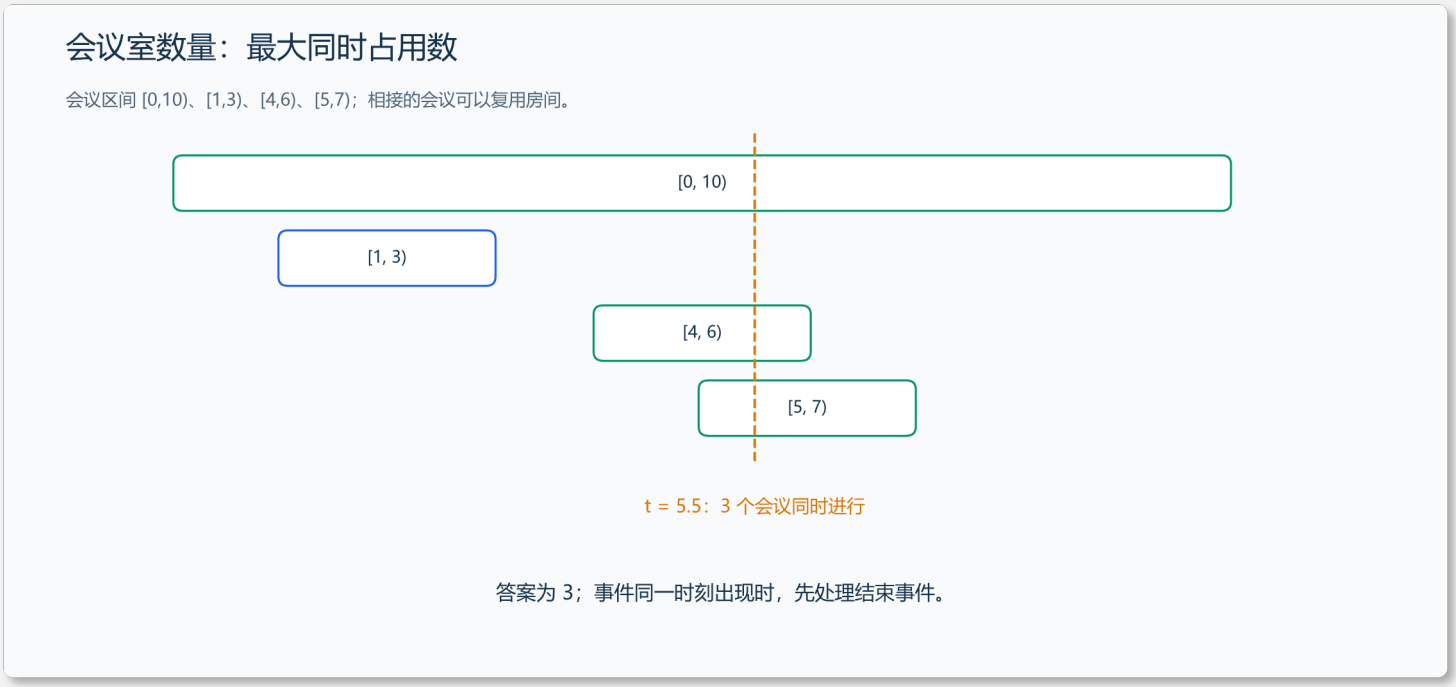
练习题： 洛谷 P1016 - 旅行家的预算（进阶变式：有限油箱、按单价购买、目标为最小费用，需重新推导贪心，不能照搬最大油量堆）。

3.3 最少会议室：扫描开始与结束事件

题目原文（自拟）： 给定若干会议半开区间 `[start,end)`，保证 `start < end`。同一会议室不能同时召开两个会议；一个会议结束的时刻可以立刻安排另一个会议。求安排全部会议需要的最少会议室数。空输入返回 `0`。

思路分析： 把开始、结束时刻分别排序，用两个下标按时间处理事件。开始时占用数加一，结束时减一，记录过程中的最大占用数。开始和结束相同时，先处理结束，符合半开区间的边界。

同时进行 `k` 场会议时，至少需要 `k` 间房，这是下界。按时间安排时，会议开始就复用空闲房，没有空闲房才开新房；增加房间时的需求恰好等于当前同时进行的数量。因此最大重叠数也能达到，便是最少房间数。



图中四场会议在时刻 5.5 同时占用三间房，答案为 3；完整调用样例采用 `[0,30)、[5,10)、[15,20)`，其最大占用为 2，下一张构造图展示两间房的安排。遇到同一时刻的结束与开始事件，应先处理结束，符合半开区间的约定。



再用 `[0,10)、[1,3)、[4,6)、[5,7)` 手算事件：时刻 0、1 开始，占用升到 2；时刻 3 结束，降到 1；时刻 4、5 开始，升到 3；后续结束逐步归零，因此需要 3 间房。下面程序的调用样例仍是前述三场会议，返回 2。这两组输入分别展示事件过程和函数结果，不要混用。



先由同时进行的会议数给出下界，再展示两间房的合法安排。

代码示例：

```cpp
#include <vector>
#include <algorithm>
using namespace std;

int minMeetingRooms(const vector<vector<int>> &meetings) {
    vector<int> start, finish;
    for (const auto &meeting : meetings) {
        start.push_back(meeting[0]);
        finish.push_back(meeting[1]);
    }
    // 分别排列开始与结束事件，用两个下标合并扫描。
    sort(start.begin(), start.end());
    sort(finish.begin(), finish.end());
    int i = 0, j = 0, rooms = 0, answer = 0;
    while (i < static_cast<int>(start.size())) {
        // 时刻相等时先释放房间，符合半开区间 [s,e) 的约定。
        if (start[i] < finish[j]) {
            ++rooms;
            answer = max(answer, rooms);
            ++i;
        } else {
            --rooms;
            ++j;
        }
    }
    return answer;
}
```

样例与解释： 上面三个区间返回 `2`；`[0,5)、[5,10)` 返回 `1`，因为时刻 5 可以交接。开始、结束数组分开后不再保留每场会议的配对，但只求占用数量时不需要配对；若要输出各会议分配到的房间，应另保存编号并用堆管理房间。

复杂度与扩展： 排序时间 $O(n \log n)$，扫描时间 $O(n)$，额外空间 $O(n)$。小范围整数时间可以用差分事件，代价为 $O(n+U)$，`U` 是时间范围；时间跨度很大时，不能直接开长度为 `U` 的数组。最少会议室与“选择最多不冲突会议”目标不同，后者常按结束时刻贪心选择。

4. 常见贪心问题与对应选择策略

例题	贪心策略
活动安排问题（区间调度）	按结束时间最早
特定面额体系的找零钱	先使用最大面额；任意面额未必最优
作业调度（deadline + reward）	按奖励排序+优先队列
加油站/最小加油次数	优先队列保存已过油站的油量

例题	贪心策略
背包问题（可分割）	按单位价值排序
吃苹果最多天数	小根堆保存过期时间
匹配问题（区间）	双指针

5. 设计贪心策略步骤

为了正确构造并验证一个贪心算法，通常需要遵循以下四个关键步骤：

5.1 明确目标（目标函数定义）

首先要明确：问题的最终目标是什么？

- 是 **最大化** 某个量？（例如收益、区间数、可覆盖范围）
- 是 **最小化** 某个量？（例如代价、加油次数、硬币数）
- 还是 **判断可行性**？（能否覆盖区间、能否到达终点）

明确目标是制定贪心策略的基础。例如，在加油问题中目标是 **最少的加油次数**，而在最小会议室问题中目标是 **最少冲突会议使用的会议室总数**。

先用题目写出要最小化或最大化的量，再手算几种方案。观察当前选择对后续决策的影响，就能继续推导贪心的交换过程，或用状态保存不同选择的结果。

5.2 尝试局部最优策略（构造贪心规则）

根据目标，思考每一步中什么选择是 **当前看来最优** 的：

- 按某种关键属性排序：如区间按结束时间升序
- 动态选择当前最优的候选项：如优先队列选择最大油量
- 优先选择花费最小或收益最大的方案

常见的贪心选择模式：

贪心方向	示例
优先选择结束最早的区间	区间调度、最多不重叠区间数
优先选择当前最大收益	本节无容量限制的加油问题
优先选择当前最小代价	Dijkstra 取最小候选距离；哈夫曼每次合并最小权值
按单位价值排序优先更优项	分数背包

贪心方向	示例
按当前局部最有利项推进	字典序最小+贪心删字符

设计时通常需要先写出若干样例，推演局部策略是否合理。

若局部最优选择会影响未来选择，使未来不可达或变差 → 贪心不可用而应使用动态规划

5.3 证明正确性

证明贪心时，分别说明两件事：

- **贪心选择性质**（局部最优可构成全局最优）
- **最优子结构**（问题可以分解成结构相同的子问题）

证明常用方法：

5.3.1 交换论证法（Exchange Argument）

证明：若最优解不是从贪心选择开始，我们可以通过交换顺序使其变得不更差且变为贪心形式



把首个活动换成更早结束者后，后续活动仍然合法，方案数量保持不变。

例如另一个问题“选择最多个互不重叠活动”，按结束时间最早选择：

- 假设最优解中第一个区间不是结束时间最早的
- 交换它们不会减少能安排的区间数

- 所以存在一个最优解以贪心项开始 → 贪心最优

练习题： 洛谷 P1803 — 凌乱的yyy / 线段覆盖。其他调度问题需要根据目标与约束重新设计交换关系。

5.3.2 最优子结构证明法

若选择某个贪心项后，剩余问题仍是相同结构的子问题，并且也可以使用贪心解决

比如加油问题：

- 选择当前位置能获取的最大油量
- 更新可达范围形成与原问题相同的新问题（规模减少）
- 子问题继续使用相同策略

0/1 背包和 LIS 具有最优子结构，但简单的局部贪心选择未必正确；动态规划的最优子结构不能替代贪心选择性质的证明。

5.4 验证样例与极端情况

设计好策略后需进行充分测试，尤其：

- **最小规模**（n = 1、m = 0 等）
- **有多个等价选择**（相同结束时间、相同油量）
- **无法完成任务**（pq 为空、区间不可覆盖）
- **反例测试贪心是否失败**（寻找破坏局部策略的例子）

比如硬币找零：

面值 1 3 4　求 6
贪心：4+1+1（3 枚）
最优：3+3　（2 枚）

贪心无法适用，需 DP

典型结论验证方式：

- 自己构造反例
- 与 DP 结果对比
- 证明局部不会造成未来损害

练习题： 洛谷 P1803 - 凌乱的 yyy / 线段覆盖、洛谷 P1090 - 合并果子、洛谷 P2240 - 部分背包问题。

（十二） 搜索：递归、回溯、DFS、BFS 与记忆化

当一次局部选择不能保证结果时，就记录当前状态并继续尝试不同分支。先沿调用进入与返回，再加入撤销、访问标记、队列和缓存。

1. 递归：把当前任务拆成更小的任务

递归是一种 **函数直接或间接调用自身** 的方法，本质是：

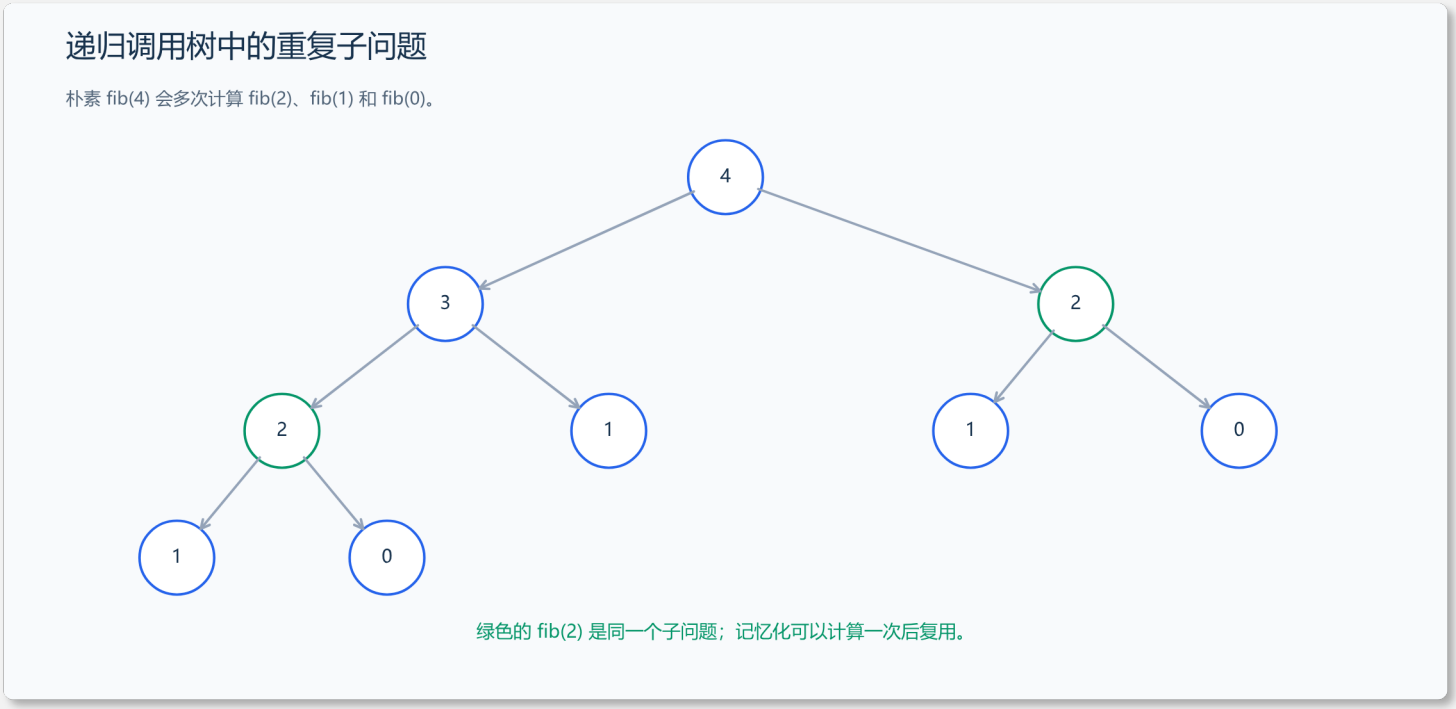
用同一个问题的更小规模版本，来解决当前问题

递归一定要满足三点：

1. **递归定义（递归关系）**
2. **递归到哪里直接返回**
3. **每次缩小问题，直到得到直接结果**

若递归不能到达终止条件，就会不断增加调用栈，最终可能栈溢出。

以经典的斐波那契数列为例



节点中的数表示参数 n，而不是返回值。相同参数的调用反复出现，说明有重复计算。

```cpp
// 递归终点为 0、1；重复子问题很多，本例只用于小 n 的递归演示。
int fib(int n) {
    if (n ≤ 1)
        return n;
    return fib(n - 1) + fib(n - 2);
}
```

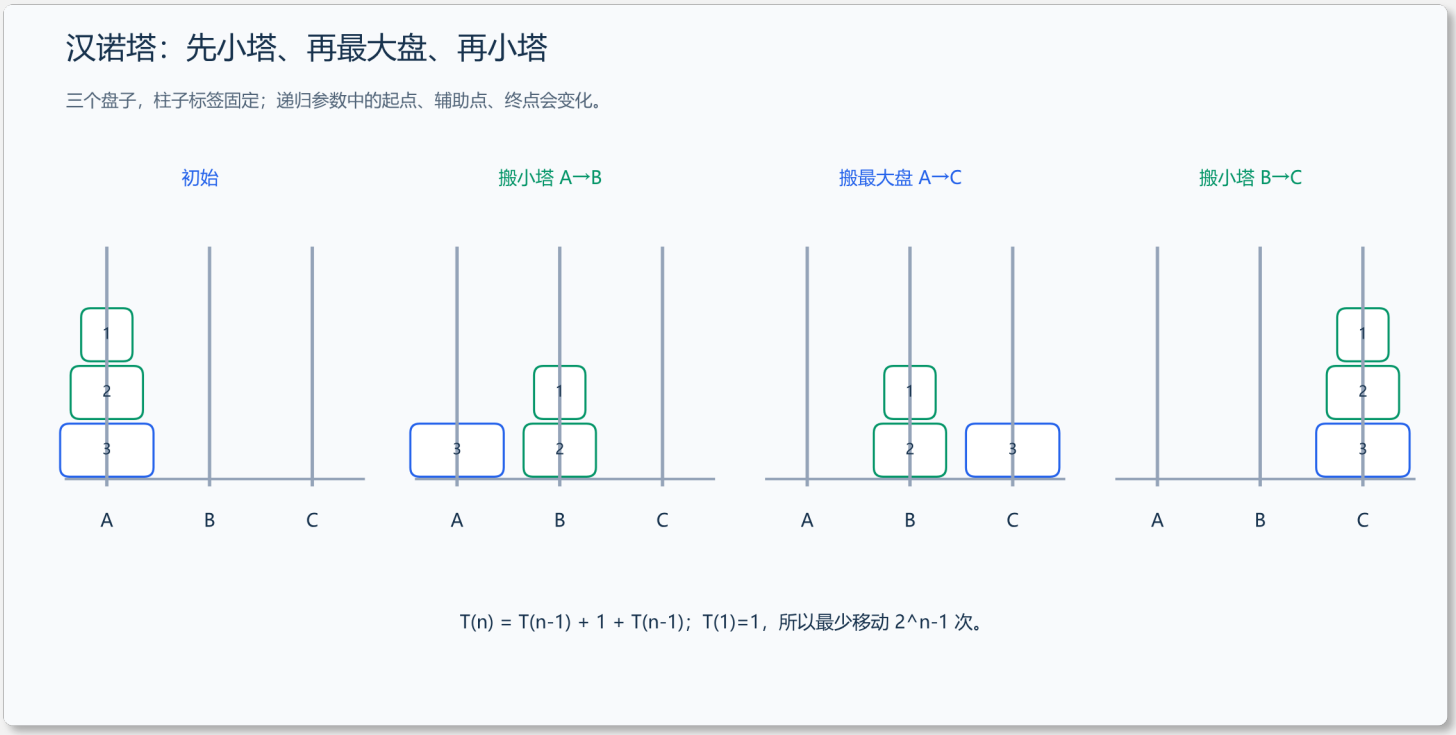
它先递归求出更小的子问题，再利用返回值计算当前项。这个朴素版本重复计算很多，时间为指数级，递归栈深度为 $O(n)$；仅用于理解递归。示例要求 `0 ≤ n ≤ 20`。

1.1 汉诺塔问题

题目原文（自拟）： 有 A、B、C 三根柱子和 `1 ≤ n ≤ 16` 个大小不同的盘子。初始时盘子都在 A 上，小盘在上、大盘在下。每次只能移动某根柱子最上面的一个盘子，大盘不能放在小盘上。把全部盘子移到 C，B 可作辅助；按顺序输出每次移动，每行格式为 `起点 → 终点`。

思路分析： 要移动最大的盘子，先把压在它上面的 `n-1` 个盘子全部移到 B；然后把最大盘从 A 移到 C；最后把 B 上的 `n-1` 个盘子移到 C。两次“小塔搬运”遵守同样的规则，所以可以递归调用。

`hanoi(n, from, aux, to)` 的参数分别表示盘子数、起点、辅助点、终点。第一次递归要把辅助点当终点，原终点当辅助点；第二次递归再把原辅助点当起点。参数位置是在描述本次任务，不是永远固定的柱子角色。



图以三个盘子为例：先搬上方两个盘子，再搬最大盘，最后搬回小塔。绿色步骤本身也由相同的递归规则完成。

代码示例：

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

```

```cpp
void hanoi(int n, char from, char aux, char to) {
    if (n ≤ 0)
        return;
    // 递归终止条件：只有一个盘子时直接移动
    if (n == 1) {
        cout << from << " → " << to << endl;
        return;
    }

    // 步骤1：将上方n-1个盘子从from移动到aux（以to为辅助）
    hanoi(n - 1, from, to, aux);

    // 步骤2：将最大的盘子从from移动到to
    cout << from << " → " << to << endl;

    // 步骤3：将n-1个盘子从aux移动到to（以from为辅助）
    hanoi(n - 1, aux, from, to);
}

int main() {
    int n;
    cin >> n;
    // 调用汉诺塔函数
    hanoi(n, 'A', 'B', 'C');
}
```

样例与解释： 输入 `3`，输出如下。可以边读边画三根柱子，检查每一步移动的都是顶端盘子。

```text
A → C
A → B
C → B
A → C
B → A
B → C
A → C
```

复杂度分析： 移动次数满足 $H(n)=2H(n-1)+1$、$H(1)=1$，因此 $H(n)=2^n-1$，时间为 $O(2^n)$。移动最大盘前后都至少需要完成一次小塔搬运，递归方案恰好达到这个下界。最大调用深度为 `n`，额外栈空间为 $O(n)$；总调用次数与栈深度是两个量。

扩展阅读： 可以用显式栈模拟这三步，也可以利用二进制计数规律生成移动序列：第 `i` 步移动的盘子编号等于 `i` 的最低有效 1 位的位置再加一。格雷码相邻编码只改变一位，也能帮助观察“一次只动一个盘子”的结构。限制移动方向或增加柱子后，需要重新设计递推，不能继续沿用三柱的步数公式。四柱版本的最优步数问题已有证明，不能把所有多柱版本都称为未解决；参见 Grosu 的论文。

练习题： 洛谷 P1242 — 新汉诺塔（进阶：初始与目标状态任意，需要扩展递归划分）。

2. 回溯：选择、递归与撤销

回溯用深度优先的顺序枚举选择。每层先作出一个决定，再进入下一层；遇到死路或处理完一个答案，就返回上一层，恢复这次选择改动的共享状态，然后尝试其他决定。

理解回溯时，要分清三个动作：**返回**改变当前执行的是哪一层函数；**撤销**恢复数组、集合或路径等共享数据；**继续枚举**尝试该层的下一个选择。函数返回不会自动撤销对共享数组的修改。

1. 定义搜索状态，说明这一层已经决定了什么，还要决定什么。
2. 列出全部合法选择，并明确怎样判断冲突和完成。
3. 对每个选择依次执行“记录选择 → 递归 → 撤销选择”，使下一分支从相同的初始状态出发。

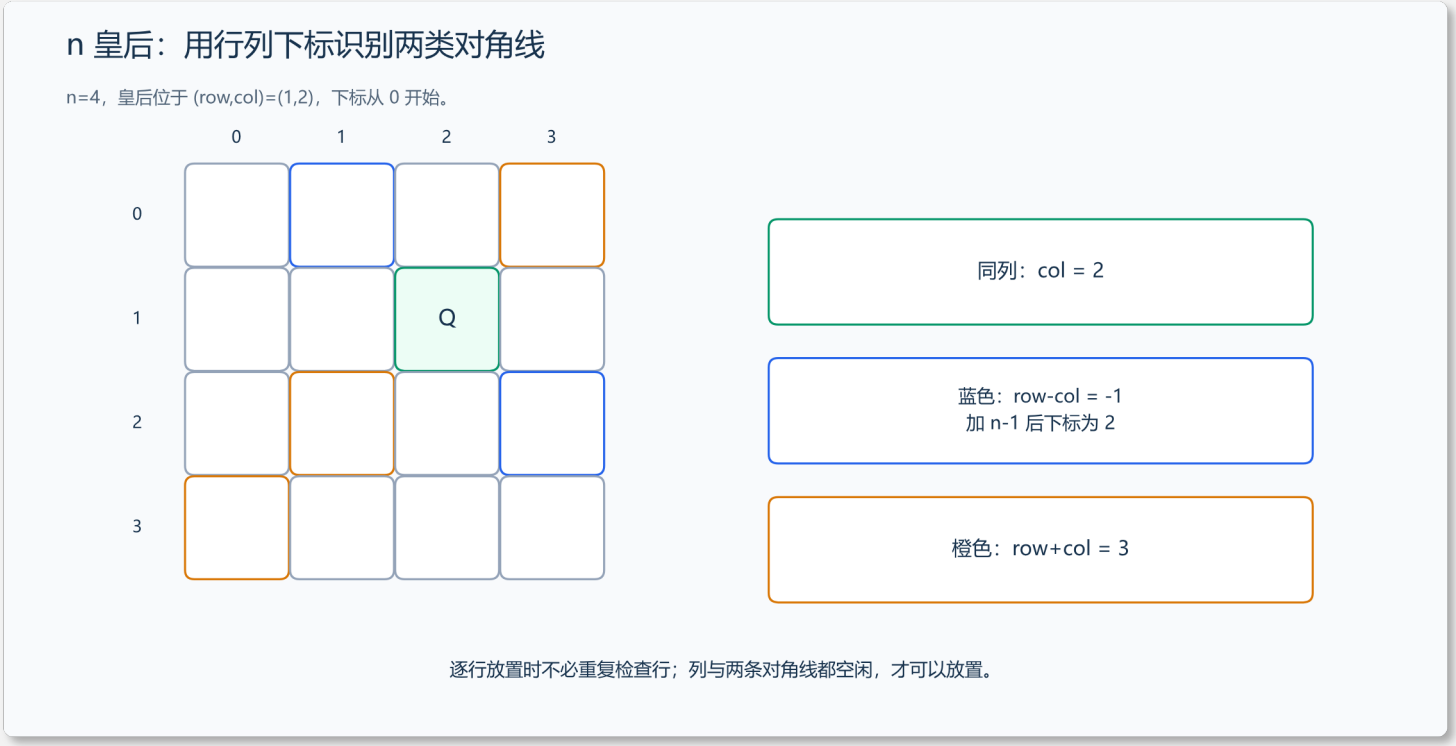
按值传递的局部变量有各自的副本，通常无需手动撤销；共享容器里的 `push_back`、占用标记等修改则需配对恢复。

2.1 n 皇后问题

题目原文（自拟）： 输入 `1 ≤ n ≤ 15`，在 `n×n` 棋盘上放置 `n` 个皇后，使任意两个皇后不在同一行、同一列或同一条对角线上。输出合法放置方案数。棋盘的旋转或镜像若位置不同，按不同方案计数。

思路分析： 一组方案在每行各放一个皇后，所以状态 `dfs(row)` 表示前 `row` 行已经合法放好，正在决定第 `row` 行的列。尝试各列，冲突就跳过；合法则标记列和对角线，进入下一行。递归返回后撤销这三个标记，恢复进入该分支前的状态。

主对角线上相邻格的行、列同时加一，因此 `row-col` 不变；副对角线上一个加一、另一个减一，因此 `row+col` 不变。两者分别用 `row-col+n-1` 和 `row+col` 作数组下标，范围都是 `0…2n-2`。



图中位置 `(1,2)` 的主对角线编号为 `1-2+3=2`，副对角线编号为 `1+2=3`；同色格子共享一个编号。

回溯的三个动作：返回、撤销、尝试下一列

n = 4；行、列从 0 开始。蓝色行正在尝试，橙色 × 是冲突位置。

1 进入 dfs(2)



第 2 行没有合法列
列 0、2 被占；列 1、3 对角冲突

占用列：[0, 2]
主对角：[2, 3]；副对角：[0, 3]

调用栈：dfs(0) → dfs(1) → dfs(2)

2 dfs(2) 返回到 dfs(1)

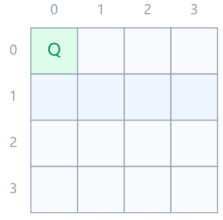


刚返回时，(1,2) 的标记仍为 true
返回动作不会自动清除共享数组

占用列：[0, 2]
主对角：[2, 3]；副对角：[0, 3]

下一条语句：撤销 (1,2) 的占用

3 执行三条 false 赋值

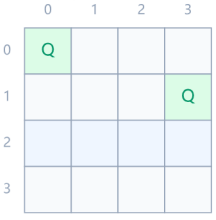


used_col[2] = false
used_diag1[2] = false
used_diag2[3] = false

占用列：[0]
主对角：[3]；副对角：[0]

栈仍为 dfs(0) → dfs(1)；c 即将变成 3

4 同一层试 c = 3

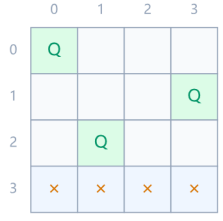


(1,3) 合法，重新写入三个 true
然后调用 dfs(2)，不是调用 dfs(1)

占用列：[0, 3]
主对角：[1, 3]；副对角：[0, 4]

这时第 2 行只能选择列 1

5 dfs(3) 又遇到死路

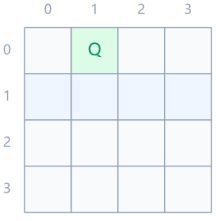


第 3 行也无合法列
逐层返回，并撤销各层刚放的皇后

占用列：[0, 1, 3]
主对角：[1, 3, 4]；副对角：[0, 3, 4]

撤销顺序：(2,1) → (1,3) → (0,0)

6 回到 dfs(0)，试 c = 1



第一行列 0 的整棵分支已搜完
三个占用数组恢复后才试列 1

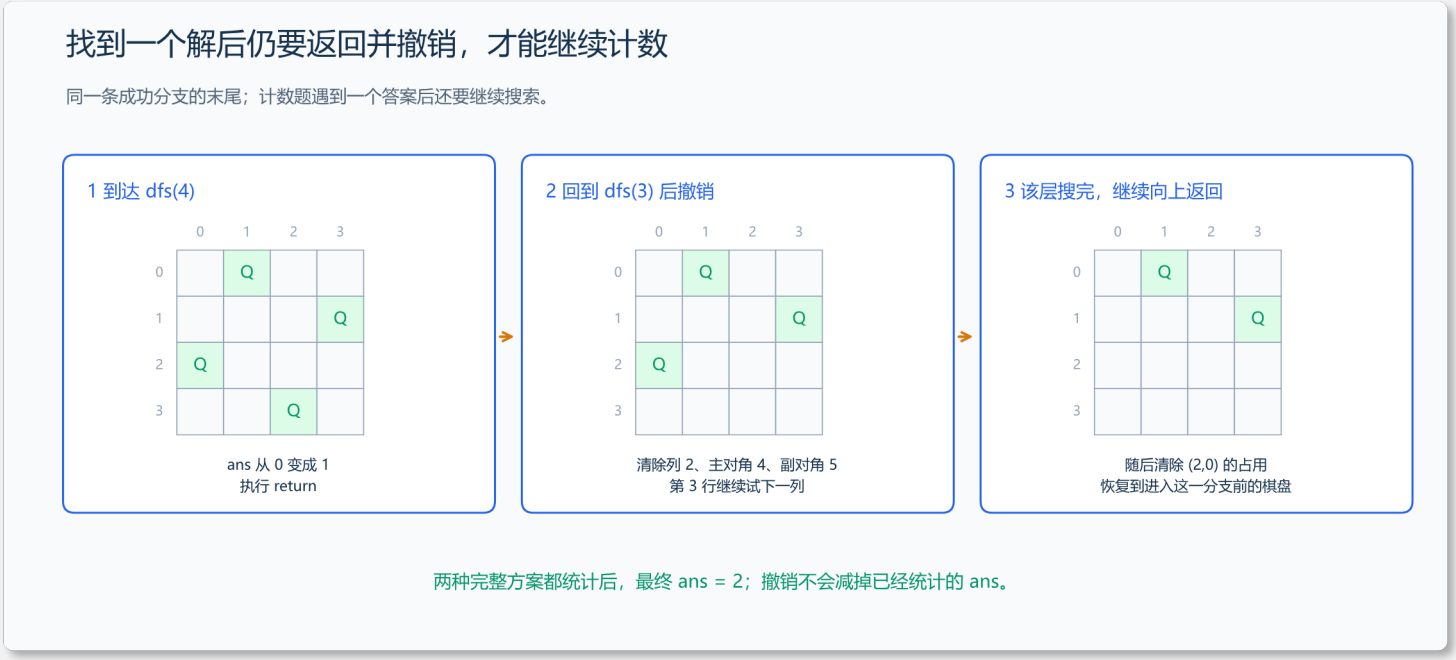
占用列：[1]
主对角：[2]；副对角：[1]

下一条成功分支：列号 1 → 3 → 0 → 2

递归局部变量 row、c 由调用栈保留；共享占用数组需要手动恢复。

按列号从小到大尝试 `n=4`。放在 `(0,0)`、`(1,2)` 后，第 2 行四个位置都冲突，`dfs(2)` 返回到调用它的 `dfs(1)`。此时 `(1,2)` 还在棋盘上：把 `used_col[2]`、`used_diag1[2]`、`used_diag2[3]` 写回 false，就恢复了放置 `(1,2)` 之前的占用状态。随后 `dfs(1)` 的循环把 `c` 从 2 增为 3，尝试 `(1,3)`。

图中第 5 步又出现死路。此时先返回并撤销 `(2,1)`；该层没有别的合法列，再返回并撤销 `(1,3)`；第一列的整个分支都搜完后，才撤销 `(0,0)`，回到第 0 行尝试列 1。每层的 `row、c` 保存在各自的调用栈里，所以能记住“返回后该继续哪个循环”。



沿列号 `[1,3,0,2]` 到达 `dfs(4)`，先让 `ans` 加一，然后返回。返回后仍需撤销第 3 行的皇后，并继续尝试余下位置，才能统计全部方案。`ans` 记录已经找到的答案，不随棋盘撤销而减少。

`col[row]` 只记录当前选择，不参与冲突检查；有效部分是已经放好的前 `row` 行。撤销时清除三个占用数组即可，旧的 `col[row]` 会在下一次合法选择时被覆盖。若用一个动态数组保存当前路径，则要在递归前 `push_back`，返回后 `pop_back`。

代码示例：

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

int n;                // 棋盘大小，皇后数量
int col[20];          // col[row] = 第row行皇后所在的列（0-based）
bool used_col[20];    // used_col[c] = 第c列是否已被占用
bool used_diag1[40]; // used_diag1[d] = 主对角线d是否已被占用
bool used_diag2[40]; // used_diag2[d] = 副对角线d是否已被占用
long long ans = 0;    // 合法解的数量

void dfs(int row) {
    // 终止条件：已成功放置n个皇后
    if (row == n) {
        ans++; // 找到一个合法解
        return;
    }

    // 遍历当前行的所有列
    for (int c = 0; c < n; c++) {
        // 检查列冲突
        if (used_col[c])
            continue;

        // 检查主对角线冲突
        int diag1_index = row - c + (n - 1); // 加(n-1)确保索引非负
```

```cpp
        if (used_diag1[diag1_index])
            continue;

        // 检查副对角线冲突
        int diag2_index = row + c;
        if (used_diag2[diag2_index])
            continue;

        // 当前位置合法，放置皇后
        col[row] = c; // 记录第row行皇后放在第c列

        // 标记占用状态
        used_col[c] = true;
        used_diag1[diag1_index] = true;
        used_diag2[diag2_index] = true;

        // 递归放置下一行的皇后
        dfs(row + 1);

        // 回溯：撤销选择，恢复状态
        used_col[c] = false;
        used_diag1[diag1_index] = false;
        used_diag2[diag2_index] = false;
    }
}

int main() {
    // 输入棋盘大小
    cin >> n;
    if (n < 1 || n > 15)
        return 0; // 教学示例的范围，搜索量增长很快

    // 开始回溯搜索
    dfs(0);

    // 输出结果
    cout << ans << endl;
}
```
cpp

样例与解释： 输入 `4` 输出 `2`，两种方案各行的列号分别是 `[1,3,0,2]` 和 `[2,0,3,1]`。输入 `1` 输出 `1`；`2`、`3` 都输出 `0`。

复杂度分析： 空间为 $O(n)$，包括列标记、两种对角线标记和递归栈。本实现每个状态仍扫描 `n` 列，按不重复列的排列树给出保守时间上界 $O(n \cdot n!)$；对角线剪枝会减少实际访问状态。实际分支数可从图中的冲突跳过过程观察。

扩展： 到达 `row==n` 时，`col` 已是一组完整方案，可以输出它，输出成本另计。位掩码可以批量表示列与对角线占用，并用 `available & -available` 依次取可选位，应使用足够宽的无符号类型；这种方法改善常数。利用镜像对称性时，左半首列的方案乘二，奇数棋盘的中央列需要单独处理。启发式局部搜索可以尝试寻找大棋盘上的一个解，但与完整计数是不同任务。

n	1	2	3	4	5	6	7	8
方案数	1	0	0	2	10	4	40	92

n	9	10	11	12	13	14	15	16
方案数	352	724	2680	14200	73712	365596	2279184	14772512

较大规模的计数仅供对照；教学代码限制 `n≤15`，实际运行时间还取决于剪枝与实现。

练习题： 洛谷 P1219 - 八皇后、洛谷 P1036 - 选数。

3. 深度优先搜索（DFS）

DFS 沿一条路径尽量深入，再返回尝试其他方向。树的前序遍历和前面的回溯都可以用 DFS 描述；图中还需要标记访问状态，避免沿环重复搜索。

题目原文（自拟示例）： 给定一个无向图，顶点编号为 `0` 到 `n-1`，以及起点 `start`。返回从 `start` 能到达的顶点访问顺序。按邻接表中存储的顺序探索邻居，每个顶点只访问一次。要求 `n ≥ 1`，输入顶点编号合法。

思路分析： 访问顶点后立即标记，再递归访问未访问的邻居。提前标记能阻止通过反向边或环再次进入同一顶点。

访问标记维护什么？ `visited[u]=true` 表示这个顶点已经被搜索进入过，后续不再从另一条边重复进入。无向边既存 `u→v` 又存 `v→u`，若进入 `u` 后不立即标记，递归到 `v` 时就可能又回到 `u`，永远绕圈。

对边 `(0,1)、(0,2)、(1,3)`，先进入 `0`，沿第一个邻居进入 `1`；`1` 的邻居 `0` 已访问，跳过，再进入 `3`；`3` 没有新邻居，返回 `1`，再返回 `0`，最后进入 `2`，得到 `0、1、3、2`。返回并不撤销这里的标记，因为目标是让每个顶点只访问一次；这与需要枚举多条路径的回溯题不同。

每个顶点只展开一次，每条邻接表记录只检查一次，因此总量是顶点数加边数。这个访问顺序只是由当前邻接表顺序决定的一种结果，不代表从起点到各顶点的最短路。



对照图使用 6 个顶点、边 `0-1、0-2、1-3、1-4、2-5`，逐步比较 DFS 的调用栈与 BFS 的待处理队列。下面函数的调用样例使用 4 个顶点的图，仍按同一个规则执行，输出见样例。

```cpp
#include <functional>
#include <vector>
using namespace std;

vector<int> dfsOrder(const vector<vector<int>> &graph, int start) {
    vector<bool> visited(graph.size(), false);
    vector<int> order;
    function<void(int)> dfs = [&](int u) {
        // 进入时就标记，防止通过环或反向边重复递归。
        visited[u] = true;
        order.push_back(u);
        for (int v : graph[u]) {
            if (!visited[v]) {
                dfs(v);
            }
        }
    };
    dfs(start);
    return order;
}
```

样例： 无向边为 `(0,1)`、`(0,2)`、`(1,3)`，邻接表按编号递增存储，从 `0` 出发的顺序为 `0 1 3 2`。

代码解释： `visited` 表示是否已经进入该顶点，`order` 保存访问顺序。这个函数只搜索起点所在的连通块；搜索整张图时，还需枚举所有未访问的顶点作为新起点。

时间 $O(V+E)$，额外空间 $O(V)$。很深的图可能使递归栈溢出，此时可以改用显式栈。

练习题： 洛谷 P1605 - 迷宫、洛谷 P5318 - 查找文献。

3.1 完整例题：迷宫终点的可达性

题目原文（自拟）： 给定 `n×m` 的迷宫，`1≤n,m≤30`。每个格子为 `.`（可走）或 `#`（障碍）。从左上角 `(0,0)` 出发，每步可以向上、下、左、右移动一格，不能越界或进入障碍。判断能否到达右下角 `(n-1,m-1)`。第一行输入 `n m`，随后输入 `n` 行、每行 `m` 个字符；可到达输出 `YES`，否则输出 `NO`，起点或终点被封住时输出 `NO`。

把迷宫转成图： 每个可走格子是一个顶点，相邻可走格子之间存在无向边。无需真的列出全部边，用四个方向随时计算邻居即可，这叫隐式建图。状态是当前格子 `(r,c)`，合法判断依次是“未越界、不是障碍、没有访问过”。

为什么不撤销 visited？ 本题只问能否到达。某个格子已展开过，它所有方向的后续可达性就已被检查；从另一条路径再到同一格不会产生新的出口。若题目改成统计不能重复经过格子的所有路径，状态还依赖当前路径中的占用情况，需要回溯撤销标记，不能照搬本程序。



图中的浅蓝格是**当前尚未返回的调用链**，编号是第一次进入格子的次序，两者不能混淆。从 `(0,2)` 向右遇墙，转向下；进入第 8 个格子 `(3,4)` 后，四个方向都没有新邻居，只能返回 `(2,4)`。父调用不会从第一个方向重做，而是接着检查后面的方向，随后向上进入 `(1,4)`、`(0,4)`。上方分支返回完，再退到 `(2,2)`，向下进入第 11 个格子 `(3,2)`，继续探索左侧通路。

阶段	调用怎样移动	哪些状态变化
深入	父调用暂停，合法邻居产生新调用	新格子设为已访问；调用链增长
无新邻居	当前循环结束，函数返回父调用	调用链缩短；已访问标记保留
换方向	父调用从刚才调用语句后继续	试下一个邻居，跳过墙、越界与已访问
搜索完成	起点的所有方向都处理完并返回	调用链为空，所有可达格子的标记仍在

本程序没有“到终点立即结束”的语句，因此找到终点后仍会搜索全部可达格子。最后检查终点是否被标记。调用返回是一种回退，但这里没有撤销 visited；若要求枚举所有简单路径，才需要把占用标记恢复，允许另一条路径重新进入。

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

int main() {
    int n, m;
    cin >> n >> m;
    vector<string> grid(n);
    for (string &row : grid) cin >> row;
    vector<vector<bool>> visited(n, vector<bool>(m, false));
    const int dr[4] = {0, 1, 0, -1}, dc[4] = {1, 0, -1, 0};
    function<void(int, int)> dfs = [&](int r, int c) {
        visited[r][c] = true; // 进入时标记，本题不用撤销。
        for (int d = 0; d < 4; ++d) {
            int nr = r + dr[d], nc = c + dc[d];
            if (nr < 0 || nr ≥ n || nc < 0 || nc ≥ m) continue;
            if (grid[nr][nc] == '#' || visited[nr][nc]) continue;
            dfs(nr, nc); // 先走完这一方向，再返回尝试下一个。
        }
    };
    if (grid[0][0] ≠ '#') dfs(0, 0);
    cout << (visited[n - 1][m - 1] ? "YES" : "NO") << '\n';
}
```

输入：

```text
4 5
...#.
.#.#.
.#...
...#.
```

输出：

```text
YES
```

样例推导： 前 8 次进入构成路径 `(0,0)→(0,1)→(0,2)→(1,2)→(2,2)→(2,3)→(2,4)→(3,4)`，因此输出 YES。随后按图中的返回与换方向继续访问，共有 15 个可达格子。编号表示先访问了哪一格。DFS 沿一个方向深入，找到终点就说明存在路径；后面的 BFS 按移动步数分层，进一步计算最少要走多少步。

代码与复杂度： `dr`、`dc` 分别是行列增量。先检查边界再下标访问，避免越界。每个格子只进入一次并检查四个方向，时间和额外空间为 $O(nm)$；递归深度最坏也为 $O(nm)$，规模很大时使用显式栈。

练习题： 洛谷 P1605 - 迷宫（统计路径，需撤销当前路径标记）、洛谷 P1451 - 求细胞数量（枚举各连通块起点）。

4. 广度优先搜索（BFS）

BFS 按与起点的边数距离逐层扩展，使用队列保存待处理顶点。

题目原文（自拟示例）： 给定一个无权图和起点 `start`，计算起点到每个顶点的最少边数。顶点编号为 `0` 到 `n-1`，`n ≥ 1`；无法到达的顶点返回 `-1`。

思路分析： 起点距离为 `0`，从距离为 `d` 的顶点首次发现其邻居时，邻居距离为 `d+1`。入队时标记，避免同一个顶点反复入队。

为什么第一次发现就能确定最少边数？ 起点先入队，距离为 0。处理距离为 `d` 的顶点时，只会产生距离为 `d+1` 的新顶点，并放到队尾；队列会先处理先前的较短层，所以不会先用更长路径抢先发现某点。按层归纳，首次发现的距离就是最少边数。

仍用前面的无向图，处理起点 `0` 后，队列为 `[1,2]`，两者距离都为 1；处理 `1` 后发现 `3`，队列为 `[2,3]`，`3` 的距离为 2。直到 `2` 处理完，才处理更远一层的 `3`。

发现新邻居时先写入距离，再把它入队；另一个顶点后来遇到这个邻居，读到已有距离，就跳过它。这里 `-1` 兼作未发现标记，合法距离从 0 开始，含义不会混淆。这里每移动一格就增加一步，因此层数正好是路径长度；图论章节会把“经过多少条边”与“累加多少费用”分别展开。

```cpp
#include <queue>
#include <vector>
using namespace std;

vector<int> bfsDistance(const vector<vector<int>> &graph, int start) {
    vector<int> distance(graph.size(), -1);
    queue<int> pending;
    distance[start] = 0;
    pending.push(start);

    while (!pending.empty()) {
        int u = pending.front();
        pending.pop();
        for (int v : graph[u]) {
            // 第一次发现就记录距离并入队；之后不再重复入队。
            if (distance[v] == -1) {
                distance[v] = distance[u] + 1;
                pending.push(v);
            }
        }
    }
    return distance;
}
```



对照图仍用前面的 6 点树，说明两种待处理结构怎样改变。下面 4 点样例中，处理 0 后队列为 `[1,2]`，处理 1 后为 `[2,3]`，队首是 2，所以先处理 2，随后才轮到 3。

样例： 使用上面的四个顶点和三条无向边，从 `0` 出发，距离为 `[0, 1, 1, 2]`。

代码解释： `distance == -1` 同时表示“尚未发现”；首次入队时立即记录距离。队列保证较小距离的顶点先处理，因此首次到达就是最短路径。

时间 $O(V+E)$，额外空间 $O(V)$。普通 BFS 适用于无权图或每条边代价相同的图；一般非负边权最短路需要 Dijkstra 等算法。

练习题： 洛谷 P1162 — 填涂颜色、洛谷 P1141 — 01迷宫（连通块预处理）。

4.1 完整例题：迷宫最少步数

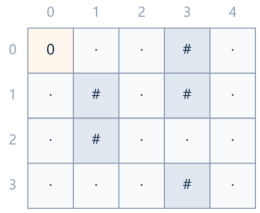
题目原文（自拟）： 给定 `n×m` 的迷宫，`1≤n,m≤500`。字符与移动规则同上：`.` 可走，`#` 不可走，只能向上下左右移动，每次移动代价为一步。从左上角到右下角最少需要几步？第一行输入 `n m`，随后输入 `n` 行迷宫；输出最少步数，不可达时输出 `-1`。起点等于终点且可走时答案为 `0`。

状态与推导： `dist[r][c]` 表示起点到该格的最少步数，`-1` 表示尚未发现。起点为第 0 层，处理第 d 层时发现的未访问邻居属于第 d+1 层。队列把所有第 d 层顶点排在新加入的第 d+1 层之前，因此第一次发现的路径不可能比另一条尚未处理的路径更长。

BFS 迷宫：出队、检查邻居、立即记距离、入队

绿色为刚出队的格子，橙底为本步首次发现；· 表示尚未发现，不是距离 0。

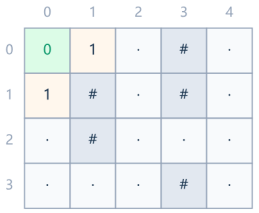
1 起点入队



先写 dist[0][0] = 0
队头是最先处理的格子

队列（左端队头）：[(0,0)]

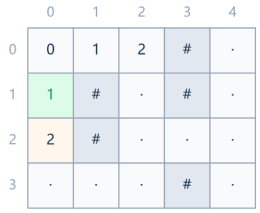
2 出队 (0,0)



先右后下发现两格
两者写距离 1 后立即入队

队列（左端队头）：[(0,1),(1,0)]

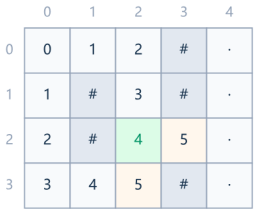
3 第 1 层处理完



(0,1)、(1,0) 都已出队
待处理格子都在第 2 层

队列（左端队头）：[(0,2),(2,0)]

4 出队 (2,2)，距离 4



发现 (2,3)、(3,2)
都先写 5，再加入队尾

队列（左端队头）：[(3,1),(2,3),(3,2)]

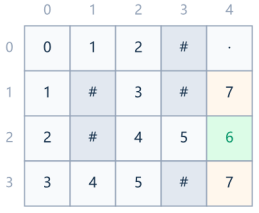
5 出队另一同层格 (3,1)



(3,2) 已有距离 5
跳过，不重复入队

队列（左端队头）：[(2,3),(3,2)]

6 出队 (2,4)，距离 6

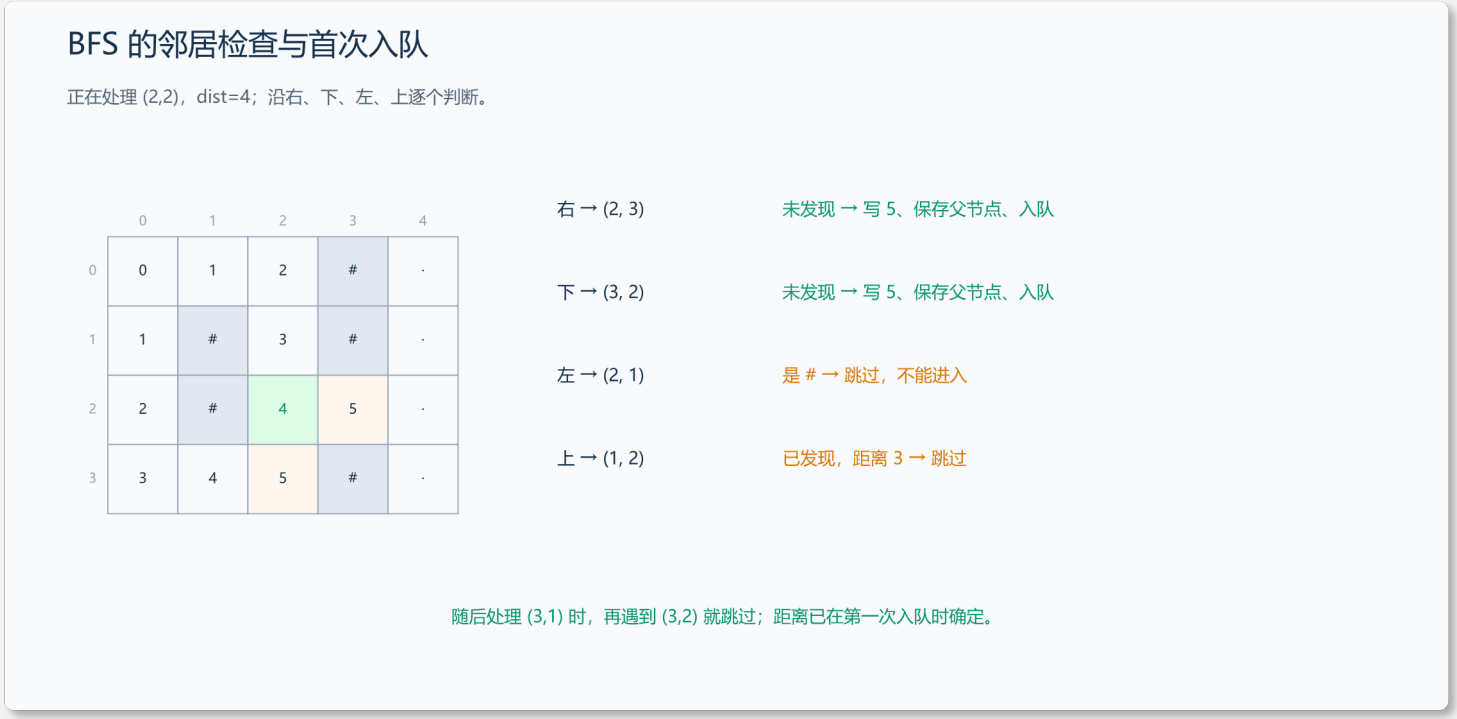


发现终点 (3,4)，写 7
此前各较短层已经处理完

队列（左端队头）：[(3,4),(1,4)]

先入队的较短层先出队；入队时就标记，避免同层两格重复加入同一个邻居。

图中的 `·` 表示尚未发现，并不表示距离 0。每一幅同时记录**刚出队的格子、这一轮新发现的格子、写入距离后的队列**。处理 `(0,0)` 后，队列为 `[(0,1),(1,0)]`；处理 `(0,1)` 后，新增 `(0,2)` 从队尾加入，排在原有 `(1,0)` 后面，所以距离 1 的格子会先于距离 2 的格子处理。



从距离 4 的 `(2,2)` 扩展：右侧 `(2,3)`、下侧 `(3,2)` 尚未发现，都写距离 5、父节点 `(2,2)` 并入队；左侧是墙，跳过；上侧 `(1,2)` 距离已为 3，跳过。轮到另一条路线从 `(3,1)` 看见 `(3,2)` 时，它已在队列里，不能重复加入。到 `(2,4)` 出队时，发现终点的距离为 `6+1=7`。

为什么首次发现就能定最短？ 队列先处理距离 0，再处理距离 1，依此类推。若第一次到 v 的路线长 d+1，却存在更短路线，那么那条短路线的前驱距离至多为 d-1，理应更早出队并发现 v，与“第一次现在才发现”矛盾。这个理由依赖每次移动代价相同；不能照搬到一般带权图。

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

int main() {
    int n, m;
    cin >> n >> m;
    vector<string> grid(n);
    for (string &row : grid) cin >> row;
    vector<vector<int>> dist(n, vector<int>(m, -1));
    vector<vector<pair<int, int>>> parent(n, vector<pair<int, int>>(m, {-1, -1}));
    // parent 保存首次发现来源；起点保留 (-1,-1)，用于结束路径回溯。
    queue<pair<int, int>> q;
    if (grid[0][0] != '#') {
        dist[0][0] = 0; // 距离在入队时确定，同时用作访问标记。
        q.push({0, 0});
    }
    const int dr[4] = {0, 1, 0, -1}, dc[4] = {1, 0, -1, 0};
    while (!q.empty()) {
        auto [r, c] = q.front();
        q.pop();
```

```cpp
        for (int d = 0; d < 4; ++d) {
            int nr = r + dr[d], nc = c + dc[d];
            if (nr < 0 || nr ≥ n || nc < 0 || nc ≥ m) continue;
            if (grid[nr][nc] == '#' || dist[nr][nc] ≠ -1) continue;
            dist[nr][nc] = dist[r][c] + 1; // 本题每条边的代价都是一步。
            parent[nr][nc] = {r, c}; // 与首次发现同时记录，之后不覆盖。
            q.push({nr, nc});
        }
    }
    cout << dist[n - 1][m - 1] << '\n'; // 未发现的终点仍为 -1。
}
```

输入：

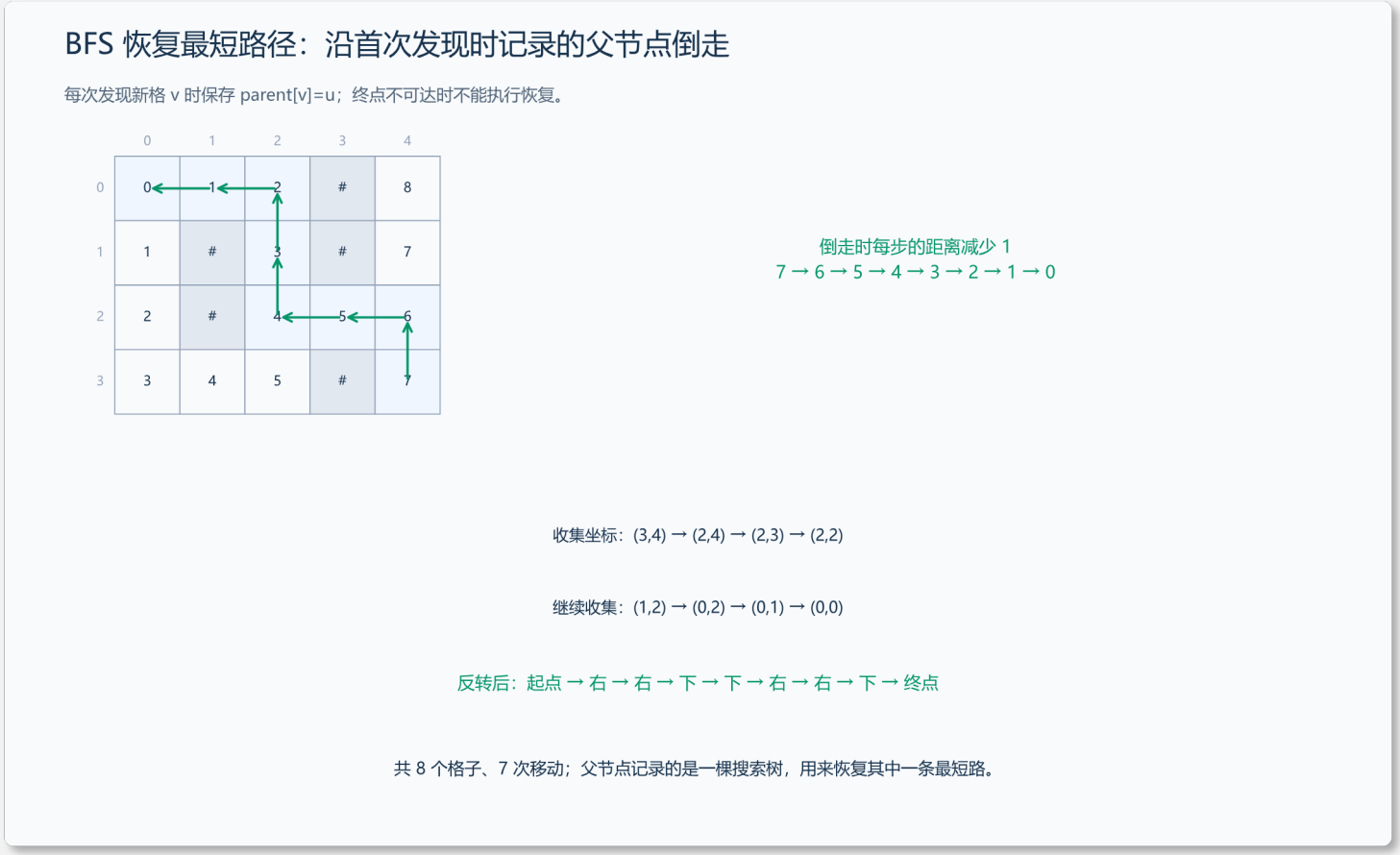
```text
4 5
...#.
.#.#.
.#...
...#.
```

输出：

```text
7
```

样例解释： 上一例中的路径经过 8 个格子，即 7 次移动，距离不是格子数。图中的每个数字都是该格的最少步数；较远格子可以有多条路径到达，但只需第一次入队。

如何恢复路径？ 上面的程序已在首次发现时保存 `parent[nr][nc]=(r,c)`。父节点的距离恰好少 1，因此不断指回去会依次经过第 6、5、…、0 层，而不会形成环。不可达时不要回溯；父指针只记录发现来源，不能当成迷宫中的全部边。



样例先收集 `(3,4)→(2,4)→(2,3)→(2,2)→(1,2)→(0,2)→(0,1)→(0,0)`，共 8 个格子；每一步都读取刚才所在格子的 parent，并更新当前坐标。最后反转才是正向路径，移动数为 `8-1=7`。队列出队顺序包含多个分支，不能直接拿它当成一条路径。

下面是可选的恢复函数；若与完整程序组合，把函数放在 main 前，补上 `<algorithm>`，BFS 结束后调用 `restoreMazePath(parent, dist)`。本题只要求输出距离，所以 main 的输出不变。函数约定两张表形状相同，且 parent 来自上面的 BFS。

```cpp
#include <algorithm>
#include <utility>
#include <vector>
using namespace std;

vector<pair<int, int>> restoreMazePath(
    const vector<vector<pair<int, int>>> &parent, const vector<vector<int>> &dist) {
    if (dist.empty() || dist[0].empty() || dist.back().back() == -1) {
        return {}; // 终点不可达，返回空路径，不能读取它的父节点继续走。
    }
    int r = static_cast<int>(dist.size()) - 1;
    int c = static_cast<int>(dist[0].size()) - 1;
    vector<pair<int, int>> path;
    while (r != -1) {
        path.push_back({r, c}); // 先记录当前格子，沿前驱回到起点后也记录起点。
        auto previous = parent[r][c];
        r = previous.first;
        c = previous.second; // 用保存的前驱同时更新坐标，避免读错位置。
    }
    reverse(path.begin(), path.end()); // 收集顺序是终点到起点，反转成正向。
```

```cpp
    return path;
}
```

恢复只沿一条父链，用时、结果空间均为 $O(L)$，`L` 是路径格子数；单格迷宫返回只含起点的路径，步数为 `0`。

代码解释： 访问标记在入队时写入，防止同层多个格子反复把同一邻居入队。四个方向就是全部合法动作，若题目改成马步或电梯跳跃，保留分层框架并修改邻居产生规则。时间和额外空间均为 $O(nm)$。

练习题： 洛谷 P1443 – 马的遍历（改为八种马步）、洛谷 P1135 – 奇怪的电梯（每层两个可能邻居）。

5. 剪枝：提前结束无用分支

剪枝在搜索中提前结束一些分支。例如当前和已超过目标，剩余数又都非负，继续选择只会让和保持或增加，这一分支便得不到目标和，可以直接返回。

- **可行性剪枝：** 已经违反约束，例如 `n` 皇后中列或对角线发生冲突。
- **最优性剪枝：** 当前方案即使继续也无法优于已知答案。
- **重复状态处理：** 通过访问标记或记忆化，避免重复搜索同一状态。

题目原文（自拟示例）： 给定 `n ≤ 25` 个正整数，以及非负目标值 `target`，判断能否选出一个子集，使元素之和恰好为 `target`。可以选择空集，每个元素至多使用一次；所有累计和在 `long long` 范围内。

思路分析： 对每个元素枚举“选择 / 不选择”。由于元素都是正数，当前和超过目标后不能再降回目标；即使把剩余元素全部选上仍不足目标时，也可以返回失败。



图中所有数都是正数，继续选择只会增大总和。当前总和已经超过目标时，这个分支就可以停止；代码中的判断来自这个实际变化。



再看 `[7,2,3]`、target=8：先选 7，再选 2 得到 9，已经超过目标，立即返回 false；回到上一层，改为不选 2，后续选 3 得到 10，不选则只得 7，两边都失败。最终退回根，不选 7 时，剩余总和只有 5，小于 8，可整枝删除。图中的返回恢复到父调用的局部 sum，而 `suffix` 始终只读；没有共享选择数组就不需要额外撤销。

先定义搜索状态，再证明删除分支安全。 `dfs(i,sum)` 表示前 `i` 个位置的选择已确定，当前和为 `sum`，接下来只决定 `i…n-1`。对位置 `i` 分成选择与不选择两条分支，覆盖所有合法子集；下一层 `i+1` 保证一个位置最多使用一次。

由于所有元素为正，继续选择只能让和变大，因此 `sum>target` 后无论怎样选择都不可能回到目标。另一方面，把剩余全选上也只能达到 `sum+suffix[i]`；若这个上界还小于目标，所有后续子集都无解。两个判断分别排除“已经太大”和“最多仍不够”。

例如 `[2,3,7]`、目标 `5`：选 `2` 后再选 `3` 立即成功；若目标是 `6`，选择 `7` 的相关分支会超过目标，跳过 `7` 后剩余可达到的和又不足，最终失败。目标为 `0` 时空集就成功，所以检查 `sum==target` 要在结束与剪枝判断之前。

这里 `sum` 按值传递，两条分支各用自己的值，无需手工减回去。如果改为共享路径数组并 `push_back`，返回上一层之前要对应 `pop_back`，否则会把前一分支的选择带进后一分支。

```cpp
#include <functional>
#include <vector>
using namespace std;

bool subsetSumPruned(const vector<int> &values, long long target) {
    int n = static_cast<int>(values.size());
```

```cpp
    vector<long long> suffix(n + 1, 0);
    for (int i = n - 1; i ≥ 0; --i) {
        suffix[i] = suffix[i + 1] + values[i];
    }
    function<bool(int, long long)> dfs = [&](int i, long long sum) {
        if (sum == target) {
            return true;
        }
        // 当前和已超目标，或加上所有剩余正数仍不够时，结束这条分支。
        if (i == n || sum > target || sum + suffix[i] < target) {
            return false;
        }
        return dfs(i + 1, sum + values[i]) || dfs(i + 1, sum);
    };
    return dfs(0, 0);
}
```

样例： `[2, 3, 7]`、目标 `5` 返回 `true`；目标 `6` 返回 `false`；目标 `0` 返回 `true`。

代码解释： `suffix[i]` 是从 `i` 开始所有剩余元素的和，构成可达到的和的上界。存在负数时，以上两个剪枝条件不再直接成立。

最坏仍需 $O(2^n)$ 时间，额外空间 $O(n)$；剪枝改善实际搜索量，并不自动改变最坏复杂度。

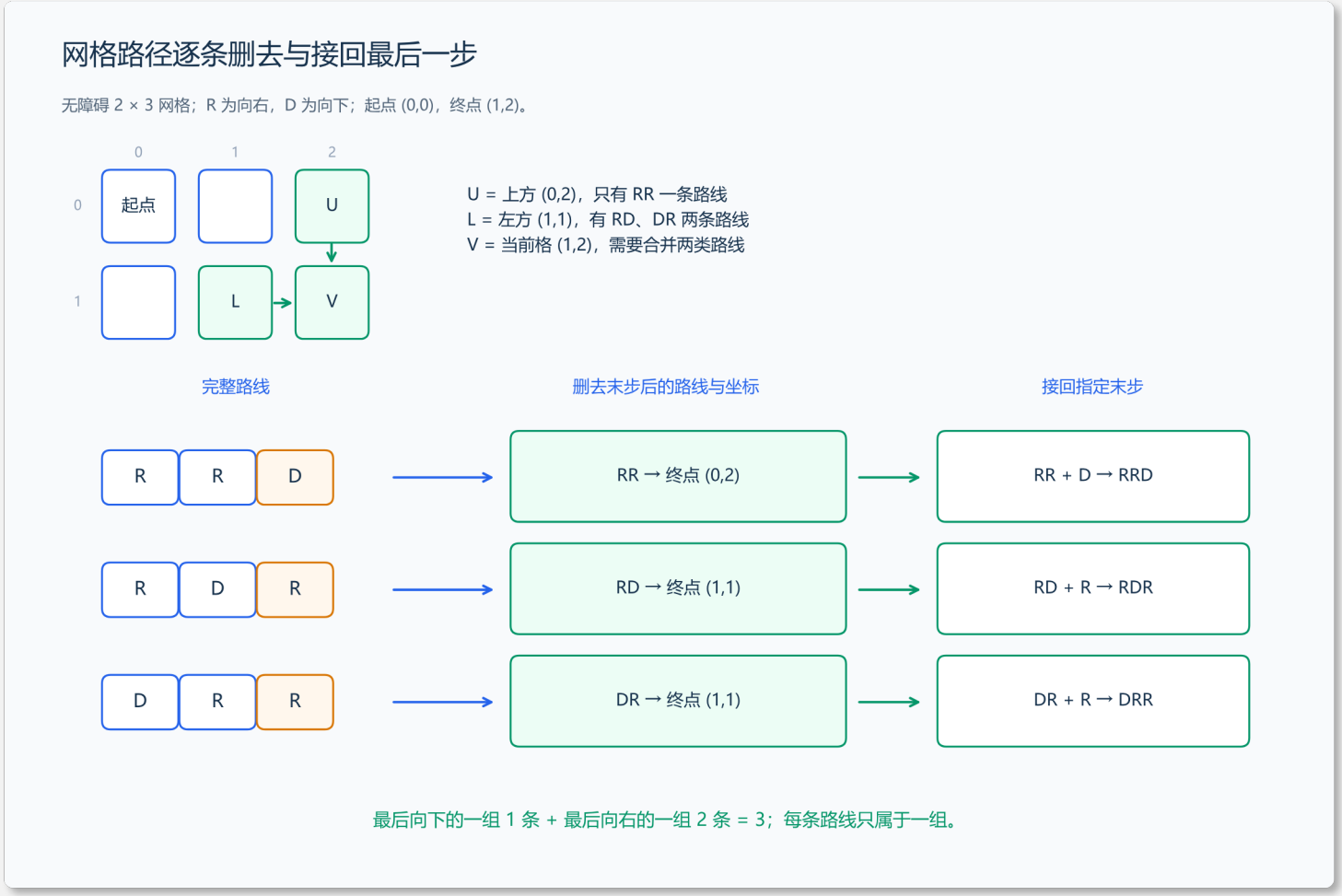
练习题： 洛谷 P1036 - 选数、洛谷 P1049 - 装箱问题（也可用 DP，比较两种思路）。

6. 记忆化搜索：把重复递归变成状态计算

记忆化搜索保留递归的表达方式，同时缓存同一状态的答案。普通 visited 只说明“是否访问过”，memo 还回答“这个状态的计算结果是什么”；如果后续需要这个结果，不能只因访问过就返回 0。

题目原文（自拟）： 有一个 `rows×columns` 的网格，`1≤rows,columns≤500`，只能向右或向下移动，不能越界。部分格子是障碍。求从左上角 `(0,0)` 到右下角 `(rows-1,columns-1)` 的合法路径数量，对 `1000000007` 取模。函数输入是等长行组成的 `vector<vector<bool>> blocked`，`true` 表示障碍；返回一个整数。起点或终点为障碍时返回 0；只有一个可走格子时，不移动计为一条路径。

先拆最后一步： 令 `F(r,c)` 表示到达格子 `(r,c)` 的路径数。普通格的最后一步只能来自 `(r-1,c)` 或 `(r,c-1)`，两类最后一步不同、不重复且覆盖全部路径，所以



先看无障碍 `2×3` 的全部路径 `RRD、RDR、DRR`。第一条最后向下，删去这一步只剩到 `(0,2)` 的 `RR`；后两条最后向右，删去它们的末步得到到 `(1,1)` 的 `RD、DR`。每个前驱接回指定末步恰好还原一条路径，两类没有重叠，所以数量是 `1+2=3`。把终点推广为 `(r,c)`，就得到下面的方程。

$$F(r,c) = \Big(F(r-1,c) + F(r,c-1)\Big) \bmod M.$$

越界或障碍返回 `0`；可走起点返回 `1`。先判断障碍，再判断起点，避免封住的起点得到一个虚构方案。



把递归分成四个动作：查表 → 请求前驱 → 合并 → 写回并返回。查到 `-1` 时，当前调用暂停并进入前驱调用；前驱返回后，当前调用从暂停的位置继续。查到合法答案则立即返回，不再展开。

图用无障碍 `2×3` 网格演示实际执行顺序，缓存中起点仍为 `-1`，因为代码遇到起点时直接返回 1，没有写入起点的缓存。它与“到每格的完整路径数表”含义不同。

执行阶段	当前动作与返回值	缓存变化
1	`F(1,2)` 未命中，暂停并调用上方 `F(0,2)`；后者也未命中，再调用 `F(0,1)`	均为 `-1`，尚未求出答案
2	`F(0,1)` 的上方越界返回 0，左方起点返回 1	写 `memo[0][1]=1`，再返回 1
3	`F(0,2)` 收到左方的 1，上方越界贡献 0	写 `memo[0][2]=1`，再返回 1

执行阶段	当前动作与返回值	缓存变化
4	`F(1,2)` 把上方返回的 1 存在局部变量 `above`，继续调用左方 `F(1,1)`	`F(1,2)` 暂未完成，不提前写缓存
5	`F(1,1)` 请求上方 `F(0,1)`，查到缓存 1，立即返回，不再请求它的前驱	命中 `memo[0][1]`
6	`F(1,1)` 再请求左方 `F(1,0)`；后者的上方起点返回 1、左方越界返回 0	写 `memo[1][0]=1`，再返回 1
7	`F(1,1)` 收到两个 1，合并为 2	写 `memo[1][1]=2`，再返回 2
8	`F(1,2)` 把已保存的上方 1 与左方返回的 2 合并	写 `memo[1][2]=3`，再返回 3

两个前驱返回后，把它们的路径数相加，再写入当前格子的缓存。后面的分支到达这个格子时，读取的就是完整答案。回溯撤销一次选择，记忆化则保留固定状态的计算结果，因此这里的缓存会一直留着供其他分支复用。

重复为什么出现？ 在无障碍 `3×3` 网格中，`F(2,2)` 请求 `F(1,2)` 和 `F(2,1)`，这两者都请求 `F(1,1)`。没有缓存时每个分支都再展开它；有缓存后，同一坐标只计算一次。

什么时候可以缓存？ 未来只受坐标与固定障碍影响，抵达这个状态之前走过的具体路线不影响前驱路径数；若规则禁止重复经过格子并允许四向移动，仅用坐标就不够，因为可用格子取决于此前路线。不能把不同历史压成一个 `memo[r][c]`。

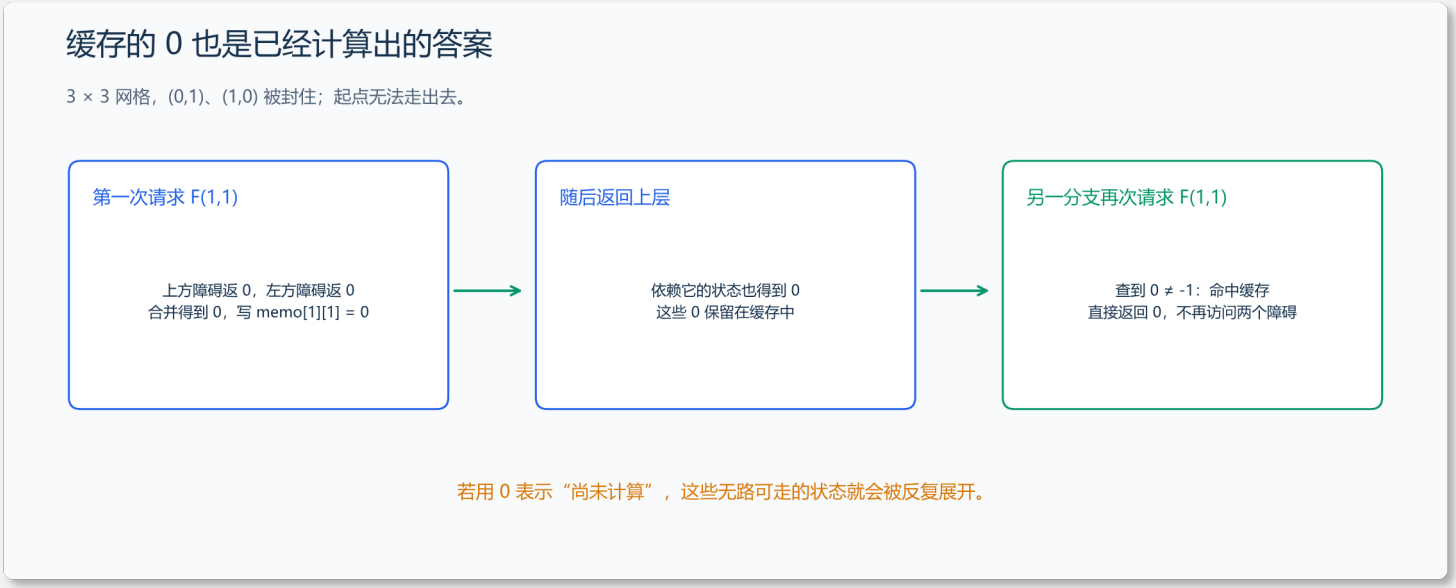
为什么计算会结束？ 请求上方或左方时，r+c 每次减小一，最后到达起点或走出网格，直接返回已有结果。沿这张依赖图计算，会先完成较小坐标和的状态，再把答案传回当前状态。

```cpp
#include <functional>
#include <vector>
using namespace std;

int gridPathsMemo(const vector<vector<bool>> &blocked) {
    constexpr int mod = 1000000007;
    int rows = static_cast<int>(blocked.size());
    int columns = static_cast<int>(blocked[0].size());
    vector<vector<int>> memo(rows, vector<int>(columns, -1));
    function<int(int, int)> dfs = [&](int row, int column) -> int {
        if (row < 0 || column < 0 || blocked[row][column]) {
            return 0; // 越界或障碍，不产生路径。
        }
        if (row == 0 && column == 0) {
            return 1;
        }
        int &answer = memo[row][column];
        if (answer != -1) {
            return answer; // 包括答案为 0 的状态，也不重复计算。
        }
        // 分开调用：先算上方，再算左方，与过程图的展开顺序一致。
        int above = dfs(row - 1, column);
        int left = dfs(row, column - 1);
        answer = static_cast<int>((1LL * above + left) % mod); // 收齐后才写回缓存。
        return answer;
    };
```

```cpp
    return dfs(rows - 1, columns - 1);
}
```

手算样例： 无障碍 `2×3` 网格的表为 `[[1,1,1],[1,2,3]]`，返回 3；若 `(0,1)` 被封住，表为 `[[1,0,0],[1,1,1]]`，返回 1。`-1` 表示尚未计算，合法答案范围为 `0…M-1`；某格算出 0 后也写入缓存：-1 表示未计算，0 表示已经算出没有路径，两种状态由不同数值区分。



再看 `3×3` 网格，封住 `(0,1)` 和 `(1,0)`。第一次计算 `F(1,1)` 时，两个前驱都为 0，所以把 `memo[1][1]` 写成 0。后面的分支再次请求它，条件 `answer≠-1` 成立，直接返回 0。这就是“已经算过，结果为零”，并非“没有算过”。

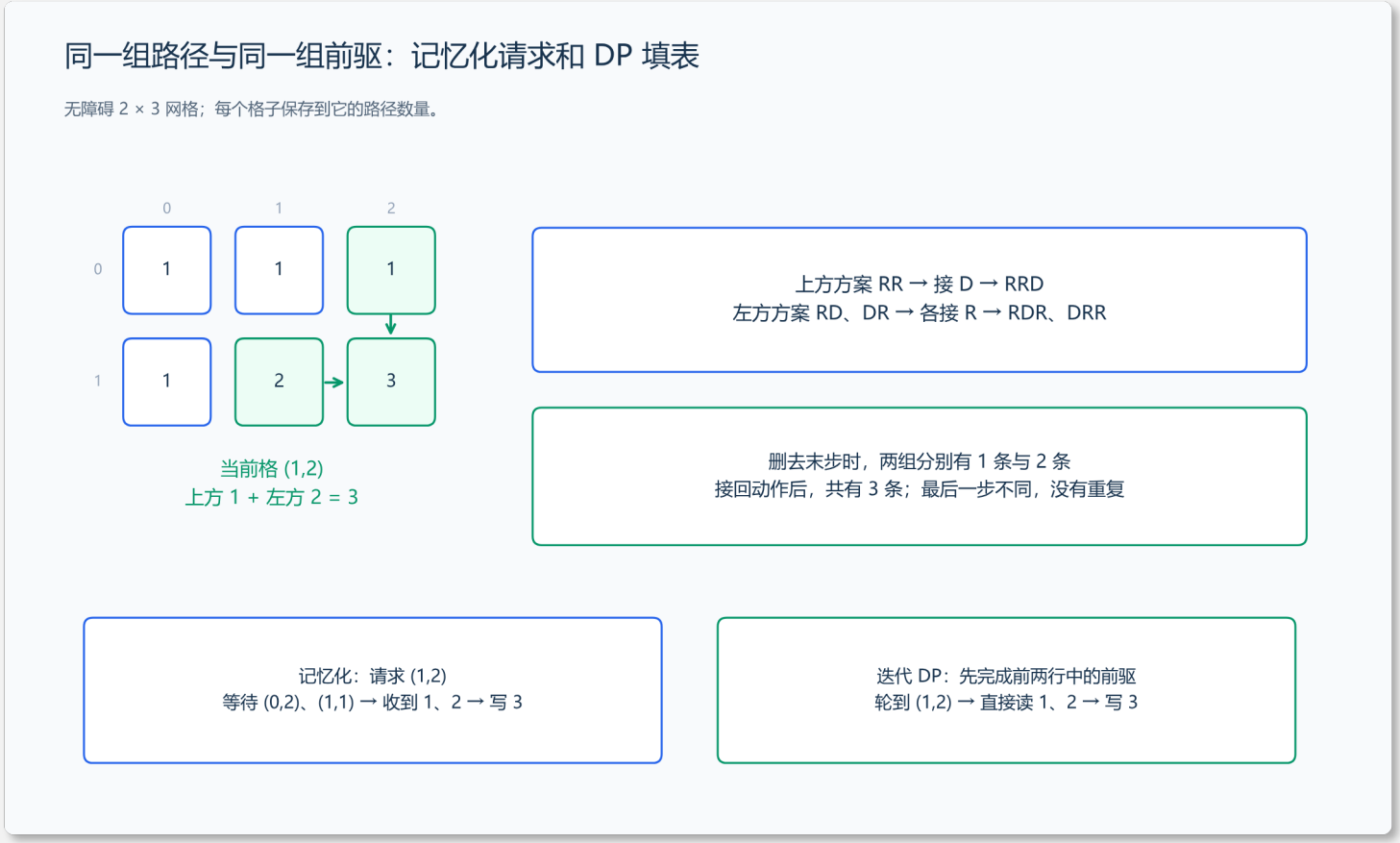
代码与复杂度： `int &answer` 引用缓存格，写入后后续调用直接读它。最多 rows×columns 个状态，每状态两条转移；缓存初始化与总计算时间均为 $O(rows \times columns)$，缓存空间相同，递归深度为 $O(rows + columns)$。很长的网格应改用接下来的迭代 DP。

练习题： 洛谷 P1002 – 过河卒（需把马及其控制范围标成障碍，并按题意处理精确计数）、洛谷 P1434 – 滑雪（高度严格下降使依赖无环）。

记忆化按需请求状态，DP 按依赖顺序计算状态。下面紧接动态规划章节，用同样的状态定义比较两种求解顺序。

（十三） 动态规划（DP）基础

上一节计算网格路径时，`F(1,2)` 要先等到 `F(0,2)` 与 `F(1,1)` 的结果，再把 `1+2` 写入缓存。在迭代表中，把同一个子问题的答案记成 `dp[r][c]`，它仍表示从起点到格子 `(r,c)` 的路径数。现在换一个计算顺序：先写起点，再依次处理上方和左方已经算好的格子。到 `(1,2)` 时两个前驱就在表里，直接读出它们即可。**问题的拆法没有变化，变化的是状态的计算顺序。**



图中每个数字都表示一组路径的数量。`dp[1][1]=2` 保存的是到这个格子的两条路径的总数；以后接上同一个向右动作时，两条路径各产生一条新路径，所以可以直接使用这个数量，不必重新列出两条路线。动态规划（Dynamic Programming，DP）就是把这些子问题整理成状态，再用已求出的状态计算后面的状态。

推导示范：从三条路线得到一次转移

先只考虑无障碍的 `2×3` 网格，`R` 表示右移，`D` 表示下移。到 `(1,2)` 的路线是 `RRD`、`RDR`、`DRR`。把最后一步暂时拿掉，逐条核对：

当前完整路线	删去最后一步	剩下的路线终点	接回的动作
`RRD`	`RR`	上方 `(0,2)`	`D`
`RDR`	`RD`	左方 `(1,1)`	`R`
`DRR`	`DR`	左方 `(1,1)`	`R`

到上方的每条路线补一个 `D`，到左方的每条路线补一个 `R`，就还原了全部路线。两个集合的最后一步不同，一条路线不会在两边各数一次。因此 `dp[1][2]=dp[0][2]+dp[1][1]=1+2=3`。

把具体坐标 `(1,2)` 换成一般坐标 `(r,c)`：上方的行号少 1，左方的列号少 1，得到

$$dp[r][c] = dp[r-1][c] + dp[r][c-1].$$

这里的加号来自“把两个不重叠的路线集合合在一起”。后面求最大价值时比较候选价值，求最少编辑时比较候选费用，合并方式随目标变化。

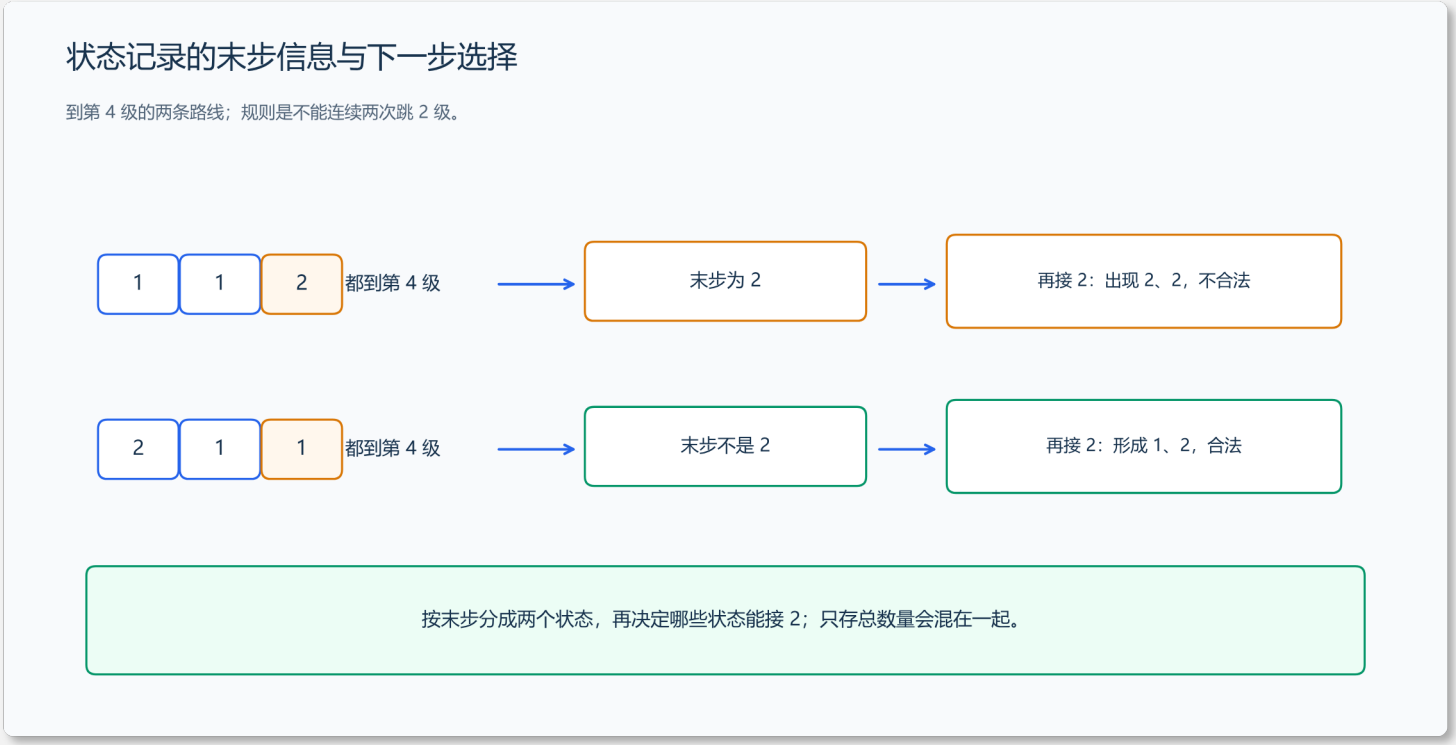
状态中的信息与后续选择

一个状态是对一个子问题的完整说明，不能只写“`dp[i]` 表示答案”。例如：

状态	它保存的答案	它允许的方案
最大子段和的 `dp[i]`	最大和	必须连续，并且必须以位置 `i` 结尾
LIS 的 `dp[i]`	最大长度	可以跳过位置，但必须以位置 `i` 结尾
LCS 的 `dp[i][j]`	最大长度	从两串的前 `i`、`j` 个字符中选择，可以不使用末尾
01 背包的 `dp[i][j]`	最大价值	只使用前 `i` 件物品，总重量不超过 `j`

“固定结尾”使我们知道下一项接在哪里；“前缀长度”限定当前能使用哪些字符；背包还需要容量，因为它决定下一件能否放下。状态里的每个下标都有具体用途。

再看楼梯：到同一层的 `[1,1,2]` 与 `[2,1,1]`，如果下一步可以任意跳 1、2、3 级，两条路线都能接同样的动作，保存总数就够了。若题目改成“不能连续两次跳 2 级”，第一条后面接 2 不合法，第二条却合法。这时就把到第 `i` 级的数量分成“最后跳 2”和“最后没有跳 2”两个状态，分别计算，再按后续动作读取。



这种“记录的信息足以决定后续操作”的性质叫**无后效性**。同一个子问题被不同分支请求，叫**子问题重叠**。求最值时，如果能把方案中的前一部分替换为同状态的更优方案，并且仍能接上当前动作，就可以只保存前驱的最优值，这叫**最优子结构**。后面的例子会直接展示这样的替换。

从方案拆分到程序

推导一个方程时，可以先计算一个具体格子，再把数字替换成变量：

1. 写清当前格子允许哪些完整方案，以及保存数量、最大值还是最小值。

2. 列出几个实际方案，按最后一步、倒数第二位置或当前物品的选择分组。
3. 删去当前动作，读出剩余子问题的下标。剩余方案中仍可使用哪些字符、物品，容量或金额还剩多少，都在这一步确定。
4. 把动作接回去。计数要核对每条方案恰好数一次；最值要核对换成更优前驱后仍然合法。
5. 把每组的结果合并，才写出方程。再用最简单的状态解释初值，沿依赖安排循环，确定最后读取哪个格子。

`dp` 表里存的是子问题的计算结果。方程说明读取哪些结果以及如何合并；循环说明这些结果按什么顺序产生。一维压缩还要区分读取的是上一轮还是本轮，这一点将在背包中用实际数组变化解释。

1. 线性（一维）DP

1.1 斐波那契

题目原文（自拟）： 定义 `F(0)=0、F(1)=1`，`n≥2` 时 `F(n)=F(n-1)+F(n-2)`。给定 `0 ≤ n ≤ 46`，返回 `F(n)`。示例采用常见 32 位 `int` 范围。

思路分析： 递归会多次请求同一项，而递推顺序 `0、1、2、…、n` 使每项都只计算一次。定义 `fib[i]=F(i)`，先填第 0、1 项，再用前两项推出当前项。

下一项只依赖最近两项，因此还能用 `prev2、prev1` 保存它们。先将两个旧值相加得到 cur，再写入 `prev2=prev1、prev1=cur`；这样计算时用的是前两项，更新后保存的又是下一轮所需的最近两项。



从 `(F(3),F(4))=(2,3)` 算出 `F(5)=5` 后，保留的窗口变为 `(3,5)`。

从定义读取依赖。 这里的递推关系来自题目定义，状态 `fib[i]` 对应 F(i)，无需从方案分类另猜一个公式。F(0)=0、F(1)=1 已给定；计算 F(2) 时读取 1 和 0 得到 1，F(3)=1+1=2，F(4)=2+1=3，F(5)=3+2=5。依赖下标始终比 i 小，正序填表合法。题目只要求第 n 项，答案就是 `fib[n]`，而不是把整张表求和。

代码示例： `fibonacci` 保存整张表，`fibonacci_optimized` 只保留两个前驱。

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

int fibonacci(int n) {
    // 边界情况处理
    if (n < 2)
        return n;

    vector<int> fib(n + 1);
    fib[0] = 0, fib[1] = 1;
    for (int i = 2; i ≤ n; ++i)
        fib[i] = fib[i - 1] + fib[i - 2];
    return fib[n];
}

int fibonacci_optimized(int n) {
    // 边界情况处理
    if (n < 2)
        return n;

    int prev2 = 0; // fib[i-2]
    int prev1 = 1; // fib[i-1]
    int cur;       // fib[i]

    // 迭代计算斐波那契数列
    for (int i = 2; i ≤ n; i++) {
        cur = prev1 + prev2; // 计算当前项
        prev2 = prev1;       // 更新prev2为前一项
        prev1 = cur;         // 更新prev1为当前项
    }
    return cur;
}
```

cpp

样例与解释： `n=0` 返回 0；`n=5` 依次得到 `0、1、1、2、3、5`，返回 5；`n=10` 返回 55。

方法	时间	额外空间	需要考虑
朴素递归	指数级，上界 $O(2^n)$	$O(n)$ 栈	重复子问题很多
记忆化递归	$O(n)$	$O(n)$	缓存与递归栈
迭代填表	$O(n)$	$O(n)$	可以查询已经计算的各项
滚动变量	$O(n)$	$O(1)$	不保留完整历史
固定大小矩阵快速幂	$O(\log n)$ 次矩阵操作	$O(1)$	大指数题还需确定取模或大整数规则

复杂度表以机器整数运算为常数代价。超出类型范围时，单纯换成快速幂不会解决精确值溢出；使用大整数后还要计入大整数运算成本。

练习题： 洛谷 P1255 - 数楼梯（斐波那契型递推，完整范围需大整数）。

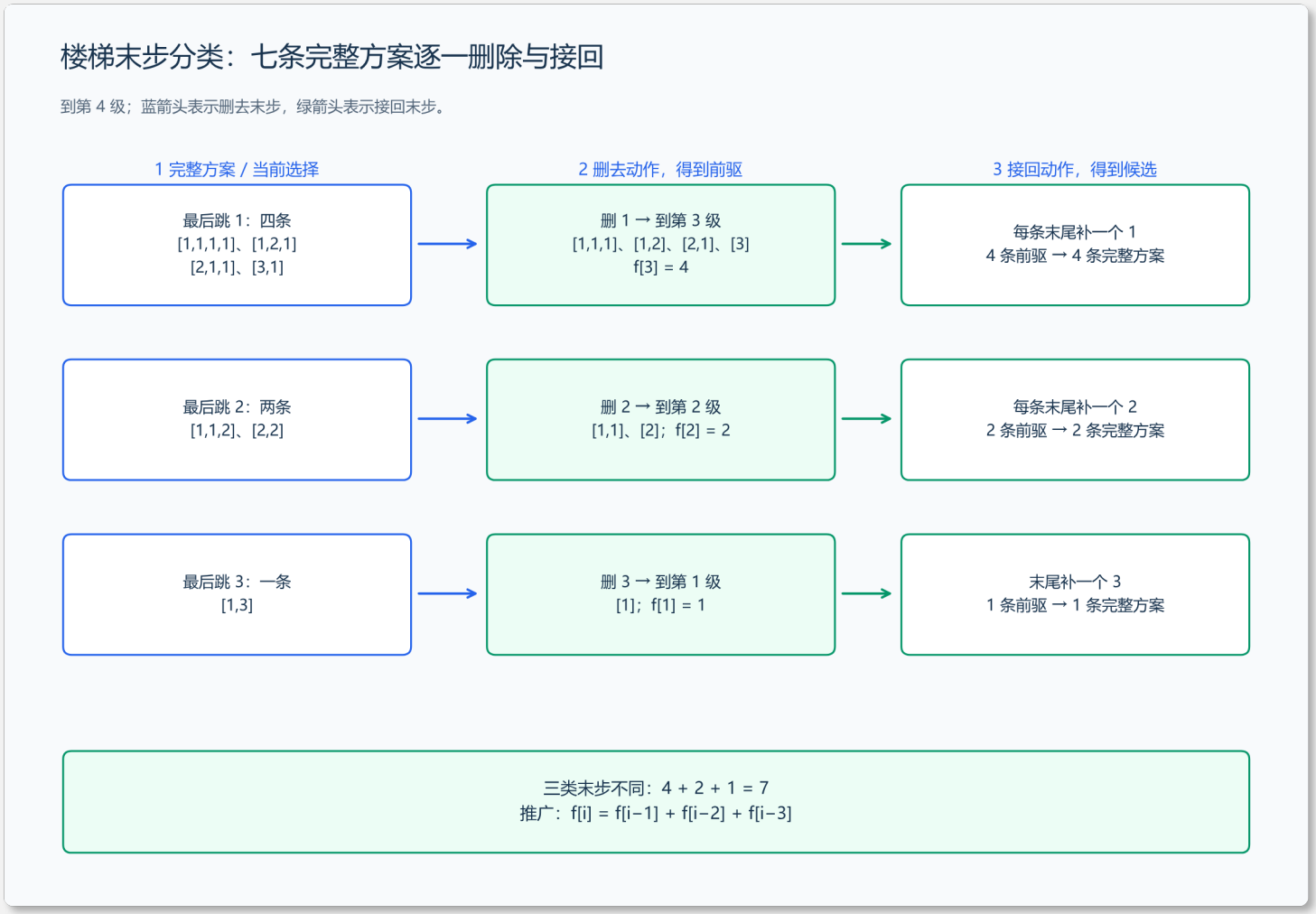
1.2 跳楼梯

题目原文（自拟）： 从第 0 级出发，每次可以上 1、2 或 3 级，求到达第 `n` 级的不同跳跃序列数量。`0 ≤ n ≤ 60`，不跳任何一步的空方案计为到达第 0 级的一种方案，结果在 `long long` 范围内。

思路分析： 先求到第 4 级的方案数。这里的一条方案是完整的跳跃序列，`[1,3]` 与 `[3,1]` 是不同方案。令 `f[i]` 表示从第 0 级恰好到第 `i` 级的序列数量。

把到第 4 级的七条方案按最后一步分组，再删去末步：

最后一步	到第 4 级的完整方案	删去末步后的方案	前驱状态
跳 1	`[1,1,1,1]`、`[1,2,1]`、`[2,1,1]`、`[3,1]`	`[1,1,1]`、`[1,2]`、`[2,1]`、`[3]`	到第 3 级，共 `f[3]=4` 条
跳 2	`[1,1,2]`、`[2,2]`	`[1,1]`、`[2]`	到第 2 级，共 `f[2]=2` 条
跳 3	`[1,3]`	`[1]`	到第 1 级，共 `f[1]=1` 条



每条完整方案都有且只有一个末步，所以三类覆盖七条方案，没有重复。反过来，到第 3 级的任意序列补一个 1，到第 2 级的任意序列补一个 2，到第 1 级的序列补一个 3，都能到第 4 级。于是 `f[4]=4+2+1=7`。

把目标 4 换成 `i`，删去末步 1、2、3 后分别到达 `i-1、i-2、i-3`，同样逐条接回，就得到

$$f[i] = f[i-1] + f[i-2] + f[i-3] \qquad (i \geq 3).$$

`f[0]=1` 表示空序列 `[]`：从地面直接跳 3 时，是给这个空序列补一个 3，因此它要贡献一条方案。到第 1 级只有 `[1]`，到第 2 级有 `[1,1]、[2]`，所以 `f[1]=1、f[2]=2`。继续算第 3 级，得到 `2+1+1=4`，恰好对应 `[1,1,1]、[2,1]、[1,2]、[3]`。

计算第 `i` 项前，前三项都已算好，因此按 `0、1、2、…、n` 填表。题目问到第 `n` 级的数量，读取 `f[n]`。滚动版本只保留最近三个数，先相加得到新值，再将这三个数向前移一项。

代码示例： 两个函数分别保存整张表和最近三个状态。

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

using ll = long long;
ll stairs(int n) {
    // 边界情况处理
    if (n < 0)
        return 0;
    if (n == 0)
        return 1;
    if (n == 1)
        return 1;
    if (n == 2)
        return 2;

    vector<ll> f(n + 1);
    f[0] = 1; // 基准情况：从地面开始算1种方式
    f[1] = 1; // 爬到第1级：1种方式（爬1级）
    f[2] = 2; // 爬到第2级：2种方式（1+1或2）

    for (int i = 3; i ≤ n; ++i)
        f[i] = f[i - 1] + f[i - 2] + f[i - 3]; // 状态转移方程

    return f[n];
}

ll stairs_optimized(int n) {
    if (n < 0)
        return 0;
    if (n == 0)
        return 1;
    if (n == 1)
        return 1;
    if (n == 2)
        return 2;

```

```cpp
    ll a = 1; // f[i-3]
    ll b = 1; // f[i-2]
    ll c = 2; // f[i-1]
    ll d;     // f[i]

    for (int i = 3; i <= n; ++i) {
        d = a + b + c; // 计算当前状态
        a = b;         // 滚动更新
        b = c;
        c = d;
    }

    return c; // 最后c存储的就是f[n]
}
```

cpp

样例与解释： `n=3` 的方案为 `1+1+1`、`1+2`、`2+1`、`3`，共 4 种；`n=4` 共 7 种。两个函数时间均为 $O(n)$，额外空间分别为 $O(n)$、$O(1)$。

状态扩展： 每次只允许 1 或 2 级时，转移改成前两项之和；允许 `1…m` 级时，$f[i]=\sum_{j=1}^{\min(i,m)} f[i-j]$，直接计算为 $O(nm)$，也可以维护前驱窗口和来优化。若禁止连续两次跳 2 级，还需记录“最后一步是否为 2”，只记录总方案数不足以判断后续是否合法。

练习题： 洛谷 P1255 - 数楼梯（题目只允许每次 1 或 2 级，需调整转移）。

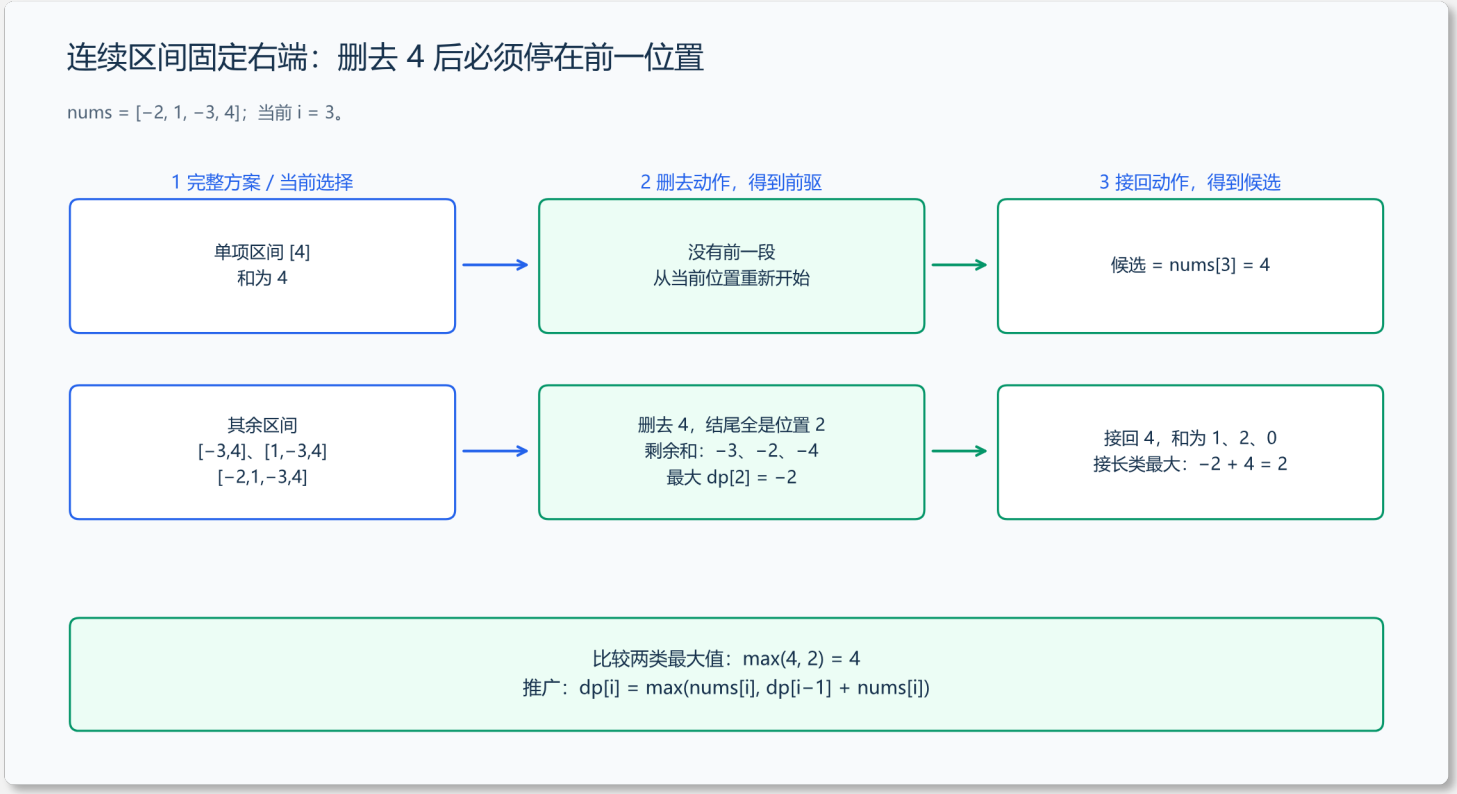
1.3 最大连续子数组和

题目原文（自拟）： 给定非空整数数组，求至少包含一个元素的连续子数组最大和；各累计和在 `long long` 范围内。还可以要求返回一组达到最大和的起止下标，结束下标包含在区间内。函数对空输入另约定返回和 0，下标为 `(-1,-1)`。

思路分析： 先看 `[-2,1,-3,4]`。如果只记录“处理完前几个数的最大和”，最佳区间可能早就结束，接上下一项未必连续。为让接长操作成立，我们先固定右端：`dp[i]` 表示**必须以位置 `i` 结尾**的非空连续子数组最大和。

计算 `dp[3]` 时，四个允许的区间都以 4 结尾。把最后一个 4 拿掉：

当前区间	当前和	去掉 4 后	剩余和
`[4]`	4	没有前一段，单独从位置 3 开始	–
`[-3,4]`	1	`[-3]`，结尾为位置 2	-3
`[1,-3,4]`	2	`[1,-3]`，结尾为位置 2	-2
`[-2,1,-3,4]`	0	`[-2,1,-3]`，结尾为位置 2	-4



长度超过 1 的区间删去末项后，都会落在“以位置 2 结尾”的状态里。三个前驱和是 `-3、-2、-4`，最大为 `dp[2]=-2`。它们接回同一个 4 后变成 `1、2、0`，最大仍来自原先最大的前驱。**给所有候选加上同一个数，不会改变大小顺序**，所以接长这一类只需计算 `dp[2]+4=2`。再与单项 `[4]` 的 4 比较，得到 `dp[3]=4`。

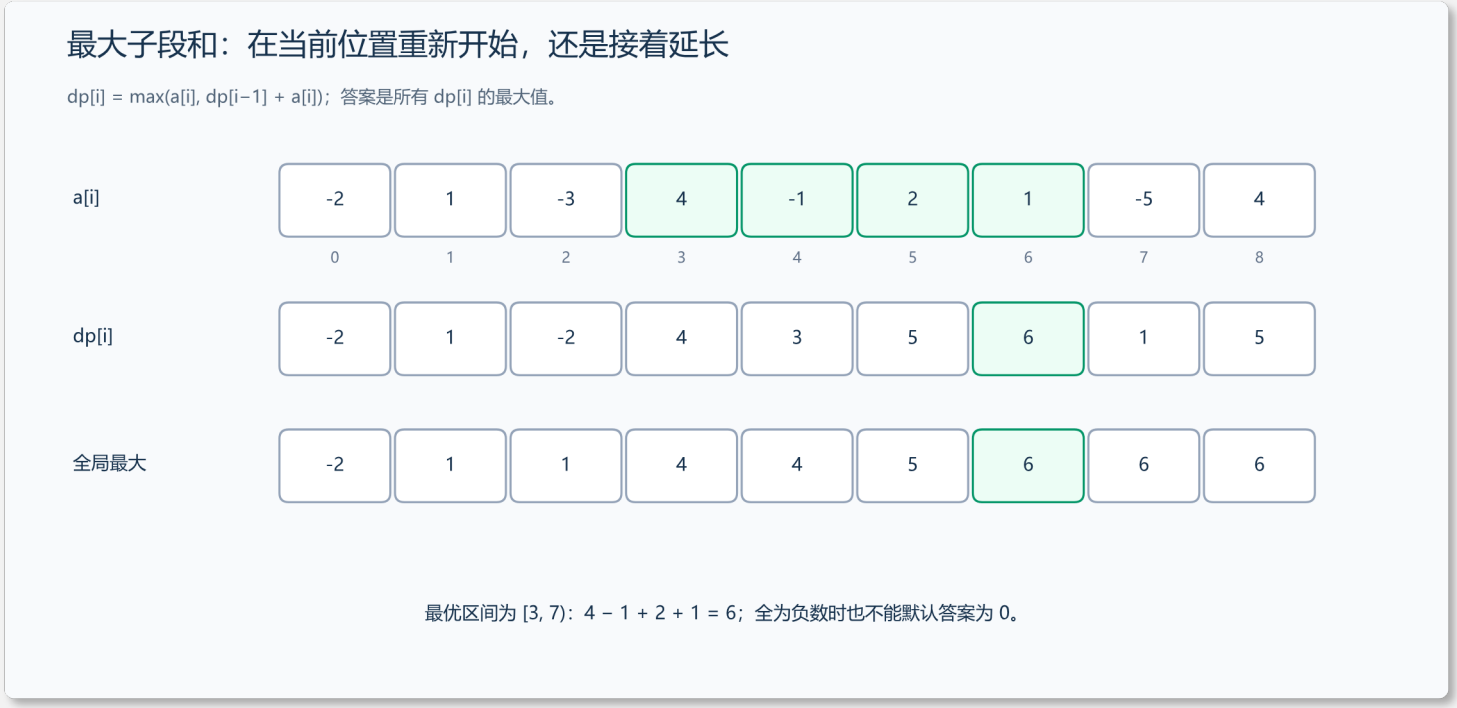
一般位置 `i` 也只有“单项开始”与“接在前一结尾之后”两类。连续性使前驱结尾必须是 `i-1`，不能跳到更早位置：

$$dp[i] = \max\big(nums[i],\ dp[i-1] + nums[i]\big).$$

式中 `nums[i]` 是单项的和；`dp[i-1]+nums[i]` 是接长这一类的最大和；`max` 选择两类中较好的一类。前驱和为负时，带着它会使结果比单项更小；前驱和为非负时，接长不会变差。

位置 i	单项开始	接长前驱	当前结尾 dp[i]	已处理结尾中的最大和
0	-2	第一个元素直接作为一段	-2	-2
1	1	-2+1=-1	1	1
2	-3	1-3=-2	-2	1
3	4	-2+4=2	4	4

初值 `dp[0]=nums[0]`，从左到右计算。题目允许最优区间在任意位置结束，因此答案是所有 `dp[i]` 的最大值。例如 `[4,-5]` 的最后结尾和为 -1，整题答案仍是第一个元素的 4。



只求和时，用 `cur` 保存当前结尾值、`ans` 保存全部已处理结尾的最大值。恢复区间时，单项开始就更新当前左端；只有刷新 `ans` 时，才把当前左右端复制为答案的两端。

1.3.1 保存每个结尾的最优和

先固定右端 i，再比较“从 i 重新开始”和“延长到 i”。`dp[0]` = `nums[0]`，其余按下标递增，最后在所有右端中取最大值。

```cpp
#include <algorithm>
#include <tuple>
#include <vector>
using namespace std;
using ll = long long;
// 先固定右端 i，再比较“从 i 重新开始”和“延长到 i”。
ll max_subarray_sum(vector<ll> const &nums) {
    // 边界情况处理：空数组
    if (nums.empty())
        return 0;

    int n = nums.size();
    vector<ll> dp(n);
    dp[0] = nums[0];    // 以第0个元素结尾的最大和就是它本身
    ll max_sum = dp[0]; // 全局最大和

    for (int i = 1; i < n; ++i) {
        // 状态转移：要么从当前位置重新开始，要么延续前面的子数组
        dp[i] = max(nums[i], dp[i - 1] + nums[i]);
        max_sum = max(max_sum, dp[i]); // 更新全局最大值
    }

    return max_sum;
}
```

1.3.2 只保存当前结尾和

前驱只有 `dp[i-1]`，因此 cur 替代整张表；ans 保存所有已经处理过的结尾中的最好值。不能让 ans 随 cur 的下降而下降。

```cpp
#include <algorithm>
#include <tuple>
#include <vector>
using namespace std;
using ll = long long;
// 前驱只有 dp[i-1]，因此 cur 替代整张表；ans 保存所有已经处理过的结尾中的最好值。
ll max_subarray_sum_optimized(vector<ll> const &nums) {
    if (nums.empty())
        return 0;

    ll cur_max = nums[0];    // 当前以i结尾的最大和
    ll global_max = nums[0]; // 全局最大和

    for (int i = 1; i < nums.size(); ++i) {
        cur_max = max(nums[i], cur_max + nums[i]); // 更新当前最大和
        global_max = max(global_max, cur_max);     // 更新全局最大和
    }

    return global_max;
}
```

1.3.3 同时恢复区间两端

重新开始时更新 cur_start；刷新历史最好值时才复制左右端点。返回值依次是最大和、左端、右端，右端包含在区间内。



当前区间与历史最优区间的端点分别保存。

```cpp
#include <algorithm>
#include <tuple>
#include <vector>
using namespace std;
using ll = long long;
// 重新开始时更新 cur_start；刷新历史最好值时才复制左右端点。
tuple<ll, int, int> max_subarray_with_indices(vector<ll> const &nums) {
    if (nums.empty())
        return {0, -1, -1};

    ll cur_max = nums[0];
    ll global_max = nums[0];
    int cur_start = 0, global_start = 0, global_end = 0;

    for (int i = 1; i < nums.size(); ++i) {
        if (nums[i] > cur_max + nums[i]) {
            // 重新开始新的子数组
            cur_max = nums[i];
            cur_start = i;
        } else {
            // 延续当前子数组
            cur_max = cur_max + nums[i];
        }

        // 更新全局最大值及其边界
        if (cur_max > global_max) {
            global_max = cur_max;
            global_start = cur_start;
            global_end = i;
        }
    }

    return {global_max, global_start, global_end};
}
```

样例与解释： `[-2,1,-3,4,-1,2,1,-5,4]` 的最大和为 6，对应 `[4,-1,2,1]`；`[-3,-1]` 答案为 `-1`，不能用 0 代替非空子数组。所有版本时间为 $O(n)$；表格版本额外空间 $O(n)$，滚动和记录下标的版本为 $O(1)$。

扩展： 最大子矩阵和可枚举上下行，把列和压成一维后应用本算法，`R×C` 矩阵常见代价为 $O(R^2C)$。环形数组分为不跨边界、跨边界两种情况，后者可用总和减去最小子数组和；全为负数时，取出最大的单项，得到非空子数组的答案。最大乘积子数组需同时维护最大、最小乘积，因为负数可能交换二者角色。对长度有限制的最大和，可以用前缀和与单调队列维护合法左端的最小前缀值。

练习题： 洛谷 P1115 – 最大子段和。

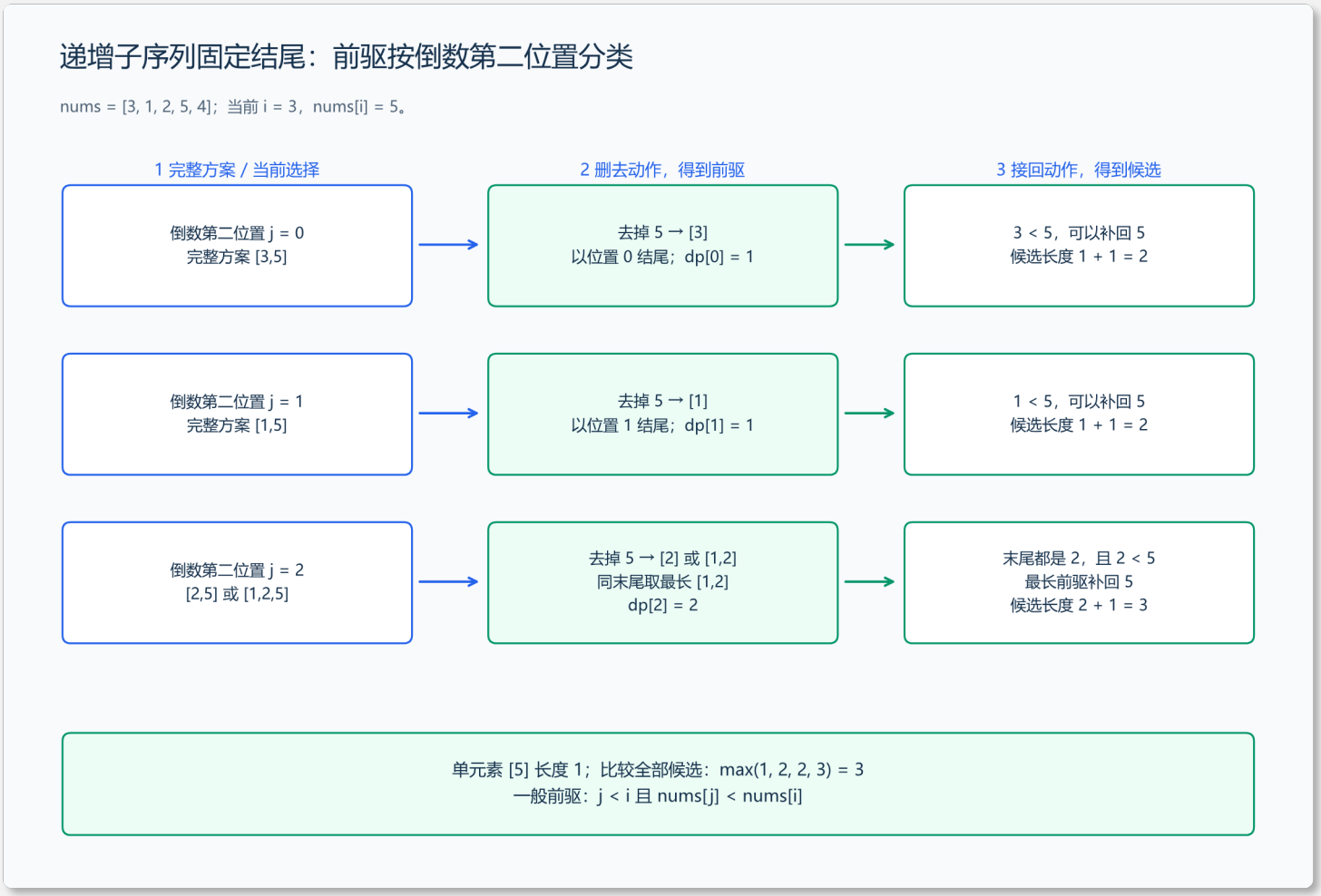
1.4 最长递增子序列

题目原文（自拟）： 给定整数数组，删除若干元素后保留其原有相对顺序，使剩余元素严格递增。求最大长度；空数组返回 0。需要时还应返回一条达到最大长度的子序列。

思路分析： 子序列可以跳过位置。与最大子段和一样，先固定结尾，令 `dp[i]` 表示**最后一个元素必须是 `nums[i]`** 的严格递增子序列最大长度。这样接上一个新元素时，只需检查已知的末尾值。

在 `[3,1,2,5,4]` 中计算以位置 3 的 5 结尾的状态。单个 `[5]` 已是长度 1 的方案。更长的方案删去 5 后，倒数第二位置只能是 0、1 或 2：

倒数第二位置 `j`	以该位置结尾的最长方案	接 5 的判断	接回后的候选
0	`[3]`，长度 1	`3<5`	`[3,5]`，长度 2
1	`[1]`，长度 1	`1<5`	`[1,5]`，长度 2
2	`[1,2]`，长度 2	`2<5`	`[1,2,5]`，长度 3



每条多元素方案都有一个倒数第二位置 `j`。删去末项后，剩余下标仍递增、数值仍严格递增，正是以 `j` 结尾的子问题。即使原方案的前一段较短，也可以换成这个结尾的最长方案：两者末尾都是 `nums[j]`，接 5 的判断完全相同，换长后仍然合法。因此每个 `j` 只需读 `dp[j]`，接回当前项后的长度是 `dp[j]+1`。

把 5 和位置 3 换成 `nums[i]` 和 `i`，前驱必须同时满足“位置在前”和“数值更小”：

$$dp[i] = \max\left(1,\ \max_{\substack{0\le j<i \\ nums[j]<nums[i]}}\left(dp[j]+1\right)\right).$$

外面的 1 是单元素方案；里面的最大值枚举所有能接上的结尾。程序先写 `dp[i]=1`，再逐个检查 `j`，每找到一个合法前驱就比较一次。没有可接前驱时，这个 1 留下。

这组数据的结尾长度为 `[1,1,2,3,3]`。算最后的 4 时，前面的 5 无法接上，但位置 2 的 2 可以，得到 `[1,2,4]`。所有前驱下标都小于 `i`，按 `i` 递增计算；整题读取所有结尾长度的最大值 3。直接算法枚举每个位置前面的全部位置，时间 $O(n^2)$，空间 $O(n)$。

记录使长度变大的来源 `prev[i]=j`，就能从最优结尾沿前驱恢复真实方案；倒着收集之后反转，元素的原下标就按递增排列。

进一步优化时，换一种记录方式：`tails[k]` 保存长度为 `k+1` 的递增子序列能达到的最小末尾。相同长度的两个方案末尾为 2 与 5，之后能接在 5 后的数也能接在 2 后，而 3 还能只接在 2 后。因此保留较小末尾即可。



读入 `x` 时，`lower_bound` 找到第一个 `≥x` 的末尾。此前的末尾都小于 `x`，相应方案可以接长；用 `x` 替换该位置，或在所有末尾都小于 `x` 时追加。



`tails` 各位置保存的是不同长度的代表，彼此可能来自不同方案。图中 `[3,5,6,2,4]` 得到 `[2,4,6]`，这个顺序不是原数组中的子序列，只有它的长度 3 是答案。



恢复内容时沿 `prev` 取原数组元素；求长度的 `tails` 表与保存方案的前驱表作用不同。

1.4.1 枚举所有合法前驱

固定结尾 i，任何长度大于 1 的递增子序列都有一个倒数第二位置 j<i，且 `nums[j]` < `nums[i]`。删去末尾后是

以 j 结尾的子问题；换成该子问题的最优解仍能接上 `nums[i]`，因此枚举所有 j 后取最大不会漏掉最优答案。

```cpp
#include <algorithm>
#include <utility>
#include <vector>
using namespace std;
// 固定结尾 i，任何长度大于 1 的递增子序列都有一个倒数第二位置 j<i，且 nums[j]<nums[i]。
int lengthOfLIS_dp(vector<int> &nums) {
    if (nums.empty())
        return 0;

    int n = nums.size();
    vector<int> dp(n, 1); // 初始化为1，每个元素本身就是一个子序列
    int max_len = 1;

    for (int i = 1; i < n; ++i) {
        // 检查i之前的所有元素
        for (int j = 0; j < i; ++j) {
            if (nums[j] < nums[i]) {
                dp[i] = max(dp[i], dp[j] + 1);
            }
        }
        max_len = max(max_len, dp[i]);
    }

    return max_len;
}
```

1.4.2 用最小末尾与二分优化

相同长度只留下最小末尾，因为它能接受较大末尾可以接受的所有后续值。lower_bound 找第一个不小于 x 的位置，替换末尾或扩展长度；不是把 x 插入原数组排序。

```cpp
#include <algorithm>
#include <utility>
#include <vector>
using namespace std;
// 相同长度只留下最小末尾，因为它能接受较大末尾可以接受的所有后续值。
int lengthOfLIS_greedy(vector<int> &nums) {
    if (nums.empty())
        return 0;

    vector<int> tails; // tails[k] = 长度为k+1的递增子序列的最小末尾元素

    for (int x : nums) {
        // 二分查找第一个≥x的元素位置
        auto it = lower_bound(tails.begin(), tails.end(), x);

        if (it == tails.end()) {
            // x比所有元素都大，延长最长子序列
            tails.push_back(x);
        } else {
            // 替换该位置的元素，保持tails的最小末尾性质
            *it = x;
```

```cpp
        }
    }

    return tails.size(); // tails的长度就是LIS长度
}
```

1.4.3 沿前驱恢复真实子序列

`prev[i]` 记录让 `dp[i]` 严格变大的前驱下标。选出最大长度的结尾，再沿 prev 倒走并反转；各元素的下标因此严格递增。tails 来自不同方案，不能直接当作恢复结果。

```cpp
#include <algorithm>
#include <utility>
#include <vector>
using namespace std;
// prev[i] 记录让 dp[i] 严格变大的前驱下标。
pair<int, vector<int>> getLIS_sequence(vector<int> &nums) {
    int n = nums.size();
    if (n == 0)
        return {0, {}};

    vector<int> dp(n, 1);
    vector<int> prev(n, -1); // 记录前驱索引，用于回溯

    int max_len = 1, max_idx = 0;

    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < i; ++j) {
            if (nums[j] < nums[i] && dp[j] + 1 > dp[i]) {
                dp[i] = dp[j] + 1;
                prev[i] = j; // 记录前驱
            }
        }
        if (dp[i] > max_len) {
            max_len = dp[i];
            max_idx = i;
        }
    }

    // 回溯构建LIS序列
    vector<int> lis;
    int cur = max_idx;
    while (cur != -1) {
        lis.push_back(nums[cur]);
        cur = prev[cur];
    }
    reverse(lis.begin(), lis.end());

    return {max_len, lis};
}
```

样例与解释： `[10,9,2,5,3,7,101,18]` 的最长长度为 4，例如 `[2,3,7,101]`；`[2,2,2]` 的严格递增长度为

1。二分版本时间 $O(n\log n)$、额外空间 $O(n)$；这里恢复方案的函数采用平方 DP。

扩展： 允许相等时，条件改为 `≤`，二分改用 `upper_bound`。严格递减可以直接使用相反比较关系；若用取负转化，要避免最小有符号整数取负溢出。二维严格嵌套可按宽度升序、同宽高度降序排序，再对高度求 LIS，避免同宽元素互相嵌套。统计最长方案数时另设 `count`：发现更长方案则覆盖数量，长度相同时累加数量；还需说明按下标方案还是按数值方案去重。

练习题： 洛谷 P1020 – 导弹拦截、洛谷 P1439 – 两个排列的最长公共子序列（进阶：排列映射到 LIS）。

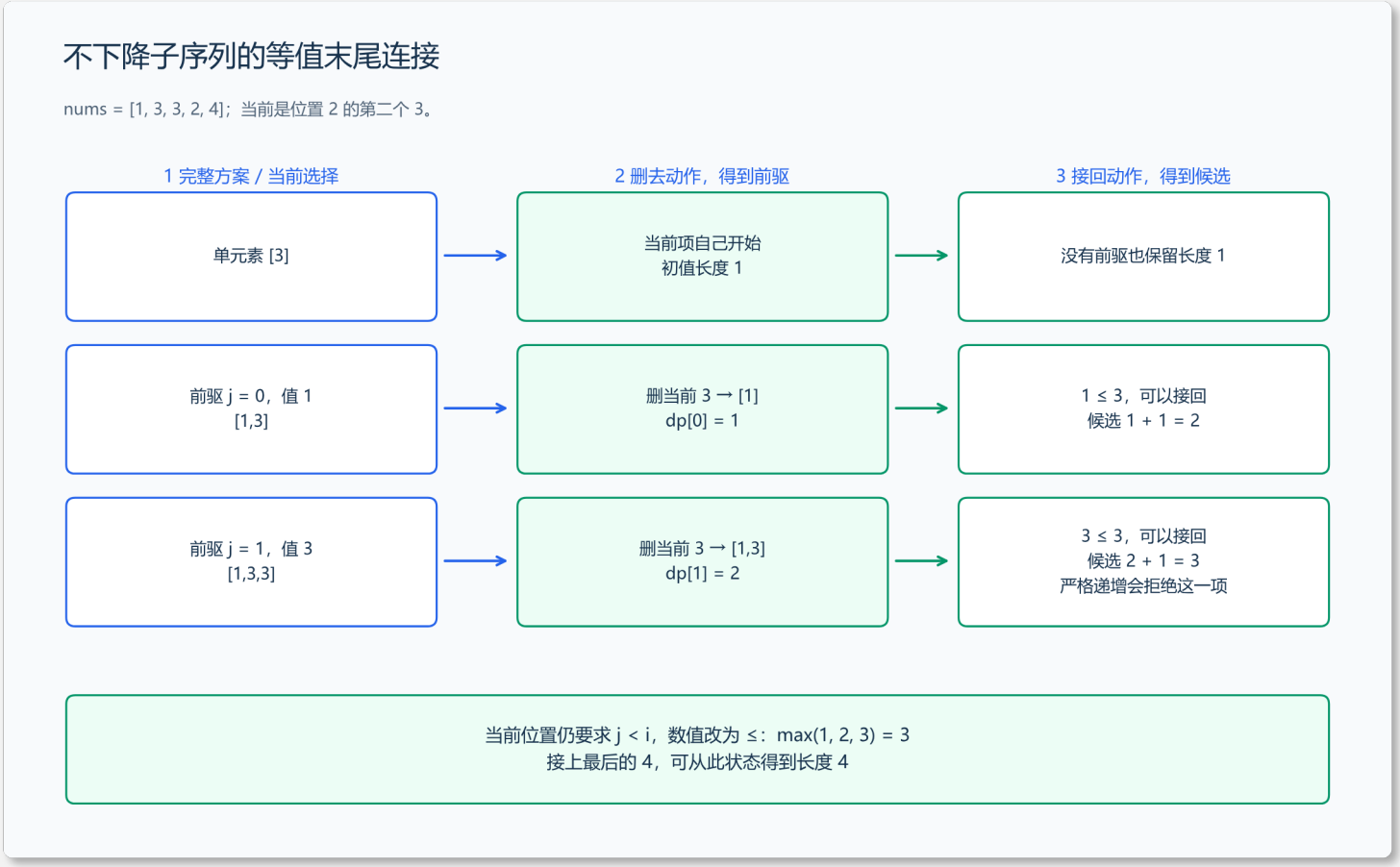
1.5 最长不下降子序列

题目原文（自拟）： 给定整数数组，求保持原相对顺序且后一项大于等于前一项的子序列最大长度；空数组返回 0。需要时返回一条对应序列。扩展任务是求后一项小于等于前一项的最长不递增子序列。

思路分析： 在 `[1,3,3,2,4]` 中，两个 3 位于不同位置。状态仍定义为“必须以位置 `i` 结尾的最大长度”，但这次后一项可以等于前一项。

计算位置 2 的第二个 3 时，先考虑单项 `[3]`，长度为 1。然后逐个删去候选方案的末项，检查它前面可以留下哪个结尾：

前驱位置	最优前驱	严格递增时	不下降时
0，值 1	`[1]`，长度 1	`1<3`，候选长度 2	`1≤3`，候选长度 2
1，值 3	`[1,3]`，长度 2	`3<3` 不成立	`3≤3`，接回得到 `[1,3,3]`，长度 3

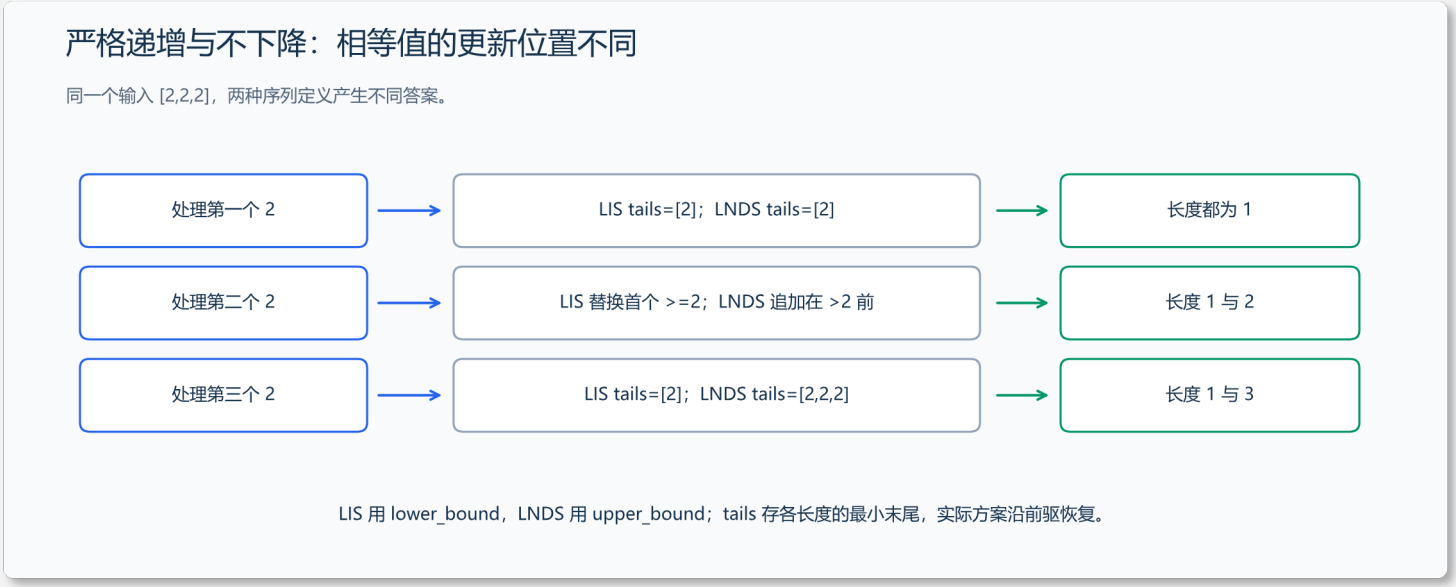


位置依然要求 `j<i`，所以不能把当前元素接在自身后面；数值判断改为 `nums[j]≤nums[i]`。把任意合法前驱换成同一结尾的最长方案，末尾值不变，当前项仍能接上。因此

$$dp[i] = \max\left(1,\ \max_{\substack{0\le j<i \\ nums[j]\le nums[i]}}\left(dp[j]+1\right)\right).$$

这组数据按结尾计算得到 `[1,2,3,2,4]`。第二个 3 的状态为 3；到值 2 时，两个 3 都不能接，只有 1 能接，当前长度降为 2；最后的 4 能接在第二个 3 后，得到 `[1,3,3,4]`，长度 4。各位置从左到右计算，答案仍取全部结尾的最大值。

二分版本也要保留等值连接。`upper_bound` 找第一个 `>x` 的末尾；等于 `x` 的末尾位于它前面，对应的方案可以补上当前 `x`，长度增加一。



输入 `[2,2,2]`，严格递增只能取一个 2；不下降可以按三个原下标全部取出。二分方法分别得到长度 1 与 3，变化来自是否允许等值末尾接长。

1.5.1 推导允许相等的前驱

状态仍是“以 i 结尾的最大长度”。仅把前驱条件改为 `nums[j]≤nums[i]`，因此 `dp[i]` 取所有合法前驱 `dp[j]` +1 与单个元素长度 1 中的最大值，最大值的范围是所有满足 j<i 且值不大的 j；单个元素提供初值 1。

```cpp
#include <algorithm>
#include <functional>
#include <utility>
#include <vector>
using namespace std;
// 状态仍是“以 i 结尾的最大长度”。
int lengthOfLNDS_dp(vector<int> &nums) {
    int n = nums.size();
    if (n == 0)
        return 0;

    vector<int> dp(n, 1); // 每个元素自身就是一个子序列
    int max_len = 1;

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < i; j++) {
            // 注意：这里是 ≤ 而不是 <
            if (nums[j] ≤ nums[i]) {
                dp[i] = max(dp[i], dp[j] + 1);
            }
        }
        max_len = max(max_len, dp[i]);
    }

    return max_len;
}
```

cpp

1.5.2 upper_bound 保留相等元素

upper_bound 找第一个大于 x 的末尾。已有末尾恰等于 x 时，允许接上 x，使更长的不下降子序列成立；不能使用会覆盖相等末尾的 lower_bound。

```cpp
#include <algorithm>
#include <functional>
#include <utility>
#include <vector>
using namespace std;
// upper_bound 找第一个大于 x 的末尾。
int lengthOfLNDS_optimal(vector<int> &nums) {
    if (nums.empty())
        return 0;

    vector<int> tails; // tails[k] = 长度为k+1的不下降子序列的最小末尾元素

    for (int x : nums) {
        // 关键区别：使用upper_bound而不是lower_bound
        // 查找第一个 > x 的位置，允许相等元素接在后面
        auto it = upper_bound(tails.begin(), tails.end(), x);

        if (it == tails.end()) {
            // x不小于tails中任何元素，可以延长最长子序列
            tails.push_back(x);
        } else {
            // 替换该位置，保持tails的最小末尾性质
            *it = x;
        }
    }

    return tails.size(); // tails的长度就是LNDS长度
}
```

cpp

1.5.3 不递增子序列的降序末尾

改成后一项不大于前一项时，tails 按降序维护。相同长度的末尾越大，后面能接的值越多；upper_bound 配合 greater() 找第一个小于 x 的位置，相等时仍可延长。



降序末尾数组按相反的比较关系替换。

```cpp
#include <algorithm>
#include <functional>
#include <utility>
#include <vector>
using namespace std;
// 改成后一项不大于前一项时，tails 按降序维护。
int lengthOfLNIS_optimal(vector<int> &nums) {
    // 直接使用降序比较，避免最小有符号整数取负时溢出。

    // 方法2: 在降序 tails 中使用 upper_bound，允许相等元素继续延长
    if (nums.empty())
        return 0;

    vector<int> tails;

    for (int x : nums) {
        // 查找第一个 < x 的位置（降序规则下的 upper_bound）
        // greater<int>() 定义降序比较关系。
        auto it = upper_bound(tails.begin(), tails.end(), x, greater<int>());

        if (it == tails.end()) {
            tails.push_back(x);
        } else {
            *it = x;
        }
    }

    return tails.size();
}
```

1.5.4 恢复不下降子序列

平方 DP 保存每个结尾及前驱，判定 `nums[j]≤nums[i]` **后才允许连接。相同长度的候选无需覆盖已保存前驱，**

因此返回一条确定的最优序列。



沿前驱下标收集后反转，保留两个相等的 3。

```cpp
#include <algorithm>
#include <functional>
#include <utility>
#include <vector>
using namespace std;
// 平方 DP 保存每个结尾及前驱，判定 nums[j]≤nums[i] 后才允许连接。
pair<int, vector<int>> getLNDS_sequence(vector<int> &nums) {
    int n = nums.size();
    if (n == 0)
        return {0, {}};

    vector<int> dp(n, 1);
    vector<int> prev(n, -1); // 前驱索引，用于回溯

    int max_len = 1, max_idx = 0;

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < i; j++) {
            // 注意是 ≤ 而不是 <
            if (nums[j] ≤ nums[i] && dp[j] + 1 > dp[i]) {
                dp[i] = dp[j] + 1;
                prev[i] = j;
            }
        }
        if (dp[i] > max_len) {
            max_len = dp[i];
            max_idx = i;
        }
    }
```

```cpp

    // 回溯构建序列
    vector<int> lnds;
    int cur = max_idx;
    while (cur != -1) {
        lnds.push_back(nums[cur]);
        cur = prev[cur];
    }
    reverse(lnds.begin(), lnds.end());

    return {max_len, lnds};
}
```

样例与解释： `[1,3,3,2,4]` 的严格递增长度为 3，例如 `[1,2,4]`；不下降长度为 4，例如 `[1,3,3,4]`。不递增扩展对 `[3,3,2]` 返回 3，在降序 `tails` 中用 `greater<int>()` 配合 `upper_bound` 找第一个 `<x` 的位置。

复杂度分析： 平方和恢复方案的实现为 $O(n^2)$ 时间，二分实现为 $O(n \log n)$ 时间；额外空间均为 $O(n)$。恢复方式与 LIS 相同，只改变前驱合法条件；`tails` 仍不能直接当作完整答案序列。

练习题： 洛谷 P1020 - 导弹拦截（同时练习非递增与严格递增）。

2. 二维 DP

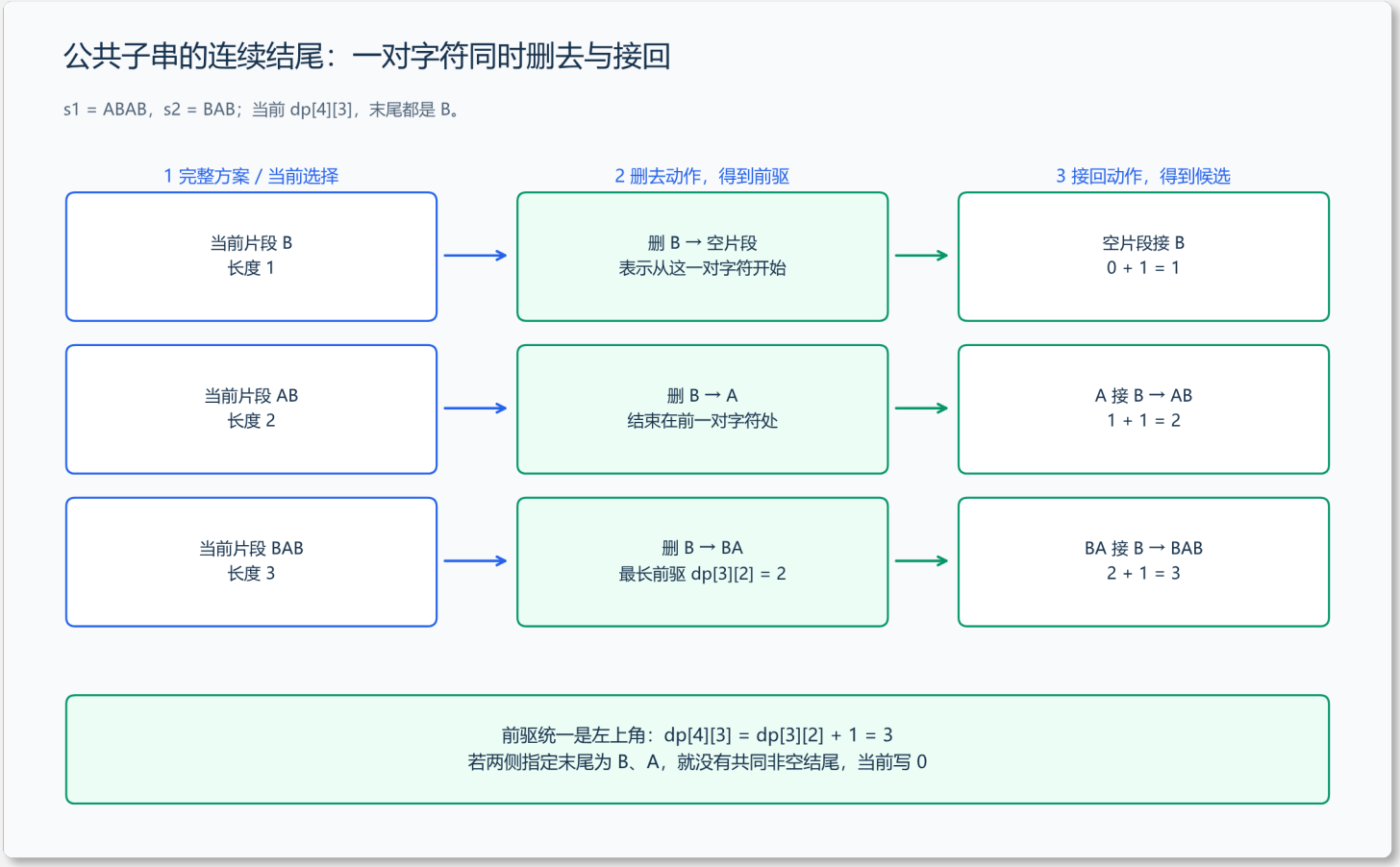
2.1 最长公共子串

题目原文（自拟）： 给定两个按单字节字符处理的字符串 `s1`、`s2`，求它们共同拥有的最长连续子串长度；空串时返回 0。也可以要求返回一个达到该长度的子串，没有公共字符时返回空串。

思路分析： 子串要求连续。令 `dp[i][j]` 表示公共子串**恰好结束在** `s1[i-1]` 和 `s2[j-1]` 时的最大长度。`i`、`j` 是前缀长度，末尾字符的数组下标才是 `i-1`、`j-1`；第 0 行、列用于表示一侧为空的情况。

例如 `s1=ABAB`、`s2=BAB`，计算 `dp[4][3]` 时指定的末尾都是 B。当前允许的非空公共子串有 `B`、`AB`、`BAB`。删去共同的最后一个 B：

当前片段	删去 B 后	剩余片段的结尾	接回后的长度
`B`	空片段	当前片段从末尾开始	`0+1=1`
`AB`	`A`	两串前一对字符处	`1+1=2`
`BAB`	`BA`	两串前一对字符处	`2+1=3`



剩余的非空片段必须在 `s1[2]、s2[1]` 处结束：连续性使它不能跳过中间字符。因此前驱坐标是 `(3,2)`。该位置的最长片段为 BA，`dp[3][2]=2`。所有候选都接回同一个 B，最长前驱接回后仍最长，得到 `dp[4][3]=2+1=3`。

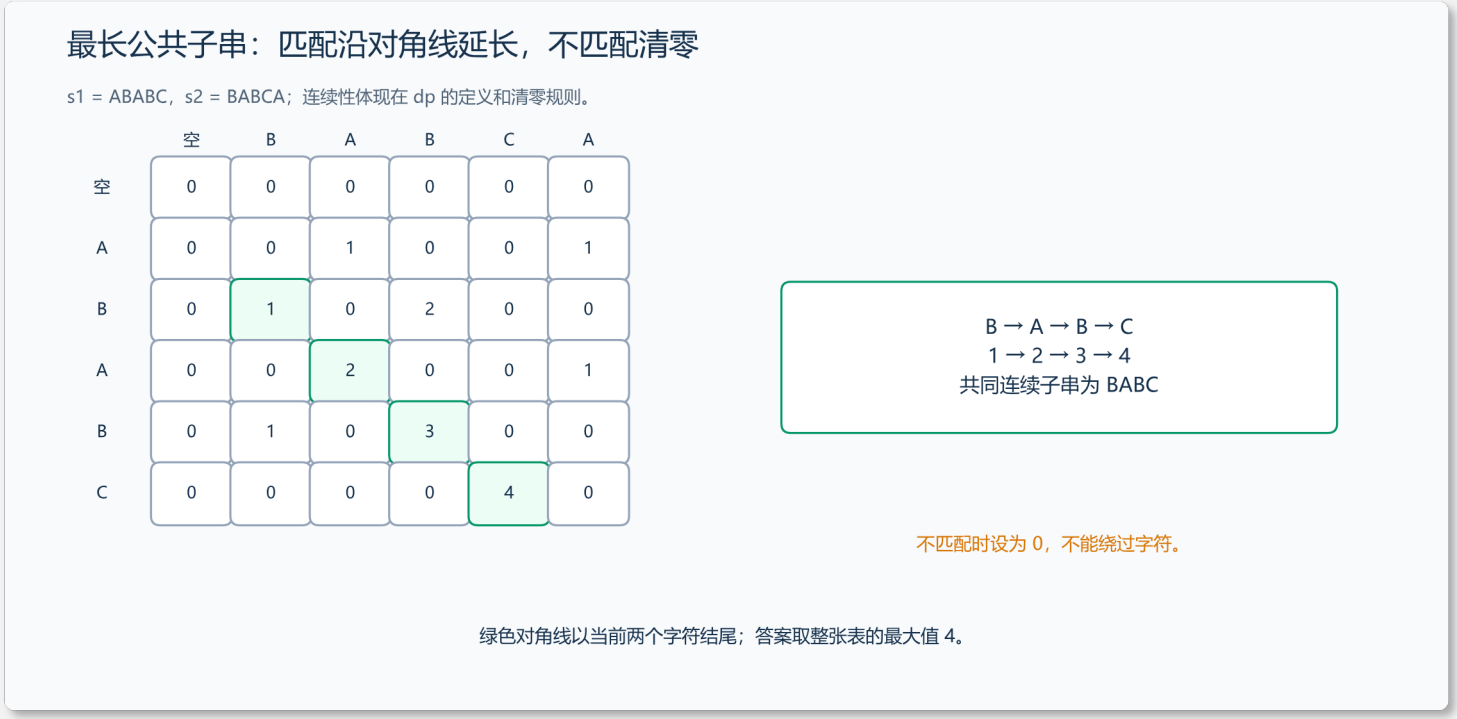
若当前末尾不相等，例如 `s1=AB、s2=BA` 的 `dp[2][2]`，指定末尾为 B 与 A。同一非空公共片段的最后字符不可能同时是 B 和 A，所以这里写 0。上方或左方的片段停在别处，拿来填当前格就改变了指定结尾。

将这两种情况推广为

$$dp[i][j] = \begin{cases} dp[i-1][j-1]+1, & s1[i-1] = s2[j-1], \\ 0, & s1[i-1] \neq s2[j-1]. \end{cases}$$

左上角表示两个结尾各退一位；`+1` 表示接回一对相同字符。左上角为 0 时，`0+1` 恰好从当前这一对字符开始一段新子串。

第 0 行、列都为 0，因为空串没有非空片段。逐行填写时，前一行的左上角已经算好。整题允许最佳子串在任意位置结束，答案取全表最大值。例如 `ab、ax` 在 `(1,1)` 得到 1，右下角为 0，答案仍是公共片段 a 的长度 1。



图中 `ABABC`、`BABCA` 的对角线对应 BABC。长度从 1、2、3 增长到 4，依次表示这段公共子串的四个连续结尾。滚动版本只保留相邻两行，但每个不匹配位置仍要覆盖为 0。

2.1.1 填写全部结尾状态

末尾相等时把两个前驱结尾的公共子串接长一位；末尾不同就不存在该结尾的非空公共子串，于是写入 0。每个匹配候选都要更新全局最大长度。

```cpp
#include <algorithm>
#include <string>
#include <vector>
using namespace std;
// 末尾相等时把两个前驱结尾的公共子串接长一位；末尾不同就不存在该结尾的非空公共子串，于是写入 0。
int longestCommonSubstring_length(string s1, string s2) {
    int m = s1.length();
    int n = s2.length();
    if (m == 0 || n == 0)
        return 0;

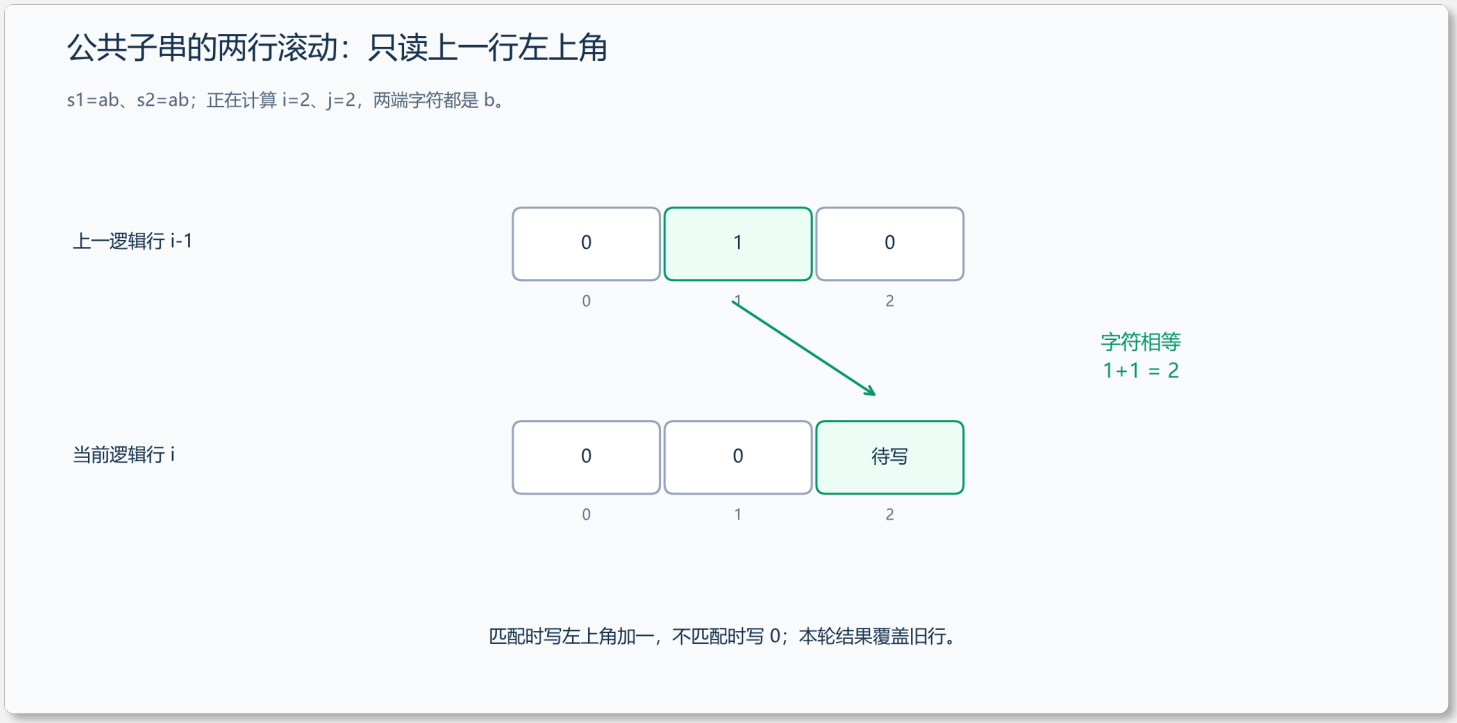
    // dp[i][j]: 以s1[i-1]和s2[j-1]结尾的最长公共子串长度
    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));
    int max_len = 0; // 记录最大长度

    for (int i = 1; i ≤ m; i++) {
        for (int j = 1; j ≤ n; j++) {
            if (s1[i - 1] == s2[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1] + 1; // 字符相等，长度+1
                max_len = max(max_len, dp[i][j]);
            } else {
                dp[i][j] = 0; // 字符不相等，重新开始（子串要求连续）
            }
        }
    }
```

```cpp
26
27      return max_len;
28  }
```

2.1.2 两行滚动与不匹配置零

当前状态只读上一行左上角。i%2 选择当前物理行，(i-1)%2 选择上一行；字符相同时写入左上角加一，不相同时写入 0。两种分支都覆盖本轮位置，因此这一行被再次使用时不会保留两轮前的值。



公共子串滚动时只读取上一逻辑行的左上角。

```cpp
#include <algorithm>
#include <string>
#include <vector>
using namespace std;
// 当前状态只读上一行左上角。
int longestCommonSubstring_space_optimized(string s1, string s2) {
    int m = s1.length();
    int n = s2.length();
    if (m == 0 || n == 0)
        return 0;

    // 列方向 s2 保持较短，减少滚动数组宽度
    if (n > m) {
        swap(s1, s2);
        swap(m, n);
    }

    // 只使用两行数组
    vector<vector<int>> dp(2, vector<int>(n + 1, 0));
    int max_len = 0;

    for (int i = 1; i ≤ m; i++) {
        for (int j = 1; j ≤ n; j++) {
```

```cpp
            if (s1[i - 1] == s2[j - 1]) {
                dp[i % 2][j] = dp[(i - 1) % 2][j - 1] + 1;
                max_len = max(max_len, dp[i % 2][j]);
            } else {
                dp[i % 2][j] = 0; // 关键：子串要求连续
            }
        }
    }

    return max_len;
}
```
cpp

2.1.3 根据结束位置取出内容

刷新最大长度时记录前缀长度 end_pos，它是半开区间的右边界，不是最后字符的下标。起点为 end_pos-max_len；substr 的第二参数是长度。



半开区间 [1,5) 对应字符串 BABC。

```cpp
#include <algorithm>
#include <string>
#include <vector>
using namespace std;
// 刷新最大长度时记录前缀长度 end_pos，它是半开区间的右边界，不是最后字符的下标。
string longestCommonSubstring_content(string s1, string s2) {
    int m = s1.length();
    int n = s2.length();
    if (m == 0 || n == 0)
        return "";

    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));
    int max_len = 0;
    int end_pos = 0; // 记录最长子串在s1中的结束位置

    for (int i = 1; i ≤ m; i++) {
        for (int j = 1; j ≤ n; j++) {
```

```cpp
            if (s1[i - 1] == s2[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1] + 1;
                if (dp[i][j] > max_len) {
                    max_len = dp[i][j];
                    end_pos = i; // s1中的结束位置
                }
            } else {
                dp[i][j] = 0;
            }
        }
    }

    // 根据结束位置和长度提取子串
    if (max_len == 0)
        return "";
    return s1.substr(end_pos - max_len, max_len);
}
```

代码解释： 表格版本保存所有结尾状态；滚动版本只读上一行的左上角，并在本行不匹配的位置写 0。把较短字符串放在列方向，可减少数组宽度。返回内容时记录最大长度 `max_len` 及其在 `s1` 中的半开结束位置 `end_pos`，取 `substr(end_pos-max_len,max_len)`。

样例与复杂度： `ABABC`、`BABCA` 返回长度 4、内容 `BABC`；`abcde`、`ace` 的最长公共子串长度为 1，而公共子序列长度为 3。时间为 $O(mn)$。完整表额外空间 $O(mn)$，滚动表为 $O(\min(m,n))$；这些函数按值接收字符串，还存在 $O(m+n)$ 的参数副本空间，需要与 DP 辅助表的口径区分。

扩展： 多字符串的公共子串可从最短字符串的候选子串开始验证，进一步可学习后缀数组或后缀自动机；复杂度要结合总长度和验证方式计算。允许若干不匹配时需要把剩余或已用次数加入状态。循环字符串可以通过拼接处理跨尾首片段；只取原字符串长度内的片段，就对应沿环至多绕一圈。需要所有最优位置时可收集最大长度对应的结束位置。

练习题： 洛谷 P2679 - 子串（进阶：子串状态扩展，不是直接套最长公共子串模板）。

2.2 最长公共子序列

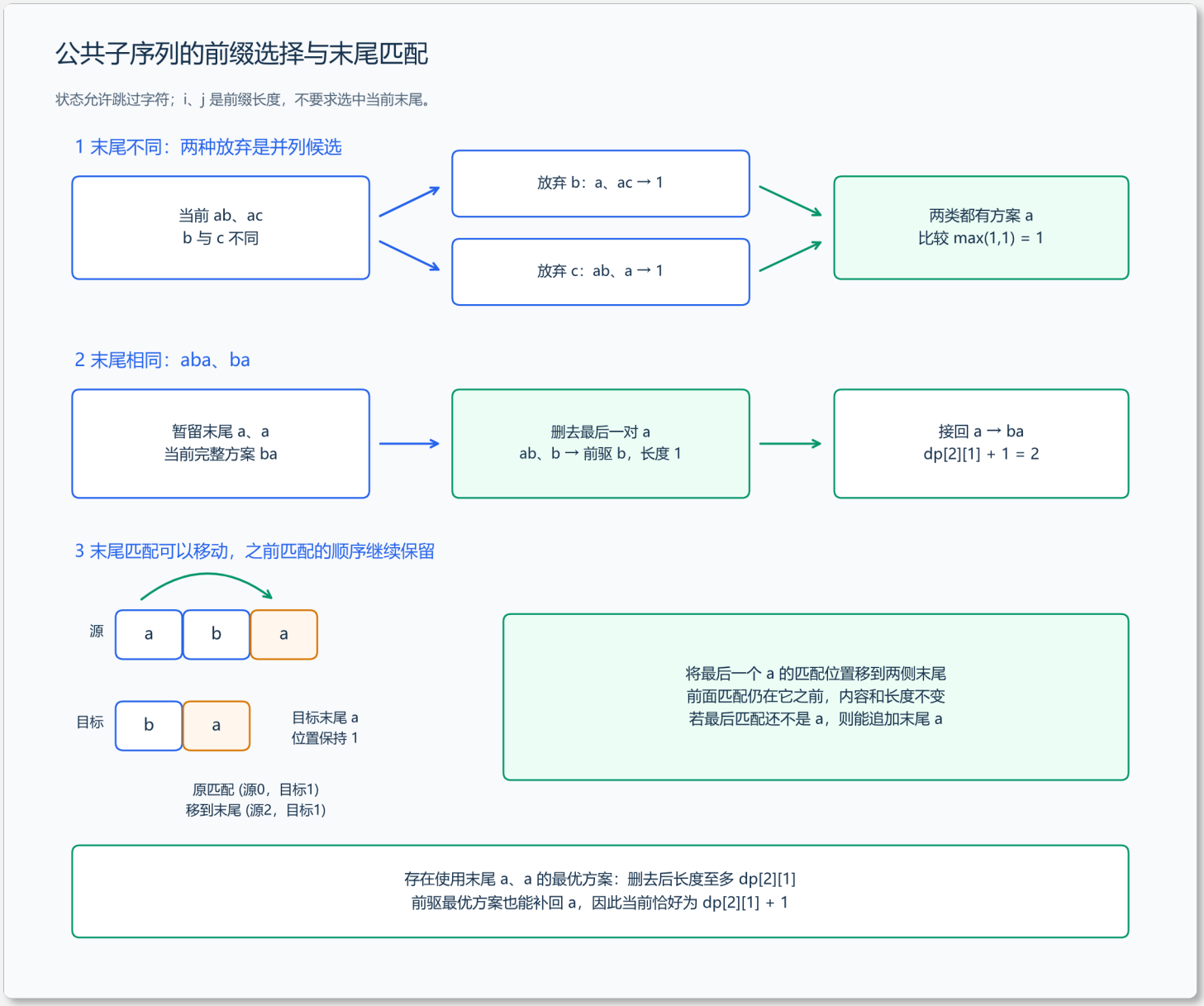
题目原文（自拟）： 给定两个按单字节字符处理的字符串，求二者公共子序列的最大长度。子序列可以跳过字符，但相对顺序必须一致；任一串为空时答案为 0。需要时返回一条最长公共子序列。

思路分析： 子序列可以跳过字符，因此把状态改成前缀中的最优值：`dp[i][j]` 表示从 `s1` 的前 `i` 个字符与 `s2` 的前 `j` 个字符中，能选出的最长公共子序列长度。**末尾可以选，也可以不选。** 这与公共子串固定两个结尾的状态不同。

先计算 `ab`、`ac` 的 `dp[2][2]`。末尾 b、c 不同，无法把它们匹配成公共子序列的最后一对。任意匹配方案至少放弃其中一个末尾：

放弃的末尾	剩余前缀	对应状态	最优内容与长度
放弃源串 b	a、ac	dp[1][2]	a，长度 1
放弃目标串 c	ab、a	dp[2][1]	a，长度 1

方案 a 同时出现在两类中。我们要最长长度，因此比较 `max(1,1)=1`；相加会把同一个方案重复计算。每个较短前缀中的方案在较长前缀中仍合法，而每个当前方案又至少属于其中一类，所以两个前驱的最大值正好是当前最优值。



再计算 `aba、ba` 的 `dp[3][2]`。末尾都是 a，先把它们留作最后一对匹配。删去这对字符，剩余前缀为 `ab、b`，最优子序列 b 长度 1；接回 a 得到 ba，长度 2，即 `dp[2][1]+1`。

还要解释这次直接选末尾会不会遗漏最优方案。取一条最长公共子序列，分两种情况：

1. 若最后匹配的字符不是 a，两侧最后的 a 都没有被使用。把这对 a 加在末尾还能保持下标顺序，长度会增加一，因此这样的方案尚未达到最长。
2. 若最后匹配的字符是 a，可以把它在两串中的匹配位置都移到最后一个 a。前面的匹配位置仍在它们之前，内

容与长度不变。例如只匹配两串靠前的 a，可以把它们改为匹配末尾的 a。

于是一定能选到一条使用这对末尾 a 的最优方案。删去它后，剩余长度最多为 `dp[2][1]`；而这个前驱的最优方案又确实能接回 a。两边得到同一个长度 `dp[2][1]+1=2`。

把前缀长度推广为 `i、j`，得到

$$dp[i][j]=\begin{cases} dp[i-1][j-1]+1, & s1[i-1]=s2[j-1], \\ \max\big(dp[i-1][j],dp[i][j-1]\big), & s1[i-1]\neq s2[j-1]. \end{cases}$$

上方去掉源末尾，左方去掉目标末尾，左上角去掉一对末尾；这里的“去掉”是缩短当前子问题的前缀，不是修改输入字符串。一侧前缀为空时只能选出空子序列，所以第 0 行、列为 0。

ab、ac 的状态	空目标	a	ac
空源	0	0	0
a	0	0+1=1	max(0,1)=1
ab	0	max(1,0)=1	max(1,1)=1

逐行、每行从左到右填时，上方、左方和左上角都已计算。右下角包含两个完整字符串的全部选择，直接读取 `dp[m][n]`。遇到不匹配仍可保留以前匹配到的字符，所以它保留前驱最优值。



恢复内容时，从右下角沿达到最优值的来源倒走。



`abcde、ace` 从 `(5,3)` 匹配 e，记录 e 并去 `(4,2)`；此处 d、c 不同，上方为 2、左方为 1，向上走。接着匹配 c，再跳过 b，最后匹配 a。逆序收集 `e、c、a`，反转得到 ace。它是在已算好的表中找一条最优来源链，表中的结果继续保留。平局时本节代码向左，选择其中一条最优方案。

2.2.1 按末尾匹配结果推导二维表

当末尾不同，公共子序列不可能同时使用这两个末尾，因此至少去掉一侧；两种子问题的最优值取最大即可。末尾相等时，可把这个字符补到较短前缀的公共子序列后；任何最优方案也能调整为使用这对末尾，故取左上角加一。

```cpp
#include <algorithm>
#include <string>
#include <vector>
using namespace std;
// 当末尾不同，公共子序列不可能同时使用这两个末尾，因此至少去掉一侧；两种子问题的最优值取最大即可。
int longestCommonSubsequence_length(string s1, string s2) {
    int m = s1.length();
    int n = s2.length();
    if (m == 0 || n == 0)
        return 0;

    // dp[i][j]: s1前i个字符和s2前j个字符的LCS长度
    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));
```

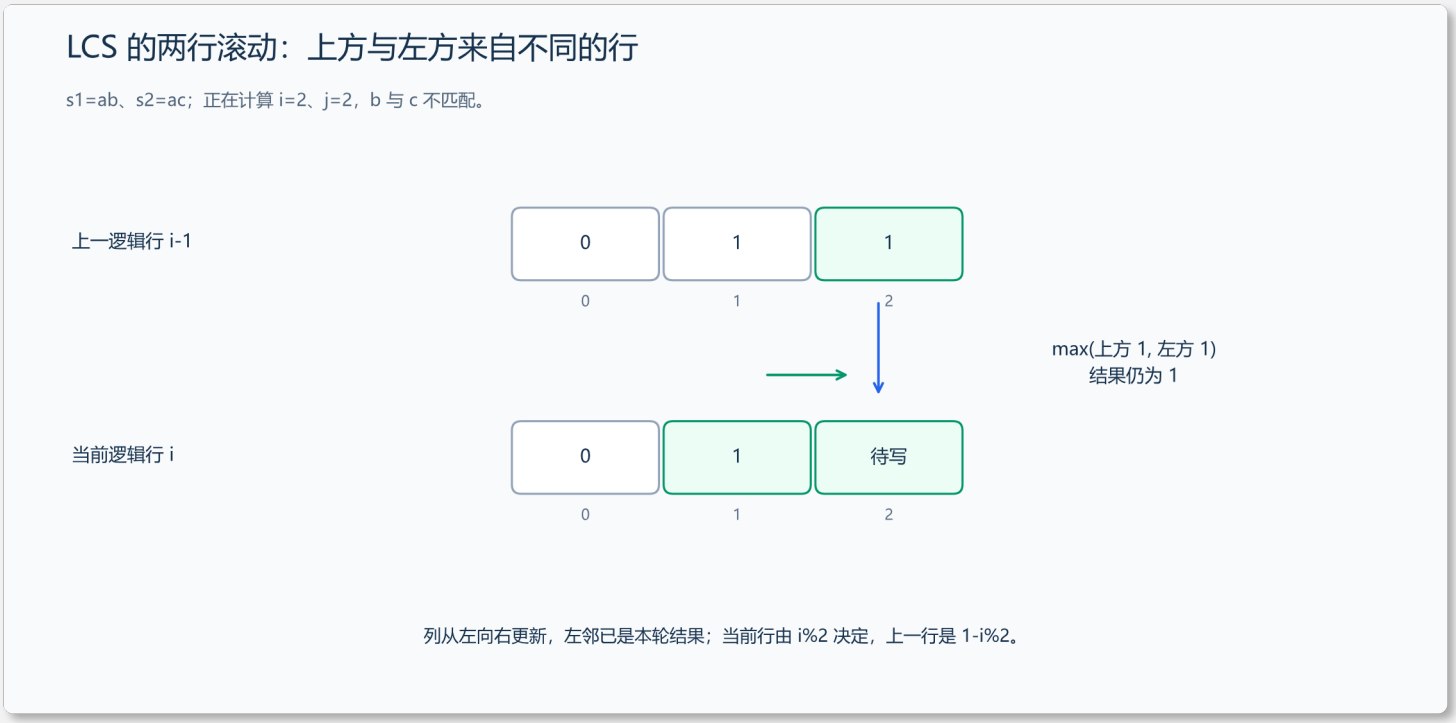
```cpp

    for (int i = 1; i ≤ m; i++) {
        for (int j = 1; j ≤ n; j++) {
            if (s1[i - 1] == s2[j - 1]) {
                // 字符匹配，长度+1
                dp[i][j] = dp[i - 1][j - 1] + 1;
            } else {
                // 字符不匹配，取两个方向的最大值
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
            }
        }
    }

    return dp[m][n];
}
```

2.2.2 分清当前行与上一行

不匹配时取上方和左方最大值：上方属于上一逻辑行，左方属于当前行；匹配时读取上一行左上角。列从左到右保证当前行的左侧已计算。



LCS 滚动表同时依赖上一行与本行。

```cpp
#include <algorithm>
#include <string>
#include <vector>
using namespace std;
// 不匹配时取上方和左方最大值：上方属于上一逻辑行，左方属于当前行；匹配时读取上一行左上角。
int longestCommonSubsequence_space_optimized(string s1, string s2) {
    int m = s1.length();
    int n = s2.length();
    if (m == 0 || n == 0)
        return 0;

```

```cpp
    // 列方向 s2 保持较短，减少滚动数组宽度
    if (n > m) {
        swap(s1, s2);
        swap(m, n);
    }

    // 只使用两行数组
    vector<vector<int>> dp(2, vector<int>(n + 1, 0));

    for (int i = 1; i ≤ m; i++) {
        int curr = i % 2;
        int prev = 1 - curr;

        for (int j = 1; j ≤ n; j++) {
            if (s1[i - 1] == s2[j - 1]) {
                dp[curr][j] = dp[prev][j - 1] + 1;
            } else {
                dp[curr][j] = max(dp[prev][j], dp[curr][j - 1]);
            }
        }
    }

    return dp[m % 2][n];
}
```

2.2.3 沿最优来源恢复内容

从右下角倒走。相同字符收集一次并同时缩短两个前缀；不同字符走向维持最优值的一侧。相等候选可以任选，本函数在平局时向左走，最后反转收集结果。

```cpp
#include <algorithm>
#include <string>
#include <vector>
using namespace std;
// 从右下角倒走。
string longestCommonSubsequence_sequence(string s1, string s2) {
    int m = s1.length();
    int n = s2.length();
    if (m == 0 || n == 0)
        return "";

    // 先计算dp表
    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));

    for (int i = 1; i ≤ m; i++) {
        for (int j = 1; j ≤ n; j++) {
            if (s1[i - 1] == s2[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1] + 1;
            } else {
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
            }
        }
    }

```

```cpp
    // 回溯构造LCS
    string lcs;
    int i = m, j = n;

    while (i > 0 && j > 0) {
        if (s1[i - 1] == s2[j - 1]) {
            // 当前字符属于LCS
            lcs.push_back(s1[i - 1]);
            i--;
            j--;
        } else if (dp[i - 1][j] > dp[i][j - 1]) {
            // 向上回溯
            i--;
        } else {
            // 向左回溯
            j--;
        }
    }

    reverse(lcs.begin(), lcs.end());
    return lcs;
}
```

样例与复杂度： `abcde`、`ace` 得到 `ace`，长度 3；`ABCDGH`、`AEDFHR` 可以得到 `ADH`；`AGGTAB`、`GXTXAYB` 可以得到 `GTAB`。时间为 $O(mn)$；完整表和此处恢复方案的表空间为 $O(mn)$，只求长度的滚动表为 $O(\min(m,n))$。按值参数还会占用字符串副本空间。若有多条最优序列，函数只返回其中一条。

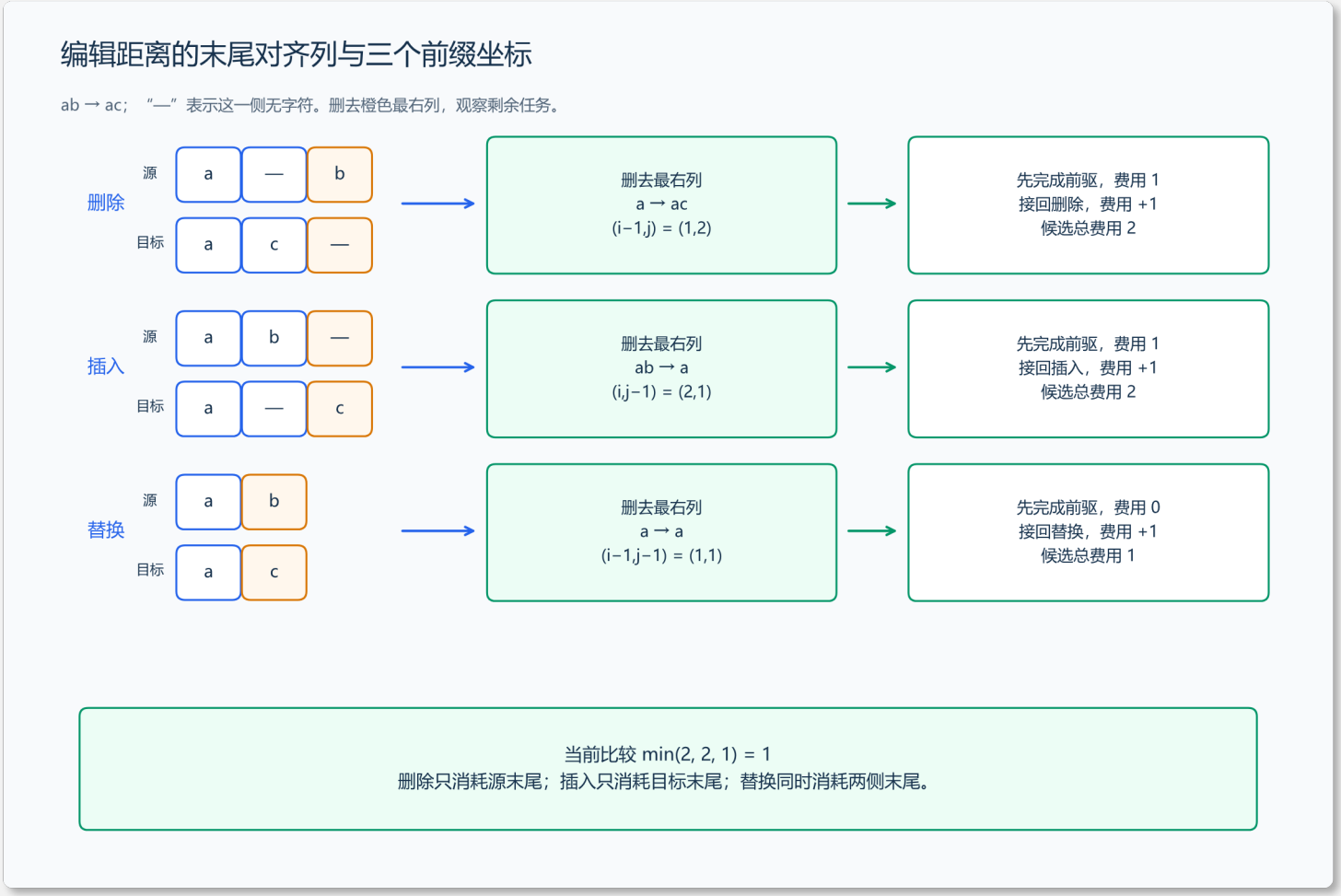
扩展： 原数组与“排序去重后的数组”做 LCS 可以得到严格 LIS 长度，但平方复杂度不适合大规模。带权匹配要把长度加一改为对应匹配收益，并检查允许跳过后的最优转移。多个序列需要更多状态维度，代价通常是各长度的乘积，还要计入每状态的转移数。只允许插入和删除时，最少操作数为 $m+n-2\,\mathrm{LCS}$；允许代价 1 的替换时不能直接使用这个公式。

练习题： 洛谷 P1435 – 回文字串（可转化为与反串的 LCS）、洛谷 P1439 – 两个排列的最长公共子序列（需优化到 LIS）。

2.3 编辑距离

题目原文（自拟）： 给定两个按单字节字符处理的字符串，允许插入、删除、替换一个字符，每次代价为 1。求把第一个变成第二个的最少代价；允许空串。扩展任务是给三种操作分别设置非负代价，或输出一组最优操作说明。所有长度与代价的计算结果都在 `int` 范围内。

思路分析： 令 `dp[i][j]` 表示把源串前 `i` 个字符变成目标串前 `j` 个字符的最小费用。先求 `ab→ac` 的 `dp[2][2]`，用字符对齐观察末尾：同一列是一个源字符与目标字符的对应，`-` 表示这一侧没有字符。



可以把完整的对齐分成三种末尾列：

末尾对齐	本列动作	删去这一列后的任务	前驱费用	接回费用
源 b、目标 `-`	删除 b	`a→ac`，源前缀短一位	`dp[1][2]=1`：插入 c	`1+1=2`
源 `-`、目标 c	插入 c	`ab→a`，目标前缀短一位	`dp[2][1]=1`：删除 b	`1+1=2`
源 b、目标 c	把 b 替换为 c	`a→a`，两个前缀各短一位	`dp[1][1]=0`：直接匹配	`0+1=1`

删除这一列时，消耗了哪个串的末尾，就将哪个前缀长度减 1。因此删除的前驱是 `(i-1,j)`，插入的前驱是 `(i,j-1)`，替换的前驱是 `(i-1,j-1)`。

例如删除分支的完整对齐是：源 `a、-、b` 对目标 `a、c、-`。前两列已经把 a 变成 ac，最后一列再删除 b。这里拆的是**对齐中最右的一列**，实际执行操作时可以安排成先删 b 再插 c，费用相同。这样不会把“删除”误解为目标串的长度也必须减少。

每种前驱中的任意合法对齐，都可以接回对应的末尾列。把前面费用为 3 的对齐换成同前缀费用为 1 的对齐，末尾字符与动作不变，完整费用也就少 2。于是每类只保留前驱的最小费用，再加本列费用。末尾不相等时，三种末尾列覆盖候选，比较三类最小费用，`ab→ac` 得到 `min(2,2,1)=1`。

末尾相同，例如 `ab→ab`，两侧 b 可以直接匹配，本列费用为 0，前驱是 `a→a`。在这里每种编辑费用都是 1，末尾相同的一对可以保留；把它们删掉再重新产生不会降低费用。因此匹配分支直接读左上角。

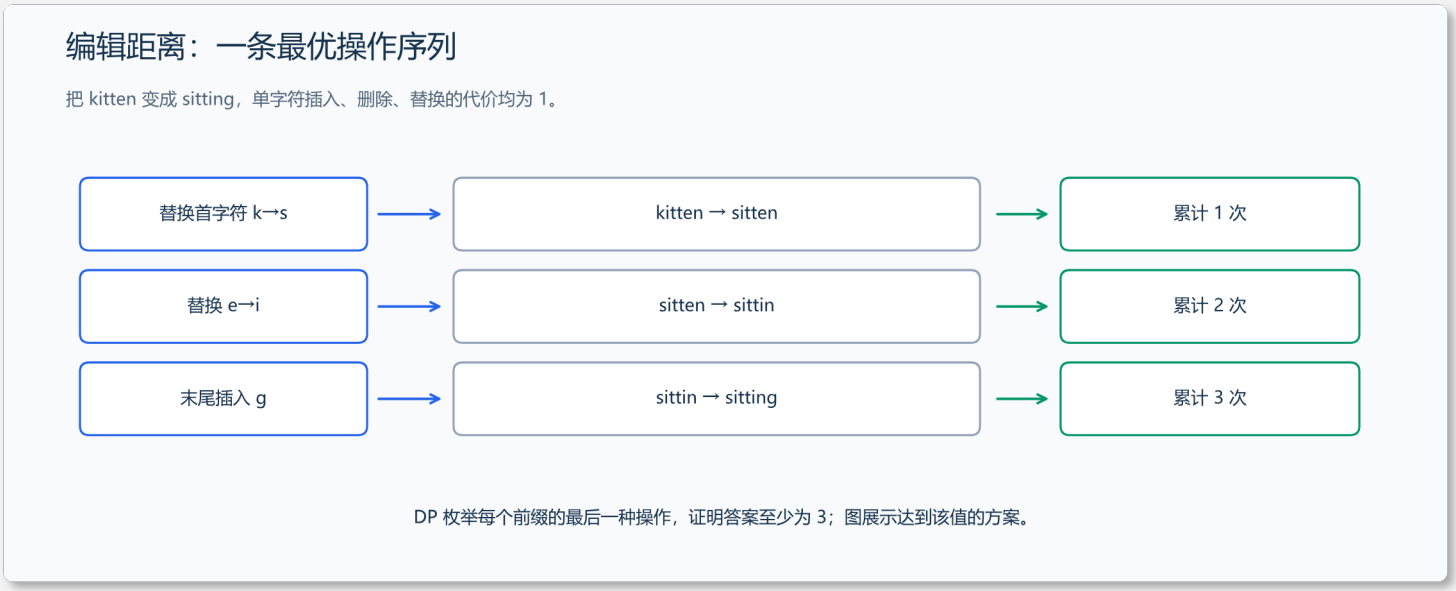
$$dp[i][j]=\begin{cases} dp[i-1][j-1], & s1[i-1]=s2[j-1], \\ \min\big(dp[i-1][j]+1,\ dp[i][j-1]+1,\ dp[i-1][j-1]+1\big), & s1[i-1]\neq s2[j-1]. \end{cases}$$

三项的 `+1` 都表示接回一个编辑动作，`min` 对应题目要求的最少次数。匹配时没有 `+1`，因为保留相同字符不算编辑。

空源串变成目标前 `j` 个字符，只能逐个插入，费用为 `j`；源前 `i` 个字符变为空串，只能逐个删除，费用为 `i`。所以 `dp[0][j]=j、dp[i][0]=i`，它们表示实际操作费用。

ab→ac 的状态	空目标	a	ac
空源	0	1	2
a	1	匹配：0	插入 c：1
ab	2	删除 b：1	比较删除 2、插入 2、替换 1，取 1

上方、左方、左上角都比当前任务少处理字符，逐行从左到右计算即可。右下角 `dp[m][n]` 是完整源串到完整目标串的费用。两行压缩保留上方与左上角，当前行从左到右产生左方。



`kitten→sitting` 可以替换 `k→s`、替换 `e→i`，再在末尾插入 g，总计 3 次。带权版本仍拆同样的末尾列，只把各分支的 `+1` 换成各自费用；空前缀的插入、删除费用也逐个累计。

2.3.1 三种操作对应三个前驱

每个单元先问：把源前缀变成目标前缀，最后处理末尾时是哪种动作？删除只消耗源末尾，插入只消耗目标末尾，替换同时消耗两侧，因此前驱分别是 (i-1,j)、(i,j-1)、(i-1,j-1)。三类都能产生合法方案，取最小值。

```
#include <algorithm>
#include <string>
#include <vector>
using namespace std;
// 每个单元先问：把源前缀变成目标前缀，最后处理末尾时是哪种动作？删除只消耗源末尾，插入只消耗目标末尾，替换同时消耗两侧，因此前驱分别是 (i-1,j)、(i,j-1)、(i-1,j-1)。
```

```cpp
int minDistance(string word1, string word2) {
    int m = word1.length();
    int n = word2.length();

    // dp[i][j]: 将word1的前i个字符转换为word2的前j个字符的最小编辑距离
    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));

    // 先填写空前缀对应的状态值。
    for (int i = 0; i ≤ m; i++) {
        dp[i][0] = i; // word1前i个字符转换为空串：删除i次
    }
    for (int j = 0; j ≤ n; j++) {
        dp[0][j] = j; // 空串转换为word2前j个字符：插入j次
    }

    // 状态转移
    for (int i = 1; i ≤ m; i++) {
        for (int j = 1; j ≤ n; j++) {
            if (word1[i - 1] == word2[j - 1]) {
                // 字符相同，无需操作
                dp[i][j] = dp[i - 1][j - 1];
            } else {
                // 字符不同，取三种操作的最小值
                int del_cost = dp[i - 1][j] + 1;      // 删除word1[i-1]
                int ins_cost = dp[i][j - 1] + 1;      // 插入word2[j-1]
                int rep_cost = dp[i - 1][j - 1] + 1; // 替换word1[i-1]为word2[j-1]
                dp[i][j] = min({del_cost, ins_cost, rep_cost});
            }
        }
    }

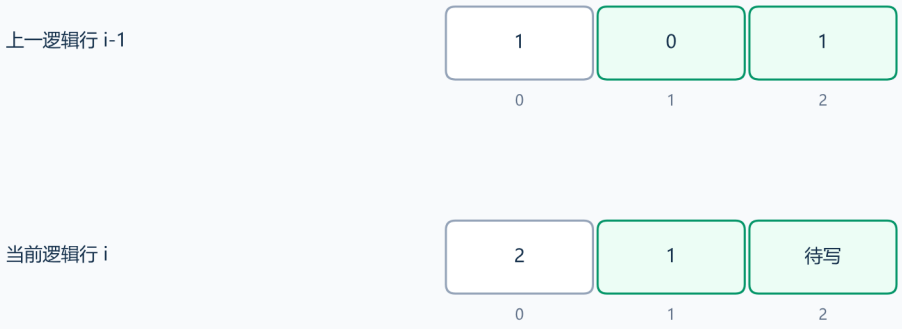
    return dp[m][n];
}
```

2.3.2 两行滚动：先填写空前缀的代价

上方和左上角读上一行，左方读本行。每轮先设置 `dp[curr][0]` =i，代表删除当前源前缀全部字符；漏掉这一步会读到上一轮的初值。

编辑距离的两行滚动：逐一读取三个前驱

word1=ab、word2=ac；i=2、j=2，b 与 c 不同。



删除：上方 1+1=2；插入：左方 1+1=2；替换：左上 0+1=1。

取最小值 1；每轮还须把当前行第 0 列写成 i，不能沿用旧边界。

上方和左上角读上一行，左方读本行。

```cpp
#include <algorithm>
#include <string>
#include <vector>
using namespace std;
// 上方和左上角读上一行，左方读本行。
int minDistance_space_optimized(string word1, string word2) {
    int m = word1.length();
    int n = word2.length();

    // 列方向 word2 保持较短，减少滚动数组宽度
    if (n > m) {
        swap(word1, word2);
        swap(m, n);
    }

    // 只使用两行数组
    vector<vector<int>> dp(2, vector<int>(n + 1, 0));

    // 初始化第一行（空串转换为word2）
    for (int j = 0; j ≤ n; j++) {
        dp[0][j] = j;
    }

    // 状态转移
    for (int i = 1; i ≤ m; i++) {
        int curr = i % 2;
        int prev = 1 - curr;

        // 初始化当前行的第一个元素
        dp[curr][0] = i;

```

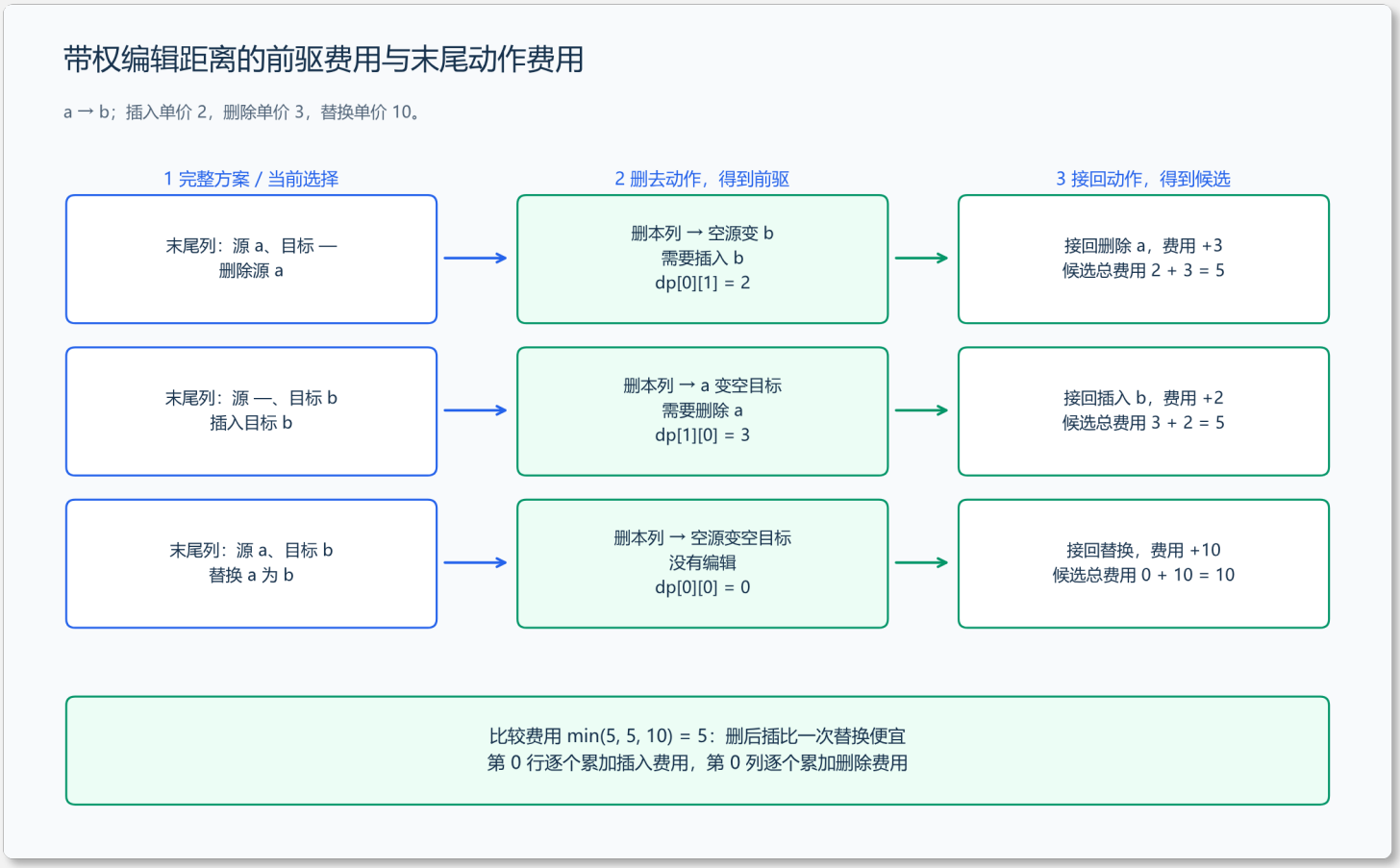
```cpp
        for (int j = 1; j <= n; j++) {
            if (word1[i - 1] == word2[j - 1]) {
                dp[curr][j] = dp[prev][j - 1];
            } else {
                int del_cost = dp[prev][j] + 1;      // 删除
                int ins_cost = dp[curr][j - 1] + 1; // 插入
                int rep_cost = dp[prev][j - 1] + 1; // 替换
                dp[curr][j] = min({del_cost, ins_cost, rep_cost});
            }
        }
    }

    return dp[m % 2][n];
}
```

2.3.3 不同代价下比较总费用

还是按末尾对齐列拆分，只把本列费用换成对应单价。把 a 变成 b，设插入费用 2、删除费用 3、替换费用 10：

接回的末尾动作	前驱任务与费用	完整费用
删除源 a	空源变 b，先插入，`dp[0][1]=2`	`2+3=5`
插入目标 b	a 变空，先删除，`dp[1][0]=3`	`3+2=5`
替换 a 为 b	空源变空目标，`dp[0][0]=0`	`0+10=10`



最小费用为 5，选择删 a 再插 b。它用了两次操作，费用却低于只替换一次的 10。不能先求最少次数，再把那条方案换算成价格；在每次转移时就要比较总费用。

末尾不匹配时，令插入、删除、替换费用分别为 `I`、`D`、`R`：

$$dp[i][j] = \min\big(dp[i-1][j] + D,\ dp[i][j-1] + I,\ dp[i-1][j-1] + R\big).$$

前驱坐标与上一节相同。空源到目标前缀要插入 `j` 个字符，故 `dp[0][j]=jI`；源前缀到空目标要删除 `i` 个字符，故 `dp[i][0]=iD`。两侧末尾相同时直接匹配，费用为 0，读取左上角。

```cpp
#include <algorithm>
#include <string>
#include <vector>
using namespace std;
// 每个前驱加对应动作的费用，空前缀的代价也按对应的插入或删除单价计算。
int minDistance_weighted(string word1, string word2, int insert_cost = 1, int delete_cost = 1,
                         int replace_cost = 1) {
    int m = word1.length();
    int n = word2.length();

    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));

    // 初始化边界（使用对应的操作代价）
    for (int i = 0; i ≤ m; i++) {
        dp[i][0] = i * delete_cost; // 删除i个字符
    }
    for (int j = 0; j ≤ n; j++) {
        dp[0][j] = j * insert_cost; // 插入j个字符
    }

    // 状态转移
    for (int i = 1; i ≤ m; i++) {
        for (int j = 1; j ≤ n; j++) {
            if (word1[i - 1] == word2[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1];
            } else {
                int del_cost = dp[i - 1][j] + delete_cost;      // 删除
                int ins_cost = dp[i][j - 1] + insert_cost;      // 插入
                int rep_cost = dp[i - 1][j - 1] + replace_cost; // 替换
                dp[i][j] = min({del_cost, ins_cost, rep_cost});
            }
        }
    }

    return dp[m][n];
}
```

2.3.4 填表时记录最优来源

dp 保存最小代价，op 保存选择了哪个前驱。平局先选替换，再选删除，最后插入，因此相同输入会按固定规则恢复。两个表的坐标含义相同，op 的数字不是编辑距离。



以 `ab→acb` 为例，`dp[2][3]=1`。从终点开始，b 与 b 匹配，向左上走到 `(1,2)`，不记录编辑；此处来源 `op[1][2]=2` 表示插入 c，只缩短目标前缀，向左走到 `(1,1)`；随后 a 与 a 匹配，向左上到 `(0,0)` 停止。操作列表只含“插入 c”，位置在 a 与 b 之间。

`op` 存的是方向和动作编号，`dp` 存的是最小代价，两者不能互换。一般情况也按此过程倒走，最后反转记录的编辑说明；匹配字符不算一次编辑。

```cpp
#include <algorithm>
#include <string>
#include <vector>
using namespace std;
// dp 保存最小代价，op 保存选择了哪个前驱。
struct EditTrace {
    vector<vector<int>> dp;
    vector<vector<int>> op; // 0 匹配或替换，1 删除，2 插入。
};

EditTrace buildEditTrace(const string &word1, const string &word2) {
    int m = word1.length();
    int n = word2.length();

    // 先计算dp表
    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));
    for (int i = 0; i ≤ m; i++)
        dp[i][0] = i;
    for (int j = 0; j ≤ n; j++)
        dp[0][j] = j;

    // 同时记录操作来源
    vector<vector<int>> op(m + 1, vector<int>(n + 1, 0));
```

```cpp
    // 0: 匹配/替换, 1: 删除, 2: 插入

    for (int i = 1; i ≤ m; i++) {
        for (int j = 1; j ≤ n; j++) {
            if (word1[i - 1] == word2[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1];
                op[i][j] = 0; // 匹配
            } else {
                int del_cost = dp[i - 1][j] + 1;
                int ins_cost = dp[i][j - 1] + 1;
                int rep_cost = dp[i - 1][j - 1] + 1;

                if (rep_cost ≤ del_cost && rep_cost ≤ ins_cost) {
                    dp[i][j] = rep_cost;
                    op[i][j] = 0; // 替换
                } else if (del_cost ≤ ins_cost) {
                    dp[i][j] = del_cost;
                    op[i][j] = 1; // 删除
                } else {
                    dp[i][j] = ins_cost;
                    op[i][j] = 2; // 插入
                }
            }
        }
    }

    return {dp, op};
}
```

2.3.5 从终点倒走并反转操作

每次按照 op 缩短一个或两个前缀，直到两者都为空。一侧已空时剩余字符只能全部插入或删除。倒走时先收集末尾动作，再收集前面的动作；反转后就按从左到右的对齐顺序输出。

```cpp
// 每次按照 op 缩短一个或两个前缀，直到两者都为空。
vector<string> recoverEditOperations(const string &word1, const string &word2,
                                     const vector<vector<int>> &op) {
    int m = word1.length(), n = word2.length();
    // 回溯获取操作序列
    vector<string> operations;
    int i = m, j = n;

    while (i > 0 || j > 0) {
        if (i == 0) {
            operations.push_back("插入 '" + string(1, word2[j - 1]) + "'");
            --j;
            continue;
        }
        if (j == 0) {
            operations.push_back("删除 '" + string(1, word1[i - 1]) + "'");
            --i;
            continue;
        }
        if (op[i][j] == 0) { // 匹配或替换
```

```cpp
            if (word1[i - 1] != word2[j - 1]) {
                operations.push_back("替换 '" + string(1, word1[i - 1]) + "' → '" +
                                     string(1, word2[j - 1]) + "'");
            }
            i--;
            j--;
        } else if (op[i][j] == 1) { // 删除
            operations.push_back("删除 '" + string(1, word1[i - 1]) + "'");
            i--;
        } else { // 插入
            operations.push_back("插入 '" + string(1, word2[j - 1]) + "'");
            j--;
        }
    }

    reverse(operations.begin(), operations.end());
    return operations;
}
```
cpp

2.3.6 用一个入口组合两个步骤

恢复操作需要第 4、5、6 段依次放在同一个文件，先声明 EditTrace 和辅助函数。前三段是独立的求值函数；它们与恢复函数可以共享本节的头文件。

```cpp
// 恢复操作需要第 4、5、6 段依次放在同一个文件，先声明 EditTrace 和辅助函数。
vector<string> getEditOperations(string word1, string word2) {
    // 先记录每格的最优来源，再从终点回溯一条操作说明。
    EditTrace trace = buildEditTrace(word1, word2);
    return recoverEditOperations(word1, word2, trace.op);
}
```
cpp

代码解释： 滚动实现保留上一行，当前行的第 0 列也要随 `i` 更新。带权函数把三个 `+1` 换成各自代价。`getEditOperations` 用来源表回溯，先处理一侧前缀为空时的剩余插入或删除，再反转步骤。它返回按字符对齐顺序排列的操作说明，没有给出动态下标；直接执行修改时还应记录作用位置，不能仅按字符值寻找第一个同名字符。

样例与复杂度： `horse→ros` 的最少次数为 3；空串到 `abc` 为 3；`kitten→sitting` 为 3；`intention→execution` 为 5。时间 $O(mn)$，完整表和来源表为 $O(mn)$ 空间，滚动 DP 表为 $O(\min(m,n))$。按值参数仍占字符串副本空间，返回的操作说明另计输出空间。

扩展： 只允许插入、删除时可用 $m+n-2\,\mathrm{LCS}$；加入相邻交换则成为另一类距离，状态转移中也加入这一步交换的来源和费用。不同字符替换可设不同代价，连续插入或删除的整段惩罚也需要额外状态。近似匹配可以允许不超过 `k` 的编辑代价，但匹配子串与转换整个字符串的初始状态不同。

练习题： 洛谷 P2758 — 编辑距离。

3. 背包问题

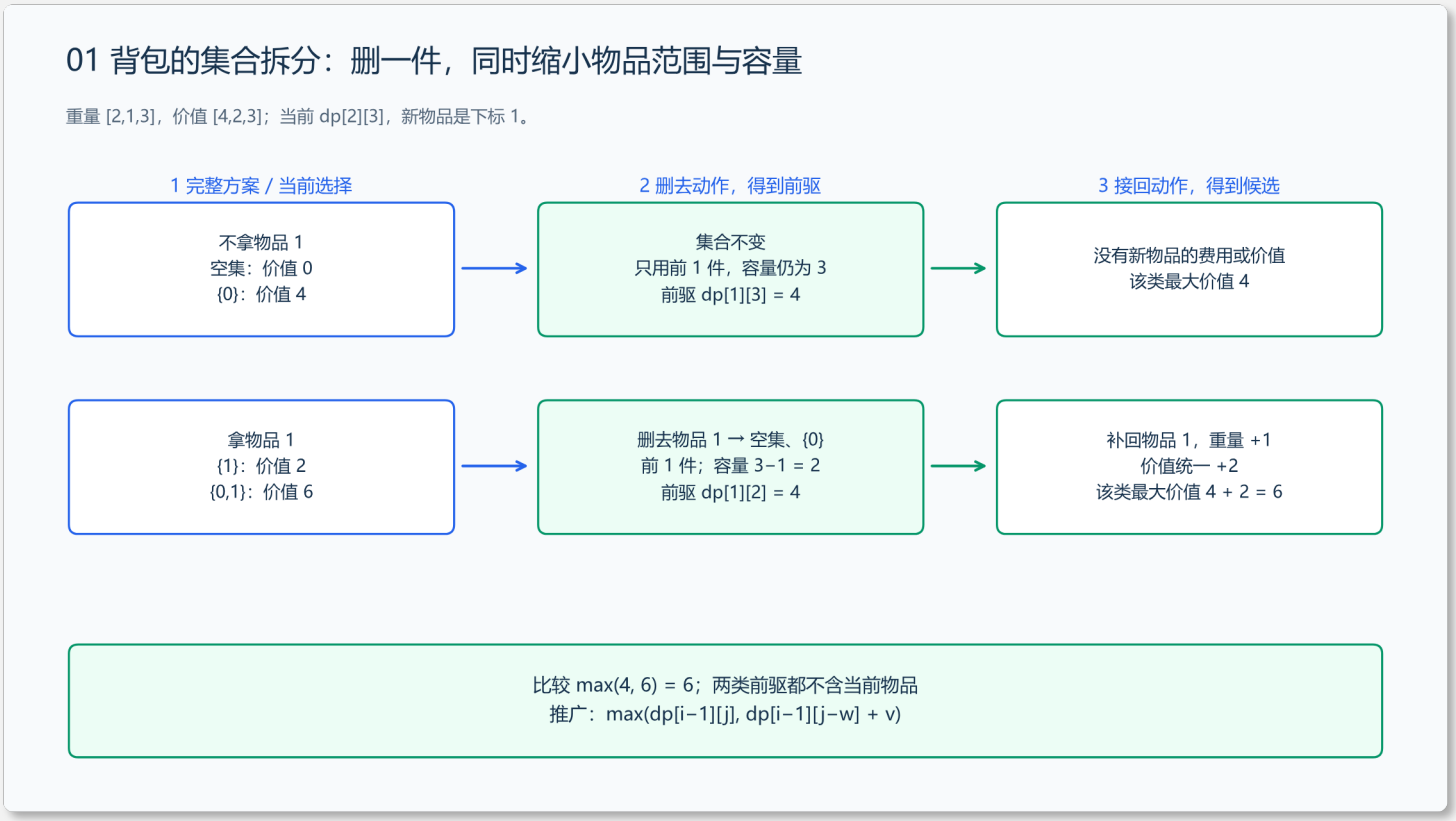
3.1 01 背包

题目原文（自拟）： 有 `N` 件物品，第 `i` 件的重量 `weight[i]` 为正整数、价值 `value[i]` 为非负整数，每件最多选择一次。背包容量 `C≥0`，求总重量不超过 `C` 的最大价值。`N` 与两个数组长度一致，价值总和在 `int` 范围内；还可以要求返回一组被选择的从 `0` 开始的物品下标。

思路分析： 先看重量 `[2,1,3]`、价值 `[4,2,3]`。状态 `dp[i][j]` 表示**只用前 `i` 件物品、总重量不超过 `j`** 时的最大价值。前 `i` 件的下标是 `0…i-1`，所以当前新加入的物品是下标 `i-1`。

求 `dp[2][3]` 时，只能选下标 0、1，重量最多为 3。四个合法集合是空集、`{0}`、`{1}`、`{0,1}`，价值分别为 `0、4、2、6`。按是否选择当前物品 1 分开：

当前集合这一类	允许的完整集合	去掉当前物品后的集合	剩余子问题	该类最大价值
不选物品 1	空集、`{0}`	集合不变	前 1 件、容量 3，`dp[1][3]`	4
选择物品 1	`{1}`、`{0,1}`	空集、`{0}`	前 1 件、容量 `3-1=2`，`dp[1][2]`	`4+2=6`



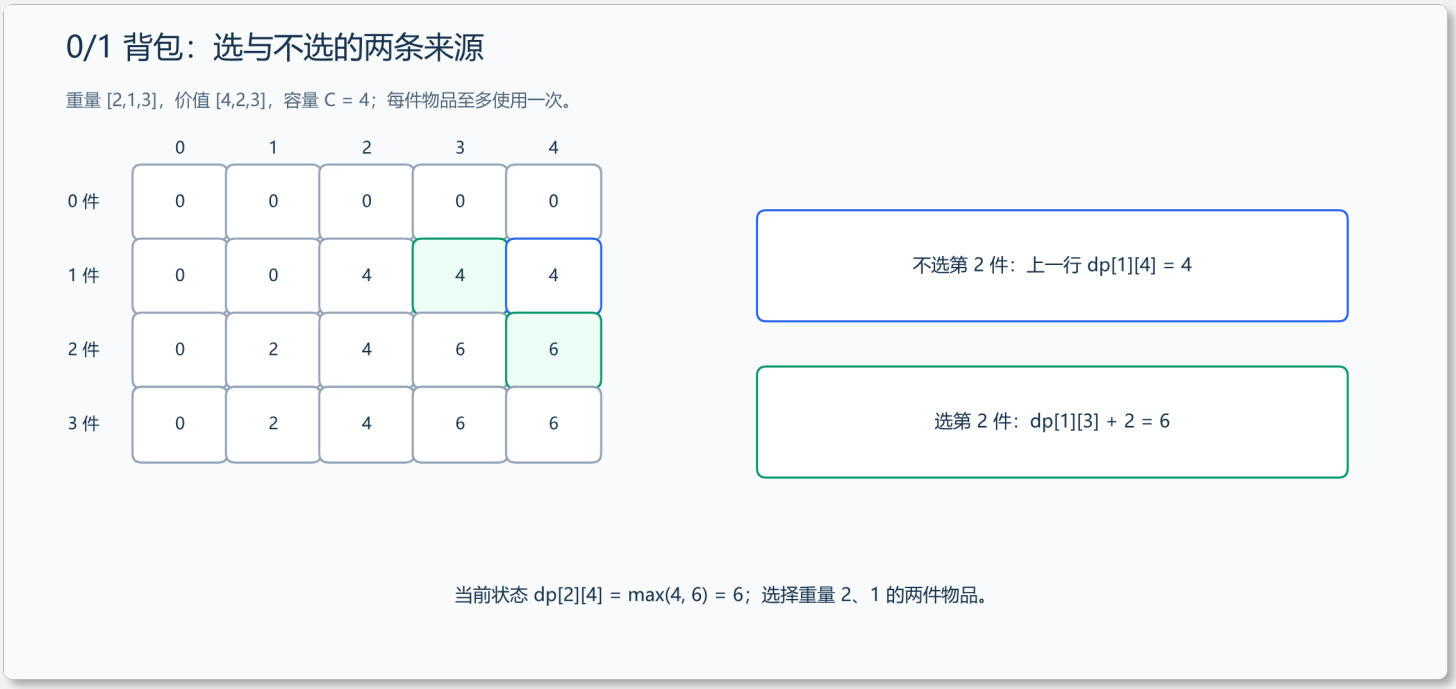
选择物品 1 的方案拿掉它后，可用物品减少一件，剩余重量最多为 2。反过来，任何只用物品 0、重量不超过 2 的集合，补回重量 1、价值 2 的物品 1，总重量就不超过 3。补回的价值始终是 2，所以剩余集合价值越大，完整价值也越大，可以直接读取 `dp[1][2]`。两类都覆盖到了，比较 `max(4,6)=6`。

令当前物品重量 `w=weight[i-1]`、价值 `v=value[i-1]`，同样得到

$$dp[i][j] = \begin{cases} dp[i-1][j], & j < w, \\ \max\big(dp[i-1][j],\ dp[i-1][j-w]+v\big), & j \ge w. \end{cases}$$

第一项保持原容量，因为没有拿当前物品；第二项先留出重量 `w`，再补回价值 `v`。两项都读 `i-1` 行，剩余集合里都没有当前物品。若物品重于容量，例如重量 3、容量 2，选择分支不存在，直接沿用不选分支。

没有物品时，各容量只对应空集，价值为 0，所以第 0 行全为 0。容量限制是“不超过”，空集对任意容量都合法；没有必要为了达到容量而拿物品。逐件填行，每个状态的两个前驱都在上一行，最终读取 `dp[N][C]`。



容量 `0…4` 的四行依次为 `[0,0,0,0,0]`、`[0,0,4,4,4]`、`[0,2,4,6,6]`、`[0,2,4,6,6]`。第三件在容量 4 处比较“不选的 6”与“容量 1 的 2 加价值 3”，得到 `max(6,5)=6`。

压缩成一行后的读取顺序。 **一行数组更新前存上一行。重量 2、价值 3 的一件物品，容量 4：逆序先算 `dp[4]=max(0,dp[2]+3)=3`，此时 `dp[2]` 仍为上一行的 0；之后才更新 `dp[2]=3`。这就保留了“剩余集合没有当前物品”的含义。**



若正序先把 `dp[2]` 写为 3，再用它算 `dp[4]`，就会得到 6。这个前驱已经拿过当前物品，接回一次就拿了两件。01 背包因此逆序容量；下一节完全背包会允许这种重复连接。

下面先给出可放进 `main` 的一维核心片段，再给出独立求值函数与恢复函数。

```cpp
#include <algorithm>
#include <vector>
using namespace std;
vector<int> w{2, 1, 3}, v{4, 2, 3};
int n = static_cast<int>(w.size()), W = 4;
vector<int> dp(W + 1, 0); // 初始化为容量不超过 W 的最优价值。
for (int i = 1; i ≤ n; i++) {
    for (int j = W; j ≥ w[i - 1]; j--) { // 注意：需要倒序遍历
        dp[j] = max(dp[j], dp[j - w[i - 1]] + v[i - 1]);
    }
}
```

恢复时，比较当前行与上一行：值变大则记录当前件并扣重量；相等则跳过也能保留一组最优方案。



样例从 `(3,4)` 开始，当前行与上一行都为 6，跳过物品 2；到 `(2,4)`，6 大于上一行的 4，拿物品 1，容量减为 3；到 `(1,3)`，4 大于上一行的 0，拿物品 0，容量减为 1。逆序记录 `[1,0]`，反转为 `[0,1]`，总重量 3、价值 6。两行相等时选择跳过只是恢复其中一组最优解，其他最优解仍可能拿当前件。

3.1.1 按选或不选划分二维状态

每个合法集合恰属于“不选当前物品”或“选当前物品”之一。选中时去掉它，剩下只能使用前 i-1 件且重量不超过 j-w；若剩余部分不是该子问题的最优解，替换它即可提高总价值，故可以使用上一行最优值。

```cpp
#include <algorithm>
#include <utility>
#include <vector>
using namespace std;
// 每个合法集合恰属于“不选当前物品”或“选当前物品”之一。
int knapsack_01_2d(int N, int C, vector<int> &weight, vector<int> &value) {
    // dp[i][j]: 考虑前i件物品，背包容量为j时的最大价值
    vector<vector<int>> dp(N + 1, vector<int>(C + 1, 0));

    for (int i = 1; i ≤ N; i++) {
        for (int j = 1; j ≤ C; j++) {
            // 不选第i件物品
            dp[i][j] = dp[i - 1][j];

            // 如果背包容量足够，考虑选择第i件物品
            if (j ≥ weight[i - 1]) {
                dp[i][j] = max(dp[i][j], dp[i - 1][j - weight[i - 1]] + value[i - 1]);
            }
        }
    }

    return dp[N][C];
}
```

3.1.2 用逆序容量保留上一行

压缩后从大容量向小容量更新。读到的 `dp[j-w]` 还没有被当前物品覆盖，对应上一行；这保证当前物品只被使用一次。

```cpp
#include <algorithm>
#include <utility>
#include <vector>
using namespace std;
// 压缩后从大容量向小容量更新。
int knapsack_01_1d(int N, int C, vector<int> &weight, vector<int> &value) {
    // dp[j]: 背包容量为j时的最大价值
    vector<int> dp(C + 1, 0);

    for (int i = 0; i < N; i++) { // 遍历物品
        // 逆序遍历容量，确保每个物品只被使用一次
        for (int j = C; j ≥ weight[i]; j--) {
            dp[j] = max(dp[j], dp[j - weight[i]] + value[i]);
        }

        // 正序会读取本轮已更新的前驱，导致同一物品重复使用：
        // 假设物品重量2，价值3，C=4
        // 正序：j=2时dp[2]=3，j=4时dp[4]=dp[2]+3=6（错误，使用了两次）
        // 逆序：j=4时dp[4]=3，j=2时dp[2]=3（正确，只使用一次）
```

```cpp
    }

    return dp[C];
}
```

3.1.3 通过相邻两行恢复物品

从最后一件和完整容量回溯。若当前行比上一行大，当前物品一定参与这条最优来源，记录下标并扣重量；值相同则不选也能保持最优，向上一行继续。

```cpp
#include <algorithm>
#include <utility>
#include <vector>
using namespace std;
// 从最后一件和完整容量回溯。
pair<int, vector<int>> knapsack_01_with_items(int N, int C, vector<int> &weight,
                                              vector<int> &value) {
    vector<vector<int>> dp(N + 1, vector<int>(C + 1, 0));

    // 计算dp表
    for (int i = 1; i ≤ N; i++) {
        for (int j = 1; j ≤ C; j++) {
            dp[i][j] = dp[i - 1][j];
            if (j ≥ weight[i - 1]) {
                dp[i][j] = max(dp[i][j], dp[i - 1][j - weight[i - 1]] + value[i - 1]);
            }
        }
    }

    // 回溯找出选择的物品
    vector<int> selected_items;
    int j = C;
    for (int i = N; i > 0; i--) {
        if (dp[i][j] ≠ dp[i - 1][j]) {
            // 说明第i件物品被选中了
            selected_items.push_back(i - 1); // 记录物品索引（0-based）
            j -= weight[i - 1];              // 减少剩余容量
        }
    }

    reverse(selected_items.begin(), selected_items.end());
    return {dp[N][C], selected_items};
}
```

样例与解释： 重量 `[2,1,3]`、价值 `[4,2,3]`、容量 4，最大价值为 6，选择下标 0、1，重量 3，没有装满但仍最优。`N=0` 或 `C=0` 时答案为 0。

复杂度： 三种函数的填表时间均为 $O(NC)$。二维与恢复方案的表空间 $O(NC)$，一维额外空间 $O(C)$；恢复结果最多 `N` 个下标。容量是数值规模，输入容量非常大时不能只看物品数就开表。

扩展： 恰好装满时，只有容量 0 初始可达，其余为负无穷，转移前还要判断前驱可达；方案计数时，`dp[0]=1`，其余为 0，按物品逆序容量累加。可以把状态改成“达到价值 `v` 的最小重量”，在总价值较小而容量很大时更合适。多重背包可把有限数量拆成若干 01 物品包，例如 13 件拆成 `1、2、4、6`，组合能表达 `0…13` 件；也可以继续学习按重量余数分类的单调队列优化。分割等和子集可先检查总和是否为偶数，再用 01 可行性 DP 判断能否达到总和的一半。

练习题： 洛谷 P1048 - 采药、洛谷 P1049 - 装箱问题、洛谷 P1164 - 小 A 点菜（方案计数）。

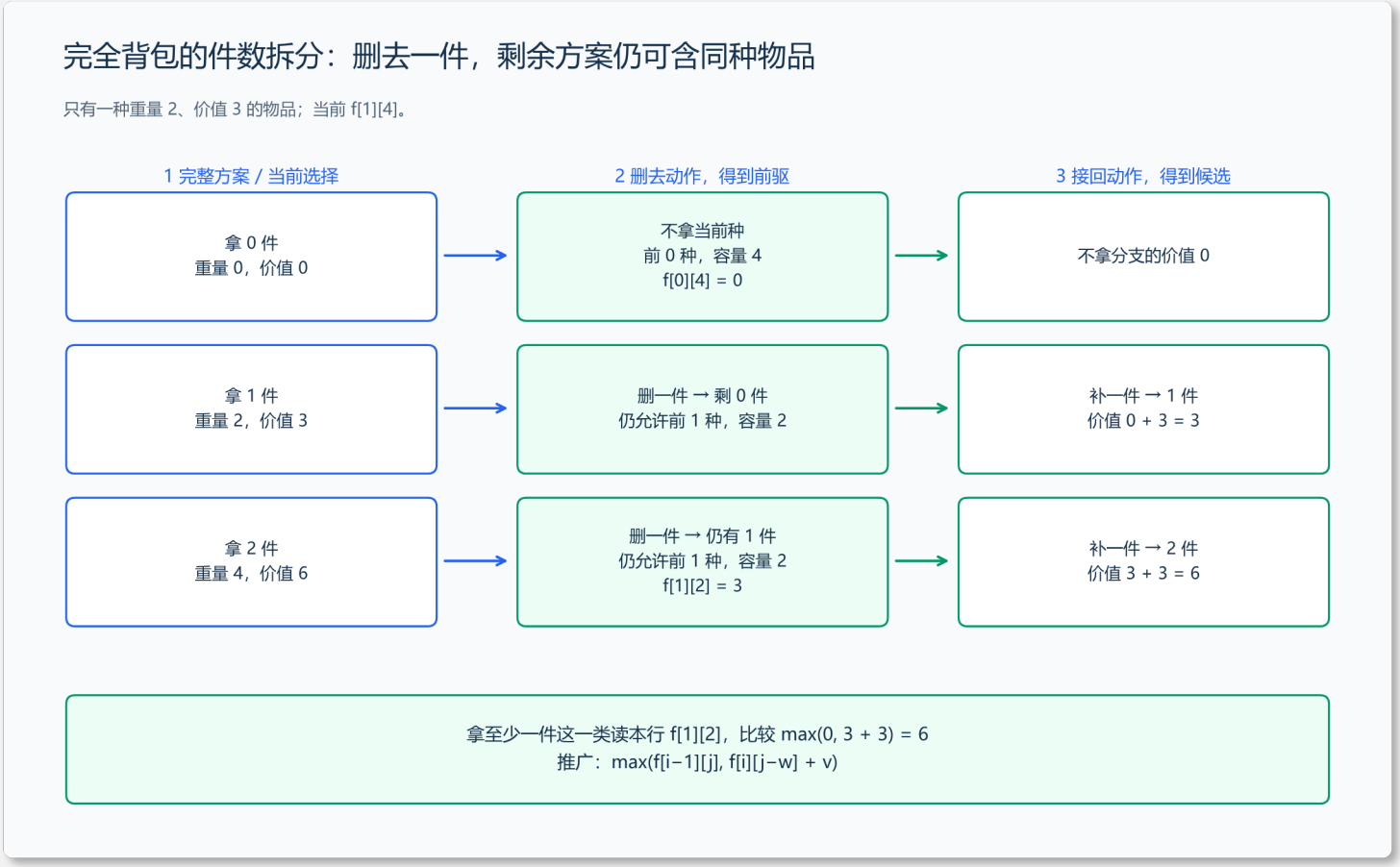
3.2 完全背包

题目原文（自拟示例）： 有 `n` 种物品，第 `i` 种重量为正整数 `weight[i]`、价值为非负整数 `value[i]`，每种可以选择任意多件。背包容量为 `capacity ≥ 0`，求重量不超过容量的最大总价值；所有答案在 `long long` 范围内。

思路分析： 与 01 背包相比，现在同一种物品可以拿多件。仍先保留两个维度：`f[i][j]` 表示只用前 `i` 种、重量不超过 `j` 的最大价值。第 `i` 种在数组中的下标是 `i-1`。

只考虑重量 2、价值 3 的一种物品，容量 4。按拿了几件列出全部方案：

件数	总重量	总价值	去掉一件后
0	0	0	没有当前物品，属于不拿分支
1	2	3	剩 0 件，重量 0、价值 0
2	4	6	剩 1 件，重量 2、价值 3



把“拿至少一件”的两个方案各拿掉一件，得到 0 件和 1 件。这些剩余方案的重量不超过 `4-2=2`，**仍然允许这一种物品**，所以前驱是本行 `f[1][2]`，最大价值为 3。给这个最优剩余方案补一件，得到 `3+3=6`；不拿分支是上一行 `f[0][4]=0`，比较后取 6。

一般容量 `j` 也按“不拿当前种”和“至少拿一件当前种”分为两类。后一类去掉一件，容量少 `w`，种类范围仍为前 `i` 种；任何这种剩余方案补一件后又回到当前问题。令 `w=weight[i-1]、v=value[i-1]`，得到

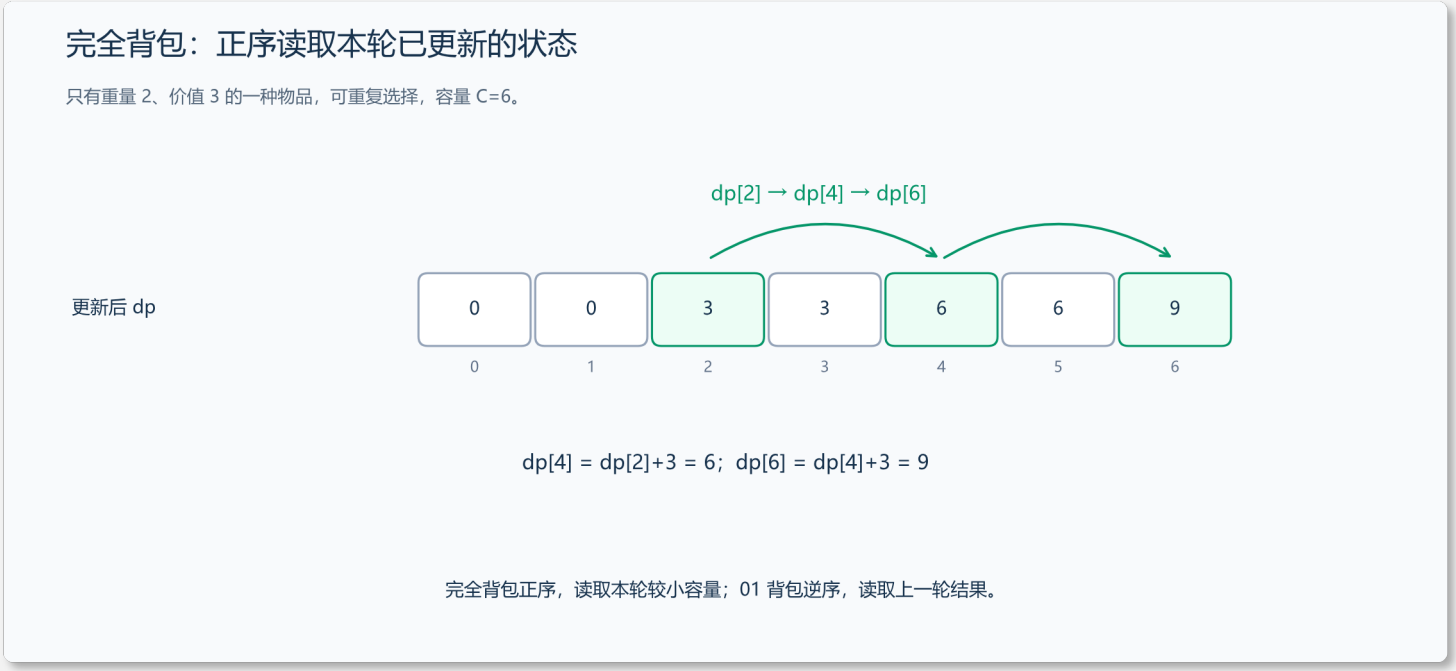
$$f[i][j]=\begin{cases} f[i-1][j], & j<w, \\ \max\big(f[i-1][j],\ f[i][j-w]+v\big), & j\ge w. \end{cases}$$

与 01 背包并排看：

去掉当前一件后	剩余方案中能否还有当前种	“拿”的前驱
01 背包	这一件已经拿掉，不能再出现	上一行 `dp[i-1][j-w]`
完全背包	可以仍有一件或多件	本行 `f[i][j-w]`

也可以直接枚举拿 `k` 件：$f[i][j]=\max_{0\le k\le \lfloor j/w\rfloor}(f[i-1][j-kw]+kv)$。上面的两分支式把“至少一件”先拆下一件，剩余件数交给本行较小容量处理，省去了每格再次枚举件数。

初始化仍是空集价值 0，允许不装满。逐种处理时，本行从小容量到大容量计算：`j-w<j`，需要的本行前驱先产生。一维数组里，更新前的 `dp[j]` 保留不拿分支；已经更新过的 `dp[j-w]` 则对应本行。



重量 2、价值 3、容量 `0…6` 时，先写 `dp[2]=3`，再由它写 `dp[4]=6`，最后由 6 写 `dp[6]=9`，全表为 `[0,0,3,3,6,6,9]`。这三个数分别对应一件、两件、三件。若逆序读旧行，就只能接回一次，计算结果会变成 01 背包。

恰好装满的任务中，只有容量 0 一开始有空方案；其他容量先记录没有方案，再寻找能接入的前驱。重量为 0、价值为正的物品则每加一件都增加价值而不占容量，最大价值可以一直增长；本题的正重量使每次接入都缩小前驱容量。

```cpp
#include <algorithm>
#include <vector>
using namespace std;

long long completeKnapsack(const vector<int> &weight, const vector<int> &value, int capacity) {
    vector<long long> dp(capacity + 1, 0);
    for (size_t i = 0; i < weight.size(); ++i) {
        // 正序容量循环，允许本轮状态继续使用同一种物品。
        for (int j = weight[i]; j ≤ capacity; ++j) {
            dp[j] = max(dp[j], dp[j - weight[i]] + value[i]);
        }
    }
    return dp[capacity];
}
```

样例： 重量 `[2, 3]`、价值 `[3, 4]`、容量 `6`，可以选三个第一种物品，最大价值为 `9`。

代码解释： 与 0/1 背包相比，关键区别是容量循环方向。正序允许重复使用本轮物品；逆序限制每种物品至多使用一次。两个数组按同一物品下标配对读取。时间 $O(nC)$，额外空间 $O(C)$。

练习题： 洛谷 P1616 – 疯狂的采药。

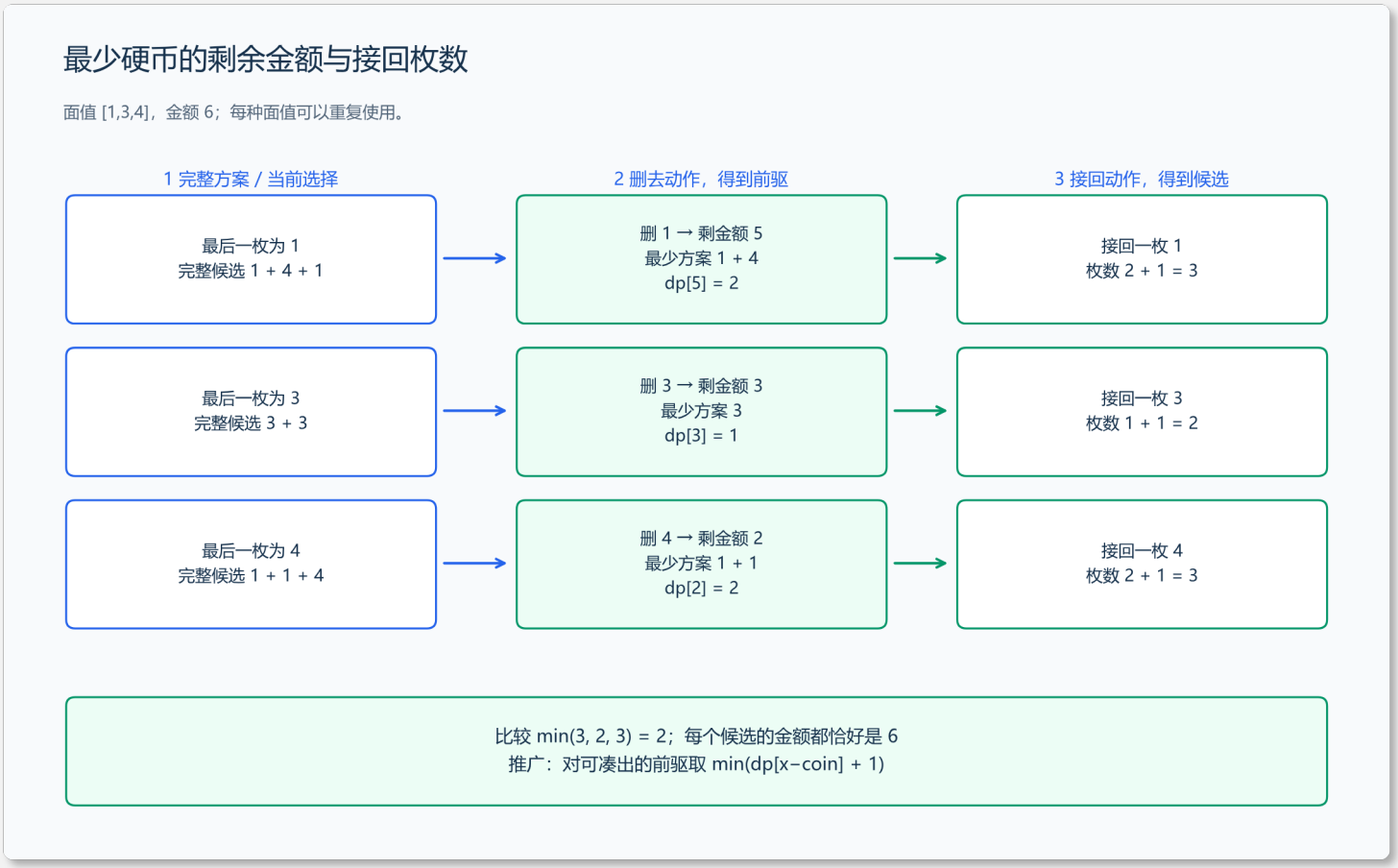
3.3 最少硬币数量

题目原文（自拟示例）： 给定一些正整数面值，每种硬币数量无限，求凑出非负金额 `amount` 所需的最少硬币数。无法凑出返回 `-1`，金额为 `0` 时返回 `0`。

思路分析： 回到贪心中面值 `[1,3,4]`、金额 6 的例子。先拿 4 会留下 `1+1`，共 3 枚；方案 `3+3` 只要 2 枚。因此逐个比较可能的末尾硬币，让较好的前驱方案参与组合。

令 `dp[x]` 表示**恰好凑出金额** `x` 的最少硬币数。金额必须恰好，金额 5 的方案不能直接当金额 6 的答案。对金额 6，拿掉选作最后一枚的硬币：

最后一枚	剩余金额	该金额的最少方案	接回后的枚数
1	`6-1=5`	`1+4`，2 枚	`dp[5]+1=3`
3	`6-3=3`	`3`，1 枚	`dp[3]+1=2`
4	`6-4=2`	`1+1`，2 枚	`dp[2]+1=3`



任意非空方案都有一枚硬币可以拿掉；剩下金额一定为 `x-coin`。同一金额的剩余方案若用了更多硬币，就把它换成最少枚数的方案，再接回这枚 `coin`，金额保持 `x`，总枚数更少。于是每种末尾硬币只需读取该剩余金额的最优值。

同一组硬币可能按不同顺序出现在多个末尾候选中，例如 `1+4+1` 与 `1+1+4`。这里比较枚数的最小值，重复候选不会改变最小值；若任务改为数硬币组合，就还要重新安排枚举方式来避免重复。

把 6 换成 `x`，逐个面值试接回来：

$$dp[x] = \min_{\substack{coin \le x \\ x-coin \text{ 能凑出}}} \big(dp[x-coin] + 1\big).$$

`x-coin` 是扣掉这一枚后的金额，`+1` 是补回这一枚，`min` 选择枚数最少的候选。没有可接前驱时，当前金额继续记录为 `INF`，表示尚无方案。

金额 0 的空方案需要 0 枚，所以 `dp[0]=0`。其他金额先写 `INF`。面值 `[2]` 时，金额 1 没有可放的硬币，保持 `INF`；算金额 3 时，放 2 后还剩金额 1，它没有方案，所以金额 3 也没有方案。程序最后将这种结果返回为 `-1`。



面值 `[1,3,4]` 的金额 `0…6` 依次得到 `[0,1,2,1,1,2,2]`。所有面值为正，剩余金额 `x-coin` 比 `x` 小，因此从小金额到大金额填写。金额 6 的三个前驱值已经得到，直接比较 `3、2、3`，最终读取 `dp[amount]`。

```cpp
#include <algorithm>
#include <limits>
#include <vector>
using namespace std;

int minCoins(const vector<int> &coins, int amount) {
    constexpr int INF = numeric_limits<int>::max() / 2;
    vector<int> dp(static_cast<size_t>(amount) + 1, INF);
    dp[0] = 0; // 凑出金额 0 不需要硬币，其他金额初始不可达。
    for (int x = 1; x ≤ amount; ++x) {
        for (int coin : coins) {
            if (coin ≤ x && dp[x - coin] ≠ INF) {
                dp[x] = min(dp[x], dp[x - coin] + 1);
            }
        }
    }
    return dp[amount] == INF ? -1 : dp[amount];
```

```
18  }                                                                    cpp
```

样例： 面值 `[1, 3, 4]`、金额 `6`，答案为 `2`，选择 `3 + 3`；面值 `[2]`、金额 `3`，返回 `-1`。

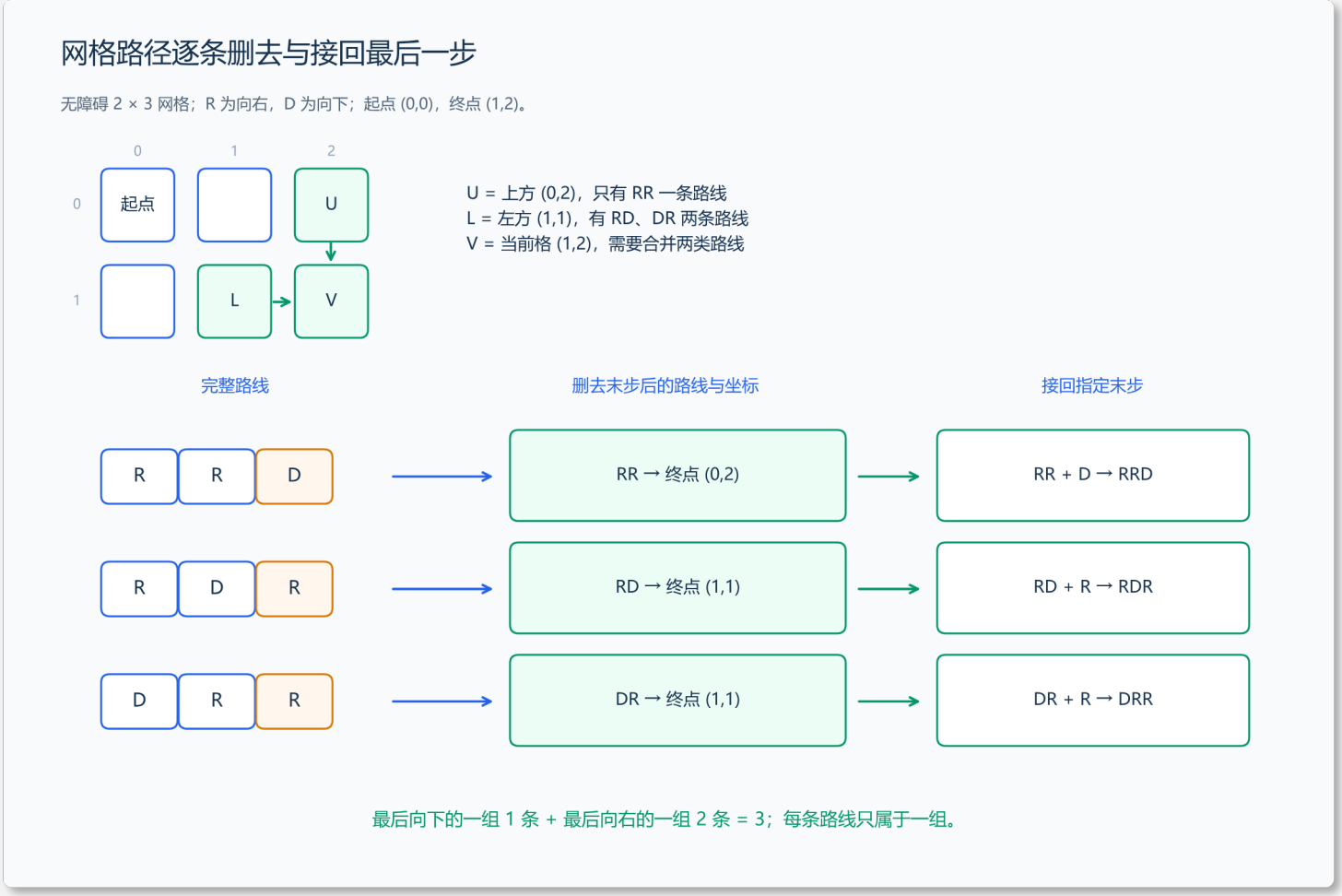
代码解释： `INF` 表示不可达；只有前驱金额可达时才能转移。这解决了前面“先拿最大面值”贪心可能失败的问题。时间 $O(kM)$，额外空间 $O(M)$，其中 `k` 是面值数量，`M` 是目标金额。

练习题： 洛谷 P2842 - 纸币问题 1。

4. 网格 DP：按依赖顺序填表

题目原文（自拟）： 有一个 `rows×columns` 的网格，`1≤rows,columns≤500`，只能向右或向下移动，不能越界。部分格子是障碍。求从左上角 `(0,0)` 到右下角 `(rows-1,columns-1)` 的合法路径数量，对 `1000000007` 取模。函数输入是等长行组成的 `vector<vector<bool>> blocked`，`true` 表示障碍；返回一个整数。起点或终点为障碍时返回 `0`；只有一个可走格子时，不移动计为一条路径。

从搜索递推到迭代顺序： 状态仍是 `dp[r][c]`，表示从起点到当前格子的路径数。先把具体路线与每个加数对应起来，再安排填表顺序。



计算 `(1,2)` 时，`RRD` 删去 D 后是到 `(0,2)` 的 RR；`RDR`、`DRR` 删去 R 后是到 `(1,1)` 的 RD、DR。给上方的每条路线接一个 D，给左方的每条路线接一个 R，完整路线全部还原。两类最后一步不同，数量为 `1+2=3`。

计算一般格子 `(r,c)` 时也是如此，上方提供最后向下的全部路线，左方提供最后向右的全部路线。取模只是控制保存的数值，路径分类仍按同样方式相加：

$$dp[r][c] = \big(dp[r-1][c] + dp[r][c-1]\big) \bmod M.$$

第一行没有上方格子，实际只接左方路线；第一列实际只接上方路线。起点的不移动空路线计为 1。遇到障碍，所有要经过该格的路线都无法接入，格子写 0。



处理位置	已经能读到的路线数	当前写入
`(0,0)`	起点，空路线	1
`(0,1)`	左方 1	1
`(0,2)`	左方 1	1
`(1,0)`	上方 1	1
`(1,1)`	上方 1，左方 1	2
`(1,2)`	上方 1，左方 2	3

按行从上到下、行内从左到右，每个上方在上一行已经完成，每个左方在当前行已经完成。记忆化从终点发出请求，直到最简单状态返回；迭代从起点逐格写入，到终点时直接读取同一份依赖产生的结果。



把 `(0,1)` 封住，RRD 与 RDR 都经过它，只有 DRR 保留。第一行变成 `1、0、0`，第二行依次为 `1、1、1`；终点从上方得到 0、从左方得到 1，仍使用同一个相加方程得到 1。

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

int gridPaths(const vector<vector<bool>> &blocked) {
    constexpr int mod = 1000000007;
    int rows = static_cast<int>(blocked.size());
    int columns = static_cast<int>(blocked[0].size());
    vector<vector<int>> dp(rows, vector<int>(columns, 0));
    for (int row = 0; row < rows; ++row) {
        for (int column = 0; column < columns; ++column) {
            // 障碍格不参与转移，起点被封住时也保持 0 条路径。
            if (blocked[row][column]) {
                continue;
            }
            if (row == 0 && column == 0) {
                dp[row][column] = 1;
            } else {
                long long count = 0;
                if (row > 0) {
                    count += dp[row - 1][column];
                }
                if (column > 0) {
                    count += dp[row][column - 1];
                }
                dp[row][column] = static_cast<int>(count % mod);
            }
        }
    }
    return dp[rows - 1][columns - 1];
}
```

样例与代码解释： 无障碍 `2×3` 返回 3；把 `(0,1)` 变成障碍后第一行是 `1、0、0`，第二行是 `1、1、1`，返回 1。先处理障碍再处理起点，与记忆化版本的起点、障碍处理方式一致。每格常数工作，时间与表空间均为 $O(rows \times columns)$；进一步压缩为一行时，上方是覆盖前的 `dp[c]`，左方是本轮的 `dp[c-1]`，遇到障碍时将 dp[c] 写为 0，表示没有路径穿过这个格子。

练习题： 洛谷 P1002 - 过河卒。

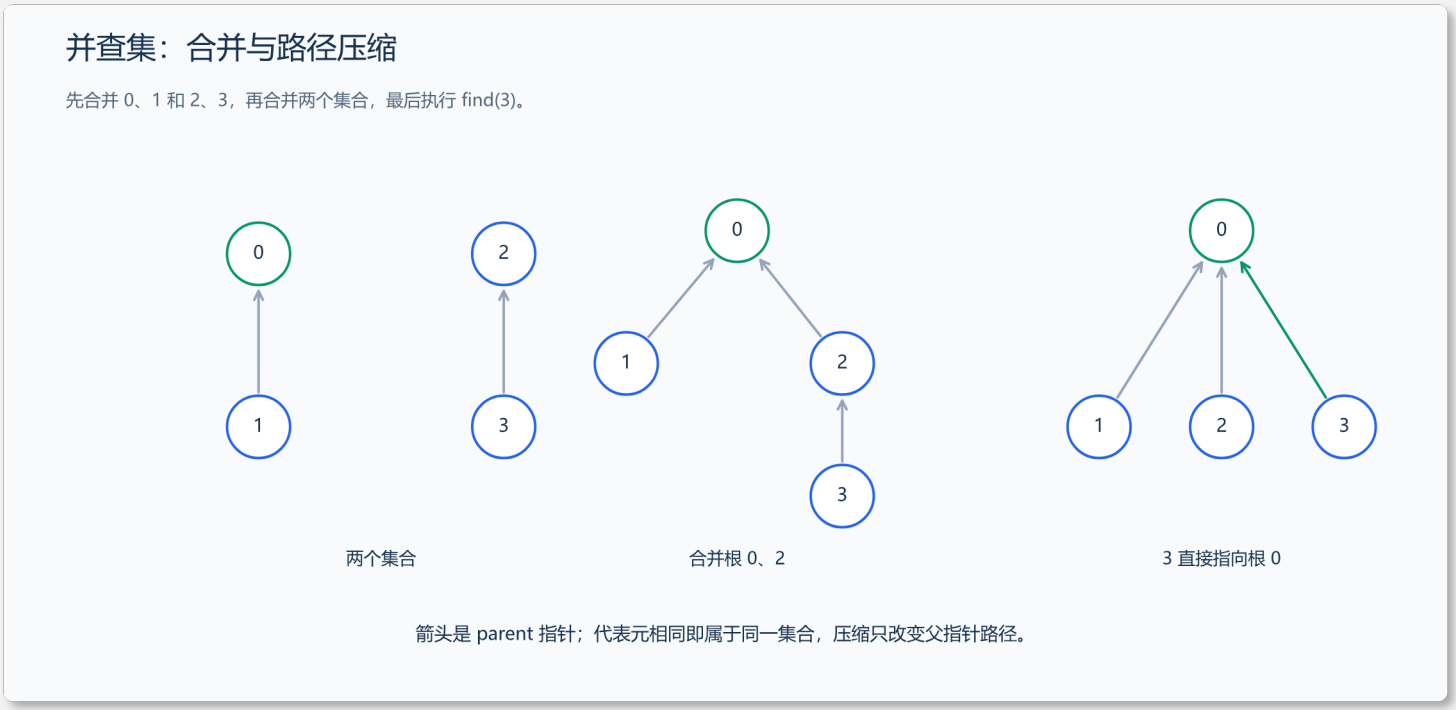
（十四） 并查集

如果题目不断连接两个对象，并询问它们是否已经属于同一组，目标是维护合并关系。先观察两次合并怎样改变组，再推导父指针与查找代表的方法。

并查集维护一些不相交的集合，支持“查找元素属于哪个集合”和“合并两个集合”。常用于连通性查询。

题目原文（自拟示例）： 初始有 `n ≥ 1` 个独立元素，编号为 `0` 到 `n-1`。操作 `1 a b` 合并 `a`、`b` 所在集合；操作 `2 a b` 查询它们是否在同一集合，输出 `YES` 或 `NO`。第一行输入 `n q`，随后输入 `q` 条操作，所有编号合法。

思路分析： 每个集合用一棵父指针树表示，根为代表元。查找时压缩路径，合并时把较小的树接到较大的树上。



图中箭头从元素指向父节点。find(3) 之后代表元不变，只把中间路径缩短。



合并 `(0,1)、(2,3)、(0,2)` 后，父数组为 `[0,0,0,2]`。`find(3)` 先调用 `find(2)`，再到根 0；返回 2 的那层执行 `parent[2]=0`，返回 3 的那层执行 `parent[3]=0`，得到 `[0,0,0,0]`。压缩发生在得到根之后的返回阶段，不会把原图新增一条边，也不会改变集合归属。

父指针表示集合归属，不表示原图中的边。 初始化时 `parent[x]=x`，每个元素自己是一个集合。顺着父指针走到 `parent[root]=root` 的位置，得到代表元；两个元素找到同一个代表元，就属于同一集合。

合并时先用 find 找到两个集合的根，再把小集合的根接到大集合的根。旧根下的整棵树随之连接，两组元素得到同一个代表。按大小合并时，size_ 在根处保存整个集合的大小；新根累计两组大小，之后查找都以新根表示合并后的集合。

样例先合并 `0、1`，集合为 `{0,1}、{2}、{3}`，所以 `0、2` 不连通。再合并 `1、2` 时，实际上合并的是 `{0,1}` 的根和 `{2}` 的根，结果为 `{0,1,2}、{3}`，此后 `0、2` 连通。

路径压缩把某条查找路径上的节点直接指向同一个根，没有改变任何节点所属集合，只减少后续查找的路径长度。普通并查集只记录“在不在同一集合”，不能由父指针恢复原图路径，也不能直接删除一条边后撤销所有连通影响。

```cpp
#include <algorithm>
#include <iostream>
#include <numeric>
#include <vector>
using namespace std;

class DisjointSet {
private:
    vector<int> parent_;
    vector<int> size_;

public:
```

```cpp
    explicit DisjointSet(int n) : parent_(n), size_(n, 1) {
        iota(parent_.begin(), parent_.end(), 0);
    }

    int find(int x) {
        if (parent_[x] != x) {
            // 路径压缩：让访问路径上的元素直接连接代表元。
            parent_[x] = find(parent_[x]);
        }
        return parent_[x];
    }

    bool unite(int a, int b) {
        a = find(a);
        b = find(b);
        if (a == b) {
            return false;
        }
        if (size_[a] < size_[b]) {
            swap(a, b);
        }
        // 把较小集合接到较大集合，控制树的高度。
        parent_[b] = a;
        size_[a] += size_[b];
        return true;
    }

    bool same(int a, int b) {
        return find(a) == find(b);
    }
};

int main() {
    int n, q;
    cin >> n >> q;
    DisjointSet sets(n);
    while (q--) {
        int operation, a, b;
        cin >> operation >> a >> b;
        if (operation == 1) {
            sets.unite(a, b);
        } else {
            cout << (sets.same(a, b) ? "YES" : "NO") << '\n';
        }
    }
}
```

输入：

```text
4 4
1 0 1
2 0 2
1 1 2
2 0 2
```

输出：

```text
NO
YES
```

代码解释： `find` 返回代表元；`unite` 返回是否进行了新合并。两个元素的代表元相同，即属于同一集合。初始化 $O(n)$，结合两种优化后，每次操作的均摊复杂度为 $O(\alpha(n))$，其中 α 是反阿克曼函数；额外空间 $O(n)$。

普通并查集不直接支持拆开集合，也不保存两点之间的具体路径。

练习题： 洛谷 P3367 — 并查集、洛谷 P1551 — 亲戚。

（十五） 图论入门

并查集能回答是否连通，却不记录从哪里经过哪些道路或路程多长。把道路写成边、地点写成点，再复用 DFS、BFS、堆和 DP 处理路径与依赖问题。

图论研究“对象之间怎样连接”。先把题目中的对象抽象为顶点，把允许的一步关系抽象为边，再选择算法；道路长度、任务先后和格子移动虽然都能建图，所求的答案却可能完全不同。

1. 顶点、边、路径与连通性

1.1 先建立准确的图模型

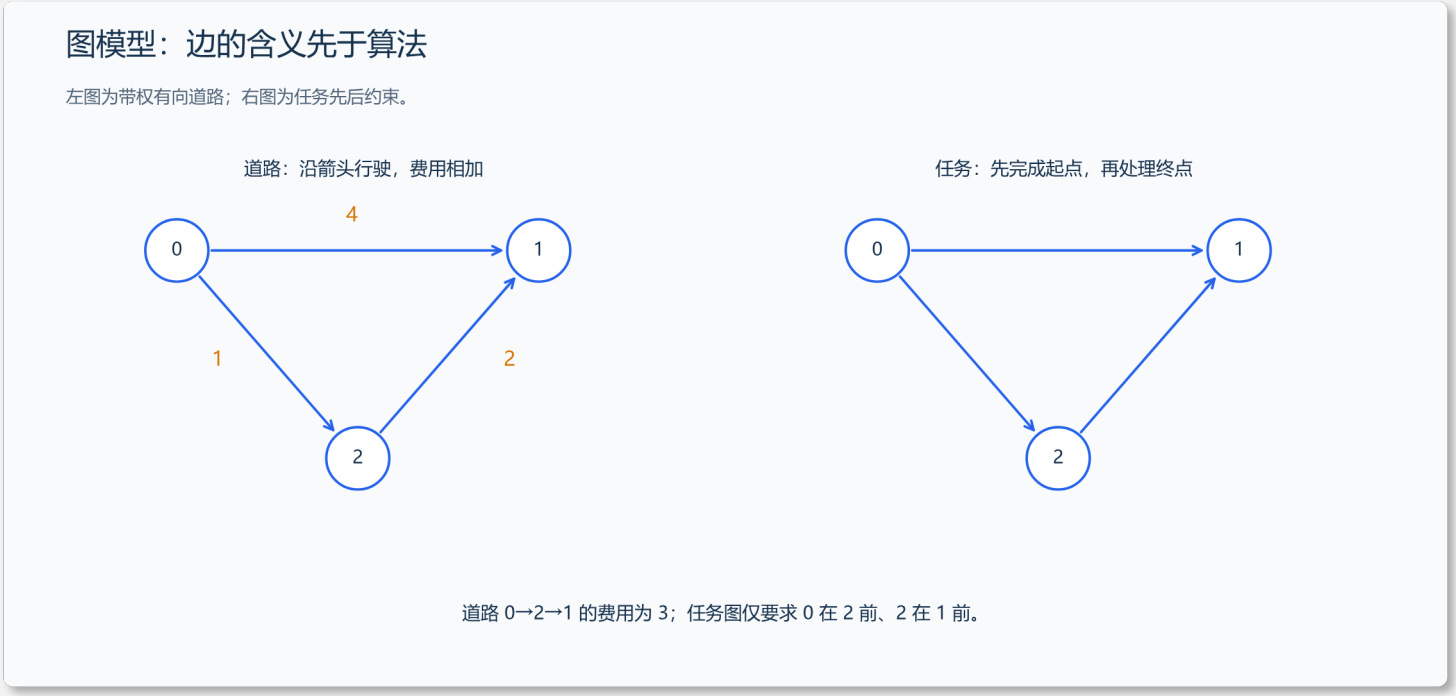
记图为 $G=(V,E)$，V 是顶点集合，E 是边的集合。本章用 n 表示顶点数、m 表示题目输入的边数，编号为 `0…n-1`。有向边记为 `u→v`；无向边连接 u、v 两端，允许双向通过，但输入时通常只写一次。

题目对象	顶点	边的含义	常见目标
城市与道路	城市	可以直接行驶的一条道路	能否到达、最短距离
人与朋友关系	人	两人之间直接的关系	连通块、是否同组
迷宫	可走格子	一次合法移动	可达性、最少步数
课程或任务	课程、任务	u 必须先于 v 完成	合法执行顺序

“有关系”不自动表示“能走”。若关系是“互相不能同组”，边表示约束，适合二分图染色；若是道路，边表示允许移动。边的定义决定后面的判断如何写。

有向与无向： `u→v` 表示从 u 到 v 的方向；从 v 回 u 则要再找相反方向的路径。一条双向道路可表示为无向边，也可在邻接表中存两条反向记录。存储记录数加倍，题目输入的边数 m 不变。

无权与带权： 无权最短路通常数边数，每步代价统一；带权图的边附有时间、长度、费用等权值，例如路线总费用是沿途边权相加；若题目问一路最高的费用，就沿途取最大边权。先写出题目怎样计算一条路径的值，再决定怎样比较和更新。零权边是存在且费用为 0 的边，不能用 0 同时表示“没有边”。



道路的方向与权值限制移动，依赖边则要求先完成起点任务。

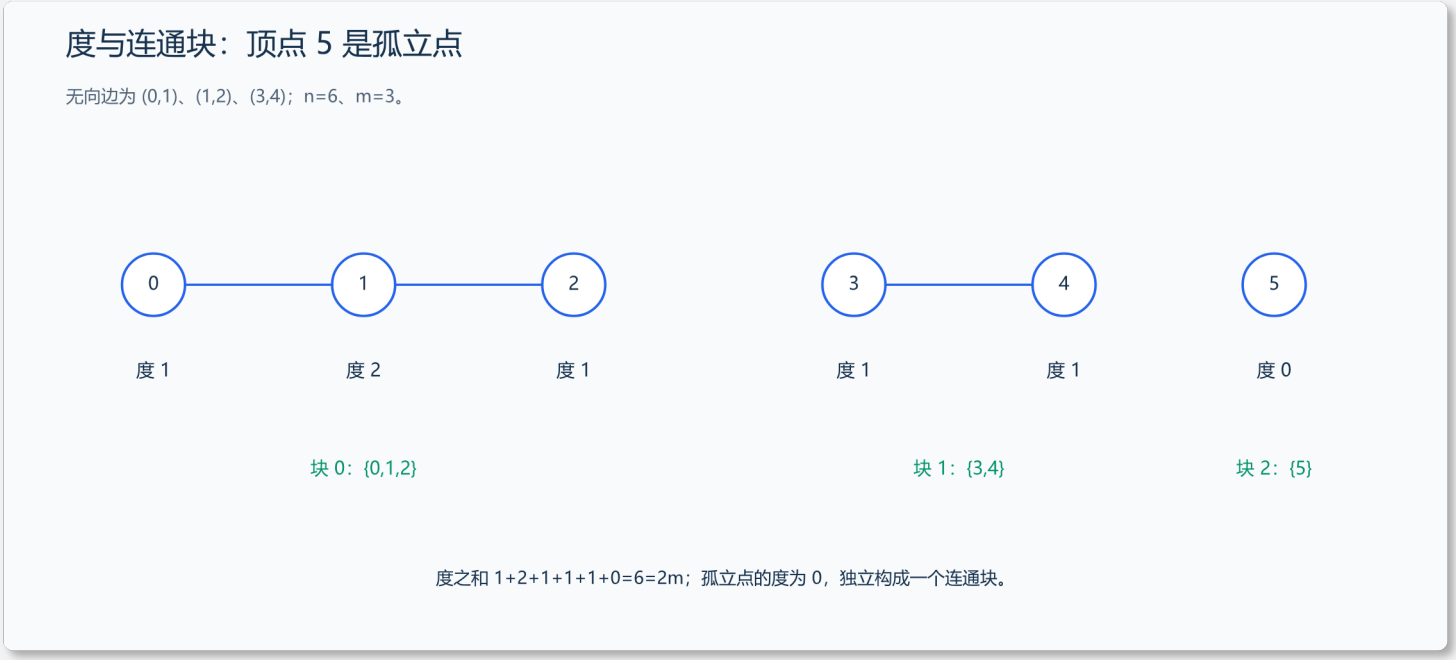
1.2 度、自环、重边与简单图

无向图中，**度 deg(u)** 是连接到 u 的边端点数量。边 (u,v) 各给两端增加 1；自环 (u,u) 在同一顶点贡献 2。因此所有顶点的度之和为 $2m$，因为每条边有两个端点。孤立点的度为 0，它仍然是图中的一个顶点。

有向图分入度和出度：u→v 给 u 的出度加 1、v 的入度加 1；所有入度之和与所有出度之和均为 m。入度为 0 不等于孤立，它可以有出边；出度为 0 也可能有入边。

自环的两端相同，**重边**是同一对端点之间存在多条边。无自环、无重边的图称为简单图；有向图的 u→v 与 v→u 是不同方向，不能把它们当作同一条重边。

无向图存自环时也写两次，邻接表长度与度相符；最短路可以保留重边或取最小边权；拓扑排序读入一条记录时给终点入度加一，处理起点时再沿这条记录给终点入度减一。二分图中的自环直接造成矛盾，因为顶点不能与自己颜色不同。



每条边给两端各贡献一次度；孤立点单独构成一个连通块。

1.3 走法、路径、环与距离

一串按边连接的顶点 `v0→v1→…→vk` 是合法走法。一般走法允许重复顶点或边；**简单路径**不重复顶点。不同教材对“路径”的用词略有差别，解题时应明确题目是否允许重复，本章求非负最短路时可以选取不绕环的最优路径。

路径的边数长度为 k，不是 k+1；加权费用是 k 条边权之和。例如 `0→2→1→3` 经过 4 个顶点、3 条边，边权为 1、2、1 时费用为 4。**环**是在合法方向下绕行回到起点的关系；无向图沿同一条边走过去再回来，不算由不同边构成的普通环。判断无向环时需要区别父边与额外边，重边还需要区别具体的边记录。

可达表示存在至少一条合法路径；**距离**是所有合法路径中最小的长度或代价。没有路径时用 `-1` 或 INF 表示不可达。起点到自己允许不走任何边，费用为 0。

负权边不等于负环。若能从起点到一个负环，再从该环到目标，反复绕环可让费用无限降低，目标没有有限最短距离。下面的道路费用都是非负数，多走一条边不会让累计费用降低，Dijkstra 正是利用这个性质逐步确定最短距离。Floyd 则枚举中转点组合路径；负环对路径的影响，会从“反复绕环”的变化来判断。

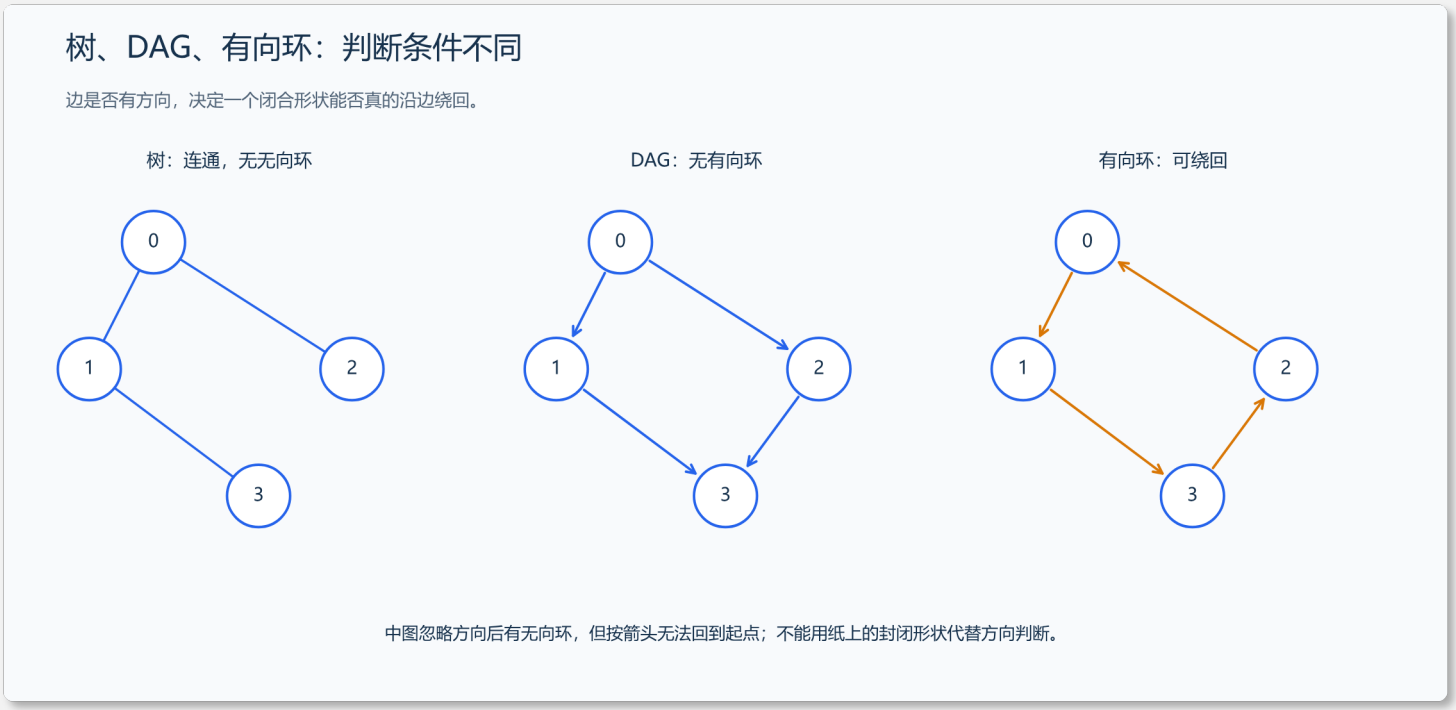
1.4 连通性、树与 DAG

无向图中，彼此可达的顶点构成一个**连通块**，不同块之间没有边。只有一个连通块的无向图是连通图；孤立点独自构成一个块。

有向图的箭头决定行走方向。例如只有 `0→1` 时，0 能到 1，1 没有返回 0 的边。任意两点互相可达叫强连通；忽略箭头后连通叫弱连通。从起点做一次 DFS 会得到这个起点能到达的点；在 `0→1` 中两点都被访问，但它们没有互相可达，因此是两个强连通分量。

树是连通且无环的无向图，n 个顶点恰有 n-1 条边，两点间有唯一简单路径。**森林**无环但可以不连通，有 c 个连通块时边数为 n-c。仅“边数为 n-1”不足以判断是树：一个三角形加一个孤立点有 4 个顶点、3 条边，却不连通且有环。

DAG（有向无环图）有方向且无有向环。忽略方向后仍可能含环，例如 `0→1、0→2、1→3、2→3`；沿方向不能绕回起点，因此是 DAG，可以用拓扑序安排状态计算。



树的无环与 DAG 的无有向环不是相同条件。

2. 建图过程与存储方式选择

2.1 邻接表、邻接矩阵与边表

图的输入通常是一组边，但搜索时经常要回答“从顶点 u 可以直接走到哪些点”。**建图**就是把输入整理成方便回答这些问题的数据结构。顶点编号只是下标约定，不代表空间位置；下面统一使用 `0…n-1`。

2.1.1 邻接表：每个顶点有一张邻居列表

可以把邻接表想成一本通讯录：每个顶点有自己的条目，条目内只记它能直接到达的邻居。`vector<vector<int>> graph(n)` 的外层有 n 个位置，`graph[u]` 是顶点 u 的内层动态数组，开始时为空；每读到一条边，就向对应列表追加终点。

这里的“表”指邻居列表，本例用 vector 实现；学习邻接表时，要先理解“每个顶点分别保存自己的邻居”，再看它采用哪种容器。

例如无向边为 `(0,1)、(0,2)、(1,3)`，最终得到：

外层下标	该顶点的邻居列表	含义
0	`[1,2]`	从 0 可以一步到 1 或 2
1	`[0,3]`	从 1 可以一步到 0 或 3
2	`[0]`	从 2 可以一步到 0
3	`[1]`	从 3 可以一步到 1

`graph[0][1]=2` 的意思是“顶点 0 的第二个邻居是 2”，第二个下标是**列表里的位置**，不是目标顶点编号。因此，不能用 `graph[u][v]` 直接判断 u 与 v 是否有边；要在 `graph[u]` 中查找值 v。没有邻居的顶点仍有一个空列表。



读 `(0,1)` 时，向 `graph[0]` 追加 1、向 `graph[1]` 追加 0；读 `(0,2)` 后，只改变第 0、2 行；读 `(1,3)` 后，只改变第 1、3 行。每一步只保存这条边真正涉及的邻居。

无向边允许从两端互相到达，所以追加两次；有向边 `u→v` 只向 `graph[u]` 追加 v。普通 vector 邻接表按追加顺序存储，不会自动排序。它也会保留重边；无向自环执行两次追加，正好对应自环对度的两次贡献。

2.1.2 邻接矩阵：每一对顶点预留一个格子

邻接矩阵是一张 n 行、n 列的表。**行 u 表示出发点，列 v 表示到达点，** `matrix[u][v]` 直接回答这一对顶点的关系。在无权“是否有边”的版本中，先全部填 0；读到 `u→v` 后写 1。

同一组无向边得到：

出发点 \ 到达点	0	1	2	3
0	0	1	1	0
1	1	0	0	1
2	1	0	0	0
3	0	1	0	0

例如 `matrix[0][2]=1` 表示 0、2 直接相邻；`matrix[1][2]=0` 表示没有直接边，即使它们能经 0 到达，也不能把这一格写成 1。这里记录的是**直接边**，不是整个图的可达关系。

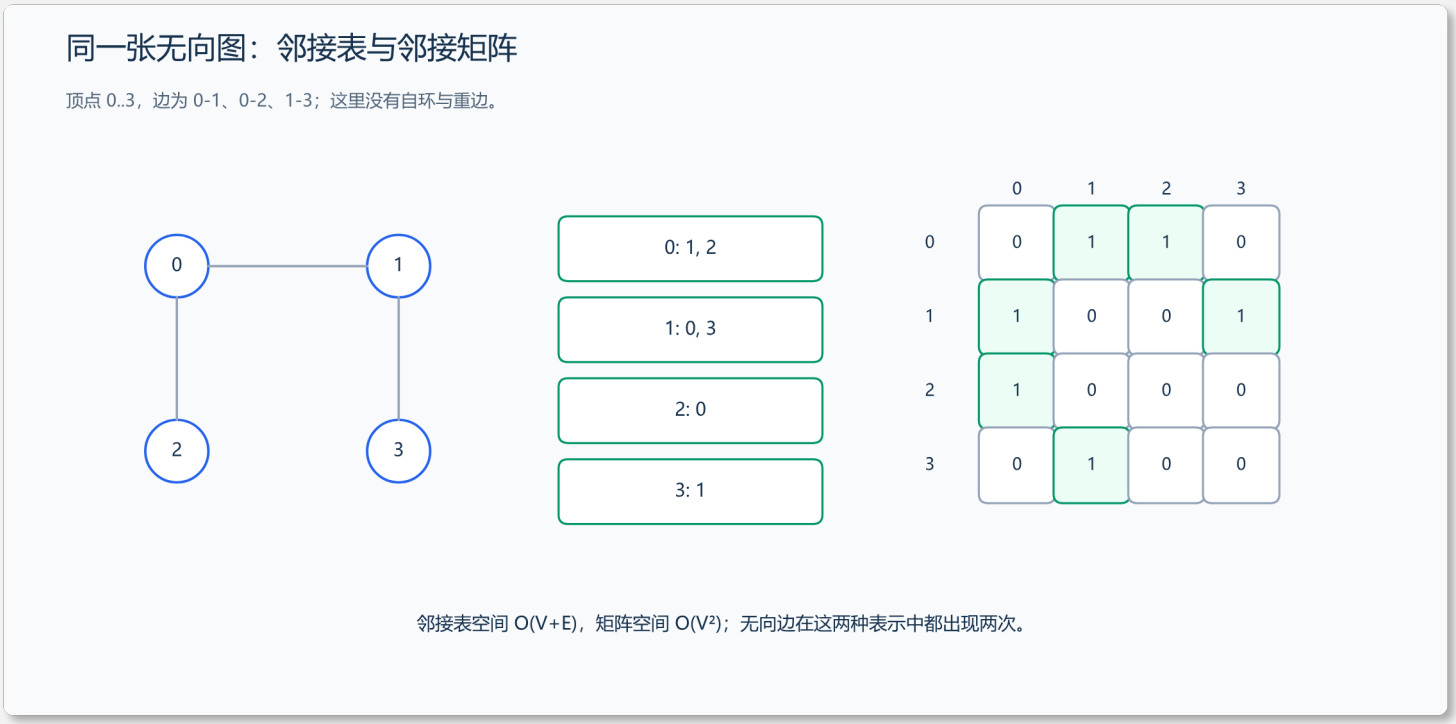


无向边 `(0,1)` 要写 `matrix[0][1]=1` 与 `matrix[1][0]=1`，因此无向图的矩阵关于主对角线对称。有向边只写规定方向，矩阵不一定对称。普通无权存在性矩阵遇到重边仍写 1，只保留“存在”，不保留条数；若需要重边数量，应另约定格子存计数并累加。

这个版本的对角线初始为 0，读到自环后对应格子变成 1。不能把它与后面最短路矩阵的“对角线为 0”混淆：那里 0 表示不移动的空路径代价，格子的含义已经改变。

2.1.3 边表：每条输入边单独存一条记录

边表直接保存 `(u,v)` 或 `(u,v,w)`，像把输入行收进一个数组。它方便枚举全部边、按权值排序，所以 Kruskal 常用边表；但若每次想找 u 的邻居，通常要扫过全部记录。无向边在边表中可只保存一次，“写两个方向”是为了邻居查询，不能把所有表示都机械地加倍。



图的结构相同，存储位置的含义不同。邻接表的各行长度随邻居数变化；矩阵的每行固定有 n 格，包括没有边的位置。

表示	空间	枚举 u 的邻居	查询 u、v 是否直接相邻	常见场景
邻接表	$O(n+m)$	$O(\deg(u))$；有向图看出度	普通 vector 需扫描 u 的列表	搜索、Dijkstra、拓扑排序
邻接矩阵	$O(n^2)$	扫描第 u 行，$O(n)$	读一个格子，$O(1)$	小规模稠密图、Floyd
边表	$O(m)$，另计顶点数据	通常需扫全部边，$O(m)$	通常需扫全部边	Kruskal 全边排序

n=100000、m=200000 时，邻接表随 n+m 增长；矩阵却有 10^{10} 个格子，每格 4 B 时约 37.25 GiB。存储选择先看规模和需要的操作：仅有少量边、主要枚举邻居时用邻接表；需要频繁读取所有点对且 n 较小时再考虑矩阵。空间还要计入 vector 对象、分配容量、队列和距离数组。

2.2 完整例题：读入并输出邻接表

题目原文（自拟）： 第一行输入 n m，$1 \le n \le 1000$、$0 \le m \le 10000$；随后 m 行各输入 u v，编号在 0…n-1，表示无向边，允许重边、自环。保留全部边记录，按顶点编号逐行输出 `u:`，后面按输入顺序输出 u 的所有邻居；没有邻居只输出 `u:`。

推导： 先创建 n 个空列表，每条无向边对 u 写 v、对 v 写 u。自环同样写两次；m 是输入边数，所有列表合计有 2m 条记录。每条边都按图中的两个追加动作处理。

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

int main() {
    int n, m;
    cin >> n >> m;
    vector<vector<int>> graph(n); // 外层 n 个顶点，内层邻居列表初始为空。
    for (int i = 0; i < m; ++i) {
        int u, v;
        cin >> u >> v;
        graph[u].push_back(v); // 从 u 可以直接到 v。
        graph[v].push_back(u); // 无向边还允许反方向；自环也按此写两次。
    }
    for (int u = 0; u < n; ++u) {
        cout << u << ':';
        // v 是邻居编号，依次取出 graph[u] 的元素即可枚举出边。
        for (int v : graph[u]) cout << ' ' << v;
        cout << '\n';
    }
}
```

输入：

```text
4 3
0 1
0 2
1 3
```

输出：

```text
0: 1 2
1: 0 3
2: 0
3: 1
```

代码与复杂度： `graph[u].push_back(v)` 新建一条从 u 到 v 的邻接记录；输出时枚举顶点，再枚举该列表。建图和输出共 $O(n+m)$ 时间、$O(n+m)$ 空间。m=0 时仍输出 n 行空列表。有向图只保留第一条追加语句；题目按 1 编号时可读入后减一，或统一开 n+1 个位置并枚举 1…n，不能只改一处。

2.3 完整例题：读入并输出邻接矩阵

题目原文（自拟）： 第一行输入 n m，`1≤n≤1000、0≤m≤10000`；随后 m 行输入 `0…n-1` 范围内的 u v，表示无向边，允许重边、自环。输出 n 行，每行 n 个整数，用空格分隔；第 u 行第 v 个整数为 1 当且仅当 u、v 之间存在直接边，否则为 0。重边只表示一次存在，自环使对应对角格为 1。

分析： 每个点对预留一格，初值全为 0。输入 `(u,v)` 时填写两个对称位置；重复赋 1 不会增加数值。读取 `matrix[u][v]` 可直接判定是否有边，枚举 u 的全部邻居则需检查整行的 n 个格子。

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

int main() {
    int n, m;
    cin >> n >> m;
    vector<vector<int>> matrix(n, vector<int>(n, 0)); // 每个点对都有一个格子。
    for (int i = 0; i < m; ++i) {
        int u, v;
        cin >> u >> v;
        matrix[u][v] = 1; // 行是出发点，列是到达点。
        matrix[v][u] = 1; // 无向边要对称填写；自环是对同一格赋值。
    }
    for (int u = 0; u < n; ++u) {
        for (int v = 0; v < n; ++v) {
            if (v > 0) cout << ' ';
            cout << matrix[u][v]; // 0 表示没有直接边，不表示两点完全不可达。
        }
        cout << '\n';
    }
}
```

输入：

```text
4 3
0 1
0 2
1 3
```

输出：

```text
0 1 1 0
1 0 0 1
1 0 0 0
0 1 0 0
```

样例推导与复杂度： 逐边填写的位置正是矩阵过程图中的绿色格；`(0,2)` 只写第 0 行第 2 列及第 2 行第 0 列，不会产生 `(1,2)`。初始化与完整输出都访问 n² 格，总时间 $O(n^2+m)$，额外空间 $O(n^2)$。矩阵中的行、列下标都是顶点编号，与邻接表的第二下标含义不同。

2.4 带权图的终点、权值与重边存储

带权邻接表不能只保存终点，因为最短路还需要知道这条边的代价。把一条记录定义为 `{to,weight}`，`graph[u]` 中的每个元素就同时包含终点和权值。边的两个端点相同而权值不同，仍是两条记录。

用于最短路的初始矩阵则保存“目前已知的最小代价”：对角线设 0，表示空路径；没有直接边的其他位置设 `INF`；读入重边时取最小权值。**零权边是合法边，不能用 0 表示没有边。** 这种矩阵会合并重边，也不保留“对角线为 0 是否源于自环”的明细；需要边的完整信息时保留邻接表或边表。



例如有向边 `(0,1,5)、(0,1,2)、(1,2,0)`：第 0 个邻接表保留 `(1,5)、(1,2)`，矩阵中 `initialDistance[0][1]` 从 INF 改为 5，再改为 2；第 1 个列表保存 `(2,0)`，矩阵的 `(1,2)` 为 0，确实存在一条免费边。

代码示例（建图函数）： 输入 n 个顶点、合法边三元组及方向标志，返回两种表示，便于小图对照。此教学函数约定 `1≤n≤1000`、权值在 `[0,10^9]`；大规模图按需求单独构造邻接表。

```cpp
#include <algorithm>
#include <tuple>
#include <vector>
using namespace std;

struct WeightedEdge {
    int to;           // 终点，不是邻接表内的位置。
    long long weight; // 从所属列表的顶点走向 to 的代价。
};

struct WeightedGraph {
    vector<vector<WeightedEdge>> list;
```

```cpp
    vector<vector<long long>> initialDistance;
};

WeightedGraph buildWeightedGraph(int n, const vector<tuple<int, int, long long>> &edges,
                                 bool directed) {
    constexpr long long INF = 1LL << 60; // 无直接路径，与合法的零权边区分。
    WeightedGraph result{vector<vector<WeightedEdge>>(n),
                         vector<vector<long long>>(n, vector<long long>(n, INF))};
    for (int u = 0; u < n; ++u) result.initialDistance[u][u] = 0; // 空路径代价。
    for (auto [u, v, w] : edges) {
        result.list[u].push_back({v, w}); // 每条边都保留完整记录，包括重边。
        result.initialDistance[u][v] = min(result.initialDistance[u][v], w);
        if (!directed) {
            result.list[v].push_back({u, w}); // 无向图补反方向；有向图不补。
            result.initialDistance[v][u] = min(result.initialDistance[v][u], w);
        }
    }
    return result;
}
```

cpp

调用与解释： `buildWeightedGraph(3, {{0,1,5},{0,1,2},{1,2,0}}, true)` 得到图中列表和初始矩阵 `[[0,2,INF],[INF,0,0],[INF,INF,0]]`。`initialDistance[0][2]` 仍是 INF，因为尚未执行最短路；通过 1 到 2 的费用 2，要由后面的算法推导出来。若方向标志改为 false，每条边再向相反方向存一次。

复杂度： 构造列表部分为 $O(n+m)$，构造矩阵部分为 $O(n^2+m)$；同时返回两者时，总时间与额外空间均为 $O(n^2+m)$。后续遍历带权邻接表时，使用 `for (auto [v,w] : graph[u])` 取出邻居和权值，不要把一条记录当成单个编号。

练习题： 洛谷 B3643 – 图的存储（同时练习矩阵与列表，注意按题面调整编号、排序和输出格式）。

3. 用搜索标记无向图的连通块

题目原文（自拟）： 输入无向图邻接表 `graph`，n ≥ 1，编号合法，允许重边与自环。返回各顶点所在连通块编号；按枚举起点 `0…n-1` 的顺序，从 0 开始给块编号，孤立点也应得到编号。

思路： 从未标记顶点做一次 BFS，得到且仅得到这个起点可达的整块顶点，给它们同一个编号；再寻找下一个未标记顶点。只搜 0 会漏掉其他块，外层起点枚举不能省略。



一次搜索覆盖一个完整块，下一起点必然属于尚未覆盖的块。

```cpp
#include <queue>
#include <vector>
using namespace std;

vector<int> componentIds(const vector<vector<int>> &graph) {
    int n = graph.size(), count = 0;
    vector<int> id(n, -1);
    for (int start = 0; start < n; ++start) {
        if (id[start] != -1) continue;
        queue<int> q;
        id[start] = count; // 当前起点对应一个新连通块。
        q.push(start);
        while (!q.empty()) {
            int u = q.front();
            q.pop();
            for (int v : graph[u]) {
                if (id[v] != -1) continue;
                id[v] = count; // 入队时标记，每个顶点只处理一次。
                q.push(v);
            }
        }
        ++count;
    }
    return id;
}
```

cpp

样例推导： n=6，边为 `(0,1)、(1,2)、(3,4)`，结果为 `[0,0,0,1,1,2]`。从 0 搜到 1、2；枚举到 1、2 时跳过；从 3 搜到 4；最后 5 独自成为第三块。两点编号相同当且仅当它们在无向图中连通。

复杂度为什么不是 n 次 BFS？ 每点标记后不再展开，每条邻接表记录也只在所属顶点处理时检查一次，所有搜索总时间 $O(n+m)$，额外空间 $O(n)$，另计图存储。静态图搜索可保留父边；动态加边并查询连通时可用并查集。

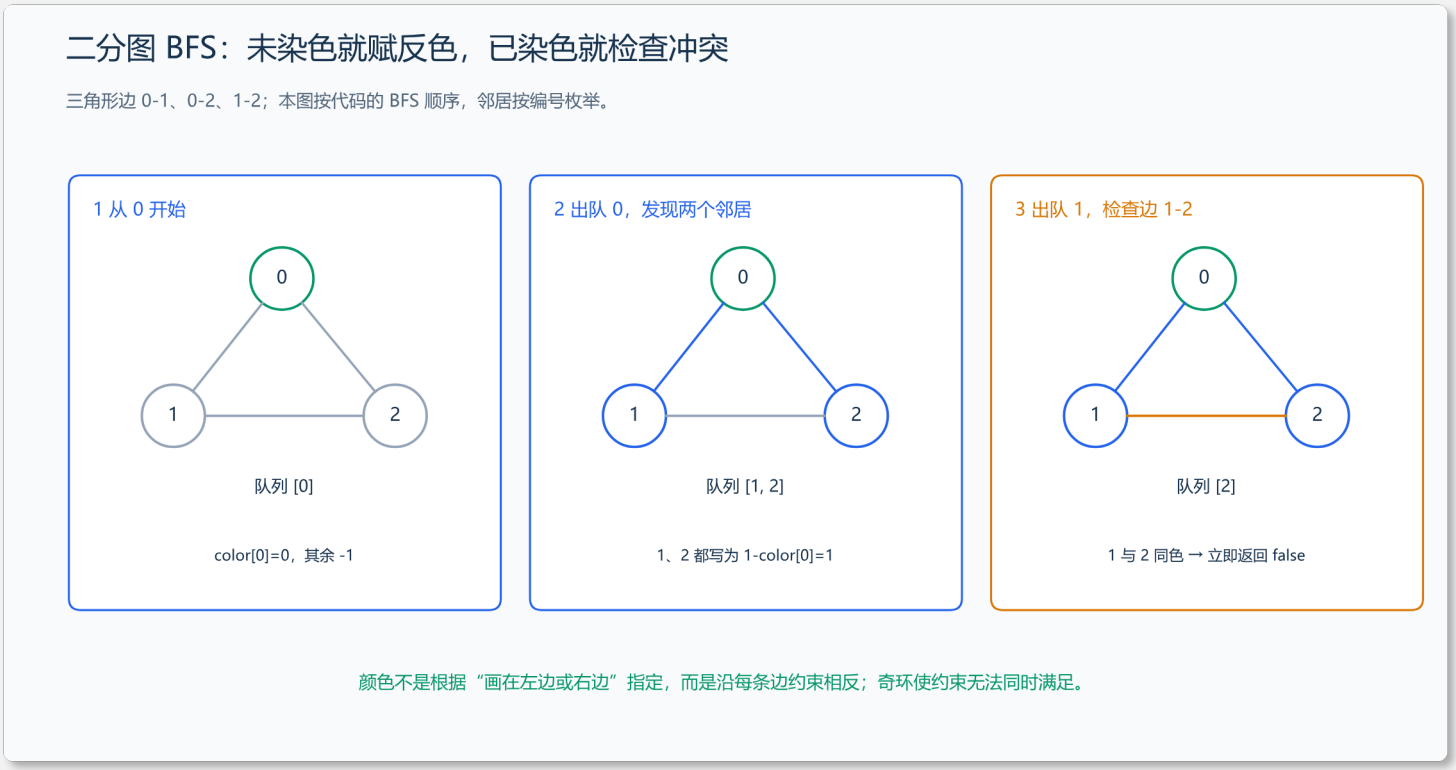
练习题： 洛谷 P1451 — 求细胞数量、洛谷 P1596 — 湖计数（允许八方向连接）。

4. 二分图：沿边给相邻顶点分组

二分图能把顶点分为两个互不相交的集合，使每条边的两端属于不同集合。它是约束能否成立的性质，和把图画在纸的左右两侧无关。

题目原文（自拟）： 输入无向图邻接表 `graph`，n≥1、编号合法，允许重边和自环。能否给所有顶点赋 0 或 1，使每条边两端异色？返回 `pair<bool,vector<int>>`；可行时为 true 和一种染色，否则为 false 和空数组。

分析： 每个未染色连通块任选一个起点为 0，沿边要求邻居为相反颜色。未染色则赋值，已染色则核对要求；遇到同色边立即失败。整块颜色全部翻转也不能消除同色边。



图按 BFS 实际执行三角形 `(0,1)、(0,2)、(1,2)`：先给 0 染色 0，出队后把 1、2 都染成 1 并入队；处理 1 时，邻居 2 已是 1，和当前颜色相同，立即返回失败。不是事后把三角形涂色得出结论，而是每次检查边都要求两端异色。四边形样例则可以完整通过这些检查。

```cpp
#include <queue>
#include <utility>
#include <vector>
using namespace std;

pair<bool, vector<int>> bipartiteColors(const vector<vector<int>> &graph) {
    vector<int> color(graph.size(), -1);
    for (int start = 0; start < static_cast<int>(graph.size()); ++start) {
        if (color[start] != -1) continue;
        queue<int> q;
```

```cpp
        color[start] = 0; // 不同连通块分别选起点颜色。
        q.push(start);
        while (!q.empty()) {
            int u = q.front();
            q.pop();
            for (int v : graph[u]) {
                if (color[v] == -1) {
                    color[v] = color[u] ^ 1; // 在 0、1 之间翻转。
                    q.push(v);
                } else if (color[v] == color[u]) {
                    return {false, {}}; // 当前边的异色约束无法满足。
                }
            }
        }
    }
    return {true, color};
}
```

样例： 边 `(0,1)、(1,2)、(2,3)、(3,0)` 返回 `[0,1,0,1]`。若再加 `(0,2)`，两点都与 1 相邻，染成了同一种颜色，但新增边要求异色，返回失败。

奇环为什么失败？ 沿每条边翻转一次，走奇数条边回到起点会要求起点与自身异色；偶数环绕回时没有这个矛盾。反过来，同色冲突边结合搜索树中两端的路径会构成奇环。无向图可二染色当且仅当无奇环，自环也应失败。

代码与复杂度： 外层枚举覆盖所有块和孤立点，`^1` 仅用于已经确定为 0、1 的颜色。每点和每条边记录检查一次，时间 $O(n+m)$，额外空间 $O(n)$。

练习题： 洛谷 P1330 - 封锁阳光大学（每块统计两侧人数，选择较少一侧，奇环无解）。

5. Dijkstra：非负边权单源最短路

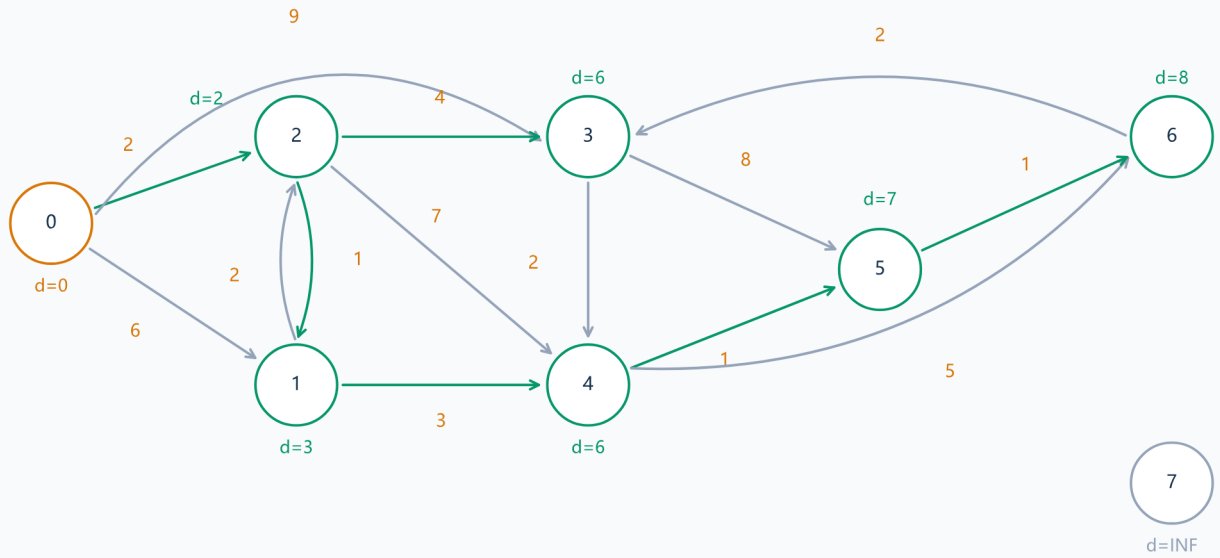
题目原文（自拟）： 第一行输入 n m start，`1≤n≤100000、0≤m≤200000`，start 和顶点编号在 `0…n-1`。随后 m 行输入 u v weight，表示有向边，`0≤weight≤10⁹`，允许重边、自环。求固定起点 start 到每个顶点的最短费用，按顶点顺序输出；不可达输出 -1。起点到自己费用为 0。

单源究竟求什么？ “单”限制起点只有一个，不是只求一个终点。固定起点 0 后，会得到到 0、1、2、… 的整份距离数组；改成从 2 出发，要重新计算对应答案。下面用 8 个顶点、14 条边，有替代路线、双向关系、有向环和一个孤立点。

状态与松弛： `distance[v]` 是目前找到的起点到 v 的最小候选费用，初始只有起点为 0，其余 INF。已知一条到 u 的路径，再走边 u→v 可构造候选 `distance[u]`+weight；更小时替换旧值，这叫松弛。例如先沿直接边 0→1 得到费用 6，之后发现 0→2→1 只花 2+1=3，就把候选从 6 更新为 3。

Dijkstra 单源最短路：8 个顶点、14 条边，固定从 0 出发

橙色点是起点；绿色点距离已确定；绿色边为最优来源；孤立点 7 不可达。



逐轮松弛记录：表中只列有效堆记录，过期记录直接跳过

初始化	只有起点 0 的距离为 0	[0,INF,INF,INF,INF,INF,INF,INF]
取出 (0,0)	1←6、2←2、3←9	[0,6,2,9,INF,INF,INF,INF]
取出 (2,2)	1←3、3←6、4←9	[0,3,2,6,9,INF,INF,INF]
取出 (3,1)	4←6	[0,3,2,6,6,INF,INF,INF]
取出 (6,3)	5←14	[0,3,2,6,6,14,INF,INF]
取出 (6,4)	5←7、6←11	[0,3,2,6,6,7,11,INF]
取出 (7,5)	6←8	[0,3,2,6,6,7,8,INF]
取出 (8,6)	出边不再改善距离	[0,3,2,6,6,7,8,INF]

直接到 1 的费用 6，被 0→2→1 的费用 2+1=3 改善；到 6 的最短费用为 8。

旧记录 (6,1)、(9,3)、(9,4)、(11,6)、(14,5) 均过期；不能据首次入堆就确定距离。

这一份数组只回答“从 0 出发”；换起点为 2，答案变为 [INF,1,0,4,4,5,6,INF]。

图中起点始终为 0，绿色边组成它的最短路径树；格外注意到 3 与到 4 的路线发生分叉，孤立点 7 保持不可达。

为什么取出最小有效候选就能确定距离？ 设已确定点集为 S，即将处理顶点 u 的候选为 d。假设存在更短路径，从起点沿它找到第一个不属于 S 的顶点 v；进入 v 的前驱在 S 中，它确定时已松弛这条边。由于后续边权非负，v 的候选不大于这条更短路径的总费用，也就小于 d；堆应先处理 v，与 u 当前最小矛盾。因此 d 已是最终距离。负边会破坏“后续费用不再降低”的条件。

有效取出记录	本轮得到的新候选	distance
初始化	只有起点 0 的距离为 0	[0,INF,INF,INF,INF,INF,INF,INF]
取出 (0,0)	1←6、2←2、3←9	[0,6,2,9,INF,INF,INF,INF]
取出 (2,2)	1←3、3←6、4←9	[0,3,2,6,9,INF,INF,INF]
取出 (3,1)	4←6	[0,3,2,6,6,INF,INF,INF]
取出 (6,3)	5←14	[0,3,2,6,6,14,INF,INF]
取出 (6,4)	5←7、6←11	[0,3,2,6,6,7,11,INF]
取出 (7,5)	6←8	[0,3,2,6,6,7,8,INF]
取出 (8,6)	出边不再改善距离	[0,3,2,6,6,7,8,INF]

旧记录为什么仍在堆里？ 更新顶点 1 时，把新记录 (3,1) 加入堆，旧记录 (6,1) 仍存在。后续取出旧项发现 `current≠distance[1]`，直接跳过，不重复扫描出边。其他过期记录同理。零权边可使用；只有严格改善时入堆，避免相等候选反复扩展。

恢复路径： 每次松弛成功时另记 `parent[v]` =u。终点可达时沿父指针回到起点并反转；不可达时不回溯。只求距离的示例省略 parent，图中的绿色边展示这一来源关系。

```cpp
#include <algorithm>
#include <functional>
#include <iostream>
#include <limits>
#include <queue>
#include <utility>
#include <vector>
using namespace std;
using ll = long long;
constexpr ll INF = numeric_limits<ll>::max() / 4;

struct Edge {
    int to;
    ll weight;
};

vector<ll> dijkstra(const vector<vector<Edge>> &graph, int start) {
    vector<ll> distance(graph.size(), INF);
    priority_queue<pair<ll, int>, vector<pair<ll, int>>, greater<pair<ll, int>>> pending;
    distance[start] = 0;
    pending.push({0, start});
    while (!pending.empty()) {
        auto [current, u] = pending.top();
```

```cpp
        pending.pop();
        // 堆中可能有旧候选距离，过期记录直接跳过。
        if (current != distance[u]) {
            continue;
        }
        for (const Edge &edge : graph[u]) {
            if (edge.weight < INF - current && current + edge.weight < distance[edge.to]) {
                distance[edge.to] = current + edge.weight;
                pending.push({distance[edge.to], edge.to});
            }
        }
    }
    return distance;
}

int main() {
    int n, m, start;
    cin >> n >> m >> start;
    vector<vector<Edge>> graph(n);
    while (m--) {
        int u, v;
        ll weight;
        cin >> u >> v >> weight;
        graph[u].push_back({v, weight});
    }
    auto distance = dijkstra(graph, start);
    for (int i = 0; i < n; ++i) {
        if (i != 0) {
            cout << ' ';
        }
        cout << (distance[i] == INF ? -1 : distance[i]);
    }
    cout << '\n';
}
```

输入：

```text
8 14 0
0 1 6
0 2 2
0 3 9
1 2 2
1 4 3
2 1 1
2 3 4
2 4 7
3 4 2
3 5 8
4 5 1
4 6 5
5 6 1
6 3 2
```

输出：

```
1  0 3 2 6 6 7 8 -1
```
text

样例解释： 到 1 为 0→2→1，费用 3；到 3 为 0→2→3，费用 6；到 6 为 0→2→1→4→5→6，费用 2+1+3+1+1=8。到 7 无路，图中 INF 对应程序输出 -1。换起点 2，距离为 [INF,1,0,4,4,5,6,INF]，这与下一节矩阵的第 2 行对应。

代码与复杂度： 堆中保存（候选费用,顶点），先取费用较小的记录；发现更短路径后重新入堆，取到过期记录时跳过。INF 用来标记尚未找到路径的点，代码把实际路径费用与这个标记分开处理。重复入堆实现的时间为 $O((n+m)\log(n+m))$、空间 $O(n+m)$；简单图常写 $O((n+m)\log n)$。无向道路在邻接表中保存两个方向，处理过程相同。

练习题： 洛谷 P3371 — 单源最短路径（弱化版）、洛谷 P4779 — 单源最短路径（标准版）。

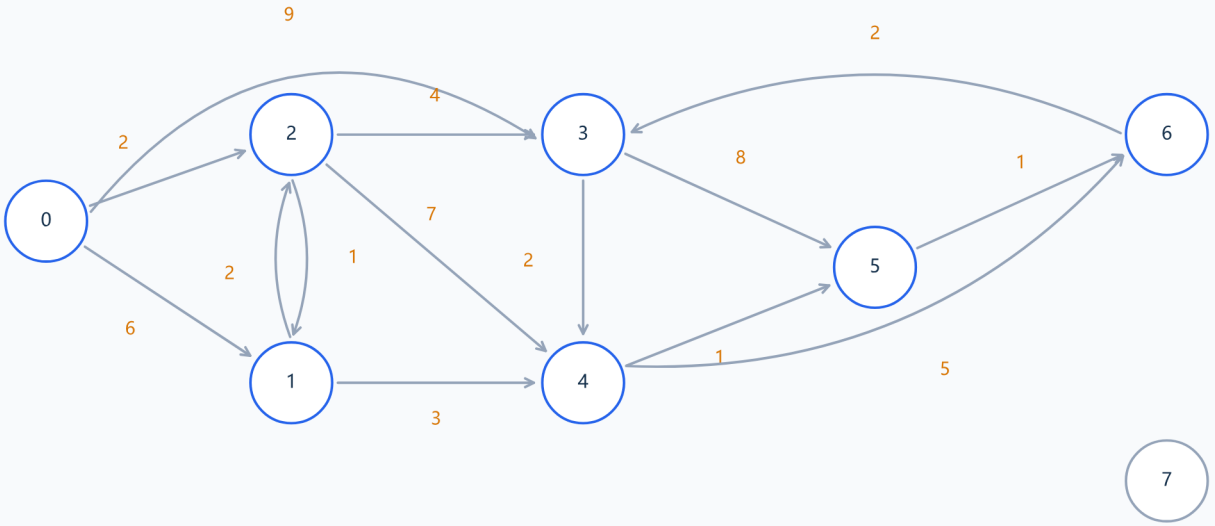
6. Floyd：所有点对最短路与距离矩阵

题目原文（自拟）： 输入一个 n×n 的有向图距离矩阵 distance，$1 \le n \le 500$。对角线为 0，非负边权不超过 10^9，无边用 `numeric_limits<long long>::max()/4` 表示，重边预先取最小权值。求任意起点到任意终点的最短费用，原地更新矩阵；不可达格仍为 INF。示例使用与上一节完全相同的 8 点 14 边图。

所有点对的结果长什么样？ 让每个顶点分别作起点，求它到每个终点的距离，得到所有点对最短路，也称全源最短路。`distance[i][j]` 的行 i 是起点，列 j 是终点；整张矩阵包含 n 份单源答案。第 0 行等于上一节从 0 出发的数组，第 2 行等于从 2 出发的数组。即使某个点与起点 0 无法互达，也有以它为起点的一行，记录它到其他各点的距离。

Floyd 多源最短路：同一张 8 点 14 边的图，计算全部起点与终点

每行 i 是起点、每列 j 是终点；∞ 表示不可达；第 0 行用橙色表示它与单源答案的对应关系。



初始矩阵：对角线 0，其他只有直接边

	0	1	2	3	4	5	6	7
0	0	6	2	9	∞	∞	∞	∞
1	∞	0	2	∞	3	∞	∞	∞
2	∞	1	0	4	7	∞	∞	∞
3	∞	∞	∞	0	2	8	∞	∞
4	∞	∞	∞	∞	0	1	5	∞
5	∞	∞	∞	∞	∞	0	1	∞
6	∞	∞	∞	2	∞	∞	0	∞
7	∞	∞	∞	∞	∞	∞	∞	0

最终矩阵：全部中转点已允许

	0	1	2	3	4	5	6	7
0	0	3	2	6	6	7	8	∞
1	∞	0	2	6	3	4	5	∞
2	∞	1	0	4	4	5	6	∞
3	∞	∞	∞	0	2	3	4	∞
4	∞	∞	∞	4	0	1	2	∞
5	∞	∞	∞	3	5	0	1	∞
6	∞	∞	∞	2	4	5	0	∞
7	∞	∞	∞	∞	∞	∞	∞	0

部分中转轮次怎样同时改善不同起点的答案

k=1：d[0][4] = 6+3 = 9；d[2][4] = 1+3 = 4

k=2：d[0][1] = 2+1 = 3；d[0][3] = 2+4 = 6；d[0][4] = 2+4 = 6

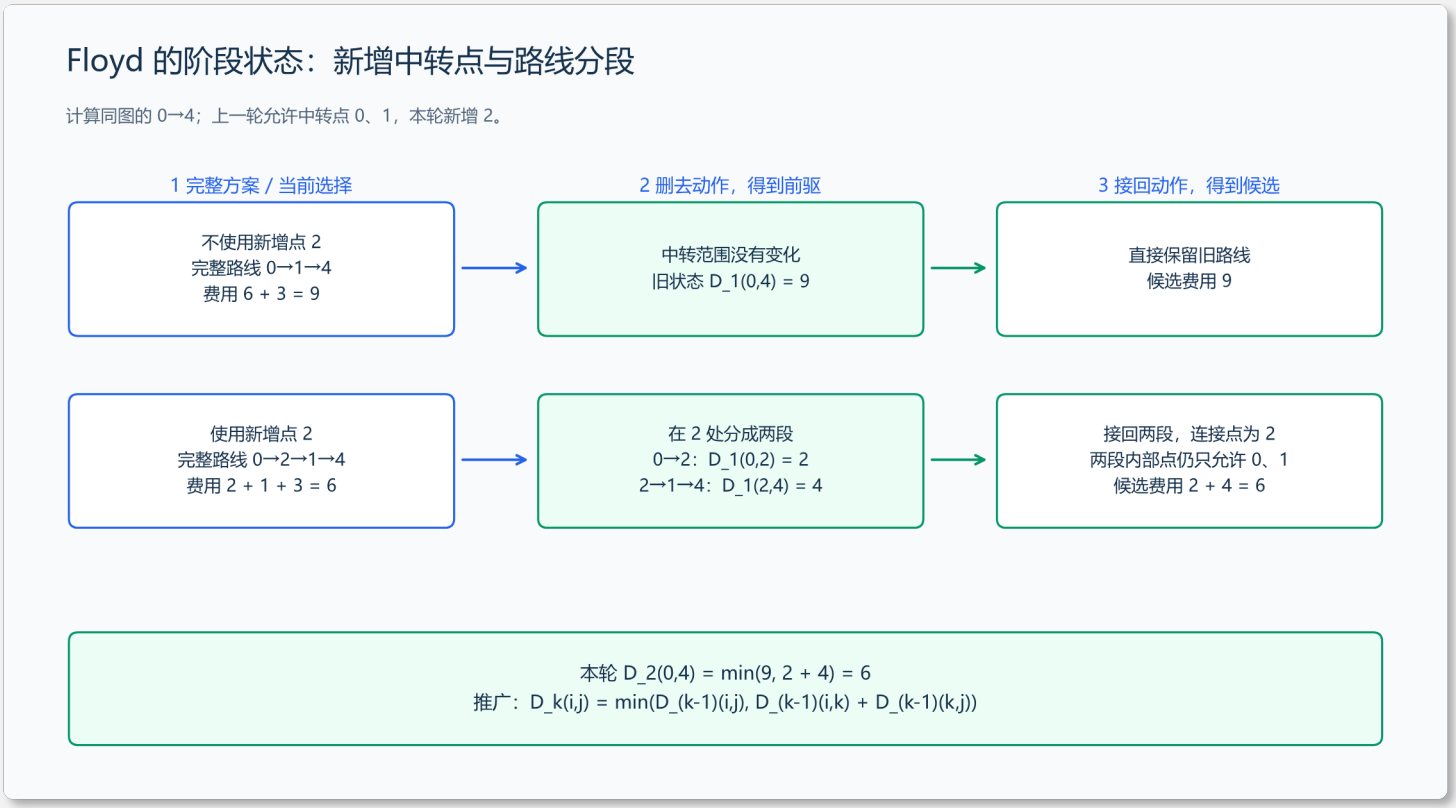
k=4、5：依次改善到 5、6 的距离；k=6：d[4][3] = 2+2 = 4

最终第 0 行等于 Dijkstra 从 0 出发的结果；其余 7 行分别是另外 7 个起点的距离。

绿色格是相对初始边权改善的距离；第 7 行只有自身为 0；有向图中 d[i][j] 未必等于 d[j][i]。

图中每个节点都可作为起点。初始只知道直接边，最终矩阵保存绕中转点后的距离；橙色第 0 行说明单源结果只是完整矩阵中的一行。

从具体路线到中转状态： 计算同图中 `0→4` 的距离。允许中转点 1 后，路线 `0→1→4` 费用为 `6+3=9`；从 2 到 4 则可经 `2→1→4`，费用为 `1+3=4`。现在新增允许中转点 2，出现新的路线 `0→2→1→4`。



把候选路线在 2 处分开，前半段 `0→2` 费用 2，后半段 `2→1→4` 费用 4。两段的内部顶点仍只使用上一轮允许的点，因此前驱已经算好，接起来得到 `2+4=6`。再与不使用新增点 2 的旧答案 9 比较，当前距离改为 6。

这里单独记录 `(i,j)` 还没有交代“哪些中转点已经可以使用”。完整状态写成 `D_k(i,j)`：起点 i、终点 j，路径内部顶点只从 `0…k` 中选择的最短费用。起终点本身不受这个中转范围限制，例如上一轮也可以把 2 作为终点。

本轮候选类别	允许的内部点	前驱与合并
不使用新增中转点 k	仍从 `0…k-1` 选择	沿用 `D_{k-1}(i,j)`
使用新增中转点 k	在 k 处分段，两段内部点仍从 `0…k-1` 选择	`D_{k-1}(i,k)+D_{k-1}(k,j)`

任意经过 k 的最短路线都能在 k 处分成两段。若某段不是相应状态的最短路线，用更短的同起终点路线替换，再拼上另一段，完整费用也会减少。非负权图中的重复绕行可以去掉，因此可以保留各段的最短费用。两段费用相加得到这一类的候选，再与不经过 k 的类别取最小：

$$D_k(i,j) = \min\Big(D_{k-1}(i,j),\ D_{k-1}(i,k) + D_{k-1}(k,j)\Big).$$

还没有允许任何内部顶点时，只能使用直接边，起终点相同时为空路线费用 0。这就是初始距离矩阵。k 是求解过程的阶段，因此放在最外层循环；这一轮完成后，所有点对一起进入允许更多中转点的下一阶段。

先找到 i 到 k、k 到 j 的两段路径，才能把它们接成一条候选路径；所以遇到 INF 时跳过这次拼接。外层固定 k，这一轮所有点对都尝试同一个中转点。对角线为 0，经过 k 再到 k 的空段不会改变费用，于是这一轮可以直接更新当前矩阵。

允许的新中转点	对哪些起点和终点起作用	候选的具体计算
k=1	0→4、2→4	6+3=9；1+3=4
k=2	0→1、0→3、0→4	2+1=3；2+4=6；2+4=6
k=4	多个起点到 5	例如 0→5 为 6+1=7
k=5	多个起点到 6	例如 0→6 为 7+1=8
k=6	4→3、5→3	2+2=4；1+2=3

后两行说明并非始终从 0 出发：同一轮同时更新多个起点的答案。后面继续依次尝试其他中转点，某轮没有更短路线时就保留原距离。

```cpp
#include <algorithm>
#include <limits>
#include <vector>
using namespace std;

void floyd(vector<vector<long long>> &distance) {
    constexpr long long INF = numeric_limits<long long>::max() / 4;
    int n = static_cast<int>(distance.size());
    // k 在最外层：当前轮只新增顶点 k 作为允许的中转点。
    for (int k = 0; k < n; ++k) {
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                if (distance[i][k] < INF && distance[k][j] < INF) {
                    distance[i][j] = min(distance[i][j], distance[i][k] + distance[k][j]);
                }
            }
        }
    }
}
```

样例输入矩阵（函数参数）： INF 使用上面约定的数值哨兵。

```text
0 6 2 9 INF INF INF INF
INF 0 2 INF 3 INF INF INF
INF 1 0 4 7 INF INF INF
INF INF INF 0 2 8 INF INF
INF INF INF INF 0 1 5 INF
INF INF INF INF INF 0 1 INF
INF INF INF 2 INF INF 0 INF
INF INF INF INF INF INF INF 0
```

调用 floyd(distance) 后的矩阵：

```text
0  3 2 6 6 7 8 INF
INF 0 2 6 3 4 5 INF
INF 1 0 4 4 5 6 INF
INF INF INF 0 2 3 4 INF
INF INF INF 4 0 1 2 INF
INF INF INF 3 5 0 1 INF
INF INF INF 2 4 5 0 INF
INF INF INF INF INF INF INF 0
```

怎样读答案？ 第 0 行为 [0,3,2,6,6,7,8,INF]；第 2 行为 [INF,1,0,4,4,5,6,INF]；第 4 行说明 4→3 要经 4→5→6→3，费用 4。第 7 行只有自身为 0。`distance[0][1]` =3，`distance[1][0]` =INF，方向使矩阵不对称，不能把它当无向图读。

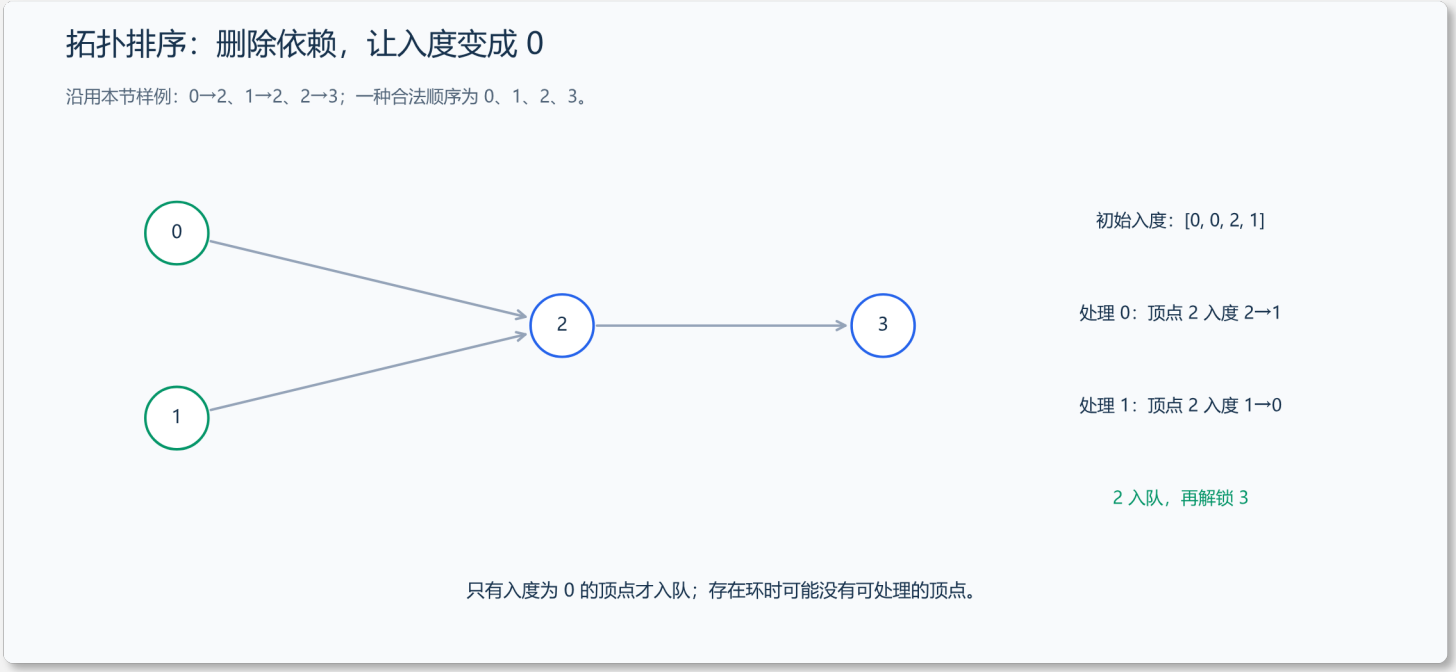
代码与复杂度： 三层循环依次枚举中转点、起点和终点，时间为 $O(n^3)$，距离矩阵占 $O(n^2)$ 空间。例如 n=500 时，核心循环约有 `500³=1.25×10⁸` 次，每次比较候选路线；矩阵有 25 万项。建表时对角线写 0，没有边的位置写 INF，重边取最小权值；零权边写 0，与 INF 区分。随后每一轮沿前面的松弛过程更新矩阵。

练习题： 洛谷 B3647 - Floyd。

7. 拓扑排序

题目原文（自拟示例）： 给定有向图，用边 `u→v` 表示任务 `u` 必须先于任务 `v` 完成。返回一个满足所有依赖的顶点排列；若有环，返回失败。顶点编号为 `0` 到 `n-1`。

思路分析： 没有前置依赖的顶点入度为 `0`，可以先处理。删除它的出边后，新的入度为 `0` 的顶点继续入队。



顶点 2 有来自 0、1 的两条依赖，初始入度为 2。处理其中一个前置任务后，入度变为 1；另一个也处理完后，入度变为 0，才把 2 加入队列。



对边 `0→2、1→2、2→3`，初始入度 `[0,0,2,1]`、队列 `[0,1]`。处理 0 只把 2 的入度减到 1，不能入队；处理 1 才减到 0，把 2 加入；处理 2 后解锁 3。若队列耗尽而输出数少于 n，剩余顶点仍互相依赖，存在有向环；孤立点初始入度为 0，也要加入。

入度记录“还有几个前置依赖没完成”。 入度为 0 的任务现在就可执行，先放进队列。处理顶点 `u` 后，它的每条出边 `u→v` 表示的一项依赖已经满足，令 `indegree[v]` 减一；只有减到 0，才说明所有前置任务都已完成，可以把 `v` 加入队列。

样例初始入度为 `[0,0,2,1]`，队列含 `0、1`。处理 `0` 后，顶点 2 的入度从 2 变成 1，还不能入队；处理 `1` 后降为 0，才入队；然后处理 `2` 解锁 `3`。

若有有向环，环上每个顶点都依赖尚未完成的环上前驱，无法靠删除环外边清掉这些依赖，所以它们不会被全部处理。反过来，如果还有顶点却没有入度 0 的点，在有限的剩余图中不断追溯前驱必会重复顶点，形成环。因此“处理数小于 `n`”既能发现有环，也不会误判 DAG。

多种合法顺序都可以满足依赖。若题目只要任意拓扑序，普通队列即可；要求字典序最小，则每次从全部可执行任务中选择编号最小者，需要小根堆，并重新计入堆操作的复杂度。

有环时怎样停住？ 对 0→1、1→2、2→1，开始只有 0 入度为 0。处理它之后，1 仍依赖 2，2 仍依赖 1，队列为空但两点未处理，返回失败。未输出顶点可能在环内，也可能只是依赖于环，不能把全部剩余点都断言为环内点。

拓扑序把每条依赖边的起点排在终点之前。两个互不依赖的任务可以交换位置，所以同一个图可能有多种拓扑序。顺序中的相邻任务也可能没有边连接；这个排列表示执行次序，路径则表示沿边经过的顶点。

```cpp
#include <queue>
#include <utility>
#include <vector>
```

```cpp
using namespace std;

pair<bool, vector<int>> topologicalOrder(const vector<vector<int>> &graph) {
    int n = static_cast<int>(graph.size());
    vector<int> indegree(n, 0);
    for (const auto &edges : graph) {
        for (int v : edges) {
            ++indegree[v];
        }
    }
    queue<int> pending;
    for (int u = 0; u < n; ++u) {
        if (indegree[u] == 0) {
            pending.push(u);
        }
    }
    vector<int> order;
    while (!pending.empty()) {
        int u = pending.front();
        pending.pop();
        order.push_back(u);
        for (int v : graph[u]) {
            // 当前顶点已经完成，删除它的依赖后，入度为 0 的邻居可处理。
            if (--indegree[v] == 0) {
                pending.push(v);
            }
        }
    }
    // 没处理完所有顶点时，剩余依赖中存在环。
    bool acyclic = static_cast<int>(order.size()) == n;
    return {acyclic, order};
}
```

样例： 依赖为 `0→2`、`1→2`、`2→3`，可以输出 `0 1 2 3`；加入 `3→0` 后存在环。

代码解释： 如果最终处理的顶点数小于 `n`，说明有顶点被环上的依赖阻塞。若要字典序最小的拓扑序，把普通队列换为小根堆。普通版本时间 $O(V+E)$，额外空间 $O(V)$。

练习题： 洛谷 B3644 — 拓扑排序 / 家谱树。

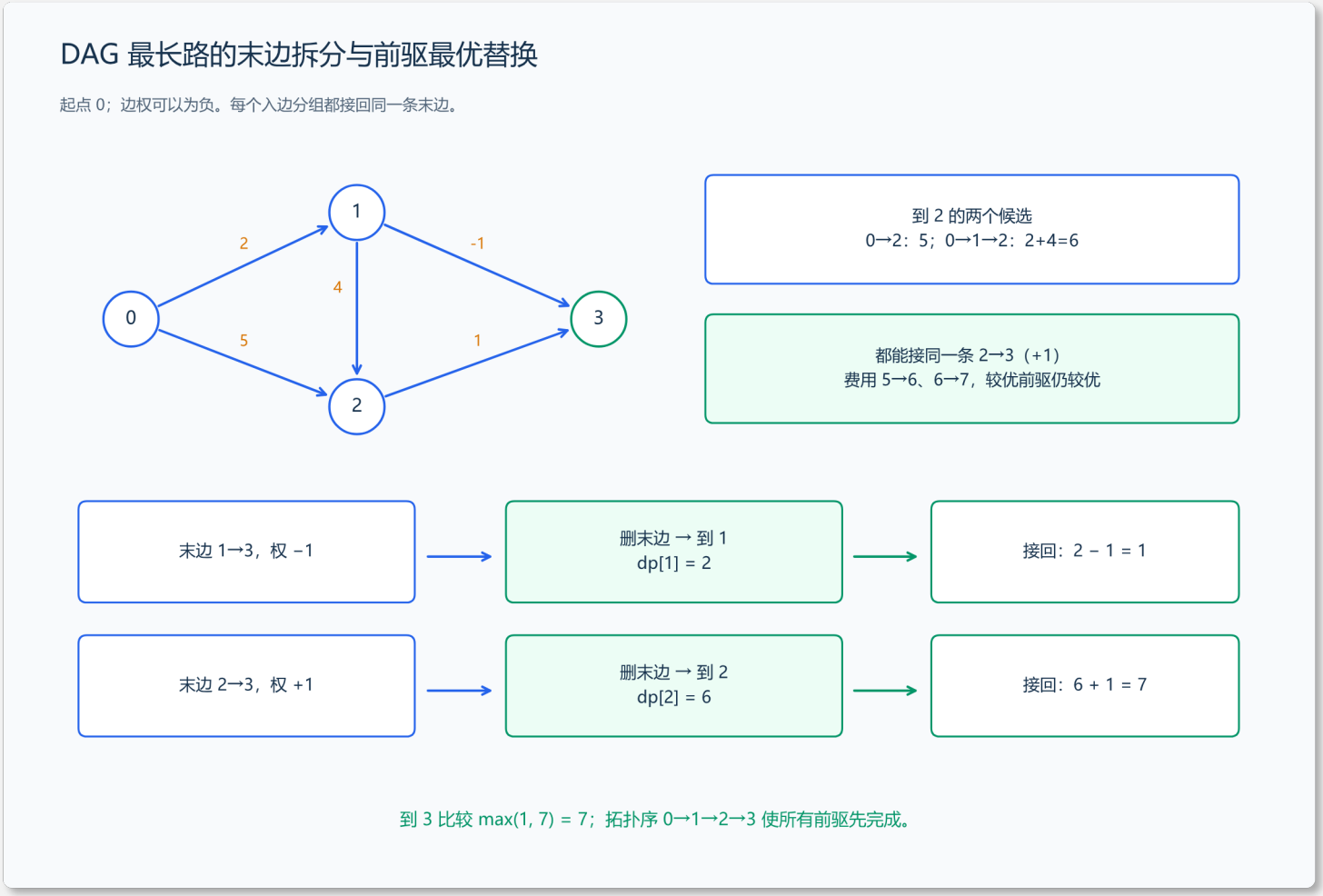
8. DAG 上的 DP：用拓扑序安排状态转移

题目原文（自拟）： 输入有向图邻接表 `graph`，每条记录为 `(终点,整数边权)`，及合法起点 start。`1≤n≤100000`、`0≤m≤200000`，边权绝对值不超过 10^9。若图有环，函数返回 false 和空数组；无环时返回 true 和起点到各点的最长路径费用。不可达用 `numeric_limits<long long>::lowest()/4` 标记，起点空路径费用为 0。

状态与推导： 令 `dp[v]` 表示从固定起点 `start` 到 v 的路径中最大费用。先看起点 0，边为 `0→1:2`、`0→2:5`、`1→2:4`、`1→3:-1`、`2→3:1`。

到 3 有三条路线：`0→1→3` 费用 1，`0→2→3` 费用 6，`0→1→2→3` 费用 7。按最后一条边分组，把这条边拿掉：

最后一条边	删去后允许的前驱路线	前驱的最大费用	接回后的候选
`1→3`，权值 -1	`0→1`	`dp[1]=2`	`2-1=1`
`2→3`，权值 1	`0→2` 或 `0→1→2`	`dp[2]=max(5,6)=6`	`6+1=7`



同一组最后都经过同一条边，它的权值不变，因此最大前驱加这条边，也得到该组的最大费用。换成更好的到 u 路线后仍能接 `u→v`：在 DAG 中，如果前驱路线已经经过 v，再接回来就会产生有向环，所以这里不存在这种冲突。枚举全部入边，覆盖所有非空路径，再比较各组最大费用。

$$dp[v] = \max_{u \text{ 从起点可达}} \underset{u\to v}{\Big(dp[u] + w(u,v)\Big)}.$$

`dp[u]` 是前一段的费用，`w(u,v)` 是接回末边的费用。只有能从起点到达 u，才有实际路径可以接入。

起点不走边的空路径费用为 0，其他点先写 `NEG` 表示还没有路径。例如只有边 `0→1:-2`，到 1 的答案应为 -2；如果先把它写成 0，这个没有路线支持的 0 就会压过真实费用。所以初值是起点 0、其余 `NEG`。

拓扑序保证 v 的所有前驱都在它之前。代码采用“处理 u 时向后继传播候选”的方式：处理 0 得 `[0,2,5,NEG]`；处理 1 时，将到 2 的 5 改成 6，并产生到 3 的候选 1；处理 2 时，再将到 3 的 1 改成 7。轮到 3 时，它的入边候选全部到齐。

```cpp
#include <algorithm>
#include <limits>
#include <queue>
#include <utility>
#include <vector>
using namespace std;

pair<bool, vector<long long>> dagLongest(
    const vector<vector<pair<int, int>>> &graph, int start) {
    const long long NEG = numeric_limits<long long>::lowest() / 4;
    int n = graph.size();
    vector<int> indegree(n, 0);
    for (const auto &edges : graph)
        for (auto [v, w] : edges) ++indegree[v];
    queue<int> q;
    for (int u = 0; u < n; ++u) if (indegree[u] == 0) q.push(u);
    vector<long long> dp(n, NEG);
    dp[start] = 0; // 起点的空路径费用为 0，其余尚不可达。
    int processed = 0;
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        ++processed;
        for (auto [v, w] : graph[u]) {
            if (dp[u] != NEG) dp[v] = max(dp[v], dp[u] + w);
            if (--indegree[v] == 0) q.push(v); // 所有前驱处理完才展开后继。
        }
    }
    if (processed != n) return {false, {}}; // 有环时不提供 DAG 答案。
    return {true, dp};
}
```

手算样例： start=0，边为 `0→1:2`、`0→2:5`、`1→2:4`、`1→3:-1`、`2→3:1`。初值 `[0,NEG,NEG,NEG]`；处理 0 得 `[0,2,5,NEG]`；处理 1 得 `[0,2,6,1]`；处理 2 得 `[0,2,6,7]`。到 3 最长为 `0→1→2→3`，费用 2+4+1=7。

代码与复杂度： indegree 记录还剩多少前置任务，dp 记录当前最长路径费用。处理一个顶点时，先删除它对后继的依赖；如果从起点能到它，再把路径费用传给后继。这样任务顺序与路径计算分别完成，时间 $O(n+m)$，额外空间 $O(n)$，另计图存储。

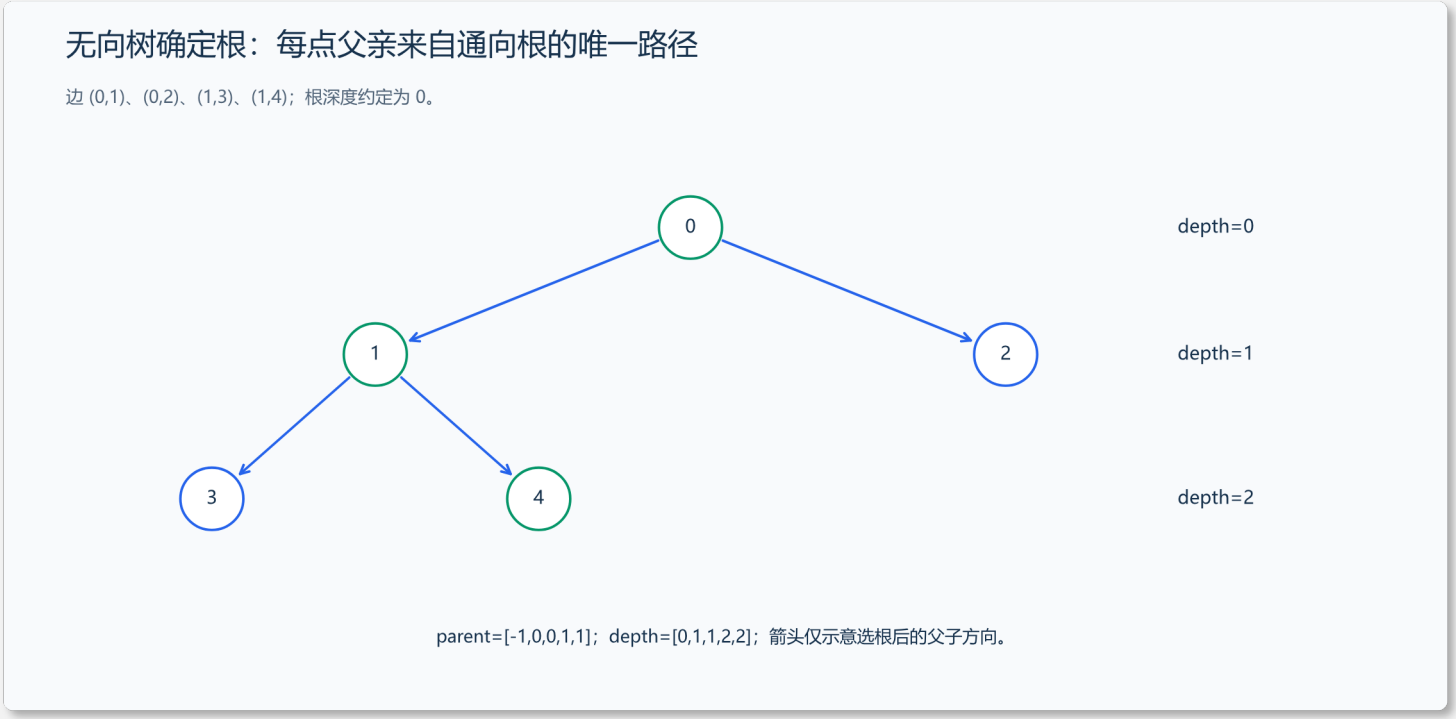
练习题： 洛谷 P1807 — 最长路。

9. 无向树的根、父节点与深度计算

无向树本来没有父子关系。指定根后，从根通向各点的唯一简单路径决定父节点：沿路径靠近根的一端为父，另一端为子。换根会改变父子与深度，却不改变边。二叉树最多有两个有次序的孩子，图论中的树允许任意多个孩子；邻接表记录的是双向邻居。

题目原文（自拟）： 输入有 n≥1 个顶点的无向树，每条边在邻接表 tree 中双向保存，并给定根 root。返回各点的 parent 与 depth；根的 parent 为 -1，深度为 0，其余点的深度是根到该点路径的边数。

分析： 从根 BFS。首次发现 v 时当前 u 就是父节点，深度比 u 多 1。反向记录会指回父节点，已发现判断将其跳过。因为树中路径唯一，这次发现的父关系没有歧义。



根为 0 时，通向 4 的唯一路径是 0→1→4，深度为 2。



根 0 出队时发现 1、2，写 `parent[1]=parent[2]=0`、深度均为 1；节点 1 出队时先跳过已发现的 0，再发现 3、4，写父节点为 1、深度为 2。父指针的写入来自这次首次发现，不是事后看到树形就填答案。无向边会产生反方向记录，访问判断防止重新入队。

```
1  #include <queue>
2  #include <utility>
```

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

pair<vector<int>, vector<int>> rootTree(const vector<vector<int>> &tree, int root) {
    vector<int> parent(tree.size(), -1), depth(tree.size(), -1);
    queue<int> q;
    depth[root] = 0;
    q.push(root);
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        for (int v : tree[u]) {
            if (depth[v] != -1) continue; // 跳过已发现节点，避免沿反向边重复入队。
            parent[v] = u;
            depth[v] = depth[u] + 1; // 根深度为 0，按边数计深度。
            q.push(v);
        }
    }
    return {parent, depth};
}
```

样例： 边 `(0,1)、(0,2)、(1,3)、(1,4)`，root=0 时 parent=`[-1,0,0,1,1]`、depth=`[0,1,1,2,2]`。root=3 时 3→1→0 的父子方向改变，0 的深度变为 2。

复杂度： 树有 n-1 条边，每个节点与连接处理一次，时间和额外空间为 $O(n)$。对一般图搜索时，首次发现的连接形成一棵搜索树；其他连接仍保存在原图中。

练习题： 洛谷 P3884 - 二叉树问题（按题面调整深度与路径代价）、洛谷 P4913 - 二叉树深度。

10. 最小生成树：Kruskal 按权值选边

生成树从连通无向图中选边，保留全部顶点并使其连通且无环；最小生成树使所选边权和最小。它优化整张网络的总建造费用，和某起点到各点的最短距离是不同目标。

题目原文（自拟）： 第一行输入 n m，`1≤n≤100000、0≤m≤200000`。随后 m 行输入 u v w，表示无向边，编号为 `0…n-1`，权值绝对值不超过 10^9，允许重边、自环。选择一些边连接全部顶点，输出最小总边权；无法连接全部顶点时输出 `IMPOSSIBLE`。只有一个顶点时空边集费用为 0。

算法： 初始每个点独自成块。边按权值升序，连接不同块就选并合并，连接同块就跳过，因为会形成环。选到 n-1 条时完成生成树；全部边检查后仍不足则不连通。



小权边先连接不同块，已连通两端之间的边会成环而跳过。

为什么能选当前最小的跨块边？ 固定已选边形成的某块 C，当前边 e 跨出 C，且在这个切分的边中权值最小。考虑一棵包含此前选择的最优生成树：若有 e 无需改变；若没有，加入 e 会成环，环上必有另一条跨出 C 的边 f。f 不是此前已选的块内边，权值不小于 e。删 f 后仍是生成树，费用不增，并保留此前选择与 e。因此存在一个最优解能接受本次选择，逐步交换即可证明正确。

10.1 排序与并查集选边

```cpp
#include <algorithm>
#include <functional>
#include <numeric>
#include <tuple>
#include <utility>
#include <vector>
using namespace std;

pair<bool, long long> kruskal(int n, vector<tuple<long long, int, int>> edges) {
    sort(edges.begin(), edges.end()); // 元组第一项是权值，升序检查。
    vector<int> parent(n), size(n, 1);
    iota(parent.begin(), parent.end(), 0);
    function<int(int)> find = [&](int x) -> int {
        if (parent[x] != x) parent[x] = find(parent[x]); // 路径压缩。
        return parent[x];
    };
    long long total = 0;
    int used = 0;
    for (auto [weight, u, v] : edges) {
        int a = find(u), b = find(v);
        if (a == b) continue; // 已连通，选这条边会成环。
        if (size[a] < size[b]) swap(a, b);
        parent[b] = a; // 只合并根，小树接到大树。
        size[a] += size[b];
        total += weight;
        if (++used == n - 1) break;
```

```cpp
    }
    return {used == n - 1, total}; // 失败时 total 仅是森林费用。
}
```

并查集含义与前一章相同，函数内部直接保存父指针和集合大小。edges 按值传入，排序不改变调用者的边表。

10.2 读入与输出

下面片段与上面的头文件和函数依次放在同一 `.cpp` 中。

```cpp
#include <iostream>
// main 之前放入上面的头文件和 kruskal 实现。
int main() {
    int n, m;
    cin >> n >> m;
    vector<tuple<long long, int, int>> edges;
    while (m--) {
        int u, v;
        long long w;
        cin >> u >> v >> w;
        edges.push_back({w, u, v}); // 无向边在边表中只保存一次。
    }
    auto [connected, total] = kruskal(n, edges);
    if (connected) cout << total << '\n';
    else cout << "IMPOSSIBLE\n";
}
```

输入：

```text
4 5
0 1 1
1 2 2
0 2 3
2 3 4
0 3 10
```

输出：

```text
7
```

样例推导： 选 (0,1,1) 得块 `{0,1}`；选 (1,2,2) 得 `{0,1,2}`；(0,2,3) 两端已连通，跳过；选 (2,3,4) 后全部连通，费用 1+2+4=7。平权边可能有多种最优选择，负权边仍可按相同规则处理，自环总被跳过。

复杂度： 排序 $O(m \log m)$，初始化 $O(n)$，所有边查询合并共 $O(m\alpha(n))$，总时间 $O(n + m \log m)$，空间 $O(n + m)$，包括排序的边表副本。失败时不能把森林费用输出为连接全图的答案。

练习题： 洛谷 P3366 — 最小生成树（顶点按 1 编号，不连通输出 `orz`）。

11. 图算法的选择与题目对应

先明确点、边、方向、权值、是否有环以及答案含义，再对照本章完整例题。规模写为 n、m，不能只看顶点数而忽略稠密图的边数。

问题模型	对应方法	核心思路
无向图可达、连通块	DFS、BFS、并查集	覆盖可达集合，或维护集合合并
每步代价相同的最少步数	BFS	较短层先处理
非负边权、单起点最短路	Dijkstra	最小有效候选不会被未处理路径改善
点很少、任意两点最短路	Floyd	逐轮允许中转点，时间 $O(n^3)$
任务依赖排列、有无有向环	拓扑排序	依次删除无前置依赖的顶点
把相邻对象分到不同组	二分图染色	沿边交替颜色并检查矛盾
DAG 最长路径	拓扑序 DP	全部前驱先计算
连接全图的最小总费用	最小生成树	连接不同块，交换论证保证最优

提交检查： 无向边是否双向存储？编号是否统一转换？不可达是否与 0 费用区分？多组图与数组是否重建？自环、重边、不连通如何处理？递归深度是否可能等于顶点数？沿具体输入运行一遍，观察每个判断怎样改变图、队列和距离。

后记

从读懂题目到独立完成一道题

学到这里，C++ 语法、容器和算法已经能连成一条解题过程：把题目中的对象和规则写清楚，用具体数据手算，提出模型，证明计算方法，再把它写成程序。程序得到答案后，还要核对输出、工作量和每一步留下的状态。

回看各章，可以把这条过程记成几件具体的事：

1. **先明确要计算什么。** “存在一组”“统计所有组”“求最优值”“输出方案”需要的信息不同。相同取值、不同位置、连续片段这些词，也会改变合法方案。
2. **从能解释的做法出发。** 先枚举小数据，再找重复计算、可以排除的候选或可以共享的状态。优化前后的程序应该回答同一个问题。
3. **给每个状态写一句含义。** 频次只包含过去的位置，BFS 的距离来自首次发现，DP 的一个格子对应某个子问题。状态含义清楚，更新次序就有了依据。
4. **把公式接回实际方案。** 状态转移方程中的每一项，都要能说出从哪里来、做了什么动作、为什么可以合并；数值算对之后，还可以沿记录恢复具体方案。

5. **用差异定位错误。** WA 时找第一处与手算不同的状态，缩小输入，再对照题面的句子和程序的更新次序。修正后留下这个输入，供下一次检查。

理解一种方法，可以用一个小任务检验：合上代码，给别人讲清变量的含义，拿一组数据走完过程，然后自己写出实现。卡在某一步，就回到那一步对应的图、推导和例题。

留给继续练习的你

第一次看懂题解、第一次独立写出 DFS、第一次自己推导出 DP，花的时间可能都比预想的长。做不出来时，先把已经确定的东西写下来：哪些方案合法，哪一步重复，哪条规则还没理解。一个能区分两种想法的小例子，往往比继续盯着整道题更有用。

一道题通过后，留几行复盘就够了：最开始怎么想，在哪个地方改了模型，哪个反例帮助了你，下次遇到相似问题能先试什么。隔几天再独立写一次，看看留下的是思路还是代码印象。

希望你在练习中慢慢形成自己的判断：面对一段题面，知道从哪里开始；面对几个做法，知道为什么选其中一个；面对一份错误程序，知道怎样把问题缩小。比赛中的进步，常常就来自这些可以重复做到的小事。

按学习阶段选择推荐书籍

书可以按当前遇到的困难来选：基础题建模还不熟，就多看入门例题；基础题能独立完成了，就按专题进阶；想把证明补得更严密，就对照另一种讲法阅读。下面的推荐按这些用途安排，不必同时从头读完几本书。

推荐书籍	适合的阶段与推荐理由	阅读方式与检索链接
刘汝佳《算法竞赛入门经典（第 2 版）》	刚学会语法、正在练习独立解题。程序设计、STL、枚举与算法例题相连，适合练习把题目写成程序	与本笔记并行读相应专题，先做例题再做习题。出版社与配套资源；Z-Library 检索，ISBN: 9787302356288
罗勇军、郭卫斌《算法竞赛》（上册、下册）	学过本笔记的基础算法，想系统扩展专题并查阅实现。基础结构、搜索、进阶结构、DP、数学、字符串和图论有连续的讲解与例题，适合按当前任务选读	先对照已学章节复习，再按后续路线补树状数组、线段树等专题；每读一种方法就独立完成相应例题。清华大学出版社书目与配套资源；Z-Library 检索，ISBN: 9787302615217，可对照上、下册书名查找
李煜东《算法竞赛进阶指南》	能独立完成基础题，准备扩展数据结构、搜索、数学、DP 与图论。适合沿一个专题连续训练，把零散方法连起来	选一个当前需要的专题，边读边实现并核对例题；作者资源社区提供配套资料与勘误。Z-Library 检索，检索词：算法竞赛进阶指南、李煜东
Antti Laaksonen, Competitive Programmer's Handbook	已理解基础算法，想用英文重新梳理方法、熟悉英文题面术语。推荐把它作为第二种讲法，对照自己的推导	每次读一个已学专题，自己复述关键关系，再做相近题。作者在 CSES 图书页提供免费 PDF；Z-Library 检索，检索词：书名、Antti Laaksonen
Steven Halim、Felix Halim、Suhendry Effendy, Competitive Programming 4	已经开始综合题和比赛训练，想按专题补足知识并选择更多练习。适合在复盘后针对薄弱处阅读	按训练中缺的方法选章节，用题目检验，而不是只收藏模板。作者配套网站提供书籍信息、勘误与训练资料；Z-Library 检索，检索词：书名、Steven Halim

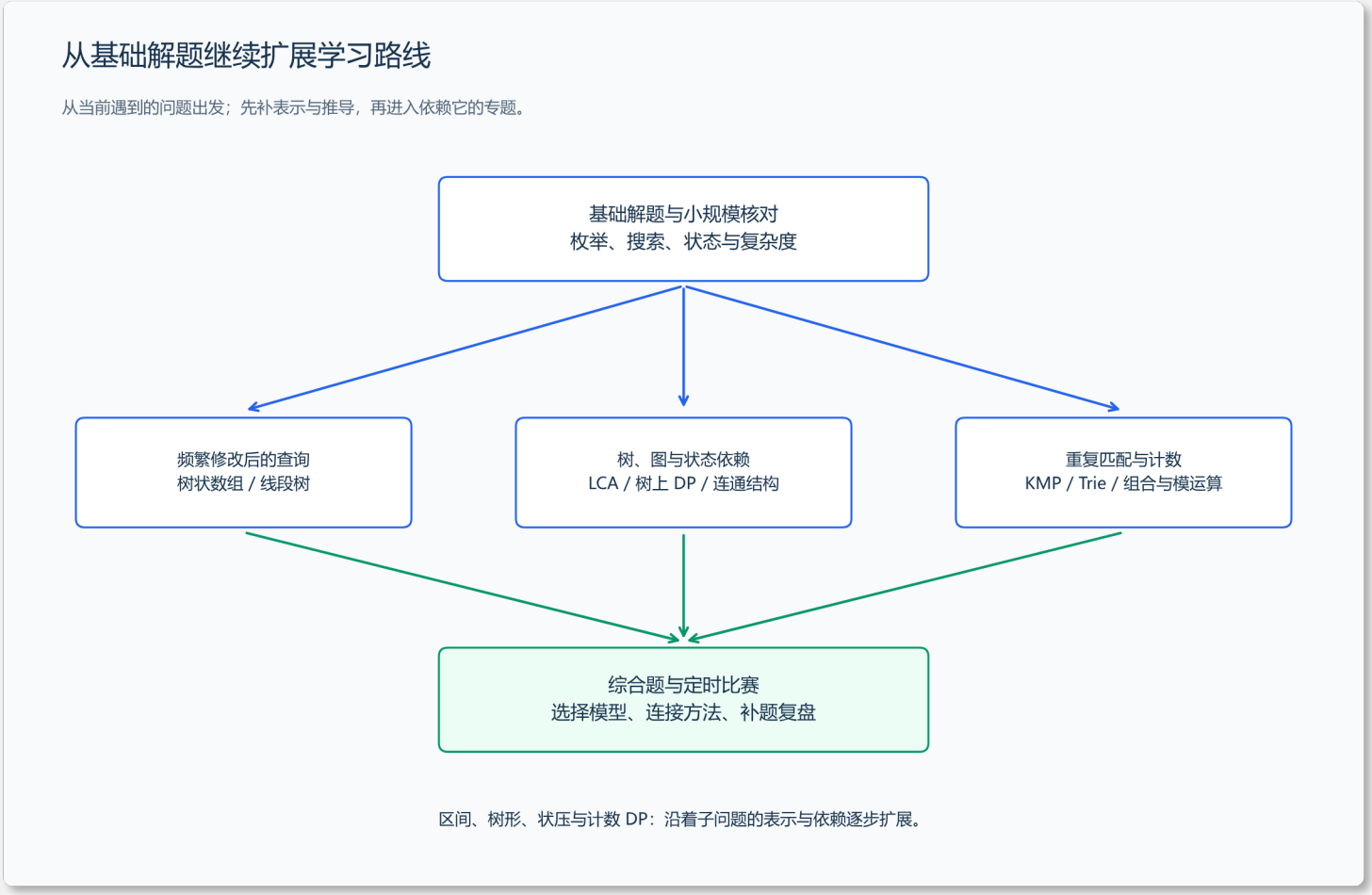
开始时可先选《算法竞赛入门经典》，或用免费的 Competitive Programmer's Handbook 配合本笔记练习；有一个专题需要继续深入时，可以从《算法竞赛》上下册、《算法竞赛进阶指南》或 Competitive Programming 4 中选一套，围绕当前专题阅读和做题。检索时对照作者、版次和 ISBN，中文书名相近的版本尤其要看清。

后续学习路线

后续不必按"先把所有算法名称学完，再开始做题"的方式推进。每次选一个当前能理解的任务，学会表示、推导和实现，再进入依赖它的下一步。

阶段	可以继续学习的内容	从已有知识接过去的理由	一次练习的完成目标
巩固基础解题	枚举、模拟、前缀和、双指针、二分、基础贪心的综合题	继续练习把文字规则变成循环、判断和单调性	独立建模，比较至少两种做法，算出复杂度
扩展查询结构	树状数组、线段树与区间修改	前缀和能快速查询；频繁修改后，旧前缀和需要重算，从这个问题引出维护局部信息的结构	画出一次修改影响的位置和一次查询合并的片段
扩展树与图	树上 DFS、树上 DP、倍增与 LCA，再学习强连通分量、桥与割点	已有根、父节点和深度；进一步处理两点关系、子树信息和图中的连通结构	先说明节点、边和答案，再画出状态或搜索记录
扩展 DP	区间 DP、树形 DP、状压 DP、计数 DP	从线性顺序走向区间、树和集合，继续使用"去掉一个决策，得到剩余子问题"的推导	展开一个具体状态的候选，推导方程、顺序和答案位置
扩展字符串与数学	Trie、KMP；质数筛、组合计数、模运算与扩展欧几里得	复用字符串比较、前缀信息、整除和递推，处理重复匹配与更大的计数	手算核心状态的变化，说明重用的信息来自哪里
进入综合训练	定时比赛、补题；按需要继续学习网络流、匹配等专题	题目不再直接告诉你应使用哪个算法，要把多个模型连接起来	比赛后补出模型、证明和反例，再独立实现一次

这些专题可以穿插学习。例如线段树与树上问题可以互相促进，字符串和数学也可以按遇到的题目提前阅读。路线中的箭头表示从已有知识继续推进，不是按书名或题号机械排队。



需要查某个专题的概念、例题和实现时，可以阅读 OI Wiki；需要一组按专题排列的练习时，可以使用 CSES Problem Set。洛谷、HDU、Codeforces 与牛客的题目也可以继续做：刚学一个专题时用分类题巩固，之后用混合题和定时比赛练习自己识别模型。

牛客周赛与 Codeforces 分级训练

平时可以用 CF 题目逐步提高难度，再用牛客周赛练习在规定时间内自己识别模型、安排做题顺序。比赛后的补题把这两件事接起来：找出当场卡住的步骤，回到相近难度的题目继续练习。

用牛客周赛练习定时解题

推荐参加牛客周赛。从牛客竞赛入口筛选“牛客周赛”，查看比赛页面并报名；往期比赛可以回顾题目，或用“创建重现赛”重新安排一次定时练习。

一场比赛可以这样做：

1. **先浏览题目。** 用几分钟看输入规模、输出目标和样例，把已经有思路的题先做出来，再回到需要推导的题。
2. **给当前想法留下记录。** 题意能理解但暂时做不出时，写下已有模型、尝试过的反例和还缺的推导，再决定继续尝试还是先看另一题。
3. **检查后再提交。** 手算一组数据，估算工作量，核对多组输入、类型和输出格式。WA 后先定位与手算不同的状态，再修改。

4. **比赛结束后补一题。** 优先选当场最接近做出来的一题。先继续独立推导；阅读题解时，找到自己缺的那一步，再合上题解重新实现。

例如一场比赛完成两题，第三题已经想到“统计过去的频次”，却因为先写入当前项而 WA，赛后就先把这个更新次序补清楚。记录失败输入，再选一道同样要区分过去与当前的计数题，比只记“这场做了两题”更能指导下一次训练。

按 CF 题目 rating 逐步提高难度

选题时看**题目难度 rating**，例如 800、1000、1200；比赛榜单上的 rank 记录当场比赛的名次。题目难度与选手的参赛评分分别记录，选题时直接使用题目旁的难度数值。字段含义可以查阅 Codeforces 官方说明。

打开 CF 题库按难度升序排列，在“Filter Problems”的“Difficulty”中填写当前练习的难度范围。刚开始可以从 800 分题试起，先判断自己卡在读题、推导还是实现，再逐级增加 100 分或 200 分。

当前练习的题目 rating	可以安排的训练任务	每题完成后检查的能力
800–1000	练习英文题面、条件判断、模拟、枚举和基础数学	能把文字规则逐项转成代码，并独立处理完整输入输出
1100–1300	比较直接枚举与排序、前缀信息、频次统计等做法	能找到重复工作，说明优化前后计算的是同一份答案
1400–1600	练习多步骤推导，结合二分、贪心、搜索、基础 DP 与图模型	能解释关键判断、证明或状态转移，并估算最大规模的工作量
1700–1900	根据补题中遇到的问题扩展专题，练习多个方法的组合	能从题面自己识别所需信息，把几个计算步骤连接成完整解法

选题时可以先做一个具体尝试：找 10 道 800 分题，记录哪些题完全独立完成、哪些题看过提示、哪些题看过完整题解。例如 7 道能独立完成，另外 3 道在复盘后也能讲清推导，就选几道 900 分题继续尝试；若只有 2 道能独立完成，则先把反复出现的问题补清楚，再换一组同难度题。

提高难度时保留一些已经熟练的题练习速度，再安排略高一档的题练习推导。遇到某一题长时间想不出，先整理已有结论，再分段阅读题解：先看模型，自己尝试；还缺推导再继续看；最后合上题解完成代码，隔几天独立重做。

做专题练习时，可以用标签找到刚学过的方法；做综合训练时，先读题自行判断方法，再核对标签。这样既能巩固某种算法，也能练习比赛中最重要的模型识别。

把周赛、分级练习和复盘接成一轮

可以按下面的顺序安排一轮练习：

环节	具体动作	留下的记录
赛前练习	做几道当前 rating 附近的 CF 题，再尝试少量高一档的题	独立完成与使用提示的情况；读题、推导、实现分别耗时
牛客周赛	在比赛时间内浏览、选择、推导、实现和检查	当场想到的模型、WA 输入、暂时没推完的步骤
赛后补题	补最接近完成的一题，复现模型与代码	缺的是哪条题意、哪段证明或哪个更新次序
返回 CF 练习	选相近难度、能练到这个薄弱步骤的题目	能否在新的题面中独立识别同一思想

例如周赛卡在“把最优化转成可行性判断”，接下来就从当前 rating 附近找二分相关题，练习判定函数和单调性的推导。把这一步练熟，再回到混合题与下一场周赛，观察自己能否独立识别这个模型。

复盘时把 rating、独立解题情况和具体困难放在一起看。比赛名次记录当场表现，练习记录则告诉我们下一步应补什么；难度的提高也随这些能力的变化逐步推进。

国内 ACM 与 OI 类竞赛

周赛和分级练习积累起来后，可以给自己选一个比赛目标：在校赛中独立完成几题，与队友合做一场题，或用一套信息学试题练习完整推导和自测。国内常见的程序设计比赛，有高校团队赛，也有面向青少年的信息学竞赛；它们使用的算法有很多相通之处，作答和计分方式则会影响比赛中的时间安排。

高校团队赛：ICPC 与 CCPC

在高校里，大家常说的“打 ACM”，通常指参加 ICPC、CCPC 以及采用相近赛制的校赛、省赛和邀请赛。**ICPC** 是国际大学生程序设计竞赛，**CCPC** 是中国大学生程序设计竞赛。国内高校既参加 ICPC 的国内赛站，也参加 CCPC 的网络赛、分站赛和总决赛。

比赛	可以了解的内容	官方入口
ICPC 国内赛站	高校组队参加的区域赛，以及国内举办的相关邀请赛；通过赛站页面查看比赛介绍、题目和榜单	ICPC 官网；区域赛与赛站
CCPC	国内高校的团队程序设计竞赛，包含网络选拔、分站赛和总决赛，也有省赛、地区赛与邀请赛	CCPC 简介；赛事公告
校赛	从自己学校的比赛开始，练习定时作答、组队和现场配合	查看学校竞赛队、计算机学院与教务处的通知
各省大学生程序设计竞赛	与省内高校队伍一起定时解题，检验基础实现、综合建模和团队配合；下一节给出各地赛事与备赛方法	从本省竞赛组织与承办学校的赛事页面进入，也可在 CCPC 赛事页查找相关活动

经典的 ICPC、CCPC 现场赛由 **3 名队员共用一台计算机，在 5 小时内合作解题**。有人读新题、有人在纸上推导、有人写代码，机器成为需要共同安排的资源。提交后会收到评测结果，队伍根据通过情况和当前进展调整做题顺序。ICPC 比赛规则、CCPC 现场赛规则示例中可以看到这些安排。

例如三人同时发现一道容易题，全部围着电脑讨论，另外两道题就没有推进。可以由一人实现容易题，第二人检查样例和构造反例，第三人读下一题；提交后再交换角色。分工围绕当前任务变化，每个人都要能向队友讲清模型、复杂度和还没证明的步骤。

开始准备时，可以先参加校赛，了解本校竞赛队的训练与选拔安排；找到队友后，选一场往年的团队赛共同作答。赛后记录读题、推导、写代码和调试各用了多久，以及电脑空闲或多人等待的时段。下一轮训练就能同时改进解题能力和配合方式。

各省大学生程序设计竞赛

学会基础算法、开始做定时训练后，可以把**本省大学生程序设计竞赛**作为一个近期目标。省赛把省内高校的队伍聚在一起，适合检验基础题能否稳定通过、综合题能否独立建模，以及三人合作时怎样分配读题、推导和电脑使用。平时参加校赛、加入学校竞赛队，再准备省赛，练习就有了具体的时间安排。

省级赛事会使用“大学生程序设计竞赛”“大学生程序设计大赛”“ICPC 某省赛”“CCPC 某省赛”等名称。可以从下面这些实际赛事了解它们的组织方式，并找到自己所在地区的入口：

地区	省级赛事举例	官方或组织学校的入口
浙江	浙江省大学生程序设计竞赛	竞赛报名平台；2026 年承办学校的赛事介绍
江苏	江苏省大学生程序设计大赛	江苏省计算机学会的赛事介绍
山东	山东省大学生程序设计竞赛	山东大学软件学院的赛事介绍
广东	广东省大学生程序设计竞赛（GDCPC）	承办学校的赛事介绍；校赛衔接省赛的选拔通知
河南	ICPC 河南省大学生程序设计竞赛、CCPC 河南省大学生程序设计竞赛	ICPC 省赛承办学校的介绍；CCPC 省赛的高校参赛记录
四川	四川省大学生程序设计大赛	承办学校的赛事介绍；校内选拔与备赛通知
湖南	湖南省大学生计算机程序设计竞赛的 ACM 程序设计类别	高校选拔与训练安排
陕西	ICPC 陕西省大学生程序设计竞赛	承办学校的赛事介绍
北京	北京市大学生程序设计竞赛	承办学校的赛事介绍

阅读赛事名称时，可以结合一个具体例子理解：河南在 2026 年分别举办了 ICPC 省赛和 CCPC 省赛，两场比赛各自有承办学校、赛题和榜单。想参加其中一场，就到相应赛事页面了解活动，再通过本校竞赛队安排报名和训练。

各省的参赛消息通常可以沿三处查找：**本校教务处或竞赛队 → 本省计算机学会、大学生科技竞赛组织 → 承办学校的赛事页面**。搜索时写“省名 + 大学生程序设计竞赛 + 年份”，再核对页面中的主办、承办单位与赛事名称；找到本省往年题后，可在牛客、HDU 等平台检索赛事全名，安排重现赛和补题。

备战省赛可以分成三步：

1. **先完成一场往年题。** 由队友一起定时作答，记录每题由谁读、谁推导、谁实现，以及第一次通过的时刻。
2. **按卡点补题。** 基础题写错，就回到输入输出、类型与模拟；综合题想不出，就整理题意、直接做法和重复计算；已有算法却没通过，就缩小 WA 输入，找第一处状态差异。
3. **再做一场检验。** 观察同类失误是否减少，机器等待是否缩短，以及是否能独立解释更多题的模型。随后把省赛中的薄弱专题带到 ICPC、CCPC 网络赛与分站赛训练中。

这里的省赛是高校程序设计比赛。后面信息学活动中的**省选**，则是选拔参加 NOI 的省代表队；两者各自接到高校团队训练与青少年信息学训练中。

青少年信息学：CSP-J/S、NOIP 与 NOI

OI 是 Olympiad in Informatics，即信息学奥林匹克。国内的 NOI 系列活动由中国计算机学会（CCF）组织。学习中会经常见到 CSP-J/S、NOIP、省选和 NOI 这些名称，先把它们各自对应的活动认清，再选择适合当前学习阶段的题目。

活动	名称与主要形式	适合接入的练习
CSP-J	CCF 非专业级软件能力认证的入门组（Junior）；第一轮侧重基础知识、读程序与推理，第二轮上机编程	配合语言基础、模拟、枚举和基础算法，练习读程序、手算状态与完整实现
CSP-S	同一认证的提高组（Senior），算法设计与编程任务进一步深入	在基础题能独立完成后，练搜索、数据结构、DP、图论与多步骤建模
NOIP	全国青少年信息学奥林匹克联赛，由各省组织的全国性个人程序设计竞赛	用往年整套题练习读完题面、分析不同规模、比较做法与分配时间
省级选拔活动	各省选拔参加 NOI 的代表队；相关方案与活动通知在各省栏目发布	继续补进阶专题，通过整套题检查多个方法的组合与实现能力
NOI	全国青少年信息学奥林匹克竞赛，各省派代表队参加，考查更深入的算法设计与程序实现	阅读较复杂的模型与证明，练习把长推导拆成可检验的步骤

其中，**CSP-J/S 是算法与编程能力认证**，两个组分别称为入门组、提高组，认证结果使用一级、二级、三级能力等级。可以通过 CCF 的认证说明 与等级名称公告 了解它的形式和结果名称。

NOIP、NOI 与省级选拔活动的介绍、报名消息和历年资料集中在 NOI 官网；系列活动简介 说明各活动的关系，各省新闻 提供本省的具体安排。准备参加活动时，可以和学校的信息学指导教师一起查看报名通知、训练计划与比赛环境，把日常练习接到实际活动上。

学习上可以先用 CSP-J 往年题检验基础，再用 CSP-S、NOIP 的题目补足建模和算法；已经能完成较长推导后，继续尝试省选与 NOI 题。每个阶段都结合自己的独立解题情况选题，遇到还没学过的专题，就回到前面的后续学习路线补相应知识。

ACM 式与 OI 式计分带来的训练重点

同样一道题，比赛评价答案的方式不同，选手就会采取不同的时间安排。

ACM 式计分先比较通过题数，再比较总罚时。 一道题得到 AC 才增加通过数；该题的罚时由通过时刻与此前计罚的失败提交组成，各已通过题的罚时再相加。下面用每次计罚失败提交加 20 分钟的 ACM 式规则，计算一份自拟记录：

队伍	已通过题数	已通过题的记录	总罚时
甲队	2	A 在第 30 分钟通过，之前有 1 次 WA；B 在第 80 分钟一次通过	$(30+20)+80=130$ 分钟
乙队	2	A 在第 40 分钟通过；B 在第 85 分钟通过，两题都一次通过	$40+85=125$ 分钟
丙队	3	三题分别在第 60、120、180 分钟一次通过	$60+120+180=360$ 分钟

丙队通过 3 题，排在两支只通过 2 题的队伍之前。甲、乙通过数相同，再比较罚时，乙排在甲之前。甲队即使在另一题上提交过失败程序，这题尚未通过时也不加入上述总罚时。这种计分方式可以在 ICPC 官方赛区的说明 中看到。

由这个例子可以得到训练重点：优先推进有把握完整通过的题；提交前检查推导、实现和样例；当前题卡住时，把已知结论交给队友，再判断继续推导还是切换题目。平时既要练独立解题，也要练习把思路讲给队友听。

OI 式题目常把分数分配给测试点或子任务。 题面可能给小规模数据、特殊结构与完整规模分别安排分数，选手需要分析每部分可以使用什么做法。NOIP 的官方题目分析 展示了从朴素算法到优化算法对应不同得分的情况。

例如一份自拟评分安排：第一组数据给 30 分，枚举即可完成；第二组给 30 分，需要复用前缀信息；第三组给 40 分，还要进一步优化查询。程序只完成第一组得到 30 分，完成前两组得到 60 分，三组全部完成得到 100 分。另一题尚未推导完整时，已经验证的做法也有对应的得分；把它实现并保存后，可以继续研究更大规模的计算。

这就接回了前面的“一题多建模”：先提出直接做法，代入规模计算工作量，再分析怎样减少重复计算。定时训练时，写清每个做法解决了哪部分数据、还缺哪一步推导；比赛后继续把已完成的部分推进到完整解法。分析子任务时仍要检验算法是否正确，不能只凭样例通过就判断能得分。

蓝桥杯与天梯赛等相关比赛

除了上述两条训练方向，也可以用其他国内程序设计比赛安排练习：

比赛	比赛中的主要任务	可以重点练习的能力	官方入口
蓝桥杯软件赛 C/C++ 程序设计	个人完成程序设计题，软件赛设有省赛等赛事安排	基础实现、模拟、枚举、搜索、数学与综合建模，练习一场题中的时间分配	蓝桥杯个人赛介绍
团体程序设计天梯赛	题目分基础、进阶、登顶三个梯级；队员独立作答，再形成团队成绩	从基础输入输出和模拟逐步进入数据结构与算法，练习稳定实现和整场安排	天梯赛官网与比赛规则

例如想检查 C++ 基础是否熟练，可以先定时做一组天梯赛基础题，记录失误出在输入、类型、循环还是题意转换；想练个人整场安排，可以用蓝桥杯往年题作答，再按“读题–建模–实现–检查”复盘每道题。

企业组织的算法与编程赛事

还可以把企业组织的赛事作为练习目标。有的比赛要求在规定时间内完成多道算法题，有的比赛围绕一个业务任务，让参赛者持续改进方案。选择时先读“要交什么、怎样评测、分数怎样计算”，再判断应当练定时解题，还是练一个方案从实现到优化的完整过程。

赛事或平台	企业与主要任务	可以接入的训练	赛事入口
百度之星程序设计大赛	百度举办的个人程序设计比赛，通过在线评测比较解题数量与用时	接在牛客周赛和 CF 分级练习之后，练个人读题、建模、完整实现与整场安排	百度之星官网；2026 年高校参赛通知
华为软件精英挑战赛	华为举办的软件编程赛事，围绕实际业务场景设计、实现和优化算法方案	把题目拆成数据表示、合法方案、评分目标与改进步骤，练团队实现、实验比较与性能优化	华为大赛入口；2026 年官方赛事介绍
阿里云天池上的企业赛事	阿里云及合作企业举办或共同发起的赛事，平台设数据算法、工程开发等类别	从赛题的输入、提交结果与评价指标出发，练数据处理、可复现的实现和多轮改进；向数据算法方向学习时继续补统计与机器学习	天池赛事列表

限时算法题可以沿用本笔记的解题流程。例如准备百度之星，就先用当前 rating 附近的 CF 题补基础，再独立完成一场往年题；复盘时区分题意没读清、模型没选对、推导没完成与代码写错，下一轮练习围绕具体问题安排。

业务优化题则需要给“改进”留下可比较的记录。先把一份完整的基础方案跑通，核对输出是否合法，再记录它的得分、耗时与主要开销。之后一次改一个步骤，使用同一组数据比较前后的结果。例如做调度任务时，先明确对象、资源和任务顺序，构造一份可执行方案；再尝试调整任务次序或减少重复计算，观察完成时间或费用怎样变化。

推进步骤	具体动作	留下的结果
读懂任务	写清对象、动作、合法方案、输出格式与评分目标	能手算一个完整小例子
实现基础方案	把输入、计算、输出和评测过程接起来	一份可运行、可验证的程序与初始结果
找到主要开销	记录时间消耗、重复工作或方案费用	确定这轮要改进的一个步骤
修改并比较	用同一组数据运行两份方案，核对结果与耗时	知道变化来自哪项修改

推进步骤	具体动作	留下的结果
整理并复现	保存程序、数据、运行方式和结果记录	队友能复现结果，下一轮能继续推进

参加这类比赛时，前面的方法论仍然有用：先验证模型，再算工作量；先解释状态，再更新状态；发现结果变差时，用具体数据找差异。数据算法与工程开发的比赛还会把这些方法接到更长的程序、更多的实验和团队文档中。

赛事页面也会写清企业的角色。例如江苏省程序设计大赛由省计算机学会主办、高校承办，华为等企业提供赞助；百度之星由百度举办。查找比赛时读清主办、承办、合作和赞助单位，便能从对应入口了解比赛的组织与任务。

从日常练习接到参赛训练

可以把牛客周赛、CF 分级题和国内比赛题连接成一轮训练：

1. **选择一个近期任务。** 例如参加校赛、与队友完成一场团队训练，或独立完成一套 CSP-J 往年题。写清要练的是基础实现、综合建模还是团队配合。
2. **用分级题补薄弱步骤。** 平时围绕当前 CF rating 做题，遇到专题困难就回到本笔记和推荐书的对应章节，用手算与实现补齐。
3. **用整场题检验。** 牛客周赛练个人定时判断；团队方向再做省赛、ICPC、CCPC 往年赛题，个人算法方向可做百度之星往年题，信息学方向再做对应阶段的整套题；业务优化方向则完成一轮基础方案与改进方案的实验。按目标比赛的作答方式训练，记录实际时间与中间进展。
4. **用复盘决定下一轮。** 保留没推完的状态、错误模型和 WA 输入。先补最接近完成的一题，再独立重做；之后根据这次暴露的问题安排下一轮练习。



图中箭头连接的是训练动作：同样从独立解题开始，省赛与团队训练加入任务分工，信息学训练加入子任务分析与自测，企业应用优化训练加入基础方案与实验比较；每轮都把实际困难带回复盘，再安排下一次练习。

开始参加比赛之后，可以继续保留同样的学习习惯：先把想法讲清楚，再写程序；通过一题后回看模型，没通过一题时留下能解释问题的具体输入。比赛目标会逐渐提高，这些步骤仍然能帮助我们判断下一步该练什么。

阅读、做题与复盘的安排

一次练习可以这样进行：先读一段讲解，手算对应过程；再合上示例实现；随后完成一题相近练习；最后选一题表述不同、但能使用同一思想的题目。若开始依赖题解才能写出每一步，就回到一个较短的例子，把缺的推导补齐。

定时比赛结束后，先补最接近自己当前能力的一题。记录没做出的原因：题意理解、模型选择、证明、实现或时间分配。下一次练习就围绕这个具体问题安排，而不是只按通过题数增加难度。

愿你继续保持对一道题的好奇，也愿你能在一次次手算、反例和改错中，逐渐把别人的方法变成自己的判断。

附录

题解仓库入口

练习题的参考解答放在 C++ / ACM 自学笔记配套题解仓库。做题时先留下自己的思路和手算过程，遇到推不下去的一步，再查对应题号。

仓库中有 107 道在线练习题和 4 项面向对象练习，共 111 份解答。同一道题出现在不同章节时，可以在题解开头找到它对应的位置。

题解先讲思路，再用具体数据手算，并附上代码和复杂度分析。递归返回、指针移动和状态转移等步骤配有图解。代码统一使用序言中的竞赛模板，每道题都有可以单独编译的 `.cpp` 文件。

按章节与题号查阅

笔记中的位置	仓库题解索引	解答数
一、C++ 语法基础；三、解题方法论	语言基础与条件建模	16
一（十五）STL	STL 容器与操作模拟	20
三、解题方法论	方法选择与事件模拟	4
四（二）大整数运算	大整数四则运算	5
五（二）至（四）	枚举、前缀和、差分与排序	6
五（五）（六）	基础数论与快速幂	6
五（九）（十）	双指针、窗口与二分	7
五（十一）	贪心策略	5
五（十二）搜索	递归、回溯、DFS、BFS 与记忆化	12
五（十三）动态规划	动态规划	14
五（八）（十四）（十五）	树、并查集与图论	12
二（九）面向对象部分的练习	面向对象的四项练习	4

按章节查阅时，从上表进入对应目录；已经知道题号时，可以打开 全部题号索引，用 `Ctrl+F` 搜索。例如：

- P1605 迷宫：沿一条路径尝试，返回时撤销标记，再继续尝试其他方向。
- P1434 滑雪：同一个格子的最长下降路径会被多次计算，保存第一次的结果，后面直接查表。
- P1048 采药：分别计算采这株药材和跳过它能得到的总价值，再取较大值。
- P2679 子串：把当前字符接在已有一段后面，或让它成为新一段的开头，分别统计两种情况。

从提示回到独立解题

还没想清楚解法时，可以先看分析，试着接着往下推。知道方法但走不清过程时，就拿一组小数据对照图中的变化。比如迷宫回溯，可以画出当前路径，看返回上一个格子时撤销了哪个标记，然后手算下一条路径。

已经写出程序但结果不同，可以用同一份输入对照自己的手算和题解中的过程。找到第一次算出不同结果的地方，再检查那一步的判断与更新。把这份输入留下来，改完程序后再运行一次。

看懂后，关掉题解重新写一遍。写到一个变量或循环时，试着说清它存的是什么、这一轮改变了什么；解释不清的地方，再回到对应讲解查阅。下一次复习可以直接从之前留下的错误输入开始。

想在本地运行参考代码，可以按仓库首页的命令编译对应的 `.cpp` 文件。`tests/verify.py` 会编译参考实现，并核对样例和手算数据。完成自己的程序后，提交到原题所在的评测平台，再对照题解回看解法。