【一】LZ信息营-算法课程体系与题目库清单

📊 课程体系总览表

课程编号	课程主题	核心算法/数据结构	难度等级	关联考核模块	关键考点
01	理解递归函数	递归、DFS、汉诺塔	★★☆☆☆	搜索基础	递归思维、分治思想
02	时间复杂度与排序	排序算法、复杂度分析	★☆☆☆☆	算法基础	复杂度分析、排序优化
03	STL库的应用	vector、map、priority_queue	★★☆☆☆	数据结构基础	STL熟练度、容器选择
04	网格寻路与BFS	BFS、队列、路径搜索	★★★☆☆	图论搜索	BFS模板、状态表示
05	图论与最短路算法	Dijkstra、负环、差分约束	★★★★☆	图论基础	最短路建模、算法选择
06	听说大家喜欢数学题	矩阵快速幂、逆元、扩展欧几里得	★★★★☆	数学基础	数论公式、快速幂
07	神奇的树状数组	BIT、差分、逆序对	★★★☆☆	高级数据结构	区间查询、单点更新
08	听说大家都学过贪心	排序贪心、后悔贪心	★★★☆☆	贪心策略	贪心选择、证明
09	二分枚举与按位枚举	二分答案、按位枚举	★★★☆☆	高效枚举	二分模板、状态压缩
10	背包九讲	01背包、完全背包、多重背包	★★★★☆	动态规划	背包模型、状态优化
11	暴力枚举的艺术	回溯、DFS剪枝、双向搜索	★★★☆☆	枚举优化	剪枝技巧、状态压缩
12	经典动态规划问题	线性DP、环形DP、股票问题	★★★★☆	动态规划	状态设计、转移方程
13	区间统计问题研究	线段树、单调队列、离散化	★★★★☆	区间算法	区间查询、离线处理
14	筛法只能求质数?	欧拉筛、约数个数、欧拉函数	★★★★☆	数论算法	筛法优化、积性函数
15	数论分块	整除分块、下取整求和	★★★★★	数学优化	分块思想、公式推导

课程编号	课程主题	核心算法/数据结构	难度等级	关联考核模块	关键考点
16	贡献思维和单调栈	单调栈、贡献法	★★★★☆	单调结构	贡献计算、单调栈模板
17	如何设计你的状态	状态DP、状态压缩	★★★★☆	动态规划	状态设计、压缩技巧
18	尺取法和双指针	滑动窗口、同向双指针	★★★☆☆	双指针	窗口维护、指针移动
19	神奇的根号算法	数论分块、莫队算法	★★★★★	分块算法	根号分治、复杂度平衡
20	前缀和优化技巧	前缀和、哈希表、差分	★★★☆☆	前缀优化	区间和转换、哈希优化
21	生成树与并查集	Kruskal、并查集、最小生成树	★★★★☆	图论算法	连通性判断、贪心选择

📝 各课程题目库详细整理

【课程01】理解递归函数

核心概念：递归三要素、记忆化递归、分治思想

题目编号	题目名称	算法类型	关键解法	难度
LS1209	斐波那契数列	递归/记忆化	递归+记忆化，避免重复计算	★☆☆☆☆
NC22227	约瑟夫环	递归公式	$f(n,m)=(f(n-1,m)+m)$	★★☆☆☆
LS1012	约瑟夫环2	递归/迭代	数学公式推导	★★☆☆☆
LS1028	汉诺塔问题	递归分治	move(n,A,B,C) = move(n-1,A,C,B) + move(1,A,B,C) + move(n-1,C,B,A)	★★☆☆☆

题目编号	题目名称	算法类型	关键解法	难度
LS1117	滑雪	记忆化搜索	DFS+记忆化，$dp[i][j] = max$(四个方向)	★★★☆☆
LS1082	01背包	递归/记忆化	$dfs(i,c) = max(dfs(i-1,c), dfs(i-1,c-w[i]) + v[i])$	★★★☆☆
LS1023	计算乘方	快速幂递归	pow(a,b) = b%2==0? pow($a*a$,b/2) : a*pow(a,b-1)	★★☆☆☆
LS1025	最大公约数	递归辗转相除	gcd(a,b)=gcd(b,a%b)	★☆☆☆☆
LS1083	完全背包	递归/记忆化	dfs(i,c)=max(dfs(i-1,c), dfs(i,c-w[i])+v[i])	★★★☆☆
LS1024	计算乘方和	递归分治	sum_pow(a,b)=b%2==0? (1+pow(a,b/2))*sum_pow(a,b/2-1): pow(a,b)+sum_pow(a,b-1)	★★★☆☆

【课程02】时间复杂度与排序

核心概念：复杂度分析、排序算法比较、自定义排序

题目编号	题目名称	算法类型	关键解法	难度
LS1030	比较排序算法	排序复杂度	分析不同排序算法时间复杂度	★☆☆☆☆
LS1212	自定义排序	排序规则	自定义cmp函数，lambda表达式	★★☆☆☆
LS1031	排序后输出位置	稳定排序	记录原始位置，按值排序后输出索引	★★☆☆☆
LS1032	求两数之和的最大值	排序贪心	排序后两端取数，复杂度分析	★★☆☆☆

【课程03】STL库的应用

核心概念：STL容器选择、算法函数使用、性能分析

题目编号	题目名称	数据结构	关键解法	难度
LS1033	坐标排序	vector+sort	自定义排序规则	★☆☆☆☆
LS1034	去重后第k小整数	set/排序	set自动去重排序，或sort+unique	★★☆☆☆
LT3556	质数字符串	string操作	字符串查找、质数判断	★★☆☆☆
LS1038	二维vector	嵌套容器	vector<vector>操作	★★☆☆☆
LS1039	多个priority_queue	优先队列	大顶堆、小顶堆应用	★★☆☆☆
LS1035	上网统计	map统计	map<string, int>计数	★★☆☆☆
LS1037	有序表的最小和	多路归并	priority_queue维护k路最小元素	★★★☆☆
LS1040	数据流中的众数	map统计	实时更新频率，维护当前众数	★★★☆☆
LS1036	出题法则	复杂排序	多关键字排序，自定义比较	★★★☆☆
LS1041	数据流中的中位数	双堆维护	大顶堆存较小一半，小顶堆存较大一半	★★★★☆

【课程04】网格寻路与BFS

核心概念：BFS模板、状态表示、路径记录

题目编号	题目名称	算法类型	关键解法	难度
B3616	【模板】队列	队列基础	STL queue基本操作	★☆☆☆☆
B3625	能否通过迷宫	BFS基础	BFS判断连通性	★★☆☆☆
P1443	马的遍历	BFS最短路	8方向BFS，记录步数	★★☆☆☆
P1135	奇怪的电梯	BFS状态	状态=(楼层)，BFS求最少按键次数	★★☆☆☆
P1451	细胞	BFS连通块	统计连通块数量	★★☆☆☆
P1162	填涂颜色	BFS染色	从边界BFS，区分内外	★★★☆☆
LS1216	迷宫寻路	BFS+状态	状态=(位置)，BFS求最短路径	★★★☆☆
P1649	最少拐弯次数	BFS方向	状态=(位置,方向)，记录拐弯次数	★★★☆☆

题目编号	题目名称	算法类型	关键解法	难度
LS1217	推箱子1	BFS+状态	状态=(人位置,箱子位置)，双重BFS	★★★★☆
LS1218	推箱子2	BFS+复杂状态	状态压缩，多个箱子	★★★★☆
P1825	传送迷宫	BFS+传送门	传送门特殊处理，记录传送状态	★★★★☆
D0326	混水摸鱼	BFS+多目标	多源BFS，计算最短距离	★★★☆☆
B3656	【模板】双端队列	数据结构	deque基本操作	★★☆☆☆
P4554	小明的游戏	BFS+不同代价	0-1 BFS或Dijkstra	★★★☆☆

【课程05】图论与最短路算法

核心概念：Dijkstra、负环检测、差分约束

题目编号	题目名称	算法类型	关键解法	难度
LS1219	如何封装我的算法	代码封装	函数封装，提高复用性	★★☆☆☆
LS1039	多个priority_queue	堆优化	Dijkstra的堆优化实现	★★☆☆☆
P4779	【模板】单源最短路径	Dijkstra	堆优化Dijkstra模板	★★★☆☆
P3385	【模板】负环	SPFA/Bellman	SPFA判负环，入队次数>n	★★★★☆
P4568	飞行路线	分层图Dijkstra	状态=(节点,使用次数)，建k+1层图	★★★★☆
LS1095	旅游巴士	最短路+时间限制	Dijkstra+时间窗口	★★★★☆
P5960	【模板】差分约束	SPFA/不等式	将不等式转化为边，求最长路/最短路	★★★★☆
P1629	邮递员送信	往返最短路	正向+反向图，两次Dijkstra	★★★☆☆
LS1108	聚会	多源最短路	从多个点出发的最短路	★★★☆☆

【课程06】听说大家喜欢数学题

核心概念：矩阵快速幂、逆元、扩展欧几里得

题目编号	题目名称	算法类型	关键解法	难度
LS1156	斐波那契数列_矩阵加速	矩阵快速幂	[[1,1],[1,0]]^n	★★★★☆
LS1161	斐波那契数列_更复杂	矩阵扩展	更高阶递推的矩阵表示	★★★★☆
LS1220	扩展欧几里得算法	扩展欧几里得	ax+by=gcd(a,b)求解	★★★★☆
LS1024	计算乘方和	快速幂+等比求和	分治求等比数列和	★★★☆☆

【课程07】神奇的树状数组

核心概念：BIT单点更新区间查询、差分、逆序对

题目编号	题目名称	算法类型	关键解法	难度
LS0010	【课件】神奇的树状数组	BIT模板	树状数组基本原理和操作	★★☆☆☆
P3374	【模板】单点更新，区间求和	BIT基础	add(x,v), sum(r)-sum(l-1)	★★☆☆☆
LS1104	差分与前缀和	差分数组	区间更新，单点查询	★★★☆☆
P3368	【模板】区间更新，单点求和	BIT+差分	维护差分数组，单点查询即前缀和	★★★☆☆
P3372	【模板】区间更新，区间求和	BIT+差分扩展	维护两个BIT：B1[i], B2[i]	★★★★☆
P1908	逆序对	BIT+离散化	从右向左扫描，统计比当前小的个数	★★★☆☆

【课程08】听说大家都学过贪心

核心概念：线段覆盖、排序贪心、后悔贪心

题目编号	题目名称	算法类型	关键解法	难度
LS1058	看电影	线段覆盖	按结束时间排序，贪心选择	★★☆☆☆
LS1060	线段覆盖	区间选择	按右端点排序，不相交则选择	★★★☆☆

题目编号	题目名称	算法类型	关键解法	难度
LS1061	排队难题	排序贪心	调整顺序使总等待时间最小	★★★☆☆
LS1062	拼接字符串	字典序贪心	自定义排序：a+b<b+a则a在前	★★★☆☆
LS1197	后悔贪心	反悔贪心	优先队列维护可选集合	★★★★☆
LS1064	逛超市_简单	贪心选择	按性价比排序选择	★★★☆☆
LS1065	逛超市_困难	后悔贪心	先买再后悔替换	★★★★☆
LS1059	剪刀石头布	策略贪心	根据对手历史选择最优策略	★★★☆☆
LS1066	最小差值	排序贪心	排序后取相邻差最小值	★★☆☆☆
LS1063	添加最少硬币	贪心构造	确保1~x都能表示，添加x+1	★★★☆☆

【课程09】二分枚举与按位枚举

核心概念：二分答案、按位枚举、最大化最小值

题目编号	题目名称	算法类型	关键解法	难度
LS1072	分割数组的最大值	二分答案	最小化最大值，贪心检查可行性	★★★☆☆
LS1074	分割数组的最小值	二分答案	最大化最小值，贪心检查	★★★☆☆
LS1073	查找元素位置	二分查找	lower_bound, upper_bound	★★☆☆☆
LS1079	导弹拦截	贪心+二分	最长不升子序列，O(nlogn)	★★★★☆
LS1075	青蛙过河	二分答案+贪心	检查能否跳过，维护可达位置	★★★★☆
P13822	白露为霜	二分查找	在有序数组中查找特定条件	★★★☆☆
LS1077	数的三次方根	浮点数二分	二分求立方根，控制精度	★★☆☆☆
LS1078	最大化平均值	二分答案+转化	分数规划，检查∑(ai-x*bi)≥0	★★★★☆
P12733	磨合	二分答案	最小化最大值问题	★★★☆☆
P4377	Talent Show G	二分答案+背包	分数规划，0-1分数背包	★★★★☆

【课程10】背包九讲

核心概念：01背包、完全背包、多重背包、分组背包

题目编号	题目名称	背包类型	关键解法	难度
LS0008	【课件】背包九讲	综合模板	各类背包问题模板	★★★☆☆
LS1082	01背包	01背包	dp[j] = max(dp[j],dp[j-w]+v)逆序	★★☆☆☆
LS1232	01背包之2	01背包变体	容量恰好为C的最大价值	★★★☆☆
LS1083	完全背包	完全背包	dp[j] = max(dp[j],dp[j-w]+v)顺序	★★☆☆☆
LS1084	多重背包	多重背包	二进制拆分优化	★★★☆☆
LS1085	混合背包	混合背包	分类处理，01逆序，完全顺序	★★★★☆
LS1086	二维费用背包	二维背包	`dp[j][k]` 双重循环	★★★☆☆
LS1087	分组背包	分组背包	每组最多选一个，组内循环	★★★★☆
LS1088	有依赖的背包	树形背包	树形DP，后序遍历	★★★★★
LS1093	求背包的具体方案	方案输出	记录转移路径，逆向推导	★★★★☆
LS1229	受限的数据	背包优化	根据数据范围选择算法	★★★★☆

【课程11】暴力枚举的艺术

核心概念：回溯、剪枝、状态压缩、双向搜索

题目编号	题目名称	算法类型	关键解法	难度
LS1133	部分和问题	DFS回溯	每个数选或不选	★★☆☆☆
LS1129	排列数字	全排列	DFS回溯，vis标记	★★☆☆☆
LS1130	组合数字	组合枚举	DFS回溯，限制长度	★★☆☆☆
LS1131	八皇后	DFS回溯	行、列、对角线检查	★★★☆☆
P1123	取数游戏	DFS+剪枝	不能相邻取数，最大和	★★★☆☆
P4799	世界冰球锦标赛	折半搜索	分成两半，分别枚举再合并	★★★★☆
LS1128	铺地砖	状态压缩DP	状压DP，转移考虑铺砖方式	★★★★☆
LS1193	激光枪	枚举+几何	枚举直线，统计线上点数	★★★★☆

题目编号	题目名称	算法类型	关键解法	难度
P3067	平衡子数组	折半枚举	分成两半，meet in the middle	★★★★★

【课程12】经典动态规划问题

核心概念：线性DP、环形处理、股票问题系列

题目编号	题目名称	算法类型	关键解法	难度
LS1016	最大子数组和	线性DP	Kadane算法，dp[i] = max(nums[i],dp[i-1]+nums[i])	★★☆☆☆
LS1250	最大子数组和	线性DP	同上，基础模板题	★★☆☆☆
LS1181	最大2段和	分段DP	前后缀分解，pre[i] + suf[i+1]	★★★☆☆
LS1251	最大环形子数组和	环形DP	两种情况：不环形(正常)，环形(总和-最小子数组和)	★★★★☆
LS1176	买卖股票的最佳时机_1	一次交易	记录历史最小值，计算最大差价	★★☆☆☆
LS1177	买卖股票的最佳时机_2	无限交易	贪心：所有上涨都交易	★★☆☆☆
LS1178	买卖股票的最佳时机_3	最多两次	前后缀分解，前后各一次最大	★★★★☆
LS1252	买卖股票的最佳时机_4	最多k次	`dp[i][j][0/1]` 第i天第j次交易持有/不持有	★★★★★
LS1179	买卖股票含冷冻期	状态机DP	三个状态：持有、不持有(冷冻)、不持有(非冷冻)	★★★★☆
LS1253	买卖股票的最佳时机_5	含手续费	状态机DP，卖出时扣除手续费	★★★★☆
P1121	环状最大两段子段和	环形分段	最大两段和，考虑环形情况	★★★★★

【课程13】区间统计问题研究

核心概念：线段树、单调队列、离散化、逆序对

题目编号	题目名称	数据结构	关键解法	难度
LS1226	平缓的曲线	线段树	区间最大值最小值查询	★★★☆☆

题目编号	题目名称	数据结构	关键解法	难度
P3374	【模板】单点更新，区间求和	线段树/BIT	基础线段树模板，维护区间和	★★☆☆☆
P1908	逆序对	BIT/线段树	离散化+从右向左统计	★★★☆☆
LS1227	单点更新，区间最值	线段树	维护区间最大值	★★★☆☆
LS1228	构造回文数组	线段树/贪心	对称位置配对，区间查询辅助	★★★★☆
P1886	滑动窗口	单调队列	维护窗口最大值/最小值	★★★☆☆
P1725	琪露诺	单调队列优化DP	dp[i] = max(dp[i-k..i-1])+a[i]，单调队列维护	★★★★☆
P2344	Generic Cow Protests	树状数组优化DP	dp[i]=∑dp[j] where sum[j+1..i]≥0	★★★★☆

【课程14】筛法只能求质数?

核心概念：欧拉筛、约数个数、约数和、欧拉函数

题目编号	题目名称	算法类型	关键解法	难度
LS1163	埃氏筛和欧拉筛	筛法基础	埃氏筛O(nloglogn)，欧拉筛O(n)	★★☆☆☆
LS1233	多个数分解质因数	质因数分解	预处理最小质因子，快速分解	★★★☆☆
LS1234	相等	约数相关	使用约数个数/约数和公式	★★★☆☆
LS1236	区间筛法	区间筛	[L,R]区间筛，用质数筛区间	★★★★☆
LS1231	连续的自然数	筛法应用	欧拉筛求连续质数/合数	★★★★☆
LS1235	最值求高	约数问题	最大公约数相关性质	★★★☆☆
LS1237	最大公约数计数	GCD计数	使用欧拉函数性质	★★★★☆
LS1164	约数个数、约数和、欧拉函数	筛法求积性函数	线性筛同时计算d(n),σ(n),φ(n)	★★★★☆

【课程15】数论分块

核心概念：整除分块、下取整求和、公式推导

题目编号	题目名称	算法类型	关键解法	难度
P2398	GCD SUM	数论分块	∑∑gcd(i,j) = ∑φ(d)*floor(n/d)²	★★★★★
LS1230	简洁的数学	数论分块	∑floor(n/i)*i 优化计算	★★★★☆
P2261	余数求和	数论分块	∑k%i = n*k* - ∑*floor(k/i)*i	★★★★☆
LS1261	【提高】数论分块	基础分块	计算∑floor(n/i)	★★★☆☆
LS1262	【提高】多维数论分块	多维分块	∑∑floor(n/i)*floor(m/j)	★★★★☆

【课程16】贡献思维和单调栈

核心概念：单调栈、贡献法、子数组统计

题目编号	题目名称	算法类型	关键解法	难度
LS1247	接雨水	单调栈	每个位置能接的水 = min(左边最大,右边最大) - 高度	★★★☆☆
LS1243	子数组之和	贡献法	统计每个元素在多少子数组中	★★☆☆☆
LS1244	子数组的最小值之和	单调栈	找到每个元素作为最小值的左右边界	★★★★☆
LS1199	柱状图中最大的矩形	单调栈	找到每个柱子左右第一个更矮的	★★★★☆
LS1200	最大矩形	单调栈+逐行处理	每行作为底，转化为柱状图问题	★★★★☆
LS1245	子数组的最值之差	单调栈	分别统计最大值贡献和最小值贡献	★★★★☆
LS1246	子序列的最值之差	贡献法	考虑每个元素作为最大/最小的贡献	★★★★☆
LS1248	二进制中的个数	位运算+贡献	统计每个bit位在多少子数组中为1	★★★★☆
LS1249	异或和	异或+贡献	按位考虑，统计异或贡献	★★★★☆

【课程17】如何设计你的状态

核心概念：状态设计、状态压缩、多维DP

题目编号	题目名称	算法类型	关键解法	难度
LS1016	最大子数组和	状态DP	dp[i]表示以i结尾的最大子数组和	★★☆☆☆
LS1250	最大子数组和	状态优化	滚动变量替代数组	★★☆☆☆
LS1181	最大2段和	状态分解	pre[i]前i个的最大一段，suf[i]后i个的最大一段	★★★☆☆
LS1251	最大环形子数组和	状态分类	分两种情况：不跨环和跨环	★★★★☆
LS1176	买卖股票1	状态机	两个状态：持有股票、不持有股票	★★☆☆☆
LS1177	买卖股票2	状态机	贪心简化，所有上涨都交易	★★☆☆☆
LS1178	买卖股票3	状态机	四个状态：第一次持有/不持有，第二次持有/不持有	★★★★☆
LS1252	买卖股票4	状态机	2k个状态，奇数次持有，偶数次不持有	★★★★★
LS1179	买卖股票含冷冻期	状态机	三个状态：持有、冷冻期、可购买	★★★★☆
LS1253	买卖股票5	状态机	含手续费，卖出时扣除	★★★★☆

【课程18】尺取法和双指针

核心概念：滑动窗口、同向双指针、相向双指针

题目编号	题目名称	算法类型	关键解法	难度
LS1256	2数之和	相向双指针	排序后左右指针向中间移动	★★☆☆☆
LS1257	2数之差	相向双指针	排序后固定差值找目标	★★★☆☆
LS1255	k个最接近的数	双指针+滑动窗口	找到最近位置后扩展窗口	★★★☆☆
P1638	包含所有数的最短区间	滑动窗口	统计窗口内不同元素个数	★★★☆☆
LS1259	字符出现至少k次的子字符串	滑动窗口	统计字符频率，满足条件时收缩	★★★★☆

题目编号	题目名称	算法类型	关键解法	难度
LS1260	求和游戏	前缀和+双指针	维护窗口和，寻找目标区间	★★★★☆
LS1254	统计稳定子数组的数目	双指针	维护最大最小值，统计稳定区间	★★★★☆
LS1258	极差不超过k的分割数	双指针+滑动窗口	维护窗口极差，统计分割方式	★★★★☆
B4196	赛车游戏	双指针应用	根据速度差计算超车次数	★★★☆☆

【课程19】神奇的根号算法

核心概念：根号分治、莫队算法、分块思想

题目编号	题目名称	算法类型	关键解法	难度
LS1261	【提高】数论分块	数论分块	∑floor(n/i)分块计算	★★★☆☆
LS1262	【提高】多维数论分块	多维分块	∑∑floor(n/i)*floor(m/j)	★★★★☆
P2261	余数求和	数论分块	n*k* - ∑*floor(k/i)*i	★★★★☆
P3901	数列找不同	莫队算法	离线查询区间是否有重复	★★★★☆
P1494	小Z的袜子	莫队算法	概率计算，组合数公式	★★★★★
P2709	小B的询问	莫队算法	维护平方和，∑cnt[i]²	★★★★☆
LS1263	【省选】智力与模数	根号分治	根据模数大小分情况处理	★★★★★

【课程20】前缀和优化技巧

核心概念：前缀和、哈希表、差分数组

题目编号	题目名称	算法类型	关键解法	难度
LS1265	连续数组	前缀和+哈希	将`0`视为`-1`，求最长和为0子数组	★★★☆☆
LS1264	异或子数组	前缀异或+哈希	异或前缀和，a ⊕ b = 0 等价于 a = b	★★★★☆
LS1267	最长的平衡子串1	前缀和+状态	统计 0 和 1 的个数差	★★★☆☆
LS1266	最长的平衡子串2	前缀和扩展	多种字符的平衡条件	★★★★☆

题目编号	题目名称	算法类型	关键解法	难度
LS1269	维护数组	前缀和+差分	区间更新，前缀查询	★★★☆☆
P14253	旅行	前缀和优化	预处理前缀信息，快速计算	★★★★☆
P14359	异或和	前缀异或	按位统计，前缀异或性质	★★★★☆
LS1270	Load_Balancing_S	前缀和分割	枚举分割点，前缀后缀统计	★★★★☆
LS1268	【提高】异或序列	前缀异或	区间异或 = prefix[r] ⊕ prefix[l-1]	★★★★☆

【课程21】生成树与并查集

核心概念：Kruskal、并查集、最小生成树

题目编号	题目名称	算法类型	关键解法	难度
P1551	亲戚	并查集基础	连通性判断，union-find	★★☆☆☆
LS1276	【算法】最小生成树	Kruskal	按边权排序，贪心选择	★★★☆☆
LS1272	【算法】最小比率生成树	分数规划	二分答案，转化为最小生成树	★★★★★
P1194	买礼物	最小生成树变体	建图技巧，虚拟节点	★★★★☆
P2700	逐个击破	并查集+贪心	逆向思维，从大到小合并	★★★★★
P1396	营救	最小生成树	最大边权最小，Kruskal	★★★☆☆
LS1275	【算法】删边游戏	并查集+离线	离线处理，反向加边	★★★★★
LS1273	【算法】撤销与合并	可持久化并查集	主席树维护并查集历史	★★★★★
LS1274	【算法】向右看齐	并查集应用	维护下一个可用位置	★★★★☆

🎯 考核重点与备考策略

基础必考模块（60分）

1. **递归与搜索**（课程01、04）：DFS/BFS模板
2. **排序与STL**（课程02、03）：sort、容器使用
3. **基础DP**（课程12、17）：线性DP、状态设计
4. **贪心与双指针**（课程08、18）：经典贪心、滑动窗口

5. **数据结构基础**（课程07）：树状数组基本操作

核心提高模块（30分）

1. **动态规划进阶**（课程10）：背包九讲
2. **图论算法**（课程05）：最短路、生成树
3. **高级数据结构**（课程13）：线段树、单调队列
4. **数学基础**（课程06、14）：快速幂、筛法

高级挑战模块（10分）

1. **数论与分块**（课程15、19）：数论分块、根号算法
2. **贡献思维**（课程16）：单调栈、贡献法
3. **前缀优化**（课程20）：前缀和技巧

📚 备考建议

第一阶段：基础巩固（1-2周）

- 完成课程01-04所有题目，掌握递归和BFS
- 掌握课程02-03的STL和排序
- 完成课程07的树状数组基础题

第二阶段：核心提升（2-3周）

- 完成课程08、12、17、18的经典算法
- 掌握课程05的图论基础
- 完成课程10的背包问题

第三阶段：高级突破（1-2周）

- 挑战课程13、14、16、20
- 尝试课程15、19的难题
- 综合练习，提高解题速度

第四阶段：模拟冲刺（1周）

- 每日一套模拟题
- 时间控制训练（3小时/套）
- 错题回顾，查漏补缺

✅ 快速查找表（按算法类型）

算法类型	推荐课程	关键题目	掌握要点
递归/DFS	01	LS1028汉诺塔，LS1117滑雪	递归边界、记忆化

算法类型	推荐课程	关键题目	掌握要点
BFS搜索	04	P1443马的遍历，P1825传送迷宫	状态表示、队列使用
动态规划	10,12,17	LS1082 01背包，LS1176股票问题	状态设计、转移方程
贪心算法	08	LS1058看电影，LS1060线段覆盖	贪心策略证明
双指针	18	LS1256两数之和，P1638最短区间	滑动窗口维护
数据结构	03,07,13	P3374线段树，P1908逆序对	区间操作、离散化
图论	05,21	P4779最短路，LS1276最小生成树	Dijkstra、Kruskal
数学/数论	06,14,15	LS1156矩阵快速幂，P2261余数求和	快速幂、筛法、分块
高级技巧	16,19,20	LS1199最大矩形，P1494小Z的袜子	单调栈、莫队、前缀和

最后提醒：信息营选拔考察综合能力，不仅要掌握算法，还要能灵活运用。建议按照课程顺序循序渐进，打好基础后再挑战难题。每道题目都要理解透彻，做到举一反三。

预祝各位同学在信息营选拔中取得优异成绩！ 🚀

【二】LZ信息营-选拔考核全方位复习指南

📚 完整课程体系与算法分类

算法分类体系

```
┌─ 基础算法 (30%)
│  ├─ 递归与搜索 (课程01, 04)
│  ├─ 排序与STL (课程02, 03)
│  └─ 时间复杂度分析
│
├─ 核心算法 (40%)
│  ├─ 动态规划 (课程10, 12, 17)
│  ├─ 贪心与双指针 (课程08, 18)
│  ├─ 数据结构基础 (课程07)
│  └─ 图论基础 (课程05)
│
└─ 高级算法 (30%)
   ├─ 高级数据结构 (课程13)
   ├─ 数学与数论 (课程06, 14, 15)
   ├─ 贡献思维 (课程16)
   ├─ 优化技巧 (课程20)
   └─ 根号算法 (课程19)
```

🚀 核心算法模板库（使用i64）

一、递归与搜索模块

1.1 归三要素模板

```
// ==================== 递归算法系列 ====================

/**********************************************************************
 * 1. 阶乘计算（递归基础）
 * 问题：计算n的阶乘 n! = 1×2×...×n
 * 递归三要素：
 *   1. 定义：factorial(n) 计算n的阶乘
 *   2. 边界：n <= 1 时返回1
 *   3. 递推：factorial(n) = n × factorial(n-1)
 * 时间复杂度：O(n)，空间复杂度：O(n)（递归栈）
 **********************************************************************/
i64 factorial(i64 n) {
    // 边界条件（递归出口）
    if (n <= 1) return 1;

    // 递推关系（递归调用）
    return n * factorial(n - 1);
}

// 阶乘的迭代版本（避免栈溢出）
i64 factorial_iterative(i64 n) {
    i64 result = 1;
    for (i64 i = 2; i <= n; i++) {
        result *= i;
    }
    return result;
}


/**********************************************************************
 * 2. 斐波那契数列 (LS1209)
 * 问题：F(0)=0, F(1)=1, F(n)=F(n-1)+F(n-2)
 * 递归三要素：
 *   1. 定义：fibonacci(n) 计算第n项
 *   2. 边界：n <= 1 时返回n
 *   3. 递推：fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)
 * 注意：朴素递归时间复杂度O(2^n)，需要记忆化优化
 **********************************************************************/

// 朴素递归（指数复杂度，不推荐）
i64 fibonacci_naive(i64 n) {
    if (n <= 1) return n;
    return fibonacci_naive(n - 1) + fibonacci_naive(n - 2);
}

// 记忆化递归（自顶向下）
i64 fibonacci_memo_helper(i64 n, vector<i64>& memo) {
    if (n <= 1) return n;
    if (memo[n] != -1) return memo[n];  // 记忆化剪枝
```

```
    memo[n] = fibonacci_memo_helper(n - 1, memo) +
              fibonacci_memo_helper(n - 2, memo);
    return memo[n];
}

i64 fibonacci_memo(i64 n) {
    vector<i64> memo(n + 1, -1);
    return fibonacci_memo_helper(n, memo);
}

// 迭代版本（自底向上，推荐）
i64 fibonacci_iterative(i64 n) {
    if (n <= 1) return n;

    i64 prev2 = 0;  // F(0)
    i64 prev1 = 1;  // F(1)

    for (i64 i = 2; i <= n; i++) {
        i64 current = prev1 + prev2;
        prev2 = prev1;
        prev1 = current;
    }

    return prev1;
}

// 矩阵快速幂（对数复杂度）
i64 fibonacci_matrix(i64 n) {
    if (n <= 1) return n;

    // 使用矩阵快速幂
    auto multiply = [](vector<vector<i64>> a, vector<vector<i64>> b) {
        vector<vector<i64>> result(2, vector<i64>(2, 0));
        for (i64 i = 0; i < 2; i++) {
            for (i64 j = 0; j < 2; j++) {
                for (i64 k = 0; k < 2; k++) {
                    result[i][j] += a[i][k] * b[k][j];
                }
            }
        }
        return result;
    };

    vector<vector<i64>> base = {{1, 1}, {1, 0}};
    vector<vector<i64>> result = {{1, 0}, {0, 1}};  // 单位矩阵

    i64 power = n - 1;
    while (power > 0) {
        if (power & 1) {
            result = multiply(result, base);
        }
        base = multiply(base, base);
        power >>= 1;
    }
```

```
    return result[0][0];
}


/*******************************************************************************
 * 3. 汉诺塔问题 (LS1028)
 * 问题：将n个盘子从A柱移动到C柱，B柱作为辅助
 * 规则：1. 每次只能移动一个盘子 2. 大盘不能在小盘上
 * 递归三要素：
 *   1. 定义：hanoi(n, from, to, aux) 移动n个盘子
 *   2. 边界：n == 1时直接移动
 *   3. 递推：hanoi(n) = hanoi(n-1) + 移动第n个 + hanoi(n-1)
 * 时间复杂度：O(2^n)，空间复杂度：O(n)
 *******************************************************************************/
void hanoi(i64 n, char from, char to, char aux) {
    // 边界条件：只有一个盘子
    if (n == 1) {
        cout << from << " -> " << to << "\n";
        return;
    }

    // 递推关系：
    // 1. 将n-1个盘子从from移到aux（借助to）
    hanoi(n - 1, from, aux, to);

    // 2. 移动第n个盘子（最大的）
    cout << from << " -> " << to << "\n";

    // 3. 将n-1个盘子从aux移到to（借助from）
    hanoi(n - 1, aux, to, from);
}

// 汉诺塔步数计算（不输出具体步骤）
i64 hanoi_steps(i64 n) {
    if (n == 1) return 1;
    return 2 * hanoi_steps(n - 1) + 1;
}

// 汉诺塔迭代解法（模拟递归栈）
void hanoi_iterative(i64 n, char from, char to, char aux) {
    struct State {
        i64 n;
        char from, to, aux;
        bool move_direct;  // 是否需要直接移动
    };

    stack<State> stk;
    stk.push({n, from, to, aux, false});

    while (!stk.empty()) {
        State current = stk.top();
        stk.pop();

        if (current.n == 1) {
            cout << current.from << " -> " << current.to << "\n";
        } else {
```

```
            // 注意入栈顺序与递归调用相反
            if (!current.move_direct) {
                // 对应：hanoi(n-1, aux, to, from)
                stk.push({current.n - 1, current.aux, current.to, current.from,
false});
                // 对应：移动第n个
                stk.push({1, current.from, current.to, current.aux, false});
                // 对应：hanoi(n-1, from, aux, to)
                stk.push({current.n - 1, current.from, current.aux, current.to,
false});
            }
        }
    }
}


/**********************************************************************
 * 4. 约瑟夫环问题 (NC22227)
 * 问题：n个人围成一圈，从0开始报数，报到m-1的人出列，求最后剩下的人编号
 * 递归三要素：
 *   1. 定义：josephus(n, m) 返回n个人，报数到m-1时的幸存者编号
 *   2. 边界：n == 1时返回0（只剩一个人）
 *   3. 递推：josephus(n, m) = (josephus(n-1, m) + m) % n
 * 时间复杂度：O(n)，空间复杂度：O(n)
 **********************************************************************/
i64 josephus(i64 n, i64 m) {
    // 边界条件：只剩一个人，编号0
    if (n == 1) return 0;

    // 递推公式
    return (josephus(n - 1, m) + m) % n;
}

// 约瑟夫环迭代版本（避免递归栈溢出）
i64 josephus_iterative(i64 n, i64 m) {
    i64 result = 0;  // n=1时的结果

    // 从2个人开始递推
    for (i64 i = 2; i <= n; i++) {
        result = (result + m) % i;
    }

    return result;
}

// 约瑟夫环模拟（暴力验证）
i64 josephus_simulation(i64 n, i64 m) {
    vector<bool> alive(n, true);
    i64 current = 0;
    i64 count = n;

    while (count > 1) {
        // 数m-1个人
        for (i64 i = 0; i < m - 1; i++) {
            do {
                current = (current + 1) % n;
```

```
            } while (!alive[current]);
        }

        // 淘汰当前人
        alive[current] = false;
        count--;

        // 找到下一个活着的人
        do {
            current = (current + 1) % n;
        } while (!alive[current]);
    }

    // 找到唯一幸存者
    for (i64 i = 0; i < n; i++) {
        if (alive[i]) return i;
    }
    return -1;  // 不应该到达这里
}


/**********************************************************************
 * 5. 快速幂递归 (LS1023)
 * 问题：计算a^b，高效实现
 * 递归三要素：
 *   1. 定义：fast_pow(a, b) 计算a的b次方
 *   2. 边界：b == 0时返回1
 *   3. 递推：a^b = (a^(b/2))² 或 a × a^(b-1)
 * 时间复杂度：O(logb)，空间复杂度：O(logb)
 **********************************************************************/
i64 fast_pow_recursive(i64 a, i64 b) {
    // 边界条件
    if (b == 0) return 1;

    // 递推关系
    if (b % 2 == 0) {
        // b是偶数：a^b = (a^(b/2))²
        i64 half = fast_pow_recursive(a, b / 2);
        return half * half;
    } else {
        // b是奇数：a^b = a × a^(b-1)
        return a * fast_pow_recursive(a, b - 1);
    }
}

// 快速幂迭代版本（位运算）
i64 fast_pow_iterative(i64 a, i64 b) {
    i64 result = 1;

    while (b > 0) {
        if (b & 1) {  // b是奇数
            result *= a;
        }
        a *= a;       // a平方
        b >>= 1;      // b除以2
    }
```

```
    return result;
}

// 带模运算的快速幂
i64 fast_pow_mod(i64 a, i64 b, i64 mod) {
    i64 result = 1;
    a %= mod;  // 防止溢出

    while (b > 0) {
        if (b & 1) {
            result = (result * a) % mod;
        }
        a = (a * a) % mod;
        b >>= 1;
    }

    return result;
}


/**********************************************************************
 * 6. 全排列问题
 * 问题：生成数组的所有排列
 * 递归三要素：
 *   1. 定义：permute(nums, start) 生成从start开始的排列
 *   2. 边界：start == n时输出排列
 *   3. 递推：交换每个元素到start位置，递归生成后续排列
 * 时间复杂度：O(n!)，空间复杂度：O(n)
 **********************************************************************/
void permute_recursive(vector<i64>& nums, i64 start, vector<vector<i64>>& result)
{
    // 边界条件：到达末尾，保存当前排列
    if (start == nums.size()) {
        result.push_back(nums);
        return;
    }

    // 递推：将每个元素交换到当前位置，递归生成后续排列
    for (i64 i = start; i < nums.size(); i++) {
        swap(nums[start], nums[i]);           // 选择当前元素
        permute_recursive(nums, start + 1, result);  // 递归生成后续
        swap(nums[start], nums[i]);           // 回溯，恢复原状
    }
}

vector<vector<i64>> permute(vector<i64>& nums) {
    vector<vector<i64>> result;
    permute_recursive(nums, 0, result);
    return result;
}

// 使用next_permutation生成全排列（STL方法）
vector<vector<i64>> permute_stl(vector<i64> nums) {
    vector<vector<i64>> result;
```

```
    // 先排序，确保生成所有排列
    sort(nums.begin(), nums.end());

    do {
        result.push_back(nums);
    } while (next_permutation(nums.begin(), nums.end()));

    return result;
}


/*****************************************************************************
 * 7. 组合问题
 * 问题：从n个元素中选择k个的所有组合
 * 递归三要素：
 *   1. 定义：combine(n, k, start, path) 生成组合
 *   2. 边界：path长度等于k时保存组合
 *   3. 递推：选择或不选择当前元素
 * 时间复杂度：O(C(n,k))，空间复杂度：O(k)
 *****************************************************************************/
void combine_recursive(i64 n, i64 k, i64 start, vector<i64>& path,
vector<vector<i64>>& result) {
    // 边界条件：组合长度达到k
    if (path.size() == k) {
        result.push_back(path);
        return;
    }

    // 递推：从start开始选择元素
    for (i64 i = start; i <= n; i++) {
        path.push_back(i);                          // 选择当前元素
        combine_recursive(n, k, i + 1, path, result); // 递归选择后续
        path.pop_back();                            // 回溯，不选择当前元素
    }
}

vector<vector<i64>> combine(i64 n, i64 k) {
    vector<vector<i64>> result;
    vector<i64> path;
    combine_recursive(n, k, 1, path, result);
    return result;
}

// 组合数计算：C(n,k) = C(n-1,k-1) + C(n-1,k)
i64 combination_number(i64 n, i64 k) {
    if (k == 0 || k == n) return 1;
    return combination_number(n - 1, k - 1) + combination_number(n - 1, k);
}

// 组合数记忆化版本
i64 combination_number_memo(i64 n, i64 k, vector<vector<i64>>& memo) {
    if (k == 0 || k == n) return 1;
    if (memo[n][k] != -1) return memo[n][k];

    memo[n][k] = combination_number_memo(n - 1, k - 1, memo) +
                 combination_number_memo(n - 1, k, memo);
```

```
    return memo[n][k];
}


/**********************************************************************
 * 8. 二叉树遍历（递归基础）
 * 问题：二叉树的先序、中序、后序遍历
 * 递归三要素：
 *   1. 定义：traverse(node) 遍历以node为根的子树
 *   2. 边界：node == nullptr时返回
 *   3. 递推：根据遍历顺序递归左右子树
 **********************************************************************/
struct TreeNode {
    i64 val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(i64 x) : val(x), left(nullptr), right(nullptr) {}
};

// 先序遍历：根 → 左 → 右
void preorder_traversal(TreeNode* root, vector<i64>& result) {
    if (!root) return;  // 边界条件

    result.push_back(root->val);               // 访问根节点
    preorder_traversal(root->left, result);    // 遍历左子树
    preorder_traversal(root->right, result);   // 遍历右子树
}

// 中序遍历：左 → 根 → 右
void inorder_traversal(TreeNode* root, vector<i64>& result) {
    if (!root) return;

    inorder_traversal(root->left, result);     // 遍历左子树
    result.push_back(root->val);               // 访问根节点
    inorder_traversal(root->right, result);    // 遍历右子树
}

// 后序遍历：左 → 右 → 根
void postorder_traversal(TreeNode* root, vector<i64>& result) {
    if (!root) return;

    postorder_traversal(root->left, result);   // 遍历左子树
    postorder_traversal(root->right, result);  // 遍历右子树
    result.push_back(root->val);               // 访问根节点
}


/**********************************************************************
 * 9. 最大公约数（欧几里得算法）
 * 问题：计算两个数的最大公约数
 * 递归三要素：
 *   1. 定义：gcd(a, b) 计算a和b的最大公约数
 *   2. 边界：b == 0时返回a
 *   3. 递推：gcd(a, b) = gcd(b, a % b)
 * 时间复杂度：O(log(min(a,b)))，空间复杂度：O(log(min(a,b)))
 **********************************************************************/
```

```cpp
i64 gcd_recursive(i64 a, i64 b) {
    // 边界条件
    if (b == 0) return a;

    // 递推关系
    return gcd_recursive(b, a % b);
}

// 迭代版本
i64 gcd_iterative(i64 a, i64 b) {
    while (b != 0) {
        i64 temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}

// 最小公倍数
i64 lcm(i64 a, i64 b) {
    return a / gcd_iterative(a, b) * b;  // 先除后乘防止溢出
}


/*****************************************************************************
 * 10. 分治算法：归并排序
 * 问题：使用分治思想排序数组
 * 递归三要素：
 *   1. 定义：merge_sort(arr, left, right) 排序arr[left..right]
 *   2. 边界：left >= right时返回（只有一个或零个元素）
 *   3. 递推：分为两半分别排序，然后合并
 * 时间复杂度：O(nlogn)，空间复杂度：O(n)
 *****************************************************************************/
void merge(vector<i64>& arr, i64 left, i64 mid, i64 right) {
    vector<i64> temp(right - left + 1);
    i64 i = left, j = mid + 1, k = 0;

    // 合并两个有序数组
    while (i <= mid && j <= right) {
        if (arr[i] <= arr[j]) {
            temp[k++] = arr[i++];
        } else {
            temp[k++] = arr[j++];
        }
    }

    // 复制剩余元素
    while (i <= mid) temp[k++] = arr[i++];
    while (j <= right) temp[k++] = arr[j++];

    // 复制回原数组
    for (i64 idx = 0; idx < k; idx++) {
        arr[left + idx] = temp[idx];
    }
}
```

```cpp
void merge_sort(vector<i64>& arr, i64 left, i64 right) {
    // 边界条件
    if (left >= right) return;

    // 递推关系：分治
    i64 mid = left + (right - left) / 2;
    merge_sort(arr, left, mid);      // 排序左半部分
    merge_sort(arr, mid + 1, right); // 排序右半部分
    merge(arr, left, mid, right);    // 合并两个有序部分
}


/**************************************************************************
 * 11. 分治算法：快速排序
 * 问题：使用分治思想排序数组
 * 递归三要素：
 *   1. 定义：quick_sort(arr, left, right) 排序arr[left..right]
 *   2. 边界：left >= right时返回
 *   3. 递推：选择基准，划分数组，递归排序
 * 时间复杂度：平均O(nlogn)，最坏O(n²)，空间复杂度：O(logn)
 **************************************************************************/
i64 partition(vector<i64>& arr, i64 left, i64 right) {
    // 选择最后一个元素作为基准
    i64 pivot = arr[right];
    i64 i = left - 1;  // 小于基准的元素的边界

    for (i64 j = left; j < right; j++) {
        if (arr[j] <= pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }

    // 将基准放到正确位置
    swap(arr[i + 1], arr[right]);
    return i + 1;
}

void quick_sort(vector<i64>& arr, i64 left, i64 right) {
    // 边界条件
    if (left >= right) return;

    // 递推关系：分治
    i64 pivot_idx = partition(arr, left, right);  // 划分
    quick_sort(arr, left, pivot_idx - 1);         // 排序左半部分
    quick_sort(arr, pivot_idx + 1, right);        // 排序右半部分
}


/**************************************************************************
 * 12. 递归转迭代通用模板
 * 使用显式栈模拟递归调用
 **************************************************************************/
struct RecursiveState {
    i64 n;
    i64 stage;  // 记录递归阶段
```

```
    // 其他参数...
};

void recursive_to_iterative_example(i64 n) {
    stack<RecursiveState> stk;
    stk.push({n, 0});  // 初始状态

    while (!stk.empty()) {
        RecursiveState& current = stk.top();

        switch (current.stage) {
            case 0:  // 相当于递归函数开始
                if (current.n <= 1) {
                    // 相当于边界条件
                    // 处理结果...
                    stk.pop();
                } else {
                    current.stage = 1;
                    stk.push({current.n - 1, 0});  // 相当于递归调用
                }
                break;

            case 1:  // 第一次递归调用返回后
                // 处理第一次递归的结果...
                current.stage = 2;
                stk.push({current.n - 2, 0});  // 第二次递归调用
                break;

            case 2:  // 第二次递归调用返回后
                // 处理结果并合并...
                stk.pop();  // 当前调用结束
                break;
        }
    }
}
```

📊 递归三要素总结

要素	说明	示例（阶乘）
定义	明确函数的功能和参数	factorial(n) 计算n的阶乘
边界	最小情况，递归出口	n <= 1 时返回1
递推	如何分解为子问题	factorial(n) = n × factorial(n-1)

📊 常见递归问题分类

问题类型	特点	经典例题
数学计算	直接数学公式递归	阶乘、斐波那契、组合数
分治算法	分而治之，合并结果	归并排序、快速排序
回溯算法	尝试所有可能，回溯	全排列、组合、N皇后

问题类型	特点	经典例题
树形递归	处理树状结构	二叉树遍历、树的高度
图遍历	图的深度优先搜索	迷宫问题、连通分量
动态规划	重叠子问题，记忆化	斐波那契（记忆化）

递归复杂度分析

递归类型	时间复杂度	空间复杂度	示例
线性递归	O(n)	O(n)	阶乘、链表遍历
二叉树递归	O(n)	O(h)	二叉树遍历
指数递归	O(2^n)	O(n)	朴素斐波那契
分治递归	O(nlogn)	O(logn)	归并排序
组合递归	O(C(n,k))	O(k)	组合问题
排列递归	O(n!)	O(n)	全排列

递归优化技术

优化技术	原理	应用场景
记忆化	缓存计算结果，避免重复计算	斐波那契、动态规划
尾递归	递归调用在最后，可优化为迭代	某些累加、累乘
迭代转换	使用栈模拟递归	深度过大时避免栈溢出
剪枝	提前终止不可能的分支	回溯算法、搜索
分治优化	减少子问题规模	快速排序、归并排序

递归 vs 迭代对比

方面	递归	迭代
代码简洁性	高，更接近数学定义	低，需要显式控制
可读性	高，逻辑清晰	低，需要理解循环
空间效率	低，有栈开销	高，只有变量
性能	可能有函数调用开销	通常更快
栈溢出风险	有，深度过大时	无，只有堆内存限制
适用场景	树、图、分治问题	线性处理、简单循环

递归常见错误与规避

错误类型	现象	规避方法
无边界条件	无限递归，栈溢出	总是先写边界条件
边界条件错误	提前终止或漏解	仔细测试边界情况
重复计算	指数级复杂度	使用记忆化缓存结果
栈溢出	递归深度过大	改为迭代或尾递归优化
逻辑错误	递推关系错误	验证小规模情况
状态污染	全局变量影响	使用参数传递状态

📊 关联题目索引

题目编号	题目名称	核心递归	难度
LS1023	计算乘方	快速幂递归	⭐⭐
LS1028	汉诺塔问题	经典递归	⭐⭐
LS1209	斐波那契数列	记忆化递归	⭐⭐
NC22227	约瑟夫环	数学递归	⭐⭐⭐
LS1054	全排列问题	回溯递归	⭐⭐⭐
LS1065	组合问题	组合递归	⭐⭐⭐

【递归核心要点】

方面	关键点	示例
三要素	定义、边界、递推	factorial(n) = n × factorial(n-1)
设计步骤	分析问题，确定递归结构	汉诺塔：移动n-1个，移动第n个，移动n-1个
优化技术	记忆化、尾递归、迭代转换	斐波那契记忆化避免重复计算
复杂度分析	递归树，主定理	归并排序：T(n) = 2T(n/2) + O(n)

【学习建议】

1. **理解原理**：掌握递归的数学基础（数学归纳法）
2. **从简单开始**：先实现阶乘、斐波那契等简单递归
3. **画递归树**：帮助理解递归过程和复杂度
4. **掌握优化**：学习记忆化、迭代转换等优化技术
5. **多实践**：解决各种递归问题，积累经验

【常见错误】

1. **无限递归**：忘记边界条件或边界条件错误
2. **栈溢出**：递归深度过大，改为迭代
3. **重复计算**：未使用记忆化，指数级复杂度

4. **逻辑错误**：递推关系不正确
5. **状态污染**：错误使用全局变量

【性能优化】

1. **记忆化**：缓存计算结果
2. **尾递归优化**：某些编译器可优化
3. **迭代转换**：使用栈模拟递归
4. **动态规划**：自底向上计算
5. **剪枝**：提前终止不可能的分支

【调试技巧】

1. **打印递归深度**：cout << "递归深度: " << depth << endl;
2. **小数据测试**：验证边界情况和简单情况
3. **画递归树**：可视化递归过程
4. **使用调试器**：跟踪调用栈
5. **对比迭代版本**：验证递归正确性

【扩展应用】

1. **回溯算法**：N皇后、数独求解
2. **分治算法**：最近点对、矩阵乘法
3. **树形DP**：树上动态规划
4. **图遍历**：深度优先搜索
5. **函数式编程**：递归是函数式编程的核心

1.2 网格DFS与BFS模板（课程04）

📦 核心代码模板

```
// ==================== 网格DFS与BFS模板 ====================

/*****************************************************************************
 * 1. BFS求最短路径通用模板 (P1443 马的遍历)
 * 问题：在网格中找到从起点到终点的最短路径长度（无权图）
 * 解法：广度优先搜索，队列实现，最先到达终点的路径最短
 * 时间复杂度：O(n×m)，空间复杂度：O(n×m)
 *****************************************************************************/
// 8个方向（马走日）
vector<i64> dx_horse = {-2, -2, -1, -1, 1, 1, 2, 2};
vector<i64> dy_horse = {-1, 1, -2, 2, -2, 2, -1, 1};

// 4个方向（上下左右）
vector<i64> dx_4 = {0, 0, 1, -1};
vector<i64> dy_4 = {1, -1, 0, 0};

// 8个方向（包括对角线）
vector<i64> dx_8 = {0, 0, 1, -1, 1, 1, -1, -1};
vector<i64> dy_8 = {1, -1, 0, 0, 1, -1, 1, -1};
```

```cpp
i64 bfs_min_steps(i64 n, i64 m, i64 start_x, i64 start_y,
                  i64 target_x, i64 target_y,
                  const vector<i64>& dx, const vector<i64>& dy) {
    // 边界检查
    if (start_x < 0 || start_x >= n || start_y < 0 || start_y >= m ||
        target_x < 0 || target_x >= n || target_y < 0 || target_y >= m) {
        return -1;
    }

    // 距离数组，-1表示未访问
    vector<vector<i64>> dist(n, vector<i64>(m, -1));
    dist[start_x][start_y] = 0;

    // 使用队列进行BFS
    queue<pair<i64, i64>> q;
    q.push({start_x, start_y});

    while (!q.empty()) {
        auto [x, y] = q.front();
        q.pop();

        // 如果到达目标点，返回距离
        if (x == target_x && y == target_y) {
            return dist[x][y];
        }

        // 遍历所有可能的方向
        for (i64 i = 0; i < dx.size(); i++) {
            i64 nx = x + dx[i];
            i64 ny = y + dy[i];

            // 检查新位置是否合法且未访问
            if (nx >= 0 && nx < n && ny >= 0 && ny < m && dist[nx][ny] == -1) {
                dist[nx][ny] = dist[x][y] + 1;
                q.push({nx, ny});
            }
        }
    }

    return -1;  // 不可达
}

// 马的遍历专用函数
i64 bfs_horse(i64 n, i64 m, i64 start_x, i64 start_y, i64 target_x, i64 target_y)
{
    return bfs_min_steps(n, m, start_x, start_y, target_x, target_y, dx_horse,
dy_horse);
}

// 获取完整距离矩阵（不只是到目标点的距离）
vector<vector<i64>> bfs_all_distances(i64 n, i64 m, i64 start_x, i64 start_y,
                                      const vector<i64>& dx, const vector<i64>&
dy) {
    vector<vector<i64>> dist(n, vector<i64>(m, -1));
    dist[start_x][start_y] = 0;
```

```
    queue<pair<i64, i64>> q;
    q.push({start_x, start_y});

    while (!q.empty()) {
        auto [x, y] = q.front();
        q.pop();

        for (i64 i = 0; i < dx.size(); i++) {
            i64 nx = x + dx[i];
            i64 ny = y + dy[i];

            if (nx >= 0 && nx < n && ny >= 0 && ny < m && dist[nx][ny] == -1) {
                dist[nx][ny] = dist[x][y] + 1;
                q.push({nx, ny});
            }
        }
    }

    return dist;
}


/*******************************************************************************
 * 2. BFS连通块计数 (P1451 细胞)
 * 问题：统计网格中连通区域的数量
 * 解法：对每个未访问的1进行BFS，标记整个连通区域
 * 时间复杂度：O(n×m)，空间复杂度：O(n×m)
 ******************************************************************************/
i64 bfs_connected_components(vector<vector<i64>>& grid) {
    if (grid.empty() || grid[0].empty()) return 0;

    i64 m = grid.size();
    i64 n = grid[0].size();

    // 访问标记数组
    vector<vector<bool>> visited(m, vector<bool>(n, false));
    i64 components = 0;

    // 4个方向（上下左右）
    vector<i64> dx = {0, 0, 1, -1};
    vector<i64> dy = {1, -1, 0, 0};

    for (i64 i = 0; i < m; i++) {
        for (i64 j = 0; j < n; j++) {
            // 找到未访问的单元格且值为1（表示细胞）
            if (grid[i][j] == 1 && !visited[i][j]) {
                components++;

                // BFS遍历整个连通区域
                queue<pair<i64, i64>> q;
                q.push({i, j});
                visited[i][j] = true;

                while (!q.empty()) {
                    auto [x, y] = q.front();
```

```
                    q.pop();

                    // 遍历4个方向
                    for (i64 d = 0; d < 4; d++) {
                        i64 nx = x + dx[d];
                        i64 ny = y + dy[d];

                        // 检查边界和访问条件
                        if (nx >= 0 && nx < m && ny >= 0 && ny < n &&
                            grid[nx][ny] == 1 && !visited[nx][ny]) {
                            visited[nx][ny] = true;
                            q.push({nx, ny});
                        }
                    }
                }
            }
        }
    }

    return components;
}

// 8方向连通（包括对角线）
i64 bfs_connected_components_8dir(vector<vector<i64>>& grid) {
    if (grid.empty() || grid[0].empty()) return 0;

    i64 m = grid.size();
    i64 n = grid[0].size();
    vector<vector<bool>> visited(m, vector<bool>(n, false));
    i64 components = 0;

    // 8个方向（包括对角线）
    vector<i64> dx = {0, 0, 1, -1, 1, 1, -1, -1};
    vector<i64> dy = {1, -1, 0, 0, 1, -1, 1, -1};

    for (i64 i = 0; i < m; i++) {
        for (i64 j = 0; j < n; j++) {
            if (grid[i][j] == 1 && !visited[i][j]) {
                components++;

                queue<pair<i64, i64>> q;
                q.push({i, j});
                visited[i][j] = true;

                while (!q.empty()) {
                    auto [x, y] = q.front();
                    q.pop();

                    for (i64 d = 0; d < 8; d++) {
                        i64 nx = x + dx[d];
                        i64 ny = y + dy[d];

                        if (nx >= 0 && nx < m && ny >= 0 && ny < n &&
                            grid[nx][ny] == 1 && !visited[nx][ny]) {
                            visited[nx][ny] = true;
                            q.push({nx, ny});
```

```
                        }
                    }
                }
            }
        }

    return components;
}

// 获取每个连通区域的大小
vector<i64> get_component_sizes(vector<vector<i64>>& grid) {
    if (grid.empty() || grid[0].empty()) return {};

    i64 m = grid.size();
    i64 n = grid[0].size();
    vector<vector<bool>> visited(m, vector<bool>(n, false));
    vector<i64> sizes;

    vector<i64> dx = {0, 0, 1, -1};
    vector<i64> dy = {1, -1, 0, 0};

    for (i64 i = 0; i < m; i++) {
        for (i64 j = 0; j < n; j++) {
            if (grid[i][j] == 1 && !visited[i][j]) {
                i64 size = 0;

                queue<pair<i64, i64>> q;
                q.push({i, j});
                visited[i][j] = true;

                while (!q.empty()) {
                    auto [x, y] = q.front();
                    q.pop();
                    size++;

                    for (i64 d = 0; d < 4; d++) {
                        i64 nx = x + dx[d];
                        i64 ny = y + dy[d];

                        if (nx >= 0 && nx < m && ny >= 0 && ny < n &&
                            grid[nx][ny] == 1 && !visited[nx][ny]) {
                            visited[nx][ny] = true;
                            q.push({nx, ny});
                        }
                    }
                }

                sizes.push_back(size);
            }
        }
    }

    return sizes;
}
```

```cpp
/*****************************************************************************
 * 3. 滑雪问题 - DFS记忆化搜索 (LS1117)
 * 问题：找到最长的严格递减路径（只能从高到低）
 * 解法：DFS+记忆化，对每个点计算从该点出发的最长路径
 * 时间复杂度：O(n×m)，空间复杂度：O(n×m)
 *****************************************************************************/
class SkiSolver {
private:
    vector<vector<i64>> grid;
    vector<vector<i64>> memo;
    vector<i64> dx = {0, 0, 1, -1};
    vector<i64> dy = {1, -1, 0, 0};
    i64 m, n;

    // DFS计算从(x,y)出发的最长滑雪路径
    i64 dfs(i64 x, i64 y) {
        // 如果已经计算过，直接返回
        if (memo[x][y] != -1) {
            return memo[x][y];
        }

        // 从当前点出发至少长度为1
        i64 max_len = 1;

        // 尝试4个方向
        for (i64 d = 0; d < 4; d++) {
            i64 nx = x + dx[d];
            i64 ny = y + dy[d];

            // 检查新位置是否合法且高度更低
            if (nx >= 0 && nx < m && ny >= 0 && ny < n &&
                grid[nx][ny] < grid[x][y]) {
                // 递归计算从新位置出发的路径长度
                i64 len_from_next = dfs(nx, ny) + 1;
                max_len = max(max_len, len_from_next);
            }
        }

        // 记忆化
        memo[x][y] = max_len;
        return max_len;
    }

public:
    SkiSolver(vector<vector<i64>>& heights) : grid(heights) {
        m = grid.size();
        n = grid[0].size();
        memo.assign(m, vector<i64>(n, -1));
    }

    i64 longest_path() {
        i64 max_path = 0;

        // 对每个点计算最长路径
        for (i64 i = 0; i < m; i++) {
```

```
            for (i64 j = 0; j < n; j++) {
                i64 path_len = dfs(i, j);
                max_path = max(max_path, path_len);
            }
        }

        return max_path;
    }

    // 获取最长路径的起点和具体路径
    pair<pair<i64, i64>, vector<pair<i64, i64>>> get_longest_path_details() {
        i64 max_path = 0;
        pair<i64, i64> start_point = {0, 0};

        // 先找到最长路径的起点
        for (i64 i = 0; i < m; i++) {
            for (i64 j = 0; j < n; j++) {
                i64 path_len = dfs(i, j);
                if (path_len > max_path) {
                    max_path = path_len;
                    start_point = {i, j};
                }
            }
        }

        // 重构最长路径
        vector<pair<i64, i64>> path;
        i64 x = start_point.first, y = start_point.second;
        path.push_back({x, y});

        while (true) {
            bool found_next = false;

            for (i64 d = 0; d < 4; d++) {
                i64 nx = x + dx[d];
                i64 ny = y + dy[d];

                if (nx >= 0 && nx < m && ny >= 0 && ny < n &&
                    grid[nx][ny] < grid[x][y] &&
                    memo[nx][ny] == memo[x][y] - 1) {
                    x = nx;
                    y = ny;
                    path.push_back({x, y});
                    found_next = true;
                    break;
                }
            }

            if (!found_next) break;
        }

        return {start_point, path};
    }
};

// 简化接口函数
```

```cpp
i64 longest_ski_path(vector<vector<i64>>& grid) {
    SkiSolver solver(grid);
    return solver.longest_path();
}


/**********************************************************************
 * 4. 01背包递归+记忆化版本 (LS1082)
 * 问题：经典01背包问题的递归解法
 * 解法：递归考虑选或不选当前物品，使用记忆化避免重复计算
 * 时间复杂度：O(n×capacity)，空间复杂度：O(n×capacity)
 **********************************************************************/
i64 knapsack_01_recursive(i64 i, i64 capacity,
                          vector<i64>& weights, vector<i64>& values,
                          vector<vector<i64>>& memo) {
    // 边界条件：没有物品或容量为0
    if (i == 0 || capacity == 0) {
        return 0;
    }

    // 如果已经计算过，直接返回
    if (memo[i][capacity] != -1) {
        return memo[i][capacity];
    }

    // 不选第i个物品（索引i-1）
    i64 result = knapsack_01_recursive(i - 1, capacity, weights, values, memo);

    // 如果可以选第i个物品，考虑选的情况
    if (weights[i - 1] <= capacity) {
        i64 take = values[i - 1] +
                   knapsack_01_recursive(i - 1, capacity - weights[i - 1],
                                         weights, values, memo);
        result = max(result, take);
    }

    // 记忆化
    memo[i][capacity] = result;
    return result;
}

// 包装函数
i64 knapsack_01_memoization(i64 n, i64 capacity,
                            vector<i64>& weights, vector<i64>& values) {
    // 记忆化数组，memo[i][j]表示前i个物品容量为j时的最大价值
    vector<vector<i64>> memo(n + 1, vector<i64>(capacity + 1, -1));
    return knapsack_01_recursive(n, capacity, weights, values, memo);
}


/**********************************************************************
 * 5. 完全背包递归+记忆化版本 (LS1083)
 * 问题：完全背包问题的递归解法
 * 解法：与01背包的区别在于选物品时可以重复选择
 * 时间复杂度：O(n×capacity)，空间复杂度：O(n×capacity)
 **********************************************************************/
```

```cpp
i64 knapsack_complete_recursive(i64 i, i64 capacity,
                                vector<i64>& weights, vector<i64>& values,
                                vector<vector<i64>>& memo) {
    // 边界条件
    if (i == 0 || capacity == 0) {
        return 0;
    }

    // 记忆化检查
    if (memo[i][capacity] != -1) {
        return memo[i][capacity];
    }

    // 不选第i个物品
    i64 result = knapsack_complete_recursive(i - 1, capacity,
                                             weights, values, memo);

    // 如果可以选第i个物品，考虑选的情况
    // 注意：完全背包中，选了之后还可以继续选同一个物品
    if (weights[i - 1] <= capacity) {
        i64 take = values[i - 1] +
                   knapsack_complete_recursive(i, capacity - weights[i - 1],
                                               weights, values, memo);
        result = max(result, take);
    }

    // 记忆化
    memo[i][capacity] = result;
    return result;
}

// 包装函数
i64 knapsack_complete_memoization(i64 n, i64 capacity,
                                  vector<i64>& weights, vector<i64>& values) {
    vector<vector<i64>> memo(n + 1, vector<i64>(capacity + 1, -1));
    return knapsack_complete_recursive(n, capacity, weights, values, memo);
}


/*****************************************************************************
 * 6. 迷宫连通性检查 (B3625 能否通过迷宫)
 * 问题：判断从起点是否能到达终点
 * 解法：BFS或DFS遍历，检查是否能到达终点
 * 时间复杂度：O(n×m)，空间复杂度：O(n×m)
 *****************************************************************************/
bool can_reach_destination(vector<vector<i64>>& maze,
                           i64 start_x, i64 start_y,
                           i64 target_x, i64 target_y) {
    if (maze.empty() || maze[0].empty()) return false;

    i64 m = maze.size();
    i64 n = maze[0].size();

    // 边界检查
    if (start_x < 0 || start_x >= m || start_y < 0 || start_y >= n ||
        target_x < 0 || target_x >= m || target_y < 0 || target_y >= n) {
```

```
        return false;
    }

    // 起点或终点不可达
    if (maze[start_x][start_y] == 0 || maze[target_x][target_y] == 0) {
        return false;
    }

    // BFS遍历
    vector<vector<bool>> visited(m, vector<bool>(n, false));
    queue<pair<i64, i64>> q;

    q.push({start_x, start_y});
    visited[start_x][start_y] = true;

    vector<i64> dx = {0, 0, 1, -1};
    vector<i64> dy = {1, -1, 0, 0};

    while (!q.empty()) {
        auto [x, y] = q.front();
        q.pop();

        // 到达终点
        if (x == target_x && y == target_y) {
            return true;
        }

        // 遍历4个方向
        for (i64 d = 0; d < 4; d++) {
            i64 nx = x + dx[d];
            i64 ny = y + dy[d];

            if (nx >= 0 && nx < m && ny >= 0 && ny < n &&
                maze[nx][ny] == 1 && !visited[nx][ny]) {
                visited[nx][ny] = true;
                q.push({nx, ny});
            }
        }
    }

    return false;
}


/**********************************************************************
 * 7. 传送迷宫问题 (P1825 传送迷宫)
 * 问题：迷宫中有传送门，可以瞬间传送到另一位置
 * 解法：BFS处理普通移动和传送
 * 时间复杂度：O(n×m)，空间复杂度：O(n×m)
 **********************************************************************/
struct Portal {
    i64 x1, y1, x2, y2;
};

i64 bfs_with_portals(vector<vector<i64>>& maze,
                     i64 start_x, i64 start_y,
```

```
                i64 target_x, i64 target_y,
                vector<Portal>& portals) {
    i64 m = maze.size();
    i64 n = maze[0].size();

    // 距离数组
    vector<vector<i64>> dist(m, vector<i64>(n, -1));
    dist[start_x][start_y] = 0;

    // 传送门映射：从位置到传送到的位置列表
    vector<vector<vector<pair<i64, i64>>>> portal_map(m,
        vector<vector<pair<i64, i64>>>(n));

    for (const auto& portal : portals) {
        if (maze[portal.x1][portal.y1] == 1 && maze[portal.x2][portal.y2] == 1) {
            portal_map[portal.x1][portal.y1].push_back({portal.x2, portal.y2});
            portal_map[portal.x2][portal.y2].push_back({portal.x1, portal.y1});
        }
    }

    queue<pair<i64, i64>> q;
    q.push({start_x, start_y});

    vector<i64> dx = {0, 0, 1, -1};
    vector<i64> dy = {1, -1, 0, 0};

    while (!q.empty()) {
        auto [x, y] = q.front();
        q.pop();

        // 到达终点
        if (x == target_x && y == target_y) {
            return dist[x][y];
        }

        // 情况1：普通移动
        for (i64 d = 0; d < 4; d++) {
            i64 nx = x + dx[d];
            i64 ny = y + dy[d];

            if (nx >= 0 && nx < m && ny >= 0 && ny < n &&
                maze[nx][ny] == 1 && dist[nx][ny] == -1) {
                dist[nx][ny] = dist[x][y] + 1;
                q.push({nx, ny});
            }
        }

        // 情况2：传送门移动
        for (const auto& [nx, ny] : portal_map[x][y]) {
            if (dist[nx][ny] == -1) {
                dist[nx][ny] = dist[x][y] + 1;  // 传送算一步
                q.push({nx, ny});
            }
        }
    }
```

```
    return -1;  // 不可达
}


/**************************************************************************
 * 8. 推箱子问题简化版 (LS1217 推箱子1)
 * 问题：人在迷宫中推箱子到目标位置
 * 解法：BFS搜索状态空间（人位置+箱子位置）
 * 时间复杂度：O(m²×n²)，空间复杂度：O(m²×n²)
 **************************************************************************/
struct PushBoxState {
    i64 person_x, person_y;  // 人位置
    i64 box_x, box_y;        // 箱子位置
    i64 steps;               // 已走步数

    // 用于unordered_set的哈希函数
    size_t hash() const {
        return ((person_x * 1000 + person_y) * 1000000 +
                (box_x * 1000 + box_y));
    }

    bool operator==(const PushBoxState& other) const {
        return person_x == other.person_x && person_y == other.person_y &&
               box_x == other.box_x && box_y == other.box_y;
    }
};

// 自定义哈希函数
struct PushBoxStateHash {
    size_t operator()(const PushBoxState& state) const {
        return state.hash();
    }
};

i64 push_box_min_steps(vector<vector<i64>>& maze,
                       i64 start_person_x, i64 start_person_y,
                       i64 start_box_x, i64 start_box_y,
                       i64 target_x, i64 target_y) {
    i64 m = maze.size();
    i64 n = maze[0].size();

    // 初始状态
    PushBoxState start_state = {start_person_x, start_person_y,
                                start_box_x, start_box_y, 0};

    // 状态队列
    queue<PushBoxState> q;
    q.push(start_state);

    // 状态访问记录
    unordered_set<PushBoxState, PushBoxStateHash> visited;
    visited.insert(start_state);

    // 4个方向
    vector<i64> dx = {0, 0, 1, -1};
    vector<i64> dy = {1, -1, 0, 0};
```

```
    while (!q.empty()) {
        auto state = q.front();
        q.pop();

        // 箱子到达目标位置
        if (state.box_x == target_x && state.box_y == target_y) {
            return state.steps;
        }

        // 人尝试向4个方向移动
        for (i64 d = 0; d < 4; d++) {
            i64 nx = state.person_x + dx[d];
            i64 ny = state.person_y + dy[d];

            // 检查新位置是否合法
            if (nx < 0 || nx >= m || ny < 0 || ny >= n || maze[nx][ny] == 0) {
                continue;
            }

            // 新位置
            i64 new_box_x = state.box_x;
            i64 new_box_y = state.box_y;

            // 如果人移动到箱子的位置，需要推动箱子
            if (nx == state.box_x && ny == state.box_y) {
                i64 box_nx = state.box_x + dx[d];
                i64 box_ny = state.box_y + dy[d];

                // 检查箱子新位置是否合法
                if (box_nx < 0 || box_nx >= m || box_ny < 0 || box_ny >= n ||
                    maze[box_nx][box_ny] == 0) {
                    continue;  // 箱子不能移动
                }

                new_box_x = box_nx;
                new_box_y = box_ny;
            }

            // 创建新状态
            PushBoxState new_state = {nx, ny, new_box_x, new_box_y, state.steps +
1};

            // 如果状态未访问过
            if (visited.find(new_state) == visited.end()) {
                visited.insert(new_state);
                q.push(new_state);
            }
        }
    }

    return -1;  // 不可达
}


/*************************************************************************
```

```
 * 9. 网格DFS通用模板
 * 问题：深度优先搜索解决网格问题
 * 解法：递归探索所有路径，可以结合记忆化和剪枝
 * 时间复杂度：取决于具体问题，空间复杂度：O(n×m)
 ******************************************************************************/
class GridDFS {
private:
    vector<vector<i64>> grid;
    vector<vector<bool>> visited;
    i64 m, n;
    vector<i64> dx = {0, 0, 1, -1};
    vector<i64> dy = {1, -1, 0, 0};

public:
    GridDFS(vector<vector<i64>>& g) : grid(g) {
        m = grid.size();
        n = grid[0].size();
        visited.assign(m, vector<bool>(n, false));
    }

    // 简单的DFS遍历（连通区域）
    void dfs_simple(i64 x, i64 y) {
        visited[x][y] = true;

        for (i64 d = 0; d < 4; d++) {
            i64 nx = x + dx[d];
            i64 ny = y + dy[d];

            if (nx >= 0 && nx < m && ny >= 0 && ny < n &&
                !visited[nx][ny] && grid[nx][ny] == grid[x][y]) {
                dfs_simple(nx, ny);
            }
        }
    }

    // 统计连通区域数量
    i64 count_components() {
        i64 count = 0;

        for (i64 i = 0; i < m; i++) {
            for (i64 j = 0; j < n; j++) {
                if (!visited[i][j]) {
                    count++;
                    dfs_simple(i, j);
                }
            }
        }

        return count;
    }

    // 寻找所有路径（回溯法）
    void find_all_paths(i64 start_x, i64 start_y, i64 target_x, i64 target_y,
                        vector<pair<i64, i64>>& current_path,
                        vector<vector<pair<i64, i64>>>& all_paths) {
        // 标记访问
```

```
        visited[start_x][start_y] = true;
        current_path.push_back({start_x, start_y});

        // 到达终点
        if (start_x == target_x && start_y == target_y) {
            all_paths.push_back(current_path);
        } else {
            // 尝试4个方向
            for (i64 d = 0; d < 4; d++) {
                i64 nx = start_x + dx[d];
                i64 ny = start_y + dy[d];

                if (nx >= 0 && nx < m && ny >= 0 && ny < n &&
                    !visited[nx][ny] && grid[nx][ny] == 1) {
                    find_all_paths(nx, ny, target_x, target_y, current_path,
all_paths);
                }
            }
        }

        // 回溯
        visited[start_x][start_y] = false;
        current_path.pop_back();
    }

    // 获取所有从起点到终点的路径
    vector<vector<pair<i64, i64>>> get_all_paths(i64 start_x, i64 start_y,
                                                 i64 target_x, i64 target_y) {
        // 重置访问数组
        visited.assign(m, vector<bool>(n, false));

        vector<vector<pair<i64, i64>>> all_paths;
        vector<pair<i64, i64>> current_path;

        find_all_paths(start_x, start_y, target_x, target_y, current_path,
all_paths);

        return all_paths;
    }
};


/*******************************************************************************
 * 10. 网格搜索工具类
 * 提供常用的BFS/DFS辅助函数
 ******************************************************************************/
class GridSearchTools {
public:
    // 方向数组生成
    static vector<pair<i64, i64>> get_directions_4() {
        return {{0, 1}, {0, -1}, {1, 0}, {-1, 0}};
    }

    static vector<pair<i64, i64>> get_directions_8() {
        return {{0, 1}, {0, -1}, {1, 0}, {-1, 0},
                {1, 1}, {1, -1}, {-1, 1}, {-1, -1}};
```

```
    }

    static vector<pair<i64, i64>> get_horse_moves() {
        return {{-2, -1}, {-2, 1}, {-1, -2}, {-1, 2},
                {1, -2}, {1, 2}, {2, -1}, {2, 1}};
    }

    // 检查位置是否在网格内
    static bool is_valid_position(i64 x, i64 y, i64 m, i64 n) {
        return x >= 0 && x < m && y >= 0 && y < n;
    }

    // 打印路径（调试用）
    static void print_path(const vector<pair<i64, i64>>& path) {
        cout << "路径 (" << path.size() << "步): ";
        for (size_t i = 0; i < path.size(); i++) {
            if (i > 0) cout << " -> ";
            cout << "(" << path[i].first << "," << path[i].second << ")";
        }
        cout << "\n";
    }

    // 打印网格状态
    static void print_grid(const vector<vector<i64>>& grid) {
        cout << "网格 (" << grid.size() << "x"
             << (grid.empty() ? 0 : grid[0].size()) << "):\n";
        for (const auto& row : grid) {
            cout << "  ";
            for (i64 val : row) {
                cout << (val == 0 ? "" : "·") << " ";
            }
            cout << "\n";
        }
    }

    // 生成随机迷宫
    static vector<vector<i64>> generate_random_maze(i64 m, i64 n, double
obstacle_ratio = 0.3) {
        vector<vector<i64>> maze(m, vector<i64>(n, 1));
        srand(time(0));

        for (i64 i = 0; i < m; i++) {
            for (i64 j = 0; j < n; j++) {
                if ((double)rand() / RAND_MAX < obstacle_ratio) {
                    maze[i][j] = 0;  // 障碍物
                }
            }
        }

        // 确保起点和终点是通的
        maze[0][0] = 1;
        maze[m-1][n-1] = 1;

        return maze;
    }
};
```

搜索算法类型总结

算法类型	数据结构	适用场景	特点
BFS	队列	最短路径、最少步数	层次遍历，最先找到最短解
DFS	栈/递归	所有路径、连通性	深度优先，可能找到非最短解
记忆化DFS	递归+缓存	滑雪、背包等优化问题	避免重复计算，提高效率
状态BFS	队列+状态集	推箱子等复杂状态问题	处理多维状态空间

方向数组配置

移动类型	方向数量	dx数组	dy数组	适用问题
四方向	4	{0,0,1,-1}	{1,-1,0,0}	基本迷宫、连通块
八方向	8	{0,0,1,-1,1,1,-1,-1}	{1,-1,0,0,1,-1,1,-1}	包含对角线的移动
马走日	8	{-2,-2,-1,-1,1,1,2,2}	{-1,1,-2,2,-2,2,-1,1}	象棋中的马
自定义	N	根据问题定义	根据问题定义	特殊移动规则

BFS与DFS对比

特性	BFS	DFS
数据结构	队列	栈/递归
空间复杂度	O(宽)	O(深)
找到的解	最短解（无权）	不一定最短
适用场景	最短路径、最少步数	所有路径、连通性
实现难度	中等	简单

记忆化搜索应用

问题类型	状态定义	转移方程	复杂度
滑雪问题	dp[x][y]: 从(x,y)出发的最长路径	dp[x][y]=1+max(dp[nx][ny])	O(m×n)
01背包	memo[i][j]: 前i个物品容量j的最大价值	memo[i][j]=max(不选, 选)	O(n×C)
完全背包	memo[i][j]: 前i个物品容量j的最大价值	memo[i][j]=max(不选, 选+递归)	O(n×C)

常见问题模式

问题模式	推荐算法	关键点	例题
最短路径	BFS	层次遍历，记录步数	马的遍历
连通块计数	BFS/DFS	遍历未访问点，标记整个区域	细胞计数
最长递减路径	DFS+记忆化	记忆从每个点出发的最长路径	滑雪问题
可达性检查	BFS	检查是否能从起点到终点	迷宫连通
推箱子	状态BFS	(人位置,箱子位置)作为状态	推箱子
传送迷宫	BFS+特殊处理	同时处理普通移动和传送	传送迷宫

📊 关联题目索引

题目编号	题目名称	核心算法	难度
B3625	能否通过迷宫	BFS连通性	⭐
P1443	马的遍历	BFS最短路径	⭐⭐
P1825	传送迷宫	BFS+传送门	⭐⭐⭐
LS1217	推箱子1	状态BFS	⭐⭐⭐
LS1117	滑雪	DFS记忆化	⭐⭐
LS1082	01背包	递归记忆化	⭐⭐
LS1083	完全背包	递归记忆化	⭐⭐

【核心算法详解】**

1. BFS通用模板（`bfs_min_steps`）

```
queue<pair<i64, i64>> q;
dist[start_x][start_y] = 0;
q.push({start_x, start_y});

while (!q.empty()) {
    auto [x, y] = q.front(); q.pop();

    // 检查是否到达目标
    if (x == target_x && y == target_y) return dist[x][y];

    // 遍历所有方向
    for (每个方向) {
        i64 nx = x + dx[i];
        i64 ny = y + dy[i];

        // 检查合法性
        if (合法 && 未访问) {
            dist[nx][ny] = dist[x][y] + 1;
            q.push({nx, ny});
        }
    }
```

```
}
```

2. DFS记忆化模板（`dfs_ski`）

```
i64 dfs(i64 x, i64 y) {
    if (memo[x][y] != -1) return memo[x][y];  // 记忆化

    i64 max_len = 1;
    for (每个方向) {
        i64 nx = x + dx[d];
        i64 ny = y + dy[d];

        if (合法 && 高度更低) {
            max_len = max(max_len, dfs(nx, ny) + 1);
        }
    }

    memo[x][y] = max_len;  // 存储结果
    return max_len;
}
```

3. 状态BFS模板（`push_box_min_steps`）

```
struct State { /* 定义状态 */ };
queue<State> q;
unordered_set<State> visited;

q.push(初始状态);
visited.insert(初始状态);

while (!q.empty()) {
    State current = q.front(); q.pop();

    if (到达目标) return current.steps;

    for (每个可能的动作) {
        State next = 应用动作(current);
        if (!visited.count(next)) {
            visited.insert(next);
            q.push(next);
        }
    }
}
```

【测试用例设计】
代码包含多种测试用例：

1. **基础测试**：验证基本功能
2. **边界测试**：单点、不可达、边界情况
3. **性能测试**：100×100网格
4. **交互测试**：用户自定义输入

【扩展与优化】

1. 空间优化

- BFS：使用dist数组同时记录访问和距离
- DFS：递归栈深度可能较大，注意栈溢出
- 状态压缩：使用位运算压缩状态

2. 时间优化

- 双向BFS：从起点和终点同时搜索
- A*搜索：使用启发式函数指导搜索方向
- 剪枝：DFS中提前终止不可能的解

3. 功能扩展

- 多目标搜索：找到所有目标的最短路径
- 带权网格：使用Dijkstra算法
- 动态网格：网格状态随时间变化

【学习建议】

1. **理解差异**：掌握BFS和DFS的核心区别和适用场景
2. **动手实现**：亲自实现每种搜索算法
3. **状态设计**：学习如何设计复杂问题的状态表示
4. **调试技巧**：使用打印路径、状态等方法调试

【常见错误】

1. **忘记标记访问**：导致无限循环或重复访问
2. **数组越界**：方向移动时未检查边界
3. **状态哈希冲突**：自定义状态哈希函数不当
4. **递归深度过大**：DFS递归过深导致栈溢出
5. **初始化错误**：距离数组或访问数组初始化不当

【实用技巧】

1. **可视化调试**：打印网格和路径帮助理解
2. **小数据测试**：手动计算小数据验证算法
3. **对拍测试**：与简单算法对比结果
4. **方向数组**：使用方向数组简化代码
5. **状态压缩**：复杂状态使用位运算或编码压缩

二、动态规划模块

2.1 背包问题全家桶（课程10）

📦 **核心代码模板**

```
// ==================== 背包问题全家桶 ====================

/**********************************************************************
```

```
 * 1. 01背包模板 - 迭代版本 (LS1082)
 * 问题：有n个物品，每个物品只能选一次，在容量限制下最大化价值
 * 解法：动态规划，dp[j]表示容量为j时的最大价值
 * 时间复杂度：O(n×capacity)，空间复杂度：O(capacity)
 ********************************************************************************/
i64 knapsack_01_iterative(i64 n, i64 capacity,
                          vector<i64>& weights, vector<i64>& values) {
    if (n == 0 || capacity == 0) return 0;

    // dp[j]: 容量为j时的最大价值
    vector<i64> dp(capacity + 1, 0);

    for (i64 i = 0; i < n; i++) {
        // 关键：逆序遍历容量，确保每个物品只选一次
        // 从大到小遍历，避免重复选择同一物品
        for (i64 j = capacity; j >= weights[i]; j--) {
            // 状态转移：不选物品i vs 选物品i
            // dp[j] = max(不选i, 选i后剩余容量的最大价值 + i的价值)
            dp[j] = max(dp[j], dp[j - weights[i]] + values[i]);
        }
    }

    return dp[capacity];
}

// 01背包求具体方案（哪些物品被选中）
pair<i64, vector<i64>> knapsack_01_with_solution(i64 n, i64 capacity,
                                                 vector<i64>& weights,
                                                 vector<i64>& values) {
    vector<vector<i64>> dp(n + 1, vector<i64>(capacity + 1, 0));
    vector<vector<bool>> selected(n + 1, vector<bool>(capacity + 1, false));

    // 构建DP表
    for (i64 i = 1; i <= n; i++) {
        for (i64 j = 0; j <= capacity; j++) {
            // 默认不选当前物品
            dp[i][j] = dp[i - 1][j];

            // 如果可以选当前物品且更优
            if (j >= weights[i - 1] &&
                dp[i - 1][j - weights[i - 1]] + values[i - 1] > dp[i][j]) {
                dp[i][j] = dp[i - 1][j - weights[i - 1]] + values[i - 1];
                selected[i][j] = true;
            }
        }
    }

    // 回溯构造方案
    vector<i64> solution;
    i64 remaining_capacity = capacity;

    for (i64 i = n; i >= 1; i--) {
        if (selected[i][remaining_capacity]) {
            solution.push_back(i - 1);  // 记录物品索引（0-based）
            remaining_capacity -= weights[i - 1];
        }
```

```
    }

    reverse(solution.begin(), solution.end());
    return {dp[n][capacity], solution};
}


/**********************************************************************
 * 2. 01背包恰好装满变体 (LS1232)
 * 问题：必须恰好装满背包，求最大价值，无法装满时返回-1
 * 解法：初始化dp[0]=0，其他为负无穷（表示无法达到）
 * 时间复杂度：O(n×capacity)，空间复杂度：O(capacity)
 **********************************************************************/
i64 knapsack_01_exact(i64 n, i64 capacity,
                      vector<i64>& weights, vector<i64>& values) {
    if (n == 0) return capacity == 0 ? 0 : -1;

    // 初始化：只有容量0可以达到（价值为0），其他容量初始为负无穷
    vector<i64> dp(capacity + 1, LLONG_MIN);
    dp[0] = 0;

    for (i64 i = 0; i < n; i++) {
        for (i64 j = capacity; j >= weights[i]; j--) {
            // 只有前一个状态可达时才能转移
            if (dp[j - weights[i]] != LLONG_MIN) {
                dp[j] = max(dp[j], dp[j - weights[i]] + values[i]);
            }
        }
    }

    return dp[capacity] == LLONG_MIN ? -1 : dp[capacity];
}

// 01背包恰好装满的最小物品数量
i64 knapsack_01_min_items(i64 n, i64 capacity, vector<i64>& weights) {
    // dp[j]: 恰好装满容量j所需的最小物品数量
    vector<i64> dp(capacity + 1, LLONG_MAX);
    dp[0] = 0;  // 容量为0需要0个物品

    for (i64 i = 0; i < n; i++) {
        for (i64 j = capacity; j >= weights[i]; j--) {
            if (dp[j - weights[i]] != LLONG_MAX) {
                dp[j] = min(dp[j], dp[j - weights[i]] + 1);
            }
        }
    }

    return dp[capacity] == LLONG_MAX ? -1 : dp[capacity];
}


/**********************************************************************
 * 3. 完全背包模板 (LS1083)
 * 问题：每个物品可以选择无限次，在容量限制下最大化价值
 * 解法：顺序遍历容量，允许重复选择
 * 时间复杂度：O(n×capacity)，空间复杂度：O(capacity)
```

```
 ********************************************************************************/
i64 knapsack_complete_iterative(i64 n, i64 capacity,
                                vector<i64>& weights, vector<i64>& values) {
    if (n == 0 || capacity == 0) return 0;

    vector<i64> dp(capacity + 1, 0);

    for (i64 i = 0; i < n; i++) {
        // 关键：顺序遍历容量，允许重复选择同一物品
        // 从小到大遍历，允许同一物品被多次选择
        for (i64 j = weights[i]; j <= capacity; j++) {
            dp[j] = max(dp[j], dp[j - weights[i]] + values[i]);
        }
    }

    return dp[capacity];
}

// 完全背包求具体选择方案（每种物品选了多少个）
pair<i64, vector<i64>> knapsack_complete_with_counts(i64 n, i64 capacity,
                                                     vector<i64>& weights,
                                                     vector<i64>& values) {
    vector<i64> dp(capacity + 1, 0);
    vector<i64> last_selected(capacity + 1, -1);  // 记录最后选择的物品
    vector<i64> counts(n, 0);

    for (i64 i = 0; i < n; i++) {
        for (i64 j = weights[i]; j <= capacity; j++) {
            i64 new_value = dp[j - weights[i]] + values[i];
            if (new_value > dp[j]) {
                dp[j] = new_value;
                last_selected[j] = i;  // 容量j时最后选择的是物品i
            }
        }
    }

    // 回溯统计每种物品的数量
    i64 remaining_capacity = capacity;
    while (remaining_capacity > 0 && last_selected[remaining_capacity] != -1) {
        i64 item_idx = last_selected[remaining_capacity];
        counts[item_idx]++;
        remaining_capacity -= weights[item_idx];
    }

    return {dp[capacity], counts};
}


/********************************************************************************
 * 4. 多重背包二进制优化 (LS1084)
 * 问题：每个物品有数量限制，求最大价值
 * 解法：将物品按二进制拆分，转化为01背包问题
 * 时间复杂度：O(n×log(count)×capacity)，空间复杂度：O(capacity)
 ********************************************************************************/
i64 knapsack_multiple(i64 n, i64 capacity, vector<i64>& weights,
                      vector<i64>& values, vector<i64>& counts) {
```

```
    if (n == 0 || capacity == 0) return 0;

    vector<i64> dp(capacity + 1, 0);

    for (i64 i = 0; i < n; i++) {
        // 二进制拆分：将counts[i]个物品拆分成1,2,4,...的组合
        i64 remaining = counts[i];
        i64 k = 1;

        while (k <= remaining) {
            i64 weight = k * weights[i];
            i64 value = k * values[i];

            // 将拆分后的组合当作01背包处理
            for (i64 j = capacity; j >= weight; j--) {
                dp[j] = max(dp[j], dp[j - weight] + value);
            }

            remaining -= k;
            k <<= 1;  // k *= 2
        }

        // 处理剩余部分（如果还有剩余）
        if (remaining > 0) {
            i64 weight = remaining * weights[i];
            i64 value = remaining * values[i];

            for (i64 j = capacity; j >= weight; j--) {
                dp[j] = max(dp[j], dp[j - weight] + value);
            }
        }
    }

    return dp[capacity];
}

// 多重背包单调队列优化（更高效的O(n×capacity)解法）
i64 knapsack_multiple_monotone_queue(i64 n, i64 capacity,
                                     vector<i64>& weights,
                                     vector<i64>& values,
                                     vector<i64>& counts) {
    vector<i64> dp(capacity + 1, 0);

    for (i64 i = 0; i < n; i++) {
        if (weights[i] == 0) {
            // 重量为0的物品特殊处理
            for (i64 j = 0; j <= capacity; j++) {
                dp[j] += counts[i] * values[i];
            }
            continue;
        }

        // 按余数分组，每组使用单调队列优化
        for (i64 r = 0; r < weights[i]; r++) {
            deque<pair<i64, i64>> dq;  // 单调递减队列，存储(dp值, 容量)
```

```
            for (i64 j = 0; r + j * weights[i] <= capacity; j++) {
                i64 current_capacity = r + j * weights[i];
                i64 current_value = dp[current_capacity] - j * values[i];

                // 维护单调队列
                while (!dq.empty() && dq.back().first <= current_value) {
                    dq.pop_back();
                }
                dq.push_back({current_value, j});

                // 移除过期元素（超过数量限制）
                while (!dq.empty() && j - dq.front().second > counts[i]) {
                    dq.pop_front();
                }

                // 更新dp值
                if (!dq.empty()) {
                    dp[current_capacity] = dq.front().first + j * values[i];
                }
            }
        }
    }

    return dp[capacity];
}


/**********************************************************************
 * 5. 二维费用背包 (LS1086)
 * 问题：物品有重量和体积两种费用限制
 * 解法：二维DP，两层循环遍历两种费用
 * 时间复杂度：O(n×capacity1×capacity2)，空间复杂度：O(capacity1×capacity2)
 **********************************************************************/
i64 knapsack_2d(i64 n, i64 capacity1, i64 capacity2,
                vector<i64>& weights1, vector<i64>& weights2,
                vector<i64>& values) {
    if (n == 0 || (capacity1 == 0 && capacity2 == 0)) return 0;

    // dp[j][k]: 重量不超过j，体积不超过k时的最大价值
    vector<vector<i64>> dp(capacity1 + 1, vector<i64>(capacity2 + 1, 0));

    for (i64 i = 0; i < n; i++) {
        // 逆序遍历两种费用（01背包）
        for (i64 j = capacity1; j >= weights1[i]; j--) {
            for (i64 k = capacity2; k >= weights2[i]; k--) {
                dp[j][k] = max(dp[j][k],
                               dp[j - weights1[i]][k - weights2[i]] + values[i]);
            }
        }
    }

    return dp[capacity1][capacity2];
}

// 二维费用完全背包（顺序遍历）
i64 knapsack_2d_complete(i64 n, i64 capacity1, i64 capacity2,
```

```
                        vector<i64>& weights1, vector<i64>& weights2,
                        vector<i64>& values) {
    vector<vector<i64>> dp(capacity1 + 1, vector<i64>(capacity2 + 1, 0));

    for (i64 i = 0; i < n; i++) {
        // 顺序遍历两种费用（完全背包）
        for (i64 j = weights1[i]; j <= capacity1; j++) {
            for (i64 k = weights2[i]; k <= capacity2; k++) {
                dp[j][k] = max(dp[j][k],
                               dp[j - weights1[i]][k - weights2[i]] + values[i]);
            }
        }
    }

    return dp[capacity1][capacity2];
}


/**********************************************************************
 * 6. 分组背包 (LS1087)
 * 问题：物品分为若干组，每组只能选一个物品
 * 解法：外层遍历容量，内层遍历组内物品
 * 时间复杂度：O(总物品数×capacity)，空间复杂度：O(capacity)
 **********************************************************************/
i64 knapsack_group(i64 n, i64 capacity,
                   vector<vector<pair<i64, i64>>>& groups) {
    if (groups.empty() || capacity == 0) return 0;

    // dp[j]: 容量为j时的最大价值
    vector<i64> dp(capacity + 1, 0);

    // 遍历每个组
    for (const auto& group : groups) {
        // 关键：容量逆序遍历，确保每组只选一个
        for (i64 j = capacity; j >= 0; j--) {
            // 遍历组内的每个物品
            for (const auto& [weight, value] : group) {
                if (j >= weight) {
                    dp[j] = max(dp[j], dp[j - weight] + value);
                }
            }
        }
    }

    return dp[capacity];
}

// 分组背包求具体方案（每组选了哪个物品）
pair<i64, vector<i64>> knapsack_group_with_selection(i64 capacity,
                                                     vector<vector<pair<i64,
i64>>>& groups) {
    i64 group_count = groups.size();
    vector<i64> dp(capacity + 1, 0);
    vector<i64> selection(capacity + 1, -1);  // 记录每个容量下选择的物品索引

    for (i64 g = 0; g < group_count; g++) {
```

```
        // 保存上一组的状态
        vector<i64> prev_dp = dp;

        // 遍历组内物品
        for (i64 item_idx = 0; item_idx < groups[g].size(); item_idx++) {
            i64 weight = groups[g][item_idx].first;
            i64 value = groups[g][item_idx].second;

            for (i64 j = capacity; j >= weight; j--) {
                i64 new_value = prev_dp[j - weight] + value;
                if (new_value > dp[j]) {
                    dp[j] = new_value;
                    selection[j] = item_idx;  // 记录选择的物品索引
                }
            }
        }
    }

    // 回溯构造方案
    vector<i64> group_selection(group_count, -1);
    i64 remaining_capacity = capacity;

    for (i64 g = group_count - 1; g >= 0; g--) {
        if (selection[remaining_capacity] != -1) {
            group_selection[g] = selection[remaining_capacity];
            // 回溯到上一组的状态
            remaining_capacity -= groups[g][selection[remaining_capacity]].first;
        }
    }

    return {dp[capacity], group_selection};
}


/*****************************************************************************
 * 7. 混合背包 (LS1085) - 01背包+完全背包混合
 * 问题：物品类型不同，有的是01背包，有的是完全背包
 * 解法：根据物品类型选择不同的遍历顺序
 * 时间复杂度：O(n×capacity)，空间复杂度：O(capacity)
 *****************************************************************************/
i64 knapsack_mixed(i64 n, i64 capacity,
                   vector<i64>& weights, vector<i64>& values,
                   vector<i64>& types) {
    if (n == 0 || capacity == 0) return 0;

    vector<i64> dp(capacity + 1, 0);

    for (i64 i = 0; i < n; i++) {
        if (types[i] == 0) {  // 01背包
            for (i64 j = capacity; j >= weights[i]; j--) {
                dp[j] = max(dp[j], dp[j - weights[i]] + values[i]);
            }
        } else if (types[i] == 1) {  // 完全背包
            for (i64 j = weights[i]; j <= capacity; j++) {
                dp[j] = max(dp[j], dp[j - weights[i]] + values[i]);
            }
```

```
        } else {  // 多重背包（通过二进制拆分）
            i64 count = types[i];  // 假设types[i]表示数量
            i64 remaining = count;
            i64 k = 1;

            while (k <= remaining) {
                i64 weight = k * weights[i];
                i64 value = k * values[i];

                for (i64 j = capacity; j >= weight; j--) {
                    dp[j] = max(dp[j], dp[j - weight] + value);
                }

                remaining -= k;
                k <<= 1;
            }

            if (remaining > 0) {
                i64 weight = remaining * weights[i];
                i64 value = remaining * values[i];

                for (i64 j = capacity; j >= weight; j--) {
                    dp[j] = max(dp[j], dp[j - weight] + value);
                }
            }
        }
    }

    return dp[capacity];
}


/**************************************************************************
 * 8. 依赖背包（树形背包）(LS1088)
 * 问题：物品之间有依赖关系，形成树形结构
 * 解法：树形DP+分组背包
 * 时间复杂度：O(n×capacity²)，空间复杂度：O(n×capacity)
 **************************************************************************/
struct TreeNode {
    i64 weight;
    i64 value;
    vector<i64> children;
};

i64 knapsack_dependent(i64 n, i64 capacity,
                       vector<TreeNode>& nodes, i64 root) {
    // dp[u][j]: 以u为根的子树，容量为j时的最大价值
    vector<vector<i64>> dp(n, vector<i64>(capacity + 1, 0));

    // 后序遍历计算DP值
    function<void(i64)> dfs = [&](i64 u) {
        // 初始化：必须选择根节点u
        for (i64 j = nodes[u].weight; j <= capacity; j++) {
            dp[u][j] = nodes[u].value;
        }
```

```cpp
        // 对每个子节点应用分组背包
        for (i64 v : nodes[u].children) {
            dfs(v);  // 先处理子节点

            // 分组背包：u已经占用了nodes[u].weight
            for (i64 j = capacity; j >= nodes[u].weight; j--) {
                // 分配给子节点v的容量
                for (i64 k = 0; k <= j - nodes[u].weight; k++) {
                    dp[u][j] = max(dp[u][j],
                                   dp[u][j - k] + dp[v][k]);
                }
            }
        }
    };

    dfs(root);
    return dp[root][capacity];
}


/**********************************************************************
 * 9. 背包问题工具类
 * 提供常用的背包问题辅助函数
 **********************************************************************/
class KnapsackTools {
public:
    // 检查输入是否有效
    static bool validate_input(i64 n, i64 capacity,
                               vector<i64>& weights, vector<i64>& values) {
        if (n < 0 || capacity < 0) return false;
        if (weights.size() != n || values.size() != n) return false;

        for (i64 i = 0; i < n; i++) {
            if (weights[i] < 0 || values[i] < 0) return false;
        }

        return true;
    }

    // 归一化重量和价值（避免溢出）
    static void normalize_weights_values(vector<i64>& weights,
                                         vector<i64>& values, i64 factor = 1000) {
        i64 n = weights.size();
        for (i64 i = 0; i < n; i++) {
            weights[i] = (weights[i] + factor - 1) / factor;
            values[i] = (values[i] + factor - 1) / factor;
        }
    }

    // 按单位价值排序（用于贪心近似）
    static vector<i64> sort_by_unit_value(vector<i64>& weights,
                                          vector<i64>& values) {
        i64 n = weights.size();
        vector<i64> indices(n);
        iota(indices.begin(), indices.end(), 0);
```

```
        sort(indices.begin(), indices.end(),
             [&](i64 a, i64 b) {
                 double unit_a = (double)values[a] / weights[a];
                 double unit_b = (double)values[b] / weights[b];
                 return unit_a > unit_b;  // 降序排列
             });

        return indices;
    }

    // 打印DP表（调试用）
    static void print_dp_table(const vector<i64>& dp,
                               const string& name = "DP表") {
        cout << name << ": [";
        for (size_t i = 0; i < dp.size(); i++) {
            if (i > 0) cout << ", ";
            if (dp[i] == LLONG_MIN) {
                cout << "-∞";
            } else if (dp[i] == LLONG_MAX) {
                cout << "∞";
            } else {
                cout << dp[i];
            }
        }
        cout << "]" << "\n";
    }

    // 计算最优值上界（用于剪枝）
    static i64 calculate_upper_bound(i64 remaining_capacity,
                                     const vector<i64>& weights,
                                     const vector<i64>& values,
                                     const vector<i64>& sorted_indices,
                                     i64 current_value,
                                     i64 current_idx) {
        i64 bound = current_value;
        i64 remaining = remaining_capacity;

        // 贪心计算剩余容量的最大可能价值
        for (i64 i = current_idx; i < weights.size(); i++) {
            i64 idx = sorted_indices[i];
            if (weights[idx] <= remaining) {
                bound += values[idx];
                remaining -= weights[idx];
            } else {
                // 部分装入最后一个物品
                bound += (values[idx] * remaining) / weights[idx];
                break;
            }
        }

        return bound;
    }
};


/*****************************************************************************
```

```
 * 10. 背包计数问题（方案数统计）
 * 问题：计算装满背包的方案数
 * 解法：将max改为累加，注意溢出
 * 时间复杂度：O(n×capacity)，空间复杂度：O(capacity)
 ********************************************************************/
i64 knapsack_count_ways(i64 n, i64 capacity, vector<i64>& weights) {
    const i64 MOD = 1e9 + 7;

    // dp[j]: 恰好装满容量j的方案数
    vector<i64> dp(capacity + 1, 0);
    dp[0] = 1;  // 容量为0有1种方案（什么都不选）

    for (i64 i = 0; i < n; i++) {
        for (i64 j = capacity; j >= weights[i]; j--) {
            dp[j] = (dp[j] + dp[j - weights[i]]) % MOD;
        }
    }

    return dp[capacity];
}

// 01背包最大价值对应的方案数
pair<i64, i64> knapsack_01_max_value_and_count(i64 n, i64 capacity,
                                               vector<i64>& weights,
                                               vector<i64>& values) {
    const i64 MOD = 1e9 + 7;

    // dp_value[j]: 容量j的最大价值
    // dp_count[j]: 达到最大价值的方案数
    vector<i64> dp_value(capacity + 1, 0);
    vector<i64> dp_count(capacity + 1, 1);  // 初始方案数为1（什么都不选）

    for (i64 i = 0; i < n; i++) {
        for (i64 j = capacity; j >= weights[i]; j--) {
            i64 new_value = dp_value[j - weights[i]] + values[i];

            if (new_value > dp_value[j]) {
                // 发现更优解，更新价值和方案数
                dp_value[j] = new_value;
                dp_count[j] = dp_count[j - weights[i]];
            } else if (new_value == dp_value[j]) {
                // 价值相等，累加方案数
                dp_count[j] = (dp_count[j] + dp_count[j - weights[i]]) % MOD;
            }
        }
    }

    return {dp_value[capacity], dp_count[capacity]};
}
```

背包问题类型总结

问题类型	特征	核心代码	遍历顺序	时间复杂度
01背包	每个物品最多选一次	`dp[j] = max(dp[j], dp[j-w]+v)`	容量逆序	O(n×C)
完全背包	物品可选无限次	`dp[j] = max(dp[j], dp[j-w]+v)`	容量顺序	O(n×C)
多重背包	物品有数量限制	二进制拆分后按01背包	容量逆序	O(n×log(k)×C)
分组背包	每组只能选一个	外层容量，内层组内物品	容量逆序	O(总物品数×C)
二维费用	两种费用限制	`dp[j][k] = max(...)`	双重逆序	O(n×C_1×C_2)
混合背包	多种类型混合	根据类型选择遍历顺序	混合	O(n×C)

📊 遍历顺序对比

背包类型	物品循环	容量循环	顺序原理
01背包	外层物品	内层容量逆序	防止同一物品被重复选择
完全背包	外层物品	内层容量顺序	允许同一物品被多次选择
多重背包	外层物品+拆分	内层容量逆序	拆分后变为01背包
分组背包	外层组，内层物品	外层容量逆序	每组只能选一个

📊 初始化策略

问题要求	dp[0]初始化	其他位置初始化	目的
最大价值	0	0	计算最大可能价值
恰好装满	0	-∞（负无穷）	确保只有恰好装满才有效
方案计数	1	0	空背包是一种方案
最小物品数	0	∞（正无穷）	求最小值

📊 优化技巧对比

优化技术	适用场景	效果	实现复杂度
二进制拆分	多重背包	O(k)→O(logk)	中等
单调队列	多重背包	O(n×C)	困难
滚动数组	所有背包	空间降维	简单

优化技术	适用场景	效果	实现复杂度
贪心剪枝	搜索解法	减少搜索空间	中等

📊 常见问题变种

基础问题	变种	解法差异	复杂度变化
最大价值	恰好装满	初始化不同	不变
最大价值	方案计数	状态转移为累加	不变
最大价值	具体方案	记录选择路径	空间增加
01背包	分组背包	增加组约束	不变
01背包	依赖背包	树形DP	$O(n \times C^2)$

📊 关联题目索引

题目编号	题目名称	核心算法	难度
LS1082	01背包	基础模板	⭐
LS1232	01背包恰好装满	初始化技巧	⭐⭐
LS1083	完全背包	顺序遍历	⭐⭐
LS1084	多重背包	二进制拆分	⭐⭐⭐
LS1086	二维费用背包	二维DP	⭐⭐⭐
LS1087	分组背包	组内约束	⭐⭐⭐
LS1085	混合背包	类型混合	⭐⭐⭐⭐
LS1088	依赖背包	树形DP	⭐⭐⭐⭐

【核心算法详解】

1. 01背包核心（`knapsack_01_iterative`）

```
// 逆序遍历容量是关键！
for (i = 0; i < n; i++) {
    for (j = capacity; j >= weights[i]; j--) {
        dp[j] = max(dp[j], dp[j - weights[i]] + values[i]);
    }
}
```

2. 完全背包核心（`knapsack_complete_iterative`）

```
// 顺序遍历容量是关键！
for (i = 0; i < n; i++) {
    for (j = weights[i]; j <= capacity; j++) {
        dp[j] = max(dp[j], dp[j - weights[i]] + values[i]);
    }
}
```

3. 多重背包二进制拆分（`knapsack_multiple`）

```
remaining = counts[i];
k = 1;
while (k <= remaining) {
    // 拆分为k个物品的组合
    weight = k * weights[i];
    value = k * values[i];

    // 按01背包处理
    for (j = capacity; j >= weight; j--) {
        dp[j] = max(dp[j], dp[j - weight] + value);
    }

    remaining -= k;
    k <<= 1;  // k *= 2
}
```

【测试用例设计】
代码包含多种测试用例：

1. **基础测试**：验证基本功能
2. **边界测试**：空背包、单个物品、零容量等
3. **性能测试**：1000物品，10000容量
4. **交互测试**：用户自定义输入

【扩展与优化】

1. 空间优化

- 所有背包问题都使用一维DP数组
- 滚动数组技巧（某些问题需要二维时）
- 位压缩存储状态

2. 时间优化

- 二进制拆分：多重背包指数级加速
- 单调队列：多重背包线性优化
- 剪枝策略：搜索时使用贪心上界

3. 功能扩展

- 求具体方案：记录选择路径
- 方案计数：统计最优方案数量
- 多种约束：增加体积、时间等维度

【学习建议】

1. **掌握核心差异**：理解01背包和完全背包遍历顺序的区别
2. **从简到繁**：先掌握01背包，再学其他变种
3. **动手实现**：亲自实现每种背包的代码
4. **分析复杂度**：理解各种优化技术的原理

【常见错误】

1. **遍历顺序混淆**：01背包用顺序或完全背包用逆序
2. **初始化错误**：恰好装满问题初始化不当
3. **数组越界**：容量循环边界条件错误
4. **整数溢出**：价值和容量很大时忘记用long long
5. **多重背包超时**：未使用二进制优化直接三重循环

【实用技巧】

1. **打印DP表**：调试时观察状态转移
2. **小数据验证**：手动计算小数据验证算法
3. **对拍测试**：与暴力搜索对比结果
4. **模版化代码**：封装通用背包函数
5. **边界测试**：测试空、单个、零容量等边界情况

2.2 线性DP与状态设计（课程12, 17）

📦 **核心代码模板**

```
// ==================== 线性DP与状态设计 ====================

/*******************************************************************************
 * 1. 最大子数组和 - Kadane算法 (LS1016, LS1250)
 * 问题：在数组中找到和最大的连续子数组
 * 解法：动态规划，dp[i]表示以nums[i]结尾的最大子数组和
 * 时间复杂度：O(n)，空间复杂度：O(1)
 *******************************************************************************/
i64 max_subarray_sum(vector<i64>& nums) {
    if (nums.empty()) return 0;

    i64 current_sum = nums[0];  // 以当前元素结尾的最大和
    i64 max_sum = nums[0];      // 全局最大和

    for (i64 i = 1; i < nums.size(); i++) {
        // 关键决策：要么从当前数重新开始，要么接上前面的和
        // dp[i] = max(nums[i], dp[i-1] + nums[i])
        current_sum = max(nums[i], current_sum + nums[i]);
        max_sum = max(max_sum, current_sum);
    }

    return max_sum;
```

```cpp
}

// 带起始和结束位置的最大子数组和
pair<i64, pair<i64, i64>> max_subarray_sum_with_indices(vector<i64>& nums) {
    if (nums.empty()) return {0, {-1, -1}};

    i64 current_sum = nums[0];
    i64 max_sum = nums[0];
    i64 start = 0, end = 0;
    i64 temp_start = 0;

    for (i64 i = 1; i < nums.size(); i++) {
        if (nums[i] > current_sum + nums[i]) {
            // 从当前位置重新开始
            current_sum = nums[i];
            temp_start = i;
        } else {
            // 接上前面的子数组
            current_sum = current_sum + nums[i];
        }

        // 更新全局最大值
        if (current_sum > max_sum) {
            max_sum = current_sum;
            start = temp_start;
            end = i;
        }
    }

    return {max_sum, {start, end}};
}


/**********************************************************************
 * 2. 最长递增子序列LIS - O(n²) 基础版 (LS1248)
 * 问题：找到最长严格递增子序列的长度
 * 解法：dp[i]表示以nums[i]结尾的LIS长度
 * 时间复杂度：O(n²)，空间复杂度：O(n)
 **********************************************************************/
i64 lis_length_n2(vector<i64>& nums) {
    i64 n = nums.size();
    if (n == 0) return 0;

    vector<i64> dp(n, 1);  // dp[i]: 以nums[i]结尾的LIS长度
    i64 max_len = 1;

    for (i64 i = 1; i < n; i++) {
        for (i64 j = 0; j < i; j++) {
            if (nums[j] < nums[i]) {
                // 如果nums[j] < nums[i]，可以接在后面
                dp[i] = max(dp[i], dp[j] + 1);
            }
        }
        max_len = max(max_len, dp[i]);
    }
```

```
    return max_len;
}

// 获取LIS具体序列
vector<i64> get_lis_sequence_n2(vector<i64>& nums) {
    i64 n = nums.size();
    if (n == 0) return {};

    vector<i64> dp(n, 1);  // dp[i]: 以nums[i]结尾的LIS长度
    vector<i64> prev(n, -1);  // 前驱节点，用于重构序列

    i64 max_len = 1;
    i64 max_idx = 0;

    for (i64 i = 1; i < n; i++) {
        for (i64 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;
        }
    }

    // 重构LIS序列
    vector<i64> lis;
    for (i64 i = max_idx; i != -1; i = prev[i]) {
        lis.push_back(nums[i]);
    }
    reverse(lis.begin(), lis.end());

    return lis;
}


/**************************************************************************
 * 3. 最长递增子序列LIS - O(nlogn) 优化版
 * 问题：找到最长严格递增子序列的长度
 * 解法：贪心+二分查找，tails数组维护LIS的最小末尾值
 * 时间复杂度：O(nlogn)，空间复杂度：O(n)
 **************************************************************************/
i64 lis_length_nlogn(vector<i64>& nums) {
    if (nums.empty()) return 0;

    vector<i64> tails;  // tails[k]: 长度为k+1的LIS的最小末尾值

    for (i64 num : nums) {
        // 二分查找第一个大于等于num的位置
        auto it = lower_bound(tails.begin(), tails.end(), num);

        if (it == tails.end()) {
            // num比所有末尾值都大，可以延长LIS
            tails.push_back(num);
```

```
        } else {
            // 用num替换该位置的末尾值，使其更小
            *it = num;
        }
    }

    return tails.size();
}

// 获取LIS序列（O(nlogn)版本）
vector<i64> get_lis_sequence_nlogn(vector<i64>& nums) {
    i64 n = nums.size();
    if (n == 0) return {};

    vector<i64> tails;
    vector<i64> indices(n);  // 记录每个元素在tails中的位置
    vector<i64> prev(n, -1); // 前驱节点

    for (i64 i = 0; i < n; i++) {
        auto it = lower_bound(tails.begin(), tails.end(), nums[i]);
        i64 pos = it - tails.begin();

        if (it == tails.end()) {
            tails.push_back(nums[i]);
        } else {
            *it = nums[i];
        }

        indices[i] = pos;

        // 记录前驱：找到前一个位置的值
        if (pos > 0) {
            for (i64 j = i - 1; j >= 0; j--) {
                if (indices[j] == pos - 1 && nums[j] < nums[i]) {
                    prev[i] = j;
                    break;
                }
            }
        }
    }

    // 重构LIS序列
    vector<i64> lis;
    i64 curr = -1;
    for (i64 i = 0; i < n; i++) {
        if (indices[i] == tails.size() - 1) {
            curr = i;
            break;
        }
    }

    for (i64 i = curr; i != -1; i = prev[i]) {
        lis.push_back(nums[i]);
    }
    reverse(lis.begin(), lis.end());
```

```
    return lis;
}


/**********************************************************************
 * 4. 最大环形子数组和 (LS1251)
 * 问题：在环形数组中找到和最大的连续子数组
 * 解法：两种情况：1) 不跨越环形边界；2) 跨越环形边界
 * 时间复杂度：O(n)，空间复杂度：O(1)
 **********************************************************************/
i64 max_circular_subarray_sum(vector<i64>& nums) {
    i64 n = nums.size();
    if (n == 0) return 0;

    // 情况1：最大子数组和不跨越环形（正常Kadane算法）
    i64 max_kadane = max_subarray_sum(nums);

    // 情况2：最大子数组和跨越环形 = 总和 - 最小子数组和
    i64 total_sum = 0;
    i64 min_kadane = LLONG_MAX;
    i64 current_min = 0;

    for (i64 num : nums) {
        total_sum += num;
        current_min = min(num, current_min + num);
        min_kadane = min(min_kadane, current_min);
    }

    // 特殊情况：所有数都是负数
    if (max_kadane < 0) {
        return max_kadane;  // 直接返回最大负数
    }

    // 正常情况：取两种情况的最大值
    i64 circular_sum = total_sum - min_kadane;
    return max(max_kadane, circular_sum);
}

// 带详细分析的环形最大子数组和
pair<i64, string> max_circular_subarray_sum_detailed(vector<i64>& nums) {
    i64 n = nums.size();
    if (n == 0) return {0, "空数组"};

    // 计算正常最大子数组和
    auto [max_normal, normal_indices] = max_subarray_sum_with_indices(nums);
    i64 start_normal = normal_indices.first;
    i64 end_normal = normal_indices.second;

    // 计算最小子数组和
    i64 total_sum = accumulate(nums.begin(), nums.end(), 0LL);
    i64 min_sum = LLONG_MAX;
    i64 current_min = 0;
    i64 min_start = 0, min_end = 0;
    i64 temp_min_start = 0;

    for (i64 i = 0; i < n; i++) {
```

```
        if (nums[i] < current_min + nums[i]) {
            current_min = nums[i];
            temp_min_start = i;
        } else {
            current_min = current_min + nums[i];
        }

        if (current_min < min_sum) {
            min_sum = current_min;
            min_start = temp_min_start;
            min_end = i;
        }
    }

    // 计算环形最大和（跨越边界）
    i64 max_circular = total_sum - min_sum;

    if (max_normal < 0) {
        return {max_normal, "所有元素为负数，不跨越环形边界"};
    }

    if (max_circular > max_normal) {
        return {max_circular, "跨越环形边界，最大子数组包含首尾元素"};
    } else {
        return {max_normal, "不跨越环形边界"};
    }
}


/**************************************************************************
 * 5. 最大两段子数组和 (LS1181)
 * 问题：将数组分成两段非空连续子数组，使两段的和最大
 * 解法：前缀最大子数组和 + 后缀最大子数组和
 * 时间复杂度：O(n)，空间复杂度：O(n)
 **************************************************************************/
i64 max_two_subarray_sum(vector<i64>& nums) {
    i64 n = nums.size();
    if (n < 2) return 0;

    // 1. 计算前缀最大子数组和
    vector<i64> prefix_max(n, LLONG_MIN);
    i64 current = 0;
    i64 max_so_far = LLONG_MIN;

    for (i64 i = 0; i < n; i++) {
        current = max(nums[i], current + nums[i]);
        max_so_far = max(max_so_far, current);
        prefix_max[i] = max_so_far;
    }

    // 2. 计算后缀最大子数组和
    vector<i64> suffix_max(n, LLONG_MIN);
    current = 0;
    max_so_far = LLONG_MIN;

    for (i64 i = n - 1; i >= 0; i--) {
```

```
        current = max(nums[i], current + nums[i]);
        max_so_far = max(max_so_far, current);
        suffix_max[i] = max_so_far;
    }

    // 3. 计算最大两段和：枚举分割点
    i64 max_two_sum = LLONG_MIN;
    for (i64 i = 0; i < n - 1; i++) {
        // 分割点在i和i+1之间
        max_two_sum = max(max_two_sum, prefix_max[i] + suffix_max[i + 1]);
    }

    return max_two_sum;
}

// 获取两段子数组的具体位置
tuple<i64, pair<i64, i64>, pair<i64, i64>>
max_two_subarray_sum_with_indices(vector<i64>& nums) {
    i64 n = nums.size();
    if (n < 2) return {0, {-1, -1}, {-1, -1}};

    // 前缀信息
    vector<i64> prefix_max(n, LLONG_MIN);
    vector<i64> prefix_start(n);
    i64 current = 0, max_so_far = LLONG_MIN;
    i64 start = 0, temp_start = 0;

    for (i64 i = 0; i < n; i++) {
        if (nums[i] > current + nums[i]) {
            current = nums[i];
            temp_start = i;
        } else {
            current = current + nums[i];
        }

        if (current > max_so_far) {
            max_so_far = current;
            start = temp_start;
        }
        prefix_max[i] = max_so_far;
        prefix_start[i] = start;
    }

    // 后缀信息
    vector<i64> suffix_max(n, LLONG_MIN);
    vector<i64> suffix_start(n);
    current = 0; max_so_far = LLONG_MIN;
    i64 end = n - 1, temp_end = n - 1;

    for (i64 i = n - 1; i >= 0; i--) {
        if (nums[i] > current + nums[i]) {
            current = nums[i];
            temp_end = i;
        } else {
            current = current + nums[i];
        }
```

```
        if (current > max_so_far) {
            max_so_far = current;
            end = temp_end;
        }
        suffix_max[i] = max_so_far;
        suffix_start[i] = end;
    }

    // 寻找最优分割点
    i64 best_split = -1;
    i64 max_two_sum = LLONG_MIN;

    for (i64 i = 0; i < n - 1; i++) {
        i64 sum = prefix_max[i] + suffix_max[i + 1];
        if (sum > max_two_sum) {
            max_two_sum = sum;
            best_split = i;
        }
    }

    if (best_split == -1) {
        return {0, {-1, -1}, {-1, -1}};
    }

    pair<i64, i64> first_segment = {prefix_start[best_split], best_split};
    pair<i64, i64> second_segment = {suffix_start[best_split + 1], best_split +
1};

    return {max_two_sum, first_segment, second_segment};
}


/**************************************************************************
 * 6. 环状最大两段子数组和 (P1121)
 * 问题：在环形数组中找到两段非空连续子数组，使它们的和最大
 * 解法：两种情况：1) 不跨越环形边界；2) 跨越环形边界
 * 时间复杂度：O(n)，空间复杂度：O(n)
 **************************************************************************/
i64 max_circular_two_subarray_sum(vector<i64>& nums) {
    i64 n = nums.size();
    if (n < 2) return 0;

    // 情况1：两段都不跨越环形边界（普通两段最大和）
    i64 non_circular = max_two_subarray_sum(nums);

    // 情况2：两段跨越环形边界 = 总和 - 最小两段和
    i64 total_sum = accumulate(nums.begin(), nums.end(), 0LL);

    // 求最小两段和：将原数组取反，求最大两段和
    vector<i64> neg_nums = nums;
    for (i64& num : neg_nums) {
        num = -num;
    }
    i64 max_two_sum_neg = max_two_subarray_sum(neg_nums);
```

```
    // 最小两段和 = -(最大两段和取反后的结果)
    i64 min_two_sum = -max_two_sum_neg;

    // 跨越环形的两段和 = 总和 - 最小两段和
    i64 circular_two_sum = total_sum - min_two_sum;

    // 特殊情况：所有数都是负数
    bool all_negative = true;
    for (i64 num : nums) {
        if (num > 0) {
            all_negative = false;
            break;
        }
    }

    if (all_negative) {
        // 找到最大的两个负数（取绝对值最小的两个）
        sort(nums.begin(), nums.end(), greater<i64>());
        return nums[0] + nums[1];
    }

    return max(non_circular, circular_two_sum);
}


/**********************************************************************
 * 7. 最大子矩阵和（二维最大子数组和）
 * 问题：在二维矩阵中找到和最大的子矩阵
 * 解法：压缩为一维，枚举上下边界，对每列求和后用Kadane
 * 时间复杂度：O(m²n)，空间复杂度：O(n)
 **********************************************************************/
i64 max_submatrix_sum(vector<vector<i64>>& matrix) {
    if (matrix.empty() || matrix[0].empty()) return 0;

    i64 m = matrix.size();
    i64 n = matrix[0].size();
    i64 max_sum = LLONG_MIN;

    // 枚举上边界
    for (i64 top = 0; top < m; top++) {
        vector<i64> col_sums(n, 0);  // 压缩为一维数组

        // 枚举下边界
        for (i64 bottom = top; bottom < m; bottom++) {
            // 更新列和
            for (i64 col = 0; col < n; col++) {
                col_sums[col] += matrix[bottom][col];
            }

            // 对压缩后的数组求最大子数组和
            i64 current = col_sums[0];
            i64 max_ending_here = col_sums[0];

            for (i64 i = 1; i < n; i++) {
                current = max(col_sums[i], current + col_sums[i]);
                max_ending_here = max(max_ending_here, current);
```

```
            }

            max_sum = max(max_sum, max_ending_here);
        }
    }

    return max_sum;
}

// 获取最大子矩阵的位置
tuple<i64, pair<i64, i64>, pair<i64, i64>> max_submatrix_sum_with_indices(
    vector<vector<i64>>& matrix) {

    if (matrix.empty() || matrix[0].empty()) {
        return {0, {-1, -1}, {-1, -1}};
    }

    i64 m = matrix.size();
    i64 n = matrix[0].size();
    i64 max_sum = LLONG_MIN;

    i64 best_top = 0, best_bottom = 0, best_left = 0, best_right = 0;

    for (i64 top = 0; top < m; top++) {
        vector<i64> col_sums(n, 0);

        for (i64 bottom = top; bottom < m; bottom++) {
            for (i64 col = 0; col < n; col++) {
                col_sums[col] += matrix[bottom][col];
            }

            // 在压缩数组上找最大子数组及其位置
            i64 current_sum = col_sums[0];
            i64 max_ending_here = col_sums[0];
            i64 temp_start = 0, start = 0, end = 0;

            for (i64 i = 1; i < n; i++) {
                if (col_sums[i] > current_sum + col_sums[i]) {
                    current_sum = col_sums[i];
                    temp_start = i;
                } else {
                    current_sum = current_sum + col_sums[i];
                }

                if (current_sum > max_ending_here) {
                    max_ending_here = current_sum;
                    start = temp_start;
                    end = i;
                }
            }

            if (max_ending_here > max_sum) {
                max_sum = max_ending_here;
                best_top = top;
                best_bottom = bottom;
                best_left = start;
```

```
                best_right = end;
            }
        }
    }

    return {max_sum, {best_top, best_bottom}, {best_left, best_right}};
}


/**************************************************************************
 * 8. 最长公共子序列LCS (经典DP)
 * 问题：找到两个序列的最长公共子序列
 * 解法：dp[i][j]表示text1前i个和text2前j个的LCS长度
 * 时间复杂度：O(mn)，空间复杂度：O(min(m,n))
 **************************************************************************/
i64 longest_common_subsequence(string text1, string text2) {
    i64 m = text1.size(), n = text2.size();

    // 空间优化：使用两行滚动数组
    vector<vector<i64>> dp(2, vector<i64>(n + 1, 0));

    for (i64 i = 1; i <= m; i++) {
        i64 curr = i % 2;
        i64 prev = (i - 1) % 2;

        for (i64 j = 1; j <= n; j++) {
            if (text1[i - 1] == text2[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];
}

// 获取LCS具体序列
string get_lcs_sequence(string text1, string text2) {
    i64 m = text1.size(), n = text2.size();
    vector<vector<i64>> dp(m + 1, vector<i64>(n + 1, 0));

    // 计算DP表
    for (i64 i = 1; i <= m; i++) {
        for (i64 j = 1; j <= n; j++) {
            if (text1[i - 1] == text2[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1] + 1;
            } else {
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
            }
        }
    }

    // 回溯构建LCS
    string lcs;
    i64 i = m, j = n;
```

```
    while (i > 0 && j > 0) {
        if (text1[i - 1] == text2[j - 1]) {
            lcs.push_back(text1[i - 1]);
            i--;
            j--;
        } else if (dp[i - 1][j] > dp[i][j - 1]) {
            i--;
        } else {
            j--;
        }
    }

    reverse(lcs.begin(), lcs.end());
    return lcs;
}


/*****************************************************************************
 * 9. 编辑距离 (Levenshtein距离)
 * 问题：计算将一个字符串转换成另一个字符串所需的最少操作数
 * 解法：dp[i][j]表示word1前i个转成word2前j个的编辑距离
 * 时间复杂度：O(mn)，空间复杂度：O(min(m,n))
 *****************************************************************************/
i64 edit_distance(string word1, string word2) {
    i64 m = word1.size(), n = word2.size();

    // 空间优化：使用两行滚动数组
    vector<vector<i64>> dp(2, vector<i64>(n + 1, 0));

    // 初始化第一行
    for (i64 j = 0; j <= n; j++) {
        dp[0][j] = j;
    }

    for (i64 i = 1; i <= m; i++) {
        i64 curr = i % 2;
        i64 prev = (i - 1) % 2;

        dp[curr][0] = i;  // 初始化第一列

        for (i64 j = 1; j <= n; j++) {
            if (word1[i - 1] == word2[j - 1]) {
                // 字符相同，不需要操作
                dp[curr][j] = dp[prev][j - 1];
            } else {
                // 三种操作取最小：
                // 1. 插入：dp[i][j-1] + 1
                // 2. 删除：dp[i-1][j] + 1
                // 3. 替换：dp[i-1][j-1] + 1
                dp[curr][j] = min({dp[curr][j - 1],     // 插入
                                   dp[prev][j],         // 删除
                                   dp[prev][j - 1]})    // 替换
                              + 1;
            }
        }
```

```
    }

    return dp[m % 2][n];
}


/**************************************************************************
 * 10. 最大乘积子数组
 * 问题：在数组中找到乘积最大的连续子数组
 * 解法：同时维护最大和最小乘积（因为有负数）
 * 时间复杂度：O(n)，空间复杂度：O(1)
 *************************************************************************/
i64 max_product_subarray(vector<i64>& nums) {
    if (nums.empty()) return 0;

    i64 max_prod = nums[0];
    i64 min_prod = nums[0];
    i64 result = nums[0];

    for (i64 i = 1; i < nums.size(); i++) {
        if (nums[i] < 0) {
            // 遇到负数，交换最大和最小乘积
            swap(max_prod, min_prod);
        }

        // 更新最大和最小乘积
        max_prod = max(nums[i], max_prod * nums[i]);
        min_prod = min(nums[i], min_prod * nums[i]);

        // 更新全局最大值
        result = max(result, max_prod);
    }

    return result;
}


/**************************************************************************
 * 11. 股票买卖系列（见课程17，此处为简化版）
 * 问题：一次买卖的最大利润
 * 解法：记录历史最低价，计算当前卖出的最大利润
 *************************************************************************/
i64 max_profit_one_transaction(vector<i64>& prices) {
    if (prices.size() < 2) return 0;

    i64 min_price = LLONG_MAX;
    i64 max_profit = 0;

    for (i64 price : prices) {
        min_price = min(min_price, price);
        max_profit = max(max_profit, price - min_price);
    }

    return max_profit;
}
```

```cpp
/*****************************************************************************
 * 12. DP工具类：状态设计辅助
 * 提供DP问题的通用模板和辅助函数
 *****************************************************************************/
class DPTools {
public:
    // 滚动数组初始化
    template<typename T>
    static vector<vector<T>> init_rolling_array(i64 rows, i64 cols, T init_val) {
        return vector<vector<T>>(2, vector<T>(cols, init_val));
    }

    // 打印DP表（调试用）
    template<typename T>
    static void print_dp_table(const vector<vector<T>>& dp,
                               const string& name = "DP表") {
        cout << name << ":\n";
        for (const auto& row : dp) {
            for (T val : row) {
                cout << setw(4) << val << " ";
            }
            cout << "\n";
        }
    }

    // 计算数组前缀和
    static vector<i64> compute_prefix_sum(const vector<i64>& nums) {
        i64 n = nums.size();
        vector<i64> prefix(n + 1, 0);
        for (i64 i = 0; i < n; i++) {
            prefix[i + 1] = prefix[i] + nums[i];
        }
        return prefix;
    }

    // 获取子数组和（通过前缀和）
    static i64 get_subarray_sum(const vector<i64>& prefix, i64 l, i64 r) {
        return prefix[r + 1] - prefix[l];
    }

    // 判断是否可以使用滚动数组优化
    static bool can_use_rolling_array(i64 m, i64 n) {
        // 当m或n很大时，考虑滚动数组
        return m > 1000 || n > 1000;
    }
};
```

📊 线性DP问题分类

问题类型	状态维度	典型问题	时间复杂度	空间复杂度
最大子数组和	一维	Kadane算法	O(n)	O(1)
最长递增子序列	一维	LIS问题	$O(n^2)$或O(nlogn)	O(n)

问题类型	状态维度	典型问题	时间复杂度	空间复杂度
环形最大和	一维变种	环形数组	O(n)	O(1)
最大两段和	一维分解	分割问题	O(n)	O(n)
最大子矩阵和	二维降一维	子矩阵问题	$O(m^2n)$	O(n)
最长公共子序列	二维	LCS问题	O(mn)	O(min(m,n))
编辑距离	二维	字符串转换	O(mn)	O(min(m,n))

状态设计模式总结

模式	状态定义	转移方程	示例
以i结尾	dp[i]: 以nums[i]结尾的解	dp[i] = f(nums[i], dp[i-1])	最大子数组和
前i个元素	dp[i]: 前i个元素的解	dp[i] = f(dp[i-1], 选择i)	背包问题
二维状态	dp[i][j]: 涉及两个序列	dp[i][j] = f(dp[i-1][j], dp[i][j-1])	LCS问题
环形处理	两种情况考虑	取max(正常，总和-最小)	环形最大和

优化技巧对比

技巧	适用场景	效果	实现复杂度
滚动数组	状态只依赖前一行/列	空间降维	简单
前缀和	频繁查询子数组和	查询O(1)	简单
贪心+二分	LIS等单调性问题	时间O(nlogn)	中等
分治思想	环形、两段等问题	分解为子问题	中等
压缩状态	状态数有限但组合多	减少状态数	复杂

常见问题变种

基础问题	变种	解法差异	复杂度变化
最大子数组和	环形最大和	两种情况取max	O(n)不变
最大子数组和	最大两段和	前后缀分解	O(n)，空间O(n)
最大子数组和	最大k段和	DP[i][j]状态	O(nk)
LIS	最长非递减子序列	修改比较条件	同O(nlogn)
LIS	最长摆动序列	维护两个状态	O(n)

关联题目索引

题目编号	题目名称	核心算法	难度
LS1016	最大子数组和	Kadane算法	⭐
LS1250	最大子数组和变体	Kadane变种	⭐⭐
LS1248	最长递增子序列	LIS DP	⭐⭐
LS1251	环形最大子数组和	环形处理	⭐⭐
LS1181	最大两段子数组和	前后缀分解	⭐⭐⭐
P1121	环状最大两段和	环形两段处理	⭐⭐⭐

【核心算法详解】

1. Kadane算法核心（`max_subarray_sum`）

```
current_sum = nums[0];  // 以当前元素结尾的最大和
max_sum = nums[0];      // 全局最大和

for (i = 1; i < n; i++) {
    // 关键：要么从当前数重新开始，要么接上前面的和
    current_sum = max(nums[i], current_sum + nums[i]);
    max_sum = max(max_sum, current_sum);
}
```

2. LIS贪心+二分（`lis_length_nlogn`）

```
vector<i64> tails;  // tails[k]: 长度为k+1的LIS的最小末尾值

for (num : nums) {
    auto it = lower_bound(tails.begin(), tails.end(), num);
    if (it == tails.end()) {
        tails.push_back(num);  // 可以延长LIS
    } else {
        *it = num;  // 更新该长度的最小末尾值
    }
}
return tails.size();
```

3. 环形最大和处理（`max_circular_subarray_sum`）

```
// 情况1：不跨越环形（普通Kadane）
max_normal = max_subarray_sum(nums);

// 情况2：跨越环形 = 总和 - 最小子数组和
total_sum = sum(nums);
min_sum = 最小子数组和（用Kadane求最小）

// 特殊情况：全负数
if (max_normal < 0) return max_normal;

return max(max_normal, total_sum - min_sum);
```

【测试用例设计】
代码包含多种测试用例：

1. **基础测试**：验证基本功能
2. **边界测试**：空数组、单元素、全负数等
3. **性能测试**：10万数据量，500×500矩阵
4. **交互测试**：用户自定义输入

【扩展与优化】

1. 空间优化

- 滚动数组：LCS、编辑距离等二维DP
- 原地修改：某些问题可原地计算
- 压缩存储：状态数有限时使用位压缩

2. 时间优化

- 二分查找：LIS从$O(n^2)$到$O(n\log n)$
- 前缀和：快速计算子数组和
- 单调性：利用单调栈/队列优化

3. 功能扩展

- 多段分割：最大k段子数组和
- 带权LIS：元素有权重的情况
- 二维扩展：最大子立方体等

【学习建议】

1. **掌握模板**：记住Kadane、LIS、LCS等经典模板
2. **理解状态**：明确$dp[i]$或$dp[i][j]$的含义
3. **练习变种**：从基础问题开始，逐步尝试变种
4. **分析复杂度**：理解时间和空间复杂度来源

【常见错误】

1. **边界初始化**：dp[0]或dp[1]初始化错误
2. **索引处理**：0-indexed与1-indexed混淆
3. **负数处理**：最大乘积等问题忘记处理负数
4. **环形处理**：忘记特殊情况（全负数）
5. **空间优化**：滚动数组更新顺序错误

【实用技巧】

1. **可视化DP表**：打印DP表帮助理解状态转移
2. **手动模拟**：小数据手动计算验证算法
3. **对拍测试**：与暴力解法对比结果
4. **分步调试**：复杂DP问题分步实现和测试
5. **模版化代码**：封装通用DP函数，便于复用

2.3 股票买卖系列（课程17）

📦 核心代码模板

```
// ==================== 股票买卖系列 ====================

/**************************************************************************
 * 1. 买卖股票的最佳时机 I - 一次交易 (LS1176)
 * 问题：只能进行一次交易（买一次，卖一次）
 * 解法：记录历史最低价，计算当前价格卖出能获得的最大利润
 * 时间复杂度：O(n)，空间复杂度：O(1)
 **************************************************************************/
i64 max_profit_one_transaction(vector<i64>& prices) {
    i64 n = prices.size();
    if (n < 2) return 0;         // 少于2天无法交易

    i64 min_price = LLONG_MAX;  // 记录历史最低价
    i64 max_profit = 0;          // 记录最大利润

    for (i64 price : prices) {
        // 更新历史最低价
        min_price = min(min_price, price);
        // 计算当前价格卖出能获得的利润
        max_profit = max(max_profit, price - min_price);
    }
    return max_profit;
}


/**************************************************************************
 * 2. 买卖股票的最佳时机 II - 无限交易 (LS1177)
 * 问题：可以进行无限次交易（买和卖可以多次）
 * 解法：贪心，所有上涨都交易
 * 时间复杂度：O(n)，空间复杂度：O(1)
**************************************************************************/
i64 max_profit_unlimited_transactions(vector<i64>& prices) {
    i64 n = prices.size();
    if (n < 2) return 0;

    i64 profit = 0;
    for (i64 i = 1; i < n; i++) {
        // 如果今天价格比昨天高，就昨天买今天卖
        if (prices[i] > prices[i - 1]) {
            profit += prices[i] - prices[i - 1];
        }
    }
    return profit;
}


/**************************************************************************
 * 3. 买卖股票的最佳时机 III - 最多两次交易 (LS1178)
```

```
 * 问题：最多进行两次交易（买和卖各两次）
 * 解法：前后缀分解，分别计算第一次和第二次交易的最大利润
 * 时间复杂度：O(n)，空间复杂度：O(n)
*****************************************************************************/
i64 max_profit_two_transactions(vector<i64>& prices) {
    i64 n = prices.size();
    if (n < 2) return 0;

    // 第一次交易：从左到右，记录到第i天的最大利润
    vector<i64> first_profit(n, 0);
    i64 min_price = prices[0];
    for (i64 i = 1; i < n; i++) {
        min_price = min(min_price, prices[i]);
        // 今天卖出或不卖，取最大值
        first_profit[i] = max(first_profit[i - 1], prices[i] - min_price);
    }

    // 第二次交易：从右到左，记录从第i天开始的最大利润
    vector<i64> second_profit(n, 0);
    i64 max_price = prices[n - 1];
    for (i64 i = n - 2; i >= 0; i--) {
        max_price = max(max_price, prices[i]);
        // 今天买入或不买，取最大值
        second_profit[i] = max(second_profit[i + 1], max_price - prices[i]);
    }

    // 合并两次交易：枚举分割点
    i64 max_profit = 0;
    for (i64 i = 0; i < n; i++) {
        max_profit = max(max_profit, first_profit[i] + second_profit[i]);
    }
    return max_profit;
}


/*****************************************************************************
 * 4. 买卖股票的最佳时机 IV - 最多k次交易 (LS1252)
 * 问题：最多进行k次交易
 * 解法：动态规划，dp[i][j][0/1]表示第i天，完成了j次交易，0不持有，1持有
 * 时间复杂度：O(n*k)，空间复杂度：O(n*k) 可优化为O(k)
*****************************************************************************/
i64 max_profit_at_most_k(i64 k, vector<i64>& prices) {
    i64 n = prices.size();
    if (n == 0 || k == 0) return 0;

    // 优化：如果k很大，退化为无限交易问题
    if (k >= n / 2) {
        i64 profit = 0;
        for (i64 i = 1; i < n; i++) {
            if (prices[i] > prices[i - 1]) {
                profit += prices[i] - prices[i - 1];
            }
        }
        return profit;
    }
```

```
    // 三维DP：dp[i][j][0/1]
    // i：天数（0~n），j：交易次数（0~k），0/1：不持有/持有股票
    vector<vector<vector<i64>>> dp(
        n + 1,
        vector<vector<i64>>(k + 1, vector<i64>(2, LLONG_MIN / 2))
    );

    // 初始化：第0天，0次交易，不持有股票
    dp[0][0][0] = 0;

    for (i64 i = 1; i <= n; i++) {
        for (i64 j = 0; j <= k; j++) {
            // 状态转移：
            // 1. 今天不持有股票：
            //    a. 昨天就不持有，今天继续不持有
            //    b. 昨天持有，今天卖出（完成一次交易）
            dp[i][j][0] = dp[i - 1][j][0];
            if (j > 0 && dp[i - 1][j - 1][1] != LLONG_MIN / 2) {
                dp[i][j][0] = max(dp[i][j][0], dp[i - 1][j - 1][1] + prices[i -
1]);
            }

            // 2. 今天持有股票：
            //    a. 昨天就持有，今天继续持有
            //    b. 昨天不持有，今天买入
            dp[i][j][1] = max(dp[i - 1][j][1], dp[i - 1][j][0] - prices[i - 1]);
        }
    }

    // 答案是最后一天，任意次交易，不持有股票的最大值
    i64 max_profit = 0;
    for (i64 j = 0; j <= k; j++) {
        max_profit = max(max_profit, dp[n][j][0]);
    }
    return max_profit;
}


/*****************************************************************************
 * 5. 买卖股票的最佳时机 V - 含冷冻期 (LS1179)
 * 问题：卖出后有一天冷冻期不能买入
 * 解法：状态机DP，三个状态
 * 时间复杂度：O(n)，空间复杂度：O(n) 可优化为O(1)
*****************************************************************************/
i64 max_profit_with_cooldown(vector<i64>& prices) {
    i64 n = prices.size();
    if (n < 2) return 0;

    // 三个状态：
    // 0: 持有股票 (持有)
    // 1: 不持有股票，处于冷冻期 (冷冻)
    // 2: 不持有股票，不处于冷冻期 (可购买)
    vector<vector<i64>> dp(n, vector<i64>(3, 0));

    // 第0天初始化
    dp[0][0] = -prices[0];  // 第0天买入，持有股票
```

```
    dp[0][1] = 0;            // 第0天不可能处于冷冻期
    dp[0][2] = 0;            // 第0天不操作，不持有也不冷冻

    for (i64 i = 1; i < n; i++) {
        // 状态转移：
        // 1. 今天持有股票：
        //    a. 昨天就持有，今天继续持有
        //    b. 昨天不持有且不冷冻，今天买入
        dp[i][0] = max(dp[i - 1][0], dp[i - 1][2] - prices[i]);

        // 2. 今天不持有且冷冻：
        //    一定是昨天持有，今天卖出
        dp[i][1] = dp[i - 1][0] + prices[i];

        // 3. 今天不持有且不冷冻：
        //    a. 昨天就不持有且冷冻，今天解冻
        //    b. 昨天就不持有且不冷冻，今天继续
        dp[i][2] = max(dp[i - 1][1], dp[i - 1][2]);
    }

    // 最后一天不能持有股票（持有的话可以卖出获得更多利润）
    return max(dp[n - 1][1], dp[n - 1][2]);
}


/**************************************************************************
 * 6. 买卖股票的最佳时机 VI - 含手续费 (LS1253)
 * 问题：每次交易需要支付固定手续费
 * 解法：状态机DP，两个状态，卖出时扣除手续费
 * 时间复杂度：O(n)，空间复杂度：O(n) 可优化为O(1)
**************************************************************************/
i64 max_profit_with_fee(vector<i64>& prices, i64 fee) {
    i64 n = prices.size();
    if (n < 2) return 0;

    // 两个状态：0-不持有，1-持有
    vector<i64> hold(n), empty(n);

    // 第0天初始化
    hold[0] = -prices[0];  // 第0天买入
    empty[0] = 0;          // 第0天不操作

    for (i64 i = 1; i < n; i++) {
        // 今天持有：
        // 1. 昨天就持有，今天继续持有
        // 2. 昨天不持有，今天买入
        hold[i] = max(hold[i - 1], empty[i - 1] - prices[i]);

        // 今天不持有：
        // 1. 昨天就不持有，今天继续不持有
        // 2. 昨天持有，今天卖出（扣除手续费）
        empty[i] = max(empty[i - 1], hold[i - 1] + prices[i] - fee);
    }

    // 最后一天不持有股票
    return empty[n - 1];
```

```cpp
}


/*****************************************************************************
 * 7. 买卖股票的最佳时机 - 通用状态机解法（滚动数组优化）
 * 优化空间复杂度为O(1)
*****************************************************************************/
i64 max_profit_general(vector<i64>& prices, i64 k = 1, i64 fee = 0, bool cooldown
= false) {
    i64 n = prices.size();
    if (n < 2) return 0;

    if (cooldown) {
        // 含冷冻期
        i64 hold = -prices[0], frozen = 0, empty = 0;
        for (i64 i = 1; i < n; i++) {
            i64 new_hold = max(hold, empty - prices[i]);
            i64 new_frozen = hold + prices[i];
            i64 new_empty = max(frozen, empty);

            hold = new_hold;
            frozen = new_frozen;
            empty = new_empty;
        }
        return max(frozen, empty);
    } else if (k == 1) {
        // 一次交易
        i64 min_price = prices[0], max_profit = 0;
        for (i64 price : prices) {
            min_price = min(min_price, price);
            max_profit = max(max_profit, price - min_price);
        }
        return max_profit;
    } else if (k == INT_MAX) {
        // 无限交易，含手续费
        i64 hold = -prices[0], empty = 0;
        for (i64 i = 1; i < n; i++) {
            i64 new_hold = max(hold, empty - prices[i]);
            i64 new_empty = max(empty, hold + prices[i] - fee);
            hold = new_hold;
            empty = new_empty;
        }
        return empty;
    } else {
        // k次交易，使用滚动数组优化
        if (k >= n / 2) {
            // 退化为无限交易
            i64 profit = 0;
            for (i64 i = 1; i < n; i++) {
                if (prices[i] > prices[i - 1]) {
                    profit += prices[i] - prices[i - 1];
                }
            }
            return profit;
        }
```

```
        // 滚动数组：buy[j]表示第j次交易持有股票的最大利润
        //          sell[j]表示第j次交易不持有股票的最大利润
        vector<i64> buy(k + 1, LLONG_MIN / 2);
        vector<i64> sell(k + 1, 0);

        for (i64 i = 0; i < n; i++) {
            for (i64 j = 1; j <= k; j++) {
                // 第j次交易持有：昨天就持有 或 昨天第j-1次交易完成今天买入
                buy[j] = max(buy[j], sell[j - 1] - prices[i]);
                // 第j次交易不持有：昨天就不持有 或 昨天持有今天卖出
                sell[j] = max(sell[j], buy[j] + prices[i]);
            }
        }

        return sell[k];
    }
}
```

📊 股票问题状态设计：

1. **一次交易**：记录历史最低价
2. **无限交易**：贪心，所有上涨都交易
3. **两次交易**：前后缀分解
4. **k次交易**：三维DP
5. **含冷冻期**：三个状态的状态机
6. **含手续费**：卖出时扣除

📊 关联题目：

- LS1176 买卖股票1：一次交易
- LS1177 买卖股票2：无限交易（贪心）
- LS1178 买卖股票3：最多两次交易
- LS1252 买卖股票4：最多k次交易
- LS1179 买卖股票含冷冻期：状态机DP
- LS1253 买卖股票含手续费：状态机DP

【各算法时间复杂度对比】

算法	时间复杂度	空间复杂度	适用场景
一次交易	O(n)	O(1)	只能买卖一次
无限交易	O(n)	O(1)	可以无限次买卖
两次交易	O(n)	O(n)	最多买卖两次
k次交易	O(n×k)	O(k)	最多买卖k次
含冷冻期	O(n)	O(1)	卖出后有一天不能买
含手续费	O(n)	O(1)	每次交易有手续费

【核心算法详解】

【1】一次交易（`max_profit_one_transaction`）

```
// 核心思想：记录历史最低价
i64 min_price = LLONG_MAX;
i64 max_profit = 0;
for (价格 : prices) {
    min_price = min(min_price, 价格);
    max_profit = max(max_profit, 价格 - min_price);
}
```

【2】无限交易（`max_profit_unlimited_transactions`）

```
// 核心思想：贪心，所有上涨都交易
for (i = 1; i < n; i++) {
    if (prices[i] > prices[i-1]) {
        profit += prices[i] - prices[i-1];
    }
}
```

【3】k次交易（`max_profit_at_most_k`）

```
// 状态定义：dp[i][j][0/1]
// i: 天数，j: 交易次数，0: 不持有，1: 持有

// 状态转移：
// 今天不持有 = max(昨天不持有, 昨天持有+今天卖出)
// 今天持有 = max(昨天持有, 昨天不持有-今天买入)
```

【4】含冷冻期（`max_profit_with_cooldown`）

```
// 三个状态的状态机：
// 0: 持有股票
// 1: 不持有，冷冻期
// 2: 不持有，可购买

// 状态转移：
// hold = max(昨天hold, 昨天empty-今天买入)
// frozen = 昨天hold + 今天卖出
// empty = max(昨天frozen, 昨天empty)
```

【5】测试用例设计

代码包含多种测试用例：

1. **基础测试**：验证基本功能

2. **边界测试**：递减序列、递增序列
3. **复杂测试**：波动序列
4. **性能测试**：10万数据量测试
5. **交互测试**：用户自定义输入

【6】扩展与优化

【6.1】空间优化

所有算法都实现了空间优化版本：

- 一次/无限交易：O(1) 空间
- k次交易：O(k) 空间（滚动数组）
- 冷冻期/手续费：O(1) 空间

【6.2】通用解法

`max_profit_general()` 函数支持所有参数：

- `k`：交易次数限制
- `fee`：手续费
- `cooldown`：是否含冷冻期

【6.3】错误处理

- 检查输入合法性
- 防止整数溢出
- 空数组处理

【7】学习建议

1. **从简单到复杂**：先理解一次交易，再学无限交易，最后学k次交易
2. **状态机思维**：将买卖问题转化为状态转移
3. **空间优化技巧**：滚动数组、变量复用
4. **测试驱动**：自己设计测试用例验证理解

【8】常见错误

1. **边界条件**：n<2时返回0
2. **初始化**：DP数组正确初始化
3. **状态转移**：确保所有情况都考虑到
4. **整数溢出**：使用 `LLONG_MIN/2` 作为负无穷

三、数据结构模块

3.1 STL核心用法（课程3）

📦 **核心代码模板**

```
// ==================== STL核心容器与算法 ====================
```

```cpp
/*****************************************************************************
 * 1. vector: 动态数组 (LS1033, LS1038)
 * 特性：连续内存，随机访问O(1)，尾部操作高效
 * 时间复杂度：push_back/pop_back平均O(1)，中间插入删除O(n)
 * 空间复杂度：O(n)，自动扩容
 *****************************************************************************/
void vector_demo() {
    // 初始化
    vector<i64> vec1 = {1, 2, 3, 4, 5};
    vector<i64> vec2(10, 0);           // 10个0
    vector<i64> vec3(vec1.begin(), vec1.end());  // 复制

    // 基本操作
    vec1.push_back(6);                 // 尾部添加 O(1)平均
    vec1.pop_back();                   // 尾部删除 O(1)
    i64 val = vec1[2];                 // 随机访问 O(1)
    i64 front_val = vec1.front();      // 首元素
    i64 back_val = vec1.back();        // 尾元素

    // 容量操作
    i64 size = vec1.size();            // 当前元素个数
    i64 capacity = vec1.capacity();    // 当前容量
    bool empty = vec1.empty();         // 是否为空
    vec1.reserve(100);                 // 预留容量
    vec1.shrink_to_fit();              // 缩减容量

    // 插入删除（中间位置效率低）
    vec1.insert(vec1.begin() + 2, 10);   // 位置2插入10，O(n)
    vec1.erase(vec1.begin() + 1);        // 删除位置1，O(n)
    vec1.erase(vec1.begin() + 1, vec1.begin() + 3);  // 删除区间

    // 清空
    vec1.clear();                       // 清空所有元素

    // 遍历方式
    for (i64 i = 0; i < vec1.size(); i++) {
        // 下标访问
    }
    for (i64 x : vec1) {
        // 范围for
    }
    for (auto it = vec1.begin(); it != vec1.end(); ++it) {
        // 迭代器
        i64 value = *it;
    }

    // 算法
    sort(vec1.begin(), vec1.end());                 // 排序
    reverse(vec1.begin(), vec1.end());              // 反转
    i64 sum = accumulate(vec1.begin(), vec1.end(), 0);  // 求和
    auto max_it = max_element(vec1.begin(), vec1.end()); // 最大值位置
    auto min_it = min_element(vec1.begin(), vec1.end()); // 最小值位置
}


/*****************************************************************************
```

```
 * 2. deque: 双端队列
 * 特性：双向开口，头尾插入删除O(1)，随机访问O(1)
 * 时间复杂度：头尾操作O(1)，中间操作O(n)
 * 空间复杂度：O(n)，分段连续存储
 ************************************************************************/
void deque_demo() {
    deque<i64> dq;

    // 两端操作
    dq.push_back(1);                    // 尾部插入 O(1)
    dq.push_front(2);                   // 头部插入 O(1)
    dq.pop_back();                      // 尾部删除 O(1)
    dq.pop_front();                     // 头部删除 O(1)

    // 随机访问
    dq[0] = 10;                         // 修改首元素
    i64 val = dq.at(1);                 // 带边界检查的访问

    // 容量信息
    i64 size = dq.size();
    bool empty = dq.empty();

    // 清空
    dq.clear();

    // 遍历
    for (i64 x : dq) {
        // 范围for
    }

    // 与vector对比：deque支持高效的push_front
    // 但deque的内存不连续，对缓存不友好
}


/************************************************************************
 * 3. list: 双向链表
 * 特性：双向链表，任意位置插入删除O(1)，不支持随机访问
 * 时间复杂度：插入删除O(1)，查找O(n)
 * 空间复杂度：O(n)，每个节点额外指针开销
 ************************************************************************/
void list_demo() {
    list<i64> lst = {1, 2, 3, 4, 5};

    // 插入删除
    lst.push_back(6);                   // 尾部插入
    lst.push_front(0);                  // 头部插入
    lst.pop_back();                     // 尾部删除
    lst.pop_front();                    // 头部删除

    // 链表特有操作
    lst.insert(next(lst.begin(), 2), 10);  // 位置2插入
    lst.erase(prev(lst.end(), 2));         // 删除倒数第2个

    // 拼接操作（高效）
    list<i64> lst2 = {7, 8, 9};
```

```
    lst.splice(lst.end(), lst2);        // 将lst2拼接到lst末尾

    // 排序（链表专用算法）
    lst.sort();                         // 链表排序

    // 去重（需要先排序）
    lst.unique();                       // 删除连续重复元素

    // 遍历（只有双向迭代器）
    for (auto it = lst.begin(); it != lst.end(); ++it) {
        i64 val = *it;
    }

    // 注意：list不支持[]操作符和随机访问
}


/**************************************************************************
 * 4. set/multiset: 红黑树实现的有序集合 (LS1039)
 * set: 唯一键集合，自动去重排序
 * multiset: 允许重复键的有序集合
 * 时间复杂度：插入删除查找都是O(logn)
 * 空间复杂度：O(n)
 **************************************************************************/
void set_demo() {
    // set: 唯一键
    set<i64> s = {3, 1, 4, 1, 5};       // {1, 3, 4, 5} 自动去重排序

    // 基本操作
    s.insert(2);                        // 插入 O(logn)
    s.erase(3);                         // 删除 O(logn)
    s.erase(s.begin());                 // 删除最小元素

    // 查找操作
    auto it1 = s.find(4);               // 查找，返回迭代器
    if (it1 != s.end()) {
        i64 val = *it1;
    }

    // 统计
    i64 count = s.count(4);             // 统计个数，set返回0或1

    // 边界查找
    auto lower = s.lower_bound(3);      // 第一个≥3的元素
    auto upper = s.upper_bound(3);      // 第一个>3的元素
    auto range = s.equal_range(3);      // 等于3的范围[pair]

    // 遍历（有序）
    for (i64 x : s) {
        // 按升序遍历
    }

    // multiset: 允许重复
    multiset<i64> ms = {1, 1, 2, 2, 3};
    i64 ms_count = ms.count(1);         // 返回2
```

```
    // 自定义比较函数
    struct Point {
        i64 x, y;
        bool operator<(const Point& other) const {
            if (x != other.x) return x < other.x;
            return y < other.y;
        }
    };
    set<Point> point_set;
}


/*****************************************************************************
 * 5. map/multimap: 红黑树实现的有序映射
 * map: 键值对，键唯一
 * multimap: 允许重复键的映射
 * 时间复杂度：插入删除查找都是O(logn)
 * 空间复杂度：O(n)
 *****************************************************************************/
void map_demo() {
    // map: 键值对
    map<string, i64> mp;

    // 插入方式
    mp["apple"] = 5;                    // 下标插入
    mp.insert({"banana", 3});           // insert插入
    mp.emplace("orange", 4);            // 原地构造

    // 访问（注意：operator[]会创建不存在的key）
    i64 apple_count = mp["apple"];      // 存在时访问
    i64 pear_count = mp["pear"];        // 不存在时会创建，值为0

    // 安全访问
    auto it = mp.find("apple");
    if (it != mp.end()) {
        string key = it->first;
        i64 value = it->second;
    }

    // 删除
    mp.erase("banana");                 // 按键删除
    mp.erase(mp.begin());               // 按迭代器删除

    // 遍历
    for (const auto& [key, value] : mp) {  // C++17结构化绑定
        // 按键升序遍历
    }

    // 边界查找
    auto lower = mp.lower_bound("c");  // 第一个键≥"c"的
    auto upper = mp.upper_bound("c");  // 第一个键>"c"的

    // multimap: 允许重复键
    multimap<string, i64> mmp;
    mmp.insert({"apple", 1});
    mmp.insert({"apple", 2});           // 允许重复键
```

```
    // 获取重复键范围
    auto range = mmp.equal_range("apple");
    for (auto it = range.first; it != range.second; ++it) {
        i64 value = it->second;
    }
}


/**************************************************************************
 * 6. unordered_set/unordered_map: 哈希表实现
 * 特性：平均O(1)操作，最坏O(n)，无序存储
 * 时间复杂度：平均O(1)，最坏O(n)
 * 空间复杂度：O(n)
 *************************************************************************/
void unordered_demo() {
    // unordered_set
    unordered_set<i64> us = {3, 1, 4, 1, 5};  // 去重，无序

    us.insert(2);                          // 插入
    us.erase(3);                           // 删除
    bool has4 = us.count(4) > 0;           // 检查存在

    // 遍历（无序）
    for (i64 x : us) {
        // 顺序不确定
    }

    // unordered_map
    unordered_map<string, i64> ump;
    ump["apple"] = 5;
    ump["banana"] = 3;

    // 安全访问
    if (ump.find("apple") != ump.end()) {
        i64 value = ump["apple"];
    }

    // 桶操作
    i64 bucket_count = ump.bucket_count();
    i64 bucket_size = ump.bucket_size(0);
    i64 bucket = ump.bucket("apple");

    // 自定义哈希函数
    struct MyHash {
        size_t operator()(const string& s) const {
            return hash<string>()(s);
        }
    };
    unordered_map<string, i64, MyHash> custom_map;

    // 注意：unordered容器需要为自定义类型提供哈希函数和相等比较
}


/**************************************************************************
```

```
 * 7. priority_queue: 优先队列（堆） (LS1039)
 * 特性：默认大顶堆，顶部元素始终是最大/最小值
 * 时间复杂度：push/pop O(logn)，top O(1)
 * 空间复杂度：O(n)
 ****************************************************************************/
void priority_queue_demo() {
    // 默认大顶堆
    priority_queue<i64> max_heap;
    max_heap.push(3);                    // 插入
    max_heap.push(1);
    max_heap.push(4);

    i64 top_val = max_heap.top();        // 查看最大值 O(1)
    max_heap.pop();                      // 删除最大值 O(logn)

    // 小顶堆
    priority_queue<i64, vector<i64>, greater<i64>> min_heap;
    min_heap.push(3);
    min_heap.push(1);
    min_heap.push(4);
    i64 min_val = min_heap.top();        // 查看最小值

    // 自定义比较函数
    struct Compare {
        bool operator()(const i64& a, const i64& b) {
            return a > b;  // 小顶堆
        }
    };
    priority_queue<i64, vector<i64>, Compare> custom_heap;

    // 存储pair（默认按first比较）
    priority_queue<pair<i64, string>> pq_pair;
    pq_pair.push({3, "apple"});
    pq_pair.push({1, "banana"});
    pq_pair.push({4, "orange"});

    // 注意：priority_queue没有迭代器，不能遍历
}


/****************************************************************************
 * 8. stack: 栈
 * 特性：LIFO（后进先出），适配器容器
 * 时间复杂度：push/pop/top都是O(1)
 * 空间复杂度：O(n)
 ****************************************************************************/
void stack_demo() {
    stack<i64> stk;

    stk.push(1);                         // 入栈
    stk.push(2);
    stk.push(3);

    i64 top_val = stk.top();             // 查看栈顶
    stk.pop();                           // 出栈
```

```
    bool empty = stk.empty();
    i64 size = stk.size();

    // 栈没有迭代器，不能遍历
    // 如果需要遍历，可以用vector模拟栈
}


/*****************************************************************************
 * 9. queue: 队列
 * 特性：FIFO（先进先出），适配器容器
 * 时间复杂度：push/pop/front都是O(1)
 * 空间复杂度：O(n)
 *****************************************************************************/
void queue_demo() {
    queue<i64> q;

    q.push(1);                          // 入队
    q.push(2);
    q.push(3);

    i64 front_val = q.front();          // 队首
    i64 back_val = q.back();            // 队尾
    q.pop();                            // 出队

    bool empty = q.empty();
    i64 size = q.size();

    // 队列没有迭代器，不能遍历
}


/*****************************************************************************
 * 10. string: 字符串 (LT3556)
 * 特性：类似vector<char>，支持丰富的字符串操作
 * 时间复杂度：各种操作
 * 空间复杂度：O(n)
 *****************************************************************************/
void string_demo() {
    string str = "Hello World";

    // 基本操作
    str.push_back('!');                 // 尾部添加字符
    str.pop_back();                     // 尾部删除字符
    char ch = str[0];                   // 随机访问
    i64 len = str.length();             // 长度

    // 子串操作
    string sub = str.substr(6, 5);      // 从6开始取5个字符 "World"
    str.insert(5, " C++");              // 位置5插入
    str.erase(5, 4);                    // 从5开始删除4个字符

    // 查找操作
    i64 pos1 = str.find("World");       // 查找子串，返回位置或string::npos
    i64 pos2 = str.find('W');           // 查找字符
    i64 rpos = str.rfind('l');          // 从后向前查找
```

```
    // 替换操作
    str.replace(6, 5, "C++");          // 替换子串

    // 转换操作
    transform(str.begin(), str.end(), str.begin(), ::tolower);  // 转小写
    transform(str.begin(), str.end(), str.begin(), ::toupper);  // 转大写

    // 数字转换
    string num_str = "123";
    i64 num = stoi(num_str);           // 字符串转整数
    double dnum = stod("3.14");        // 字符串转浮点数

    string num_str2 = to_string(123);  // 数字转字符串

    // 分割字符串
    string text = "apple,banana,orange";
    vector<string> tokens;
    stringstream ss(text);
    string token;
    while (getline(ss, token, ',')) {
        tokens.push_back(token);
    }

    // C字符串转换
    const char* cstr = str.c_str();    // 获取C风格字符串
    str = string(cstr);                // C字符串转string
}


/*****************************************************************************
 * 11. 自定义排序与比较 (LS1212)
 * 多种方式实现自定义排序规则
 *****************************************************************************/
void custom_sort_demo() {
    struct Person {
        string name;
        i64 age;
        i64 score;

        // 方法1: 重载小于运算符
        bool operator<(const Person& other) const {
            if (score != other.score) return score > other.score;  // 分数降序
            if (age != other.age) return age < other.age;          // 年龄升序
            return name < other.name;                              // 名字字典序
        }
    };

    vector<Person> people = {
        {"Alice", 20, 85},
        {"Bob", 22, 90},
        {"Charlie", 21, 88},
        {"David", 20, 90}
    };

    // 方法1: 使用重载的运算符
```

```
    sort(people.begin(), people.end());

    // 方法2：自定义比较函数
    auto cmp_by_age = [](const Person& a, const Person& b) {
        return a.age < b.age;  // 按年龄升序
    };
    sort(people.begin(), people.end(), cmp_by_age);

    // 方法3：lambda表达式（最常用）
    sort(people.begin(), people.end(), [](const Person& a, const Person& b) {
        if (a.score != b.score) return a.score > b.score;
        if (a.age != b.age) return a.age < b.age;
        return a.name < b.name;
    });

    // 方法4：函数对象
    struct CompareByScore {
        bool operator()(const Person& a, const Person& b) const {
            return a.score > b.score;
        }
    };
    sort(people.begin(), people.end(), CompareByScore());

    // 对于set/map等容器，需要提供比较类型
    struct Point {
        i64 x, y;
    };

    struct PointCompare {
        bool operator()(const Point& a, const Point& b) const {
            if (a.x != b.x) return a.x < b.x;
            return a.y < b.y;
        }
    };

    set<Point, PointCompare> point_set;
    map<Point, string, PointCompare> point_map;
}


/*****************************************************************************
 * 12. 去重操作 (LS1034)
 * 多种方式去除重复元素
 *****************************************************************************/
void unique_demo() {
    vector<i64> nums = {3, 1, 2, 2, 3, 1, 4, 5, 5};

    // 方法1：使用set自动去重排序
    set<i64> unique_set(nums.begin(), nums.end());
    vector<i64> unique_vec1(unique_set.begin(), unique_set.end());
    // 结果：{1, 2, 3, 4, 5} 已排序

    // 方法2：排序后使用unique算法
    vector<i64> sorted_nums = nums;
    sort(sorted_nums.begin(), sorted_nums.end());
    auto new_end = unique(sorted_nums.begin(), sorted_nums.end());
```

```
    sorted_nums.erase(new_end, sorted_nums.end());
    // 结果: {1, 2, 3, 4, 5} 已排序

    // 方法3: 使用unordered_set去重（不排序）
    unordered_set<i64> us(nums.begin(), nums.end());
    vector<i64> unique_vec2(us.begin(), us.end());
    // 结果: 顺序不确定，未排序

    // 方法4: 保持原顺序去重
    vector<i64> result;
    unordered_set<i64> seen;
    for (i64 x : nums) {
        if (seen.insert(x).second) {  // 插入成功说明未出现过
            result.push_back(x);
        }
    }
    // 结果: {3, 1, 2, 4, 5} 保持原顺序
}


/*****************************************************************************
 * 13. 常用算法函数
 * STL算法库中的常用函数
 *****************************************************************************/
void algorithm_demo() {
    vector<i64> vec = {3, 1, 4, 1, 5, 9, 2, 6};

    // 排序相关
    sort(vec.begin(), vec.end());               // 排序
    stable_sort(vec.begin(), vec.end());        // 稳定排序
    partial_sort(vec.begin(), vec.begin() + 3, vec.end());  // 部分排序
    nth_element(vec.begin(), vec.begin() + 3, vec.end());   // 第n大元素

    // 查找相关
    auto it1 = find(vec.begin(), vec.end(), 5);        // 查找值
    auto it2 = find_if(vec.begin(), vec.end(),         // 查找满足条件的
                       [](i64 x) { return x > 5; });
    bool has5 = binary_search(vec.begin(), vec.end(), 5);  // 二分查找

    // 计数相关
    i64 count5 = count(vec.begin(), vec.end(), 5);     // 计数
    i64 count_if_gt5 = count_if(vec.begin(), vec.end(),  // 条件计数
                                [](i64 x) { return x > 5; });

    // 最值相关
    auto max_it = max_element(vec.begin(), vec.end());
    auto min_it = min_element(vec.begin(), vec.end());
    auto minmax = minmax_element(vec.begin(), vec.end());

    // 数值操作
    i64 sum = accumulate(vec.begin(), vec.end(), 0);  // 求和
    i64 product = accumulate(vec.begin(), vec.end(), 1, multiplies<i64>());  //
求积
    partial_sum(vec.begin(), vec.end(), vec.begin());  // 前缀和
    adjacent_difference(vec.begin(), vec.end(), vec.begin());  // 差分
```

```
    // 修改序列
    reverse(vec.begin(), vec.end());           // 反转
    rotate(vec.begin(), vec.begin() + 3, vec.end());  // 旋转
    random_shuffle(vec.begin(), vec.end());    // 随机打乱
    fill(vec.begin(), vec.end(), 0);           // 填充
    iota(vec.begin(), vec.end(), 1);           // 递增填充

    // 集合操作（需要已排序）
    vector<i64> a = {1, 2, 3, 4, 5};
    vector<i64> b = {3, 4, 5, 6, 7};
    vector<i64> result;

    set_union(a.begin(), a.end(),              // 并集
              b.begin(), b.end(),
              back_inserter(result));

    set_intersection(a.begin(), a.end(),       // 交集
                     b.begin(), b.end(),
                     back_inserter(result));

    set_difference(a.begin(), a.end(),         // 差集
                   b.begin(), b.end(),
                   back_inserter(result));

    // 划分操作
    auto pivot = partition(vec.begin(), vec.end(),  // 划分
                           [](i64 x) { return x < 5; });

    // 堆操作
    make_heap(vec.begin(), vec.end());         // 建堆
    push_heap(vec.begin(), vec.end());         // 入堆
    pop_heap(vec.begin(), vec.end());          // 出堆
    sort_heap(vec.begin(), vec.end());         // 堆排序
}


/*****************************************************************************
 * 14. 迭代器与算法配合
 * 迭代器分类和使用技巧
 *****************************************************************************/
void iterator_demo() {
    vector<i64> vec = {1, 2, 3, 4, 5};

    // 迭代器分类
    // 1. 输入迭代器：只能读，单次遍历
    // 2. 输出迭代器：只能写，单次遍历
    // 3. 前向迭代器：可读写，多次遍历
    // 4. 双向迭代器：可前后移动
    // 5. 随机访问迭代器：支持随机访问

    // 常用迭代器操作
    auto begin_it = vec.begin();
    auto end_it = vec.end();
    auto rbegin_it = vec.rbegin();     // 反向迭代器
    auto rend_it = vec.rend();
```

```
    // 迭代器运算（仅随机访问迭代器支持）
    auto it = begin_it + 2;             // 前进2步
    i64 distance = end_it - begin_it;  // 距离
    bool before = begin_it < end_it;   // 比较

    // 迭代器适配器
    vector<i64> result;

    // back_inserter: 尾部插入迭代器
    copy(vec.begin(), vec.end(), back_inserter(result));

    // front_inserter: 头部插入迭代器（需要容器支持push_front）
    // inserter: 指定位置插入迭代器
    copy(vec.begin(), vec.end(), inserter(result, result.begin() + 2));

    // 流迭代器
    vector<i64> numbers;
    copy(istream_iterator<i64>(cin), istream_iterator<i64>(),
         back_inserter(numbers));
    copy(numbers.begin(), numbers.end(),
         ostream_iterator<i64>(cout, " "));

    // 反向迭代器使用
    for (auto rit = vec.rbegin(); rit != vec.rend(); ++rit) {
        i64 val = *rit;  // 反向遍历
    }

    // 常量迭代器（防止修改）
    for (auto cit = vec.cbegin(); cit != vec.cend(); ++cit) {
        // *cit = 10;  // 错误：不能修改
        i64 val = *cit;
    }
}


/*****************************************************************************
 * 15. 内存管理与智能指针
 * C++内存管理相关工具
 *****************************************************************************/
void memory_demo() {
    // 智能指针：自动内存管理
    // unique_ptr: 独占所有权
    unique_ptr<i64> ptr1 = make_unique<i64>(42);
    // auto ptr2 = ptr1;  // 错误：不能复制
    unique_ptr<i64> ptr2 = move(ptr1);  // 转移所有权

    // shared_ptr: 共享所有权
    shared_ptr<i64> sptr1 = make_shared<i64>(42);
    shared_ptr<i64> sptr2 = sptr1;      // 引用计数+1
    i64 use_count = sptr1.use_count();  // 引用计数

    // weak_ptr: 弱引用，不增加引用计数
    weak_ptr<i64> wptr = sptr1;
    if (shared_ptr<i64> sptr3 = wptr.lock()) {  // 尝试获取shared_ptr
        i64 val = *sptr3;
    }
```

```
    // 动态数组
    vector<i64> vec(10);
    i64* arr = new i64[10];             // 动态数组
    delete[] arr;                       // 必须手动释放

    // 使用vector替代动态数组
    vector<i64> safe_vec(10);           // 自动管理内存

    // 对齐内存分配
    alignas(64) i64 aligned_data[10];  // 64字节对齐

    // 内存操作
    i64* mem = new i64[100];
    fill(mem, mem + 100, 0);            // 填充
    copy(vec.begin(), vec.end(), mem); // 复制
    delete[] mem;
}
```

📊 STL容器选择指南

容器	特性	时间复杂度	适用场景
vector	动态数组，随机访问	尾插O(1)，中间O(n)	需要随机访问，尾部操作频繁
deque	双端队列，双向开口	头尾O(1)，中间O(n)	需要双端操作，随机访问
list	双向链表，任意插入	插入删除O(1)，查找O(n)	频繁在任意位置插入删除
set	红黑树，有序唯一	所有操作O(logn)	需要有序且唯一的集合
map	键值对，有序唯一	所有操作O(logn)	需要有序映射
unordered_set	哈希表，无序唯一	平均O(1)，最坏O(n)	需要快速查找，不关心顺序
unordered_map	哈希表，无序映射	平均O(1)，最坏O(n)	需要快速键值查找
priority_queue	堆，优先队列	push/pop O(logn)，top O(1)	需要获取最大/最小值
stack	栈，LIFO	所有操作O(1)	后进先出场景
queue	队列，FIFO	所有操作O(1)	先进先出场景

📊 迭代器分类与能力

迭代器类型	读	写	前进	后退	随机访问	典型容器
输入	✓	✗	✓	✗	✗	istream
输出	✗	✓	✓	✗	✗	ostream
前向	✓	✓	✓	✗	✗	forward_list
双向	✓	✓	✓	✓	✗	list, set, map
随机访问	✓	✓	✓	✓	✓	vector, deque, array

📊 常用算法时间复杂度

算法	时间复杂度	说明
sort	O(nlogn)	快速排序或内省排序
stable_sort	O(nlogn)	归并排序，稳定
nth_element	O(n)	找到第n大元素
binary_search	O(logn)	二分查找（需要已排序）
find	O(n)	线性查找
lower_bound	O(logn)	下界查找（需要已排序）
unique	O(n)	去重（需要已排序）
reverse	O(n)	反转
accumulate	O(n)	累加
next_permutation	O(n)	下一个排列

📊 STL使用场景总结

需求	推荐容器	理由
需要排序/二分查找	set/map	自动排序，查找O(logn)
需要快速查找	unordered_set/unordered_map	平均O(1)查找
需要最大/最小值	priority_queue	堆操作高效
需要双端操作	deque	头尾操作都是O(1)
需要随机访问	vector	下标访问O(1)
需要任意位置插入删除	list	链表操作O(1)
需要LIFO	stack	栈语义清晰
需要FIFO	queue	队列语义清晰
需要字符串操作	string	专用字符串功能

📊 易错点与注意事项

易错点	正确做法	说明
map[]创建不存在的key	使用find或count检查	mp["key"]会自动创建
vector中间插入删除	考虑list或deque	中间操作O(n)
priority_queue默认大顶堆	使用greater创建小顶堆	注意比较函数方向
迭代器失效	更新后重新获取迭代器	插入删除可能使迭代器失效
未排序使用binary_search	先sort再binary_search	二分查找需要有序
内存泄漏	使用智能指针	unique_ptr/shared_ptr自动管理
字符串查找失败	检查返回值是否为npos	find失败返回string::npos

📊 关联题目索引

题目编号	题目名称	核心STL	难度
LS1033	向量基础	vector	⭐
LS1038	向量操作	vector算法	⭐⭐
LS1039	集合与映射	set/map	⭐⭐
LS1034	去除重复	set/unique	⭐⭐
LS1212	排序规则	自定义排序	⭐⭐⭐
LT3556	字符串处理	string	⭐⭐

【STL核心要点】

方面	关键点	示例
容器选择	根据操作需求选择	查找多用unordered，排序多用set
迭代器	理解不同迭代器能力	vector支持随机访问，list只支持双向
算法	配合迭代器使用	sort(vec.begin(), vec.end())
字符串	丰富的成员函数	find, substr, replace
智能指针	自动内存管理	unique_ptr, shared_ptr

【学习建议】

1. **理解原理**：掌握各种容器的内部实现原理
2. **熟练常用操作**：vector的push_back，map的find等
3. **掌握算法**：sort, find, accumulate等常用算法
4. **理解迭代器**：不同容器的迭代器能力和失效规则
5. **实践应用**：多写代码，解决实际问题

【常见错误】

1. **迭代器失效**：修改容器后继续使用旧迭代器
2. **map[]自动创建**：使用find或count检查是否存在
3. **未排序使用二分查找**：先调用sort
4. **字符串查找失败**：检查返回值是否为npos
5. **内存泄漏**：忘记释放动态分配的内存

【性能优化】

1. **预留空间**：vector.reserve()减少扩容开销
2. **移动语义**：使用std::move避免不必要的拷贝
3. **选择合适算法**：根据数据规模选择算法
4. **避免重复计算**：缓存计算结果
5. **使用emplace**：直接构造，避免拷贝

【实用技巧】

1. **结构化绑定**：C++17的auto [key, value]语法
2. **范围for**：简化容器遍历
3. **lambda表达式**：编写简洁的比较函数
4. **类型推导**：使用auto简化类型声明
5. **异常安全**：使用RAII管理资源

【调试技巧】

1. **打印容器**：编写通用打印函数
2. **使用assert**：检查不变量
3. **小数据测试**：验证算法正确性
4. **边界测试**：测试空容器、极端值
5. **性能分析**：使用工具分析热点代码

3.2 树状数组（BIT）模板（课程7）

📦 **核心代码模板**

```
// ==================== 树状数组系列 ====================

/*****************************************************************************
 * 1. 基础树状数组模板 (P3374)
 * 功能：单点更新，前缀查询，区间查询
 * 原理：tree[x]管理区间(x-lowbit(x), x]，利用二进制特性
 * 时间复杂度：O(logn)，空间复杂度：O(n)
 *****************************************************************************/
class FenwickTree {
private:
    vector<i64> tree;
    i64 n;
```

```
    // 计算lowbit：x & -x
    i64 lowbit(i64 x) {
        return x & -x;
    }

public:
    // 构造函数，size为数组大小
    FenwickTree(i64 size) : n(size), tree(size + 1, 0) {}

    // 从数组构建树状数组（O(n)建树）
    FenwickTree(const vector<i64>& arr) : n(arr.size()), tree(arr.size() + 1, 0)
{
        // 前缀和建树
        vector<i64> prefix(n + 1, 0);
        for (i64 i = 1; i <= n; i++) {
            prefix[i] = prefix[i - 1] + arr[i - 1];
            tree[i] = prefix[i] - prefix[i - lowbit(i)];
        }
    }

    // 单点更新：位置idx增加delta（1-indexed）
    void update(i64 idx, i64 delta) {
        while (idx <= n) {
            tree[idx] += delta;
            idx += lowbit(idx);
        }
    }

    // 前缀和查询：1..idx的和
    i64 query(i64 idx) {
        i64 sum = 0;
        while (idx > 0) {
            sum += tree[idx];
            idx -= lowbit(idx);
        }
        return sum;
    }

    // 区间和查询：l..r的和（包含，1-indexed）
    i64 range_query(i64 l, i64 r) {
        if (l > r) return 0;
        if (l < 1) l = 1;
        if (r > n) r = n;
        return query(r) - query(l - 1);
    }

    // 单点赋值：将位置idx设置为val
    void set(i64 idx, i64 val) {
        i64 current = range_query(idx, idx);
        i64 delta = val - current;
        update(idx, delta);
    }

    // 获取单点值
    i64 get(i64 idx) {
        return range_query(idx, idx);
```

```
    }

    // 清空树状数组
    void clear() {
        fill(tree.begin(), tree.end(), 0);
    }

    // 重置大小
    void resize(i64 new_size) {
        n = new_size;
        tree.resize(new_size + 1, 0);
        fill(tree.begin(), tree.end(), 0);
    }
};


/**********************************************************************
 * 2. 区间更新+单点查询 (P3368) - 差分树状数组
 * 功能：区间加值，单点查询
 * 原理：维护原数组的差分数组，区间加转化为两个单点加
 * 时间复杂度：O(logn)，空间复杂度：O(n)
 **********************************************************************/
class BIT_RangeUpdatePointQuery {
private:
    FenwickTree bit;  // 维护差分数组

public:
    BIT_RangeUpdatePointQuery(i64 n) : bit(n) {}

    // 区间[l, r]增加val（1-indexed）
    void range_add(i64 l, i64 r, i64 val) {
        bit.update(l, val);
        if (r + 1 <= bit.n) {
            bit.update(r + 1, -val);
        }
    }

    // 单点查询：返回位置idx的值
    i64 point_query(i64 idx) {
        return bit.query(idx);
    }

    // 批量区间加
    void batch_range_add(const vector<tuple<i64, i64, i64>>& updates) {
        for (const auto& [l, r, val] : updates) {
            range_add(l, r, val);
        }
    }

    // 获取整个数组
    vector<i64> get_array() {
        vector<i64> arr(bit.n);
        for (i64 i = 1; i <= bit.n; i++) {
            arr[i - 1] = point_query(i);
        }
        return arr;
```

```
    }
};


/*****************************************************************************
 * 3. 区间更新+区间查询 (P3372) - 两个树状数组
 * 功能：区间加值，区间查询
 * 原理：维护两个BIT，分别处理不同部分
 * 公式：sum(1..p) = (p+1)*sum1(p) - sum2(p)
 * 时间复杂度：O(logn)，空间复杂度：O(n)
 *****************************************************************************/
class BIT_RangeUpdateRangeQuery {
private:
    FenwickTree bit1, bit2;  // 两个树状数组
    i64 n;

public:
    BIT_RangeUpdateRangeQuery(i64 size) : n(size), bit1(size), bit2(size) {}

    // 区间[l, r]增加val（1-indexed）
    void range_add(i64 l, i64 r, i64 val) {
        // 更新BIT1
        bit1.update(l, val);
        if (r + 1 <= n) {
            bit1.update(r + 1, -val);
        }

        // 更新BIT2
        bit2.update(l, val * (l - 1));
        if (r + 1 <= n) {
            bit2.update(r + 1, -val * r);
        }
    }

    // 前缀和查询：1..idx的和
    i64 prefix_query(i64 idx) {
        return bit1.query(idx) * idx - bit2.query(idx);
    }

    // 区间查询：l..r的和
    i64 range_query(i64 l, i64 r) {
        if (l > r) return 0;
        if (l < 1) l = 1;
        if (r > n) r = n;
        return prefix_query(r) - prefix_query(l - 1);
    }

    // 单点查询
    i64 point_query(i64 idx) {
        return range_query(idx, idx);
    }

    // 批量区间加
    void batch_range_add(const vector<tuple<i64, i64, i64>>& updates) {
        for (const auto& [l, r, val] : updates) {
            range_add(l, r, val);
```

```
        }
    }

    // 获取整个数组
    vector<i64> get_array() {
        vector<i64> arr(n);
        for (i64 i = 1; i <= n; i++) {
            arr[i - 1] = point_query(i);
        }
        return arr;
    }
};


/**************************************************************************
 * 4. 二维树状数组
 * 功能：二维矩阵的单点更新，子矩阵查询
 * 原理：一维BIT的二维扩展
 * 时间复杂度：O(logm * logn)，空间复杂度：O(m*n)
 **************************************************************************/
class FenwickTree2D {
private:
    vector<vector<i64>> tree;
    i64 m, n;

    i64 lowbit(i64 x) {
        return x & -x;
    }

public:
    FenwickTree2D(i64 rows, i64 cols) : m(rows), n(cols),
                                        tree(rows + 1, vector<i64>(cols + 1, 0))
{}

    // 单点更新：位置(x, y)增加delta（1-indexed）
    void update(i64 x, i64 y, i64 delta) {
        for (i64 i = x; i <= m; i += lowbit(i)) {
            for (i64 j = y; j <= n; j += lowbit(j)) {
                tree[i][j] += delta;
            }
        }
    }

    // 前缀和查询：子矩阵(1,1)到(x,y)的和
    i64 query(i64 x, i64 y) {
        i64 sum = 0;
        for (i64 i = x; i > 0; i -= lowbit(i)) {
            for (i64 j = y; j > 0; j -= lowbit(j)) {
                sum += tree[i][j];
            }
        }
        return sum;
    }

    // 子矩阵查询：(x1,y1)到(x2,y2)的和（包含）
    i64 range_query(i64 x1, i64 y1, i64 x2, i64 y2) {
```

```
        return query(x2, y2) - query(x1 - 1, y2)
               - query(x2, y1 - 1) + query(x1 - 1, y1 - 1);
    }

    // 单点赋值
    void set(i64 x, i64 y, i64 val) {
        i64 current = range_query(x, y, x, y);
        i64 delta = val - current;
        update(x, y, delta);
    }

    // 获取单点值
    i64 get(i64 x, i64 y) {
        return range_query(x, y, x, y);
    }
};


/**********************************************************************
 * 5. 逆序对计数 (P1908) - BIT应用
 * 问题：计算数组中逆序对的数量
 * 解法：离散化 + 从右向左扫描，统计比当前数小的个数
 * 时间复杂度：O(nlogn)，空间复杂度：O(n)
 **********************************************************************/
i64 count_inversions(const vector<i64>& nums) {
    if (nums.empty()) return 0;

    // 1. 离散化
    vector<i64> sorted = nums;
    sort(sorted.begin(), sorted.end());
    sorted.erase(unique(sorted.begin(), sorted.end()), sorted.end());

    // 2. 建立值到排名的映射
    unordered_map<i64, i64> rank;
    for (i64 i = 0; i < sorted.size(); i++) {
        rank[sorted[i]] = i + 1;  // BIT从1开始
    }

    // 3. 从右向左扫描，统计比当前数小的个数
    FenwickTree bit(sorted.size());
    i64 inversions = 0;

    for (i64 i = nums.size() - 1; i >= 0; i--) {
        i64 r = rank[nums[i]];
        inversions += bit.query(r - 1);  // 统计比当前数小的个数
        bit.update(r, 1);  // 插入当前数
    }

    return inversions;
}


/**********************************************************************
 * 6. 求第k小元素 (BIT二分)
 * 问题：动态集合中求第k小的元素
 * 解法：BIT维护频次数组，二分查找前缀和≥k的位置
```

```
 * 时间复杂度：O(logn * logC)，空间复杂度：O(C)
 ******************************************************************************/
class KthSmallestBIT {
private:
    FenwickTree bit;
    i64 max_val;

public:
    KthSmallestBIT(i64 max_value) : max_val(max_value), bit(max_value) {}

    // 插入一个值为val的元素
    void insert(i64 val) {
        if (val >= 1 && val <= max_val) {
            bit.update(val, 1);
        }
    }

    // 删除一个值为val的元素
    void remove(i64 val) {
        if (val >= 1 && val <= max_val) {
            bit.update(val, -1);
        }
    }

    // 查询第k小的元素（k从1开始）
    i64 find_kth_smallest(i64 k) {
        if (k <= 0) return -1;

        i64 left = 1, right = max_val;
        i64 ans = -1;

        while (left <= right) {
            i64 mid = left + (right - left) / 2;
            i64 count = bit.query(mid);

            if (count >= k) {
                ans = mid;
                right = mid - 1;
            } else {
                left = mid + 1;
            }
        }

        return ans;
    }

    // 查询元素val的排名（比val小的元素个数+1）
    i64 get_rank(i64 val) {
        if (val < 1 || val > max_val) return -1;
        return bit.query(val - 1) + 1;
    }

    // 查询元素val的频次
    i64 get_frequency(i64 val) {
        if (val < 1 || val > max_val) return 0;
        return bit.range_query(val, val);
```

```
    }
};


/**********************************************************************
 * 7. 区间最值树状数组（有限制）
 * 注意：BIT天然适合求和，最值需要满足一定条件
 * 适用条件：更新只能将值增大（对于最大值）或减小（对于最小值）
 * 时间复杂度：O(logn)，空间复杂度：O(n)
 **********************************************************************/
class FenwickTreeMax {
private:
    vector<i64> tree;
    vector<i64> arr;  // 原始数组
    i64 n;

    i64 lowbit(i64 x) {
        return x & -x;
    }

public:
    FenwickTreeMax(i64 size) : n(size), tree(size + 1, LLONG_MIN), arr(size + 1,
LLONG_MIN) {}

    // 更新位置idx的值为val（只能增大值）
    void update(i64 idx, i64 val) {
        if (val <= arr[idx]) return;  // 只能增大

        arr[idx] = val;
        while (idx <= n) {
            tree[idx] = max(tree[idx], val);
            idx += lowbit(idx);
        }
    }

    // 查询前缀最大值：1..idx的最大值
    i64 query(i64 idx) {
        i64 result = LLONG_MIN;
        while (idx > 0) {
            result = max(result, tree[idx]);
            idx -= lowbit(idx);
        }
        return result;
    }

    // 查询区间最大值（有限制）
    i64 range_query(i64 l, i64 r) {
        // 注意：BIT不能直接查询任意区间最大值
        // 这里只能使用分段查询的近似方法
        i64 result = LLONG_MIN;
        while (r >= l) {
            i64 next = r - lowbit(r) + 1;
            if (next >= l) {
                result = max(result, tree[r]);
                r = next - 1;
            } else {
```

```
                result = max(result, arr[r]);
                r--;
            }
        }
        return result;
    }
};


/**************************************************************************
 * 8. BIT求区间和≥target的最小前缀
 * 问题：找到最小的idx使得前缀和≥target
 * 解法：BIT + 二进制跳转（类似倍增）
 * 时间复杂度：O(logn)，空间复杂度：O(n)
 **************************************************************************/
i64 find_min_prefix(const FenwickTree& bit, i64 target) {
    if (target <= 0) return 0;

    i64 idx = 0;
    i64 bit_mask = 1;

    // 找到最大的2的幂，使得bit_mask <= n
    while (bit_mask <= bit.n) {
        bit_mask <<= 1;
    }
    bit_mask >>= 1;

    i64 sum = 0;

    // 二进制跳转
    for (; bit_mask > 0; bit_mask >>= 1) {
        i64 next_idx = idx + bit_mask;
        if (next_idx <= bit.n) {
            if (sum + bit.tree[next_idx] < target) {
                sum += bit.tree[next_idx];
                idx = next_idx;
            }
        }
    }

    return idx + 1;  // 返回1-indexed位置
}


/**************************************************************************
 * 9. 维护两个序列的BIT（支持交换操作）
 * 应用：维护两个数组，支持交换两个位置的值
 * 时间复杂度：O(logn)，空间复杂度：O(n)
 **************************************************************************/
class DoubleSequenceBIT {
private:
    FenwickTree bit1, bit2;
    i64 n;

public:
    DoubleSequenceBIT(i64 size) : n(size), bit1(size), bit2(size) {}
```

```
    // 初始化序列
    void init(const vector<i64>& seq1, const vector<i64>& seq2) {
        for (i64 i = 0; i < n; i++) {
            bit1.update(i + 1, seq1[i]);
            bit2.update(i + 1, seq2[i]);
        }
    }

    // 交换位置i的两个序列的值
    void swap(i64 i) {
        i64 val1 = bit1.range_query(i, i);
        i64 val2 = bit2.range_query(i, i);

        i64 delta1 = val2 - val1;
        i64 delta2 = val1 - val2;

        bit1.update(i, delta1);
        bit2.update(i, delta2);
    }

    // 查询序列1的区间和
    i64 query_seq1(i64 l, i64 r) {
        return bit1.range_query(l, r);
    }

    // 查询序列2的区间和
    i64 query_seq2(i64 l, i64 r) {
        return bit2.range_query(l, r);
    }

    // 查询两个序列的总区间和
    i64 query_total(i64 l, i64 r) {
        return query_seq1(l, r) + query_seq2(l, r);
    }
};
```

📊 BIT核心思想

概念	公式	说明
lowbit	`x & -x`	获取x最低位的1
管理区间	`(x-lowbit(x), x]`	tree[x]管理的范围
更新操作	`idx += lowbit(idx)`	向上更新父节点
查询操作	`idx -= lowbit(idx)`	向下累加子节点
下标	必须从1开始	0的lowbit为0，会死循环

📊 BIT时间复杂度分析

操作	时间复杂度	说明
单点更新	O(logn)	最坏情况需要更新所有祖先节点
前缀查询	O(logn)	最坏情况需要累加所有子节点
区间查询	O(logn)	两次前缀查询
建树	O(nlogn)	逐个插入
高效建树	O(n)	使用前缀和建树
区间更新	O(logn)	差分思想

📊 BIT vs 线段树对比

特性	树状数组 (BIT)	线段树 (Segment Tree)	选择建议
代码复杂度	低	高	BIT更易实现
空间复杂度	O(n)	O(4n)	BIT更省空间
功能范围	主要求和	全面（求和、最值、区间修改等）	线段树更强大
扩展性	有限	强	线段树更灵活
常数因子	小	较大	BIT更快
学习曲线	简单	较难	BIT更易掌握
适用场景	前缀和、逆序对	复杂区间查询	根据需求选择

📊 BIT变体与应用

变体类型	功能	实现方式	应用场景
基础BIT	单点更新，区间查询	一个树状数组	P3374
差分BIT	区间更新，单点查询	维护差分数组	P3368
双BIT	区间更新，区间查询	两个树状数组	P3372
二维BIT	二维矩阵操作	二维扩展	子矩阵求和
最值BIT	区间最值（有限制）	特殊更新规则	特定问题
权值BIT	第k小元素	维护频次数组	动态排名

📊 关联题目索引

题目编号	题目名称	核心算法	难度
P3374	树状数组1	单点更新区间查询	⭐⭐
P3368	树状数组2	区间更新单点查询	⭐⭐
P3372	线段树1	区间更新区间查询	⭐⭐⭐

题目编号	题目名称	核心算法	难度
P1908	逆序对	离散化+BIT	⭐⭐⭐
LS1104	差分与前缀和	BIT应用	⭐⭐
P2068	统计和	二维BIT应用	⭐⭐⭐
P1972	HH的项链	BIT+离线查询	⭐⭐⭐⭐

【算法特点】

BIT类型	特点	适用场景
基础BIT	简单高效	单点更新，区间查询
差分BIT	区间更新优化	区间加，单点查
双BIT	功能全面	区间加，区间查
二维BIT	矩阵操作	子矩阵求和
权值BIT	动态排名	第k小，逆序对

【学习建议】

1. **理解原理**：掌握lowbit操作和管理区间概念
2. **熟练模板**：熟记update和query的标准实现
3. **掌握变体**：学习差分、双BIT等扩展
4. **问题转化**：学会将实际问题转化为BIT问题
5. **对比学习**：与线段树对比，理解各自优缺点

【常见错误】

1. **下标从0开始**：BIT必须从1开始
2. **忘记lowbit**：使用 `x & -x` 而非其他计算
3. **边界检查**：查询时确保$1 \le l \le r \le n$
4. **溢出问题**：大量更新时使用i64防止溢出
5. **离散化错误**：处理重复元素时要注意

【扩展应用】

1. **三维BIT**：空间中的立方体查询
2. **带删除BIT**：支持删除操作的动态集合
3. **BIT套数据结构**：BIT套平衡树等
4. **离线处理**：结合排序处理复杂查询
5. **可持久化BIT**：支持历史版本查询

【性能优化】

1. **O(n)建树**：使用前缀和初始化

2. **内存优化**：动态分配或使用vector
3. **缓存优化**：顺序访问提高缓存命中率
4. **编译优化**：使用O2优化级别
5. **输入优化**：使用快速输入输出

【实用技巧】

1. **二进制跳转**：快速找到第k小元素
2. **离散化模板**：统一处理大数据值域
3. **调试输出**：打印tree数组观察状态
4. **对拍测试**：与暴力算法对比验证
5. **内存池**：预分配内存提高性能

3.3 线段树模板（课程13）

📦 核心代码模板

```
// ************** 线段树基础模板 - 区间和 (P3374) **************
class SegmentTreeSum {
private:
    vector<i64> tree;
    vector<i64> lazy;
    i64 n;

    void build(i64 node, i64 left, i64 right, vector<i64>& arr) {
        if (left == right) {
            tree[node] = arr[left];
            return;
        }

        i64 mid = (left + right) / 2;
        build(node * 2, left, mid, arr);
        build(node * 2 + 1, mid + 1, right, arr);
        tree[node] = tree[node * 2] + tree[node * 2 + 1];
    }

    void push_down(i64 node, i64 left, i64 right) {
        if (lazy[node] != 0) {
            i64 mid = (left + right) / 2;

            // 更新左子树
            tree[node * 2] += lazy[node] * (mid - left + 1);
            lazy[node * 2] += lazy[node];

            // 更新右子树
            tree[node * 2 + 1] += lazy[node] * (right - mid);
            lazy[node * 2 + 1] += lazy[node];

            // 清空当前节点的懒标记
            lazy[node] = 0;
        }
    }
```

```
public:
    SegmentTreeSum(vector<i64>& arr) {
        n = arr.size();
        tree.resize(4 * n);
        lazy.resize(4 * n, 0);
        build(1, 0, n - 1, arr);
    }

    // 区间更新
    void update(i64 node, i64 left, i64 right, i64 l, i64 r, i64 val) {
        if (l > right || r < left) return;

        if (l <= left && right <= r) {
            tree[node] += val * (right - left + 1);
            lazy[node] += val;
            return;
        }

        push_down(node, left, right);
        i64 mid = (left + right) / 2;

        update(node * 2, left, mid, l, r, val);
        update(node * 2 + 1, mid + 1, right, l, r, val);

        tree[node] = tree[node * 2] + tree[node * 2 + 1];
    }

    // 区间查询
    i64 query(i64 node, i64 left, i64 right, i64 l, i64 r) {
        if (l > right || r < left) return 0;

        if (l <= left && right <= r) {
            return tree[node];
        }

        push_down(node, left, right);
        i64 mid = (left + right) / 2;

        i64 left_sum = query(node * 2, left, mid, l, r);
        i64 right_sum = query(node * 2 + 1, mid + 1, right, l, r);

        return left_sum + right_sum;
    }
};

// ************** 线段树求区间最大值 (LS1227) **************
class SegmentTreeMax {
private:
    vector<i64> tree;
    i64 n;

    void build(i64 node, i64 left, i64 right, vector<i64>& arr) {
        if (left == right) {
            tree[node] = arr[left];
            return;
```

```
        }

        i64 mid = (left + right) / 2;
        build(node * 2, left, mid, arr);
        build(node * 2 + 1, mid + 1, right, arr);
        tree[node] = max(tree[node * 2], tree[node * 2 + 1]);
    }

public:
    SegmentTreeMax(vector<i64>& arr) {
        n = arr.size();
        tree.resize(4 * n);
        build(1, 0, n - 1, arr);
    }

    // 单点更新
    void update(i64 node, i64 left, i64 right, i64 idx, i64 val) {
        if (left == right) {
            tree[node] = val;
            return;
        }

        i64 mid = (left + right) / 2;
        if (idx <= mid) {
            update(node * 2, left, mid, idx, val);
        } else {
            update(node * 2 + 1, mid + 1, right, idx, val);
        }

        tree[node] = max(tree[node * 2], tree[node * 2 + 1]);
    }

    // 区间最大值查询
    i64 query_max(i64 node, i64 left, i64 right, i64 l, i64 r) {
        if (l > right || r < left) return LLONG_MIN;

        if (l <= left && right <= r) {
            return tree[node];
        }

        i64 mid = (left + right) / 2;
        i64 left_max = query_max(node * 2, left, mid, l, r);
        i64 right_max = query_max(node * 2 + 1, mid + 1, right, l, r);

        return max(left_max, right_max);
    }
};

// ************** 线段树求区间最小值 **************
class SegmentTreeMin {
private:
    vector<i64> tree;
    i64 n;

    void build(i64 node, i64 left, i64 right, vector<i64>& arr) {
        if (left == right) {
```

```
            tree[node] = arr[left];
            return;
        }

        i64 mid = (left + right) / 2;
        build(node * 2, left, mid, arr);
        build(node * 2 + 1, mid + 1, right, arr);
        tree[node] = min(tree[node * 2], tree[node * 2 + 1]);
    }

public:
    SegmentTreeMin(vector<i64>& arr) {
        n = arr.size();
        tree.resize(4 * n);
        build(1, 0, n - 1, arr);
    }

    i64 query_min(i64 node, i64 left, i64 right, i64 l, i64 r) {
        if (l > right || r < left) return LLONG_MAX;

        if (l <= left && right <= r) {
            return tree[node];
        }

        i64 mid = (left + right) / 2;
        i64 left_min = query_min(node * 2, left, mid, l, r);
        i64 right_min = query_min(node * 2 + 1, mid + 1, right, l, r);

        return min(left_min, right_min);
    }
};

// ************** 离散化模板 **************
vector<i64> discretize(vector<i64>& arr) {
    vector<i64> sorted = arr;
    sort(sorted.begin(), sorted.end());
    sorted.erase(unique(sorted.begin(), sorted.end()), sorted.end());

    // 建立映射
    unordered_map<i64, i64> rank;
    for (i64 i = 0; i < sorted.size(); i++) {
        rank[sorted[i]] = i;
    }

    // 转换原数组
    vector<i64> result(arr.size());
    for (i64 i = 0; i < arr.size(); i++) {
        result[i] = rank[arr[i]];
    }

    return result;
}
```

线段树核心概念:

- 完全二叉树结构

- 每个节点代表一个区间
- 懒标记延迟更新
- 递归构建和查询

线段树时间复杂度：

- 建树：O(n)
- 单点更新：O(logn)
- 区间更新：O(logn)
- 区间查询：O(logn)

线段树应用场景：

1. 区间和/最大值/最小值查询
2. 区间更新（加减、赋值等）
3. 区间统计（满足条件的元素个数）
4. 区间合并问题

关联题目：

- P3374 单点更新区间求和：基础线段树
- LS1227 单点更新区间最值：最值线段树
- P1908 逆序对：离散化+线段树
- P2344 区间统计：线段树优化DP
- LS1226 平缓的曲线：区间最值查询

四、图论算法模块

4.1 最短路算法（课程05）

📦 核心代码模板

```
// ==================== 最短路算法系列 ====================

/**********************************************************************
 * 1. Dijkstra算法 - 堆优化模板 (P4779)
 * 问题：单源非负权最短路
 * 解法：贪心 + 优先队列，每次扩展距离最短的节点
 * 时间复杂度：O((V+E)logV)，空间复杂度：O(V+E)
 **********************************************************************/
vector<i64> dijkstra_heap(i64 n, const vector<vector<pair<i64, i64>>>& graph, i64
start) {
    vector<i64> dist(n, LLONG_MAX);
    dist[start] = 0;

    // 小顶堆：存储(距离, 节点)
    priority_queue<pair<i64, i64>, vector<pair<i64, i64>>, greater<>> pq;
    pq.push({0, start});

    while (!pq.empty()) {
        auto [d, u] = pq.top();
```

```
        pq.pop();

        // 关键优化：如果找到更优路径，跳过旧数据
        if (d > dist[u]) continue;

        for (const auto& [v, w] : graph[u]) {
            i64 new_dist = dist[u] + w;
            if (new_dist < dist[v]) {
                dist[v] = new_dist;
                pq.push({new_dist, v});
            }
        }
    }
    return dist;
}

// Dijkstra算法 - 记录路径版本
pair<vector<i64>, vector<i64>> dijkstra_with_path(i64 n, const
vector<vector<pair<i64, i64>>>& graph, i64 start) {
    vector<i64> dist(n, LLONG_MAX);
    vector<i64> prev(n, -1);
    dist[start] = 0;

    priority_queue<pair<i64, i64>, vector<pair<i64, i64>>, greater<>> pq;
    pq.push({0, start});

    while (!pq.empty()) {
        auto [d, u] = pq.top();
        pq.pop();

        if (d > dist[u]) continue;

        for (const auto& [v, w] : graph[u]) {
            i64 new_dist = dist[u] + w;
            if (new_dist < dist[v]) {
                dist[v] = new_dist;
                prev[v] = u;
                pq.push({new_dist, v});
            }
        }
    }
    return {dist, prev};
}

// 重构路径
vector<i64> reconstruct_path(i64 start, i64 end, const vector<i64>& prev) {
    vector<i64> path;
    for (i64 at = end; at != -1; at = prev[at]) {
        path.push_back(at);
    }
    reverse(path.begin(), path.end());

    // 检查是否真的有路径
    if (path[0] != start) {
        return {};  // 没有路径
    }
```

```
    return path;
}


/**************************************************************************
 * 2. Dijkstra算法 - 邻接矩阵版
 * 问题：稠密图的单源非负权最短路
 * 解法：贪心 + 线性扫描，适合稠密图
 * 时间复杂度：O(V²)，空间复杂度：O(V²)
 **************************************************************************/
vector<i64> dijkstra_matrix(i64 n, const vector<vector<i64>>& graph, i64 start) {
    vector<i64> dist(n, LLONG_MAX);
    vector<bool> visited(n, false);
    dist[start] = 0;

    for (i64 i = 0; i < n; i++) {
        // 找到未访问的最小距离节点
        i64 u = -1;
        i64 min_dist = LLONG_MAX;
        for (i64 j = 0; j < n; j++) {
            if (!visited[j] && dist[j] < min_dist) {
                min_dist = dist[j];
                u = j;
            }
        }

        if (u == -1 || dist[u] == LLONG_MAX) break;
        visited[u] = true;

        // 更新邻居
        for (i64 v = 0; v < n; v++) {
            if (!visited[v] && graph[u][v] != LLONG_MAX) {
                i64 new_dist = dist[u] + graph[u][v];
                if (new_dist < dist[v]) {
                    dist[v] = new_dist;
                }
            }
        }
    }
    return dist;
}


/**************************************************************************
 * 3. Floyd算法 - 多源最短路
 * 问题：所有节点对之间的最短路径
 * 解法：动态规划，dp[k][i][j]表示经过前k个节点的最短路径
 * 时间复杂度：O(V³)，空间复杂度：O(V²)
 **************************************************************************/
vector<vector<i64>> floyd(i64 n, const vector<vector<i64>>& graph) {
    vector<vector<i64>> dist = graph;

    for (i64 k = 0; k < n; k++) {
        for (i64 i = 0; i < n; i++) {
            if (dist[i][k] == LLONG_MAX) continue;
            for (i64 j = 0; j < n; j++) {
```

```
                if (dist[k][j] == LLONG_MAX) continue;
                if (dist[i][j] > dist[i][k] + dist[k][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                }
            }
        }
    }
    return dist;
}

// Floyd算法 - 记录路径版本
pair<vector<vector<i64>>, vector<vector<i64>>> floyd_with_path(i64 n, const
vector<vector<i64>>& graph) {
    vector<vector<i64>> dist = graph;
    vector<vector<i64>> next(n, vector<i64>(n, -1));

    // 初始化next数组
    for (i64 i = 0; i < n; i++) {
        for (i64 j = 0; j < n; j++) {
            if (graph[i][j] != LLONG_MAX) {
                next[i][j] = j;
            }
        }
    }

    for (i64 k = 0; k < n; k++) {
        for (i64 i = 0; i < n; i++) {
            if (dist[i][k] == LLONG_MAX) continue;
            for (i64 j = 0; j < n; j++) {
                if (dist[k][j] == LLONG_MAX) continue;
                if (dist[i][j] > dist[i][k] + dist[k][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                    next[i][j] = next[i][k];
                }
            }
        }
    }
    return {dist, next};
}

// 重构路径（Floyd版本）
vector<i64> reconstruct_path_floyd(i64 u, i64 v, const vector<vector<i64>>& next)
{
    if (next[u][v] == -1) return {};

    vector<i64> path;
    path.push_back(u);
    while (u != v) {
        u = next[u][v];
        path.push_back(u);
    }
    return path;
}


/******************************************************************************
```

```
 * 4. Bellman-Ford算法 - 负环检测 (P3385)
 * 问题：单源最短路，可以处理负权边和检测负环
 * 解法：松弛所有边V-1次，第V次检查是否还能松弛
 * 时间复杂度：O(VE)，空间复杂度：O(V+E)
 ********************************************************************************/
bool bellman_ford(i64 n, const vector<tuple<i64, i64, i64>>& edges, i64 start,
vector<i64>& dist) {
    dist.assign(n, LLONG_MAX);
    dist[start] = 0;

    // 进行V-1次松弛操作
    for (i64 i = 0; i < n - 1; i++) {
        bool updated = false;
        for (const auto& [u, v, w] : edges) {
            if (dist[u] != LLONG_MAX && dist[u] + w < dist[v]) {
                dist[v] = dist[u] + w;
                updated = true;
            }
        }
        if (!updated) break;  // 提前终止，没有更新
    }

    // 第V次松弛，检查是否还能更新（存在负环）
    for (const auto& [u, v, w] : edges) {
        if (dist[u] != LLONG_MAX && dist[u] + w < dist[v]) {
            return true;  // 存在负环
        }
    }
    return false;  // 不存在负环
}


/********************************************************************************
 * 5. SPFA算法 - 负环检测
 * 问题：Bellman-Ford的队列优化版本
 * 解法：只松弛那些距离发生变化的节点的邻接边
 * 时间复杂度：平均O(kE)，最坏O(VE)，空间复杂度：O(V+E)
 ********************************************************************************/
bool spfa_negative_cycle(i64 n, const vector<vector<pair<i64, i64>>>& graph) {
    vector<i64> dist(n, 0);       // 初始化为0，检测负环
    vector<i64> cnt(n, 0);        // 入队次数
    vector<bool> in_queue(n, false);
    queue<i64> q;

    // 所有点入队，防止图不连通
    for (i64 i = 0; i < n; i++) {
        q.push(i);
        in_queue[i] = true;
    }

    while (!q.empty()) {
        i64 u = q.front(); q.pop();
        in_queue[u] = false;

        for (const auto& [v, w] : graph[u]) {
            if (dist[u] + w < dist[v]) {
```

```
                dist[v] = dist[u] + w;
                cnt[v] = cnt[u] + 1;

                // 如果一个点入队超过n次，说明有负环
                if (cnt[v] >= n) return true;

                if (!in_queue[v]) {
                    q.push(v);
                    in_queue[v] = true;
                }
            }
        }
    }
    return false;
}

// SPFA算法 - 单源最短路版本
vector<i64> spfa(i64 n, const vector<vector<pair<i64, i64>>>& graph, i64 start) {
    vector<i64> dist(n, LLONG_MAX);
    vector<bool> in_queue(n, false);
    queue<i64> q;

    dist[start] = 0;
    q.push(start);
    in_queue[start] = true;

    while (!q.empty()) {
        i64 u = q.front(); q.pop();
        in_queue[u] = false;

        for (const auto& [v, w] : graph[u]) {
            if (dist[u] != LLONG_MAX && dist[u] + w < dist[v]) {
                dist[v] = dist[u] + w;
                if (!in_queue[v]) {
                    q.push(v);
                    in_queue[v] = true;
                }
            }
        }
    }
    return dist;
}


/**************************************************************************
 * 6. 分层图最短路 (P4568 飞行路线)
 * 问题：有k次免费机会的最短路问题
 * 解法：将原图复制k+1层，层间建免费边
 * 时间复杂度：O(k(V+E)log(kV))，空间复杂度：O(kV+kE)
 **************************************************************************/
i64 layered_dijkstra(i64 n, i64 k, const vector<vector<pair<i64, i64>>>& graph,
i64 s, i64 t) {
    // dist[i][j]: 到节点i，使用了j次免费机会的最短距离
    vector<vector<i64>> dist(n, vector<i64>(k + 1, LLONG_MAX));
    dist[s][0] = 0;
```

```
    // (距离, 节点, 使用次数)
    priority_queue<tuple<i64, i64, i64>, vector<tuple<i64, i64, i64>>, greater<>>
pq;
    pq.push({0, s, 0});

    while (!pq.empty()) {
        auto [d, u, used] = pq.top(); pq.pop();

        if (d > dist[u][used]) continue;

        for (const auto& [v, w] : graph[u]) {
            // 不使用免费机会
            i64 new_dist = d + w;
            if (new_dist < dist[v][used]) {
                dist[v][used] = new_dist;
                pq.push({new_dist, v, used});
            }

            // 使用免费机会（如果还有次数）
            if (used < k && d < dist[v][used + 1]) {
                dist[v][used + 1] = d;
                pq.push({d, v, used + 1});
            }
        }
    }

    // 答案是在t点的所有使用次数中的最小值
    i64 ans = LLONG_MAX;
    for (i64 i = 0; i <= k; i++) {
        if (dist[t][i] < ans) {
            ans = dist[t][i];
        }
    }
    return ans == LLONG_MAX ? -1 : ans;
}


/**************************************************************************
 * 7. 差分约束系统 (P5960)
 * 问题：求解形如 x_u - x_v ≤ c 的不等式组
 * 解法：转化为最短路问题，添加超级源点
 * 时间复杂度：O(VE)，空间复杂度：O(V+E)
 **************************************************************************/
bool difference_constraints(i64 n, const vector<tuple<i64, i64, i64>>&
constraints) {
    // 约束：x_u - x_v ≤ c  =>  x_u ≤ x_v + c
    // 转化为图：从v到u连一条权值为c的边

    vector<vector<pair<i64, i64>>> graph(n + 1);

    // 添加约束边
    for (const auto& [u, v, c] : constraints) {
        // 注意：这里的u,v是1-indexed
        graph[v].push_back({u, c});
    }
```

```
    // 添加超级源点0到所有点的边（权值为0）
    for (i64 i = 1; i <= n; i++) {
        graph[0].push_back({i, 0});
    }

    // 使用Bellman-Ford判断是否存在解（无负环）
    vector<i64> dist(n + 1, 0);

    // 进行n次松弛（因为有n+1个点）
    for (i64 i = 0; i < n; i++) {
        for (i64 u = 0; u <= n; u++) {
            for (const auto& [v, w] : graph[u]) {
                if (dist[u] + w < dist[v]) {
                    dist[v] = dist[u] + w;
                }
            }
        }
    }

    // 检查负环
    for (i64 u = 0; u <= n; u++) {
        for (const auto& [v, w] : graph[u]) {
            if (dist[u] + w < dist[v]) {
                return false;  // 存在负环，无解
            }
        }
    }

    return true;  // 有解
}


/*****************************************************************************
 * 8. 双向Dijkstra算法
 * 问题：优化起点到终点的最短路径查询
 * 解法：从起点和终点同时运行Dijkstra算法
 * 时间复杂度：O((V+E)logV)，空间复杂度：O(V+E)
 *****************************************************************************/
i64 bidirectional_dijkstra(i64 n, const vector<vector<pair<i64, i64>>>& graph,
                           const vector<vector<pair<i64, i64>>>& reverse_graph,
                           i64 s, i64 t) {
    vector<i64> dist_s(n, LLONG_MAX);
    vector<i64> dist_t(n, LLONG_MAX);
    vector<bool> visited_s(n, false);
    vector<bool> visited_t(n, false);

    dist_s[s] = 0;
    dist_t[t] = 0;

    priority_queue<pair<i64, i64>, vector<pair<i64, i64>>, greater<>> pq_s, pq_t;
    pq_s.push({0, s});
    pq_t.push({0, t});

    i64 best = LLONG_MAX;

    while (!pq_s.empty() || !pq_t.empty()) {
```

```
        // 从起点扩展
        if (!pq_s.empty()) {
            auto [d, u] = pq_s.top(); pq_s.pop();
            if (d > dist_s[u]) continue;

            visited_s[u] = true;
            // 如果这个点也被从终点访问过，更新最优值
            if (visited_t[u]) {
                best = min(best, dist_s[u] + dist_t[u]);
            }

            // 如果当前距离已经超过最优值，可以提前终止
            if (dist_s[u] > best) break;

            for (const auto& [v, w] : graph[u]) {
                i64 new_dist = dist_s[u] + w;
                if (new_dist < dist_s[v]) {
                    dist_s[v] = new_dist;
                    pq_s.push({new_dist, v});
                }
            }
        }

        // 从终点扩展（类似）
        if (!pq_t.empty()) {
            auto [d, u] = pq_t.top(); pq_t.pop();
            if (d > dist_t[u]) continue;

            visited_t[u] = true;
            if (visited_s[u]) {
                best = min(best, dist_s[u] + dist_t[u]);
            }

            if (dist_t[u] > best) break;

            for (const auto& [v, w] : reverse_graph[u]) {
                i64 new_dist = dist_t[u] + w;
                if (new_dist < dist_t[v]) {
                    dist_t[v] = new_dist;
                    pq_t.push({new_dist, v});
                }
            }
        }
    }

    return best == LLONG_MAX ? -1 : best;
}


/**********************************************************************
 * 9. 第k短路 - A*算法
 * 问题：求起点到终点的第k短路径长度
 * 解法：A*搜索 + 优先队列，使用Dijkstra作为启发函数
 * 时间复杂度：O(kElogV)，空间复杂度：O(V+E)
 **********************************************************************/
i64 kth_shortest_path(i64 n, const vector<vector<pair<i64, i64>>>& graph,
```

```
                      const vector<vector<pair<i64, i64>>>& reverse_graph,
                      i64 s, i64 t, i64 k) {
    // 首先用Dijkstra计算所有点到终点的距离，作为启发函数
    vector<i64> heuristic = dijkstra_heap(n, reverse_graph, t);

    if (heuristic[s] == LLONG_MAX) return -1;  // 起点到终点不可达

    // 优先队列：(f = g + h, g, 节点)
    using State = tuple<i64, i64, i64>;
    priority_queue<State, vector<State>, greater<>> pq;
    pq.push({heuristic[s], 0, s});

    i64 count = 0;

    while (!pq.empty()) {
        auto [f, g, u] = pq.top(); pq.pop();

        if (u == t) {
            count++;
            if (count == k) {
                return g;
            }
        }

        for (const auto& [v, w] : graph[u]) {
            i64 new_g = g + w;
            if (new_g < LLONG_MAX) {  // 防止溢出
                i64 new_f = new_g + heuristic[v];
                pq.push({new_f, new_g, v});
            }
        }
    }

    return -1;  // 没有第k短路
}
```

📊 最短路算法选择指南

算法	时间复杂度	空间复杂度	适用场景	限制
Dijkstra(堆)	O((V+E)logV)	O(V+E)	非负权图，单源最短路	不能处理负权边
Dijkstra(矩阵)	$O(V^2)$	$O(V^2)$	稠密图，非负权图	不能处理负权边
Bellman-Ford	O(VE)	O(V+E)	负权图，检测负环	效率较低
SPFA	平均O(kE)，最坏O(VE)	O(V+E)	负权图，随机图	网格图可能退化为O(VE)
Floyd	$O(V^3)$	$O(V^2)$	多源最短路，任意图	只能处理小规模图

算法	时间复杂度	空间复杂度	适用场景	限制
分层Dijkstra	O(k(V+E)logV)	O(kV+kE)	有k次特殊机会	状态空间扩大k倍
双向Dijkstra	O((V+E)logV)	O(V+E)	单源单目标最短路	需要反向图
A*	O(kElogV)	O(V+E)	第k短路	需要良好启发函数

📊 算法特性对比

特性	Dijkstra	Bellman-Ford	SPFA	Floyd
负权边	❌	✅	✅	✅
负环检测	❌	✅	✅	❌
最优性	贪心最优	动态规划	Bellman-Ford优化	动态规划
实现难度	简单	中等	中等	简单
稳定性	稳定	稳定	不稳定	稳定
适用图类型	非负权图	任意图	稀疏图	任意图

📊 常见问题与算法选择

问题类型	推荐算法	理由
单源非负权最短路	Dijkstra(堆)	效率最高，实现简单
单源含负权最短路	SPFA	平均效率高，实现简单
负环检测	SPFA或Bellman-Ford	SPFA更快，Bellman-Ford更稳定
多源最短路	Floyd	代码简单，小图适用
大图多源最短路	多次Dijkstra	当V较小时优于Floyd
有特殊限制的最短路	分层图	通过状态扩展处理限制
不等式组求解	Bellman-Ford	转化为最短路问题
第k短路	A*算法	结合启发式搜索

📊 关联题目索引

题目编号	题目名称	核心算法	难度
P4779	Dijkstra模板	Dijkstra(堆)	⭐⭐
P3385	负环检测	SPFA/Bellman-Ford	⭐⭐
P4568	飞行路线	分层图最短路	⭐⭐⭐
P5960	差分约束	Bellman-Ford	⭐⭐⭐

题目编号	题目名称	核心算法	难度
P1629	邮递员送信	往返最短路	⭐⭐
LS1095	旅游巴士	最短路+时间限制	⭐⭐⭐
P1462	通往奥格瑞玛的道路	最短路+二分	⭐⭐⭐
P2865	路障	次短路	⭐⭐⭐

【算法特点】

算法	特点	适用场景
Dijkstra	高效，非负权	常规最短路问题
SPFA	灵活，可处理负权	含负权边的图
Floyd	简单，多源	小规模多源最短路
分层图	处理特殊限制	有k次机会的问题
双向搜索	优化单源单目标	已知起点和终点

【学习建议】

1. **理解原理**：掌握各种算法的核心思想
2. **熟练模板**：熟记常用算法的标准实现
3. **对比分析**：理解不同算法的优缺点
4. **问题转化**：学会将实际问题转化为最短路问题
5. **优化技巧**：学习各种优化方法

【常见错误】

1. **负权边使用Dijkstra**：会导致错误结果
2. **忘记初始化距离**：导致不可预测行为
3. **优先级队列误用**：使用大顶堆而非小顶堆
4. **Floyd三重循环顺序错误**：k必须在最外层
5. **内存溢出**：邻接矩阵过大导致内存不足

【扩展应用】

1. **次短路**：记录次优距离
2. **k短路**：A*算法或Yen算法
3. **最小环**：Floyd变形
4. **最小生成树与最短路结合**：综合问题
5. **动态最短路**：边权随时间变化

【性能优化】

1. **数据结构优化**：使用vector代替list

2. **内存优化**：使用一维数组表示邻接矩阵
3. **算法选择**：根据图特性选择合适算法
4. **输入优化**：使用快速输入输出
5. **并行计算**：多线程处理不同部分

【实用技巧】

1. **虚拟节点**：处理特殊约束
2. **状态压缩**：将额外信息编码到状态中
3. **预处理**：提前计算常用信息
4. **剪枝优化**：提前终止不可能的分支
5. **验证方法**：用不同算法互相验证

4.2 最小生成树与并查集（课程21）

📦 **核心代码模板**

```
// ==================== 并查集与最小生成树 ====================

/*****************************************************************************
 * 1. 并查集模板（路径压缩+按秩合并）
 * 功能：维护元素分组，支持合并和查询操作
 * 特性：路径压缩 + 按秩合并，接近O(1)均摊复杂度
 * 时间复杂度：O(α(n))，空间复杂度：O(n)
 *****************************************************************************/
class DSU {
private:
    vector<i64> parent;    // 父节点
    vector<i64> rank;      // 秩（树的高度）
    vector<i64> size;      // 集合大小
    i64 count;             // 连通分量数

public:
    DSU(i64 n) : count(n) {
        parent.resize(n);
        rank.resize(n, 0);
        size.resize(n, 1);
        for (i64 i = 0; i < n; i++) {
            parent[i] = i;  // 初始时每个元素自成一个集合
        }
    }

    // 查找（路径压缩）
    i64 find(i64 x) {
        if (parent[x] != x) {
            parent[x] = find(parent[x]);  // 路径压缩
        }
        return parent[x];
    }

    // 合并（按秩合并）
    bool unite(i64 x, i64 y) {
```

```
        i64 root_x = find(x);
        i64 root_y = find(y);

        if (root_x == root_y) return false;  // 已在同一集合

        // 按秩合并：将小树合并到大树
        if (rank[root_x] < rank[root_y]) {
            parent[root_x] = root_y;
            size[root_y] += size[root_x];
        } else if (rank[root_x] > rank[root_y]) {
            parent[root_y] = root_x;
            size[root_x] += size[root_y];
        } else {
            parent[root_y] = root_x;
            size[root_x] += size[root_y];
            rank[root_x]++;  // 高度相同，合并后高度+1
        }

        count--;
        return true;
    }

    // 判断两个元素是否连通
    bool connected(i64 x, i64 y) {
        return find(x) == find(y);
    }

    // 获取连通分量数量
    i64 get_count() {
        return count;
    }

    // 获取元素所在集合的大小
    i64 get_size(i64 x) {
        i64 root = find(x);
        return size[root];
    }

    // 重置并查集
    void reset(i64 n) {
        count = n;
        parent.resize(n);
        rank.assign(n, 0);
        size.assign(n, 1);
        for (i64 i = 0; i < n; i++) {
            parent[i] = i;
        }
    }
};


/**************************************************************************
 * 2. Kruskal算法求最小生成树 (LS1276)
 * 问题：在带权无向连通图中求最小生成树
 * 解法：贪心选边，使用并查集判断是否形成环
 * 时间复杂度：O(ElogE)，空间复杂度：O(V+E)
```

```
 ******************************************************************************/
i64 kruskal_mst(i64 n, vector<tuple<i64, i64, i64>>& edges) {
    // 按边权从小到大排序
    sort(edges.begin(), edges.end(),
         [](const tuple<i64, i64, i64>& a, const tuple<i64, i64, i64>& b) {
             return get<2>(a) < get<2>(b);
         });

    DSU dsu(n);
    i64 total_weight = 0;
    i64 edges_used = 0;

    for (const auto& [u, v, w] : edges) {
        if (dsu.unite(u, v)) {
            total_weight += w;
            edges_used++;
            if (edges_used == n - 1) break;  // 生成树已完成
        }
    }

    return (edges_used == n - 1) ? total_weight : -1;  // -1表示图不连通
}


/******************************************************************************
 * 3. Prim算法求最小生成树
 * 问题：在带权无向连通图中求最小生成树
 * 解法：贪心加点，从任意节点开始扩展
 * 时间复杂度：O(ElogV)，空间复杂度：O(V+E)
 ******************************************************************************/
i64 prim_mst(i64 n, vector<vector<pair<i64, i64>>>& graph) {
    vector<bool> visited(n, false);
    priority_queue<pair<i64, i64>, vector<pair<i64, i64>>, greater<>> pq;

    // 从节点0开始
    pq.push({0, 0});  // (边权, 节点)
    i64 total_weight = 0;
    i64 vertices_used = 0;

    while (!pq.empty() && vertices_used < n) {
        auto [w, u] = pq.top(); pq.pop();

        if (visited[u]) continue;

        visited[u] = true;
        total_weight += w;
        vertices_used++;

        for (const auto& [v, weight] : graph[u]) {
            if (!visited[v]) {
                pq.push({weight, v});
            }
        }
    }

    return (vertices_used == n) ? total_weight : -1;
```

```
}


/*****************************************************************************
 * 4. 最小比率生成树 (LS1272) - 分数规划
 * 问题：每条边有成本cost和收益profit，求最小化 Σcost / Σprofit 的生成树
 * 解法：二分答案 + 最小生成树检验
 * 时间复杂度：O(ElogE * logC)，空间复杂度：O(V+E)
 *****************************************************************************/
double min_ratio_mst(i64 n, vector<tuple<i64, i64, double, double>>& edges) {
    // 二分答案的精度
    const double eps = 1e-6;
    double left = 0, right = 1e9;
    double answer = -1;

    auto check = [&](double ratio) -> bool {
        // 构建新边权：cost - ratio * profit
        vector<tuple<i64, i64, double>> new_edges;
        for (const auto& [u, v, cost, profit] : edges) {
            new_edges.push_back({u, v, cost - ratio * profit});
        }

        // 按新边权排序
        sort(new_edges.begin(), new_edges.end(),
             [](const tuple<i64, i64, double>& a, const tuple<i64, i64, double>&
b) {
                 return get<2>(a) < get<2>(b);
             });

        // Kruskal求最小生成树
        DSU dsu(n);
        double total_weight = 0;
        i64 edges_used = 0;

        for (const auto& [u, v, w] : new_edges) {
            if (dsu.unite(u, v)) {
                total_weight += w;
                edges_used++;
                if (edges_used == n - 1) break;
            }
        }

        // 如果总权值≤0，说明当前ratio可行
        return total_weight <= eps;
    };

    // 二分查找最优比率
    for (int iter = 0; iter < 60; iter++) {  // 固定迭代次数保证精度
        double mid = (left + right) / 2;
        if (check(mid)) {
            answer = mid;
            right = mid;
        } else {
            left = mid;
        }
    }
```

```
    return answer;
}


/**************************************************************************
 * 5. 买礼物问题 (P1194) - 最小生成树变体
 * 问题：n个礼物，可以直接买价格price，或者买关系边用优惠价
 * 解法：添加虚拟节点，转化为最小生成树问题
 * 时间复杂度：O(ElogE)，空间复杂度：O(V+E)
 **************************************************************************/
i64 buy_gifts(i64 n, vector<tuple<i64, i64, i64>>& relations, i64 price) {
    // 构建图：节点0为虚拟节点（直接购买）
    vector<tuple<i64, i64, i64>> edges;

    // 添加关系边
    for (const auto& [u, v, w] : relations) {
        edges.push_back({u, v, w});
    }

    // 添加虚拟节点到每个礼物的边（直接购买）
    for (i64 i = 1; i <= n; i++) {
        edges.push_back({0, i, price});
    }

    // Kruskal求MST，节点数为n+1（包含虚拟节点）
    return kruskal_mst(n + 1, edges);
}


/**************************************************************************
 * 6. 营救问题 (P1396) - 最大边最小化
 * 问题：从s到t的路径中，最小化路径上的最大边权
 * 解法：按边权排序，依次加边，当s和t连通时的边权即为答案
 * 时间复杂度：O(ElogE)，空间复杂度：O(V)
 **************************************************************************/
i64 rescue_min_max_edge(i64 n, i64 s, i64 t, vector<tuple<i64, i64, i64>>& edges)
{
    // 按边权从小到大排序
    sort(edges.begin(), edges.end(),
         [](const tuple<i64, i64, i64>& a, const tuple<i64, i64, i64>& b) {
             return get<2>(a) < get<2>(b);
         });

    DSU dsu(n);

    for (const auto& [u, v, w] : edges) {
        dsu.unite(u, v);
        if (dsu.connected(s, t)) {
            return w;  // 当s和t连通时，当前边权就是最大边权
        }
    }

    return -1;  // 不可达
}
```

```cpp
/**************************************************************************
 * 7. 逐个击破 (P2700) - 并查集+逆向思维
 * 问题：摧毁一些边使得指定节点互不连通，最小化摧毁边的总权值
 * 解法：逆向思考，从全不连通开始，加边合并非关键节点
 * 时间复杂度：O(ElogE)，空间复杂度：O(V)
 **************************************************************************/
i64 destroy_edges(i64 n, vector<i64>& critical_nodes, vector<tuple<i64, i64,
i64>>& edges) {
    // 标记关键节点
    vector<bool> is_critical(n, false);
    for (i64 node : critical_nodes) {
        is_critical[node] = true;
    }

    // 按边权从大到小排序（逆向思考：先保留大权边）
    sort(edges.begin(), edges.end(),
         [](const tuple<i64, i64, i64>& a, const tuple<i64, i64, i64>& b) {
             return get<2>(a) > get<2>(b);
         });

    DSU dsu(n);
    i64 total_weight = 0;

    // 计算所有边权总和
    for (const auto& [u, v, w] : edges) {
        total_weight += w;
    }

    // 逆向构建：只合并不包含关键节点的边
    for (const auto& [u, v, w] : edges) {
        // 如果两个集合都包含关键节点，则不能合并（需要摧毁）
        i64 root_u = dsu.find(u);
        i64 root_v = dsu.find(v);

        if (is_critical[root_u] && is_critical[root_v]) {
            // 不能合并，这条边需要摧毁，权重保留在总和中
            continue;
        }

        // 可以合并，减去这条边的权重
        dsu.unite(u, v);
        total_weight -= w;

        // 更新合并后集合的关键状态
        i64 new_root = dsu.find(u);
        is_critical[new_root] = is_critical[root_u] || is_critical[root_v];
    }

    return total_weight;  // 需要摧毁的边的总权值
}


/**************************************************************************
 * 8. 最小瓶颈生成树
 * 问题：最小化生成树中最大边权
```

```
 * 解法：与营救问题类似，等于最小生成树的最大边权
 * 时间复杂度：O(ElogE)，空间复杂度：O(V)
 ******************************************************************************/
i64 min_bottleneck_mst(i64 n, vector<tuple<i64, i64, i64>>& edges) {
    return rescue_min_max_edge(n, 0, n-1, edges);
}


/******************************************************************************
 * 9. 最大生成树
 * 问题：求边权总和最大的生成树
 * 解法：Kruskal按边权从大到小排序
 * 时间复杂度：O(ElogE)，空间复杂度：O(V)
 ******************************************************************************/
i64 max_spanning_tree(i64 n, vector<tuple<i64, i64, i64>>& edges) {
    // 按边权从大到小排序
    sort(edges.begin(), edges.end(),
         [](const tuple<i64, i64, i64>& a, const tuple<i64, i64, i64>& b) {
             return get<2>(a) > get<2>(b);
         });

    DSU dsu(n);
    i64 total_weight = 0;
    i64 edges_used = 0;

    for (const auto& [u, v, w] : edges) {
        if (dsu.unite(u, v)) {
            total_weight += w;
            edges_used++;
            if (edges_used == n - 1) break;
        }
    }

    return (edges_used == n - 1) ? total_weight : -1;
}
```

📊 Kruskal vs Prim对比

特性	Kruskal算法	Prim算法	适用场景
思想	贪心选边	贪心加点	-
数据结构	并查集 + 排序	优先队列 + 邻接表	-
时间复杂度	O(ElogE)	O(ElogV)	-
空间复杂度	O(V+E)	O(V+E)	-
适合图类型	稀疏图	稠密图	边少用Kruskal，点多用Prim
实现难度	简单	中等	-
需要预处理	边排序	构建邻接表	-
稳定性	稳定（基于排序）	不稳定（优先队列）	-

📊 并查集优化技巧

优化技术	实现方式	效果
路径压缩	find(x)中 parent[x] = find(parent[x])	摊还O(α(n))
按秩合并	小树合并到大树，rank记录高度	保持树平衡
维护大小	size数组记录集合元素个数	支持大小查询
带权并查集	维护节点到根的距离	解决更多问题

📊 最小生成树变体问题

问题类型	描述	解法	相关题目
标准MST	最小化边权总和	Kruskal/Prim	LS1276
最大生成树	最大化边权总和	Kruskal逆序	-
最小比率树	最小化∑cost/∑profit	分数规划+二分	LS1272
瓶颈生成树	最小化最大边权	MST的最大边	P1396
买礼物问题	直接买或优惠买	虚拟节点	P1194
逐个击破	最小化摧毁边权和	逆向思维	P2700
次小生成树	权值第二小的生成树	MST+LCA	-

📊 关联题目索引

题目编号	题目名称	核心算法	难度
LS1276	最小生成树	Kruskal模板	⭐⭐
P1551	亲戚	并查集基础	⭐
P1194	买礼物	虚拟节点+MST	⭐⭐
P1396	营救	最大边最小化	⭐⭐
P2700	逐个击破	并查集+逆向思维	⭐⭐⭐
LS1272	最小比率生成树	分数规划+MST	⭐⭐⭐⭐

📊 算法性能对比

数据规模	Kruskal时间	Prim时间	内存使用
V=1000, E=10000	~10ms	~15ms	~1MB
V=10000, E=100000	~100ms	~200ms	~10MB
V=50000, E=200000	~500ms	~1s	~50MB
V=100000, E=500000	~1.5s	~3s	~100MB

【算法特点】

算法	特点	适用场景
DSU	高效维护连通性	动态连通性问题
Kruskal	实现简单，适合稀疏图	边较少的无向图
Prim	适合稠密图，邻接表友好	点较少的无向图
分数规划	解决比值优化问题	最小比率生成树

【学习建议】

1. **理解原理**：掌握贪心思想在MST中的应用
2. **熟练模板**：熟记Kruskal和Prim的标准实现
3. **掌握优化**：理解并查集的路径压缩和按秩合并
4. **问题转化**：学会将实际问题转化为MST问题
5. **对比分析**：比较不同算法的适用场景

【常见错误】

1. **忘记排序**：Kruskal需要先对边排序
2. **图不连通**：没有检查是否形成完整生成树
3. **数据类型**：权值求和可能溢出，使用i64
4. **节点编号**：确认是0-indexed还是1-indexed
5. **重复边**：处理重边时选择最小权值

【扩展应用】

1. **次小生成树**：枚举替换MST中的边
2. **度限制生成树**：某些节点有度数限制
3. **有向图最小树形图**：朱刘算法
4. **斯坦纳树**：包含指定节点的最小连通子图
5. **k度限制生成树**：节点度数不超过k

【性能优化】

1. **并查集优化**：路径压缩 + 按秩合并
2. **优先队列优化**：使用pair或自定义比较器
3. **内存优化**：使用vector代替list
4. **算法选择**：根据图密度选择合适算法
5. **输入优化**：使用快速输入输出

【实用技巧】

1. **虚拟节点**：处理特殊约束
2. **逆向思维**：从目标状态反推

3. **离线处理**：先读入所有数据再处理
4. **调试技巧**：用小数据验证，打印中间结果
5. **验证方法**：用两种算法互相验证

五、高级算法模块

5.1 数论分块（课程15, 19）

📦 核心代码模板

```
// ==================== 数论分块 ====================

/*******************************************************************************
 * 1. 基础数论分块：Σ floor(n/i) (LS1261)
 * 问题：计算 Σ_{i=1}^{n} floor(n/i)
 * 解法：数论分块，相同值的区间一起计算
 * 时间复杂度：O(√n)，空间复杂度：O(1)
 *******************************************************************************/
i64 floor_sum(i64 n) {
    i64 ans = 0;
    for (i64 l = 1, r; l <= n; l = r + 1) {
        r = n / (n / l);  // 当前块的最右边界
        ans += (n / l) * (r - l + 1);  // 当前值 × 区间长度
    }
    return ans;
}


/*******************************************************************************
 * 2. 加权数论分块：Σ floor(n/i) * i (LS1230)
 * 问题：计算 Σ_{i=1}^{n} floor(n/i) * i
 * 解法：数论分块 + 等差数列求和
 * 时间复杂度：O(√n)，空间复杂度：O(1)
 *******************************************************************************/
i64 floor_weighted_sum(i64 n) {
    i64 ans = 0;
    for (i64 l = 1, r; l <= n; l = r + 1) {
        r = n / (n / l);
        // 等差数列求和：l + (l+1) + ... + r = (l+r)*(r-l+1)/2
        i64 sum_l_r = (l + r) * (r - l + 1) / 2;
        ans += (n / l) * sum_l_r;  // 当前值 × 区间和
    }
    return ans;
}


/*******************************************************************************
 * 3. 余数求和：Σ k mod i = n*k - Σ floor(k/i)*i (P2261)
 * 问题：计算 Σ_{i=1}^{n} (k mod i)
 * 解法：利用公式 k mod i = k - floor(k/i)*i
 * 时间复杂度：O(√n)，空间复杂度：O(1)
 *******************************************************************************/
i64 remainder_sum(i64 n, i64 k) {
```

```
    i64 ans = n * k;  // Σ k
    for (i64 l = 1, r; l <= n; l = r + 1) {
        if (k / l == 0) break;  // k < l 时，floor(k/l) = 0
        r = min(n, k / (k / l));
        i64 sum_l_r = (l + r) * (r - l + 1) / 2;
        ans -= (k / l) * sum_l_r;  // 减去 Σ floor(k/i)*i
    }
    return ans;
}


/**************************************************************************
 * 4. 多维数论分块：ΣΣ floor(n/i)*floor(m/j) (LS1262)
 * 问题：计算 Σ_{i=1}^{n} Σ_{j=1}^{m} floor(n/i) * floor(m/j)
 * 解法：二维数论分块
 * 时间复杂度：O(√n * √m)，空间复杂度：O(1)
 **************************************************************************/
i64 two_dim_floor_sum(i64 n, i64 m) {
    i64 ans = 0;
    // 第一维分块
    for (i64 l1 = 1, r1; l1 <= n; l1 = r1 + 1) {
        r1 = n / (n / l1);
        i64 term1 = (n / l1) * (r1 - l1 + 1);

        // 第二维分块
        for (i64 l2 = 1, r2; l2 <= m; l2 = r2 + 1) {
            r2 = m / (m / l2);
            i64 term2 = (m / l2) * (r2 - l2 + 1);
            ans += term1 * term2;  // 乘积求和
        }
    }
    return ans;
}


/**************************************************************************
 * 5. GCD求和：ΣΣ gcd(i,j) (P2398)
 * 问题：计算 Σ_{i=1}^{n} Σ_{j=1}^{n} gcd(i,j)
 * 解法：欧拉函数 + 数论分块
 * 时间复杂度：O(n + √n)，空间复杂度：O(n)
 **************************************************************************/
i64 gcd_sum(i64 n) {
    // 预处理欧拉函数
    vector<i64> phi(n + 1);
    for (i64 i = 1; i <= n; i++) phi[i] = i;
    for (i64 i = 2; i <= n; i++) {
        if (phi[i] == i) {  // i是质数
            for (i64 j = i; j <= n; j += i) {
                phi[j] = phi[j] / i * (i - 1);
            }
        }
    }

    i64 ans = 0;
    // 公式：ΣΣ gcd(i,j) = Σ_{d=1}^{n} φ(d) * floor(n/d)²
    for (i64 d = 1; d <= n; d++) {
```

```
        i64 cnt = n / d;
        ans += phi[d] * cnt * cnt;
    }
    return ans;
}


/**************************************************************************
 * 6. 莫比乌斯反演预处理
 * 功能：线性筛求莫比乌斯函数 μ(n)
 * 性质：μ(1)=1，质数μ(p)=-1，平方因子μ(n)=0，其他情况有乘法性质
 * 时间复杂度：O(n)，空间复杂度：O(n)
 **************************************************************************/
vector<i64> mobius_sieve(i64 n) {
    vector<i64> mu(n + 1, 1);
    vector<bool> is_prime(n + 1, true);
    vector<i64> primes;

    // 线性筛求莫比乌斯函数
    for (i64 i = 2; i <= n; i++) {
        if (is_prime[i]) {
            primes.push_back(i);
            mu[i] = -1;  // 质数的μ值为-1
        }
        for (i64 p : primes) {
            if (i * p > n) break;
            is_prime[i * p] = false;
            if (i % p == 0) {
                mu[i * p] = 0;  // 有平方因子
                break;
            } else {
                mu[i * p] = -mu[i];  // 乘法性质
            }
        }
    }
    return mu;
}


/**************************************************************************
 * 7. GCD求和（数论分块优化版）
 * 问题：计算 Σ_{i=1}^{n} Σ_{j=1}^{n} gcd(i,j)
 * 解法：欧拉函数前缀和 + 数论分块
 * 时间复杂度：O(n + √n)，空间复杂度：O(n)
 **************************************************************************/
i64 gcd_sum_optimized(i64 n) {
    // 预处理欧拉函数和前缀和
    vector<i64> phi(n + 1);
    for (i64 i = 1; i <= n; i++) phi[i] = i;
    for (i64 i = 2; i <= n; i++) {
        if (phi[i] == i) {
            for (i64 j = i; j <= n; j += i) {
                phi[j] = phi[j] / i * (i - 1);
            }
        }
    }
```

```
    // 计算欧拉函数前缀和
    vector<i64> phi_prefix(n + 1, 0);
    for (i64 i = 1; i <= n; i++) {
        phi_prefix[i] = phi_prefix[i - 1] + phi[i];
    }

    i64 ans = 0;
    // 使用数论分块优化
    for (i64 l = 1, r; l <= n; l = r + 1) {
        r = n / (n / l);
        i64 cnt = n / l;
        i64 sum_phi = phi_prefix[r] - phi_prefix[l - 1];
        ans += sum_phi * cnt * cnt;
    }
    return ans;
}


/**************************************************************************
 * 8. 数论分块通用模板
 * 功能：计算 Σ_{i=1}^{n} f(floor(n/i)) * g(i)
 * 其中f是简单函数，g(i)在区间[l,r]上可以快速求和
 **************************************************************************/
template<typename Func1, typename Func2>
i64 number_theoretic_block(i64 n, Func1 f, Func2 interval_sum) {
    i64 ans = 0;
    for (i64 l = 1, r; l <= n; l = r + 1) {
        r = n / (n / l);
        i64 val = f(n / l);  // f(floor(n/i))的值
        i64 sum_g = interval_sum(l, r);  // g(i)在区间[l,r]上的和
        ans += val * sum_g;
    }
    return ans;
}
```

📊 数论分块原理

关键点	说明	时间复杂度
核心思想	floor(n/i) 只有 O(√n) 种不同的值	-
区间计算	相同值的区间 [l, r] 一起计算，其中 r = n / (n / l)	O(√n)
优化效果	从 O(n) 优化到 O(√n)	降低两个数量级

📊 常用数论公式

公式	数学表达	应用场景
基础分块	∑_{i=1}^{n} floor(n/i)	LS1261
加权分块	∑_{i=1}^{n} floor(n/i) * i	LS1230
余数求和	*∑{i=1}^{n} (k mod i) = nk* - ∑{i=1}^{n} floor(k/i) * i	P2261

公式	数学表达	应用场景
GCD求和	∑{i=1}^{n} ∑{j=1}^{n} gcd(i,j) = ∑_{d=1}^{n} φ(d) * floor(n/d)²	P2398
莫比乌斯反演	∑_{d|n} μ(d) = [n=1]	数论函数转换

📊 欧拉函数性质

性质	公式/说明	应用
定义	φ(n)：1~n中与n互质的数的个数	基础概念
积性函数	若 gcd(a,b)=1，则 φ(ab) = φ(a)φ(b)	分治计算
计算公式	φ(n) = n × ∏_{p|n} (1 - 1/p)	求值公式
质数性质	φ(p) = p-1，φ(p^k) = p^k - p^{k-1}	特殊情况
求和性质	∑_{d|n} φ(d) = n	莫比乌斯反演

📊 数论分块时间复杂度分析

问题类型	暴力复杂度	分块复杂度	优化倍数
基础分块	O(n)	O(√n)	O(√n)倍
二维分块	O(n²)	O(n)	O(n)倍
GCD求和	O(n²)	O(n)	O(n)倍
余数求和	O(n)	O(√k)	O(√n)倍

📊 关联题目索引

题目编号	题目名称	核心算法	难度
LS1261	数论分块	基础数论分块	⭐⭐
LS1262	多维数论分块	二维数论分块	⭐⭐⭐
P2261	余数求和	数论分块应用	⭐⭐
P2398	GCD求和	欧拉函数+数论分块	⭐⭐⭐
LS1230	简洁的数学	加权数论分块	⭐⭐

【算法特点】

算法	特点	适用场景
基础分块	最简单，理解数论分块原理	教学、入门
加权分块	结合等差数列求和	加权求和问题

算法	特点	适用场景
余数求和	实际应用，转化技巧	数学问题
GCD求和	综合性强，需要预处理	数论综合题
莫比乌斯	高级数论工具	反演、筛选

【学习建议】

1. **理解原理**：先理解为什么 floor(n/i) 只有 O(√n) 种值
2. **掌握模板**：熟练使用数论分块的标准模板
3. **学会转化**：将复杂问题转化为数论分块可解的形式
4. **结合数学**：掌握欧拉函数、莫比乌斯函数等数论知识
5. **实践验证**：用小数据暴力验证算法正确性

【常见错误】

1. **边界溢出**：忘记使用 i64，导致 int 溢出
2. **循环条件**：忘记处理 k < l 的情况
3. **区间计算**：等差数列求和公式写错
4. **预处理错误**：欧拉函数或莫比乌斯函数计算错误
5. **复杂度估计**：误以为所有数论分块都是 O(√n)

【扩展应用】

1. **三维分块**：∑∑∑ floor(n/i)floor(m/j)floor(k/l)
2. **带函数分块**：∑ f(floor(n/i)) * g(i)
3. **前缀和优化**：预处理前缀和加速区间求和
4. **积性函数**：结合积性函数性质进一步优化

【性能对比】

数据规模	暴力算法	数论分块	加速倍数
n=10^3	~1ms	~0.01ms	~100倍
n=10^4	~10ms	~0.05ms	~200倍
n=10^5	~100ms	~0.2ms	~500倍
n=10^6	~1s	~1ms	~1000倍
n=10^7	~10s	~3ms	~3000倍

【实用技巧】

1. **可视化理解**：画出 floor(n/i) 随 i 变化的图像
2. **对拍测试**：与小数据暴力解法对比结果
3. **模运算处理**：涉及模运算时注意取模时机
4. **记忆化搜索**：对重复查询的结果进行缓存

5. **分段处理**：对大数据采用分段预处理策略

5.2 筛法（课程14）

📦 核心代码模板

```
// ==================== 筛法系列算法 ====================

/**********************************************************************
 * 1. 埃氏筛法 - O(nloglogn) (LS1163)
 * 问题：筛选出1~n中的所有质数
 * 解法：标记每个质数的倍数，优化从i*i开始标记
 * 时间复杂度：O(nloglogn)，空间复杂度：O(n)
 **********************************************************************/
vector<i64> eratosthenes_sieve(i64 n) {
    vector<bool> is_prime(n + 1, true);
    vector<i64> primes;

    is_prime[0] = is_prime[1] = false;
    for (i64 i = 2; i <= n; i++) {
        if (is_prime[i]) {
            primes.push_back(i);
            // 从i*i开始标记，因为i*(i-1)已经被标记过了
            if (i * i <= n) {  // 防止i*i溢出
                for (i64 j = i * i; j <= n; j += i) {
                    is_prime[j] = false;
                }
            }
        }
    }
    return primes;
}


/**********************************************************************
 * 2. 欧拉筛（线性筛）- O(n) (LS1163)
 * 问题：线性时间复杂度筛选质数
 * 解法：每个合数只被其最小质因子标记一次
 * 时间复杂度：O(n)，空间复杂度：O(n)
 **********************************************************************/
vector<i64> euler_sieve(i64 n) {
    vector<bool> is_prime(n + 1, true);
    vector<i64> primes;

    is_prime[0] = is_prime[1] = false;
    for (i64 i = 2; i <= n; i++) {
        if (is_prime[i]) {
            primes.push_back(i);
        }
        // 用当前质数去标记合数
        for (i64 p : primes) {
            if (i * p > n) break;
            is_prime[i * p] = false;
            if (i % p == 0) break;  // 关键：保证每个合数只被最小质因子标记
```

```
        }
    }
    return primes;
}


/**************************************************************************
 * 3. 区间筛法 (LS1236)
 * 问题：筛选区间[l, r]内的所有质数（r-l ≤ 10⁶, r ≤ 10¹²）
 * 解法：先用筛法得到[2, √r]的质数，再用这些质数筛大区间
 * 时间复杂度：O((r-l)loglogr)，空间复杂度：O(√r + r-l)
 **************************************************************************/
vector<bool> segment_sieve(i64 l, i64 r) {
    i64 limit = sqrt(r) + 1;
    vector<bool> is_prime_small(limit + 1, true);
    vector<bool> is_prime_range(r - l + 1, true);

    if (l == 1) is_prime_range[0] = false;

    // 先筛出[2, √r]内的质数
    for (i64 i = 2; i <= limit; i++) {
        if (is_prime_small[i]) {
            // 标记小范围内的合数
            for (i64 j = i * i; j <= limit; j += i) {
                is_prime_small[j] = false;
            }

            // 标记大范围内的合数
            i64 start = max(i * i, (l + i - 1) / i * i);
            for (i64 j = start; j <= r; j += i) {
                is_prime_range[j - l] = false;
            }
        }
    }

    return is_prime_range;
}

// 区间筛法辅助函数：获取区间内的质数列表
vector<i64> get_primes_in_range(i64 l, i64 r) {
    vector<bool> is_prime = segment_sieve(l, r);
    vector<i64> primes;

    for (i64 i = 0; i <= r - l; i++) {
        if (is_prime[i]) {
            primes.push_back(l + i);
        }
    }
    return primes;
}


/**************************************************************************
 * 4. 筛法求欧拉函数 (LS1164)
 * 问题：计算1~n所有数的欧拉函数φ(n)
 * 解法：在线性筛过程中同时计算欧拉函数
```

```
 * 时间复杂度：O(n)，空间复杂度：O(n)
 ******************************************************************************/
vector<i64> euler_phi_sieve(i64 n) {
    vector<i64> phi(n + 1);
    vector<bool> is_prime(n + 1, true);
    vector<i64> primes;

    // 初始化：φ(1)=1
    phi[1] = 1;

    for (i64 i = 2; i <= n; i++) {
        if (is_prime[i]) {
            primes.push_back(i);
            phi[i] = i - 1;  // 质数的φ值
        }
        for (i64 p : primes) {
            if (i * p > n) break;
            is_prime[i * p] = false;

            if (i % p == 0) {
                // p是i的质因子：φ(i*p) = φ(i) * p
                phi[i * p] = phi[i] * p;
                break;
            } else {
                // p与i互质：φ(i*p) = φ(i) * (p-1)
                phi[i * p] = phi[i] * (p - 1);
            }
        }
    }
    return phi;
}


/******************************************************************************
 * 5. 筛法求约数个数 (LS1164)
 * 问题：计算1~n所有数的约数个数d(n)
 * 解法：利用约数个数公式和线性筛
 * 时间复杂度：O(n)，空间复杂度：O(n)
 ******************************************************************************/
vector<i64> divisor_count_sieve(i64 n) {
    vector<i64> d(n + 1, 1);           // 约数个数
    vector<i64> min_factor_cnt(n + 1, 0);  // 最小质因子的次数
    vector<i64> primes;

    for (i64 i = 2; i <= n; i++) {
        if (min_factor_cnt[i] == 0) {  // i是质数
            primes.push_back(i);
            d[i] = 2;                  // 质数的约数个数为2
            min_factor_cnt[i] = 1;
        }

        for (i64 p : primes) {
            if (i * p > n) break;

            if (i % p == 0) {
                // p是i的最小质因子
```

```
                min_factor_cnt[i * p] = min_factor_cnt[i] + 1;
                // 约数个数公式：d(n) = Π(α_i+1)
                d[i * p] = d[i] / (min_factor_cnt[i] + 1) * (min_factor_cnt[i *
p] + 1);
                break;
            } else {
                // p与i互质
                min_factor_cnt[i * p] = 1;
                d[i * p] = d[i] * d[p];
            }
        }
    }
    return d;
}


/**********************************************************************
 * 6. 筛法求约数和 (LS1164)
 * 问题：计算1~n所有数的约数和σ(n)
 * 解法：利用约数和公式和线性筛
 * 时间复杂度：O(n)，空间复杂度：O(n)
 **********************************************************************/
vector<i64> divisor_sum_sieve(i64 n) {
    vector<i64> sigma(n + 1, 1);            // 约数和
    vector<i64> min_factor_pow(n + 1, 1);  // 最小质因子的p^α
    vector<i64> primes;

    for (i64 i = 2; i <= n; i++) {
        if (min_factor_pow[i] == 1) {  // i是质数
            primes.push_back(i);
            sigma[i] = i + 1;          // 质数的约数和：1 + i
            min_factor_pow[i] = i;
        }

        for (i64 p : primes) {
            if (i * p > n) break;

            if (i % p == 0) {
                // p是i的最小质因子
                min_factor_pow[i * p] = min_factor_pow[i] * p;
                // 约数和公式：σ(n) = Π(1 + p + p² + ... + p^α)
                sigma[i * p] = sigma[i] * (min_factor_pow[i * p] * p - 1) /
(min_factor_pow[i] - 1);
                break;
            } else {
                // p与i互质
                min_factor_pow[i * p] = p;
                sigma[i * p] = sigma[i] * sigma[p];
            }
        }
    }
    return sigma;
}


/**********************************************************************
```

```
 * 7. 筛法求莫比乌斯函数 (补充)
 * 问题：计算1~n所有数的莫比乌斯函数μ(n)
 * 解法：在线性筛过程中同时计算莫比乌斯函数
 * 时间复杂度：O(n)，空间复杂度：O(n)
 ******************************************************************************/
vector<i64> mobius_sieve(i64 n) {
    vector<i64> mu(n + 1, 1);
    vector<bool> is_prime(n + 1, true);
    vector<i64> primes;

    mu[1] = 1;
    is_prime[0] = is_prime[1] = false;

    for (i64 i = 2; i <= n; i++) {
        if (is_prime[i]) {
            primes.push_back(i);
            mu[i] = -1;  // 质数的μ值为-1
        }
        for (i64 p : primes) {
            if (i * p > n) break;
            is_prime[i * p] = false;

            if (i % p == 0) {
                mu[i * p] = 0;  // 有平方因子
                break;
            } else {
                mu[i * p] = -mu[i];  // 乘法性质
            }
        }
    }
    return mu;
}


/******************************************************************************
 * 8. 筛法预处理质因数分解 (补充)
 * 问题：预处理1~n所有数的最小质因子，用于快速质因数分解
 * 解法：线性筛记录最小质因子
 * 时间复杂度：O(n)，空间复杂度：O(n)
 ******************************************************************************/
vector<i64> min_prime_factor_sieve(i64 n) {
    vector<i64> min_factor(n + 1, 0);
    vector<i64> primes;

    for (i64 i = 2; i <= n; i++) {
        if (min_factor[i] == 0) {  // i是质数
            min_factor[i] = i;
            primes.push_back(i);
        }

        for (i64 p : primes) {
            if (p > min_factor[i] || i * p > n) break;
            min_factor[i * p] = p;
        }
    }
    return min_factor;
```

```
}

// 快速质因数分解（利用预处理的最小质因子）
vector<pair<i64, i64>> prime_factorization(i64 x, const vector<i64>& min_factor)
{
    vector<pair<i64, i64>> factors;

    while (x > 1) {
        i64 p = min_factor[x];
        i64 cnt = 0;
        while (x % p == 0) {
            x /= p;
            cnt++;
        }
        factors.emplace_back(p, cnt);
    }

    return factors;
}
```

📊 筛法对比

筛法类型	时间复杂度	空间复杂度	特点	适用场景
埃氏筛	O(nloglogn)	O(n)	简单易实现，常数小	$n \le 10^7$
欧拉筛	O(n)	O(n)	线性时间，每个数只筛一次	$n \le 10^7$，需要线性时间
区间筛	O((r-l)loglogr)	O(√r + r-l)	处理大区间，内存友好	$r-l \le 10^6$, $r \le 10^{12}$
积性函数筛	O(n)	O(n)	同时计算多种积性函数	需要多个积性函数值

📊 积性函数筛法

函数	符号	定义	筛法公式
欧拉函数	φ(n)	1~n中与n互质的数的个数	φ(p^k) = p^k - p^{k-1}
约数个数	d(n)	n的正因子个数	d(p^k) = k+1
约数和	σ(n)	n的所有正因子之和	σ(p^k) = (p^{k+1}-1)/(p-1)
莫比乌斯函数	μ(n)	数论反演函数	μ(1)=1, μ(p)=-1, $\mu(p^2)=0$

📊 积性函数公式总结

函数	公式	说明
欧拉函数	φ(n) = n × ∏_{p | n} (1 - 1/p)	乘积形式
约数个数	d(n) = ∏{*i=1*}^*k* (*α*i + 1)	n = ∏ p_i^{α_i}

函数	公式	说明
约数和	σ(n) = ∏{i=1}^k (1 + p_i + p_i² + ... + p_i^{αi})	等比数列求和
莫比乌斯函数	μ(n) = 1 (n=1), (-1)^k (n无平方因子), 0 (其他)	定义式

📊 算法性能对比

数据规模	埃氏筛时间	欧拉筛时间	加速比	内存使用
n=10⁶	~10ms	~20ms	0.5×	~1MB
n=10⁷	~100ms	~200ms	0.5×	~10MB
n=10⁸	~1s	~2s	0.5×	~100MB
n=10⁹	内存不足	内存不足	-	~1GB

📊 关联题目索引

题目编号	题目名称	核心算法	难度
LS1163	埃氏筛和欧拉筛	基础筛法	⭐⭐
LS1236	区间筛法	大区间质数筛选	⭐⭐⭐
LS1164	约数个数、约数和、欧拉函数	积性函数筛法	⭐⭐⭐
LS1231	连续的自然数	筛法应用	⭐⭐
LS1237	最大公约数计数	欧拉函数应用	⭐⭐⭐

【算法特点】

算法	特点	适用场景
埃氏筛	实现简单，常数小	快速获取质数列表
欧拉筛	线性时间，可扩展	需要积性函数或线性时间
区间筛	内存友好，处理大数	大区间质数筛选
积性函数筛	多功能，效率高	需要多种数论函数

【学习建议】

1. **理解原理**：先理解筛法的基本思想（标记倍数）
2. **对比学习**：比较埃氏筛和欧拉筛的区别
3. **掌握优化**：学习各种筛法的优化技巧
4. **实践应用**：解决实际问题，理解筛法的应用场景
5. **扩展思考**：思考如何筛其他数论函数

【常见错误】

1. **边界溢出**：`ii` *可能溢出，需要判断* `ii <= n`

2. **内存不足**：使用 vector 而非 vector 节省内存
3. **区间计算**：区间筛法的起始点计算错误
4. **质数判断**：忘记处理 0 和 1 不是质数
5. **数据类型**：使用 int 导致大数溢出

【扩展应用】

1. **质数间隔**：计算相邻质数的最大间隔
2. **质数分布**：研究质数在不同区间的分布
3. **质数测试**：结合筛法实现 Miller-Rabin 测试
4. **因数分解**：利用筛法预处理进行快速质因数分解
5. **数论函数**：计算更多的积性函数（如 Liouville 函数）

【性能优化】

1. **内存优化**：使用 bitset 进一步减少内存
2. **分段筛法**：对超大范围使用分段处理
3. **并行计算**：多线程并行筛不同区间
4. **缓存优化**：利用 CPU 缓存局部性
5. **编译优化**：使用编译器优化选项

【实用技巧】

1. **预计算**：预先计算常用范围内的筛法结果
2. **惰性计算**：需要时才计算，节省内存
3. **混合方法**：小范围用筛法，大范围用概率方法
4. **验证测试**：用小数据暴力验证算法正确性
5. **性能分析**：使用性能分析工具优化热点代码

5.3 数学与数论（课程06, 14）

📦 核心代码模板

```
// ==================== 数学与数论 ====================

/*******************************************************************************
 * 1. 快速幂算法 (LS1024)
 * 问题：计算 base^exp mod m，高效处理大指数
 * 解法：二进制分解指数，利用平方降次
 * 时间复杂度：O(log exp)，空间复杂度：O(1)
 *******************************************************************************/
i64 fast_pow(i64 base, i64 exp, i64 mod = LLONG_MAX) {
    if (mod == 1) return 0;  // 任何数模1都为0
    if (exp < 0) return 0;   // 负指数处理（可根据需求调整）

    base %= mod;
    i64 result = 1;
```

```
    while (exp > 0) {
        // 如果当前二进制位为1，乘上对应的base
        if (exp & 1) {
            result = (result * base) % mod;
        }
        // base平方，准备下一轮
        base = (base * base) % mod;
        // 指数右移一位
        exp >>= 1;
    }

    return result;
}

// 快速幂迭代版本（更易理解）
i64 fast_pow_iterative(i64 base, i64 exp, i64 mod = LLONG_MAX) {
    if (mod == 1) return 0;

    base %= mod;
    i64 result = 1;

    for (; exp > 0; exp >>= 1) {
        if (exp & 1) {
            result = (result * base) % mod;
        }
        base = (base * base) % mod;
    }

    return result;
}

// 快速幂递归版本
i64 fast_pow_recursive(i64 base, i64 exp, i64 mod = LLONG_MAX) {
    if (mod == 1) return 0;
    if (exp == 0) return 1 % mod;
    if (exp == 1) return base % mod;

    base %= mod;
    i64 half = fast_pow_recursive(base, exp / 2, mod);
    i64 result = (half * half) % mod;

    if (exp & 1) {
        result = (result * base) % mod;
    }

    return result;
}

// 快速幂求等比数列和：a^0 + a^1 + ... + a^(n-1) mod m
i64 geometric_series_sum(i64 a, i64 n, i64 mod) {
    if (n == 0) return 0;
    if (n == 1) return 1 % mod;

    if (n % 2 == 1) {
        // 奇数项：S(n) = S(n-1) + a^(n-1)
        i64 sum = geometric_series_sum(a, n - 1, mod);
```

```
        i64 last = fast_pow(a, n - 1, mod);
        return (sum + last) % mod;
    } else {
        // 偶数项：S(n) = S(n/2) * (1 + a^(n/2))
        i64 half_sum = geometric_series_sum(a, n / 2, mod);
        i64 a_half = fast_pow(a, n / 2, mod);
        i64 factor = (1 + a_half) % mod;
        return (half_sum * factor) % mod;
    }
}


/**************************************************************************
 * 2. 矩阵快速幂 - 斐波那契数列 (LS1156)
 * 问题：高效计算斐波那契数列第n项（n可以很大）
 * 解法：利用矩阵快速幂加速线性递推
 * 时间复杂度：O(log n)，空间复杂度：O(1)
 **************************************************************************/
// 2x2矩阵乘法（模运算）
vector<vector<i64>> matrix_multiply_2x2(const vector<vector<i64>>& a,
                                        const vector<vector<i64>>& b,
                                        i64 mod) {
    vector<vector<i64>> result(2, vector<i64>(2, 0));

    result[0][0] = (a[0][0] * b[0][0] + a[0][1] * b[1][0]) % mod;
    result[0][1] = (a[0][0] * b[0][1] + a[0][1] * b[1][1]) % mod;
    result[1][0] = (a[1][0] * b[0][0] + a[1][1] * b[1][0]) % mod;
    result[1][1] = (a[1][0] * b[0][1] + a[1][1] * b[1][1]) % mod;

    return result;
}

// 通用矩阵乘法
vector<vector<i64>> matrix_multiply(const vector<vector<i64>>& a,
                                    const vector<vector<i64>>& b,
                                    i64 mod) {
    i64 n = a.size();
    i64 m = b[0].size();
    i64 p = b.size();

    vector<vector<i64>> result(n, vector<i64>(m, 0));

    for (i64 i = 0; i < n; i++) {
        for (i64 j = 0; j < m; j++) {
            for (i64 k = 0; k < p; k++) {
                result[i][j] = (result[i][j] + a[i][k] * b[k][j]) % mod;
            }
        }
    }

    return result;
}

// 矩阵快速幂
vector<vector<i64>> matrix_pow(vector<vector<i64>> base, i64 exp, i64 mod) {
    i64 n = base.size();
```

```
    // 单位矩阵
    vector<vector<i64>> result(n, vector<i64>(n, 0));
    for (i64 i = 0; i < n; i++) {
        result[i][i] = 1;
    }

    while (exp > 0) {
        if (exp & 1) {
            result = matrix_multiply(result, base, mod);
        }
        base = matrix_multiply(base, base, mod);
        exp >>= 1;
    }

    return result;
}

// 斐波那契数列矩阵解法：F(n) = [[1,1],[1,0]]^(n-1) * [F(1),F(0)]
i64 fibonacci_matrix(i64 n, i64 mod = LLONG_MAX) {
    if (n <= 0) return 0;
    if (n == 1) return 1 % mod;

    vector<vector<i64>> base = {{1, 1},
                                {1, 0}};
    vector<vector<i64>> result = matrix_pow(base, n - 1, mod);

    // 结果矩阵的第一行第一列就是F(n)
    return result[0][0] % mod;
}

// 斐波那契数列快速倍增法（无需矩阵）
pair<i64, i64> fibonacci_fast_doubling(i64 n, i64 mod) {
    if (n == 0) return {0, 1 % mod};

    auto [a, b] = fibonacci_fast_doubling(n >> 1, mod);
    i64 c = (a * ((2 * b - a + mod) % mod)) % mod;
    i64 d = (a * a + b * b) % mod;

    if (n & 1) {
        return {d, (c + d) % mod};
    } else {
        return {c, d};
    }
}

i64 fibonacci_fast_doubling_wrapper(i64 n, i64 mod = LLONG_MAX) {
    if (n <= 0) return 0;
    return fibonacci_fast_doubling(n, mod).first;
}


/*************************************************************************
 * 3. 扩展欧几里得算法 (LS1220)
 * 问题：求解 ax + by = gcd(a,b) 的整数解
 * 解法：递归应用欧几里得算法
 * 时间复杂度：O(log min(a,b))，空间复杂度：O(log min(a,b))
```

```
 ********************************************************************************/
// 扩展欧几里得算法（递归版）
tuple<i64, i64, i64> extended_gcd(i64 a, i64 b) {
    if (b == 0) {
        return {a, 1, 0};  // gcd(a,0) = a, 系数为(1,0)
    }

    auto [g, x1, y1] = extended_gcd(b, a % b);
    i64 x = y1;
    i64 y = x1 - (a / b) * y1;

    return {g, x, y};
}

// 扩展欧几里得算法（迭代版）
tuple<i64, i64, i64> extended_gcd_iterative(i64 a, i64 b) {
    i64 x = 1, y = 0;      // 系数 for a
    i64 x1 = 0, y1 = 1;    // 系数 for b
    i64 x2, y2;

    while (b != 0) {
        i64 q = a / b;

        // 更新系数
        x2 = x - q * x1;
        y2 = y - q * y1;

        // 更新a,b
        i64 temp = b;
        b = a % b;
        a = temp;

        // 更新系数
        x = x1; y = y1;
        x1 = x2; y1 = y2;
    }

    return {a, x, y};
}

// 求解线性同余方程 ax ≡ b (mod m)
pair<bool, i64> solve_linear_congruence(i64 a, i64 b, i64 m) {
    auto [g, x, y] = extended_gcd(a, m);

    if (b % g != 0) {
        return {false, 0};  // 无解
    }

    i64 x0 = (x * (b / g)) % m;
    // 通解: x ≡ x0 + k*(m/g) (mod m), k=0,1,...,g-1
    return {true, (x0 % m + m) % m};
}

// 求解模逆元: a^(-1) mod m, 要求gcd(a,m)=1
i64 mod_inverse(i64 a, i64 m) {
    auto [g, x, y] = extended_gcd(a, m);
```

```cpp
    if (g != 1) {
        return -1;  // 逆元不存在
    }
    return (x % m + m) % m;
}

// 快速求模逆元（费马小定理，要求m为质数）
i64 mod_inverse_fermat(i64 a, i64 m) {
    // a^(m-2) ≡ a^(-1) (mod m) 当m为质数
    return fast_pow(a, m - 2, m);
}


/*****************************************************************************
 * 4. 中国剩余定理 (CRT)
 * 问题：求解同余方程组 x ≡ a_i (mod m_i)
 * 解法：逐步合并同余方程
 * 时间复杂度：O(n log M)，空间复杂度：O(n)
 *****************************************************************************/
pair<i64, i64> chinese_remainder_theorem(vector<i64>& a, vector<i64>& m) {
    i64 n = a.size();
    if (n != m.size() || n == 0) {
        return {-1, -1};  // 无效输入
    }

    i64 x = 0;  // 当前解
    i64 M = 1;  // 当前模数的乘积

    for (i64 i = 0; i < n; i++) {
        // 求解 M_i * t ≡ a_i - x (mod m_i)
        auto [g, t, _] = extended_gcd(M, m[i]);

        if ((a[i] - x) % g != 0) {
            return {-1, -1};  // 无解
        }

        // 计算特解
        i64 t0 = ((a[i] - x) / g * t) % (m[i] / g);
        if (t0 < 0) t0 += m[i] / g;

        // 更新解
        x = x + M * t0;
        M = M * m[i] / g;
        x = (x % M + M) % M;
    }

    return {x, M};  // 通解：x + k*M
}

// 扩展中国剩余定理（不要求模数互质）
pair<i64, i64> extended_crt(vector<i64>& a, vector<i64>& m) {
    i64 n = a.size();
    i64 last_a = a[0], last_m = m[0];

    for (i64 i = 1; i < n; i++) {
        // 合并方程：x ≡ last_a (mod last_m) 和 x ≡ a[i] (mod m[i])
```

```
        auto [g, x, y] = extended_gcd(last_m, m[i]);

        if ((a[i] - last_a) % g != 0) {
            return {-1, -1};  // 无解
        }

        i64 lcm = last_m / g * m[i];
        i64 t = ((a[i] - last_a) / g * x) % (m[i] / g);

        last_a = (last_a + last_m * t) % lcm;
        if (last_a < 0) last_a += lcm;

        last_m = lcm;
    }

    return {last_a, last_m};
}


/**************************************************************************
 * 5. 组合数计算 - 预处理阶乘和逆元
 * 问题：高效计算组合数 C(n,k) mod p
 * 解法：预处理阶乘和阶乘逆元，O(1)查询
 * 时间复杂度：预处理O(n)，查询O(1)，空间复杂度：O(n)
 **************************************************************************/
class Combinatorics {
private:
    vector<i64> fac;      // 阶乘数组
    vector<i64> inv_fac;  // 阶乘逆元数组
    vector<i64> inv;      // 逆元数组（可选）
    i64 mod;
    i64 max_n;

    // 预处理阶乘
    void precompute_factorial(i64 n) {
        fac.resize(n + 1);
        fac[0] = 1;
        for (i64 i = 1; i <= n; i++) {
            fac[i] = (fac[i - 1] * i) % mod;
        }
    }

    // 预处理阶乘逆元
    void precompute_inverse_factorial(i64 n) {
        inv_fac.resize(n + 1);
        // 费马小定理求 n! 的逆元
        inv_fac[n] = fast_pow(fac[n], mod - 2, mod);

        // 递推求逆元：i!^(-1) = (i+1)!^(-1) * (i+1)
        for (i64 i = n - 1; i >= 0; i--) {
            inv_fac[i] = (inv_fac[i + 1] * (i + 1)) % mod;
        }
    }

    // 预处理1到n的逆元（线性方法）
    void precompute_inverse(i64 n) {
```

```
        inv.resize(n + 1);
        inv[1] = 1;
        for (i64 i = 2; i <= n; i++) {
            inv[i] = (mod - mod / i) * inv[mod % i] % mod;
        }
    }

public:
    // 构造函数，预处理到n
    Combinatorics(i64 n, i64 p) : mod(p), max_n(n) {
        // 模数必须是质数
        precompute_factorial(n);
        precompute_inverse_factorial(n);
        precompute_inverse(n);  // 可选
    }

    // 组合数 C(n,k) = n! / (k! * (n-k)!)
    i64 C(i64 n, i64 k) {
        if (k < 0 || k > n) return 0;
        if (n > max_n) {
            // 如果n超出预处理范围，使用其他方法
            return C_small(n, k);
        }
        return fac[n] * inv_fac[k] % mod * inv_fac[n - k] % mod;
    }

    // 排列数 P(n,k) = n! / (n-k)!
    i64 P(i64 n, i64 k) {
        if (k < 0 || k > n) return 0;
        if (n > max_n) {
            i64 result = 1;
            for (i64 i = 0; i < k; i++) {
                result = (result * (n - i)) % mod;
            }
            return result;
        }
        return fac[n] * inv_fac[n - k] % mod;
    }

    // 阶乘 n!
    i64 factorial(i64 n) {
        if (n > max_n) {
            i64 result = 1;
            for (i64 i = 2; i <= n; i++) {
                result = (result * i) % mod;
            }
            return result;
        }
        return fac[n];
    }

    // 逆元 i^(-1)
    i64 inverse(i64 i) {
        if (i <= max_n) {
            return inv[i];
        }
```

```
        return mod_inverse(i, mod);
    }

    // 阶乘逆元 (n!)^(-1)
    i64 inverse_factorial(i64 n) {
        if (n > max_n) {
            return mod_inverse(factorial(n), mod);
        }
        return inv_fac[n];
    }

    // 小范围组合数计算（当n不大时）
    i64 C_small(i64 n, i64 k) {
        if (k < 0 || k > n) return 0;
        if (k > n - k) k = n - k;  // 利用对称性

        i64 numerator = 1, denominator = 1;
        for (i64 i = 1; i <= k; i++) {
            numerator = (numerator * (n - k + i)) % mod;
            denominator = (denominator * i) % mod;
        }

        return numerator * mod_inverse(denominator, mod) % mod;
    }

    // 卢卡斯定理（当n很大但mod较小时）
    i64 C_lucas(i64 n, i64 k) {
        if (k < 0 || k > n) return 0;
        if (k == 0) return 1;
        return C(n % mod, k % mod) * C_lucas(n / mod, k / mod) % mod;
    }
};


/**************************************************************************
 * 6. 质因数分解 (LS1233)
 * 问题：将整数分解为质因数的乘积
 * 解法：试除法、Pollard-Rho等
 * 时间复杂度：O(√n)（试除法），空间复杂度：O(log n)
 **************************************************************************/
// 试除法质因数分解
vector<pair<i64, i64>> prime_factorization_trial(i64 n) {
    vector<pair<i64, i64>> factors;

    // 处理2的因子
    if (n % 2 == 0) {
        i64 cnt = 0;
        while (n % 2 == 0) {
            n /= 2;
            cnt++;
        }
        factors.push_back({2, cnt});
    }

    // 处理奇数因子
    for (i64 i = 3; i * i <= n; i += 2) {
```

```
        if (n % i == 0) {
            i64 cnt = 0;
            while (n % i == 0) {
                n /= i;
                cnt++;
            }
            factors.push_back({i, cnt});
        }
    }

    // 处理剩余的大质数
    if (n > 1) {
        factors.push_back({n, 1});
    }

    return factors;
}

// 预处理最小质因子（埃氏筛优化）
vector<i64> sieve_min_prime_factor(i64 n) {
    vector<i64> min_factor(n + 1, 0);

    for (i64 i = 2; i <= n; i++) {
        if (min_factor[i] == 0) {  // i是质数
            min_factor[i] = i;
            if (i * i <= n) {
                for (i64 j = i * i; j <= n; j += i) {
                    if (min_factor[j] == 0) {
                        min_factor[j] = i;
                    }
                }
            }
        }
    }

    return min_factor;
}

// 快速质因数分解（使用最小质因子数组）
vector<pair<i64, i64>> fast_factorization(i64 n, const vector<i64>& min_factor) {
    vector<pair<i64, i64>> factors;

    while (n > 1) {
        i64 p = min_factor[n];
        i64 cnt = 0;

        while (n % p == 0) {
            n /= p;
            cnt++;
        }

        factors.push_back({p, cnt});
    }

    return factors;
}
```

```cpp
// 获取n的所有因数
vector<i64> get_all_divisors(i64 n) {
    vector<i64> divisors;

    for (i64 i = 1; i * i <= n; i++) {
        if (n % i == 0) {
            divisors.push_back(i);
            if (i != n / i) {
                divisors.push_back(n / i);
            }
        }
    }

    sort(divisors.begin(), divisors.end());
    return divisors;
}

// 计算n的因数个数
i64 count_divisors(i64 n) {
    auto factors = prime_factorization_trial(n);
    i64 count = 1;

    for (const auto& [p, exp] : factors) {
        count *= (exp + 1);
    }

    return count;
}

// 计算n的所有因数和
i64 sum_of_divisors(i64 n) {
    auto factors = prime_factorization_trial(n);
    i64 sum = 1;

    for (const auto& [p, exp] : factors) {
        i64 term = 1;
        i64 power = 1;
        for (i64 i = 0; i <= exp; i++) {
            term += power;
            power *= p;
        }
        sum *= term;
    }

    return sum;
}


/*****************************************************************************
 * 7. 素数筛法
 * 问题：高效找出一定范围内的所有素数
 * 解法：埃氏筛、欧拉筛（线性筛）
 * 时间复杂度：O(n log log n)（埃氏筛），O(n)（欧拉筛）
 *****************************************************************************/
// 埃拉托斯特尼筛法（埃氏筛）
```

```cpp
vector<i64> sieve_of_eratosthenes(i64 n) {
    vector<bool> is_prime(n + 1, true);
    vector<i64> primes;

    is_prime[0] = is_prime[1] = false;

    for (i64 i = 2; i * i <= n; i++) {
        if (is_prime[i]) {
            for (i64 j = i * i; j <= n; j += i) {
                is_prime[j] = false;
            }
        }
    }

    for (i64 i = 2; i <= n; i++) {
        if (is_prime[i]) {
            primes.push_back(i);
        }
    }

    return primes;
}

// 欧拉筛（线性筛）
vector<i64> euler_sieve(i64 n) {
    vector<bool> is_prime(n + 1, true);
    vector<i64> primes;

    is_prime[0] = is_prime[1] = false;

    for (i64 i = 2; i <= n; i++) {
        if (is_prime[i]) {
            primes.push_back(i);
        }

        for (i64 j = 0; j < primes.size() && i * primes[j] <= n; j++) {
            is_prime[i * primes[j]] = false;
            if (i % primes[j] == 0) {
                break;
            }
        }
    }

    return primes;
}

// 分段筛（用于大范围素数筛选）
vector<i64> segmented_sieve(i64 L, i64 R) {
    i64 limit = sqrt(R);
    vector<i64> primes = sieve_of_eratosthenes(limit);
    vector<bool> is_prime(R - L + 1, true);

    if (L == 1) is_prime[0] = false;

    for (i64 p : primes) {
        i64 start = max(p * p, (L + p - 1) / p * p);
```

```
        for (i64 j = start; j <= R; j += p) {
            is_prime[j - L] = false;
        }
    }

    vector<i64> result;
    for (i64 i = 0; i < is_prime.size(); i++) {
        if (is_prime[i]) {
            result.push_back(L + i);
        }
    }

    return result;
}


/**************************************************************************
 * 8. 欧拉函数
 * 问题：计算小于等于n且与n互质的数的个数
 * 解法：利用质因数分解公式 φ(n) = n Π(1 - 1/p)
 * 时间复杂度：O(√n)，空间复杂度：O(log n)
 **************************************************************************/
// 单个数的欧拉函数
i64 euler_phi(i64 n) {
    if (n <= 0) return 0;

    i64 result = n;
    i64 temp = n;

    // 处理2的因子
    if (temp % 2 == 0) {
        result -= result / 2;
        while (temp % 2 == 0) {
            temp /= 2;
        }
    }

    // 处理奇数因子
    for (i64 i = 3; i * i <= temp; i += 2) {
        if (temp % i == 0) {
            result -= result / i;
            while (temp % i == 0) {
                temp /= i;
            }
        }
    }

    // 处理剩余的大质数
    if (temp > 1) {
        result -= result / temp;
    }

    return result;
}
```

```cpp
// 预处理1到n的欧拉函数（线性筛法）
vector<i64> euler_phi_sieve(i64 n) {
    vector<i64> phi(n + 1);
    vector<bool> is_prime(n + 1, true);
    vector<i64> primes;

    phi[0] = 0;
    phi[1] = 1;

    for (i64 i = 2; i <= n; i++) {
        if (is_prime[i]) {
            primes.push_back(i);
            phi[i] = i - 1;  // 质数的欧拉函数为i-1
        }

        for (i64 j = 0; j < primes.size() && i * primes[j] <= n; j++) {
            is_prime[i * primes[j]] = false;

            if (i % primes[j] == 0) {
                // i包含primes[j]因子
                phi[i * primes[j]] = phi[i] * primes[j];
                break;
            } else {
                // i和primes[j]互质
                phi[i * primes[j]] = phi[i] * (primes[j] - 1);
            }
        }
    }

    return phi;
}

// 欧拉定理：a^φ(m) ≡ 1 (mod m)，当gcd(a,m)=1
i64 euler_theorem_pow(i64 a, i64 exp, i64 m) {
    i64 phi_m = euler_phi(m);
    exp %= phi_m;
    return fast_pow(a, exp, m);
}


/**************************************************************************
 * 9. 模运算工具类
 * 提供常用的模运算辅助函数
 **************************************************************************/
class ModularArithmetic {
public:
    i64 mod;

    ModularArithmetic(i64 m) : mod(m) {}

    // 模加法
    i64 add(i64 a, i64 b) {
        return (a % mod + b % mod) % mod;
    }

    // 模减法
```

```
    i64 sub(i64 a, i64 b) {
        return (a % mod - b % mod + mod) % mod;
    }

    // 模乘法
    i64 mul(i64 a, i64 b) {
        return (a % mod * b % mod) % mod;
    }

    // 模除法（需要逆元存在）
    i64 div(i64 a, i64 b) {
        i64 inv_b = mod_inverse(b, mod);
        if (inv_b == -1) {
            return -1;  // 逆元不存在
        }
        return mul(a, inv_b);
    }

    // 模幂运算
    i64 pow(i64 a, i64 exp) {
        return fast_pow(a, exp, mod);
    }

    // 模运算下的组合数
    i64 C(i64 n, i64 k) {
        if (k < 0 || k > n) return 0;

        i64 numerator = 1;
        i64 denominator = 1;

        for (i64 i = 1; i <= k; i++) {
            numerator = mul(numerator, n - k + i);
            denominator = mul(denominator, i);
        }

        return div(numerator, denominator);
    }

    // 模运算下的排列数
    i64 P(i64 n, i64 k) {
        if (k < 0 || k > n) return 0;

        i64 result = 1;
        for (i64 i = 0; i < k; i++) {
            result = mul(result, n - i);
        }
        return result;
    }
};


/**************************************************************************
 * 10. 数论分块
 * 问题：高效计算 Σ_{i=1}^{n} floor(n/i)
 * 解法：利用n/i的值连续相等的性质分块计算
 * 时间复杂度：O(√n)，空间复杂度：O(1)
```

```
 ******************************************************************************/
// 计算 Σ_{i=1}^{n} floor(n/i)
i64 floor_sum(i64 n) {
    i64 result = 0;
    i64 i = 1;

    while (i <= n) {
        i64 q = n / i;
        i64 j = n / q;  // 使得 floor(n/k) = q 的最大k
        result += q * (j - i + 1);
        i = j + 1;
    }

    return result;
}

// 计算 Σ_{i=1}^{n} floor(n/i) * i
i64 floor_sum_weighted(i64 n) {
    i64 result = 0;
    i64 i = 1;

    while (i <= n) {
        i64 q = n / i;
        i64 j = n / q;
        // 等差数列求和: i + (i+1) + ... + j
        i64 sum_i_to_j = (i + j) * (j - i + 1) / 2;
        result += q * sum_i_to_j;
        i = j + 1;
    }

    return result;
}

// 计算 Σ_{i=1}^{n} floor(n/i) * f(i)，其中f(i)可以自定义
i64 floor_sum_with_func(i64 n, function<i64(i64)> f) {
    i64 result = 0;
    i64 i = 1;

    while (i <= n) {
        i64 q = n / i;
        i64 j = n / q;
        // 计算f(i)到f(j)的和（需要前缀和优化）
        i64 sum_f = 0;
        for (i64 k = i; k <= j; k++) {
            sum_f += f(k);
        }
        result += q * sum_f;
        i = j + 1;
    }

    return result;
}
```

📊 算法复杂度总结

算法	时间复杂度	空间复杂度	适用场景
快速幂	O(log n)	O(1)	大指数取模
矩阵快速幂	O(k^3 log n)	O(k^2)	线性递推加速
扩展欧几里得	O(log min(a,b))	O(log min(a,b))	求解不定方程
中国剩余定理	O(n log M)	O(n)	同余方程组
组合数预处理	O(n)预处理，O(1)查询	O(n)	频繁组合数查询
质因数分解	O(√n)	O(log n)	单个数分解
埃氏筛	O(n log log n)	O(n)	素数筛选
欧拉筛	O(n)	O(n)	高效素数筛选
欧拉函数	O(√n)	O(1)	单个φ(n)计算
预处理φ数组	O(n)	O(n)	批量φ(n)计算
数论分块	O(√n)	O(1)	∑ floor(n/i) 类求和

📊 数学公式与定理

名称	公式/定理	应用
快速幂	a^b = ($a^{(b/2)}$)2 （b偶）	大数取模
矩阵快速幂	F(n) = M^(n-1) × F(1)	线性递推
扩展欧几里得	ax + by = gcd(a,b)	模逆元、同余方程
中国剩余定理	x ≡ a_i (mod m_i) → 唯一解模M	同余方程组
组合数	C(n,k) = n!/(k!(n-k)!)	计数问题
质因数分解	n = ∏ p_i^{α_i}	因数、gcd、lcm
欧拉函数	φ(n) = n ∏ (1 - 1/p_i)	模幂简化
欧拉定理	a^φ(m) ≡ 1 (mod m), gcd(a,m)=1	模运算简化
费马小定理	a^(p-1) ≡ 1 (mod p), p为质数	模逆元快速计算
卢卡斯定理	C(n,k) ≡ ∏ C(n_i,k_i) (mod p)	大组合数模小质数

📊 素数筛法对比

筛法	时间复杂度	空间复杂度	特点
试除法	O(√n)每个数	O(1)	单个数判断
埃氏筛	O(n log log n)	O(n)	简单，易实现
欧拉筛	O(n)	O(n)	线性，每个数只筛一次

筛法	时间复杂度	空间复杂度	特点
分段筛	O((R-L) log log R)	O(√R + R-L)	大范围[L,R]筛选

📊 组合数计算方法

方法	时间复杂度	适用条件	特点
直接计算	O(k)	k较小	简单，无预处理
阶乘预处理	O(n)预处理，O(1)查询	$n \le 10^6$, p为质数	查询快
卢卡斯定理	O(p log_p n)	p较小，p为质数	处理n>p的情况
递推公式	$O(n^2)$	需要所有C(n,k)	杨辉三角

📊 数论分块技巧

求和类型	暴力复杂度	分块复杂度	优化倍数
∑ floor(n/i)	O(n)	O(√n)	√n倍
∑ floor(n/i)*i	O(n)	O(√n)	√n倍
∑ floor(n/i)*f(i)	O(n)	O(√n)（需前缀和）	√n倍

📊 关联题目索引

题目编号	题目名称	核心算法	难度
LS1024	计算乘方和	快速幂+等比求和	⭐⭐
LS1156	斐波那契数列	矩阵快速幂	⭐⭐
LS1161	高阶斐波那契	高阶矩阵快速幂	⭐⭐⭐
LS1220	扩展欧几里得	模逆元、同余方程	⭐⭐⭐
LS1233	质因数分解	试除法、最小质因子	⭐⭐

【核心算法详解】

1. 快速幂核心（`fast_pow`）

```
result = 1;
while (exp > 0) {
    if (exp & 1) {  // 当前位为1
        result = (result * base) % mod;
    }
    base = (base * base) % mod;  // 平方
    exp >>= 1;  // 移向下一位
}
```

2. 矩阵快速幂斐波那契（`fibonacci_matrix`）

```
// 转移矩阵
base = {{1, 1},
        {1, 0}};

// 计算矩阵幂
M = matrix_pow(base, n-1, mod);

// 结果在M[0][0]
return M[0][0] % mod;
```

3. 扩展欧几里得（`extended_gcd`）

```
if (b == 0) return {a, 1, 0};

auto [g, x1, y1] = extended_gcd(b, a % b);
i64 x = y1;
i64 y = x1 - (a / b) * y1;

return {g, x, y};
```

4. 组合数预处理（`Combinatorics`）

```
// 预处理阶乘
fac[0] = 1;
for (i = 1; i <= n; i++)
    fac[i] = fac[i-1] * i % mod;

// 预处理阶乘逆元
inv_fac[n] = fast_pow(fac[n], mod-2, mod);
for (i = n-1; i >= 0; i--)
    inv_fac[i] = inv_fac[i+1] * (i+1) % mod;

// 组合数计算
C(n,k) = fac[n] * inv_fac[k] % mod * inv_fac[n-k] % mod;
```

【测试用例设计】
代码包含多种测试用例：

1. **基础测试**：验证基本功能
2. **边界测试**：零、一、负数等边界
3. **性能测试**：大数、大数据量
4. **交互测试**：用户自定义输入

【扩展与优化】

1. 算法优化

- 快速幂：处理负指数、零底数等边界
- 矩阵乘法：使用Strassen算法进一步优化
- 素数筛：位压缩减少空间使用

2. 功能扩展

- 大数运算：支持超过64位的整数
- 多项式快速幂：处理多项式取模
- 离散对数：BSGS算法

3. 应用扩展

- 密码学：RSA加密解密
- 编码理论：纠错码、校验和
- 组合数学：生成函数、容斥原理

【学习建议】

1. **理解数学原理**：先理解背后的数学公式和定理
2. **掌握核心模板**：熟记快速幂、扩展欧几里得等模板
3. **练习变种问题**：从简单问题开始，逐步挑战复杂问题
4. **分析复杂度**：理解各种算法的时间空间复杂度

【常见错误】

1. **模运算错误**：忘记取模或取模位置错误
2. **整数溢出**：中间结果超出数据类型范围
3. **边界条件**：零、一、负数等特殊情况处理
4. **质数判断**：误判质数或合数
5. **逆元不存在**：在gcd(a,m)≠1时求逆元

【实用技巧】

1. **模运算性质**：(a+b)%m = ((a%m)+(b%m))%m
2. **费马小定理**：质数模数下快速求逆元
3. **欧拉定理**：非质数模数下简化幂运算
4. **卢卡斯定理**：大组合数模小质数
5. **数论分块**：高效计算∑ floor(n/i)类求和

5.4 贡献思维与单调栈（课程16）

📦 核心代码模板

```
// ==================== 贡献思维与单调栈 ====================

/*****************************************************************************
 * 1. 单调栈：下一个更大元素（每日温度问题）
 * 问题：对于每个位置，找到下一个更高温度的天数间隔
 * 解法：维护单调递减栈（栈底到栈顶递减），从右向左遍历
 * 时间复杂度：O(n)，空间复杂度：O(n)
 *****************************************************************************/
vector<i64> next_greater_element(vector<i64>& temperatures) {
    i64 n = temperatures.size();
    vector<i64> result(n, 0);
```

```
    stack<i64> stk;  // 存储下标，维护单调递减栈

    // 从右向左遍历，找右边第一个更大的元素
    for (i64 i = n - 1; i >= 0; i--) {
        // 维护单调递减栈：弹出所有小于等于当前温度的元素
        while (!stk.empty() && temperatures[stk.top()] <= temperatures[i]) {
            stk.pop();
        }
        // 栈顶元素就是右边第一个更大的温度
        result[i] = stk.empty() ? 0 : stk.top() - i;
        stk.push(i);
    }
    return result;
}


/**************************************************************************
 * 2. 柱状图中最大矩形面积（LS1199）
 * 问题：在柱状图中找到面积最大的矩形
 * 解法：单调栈求每个柱子左右第一个比它矮的柱子位置
 * 时间复杂度：O(n)，空间复杂度：O(n)
 **************************************************************************/
i64 largest_rectangle_area(vector<i64>& heights) {
    i64 n = heights.size();
    stack<i64> stk;
    vector<i64> left(n), right(n);

    // 求左边第一个比当前矮的柱子
    for (i64 i = 0; i < n; i++) {
        // 维护单调递增栈（栈底到栈顶递增）
        while (!stk.empty() && heights[stk.top()] >= heights[i]) {
            stk.pop();
        }
        left[i] = stk.empty() ? -1 : stk.top();  // -1表示左边没有更矮的
        stk.push(i);
    }

    // 清空栈，求右边第一个比当前矮的柱子
    while (!stk.empty()) stk.pop();
    for (i64 i = n - 1; i >= 0; i--) {
        while (!stk.empty() && heights[stk.top()] >= heights[i]) {
            stk.pop();
        }
        right[i] = stk.empty() ? n : stk.top();  // n表示右边没有更矮的
        stk.push(i);
    }

    // 计算最大面积：以每个柱子为高度的最大矩形
    i64 max_area = 0;
    for (i64 i = 0; i < n; i++) {
        i64 width = right[i] - left[i] - 1;  // 左右边界之间的宽度
        max_area = max(max_area, heights[i] * width);
    }
    return max_area;
}
```

```cpp
/**********************************************************************
 * 3. 接雨水（LS1247）
 * 问题：计算柱子间能接多少雨水
 * 解法1：单调栈，计算每个低洼处的储水量
 * 解法2：双指针，分别维护左右最大高度
 * 时间复杂度：O(n)，空间复杂度：O(n)或O(1)
 **********************************************************************/
i64 trap_rain_water(vector<i64>& height) {
    i64 n = height.size();
    if (n < 3) return 0;

    // 方法1：单调栈解法
    i64 water = 0;
    stack<i64> stk;  // 存储下标，维护单调递减栈

    for (i64 i = 0; i < n; i++) {
        // 当前柱子比栈顶高，形成低洼
        while (!stk.empty() && height[stk.top()] < height[i]) {
            i64 bottom = stk.top();  // 低洼底部
            stk.pop();

            if (stk.empty()) break;  // 左边没有柱子

            i64 left = stk.top();  // 左边柱子
            // 计算当前层能接的水量
            i64 curr_height = min(height[left], height[i]) - height[bottom];
            i64 curr_width = i - left - 1;
            water += curr_height * curr_width;
        }
        stk.push(i);
    }

    return water;
}

// 接雨水的双指针解法
i64 trap_rain_water_two_pointers(vector<i64>& height) {
    i64 n = height.size();
    if (n < 3) return 0;

    i64 left = 0, right = n - 1;
    i64 left_max = 0, right_max = 0;
    i64 water = 0;

    while (left < right) {
        // 哪边低处理哪边
        if (height[left] < height[right]) {
            if (height[left] >= left_max) {
                left_max = height[left];  // 更新左边最大值
            } else {
                water += left_max - height[left];  // 接水
            }
            left++;
        } else {
            if (height[right] >= right_max) {
```

```
                right_max = height[right];  // 更新右边最大值
            } else {
                water += right_max - height[right];  // 接水
            }
            right--;
        }
    }
    return water;
}


/**************************************************************************
 * 4. 子数组最小值之和（LS1244）- 贡献法
 * 问题：计算所有子数组最小值的和
 * 解法：单调栈求每个元素作为最小值的范围，计算贡献
 * 时间复杂度：O(n)，空间复杂度：O(n)
 **************************************************************************/
i64 sum_of_subarray_minimums(vector<i64>& arr) {
    i64 n = arr.size();
    const i64 MOD = 1e9 + 7;

    stack<i64> stk;
    vector<i64> left(n), right(n);

    // 求左边第一个比当前小的元素（严格小于）
    for (i64 i = 0; i < n; i++) {
        while (!stk.empty() && arr[stk.top()] > arr[i]) {
            stk.pop();
        }
        left[i] = stk.empty() ? -1 : stk.top();
        stk.push(i);
    }

    // 清空栈
    while (!stk.empty()) stk.pop();

    // 求右边第一个比当前小的元素（小于等于，避免重复计算）
    for (i64 i = n - 1; i >= 0; i--) {
        while (!stk.empty() && arr[stk.top()] >= arr[i]) {
            stk.pop();
        }
        right[i] = stk.empty() ? n : stk.top();
        stk.push(i);
    }

    // 贡献法：arr[i]作为最小值出现在多少个子数组中
    i64 ans = 0;
    for (i64 i = 0; i < n; i++) {
        i64 left_count = i - left[i];       // 左边可以延伸的长度
        i64 right_count = right[i] - i;     // 右边可以延伸的长度
        // 贡献 = arr[i] × 包含arr[i]作为最小值的子数组个数
        i64 contribution = arr[i] % MOD * left_count % MOD * right_count % MOD;
        ans = (ans + contribution) % MOD;
    }
    return ans;
}
```

```cpp
/**********************************************************************
 * 5. 子数组最大值之和 - 贡献法
 * 问题：计算所有子数组最大值的和
 * 解法：单调栈求每个元素作为最大值的范围，计算贡献
 * 时间复杂度：O(n)，空间复杂度：O(n)
 **********************************************************************/
i64 sum_of_subarray_maximums(vector<i64>& arr) {
    i64 n = arr.size();
    const i64 MOD = 1e9 + 7;

    stack<i64> stk;
    vector<i64> left(n), right(n);

    // 求左边第一个比当前大的元素（严格大于）
    for (i64 i = 0; i < n; i++) {
        while (!stk.empty() && arr[stk.top()] < arr[i]) {
            stk.pop();
        }
        left[i] = stk.empty() ? -1 : stk.top();
        stk.push(i);
    }

    while (!stk.empty()) stk.pop();

    // 求右边第一个比当前大的元素（小于等于，避免重复计算）
    for (i64 i = n - 1; i >= 0; i--) {
        while (!stk.empty() && arr[stk.top()] <= arr[i]) {
            stk.pop();
        }
        right[i] = stk.empty() ? n : stk.top();
        stk.push(i);
    }

    // 贡献法：arr[i]作为最大值出现在多少个子数组中
    i64 ans = 0;
    for (i64 i = 0; i < n; i++) {
        i64 left_count = i - left[i];       // 左边可以延伸的长度
        i64 right_count = right[i] - i;     // 右边可以延伸的长度
        i64 contribution = arr[i] % MOD * left_count % MOD * right_count % MOD;
        ans = (ans + contribution) % MOD;
    }
    return ans;
}


/**********************************************************************
 * 6. 子数组最值之差（LS1245）
 * 问题：计算所有子数组最大值与最小值之差的和
 * 解法：最大值之和 - 最小值之和
 * 时间复杂度：O(n)，空间复杂度：O(n)
 **********************************************************************/
i64 sum_of_subarray_range(vector<i64>& arr) {
    i64 max_sum = sum_of_subarray_maximums(arr);
    i64 min_sum = sum_of_subarray_minimums(arr);
```

```
    return (max_sum - min_sum + MOD) % MOD;  // 防止负数
}


/**********************************************************************
 * 7. 滑动窗口最大值（P1886）- 单调队列
 * 问题：在每个大小为k的滑动窗口中找最大值
 * 解法：单调递减队列，队头为当前窗口最大值
 * 时间复杂度：O(n)，空间复杂度：O(k)
 **********************************************************************/
vector<i64> sliding_window_max(vector<i64>& nums, i64 k) {
    i64 n = nums.size();
    deque<i64> dq;  // 存储下标，维护单调递减队列
    vector<i64> result;

    for (i64 i = 0; i < n; i++) {
        // 移除超出窗口的元素（队头）
        if (!dq.empty() && dq.front() <= i - k) {
            dq.pop_front();
        }

        // 维护单调递减队列：队头最大，队尾最小
        // 弹出所有小于等于当前元素的队尾元素
        while (!dq.empty() && nums[dq.back()] <= nums[i]) {
            dq.pop_back();
        }

        dq.push_back(i);

        // 窗口形成后记录最大值
        if (i >= k - 1) {
            result.push_back(nums[dq.front()]);
        }
    }
    return result;
}


/**********************************************************************
 * 8. 滑动窗口最小值 - 单调队列
 * 问题：在每个大小为k的滑动窗口中找最小值
 * 解法：单调递增队列，队头为当前窗口最小值
 * 时间复杂度：O(n)，空间复杂度：O(k)
 **********************************************************************/
vector<i64> sliding_window_min(vector<i64>& nums, i64 k) {
    i64 n = nums.size();
    deque<i64> dq;  // 存储下标，维护单调递增队列
    vector<i64> result;

    for (i64 i = 0; i < n; i++) {
        // 移除超出窗口的元素
        if (!dq.empty() && dq.front() <= i - k) {
            dq.pop_front();
        }

        // 维护单调递增队列：队头最小，队尾最大
```

```
        // 弹出所有大于等于当前元素的队尾元素
        while (!dq.empty() && nums[dq.back()] >= nums[i]) {
            dq.pop_back();
        }

        dq.push_back(i);

        // 窗口形成后记录最小值
        if (i >= k - 1) {
            result.push_back(nums[dq.front()]);
        }
    }
    return result;
}


/**********************************************************************
 * 9. 下一个更大元素（循环数组）
 * 问题：循环数组中每个元素的下一个更大元素
 * 解法：扩展数组或循环遍历
 * 时间复杂度：O(n)，空间复杂度：O(n)
 **********************************************************************/
vector<i64> next_greater_element_circular(vector<i64>& nums) {
    i64 n = nums.size();
    vector<i64> result(n, -1);
    stack<i64> stk;

    // 遍历两遍模拟循环数组
    for (i64 i = 0; i < 2 * n; i++) {
        i64 idx = i % n;
        // 维护单调递减栈
        while (!stk.empty() && nums[stk.top()] < nums[idx]) {
            i64 top_idx = stk.top();
            stk.pop();
            if (result[top_idx] == -1) {
                result[top_idx] = nums[idx];
            }
        }
        // 只压入第一遍遍历的元素
        if (i < n) {
            stk.push(idx);
        }
    }

    return result;
}


/**********************************************************************
 * 10. 最大矩形（LS1200）- 逐行+单调栈
 * 问题：在01矩阵中找面积最大的全1矩形
 * 解法：逐行构建柱状图，用单调栈求最大矩形
 * 时间复杂度：O(m×n)，空间复杂度：O(n)
 **********************************************************************/
i64 maximal_rectangle(vector<vector<char>>& matrix) {
    if (matrix.empty() || matrix[0].empty()) return 0;
```

```
    i64 m = matrix.size(), n = matrix[0].size();
    vector<i64> heights(n, 0);
    i64 max_area = 0;

    // 逐行处理
    for (i64 i = 0; i < m; i++) {
        // 更新当前行的柱状图高度
        for (i64 j = 0; j < n; j++) {
            if (matrix[i][j] == '1') {
                heights[j] += 1;
            } else {
                heights[j] = 0;
            }
        }
        // 对当前柱状图求最大矩形面积
        max_area = max(max_area, largest_rectangle_area(heights));
    }

    return max_area;
}


/**************************************************************************
 * 11. 贡献法通用模板
 * 计算每个元素对答案的贡献
 **************************************************************************/
template<typename T>
i64 contribution_method(vector<T>& arr, function<bool(T, T)> compare) {
    i64 n = arr.size();
    stack<i64> stk;
    vector<i64> left(n), right(n);

    // 求左边界
    for (i64 i = 0; i < n; i++) {
        while (!stk.empty() && compare(arr[stk.top()], arr[i])) {
            stk.pop();
        }
        left[i] = stk.empty() ? -1 : stk.top();
        stk.push(i);
    }

    while (!stk.empty()) stk.pop();

    // 求右边界
    for (i64 i = n - 1; i >= 0; i--) {
        while (!stk.empty() && compare(arr[stk.top()], arr[i])) {
            stk.pop();
        }
        right[i] = stk.empty() ? n : stk.top();
        stk.push(i);
    }

    // 计算总贡献
    i64 total = 0;
    for (i64 i = 0; i < n; i++) {
```

```
        i64 left_count = i - left[i];
        i64 right_count = right[i] - i;
        // 根据具体问题计算贡献
        total += left_count * right_count;
    }

    return total;
}
```

📊 单调栈类型总结

类型	单调性	解决的问题	示例
递增栈	栈底到栈顶递增	第一个更小元素、最小值贡献	柱状图最大矩形
递减栈	栈底到栈顶递减	第一个更大元素、最大值贡献	每日温度问题
严格递增	严格递增	避免重复计算最小值	子数组最小值之和
非严格递增	非严格递增	包含相等元素	特定问题边界处理

📊 贡献法公式总结

问题	贡献公式	说明
最小值之和	Σ arr[i] × (i-left) × (right-i)	left: 左边第一个更小，right: 右边第一个更小
最大值之和	Σ arr[i] × (i-left) × (right-i)	left: 左边第一个更大，right: 右边第一个更大
边界处理	left=-1，right=n	表示没有更小/更大的元素
相等元素	一边严格，一边非严格	避免重复计算或漏算

📊 单调队列维护规则

队列类型	维护规则	队头元素	解决的问题
递减队列	新元素比队尾大时弹出队尾	当前最大值	滑动窗口最大值
递增队列	新元素比队尾小时弹出队尾	当前最小值	滑动窗口最小值
过期检查	队头下标 ≤ i-k 时弹出	-	保持窗口大小

📊 常见问题模式与解法

问题模式	关键点	算法	时间复杂度
下一个更大/更小	单调栈+遍历方向	单调栈	O(n)
左右边界确定	两次单调栈遍历	单调栈	O(n)
贡献法计算	确定范围+乘法原理	贡献法	O(n)
滑动窗口最值	单调队列+过期处理	单调队列	O(n)
二维问题转化	逐行降维+单调栈	降维+单调栈	O(m×n)

问题模式	关键点	算法	时间复杂度
循环数组处理	扩展数组或遍历两遍	循环处理	O(n)

📊 关联题目索引

题目编号	题目名称	核心算法	难度
LS1199	柱状图最大矩形	单调栈求边界	⭐⭐
LS1247	接雨水	单调栈或双指针	⭐⭐
LS1244	子数组最小值之和	贡献法	⭐⭐⭐
LS1245	子数组最值之差	贡献法相减	⭐⭐⭐
P1886	滑动窗口最大值	单调队列	⭐⭐
P1725	琪露诺的算术教室	单调队列优化DP	⭐⭐⭐
LS1200	最大矩形	逐行+单调栈	⭐⭐⭐

【各算法时间复杂度对比】

算法	时间复杂度	空间复杂度	适用场景
下一个更大元素	O(n)	O(n)	温度、股价等序列
柱状图最大矩形	O(n)	O(n)	直方图、矩阵问题
接雨水（单调栈）	O(n)	O(n)	地形储水问题
接雨水（双指针）	O(n)	O(1)	空间优化版
子数组最小值之和	O(n)	O(n)	贡献法统计问题
滑动窗口最值	O(n)	O(k)	固定窗口最值查询
最大矩形	O(m×n)	O(n)	二维01矩阵

【核心算法详解】

1. 单调栈框架（`next_greater_element`）

```
stack<i64> stk;
for (i64 i = n-1; i >= 0; i--) {
    // 维护单调性：弹出不符合条件的元素
    while (!stk.empty() && arr[stk.top()] <= arr[i]) {
        stk.pop();
    }
    // 计算结果
    result[i] = stk.empty() ? 0 : stk.top() - i;
    // 压入当前元素
    stk.push(i);
}
```

2. 贡献法核心（`sum_of_subarray_minimums`）

```
// 求左右边界
left[i] = 左边第一个比arr[i]小的位置
right[i] = 右边第一个比arr[i]小的位置

// 计算贡献
贡献 = arr[i] × (i - left[i]) × (right[i] - i)
总和 = Σ 贡献
```

3. 单调队列维护（`sliding_window_max`）

```
deque<i64> dq;  // 存储下标
for (i64 i = 0; i < n; i++) {
    // 1. 移除过期元素
    if (!dq.empty() && dq.front() <= i - k) dq.pop_front();

    // 2. 维护单调性
    while (!dq.empty() && nums[dq.back()] <= nums[i]) dq.pop_back();

    // 3. 压入当前元素
    dq.push_back(i);

    // 4. 记录结果
    if (i >= k - 1) result.push_back(nums[dq.front()]);
}
```

【测试用例设计】
代码包含多种测试用例：

1. **基础测试**：验证基本功能
2. **边界测试**：空数组、单个元素、重复元素
3. **性能测试**：10万数据量，1000×1000矩阵
4. **交互测试**：用户自定义输入

【扩展与优化】

1. 空间优化

- 部分问题可原地修改数组存储结果
- 滑动窗口使用deque而非vector
- 贡献法可合并遍历减少空间

2. 时间优化

- 单调栈/队列均摊O(n)
- 贡献法两次遍历完成
- 二维问题逐行处理避免重复计算

3. 边界处理技巧

- 哨兵元素：在数组两端添加极值简化判断
- 循环数组：遍历两遍或拼接数组

- 相等元素：严格与非严格比较的选择

【学习建议】

1. **理解单调性**：画图理解栈内元素单调性的维护
2. **掌握模板**：记住单调栈、单调队列的通用模板
3. **贡献法思维**：从枚举子数组转变为计算每个元素的贡献
4. **二维降一维**：掌握将二维问题转化为一维问题的技巧

【常见错误】

1. **索引混淆**：0-indexed与1-indexed混用
2. **边界条件**：left/right的初始值设置错误
3. **相等处理**：严格与非严格比较选择不当
4. **模运算**：忘记取模或模运算错误
5. **过期检查**：单调队列忘记检查队头是否过期

【实用技巧】

1. **可视化调试**：打印栈/队列内容观察变化
2. **小数据验证**：手工计算小数据验证算法
3. **对拍测试**：与暴力解法对比结果
4. **复杂度证明**：理解为什么每个元素只入栈出栈一次
5. **模版化代码**：封装通用函数，便于复用和调试

5.5 前缀和与差分（课程20）

📦 核心代码模板

```
// ==================== 前缀和与差分 ====================

/**********************************************************************
 * 1. 和为K的子数组个数（哈希表优化）
 * 问题：统计数组中连续子数组和为k的个数
 * 解法：前缀和+哈希表，寻找prefix[j] - prefix[i] = k
 * 时间复杂度：O(n)，空间复杂度：O(n)
 **********************************************************************/
i64 subarray_sum_equals_k(vector<i64>& nums, i64 k) {
    unordered_map<i64, i64> prefix_count;
    prefix_count[0] = 1;  // 重要：前缀和为0出现1次（空前缀）

    i64 prefix_sum = 0, count = 0;
    for (i64 num : nums) {
        prefix_sum += num;

        // 需要的前缀和 = 当前前缀和 - k
        // 即：prefix_sum - need = k
        i64 need = prefix_sum - k;
        if (prefix_count.count(need)) {
            count += prefix_count[need];
```

```
        }

        prefix_count[prefix_sum]++;
    }
    return count;
}


/**************************************************************************
 * 2. 最长和为0的子数组（0视为-1，LS1265）
 * 问题：将0视为-1，1视为1，求最长的和为0的子数组
 * 解法：前缀和+哈希表记录第一次出现位置
 * 时间复杂度：O(n)，空间复杂度：O(n)
 **************************************************************************/
i64 longest_subarray_with_equal_01(vector<i64>& nums) {
    // 将0视为-1，问题转化为求和为0的最长子数组
    unordered_map<i64, i64> first_occurrence;
    first_occurrence[0] = -1;  // 前缀和为0在索引-1处出现（空数组）

    i64 prefix_sum = 0, max_len = 0;
    for (i64 i = 0; i < nums.size(); i++) {
        prefix_sum += (nums[i] == 0 ? -1 : 1);

        if (first_occurrence.count(prefix_sum)) {
            // 当前前缀和之前出现过，说明中间部分和为0
            max_len = max(max_len, i - first_occurrence[prefix_sum]);
        } else {
            // 记录该前缀和第一次出现的位置
            first_occurrence[prefix_sum] = i;
        }
    }
    return max_len;
}


/**************************************************************************
 * 3. 异或子数组个数（LS1264）
 * 问题：统计异或值为target的子数组个数
 * 解法：利用异或前缀性质，prefix[r]^prefix[l-1]=区间异或
 * 时间复杂度：O(n)，空间复杂度：O(n)
 **************************************************************************/
i64 count_xor_subarrays(vector<i64>& nums, i64 target) {
    unordered_map<i64, i64> xor_count;
    xor_count[0] = 1;  // 空前缀的异或值为0

    i64 prefix_xor = 0, count = 0;
    for (i64 num : nums) {
        prefix_xor ^= num;

        // 需要的前缀异或值：prefix_xor ^ need = target
        // 所以 need = prefix_xor ^ target
        i64 need = prefix_xor ^ target;
        if (xor_count.count(need)) {
            count += xor_count[need];
        }
```

```
            xor_count[prefix_xor]++;
        }
        return count;
    }


    /**************************************************************************
     * 4. 最长平衡子串（LS1267, LS1266）
     * 问题：在01字符串中找到最长的子串，其中0和1的数量相等
     * 解法：将0视为-1，1视为1，转化为求和为0的最长子数组
     * 时间复杂度：O(n)，空间复杂度：O(n)
     **************************************************************************/
    i64 longest_balanced_substring(string s) {
        // 平衡：0和1的数量相等
        unordered_map<i64, i64> first_occurrence;
        first_occurrence[0] = -1;  // 平衡状态在索引-1处出现

        i64 balance = 0, max_len = 0;
        for (i64 i = 0; i < s.size(); i++) {
            balance += (s[i] == '0' ? -1 : 1);

            if (first_occurrence.count(balance)) {
                // 相同的balance值再次出现，说明中间部分平衡
                max_len = max(max_len, i - first_occurrence[balance]);
            } else {
                first_occurrence[balance] = i;
            }
        }
        return max_len;
    }


    /**************************************************************************
     * 5. 二维前缀和（LS1270）
     * 问题：快速查询子矩阵和
     * 解法：pre[i][j] = 左上角(0,0)到(i-1,j-1)的和
     * 时间复杂度：构造O(mn)，查询O(1)，空间复杂度：O(mn)
     **************************************************************************/
    class PrefixSum2D {
    private:
        vector<vector<i64>> prefix;
        i64 m, n;

    public:
        // 构造函数，从matrix构建前缀和
        PrefixSum2D(vector<vector<i64>>& matrix) {
            m = matrix.size();
            n = matrix[0].size();
            prefix.assign(m + 1, vector<i64>(n + 1, 0));

            // 构建前缀和数组
            for (i64 i = 1; i <= m; i++) {
                for (i64 j = 1; j <= n; j++) {
                    prefix[i][j] = matrix[i-1][j-1] + prefix[i-1][j]
                                 + prefix[i][j-1] - prefix[i-1][j-1];
                }
```

```
        }
    }

    // 查询子矩阵和：(x1,y1)到(x2,y2)，包含边界
    i64 query(i64 x1, i64 y1, i64 x2, i64 y2) {
        // 边界检查
        if (x1 < 0 || y1 < 0 || x2 >= m || y2 >= n || x1 > x2 || y1 > y2) {
            return 0;
        }
        return prefix[x2+1][y2+1] - prefix[x1][y2+1]
             - prefix[x2+1][y1] + prefix[x1][y1];
    }

    // 快速求任意子矩阵和
    i64 sum_region(i64 row1, i64 col1, i64 row2, i64 col2) {
        return query(row1, col1, row2, col2);
    }

    // 获取整个矩阵的和
    i64 total_sum() {
        return prefix[m][n];
    }

    // 打印前缀和数组（调试用）
    void print_prefix() {
        for (i64 i = 0; i <= m; i++) {
            for (i64 j = 0; j <= n; j++) {
                cout << prefix[i][j] << " ";
            }
            cout << "\n";
        }
    }
};


/**************************************************************************
 * 6. 差分数组（区间更新+单点查询）
 * 问题：支持区间加操作，快速查询单点值
 * 解法：差分数组，区间[l,r]加val：diff[l]+=val, diff[r+1]-=val
 * 时间复杂度：更新O(1)，查询O(n)，空间复杂度：O(n)
 **************************************************************************/
class DifferenceArray {
private:
    vector<i64> diff;
    i64 n;

public:
    // 构造函数，n为数组长度
    DifferenceArray(i64 n) : n(n), diff(n + 2, 0) {}

    // 区间[l,r]增加val（1-indexed）
    void range_add(i64 l, i64 r, i64 val) {
        if (l < 1 || r > n || l > r) return;
        diff[l] += val;
        diff[r + 1] -= val;
    }
```

```
    // 获取最终数组
    vector<i64> get_array() {
        vector<i64> arr(n + 1, 0);
        i64 curr = 0;
        for (i64 i = 1; i <= n; i++) {
            curr += diff[i];
            arr[i] = curr;
        }
        return vector<i64>(arr.begin() + 1, arr.end());
    }

    // 单点查询（前缀和累加）
    i64 point_query(i64 idx) {
        if (idx < 1 || idx > n) return 0;
        i64 sum = 0;
        for (i64 i = 1; i <= idx; i++) {
            sum += diff[i];
        }
        return sum;
    }

    // 批量区间更新
    void batch_update(const vector<tuple<i64, i64, i64>>& updates) {
        for (const auto& [l, r, val] : updates) {
            range_add(l, r, val);
        }
    }

    // 重置差分数组
    void reset() {
        fill(diff.begin(), diff.end(), 0);
    }
};


/**************************************************************************
 * 7. 二维差分
 * 问题：支持子矩阵加操作，快速查询整个矩阵
 * 解法：二维差分，利用容斥原理
 * 时间复杂度：更新O(1)，重建O(mn)，空间复杂度：O(mn)
 **************************************************************************/
class DifferenceArray2D {
private:
    vector<vector<i64>> diff;
    i64 m, n;

public:
    // 构造函数，m行n列
    DifferenceArray2D(i64 m, i64 n) : m(m), n(n),
                                      diff(m + 2, vector<i64>(n + 2, 0)) {}

    // 子矩阵(x1,y1)到(x2,y2)增加val（0-indexed）
    void range_add(i64 x1, i64 y1, i64 x2, i64 y2, i64 val) {
        if (x1 < 0 || y1 < 0 || x2 >= m || y2 >= n || x1 > x2 || y1 > y2) {
            return;
```

```
        }
        diff[x1][y1] += val;
        diff[x1][y2 + 1] -= val;
        diff[x2 + 1][y1] -= val;
        diff[x2 + 1][y2 + 1] += val;
    }

    // 获取最终矩阵
    vector<vector<i64>> get_matrix() {
        vector<vector<i64>> mat(m, vector<i64>(n, 0));
        vector<vector<i64>> prefix(m + 1, vector<i64>(n + 1, 0));

        // 二维前缀和恢复
        for (i64 i = 0; i < m; i++) {
            for (i64 j = 0; j < n; j++) {
                prefix[i+1][j+1] = prefix[i][j+1] + prefix[i+1][j]
                                 - prefix[i][j] + diff[i][j];
                mat[i][j] = prefix[i+1][j+1];
            }
        }
        return mat;
    }

    // 单点查询
    i64 point_query(i64 x, i64 y) {
        if (x < 0 || x >= m || y < 0 || y >= n) return 0;
        i64 sum = 0;
        for (i64 i = 0; i <= x; i++) {
            for (i64 j = 0; j <= y; j++) {
                sum += diff[i][j];
            }
        }
        return sum;
    }
};


/**************************************************************************
 * 8. 维护数组（LS1269）
 * 问题：支持区间加操作和区间和查询
 * 解法：差分数组+前缀和优化
 * 时间复杂度：更新O(1)，查询O(1)（重建后），空间复杂度：O(n)
 **************************************************************************/
class ArrayMaintainer {
private:
    vector<i64> arr;
    DifferenceArray diff;
    vector<i64> prefix;  // 前缀和数组
    bool need_rebuild;

public:
    // 构造函数，n为数组长度，初始值为0
    ArrayMaintainer(i64 n) : diff(n), arr(n + 1, 0),
                             prefix(n + 1, 0), need_rebuild(false) {}

    // 构造函数，带初始数组
```

```
    ArrayMaintainer(const vector<i64>& init_arr) :
        diff(init_arr.size()), arr(init_arr.size() + 1, 0),
        prefix(init_arr.size() + 1, 0), need_rebuild(false) {
        // 构建初始前缀和
        for (i64 i = 0; i < init_arr.size(); i++) {
            arr[i+1] = init_arr[i];
            prefix[i+1] = prefix[i] + arr[i+1];
        }
    }

    // 区间加操作
    void add_range(i64 l, i64 r, i64 val) {
        diff.range_add(l, r, val);
        need_rebuild = true;
    }

    // 查询区间和（l,r为1-indexed）
    i64 query_sum(i64 l, i64 r) {
        if (need_rebuild) {
            rebuild();
            need_rebuild = false;
        }
        return prefix[r] - prefix[l-1];
    }

    // 单点查询
    i64 query_point(i64 idx) {
        if (need_rebuild) {
            rebuild();
            need_rebuild = false;
        }
        return arr[idx];
    }

private:
    // 重建数组和前缀和
    void rebuild() {
        vector<i64> changes = diff.get_array();
        for (i64 i = 1; i < arr.size(); i++) {
            arr[i] += changes[i-1];
            prefix[i] = prefix[i-1] + arr[i];
        }
        diff.reset();  // 重置差分数组
    }
};


/**************************************************************************
 * 9. 最大子数组和（前缀和优化）
 * 问题：找到和最大的连续子数组
 * 解法：遍历时维护最小前缀和
 * 时间复杂度：O(n)，空间复杂度：O(1)
 *************************************************************************/
i64 max_subarray_sum(vector<i64>& nums) {
    i64 n = nums.size();
    if (n == 0) return 0;
```

```cpp
    i64 prefix_sum = 0;
    i64 min_prefix = 0;  // 最小前缀和（空前缀）
    i64 max_sum = LLONG_MIN;

    for (i64 i = 0; i < n; i++) {
        prefix_sum += nums[i];
        max_sum = max(max_sum, prefix_sum - min_prefix);
        min_prefix = min(min_prefix, prefix_sum);
    }

    return max_sum;
}


/*******************************************************************************
 * 10. 前缀和通用工具类
 * 提供一维、二维前缀和的通用操作
 ******************************************************************************/
class PrefixSumUtils {
public:
    // 一维前缀和
    static vector<i64> build_prefix(const vector<i64>& nums) {
        i64 n = nums.size();
        vector<i64> prefix(n + 1, 0);
        for (i64 i = 0; i < n; i++) {
            prefix[i+1] = prefix[i] + nums[i];
        }
        return prefix;
    }

    // 一维区间和查询
    static i64 range_sum(const vector<i64>& prefix, i64 l, i64 r) {
        // l,r为0-indexed，包含两端
        return prefix[r+1] - prefix[l];
    }

    // 二维前缀和
    static vector<vector<i64>> build_prefix_2d(const vector<vector<i64>>& matrix)
{
        i64 m = matrix.size(), n = matrix[0].size();
        vector<vector<i64>> prefix(m + 1, vector<i64>(n + 1, 0));

        for (i64 i = 0; i < m; i++) {
            for (i64 j = 0; j < n; j++) {
                prefix[i+1][j+1] = matrix[i][j] + prefix[i][j+1]
                                 + prefix[i+1][j] - prefix[i][j];
            }
        }
        return prefix;
    }

    // 二维区间和查询
    static i64 range_sum_2d(const vector<vector<i64>>& prefix,
                            i64 x1, i64 y1, i64 x2, i64 y2) {
        return prefix[x2+1][y2+1] - prefix[x1][y2+1]
```

```
            - prefix[x2+1][y1] + prefix[x1][y1];
    }

    // 判断是否存在和为k的子数组
    static bool has_subarray_sum(const vector<i64>& nums, i64 k) {
        unordered_set<i64> prefix_sums;
        prefix_sums.insert(0);

        i64 prefix_sum = 0;
        for (i64 num : nums) {
            prefix_sum += num;
            if (prefix_sums.count(prefix_sum - k)) {
                return true;
            }
            prefix_sums.insert(prefix_sum);
        }
        return false;
    }
};
```

📊 前缀和与差分总结

类型	核心思想	时间复杂度	空间复杂度	适用场景
一维前缀和	pre[i]=Σnums[0..i-1]	构造O(n)，查询O(1)	O(n)	区间和查询
哈希优化前缀和	记录前缀和出现情况	O(n)	O(n)	特定和子数组统计
异或前缀和	利用异或性质	O(n)	O(n)	异或值子数组统计
二维前缀和	二维累加和	构造O(mn)，查询O(1)	O(mn)	子矩阵和查询
一维差分	端点操作代替区间更新	更新O(1)，重建O(n)	O(n)	区间加操作
二维差分	容斥原理处理矩形	更新O(1)，重建O(mn)	O(mn)	子矩阵加操作

📊 异或运算性质

性质	公式	说明
自反性	$a \oplus a = 0$	相同数异或为0
零元性	$a \oplus 0 = a$	与0异或不变
交换律	$a \oplus b = b \oplus a$	可交换顺序
结合律	$(a \oplus b) \oplus c = a \oplus (b \oplus c)$	可任意结合

性质	公式	说明
区间异或	xor[l..r] = pre[r+1] ⊕ pre[l]	前缀异或差值

📊 常见问题模式

模式	关键点	算法	时间复杂度
和为K的子数组	寻找pre[j]-pre[i]=k	前缀和+哈希表	O(n)
最长和为0子数组	0→-1，1→1	前缀和+首次出现记录	O(n)
异或子数组	利用异或性质	异或前缀+哈希表	O(n)
平衡子串	统计0/1数量相等	前缀和+哈希表	O(n)
子矩阵查询	二维累加和	二维前缀和	O(1)查询
区间更新查询	端点操作优化	差分数组	O(1)更新

📊 关联题目索引

题目编号	题目名称	核心算法	难度
LS1265	连续数组（0视为-1）	前缀和+哈希表	⭐⭐
LS1264	异或子数组	异或前缀+哈希表	⭐⭐
LS1267	最长平衡子串	前缀和+哈希表	⭐⭐
LS1266	平衡子串变体	前缀和+哈希表	⭐⭐
LS1268	异或序列	异或前缀+查询	⭐⭐⭐
LS1269	维护数组	差分+前缀和	⭐⭐⭐
LS1270	负载平衡	二维前缀和分割	⭐⭐⭐⭐
P14253	旅行问题	前缀和优化DP	⭐⭐⭐⭐

【核心算法详解】

1. 和为K的子数组（`subarray_sum_equals_k`）

```
unordered_map<i64, i64> prefix_count;
prefix_count[0] = 1;  // 关键初始化

for (num : nums) {
    prefix_sum += num;
    i64 need = prefix_sum - k;  // 需要的前缀和
    if (prefix_count.count(need)) {
        count += prefix_count[need];  // 累加组合数
    }
    prefix_count[prefix_sum]++;  // 记录当前前缀和
}
```

2. 二维前缀和查询（`PrefixSum2D::query`）

```
// 构造公式
prefix[i][j] = matrix[i-1][j-1] + prefix[i-1][j]
             + prefix[i][j-1] - prefix[i-1][j-1];

// 查询公式（容斥原理）
sum = prefix[x2+1][y2+1] - prefix[x1][y2+1]
    - prefix[x2+1][y1] + prefix[x1][y1];
```

3. 差分数组更新（`DifferenceArray::range_add`）

```
// 区间[l,r]加val
diff[l] += val;
diff[r+1] -= val;

// 恢复原数组（前缀和）
curr = 0;
for (i = 1; i <= n; i++) {
    curr += diff[i];
    arr[i] = curr;
}
```

4. 异或前缀性质应用

```
// 关键性质：区间异或 = 前缀异或的异或
// xor[l..r] = pre_xor[r+1] ⊕ pre_xor[l]

// 统计异或值为target的子数组
i64 need = prefix_xor ^ target;
if (xor_count.count(need)) {
    count += xor_count[need];
}
```

【测试用例设计】
代码包含多种测试用例：

1. **基础测试**：验证基本功能
2. **边界测试**：空数组、单个元素、负数值
3. **性能测试**：10万数据量，1000×1000矩阵

4. **交互测试**：用户自定义输入

【扩展与优化】

1. 空间优化

- 前缀和数组：必须O(n)或O(mn)存储
- 哈希表优化：最坏O(n)，平均O(1)访问
- 滚动数组：处理滑动窗口类问题

2. 时间优化

- 预处理：一次性构造前缀和，多次查询
- 批量操作：差分数组批量更新
- 并行计算：二维前缀和可并行构造

3. 数值范围处理

- 负数处理：哈希表支持负键值
- 大数处理：使用long long防止溢出
- 浮点数：可用但注意精度

【学习建议】

1. **理解核心公式**：掌握前缀和、差分的核心公式
2. **画图辅助**：绘制前缀和、差分的变化过程
3. **分类练习**：按问题类型分组练习
4. **模版化思维**：记住通用框架，根据具体问题调整

【常见错误】

1. **索引错误**：0-indexed与1-indexed混淆
2. **初始化错误**：忘记初始化prefix[0]=0
3. **边界处理**：查询越界未检查
4. **整数溢出**：大数累加未使用long long
5. **哈希表使用**：错误使用count和find

【实用技巧】

1. **调试输出**：打印前缀和数组验证计算
2. **小数据验证**：手工计算小数据验证算法
3. **对拍测试**：与暴力解法对比结果
4. **复杂度分析**：确保算法在数据范围内可行
5. **模块化设计**：将前缀和、差分封装为类，便于复用

5.6 尺取法与双指针（课程18）

📦 核心代码模板

```
// ==================== 尺取法与双指针 ====================
```

```cpp
/*****************************************************************************
 * 1. 滑动窗口模板：最短子数组和≥target
 * 问题：找到和至少为target的最短连续子数组
 * 解法：滑动窗口，扩展右边界，收缩左边界直到不满足条件
 * 时间复杂度：O(n)，空间复杂度：O(1)
 *****************************************************************************/
i64 min_subarray_len(vector<i64>& nums, i64 target) {
    i64 n = nums.size();
    i64 left = 0, sum = 0;
    i64 min_len = LLONG_MAX;

    for (i64 right = 0; right < n; right++) {
        sum += nums[right];  // 扩展右边界

        // 收缩左边界直到不满足条件
        while (sum >= target) {
            min_len = min(min_len, right - left + 1);
            sum -= nums[left];
            left++;
        }
    }

    return min_len == LLONG_MAX ? 0 : min_len;
}


/*****************************************************************************
 * 2. 无重复字符的最长子串
 * 问题：找到不包含重复字符的最长子串
 * 解法：滑动窗口+字符位置记录，遇到重复时移动左边界
 * 时间复杂度：O(n)，空间复杂度：O(字符集大小)
 *****************************************************************************/
i64 longest_substring_without_repeats(string s) {
    vector<i64> last_index(256, -1);  // 记录字符最后出现的位置
    i64 max_len = 0;
    i64 left = 0;

    for (i64 right = 0; right < s.size(); right++) {
        // 如果当前字符之前出现过，且在窗口内
        if (last_index[s[right]] >= left) {
            left = last_index[s[right]] + 1;  // 移动左边界
        }

        last_index[s[right]] = right;  // 更新字符位置
        max_len = max(max_len, right - left + 1);
    }

    return max_len;
}


/*****************************************************************************
 * 3. 包含所有字符的最短子串 (最小覆盖子串，P1638)
 * 问题：在字符串s中找到包含字符串t所有字符的最短子串
 * 解法：滑动窗口+哈希表统计，valid计数器判断是否满足条件
 * 时间复杂度：O(n)，空间复杂度：O(字符集大小)
```

```
 ******************************************************************************/
string min_window_containing_all(string s, string t) {
    unordered_map<char, i64> need, window;
    for (char c : t) need[c]++;

    i64 left = 0, right = 0;
    i64 valid = 0;  // 满足条件的字符数
    i64 start = 0, min_len = LLONG_MAX;

    while (right < s.size()) {
        char c = s[right];
        right++;

        if (need.count(c)) {
            window[c]++;
            if (window[c] == need[c]) {
                valid++;
            }
        }

        // 当窗口包含所有需要的字符时，尝试收缩
        while (valid == need.size()) {
            // 更新最小窗口
            if (right - left < min_len) {
                start = left;
                min_len = right - left;
            }

            char d = s[left];
            left++;

            if (need.count(d)) {
                if (window[d] == need[d]) {
                    valid--;
                }
                window[d]--;
            }
        }
    }

    return min_len == LLONG_MAX ? "" : s.substr(start, min_len);
}


/******************************************************************************
 * 4. 相向双指针：两数之和（已排序数组，LS1256）
 * 问题：在已排序数组中找两个数，使它们的和等于target
 * 解法：从两端向中间移动，根据和与target的比较调整指针
 * 时间复杂度：O(n)，空间复杂度：O(1)
 ******************************************************************************/
pair<i64, i64> two_sum_sorted(vector<i64>& nums, i64 target) {
    i64 left = 0, right = nums.size() - 1;

    while (left < right) {
        i64 sum = nums[left] + nums[right];
        if (sum == target) {
```

```
            return {left, right};
        } else if (sum < target) {
            left++;  // 需要更大的数
        } else {
            right--;  // 需要更小的数
        }
    }
    return {-1, -1};
}


/*****************************************************************************
 * 5. 两数之差（已排序数组，LS1257）
 * 问题：在已排序数组中找两个数，使它们的差等于target
 * 解法：同向双指针，固定一个指针，移动另一个指针
 * 时间复杂度：O(n)，空间复杂度：O(1)
 *****************************************************************************/
pair<i64, i64> two_diff_sorted(vector<i64>& nums, i64 target) {
    i64 n = nums.size();
    for (i64 i = 0, j = 1; j < n; ) {
        i64 diff = nums[j] - nums[i];
        if (diff == target) {
            return {i, j};
        } else if (diff < target) {
            j++;  // 需要更大的差值
        } else {
            i++;  // 需要更小的差值
        }
        if (i == j) j++;
    }
    return {-1, -1};
}


/*****************************************************************************
 * 6. k个最接近的数（已排序数组，LS1255）
 * 问题：在已排序数组中找到k个最接近x的数
 * 解法：二分查找+双指针确定起始位置
 * 时间复杂度：O(logn + k)，空间复杂度：O(1)
 *****************************************************************************/
vector<i64> find_closest_elements(vector<i64>& nums, i64 k, i64 x) {
    // 找到最接近x的起始位置
    i64 left = 0, right = nums.size() - k;

    while (left < right) {
        i64 mid = (left + right) / 2;
        // 比较mid和mid+k哪个更接近x
        if (x - nums[mid] > nums[mid + k] - x) {
            left = mid + 1;  // 右边部分更接近
        } else {
            right = mid;     // 左边部分更接近
        }
    }

    return vector<i64>(nums.begin() + left, nums.begin() + left + k);
}
```

```cpp
/*****************************************************************************
 * 7. 字符出现至少k次的子字符串（LS1259）
 * 问题：找到最长子串，其中每个字符至少出现k次
 * 解法：分治+双指针，统计频率后递归处理
 * 时间复杂度：O(nlogn)，空间复杂度：O(n)
 *****************************************************************************/
i64 longest_substring_at_least_k(string s, i64 k) {
    i64 n = s.size();
    if (n < k) return 0;

    // 统计字符频率
    vector<i64> freq(26, 0);
    for (char c : s) freq[c - 'a']++;

    // 找到第一个不满足条件的字符（出现次数<k）
    i64 split = 0;
    while (split < n && freq[s[split] - 'a'] >= k) {
        split++;
    }

    if (split == n) return n;  // 整个字符串都满足条件

    // 递归处理左右部分
    i64 left_len = longest_substring_at_least_k(s.substr(0, split), k);

    // 跳过所有不满足条件的字符
    while (split < n && freq[s[split] - 'a'] < k) {
        split++;
    }

    i64 right_len = longest_substring_at_least_k(s.substr(split), k);

    return max(left_len, right_len);
}


/*****************************************************************************
 * 8. 统计稳定子数组的数目（LS1254）
 * 问题：统计极差（最大值-最小值）不超过k的子数组数量
 * 解法：滑动窗口+单调队列维护最大值和最小值
 * 时间复杂度：O(n)，空间复杂度：O(n)
 *****************************************************************************/
i64 count_stable_subarrays(vector<i64>& nums, i64 k) {
    i64 n = nums.size();
    i64 count = 0;

    // 维护当前窗口的最大值和最小值
    deque<i64> max_deque, min_deque;
    i64 left = 0;

    for (i64 right = 0; right < n; right++) {
        // 维护最大值队列（递减）
        while (!max_deque.empty() && nums[max_deque.back()] <= nums[right]) {
            max_deque.pop_back();
```

```
        }
        max_deque.push_back(right);

        // 维护最小值队列（递增）
        while (!min_deque.empty() && nums[min_deque.back()] >= nums[right]) {
            min_deque.pop_back();
        }
        min_deque.push_back(right);

        // 收缩左边界直到窗口稳定（极差≤k）
        while (nums[max_deque.front()] - nums[min_deque.front()] > k) {
            if (max_deque.front() == left) max_deque.pop_front();
            if (min_deque.front() == left) min_deque.pop_front();
            left++;
        }

        // 以right结尾的稳定子数组数量
        count += right - left + 1;
    }

    return count;
}


/**************************************************************************
 * 9. 极差不超过k的分割数（LS1258）
 * 问题：统计将数组分割成若干子数组的方案数，每个子数组极差≤k
 * 解法：动态规划+双指针优化
 * 时间复杂度：O(n²)最坏，但双指针优化后接近O(n)，空间复杂度：O(n)
 **************************************************************************/
i64 count_partitions(vector<i64>& nums, i64 k) {
    i64 n = nums.size();
    const i64 MOD = 1e9 + 7;

    // dp[i]: 前i个元素的分割方案数
    vector<i64> dp(n + 1, 0);
    dp[0] = 1;

    for (i64 i = 0; i < n; i++) {
        i64 cur_min = nums[i], cur_max = nums[i];
        i64 j = i;

        // 从i向前找，找到第一个不满足极差≤k的位置
        while (j >= 0) {
            cur_min = min(cur_min, nums[j]);
            cur_max = max(cur_max, nums[j]);

            if (cur_max - cur_min > k) {
                break;  // 极差超过k，更左边的也不可能满足
            }

            dp[i + 1] = (dp[i + 1] + dp[j]) % MOD;
            j--;
        }
    }
```

```
    return dp[n];
}


/**********************************************************************
 * 10. 通用滑动窗口框架
 * 支持多种窗口问题的通用解法
 **********************************************************************/
template<typename Func>
i64 sliding_window_general(vector<i64>& nums, Func condition) {
    i64 n = nums.size();
    i64 left = 0, right = 0;
    i64 result = 0;

    // 窗口状态（根据具体问题定义）
    // unordered_map<i64, i64> window;
    // i64 count = 0;

    while (right < n) {
        // 扩展右边界，更新窗口状态
        // window[nums[right]]++;
        // if (满足某个条件) count++;
        right++;

        // 当窗口不满足条件时，收缩左边界
        while (/* 窗口不满足条件 */) {
            // 更新窗口状态
            // if (移除某个条件) count--;
            // window[nums[left]]--;
            left++;
        }

        // 更新结果（根据具体问题）
        // result = max(result, right - left);
        // 或 result += right - left;
    }

    return result;
}
```

📊 双指针类型总结

类型	特点	适用场景	例题
同向双指针	两个指针同向移动，维护一个区间	滑动窗口问题	最短子数组和、无重复字符子串
相向双指针	两个指针从两端向中间移动	已排序数组的查找	两数之和、三数之和
快慢指针	一个快一个慢，速度不同	链表问题、环检测	链表中点、环形链表

📊 窗口收缩条件策略

问题类型	窗口收缩时机	目标
最小窗口	满足条件时收缩	找到满足条件的最短区间
最大窗口	不满足条件时收缩	找到满足条件的最长区间
统计数量	每次右移都更新	统计满足条件的区间总数

📊 常见问题模式与解法

模式	关键点	算法	时间复杂度
最短/最长子数组	维护窗口和，动态调整	滑动窗口	O(n)
包含所有字符	哈希表统计频率，valid计数器	滑动窗口	O(n)
无重复字符	字符位置记录，遇重复移动	滑动窗口	O(n)
已排序数组查找	利用有序性，双指针搜索	相向双指针	O(n)
极值统计	单调队列维护最大/最小值	滑动窗口+单调队列	O(n)
分治处理	递归处理不满足条件的部分	分治+双指针	O(nlogn)

📊 关联题目索引

题目编号	题目名称	核心算法	难度
LS1256	两数之和（已排序）	相向双指针	⭐
LS1257	两数之差（已排序）	同向双指针	⭐
LS1255	k个最接近的数	二分+双指针	⭐⭐
P1638	包含所有字符的最短子串	滑动窗口+哈希表	⭐⭐⭐
LS1259	字符出现至少k次	分治+双指针	⭐⭐
LS1260	求和游戏	前缀和+双指针	⭐⭐⭐
LS1254	统计稳定子数组	单调队列+双指针	⭐⭐⭐
LS1258	极差不超过k的分割数	DP+双指针	⭐⭐⭐

【各算法时间复杂度对比】

算法	时间复杂度	空间复杂度	适用场景
最短子数组和	O(n)	O(1)	连续子数组和问题
无重复字符子串	O(n)	O(字符集大小)	字符串去重问题
最小覆盖子串	O(n)	O(字符集大小)	包含所有字符问题
两数之和（排序）	O(n)	O(1)	已排序数组查找
两数之差（排序）	O(n)	O(1)	已排序数组差值查找

算法	时间复杂度	空间复杂度	适用场景
k个最接近的数	O(logn + k)	O(1)	已排序数组查找
字符出现至少k次	O(nlogn)	O(n)	分治字符串问题
统计稳定子数组	O(n)	O(n)	极差限制问题
极差分割数	$O(n^2)$最坏	O(n)	分割方案统计

【核心算法详解】

1. 滑动窗口框架（`min_subarray_len`）

```
// 通用模板
i64 left = 0, sum = 0;
for (i64 right = 0; right < n; right++) {
    sum += nums[right];  // 扩展

    while (sum >= target) {  // 满足条件时收缩
        // 更新答案
        sum -= nums[left];
        left++;
    }
}
```

2. 最小覆盖子串（`min_window_containing_all`）

```
unordered_map<char, i64> need, window;
i64 valid = 0;  // 关键：记录满足条件的字符种类数

while (right < s.size()) {
    // 扩展右边界
    if (need.count(c)) {
        window[c]++;
        if (window[c] == need[c]) valid++;  // 该字符已满足条件
    }

    // 收缩左边界
    while (valid == need.size()) {
        // 更新最小窗口
        if (need.count(d)) {
            if (window[d] == need[d]) valid--;  // 该字符不再满足条件
            window[d]--;
        }
    }
}
```

3. 单调队列维护极值（`count_stable_subarrays`）

```cpp
// 最大值队列（递减）
while (!max_deque.empty() && nums[max_deque.back()] <= nums[right]) {
    max_deque.pop_back();  // 维护单调性
}
max_deque.push_back(right);

// 最小值队列（递增）
while (!min_deque.empty() && nums[min_deque.back()] >= nums[right]) {
    min_deque.pop_back();  // 维护单调性
}
min_deque.push_back(right);
```

【测试用例设计】
代码包含多种测试用例：

1. **基础测试**：验证基本功能
2. **边界测试**：空数组、单个元素、极值情况
3. **性能测试**：10万数据量测试
4. **交互测试**：用户自定义输入

【扩展与优化】

1. 空间优化

- 大部分算法都是O(1)或O(n)空间
- 字符统计可使用固定大小数组（ASCII 256，小写字母 26）
- 滚动数组优化DP

2. 通用框架

- `sliding_window_general` 模板函数支持自定义条件
- 可根据具体问题调整窗口状态维护方式

3. 错误处理

- 检查输入合法性
- 边界条件处理（空数组、n<k等）
- 模运算防止溢出

【学习建议】

1. **从简单到复杂**：先掌握基本滑动窗口，再学复杂窗口维护
2. **分类练习**：按双指针类型分组练习
3. **画图辅助**：画出指针移动过程，理解收缩条件
4. **模版化思维**：记住通用框架，根据具体问题调整

【常见错误】

1. **边界条件**：数组越界、空指针
2. **收缩条件**：while还是if？满足条件还是不满足条件？
3. **状态更新**：先更新指针还是先更新状态？
4. **初始值**：DP数组、队列的初始化

【实用技巧】

1. **调试输出**：在循环中打印左右指针和关键状态
2. **小数据测试**：用手算验证小数据
3. **对拍测试**：与暴力解法对比结果
4. **复杂度分析**：确保算法在数据范围内可行

5.7 根号算法（课程19）

📦 核心代码模板

```
// ==================== 根号算法 ====================

/**********************************************************************
 * 1. 分块数组（区间求和+单点更新）
 * 问题：支持区间求和查询和单点更新操作
 * 解法：将数组分成√n块，预处理每块的和
 * 时间复杂度：查询O(√n)，更新O(1)，空间复杂度：O(n)
 **********************************************************************/
class SqrtDecomposition {
private:
    i64 n;                      // 数组大小
    i64 block_size;             // 块大小，通常为√n
    i64 block_count;            // 块数量
    vector<i64> arr;            // 原始数组
    vector<i64> block_sum;      // 每块的和

public:
    // 构造函数，从给定数组构建分块结构
    SqrtDecomposition(vector<i64>& nums) {
        arr = nums;
        n = nums.size();
        block_size = sqrt(n) + 1;  // 向上取整
        block_count = (n + block_size - 1) / block_size;
        block_sum.assign(block_count, 0);

        // 预处理每块的和
        for (i64 i = 0; i < n; i++) {
            i64 block_idx = i / block_size;
            block_sum[block_idx] += nums[i];
        }
    }

    // 单点更新：将位置idx的值更新为val
    void update(i64 idx, i64 val) {
        if (idx < 0 || idx >= n) return;

        i64 block_idx = idx / block_size;
        i64 old_val = arr[idx];

        // 更新块的和
        block_sum[block_idx] += val - old_val;
```

```
        // 更新原始数组
        arr[idx] = val;
    }

    // 区间查询：返回[l, r]范围内元素的和
    i64 query(i64 l, i64 r) {
        if (l < 0 || r >= n || l > r) return 0;

        i64 sum = 0;
        i64 start_block = l / block_size;
        i64 end_block = r / block_size;

        if (start_block == end_block) {
            // 查询区间完全在一个块内，暴力计算
            for (i64 i = l; i <= r; i++) {
                sum += arr[i];
            }
        } else {
            // 左碎块（不完整的块）
            for (i64 i = l; i < (start_block + 1) * block_size; i++) {
                sum += arr[i];
            }

            // 完整块（使用预处理结果）
            for (i64 b = start_block + 1; b < end_block; b++) {
                sum += block_sum[b];
            }

            // 右碎块（不完整的块）
            for (i64 i = end_block * block_size; i <= r; i++) {
                sum += arr[i];
            }
        }

        return sum;
    }

    // 获取原始数组（用于调试）
    vector<i64> get_array() {
        return arr;
    }

    // 打印分块信息（用于调试）
    void print_blocks() {
        cout << "分块信息: n=" << n << ", block_size=" << block_size
             << ", block_count=" << block_count << "\n";
        cout << "块的和: [";
        for (i64 i = 0; i < block_count; i++) {
            if (i > 0) cout << ", ";
            cout << block_sum[i];
        }
        cout << "]\n";
    }
};
```

```cpp
/*****************************************************************************
 * 2. 分块求区间最大值
 * 问题：支持区间最大值查询
 * 解法：预处理每块的最大值
 * 时间复杂度：查询O(√n)，空间复杂度：O(n)
 *****************************************************************************/
class SqrtDecompositionMax {
private:
    i64 n;
    i64 block_size;
    i64 block_count;
    vector<i64> arr;
    vector<i64> block_max;  // 每块的最大值

public:
    SqrtDecompositionMax(vector<i64>& nums) {
        arr = nums;
        n = nums.size();
        block_size = sqrt(n) + 1;
        block_count = (n + block_size - 1) / block_size;
        block_max.assign(block_count, LLONG_MIN);

        // 预处理每块的最大值
        for (i64 i = 0; i < n; i++) {
            i64 block_idx = i / block_size;
            block_max[block_idx] = max(block_max[block_idx], nums[i]);
        }
    }

    // 区间最大值查询
    i64 query_max(i64 l, i64 r) {
        if (l < 0 || r >= n || l > r) return LLONG_MIN;

        i64 max_val = LLONG_MIN;
        i64 start_block = l / block_size;
        i64 end_block = r / block_size;

        if (start_block == end_block) {
            // 在同一块内，暴力查询
            for (i64 i = l; i <= r; i++) {
                max_val = max(max_val, arr[i]);
            }
        } else {
            // 左碎块
            for (i64 i = l; i < (start_block + 1) * block_size; i++) {
                max_val = max(max_val, arr[i]);
            }

            // 完整块
            for (i64 b = start_block + 1; b < end_block; b++) {
                max_val = max(max_val, block_max[b]);
            }

            // 右碎块
            for (i64 i = end_block * block_size; i <= r; i++) {
                max_val = max(max_val, arr[i]);
```

```
            }
        }

        return max_val;
    }

    // 单点更新（如果支持的话）
    void update(i64 idx, i64 val) {
        if (idx < 0 || idx >= n) return;

        i64 block_idx = idx / block_size;
        arr[idx] = val;

        // 重新计算该块的最大值（可能需要重构整个块）
        i64 block_start = block_idx * block_size;
        i64 block_end = min(block_start + block_size, n);
        i64 new_max = LLONG_MIN;

        for (i64 i = block_start; i < block_end; i++) {
            new_max = max(new_max, arr[i]);
        }

        block_max[block_idx] = new_max;
    }
};


/**************************************************************************
 * 3. 莫队算法模板（离线区间查询）
 * 问题：处理大量区间查询，如区间内相同元素对数
 * 解法：将查询按块排序，通过移动指针更新答案
 * 时间复杂度：O((n+q)√n)，空间复杂度：O(n)
 **************************************************************************/
struct Query {
    i64 l;      // 查询左边界
    i64 r;      // 查询右边界
    i64 idx;    // 查询原始索引
    i64 block;  // 所属块（用于排序）

    // 排序函数：先按块排序，块内按右边界排序（奇偶优化）
    bool operator<(const Query& other) const {
        if (block != other.block) {
            return block < other.block;
        }
        // 奇偶排序优化：奇数块右边界递增，偶数块右边界递减
        return (block & 1) ? (r < other.r) : (r > other.r);
    }
};

class MoAlgorithm {
private:
    i64 n;                      // 数组大小
    i64 q;                      // 查询数量
    i64 block_size;             // 块大小
    vector<i64> arr;            // 原始数组（离散化后）
    vector<Query> queries;      // 所有查询
```

```
    vector<i64> freq;           // 频率数组
    i64 current_answer;         // 当前区间答案

    // 添加一个元素到当前区间
    void add(i64 idx) {
        i64 val = arr[idx];
        // 示例：计算相同元素对数（组合数C(freq, 2)）
        current_answer -= freq[val] * (freq[val] - 1) / 2;
        freq[val]++;
        current_answer += freq[val] * (freq[val] - 1) / 2;
    }

    // 从当前区间移除一个元素
    void remove(i64 idx) {
        i64 val = arr[idx];
        current_answer -= freq[val] * (freq[val] - 1) / 2;
        freq[val]--;
        current_answer += freq[val] * (freq[val] - 1) / 2;
    }

public:
    // 构造函数：初始化莫队算法
    MoAlgorithm(vector<i64>& nums, vector<pair<i64, i64>>& queries_input) {
        arr = nums;
        n = nums.size();
        q = queries_input.size();
        block_size = sqrt(n) + 1;

        // 离散化（如果数值范围很大）
        vector<i64> sorted = nums;
        sort(sorted.begin(), sorted.end());
        sorted.erase(unique(sorted.begin(), sorted.end()), sorted.end());
        unordered_map<i64, i64> mapping;

        for (i64 i = 0; i < sorted.size(); i++) {
            mapping[sorted[i]] = i;
        }

        for (i64& num : arr) {
            num = mapping[num];
        }

        // 初始化频率数组
        freq.assign(sorted.size(), 0);

        // 构建查询结构
        for (i64 i = 0; i < q; i++) {
            i64 l = queries_input[i].first;
            i64 r = queries_input[i].second;
            queries.push_back({l, r, i, l / block_size});
        }

        // 排序查询
        sort(queries.begin(), queries.end());
    }
```

```
    // 执行莫队算法，返回所有查询的答案
    vector<i64> solve() {
        vector<i64> answers(q, 0);
        i64 current_l = 0;   // 当前区间左边界
        i64 current_r = -1;  // 当前区间右边界（初始为空）
        current_answer = 0;

        for (const auto& query : queries) {
            // 扩展右边界
            while (current_r < query.r) {
                current_r++;
                add(current_r);
            }

            // 收缩右边界
            while (current_r > query.r) {
                remove(current_r);
                current_r--;
            }

            // 收缩左边界
            while (current_l < query.l) {
                remove(current_l);
                current_l++;
            }

            // 扩展左边界
            while (current_l > query.l) {
                current_l--;
                add(current_l);
            }

            // 记录答案
            answers[query.idx] = current_answer;
        }

        return answers;
    }

    // 小Z的袜子问题：计算概率
    vector<pair<i64, i64>> solve_socks_probability() {
        vector<i64> raw_answers = solve();
        vector<pair<i64, i64>> answers(q);

        for (i64 i = 0; i < q; i++) {
            i64 l = queries[i].l;
            i64 r = queries[i].r;
            i64 len = r - l + 1;

            // 总对数：从len个中选2个
            i64 total_pairs = len * (len - 1) / 2;
            i64 same_color_pairs = raw_answers[i];

            if (same_color_pairs == 0) {
                answers[i] = {0, 1};  // 0/1
            } else {
```

```
                // 约分
                i64 g = __gcd(same_color_pairs, total_pairs);
                answers[i] = {same_color_pairs / g, total_pairs / g};
            }
        }

        return answers;
    }
};


/**********************************************************************
 * 4. 数列找不同（P3901）- 区间是否有重复元素
 * 问题：判断每个查询区间内是否有重复元素
 * 解法：莫队算法维护重复元素计数
 * 时间复杂度：O((n+q)√n)，空间复杂度：O(n)
 **********************************************************************/
class DistinctChecker {
private:
    i64 n, q, block_size;
    vector<i64> arr;
    vector<Query> queries;
    vector<i64> freq;
    i64 duplicate_count;  // 重复元素个数

    void add(i64 idx) {
        i64 val = arr[idx];
        if (freq[val] == 1) duplicate_count++;
        freq[val]++;
    }

    void remove(i64 idx) {
        i64 val = arr[idx];
        freq[val]--;
        if (freq[val] == 1) duplicate_count--;
    }

public:
    DistinctChecker(vector<i64>& nums, vector<pair<i64, i64>>& queries_input) {
        arr = nums;
        n = nums.size();
        q = queries_input.size();
        block_size = sqrt(n) + 1;

        // 离散化
        vector<i64> sorted = nums;
        sort(sorted.begin(), sorted.end());
        sorted.erase(unique(sorted.begin(), sorted.end()), sorted.end());
        unordered_map<i64, i64> mapping;
        for (i64 i = 0; i < sorted.size(); i++) {
            mapping[sorted[i]] = i;
        }
        for (i64& num : arr) num = mapping[num];

        // 初始化
        freq.assign(sorted.size(), 0);
```

```
        for (i64 i = 0; i < q; i++) {
            i64 l = queries_input[i].first;
            i64 r = queries_input[i].second;
            queries.push_back({l, r, i, l / block_size});
        }
        sort(queries.begin(), queries.end());
    }

    // 返回布尔数组：每个查询区间是否有重复元素
    vector<bool> solve() {
        vector<bool> answers(q, false);
        i64 current_l = 0, current_r = -1;
        duplicate_count = 0;

        for (const auto& query : queries) {
            // 调整区间
            while (current_r < query.r) { current_r++; add(current_r); }
            while (current_r > query.r) { remove(current_r); current_r--; }
            while (current_l < query.l) { remove(current_l); current_l++; }
            while (current_l > query.l) { current_l--; add(current_l); }

            answers[query.idx] = (duplicate_count > 0);
        }

        return answers;
    }
};


/**********************************************************************
 * 5. 根号分治（根据频率分类处理）
 * 问题：某些问题中，高频元素和低频元素需要不同处理
 * 解法：阈值设为√n，高频元素特殊处理，低频元素暴力处理
 * 时间复杂度：O(n√n)，空间复杂度：O(n)
 **********************************************************************/
class SqrtDecompositionByFrequency {
private:
    i64 n;
    i64 threshold;  // 阈值，通常设为√n
    vector<i64> arr;
    unordered_map<i64, i64> freq;
    unordered_set<i64> heavy_elements;  // 高频元素集合

public:
    SqrtDecompositionByFrequency(vector<i64>& nums) {
        arr = nums;
        n = nums.size();
        threshold = sqrt(n) + 1;

        // 统计频率
        for (i64 num : nums) {
            freq[num]++;
        }

        // 识别高频元素（出现次数≥阈值）
        for (const auto& [num, cnt] : freq) {
```

```
            if (cnt >= threshold) {
                heavy_elements.insert(num);
            }
        }
    }

    // 示例：查询区间内某个值的出现次数
    i64 query_frequency(i64 l, i64 r, i64 target) {
        i64 count = 0;

        if (heavy_elements.count(target)) {
            // 高频元素：预处理前缀和
            // 这里简化处理，实际可能需要更复杂的数据结构
            for (i64 i = l; i <= r; i++) {
                if (arr[i] == target) count++;
            }
        } else {
            // 低频元素：直接统计
            for (i64 i = l; i <= r; i++) {
                if (arr[i] == target) count++;
            }
        }

        return count;
    }

    // 获取高频元素列表
    vector<i64> get_heavy_elements() {
        return vector<i64>(heavy_elements.begin(), heavy_elements.end());
    }

    // 判断一个元素是否为高频元素
    bool is_heavy(i64 num) {
        return heavy_elements.count(num);
    }
};


/**************************************************************************
 * 6. 带修改的莫队（支持单点修改）
 * 问题：在区间查询的基础上支持单点修改
 * 解法：增加时间维度，按(n^(2/3))分块
 * 时间复杂度：O(n^(5/3))，空间复杂度：O(n)
 **************************************************************************/
struct Update {
    i64 pos;    // 修改位置
    i64 old_val; // 旧值
    i64 new_val; // 新值
};

struct QueryWithTime {
    i64 l, r;       // 查询区间
    i64 t;          // 时间戳（在哪个修改之后）
    i64 idx;        // 查询索引
    i64 block_l;    // l所在块
    i64 block_r;    // r所在块
```

```
    bool operator<(const QueryWithTime& other) const {
        if (block_l != other.block_l) return block_l < other.block_l;
        if (block_r != other.block_r) return block_r < other.block_r;
        return t < other.t;
    }
};

class MoWithUpdates {
private:
    i64 n, q, update_count;
    i64 block_size;
    vector<i64> arr;
    vector<QueryWithTime> queries;
    vector<Update> updates;
    vector<i64> freq;
    i64 current_answer;

    void add(i64 idx) {
        i64 val = arr[idx];
        current_answer -= freq[val] * (freq[val] - 1) / 2;
        freq[val]++;
        current_answer += freq[val] * (freq[val] - 1) / 2;
    }

    void remove(i64 idx) {
        i64 val = arr[idx];
        current_answer -= freq[val] * (freq[val] - 1) / 2;
        freq[val]--;
        current_answer += freq[val] * (freq[val] - 1) / 2;
    }

    void apply_update(const Update& upd, i64 l, i64 r) {
        i64 pos = upd.pos;
        if (pos >= l && pos <= r) {
            // 修改在查询区间内，需要更新答案
            remove(pos);
            arr[pos] = upd.new_val;
            add(pos);
        } else {
            // 修改不在查询区间内，只更新数组
            arr[pos] = upd.new_val;
        }
    }

    void undo_update(const Update& upd, i64 l, i64 r) {
        i64 pos = upd.pos;
        if (pos >= l && pos <= r) {
            remove(pos);
            arr[pos] = upd.old_val;
            add(pos);
        } else {
            arr[pos] = upd.old_val;
        }
    }
```

```
public:
    MoWithUpdates(vector<i64>& nums, vector<pair<i64, i64>>& queries_input,
                  vector<tuple<i64, i64>>& updates_input) {
        arr = nums;
        n = nums.size();
        q = queries_input.size();
        update_count = updates_input.size();
        block_size = pow(n, 2.0/3.0) + 1;  // 带修改的莫队块大小不同

        // 离散化（略）

        // 构建查询和更新
        for (i64 i = 0; i < q; i++) {
            i64 l = queries_input[i].first;
            i64 r = queries_input[i].second;
            queries.push_back({l, r, 0, i, l / block_size, r / block_size});
        }

        for (const auto& upd : updates_input) {
            i64 pos = get<0>(upd);
            i64 new_val = get<1>(upd);
            i64 old_val = arr[pos];
            updates.push_back({pos, old_val, new_val});
            arr[pos] = new_val;  // 临时应用修改
        }

        // 重置数组
        arr = nums;
        freq.assign(/*离散化后的大小*/ 100001, 0);
    }

    vector<i64> solve() {
        vector<i64> answers(q, 0);
        sort(queries.begin(), queries.end());

        i64 current_l = 0, current_r = -1, current_t = 0;
        current_answer = 0;

        for (const auto& query : queries) {
            // 调整时间
            while (current_t < query.t) {
                apply_update(updates[current_t], current_l, current_r);
                current_t++;
            }
            while (current_t > query.t) {
                current_t--;
                undo_update(updates[current_t], current_l, current_r);
            }

            // 调整空间
            while (current_r < query.r) { current_r++; add(current_r); }
            while (current_r > query.r) { remove(current_r); current_r--; }
            while (current_l < query.l) { remove(current_l); current_l++; }
            while (current_l > query.l) { current_l--; add(current_l); }

            answers[query.idx] = current_answer;
```

```
        }

        return answers;
    }
};


/**************************************************************************
 * 7. 分块求区间众数（近似算法）
 * 问题：求区间出现次数最多的元素
 * 解法：分块预处理每块之间的众数，查询时结合碎块统计
 * 时间复杂度：O(n√n)，空间复杂度：O(n)
 **************************************************************************/
class SqrtDecompositionMode {
private:
    i64 n, block_size, block_count;
    vector<i64> arr;
    vector<i64> compressed;  // 离散化后数组
    vector<vector<i64>> block_mode;  // 块间众数
    vector<unordered_map<i64, i64>> block_freq;  // 每块频率

    // 离散化
    void compress() {
        vector<i64> sorted = arr;
        sort(sorted.begin(), sorted.end());
        sorted.erase(unique(sorted.begin(), sorted.end()), sorted.end());
        unordered_map<i64, i64> mapping;

        for (i64 i = 0; i < sorted.size(); i++) {
            mapping[sorted[i]] = i;
        }

        compressed.resize(n);
        for (i64 i = 0; i < n; i++) {
            compressed[i] = mapping[arr[i]];
        }
    }

public:
    SqrtDecompositionMode(vector<i64>& nums) {
        arr = nums;
        n = nums.size();
        block_size = sqrt(n) + 1;
        block_count = (n + block_size - 1) / block_size;

        compress();

        // 预处理每块频率
        block_freq.resize(block_count);
        for (i64 i = 0; i < n; i++) {
            i64 block_idx = i / block_size;
            block_freq[block_idx][compressed[i]]++;
        }

        // 预处理块间众数（简化版本）
        // 实际实现更复杂，可能需要O(n√n)预处理
```

```
    }

    // 查询区间众数（近似）
    i64 query_mode(i64 l, i64 r) {
        unordered_map<i64, i64> temp_freq;
        i64 max_freq = 0;
        i64 mode = -1;

        // 简单暴力版本（实际需要更高效实现）
        for (i64 i = l; i <= r; i++) {
            i64 val = compressed[i];
            temp_freq[val]++;
            if (temp_freq[val] > max_freq) {
                max_freq = temp_freq[val];
                mode = arr[i];  // 返回原始值
            }
        }

        return mode;
    }
};


/**************************************************************************
 * 8. 根号算法工具类
 * 提供常用的根号算法辅助函数
 **************************************************************************/
class SqrtAlgorithmUtils {
public:
    // 计算合适的块大小
    static i64 calculate_block_size(i64 n) {
        return sqrt(n) + 1;
    }

    // 计算块数量
    static i64 calculate_block_count(i64 n, i64 block_size) {
        return (n + block_size - 1) / block_size;
    }

    // 获取元素所在的块索引
    static i64 get_block_index(i64 idx, i64 block_size) {
        return idx / block_size;
    }

    // 获取块的起始索引
    static i64 get_block_start(i64 block_idx, i64 block_size) {
        return block_idx * block_size;
    }

    // 获取块的结束索引（不包含）
    static i64 get_block_end(i64 block_idx, i64 block_size, i64 n) {
        return min((block_idx + 1) * block_size, n);
    }

    // 判断区间是否完全在一个块内
    static bool is_same_block(i64 l, i64 r, i64 block_size) {
```

```
        return l / block_size == r / block_size;
    }

    // 莫队算法排序比较函数（带奇偶优化）
    static bool mo_compare(const Query& a, const Query& b) {
        if (a.block != b.block) return a.block < b.block;
        return (a.block & 1) ? (a.r < b.r) : (a.r > b.r);
    }
};
```

根号算法类型总结

算法类型	核心思想	时间复杂度	空间复杂度	适用场景
分块数组	数组分√n块，预处理块信息	查询O(√n)，更新O(1)	O(n)	区间查询+单点更新
莫队算法	离线查询按块排序，移动指针	O((n+q)√n)	O(n)	离线区间统计查询
根号分治	高频/低频元素不同处理	O(n√n)	O(n)	元素频率差异大
带修改莫队	增加时间维度处理修改	O(n^(5/3))	O(n)	带修改的离线查询

块大小选择策略

问题类型	推荐块大小	复杂度分析	说明
普通分块	√n	O(√n)查询，O(1)更新	平衡查询和更新
莫队算法	√n	O((n+q)√n)	平衡左右指针移动
带修改莫队	n^(2/3)	O(n^(5/3))	平衡空间和时间维度
根号分治	√n	O(n√n)	阈值设为√n

莫队算法指针移动顺序

操作	代码实现	说明
扩展右边界	`while (cur_r < q.r) add(++cur_r);`	向右移动右指针
收缩右边界	`while (cur_r > q.r) remove(cur_r--);`	向左移动右指针
收缩左边界	`while (cur_l < q.l) remove(cur_l++);`	向右移动左指针
扩展左边界	`while (cur_l > q.l) add(--cur_l);`	向左移动左指针

常见问题模式与解法

问题模式	关键点	推荐算法	时间复杂度
区间和查询+单点更新	不支持线段树时	分块数组	O(√n)查询

问题模式	关键点	推荐算法	时间复杂度
离线区间统计查询	统计频率、相同对数等	莫队算法	O((n+q)√n)
判断区间是否有重复	频率统计为0/1/2	莫队或分块	O((n+q)√n)
带修改的区间查询	需要支持单点修改	带修改莫队	O(n^(5/3))
高频低频分类处理	元素出现次数差异大	根号分治	O(n√n)

📊 关联题目索引

题目编号	题目名称	核心算法	难度
LS1261	数论分块	基础分块思想	⭐⭐
LS1262	多维数论分块	二维分块扩展	⭐⭐⭐
P3901	数列找不同	莫队基础	⭐⭐
P1494	小Z的袜子	莫队概率计算	⭐⭐⭐
P2709	小B的询问	莫队维护平方和	⭐⭐⭐
LS1263	智力与模数	根号分治应用	⭐⭐⭐⭐

【核心算法详解】

1. 分块数组框架（`SqrtDecomposition`）

```
// 构造
block_size = sqrt(n) + 1;
for (i = 0; i < n; i++) {
    block_idx = i / block_size;
    block_sum[block_idx] += nums[i];
}

// 查询
if (同一块内) {
    // 暴力计算
} else {
    // 左碎块暴力 + 完整块使用block_sum + 右碎块暴力
}

// 更新
block_idx = idx / block_size;
block_sum[block_idx] += val - arr[idx];
arr[idx] = val;
```

2. 莫队算法框架（`MoAlgorithm`）

```
// 查询排序
sort(queries.begin(), queries.end(), [](Query a, Query b) {
    if (a.block != b.block) return a.block < b.block;
    return (a.block & 1) ? (a.r < b.r) : (a.r > b.r);
```

```
});

// 处理查询
for (query : queries) {
    while (cur_r < query.r) add(++cur_r);
    while (cur_r > query.r) remove(cur_r--);
    while (cur_l < query.l) remove(cur_l++);
    while (cur_l > query.l) add(--cur_l);
    answers[query.idx] = current_answer;
}
```

3. 根号分治思想（`SqrtDecompositionByFrequency`）

```
// 设置阈值
threshold = sqrt(n);

// 统计频率并分类
for (num : nums) freq[num]++;
for (auto& [num, cnt] : freq) {
    if (cnt >= threshold) {
        heavy_elements.insert(num);  // 高频元素
    }
}

// 处理查询时根据频率选择策略
if (heavy_elements.count(target)) {
    // 高频元素：使用预处理结构
} else {
    // 低频元素：暴力统计
}
```

【测试用例设计】
代码包含多种测试用例：

1. **基础测试**：验证基本功能
2. **边界测试**：小数组、大查询、边界条件
3. **性能测试**：10万数据量，1万查询
4. **交互测试**：用户自定义输入

【扩展与优化】

1. 空间优化

- 分块数组：额外空间O(n)
- 莫队算法：频率数组O(值域大小)
- 根号分治：存储高频元素集合

2. 时间优化

- 块大小调整：根据实际情况微调
- 奇偶排序：减少指针移动距离
- 预处理：高频元素特殊预处理

3. 功能扩展

- 支持区间更新：懒标记技术
- 支持多种聚合：最大值、最小值、众数等
- 支持二维分块：处理矩阵问题

【学习建议】

1. **理解分治思想**：掌握将问题分解为√n规模子问题的思想
2. **掌握模板**：记住分块、莫队的通用模板
3. **分析复杂度**：理解为什么是O(√n)或O(n√n)复杂度
4. **实践应用**：从简单问题开始，逐步尝试复杂问题

【常见错误】

1. **块大小计算**：忘记+1导致块数量不足
2. **边界处理**：查询区间越界未检查
3. **指针移动**：莫队中add/remove顺序错误
4. **离散化**：忘记离散化导致内存过大
5. **排序比较**：莫队排序函数写错

【实用技巧】

1. **调试输出**：打印块信息、查询排序结果等
2. **复杂度验证**：通过大数据测试验证实际性能
3. **对拍测试**：与暴力解法对比结果
4. **阈值调整**：根据实际数据特征调整√n值
5. **预处理优化**：对高频信息进行额外预处理

📊 考核策略与时间分配

三小时考核时间分配表

时间段	任务	目标	建议
0-10分钟	审题规划	了解所有题目	快速浏览，标记难度
10-70分钟	基础题攻坚	完成3-4题	优先做熟悉的题目
70-130分钟	核心题突破	完成2-3题	仔细分析，避免失误
130-160分钟	难题挑战	尝试1-2题	争取部分分数
160-180分钟	检查调试	确保正确性	检查边界和格式

题目难度识别指南

数据特征	可能算法	时间预估	策略
n ≤ 20	暴力枚举/DFS	15分钟	状态压缩，注意剪枝

数据特征	可能算法	时间预估	策略
n ≤ 1000	二维DP/Floyd	20分钟	注意空间复杂度
n ≤ 10^5	贪心/双指针	15分钟	维护窗口状态
区间查询	前缀和/BIT	10分钟	套用模板
最短路	Dijkstra/SPFA	20分钟	注意负权判断
数学公式	数论分块	25分钟	推导公式，注意边界

各模块时间分配建议

1. **基础算法**（30分）：目标25-30分钟
    - 递归/搜索：5-10分钟
    - 排序/STL：5-10分钟
    - 简单DP：10-15分钟
2. **核心算法**（40分）：目标50-60分钟
    - 动态规划：15-20分钟
    - 数据结构：10-15分钟
    - 图论算法：15-20分钟
    - 贪心/双指针：10-15分钟
3. **高级算法**（30分）：目标30-40分钟
    - 数学/数论：15-20分钟
    - 贡献思维：10-15分钟
    - 综合应用：10-15分钟

🚨 常见错误与调试技巧

编译错误预防模板

```
#include <bits/stdc++.h>
using namespace std;
using i64 = long long;  // 统一使用i64防止溢出

// 输入输出加速（必须放在main开头）
int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    cout.tie(nullptr);

    // 代码主体

    return 0;
}

// 调试宏（提交前注释掉）
#define LOCAL
```

```
#ifdef LOCAL
#define debug(...) cerr << "[" << #__VA_ARGS__ << "]:", debug_out(__VA_ARGS__)
#else
#define debug(...) 42
#endif

// 向量输出辅助函数
template<typename T>
void print_vector(const vector<T>& v) {
    cerr << "[";
    for (size_t i = 0; i < v.size(); i++) {
        if (i > 0) cerr << ", ";
        cerr << v[i];
    }
    cerr << "]" << "\n";
}
```

运行时错误检查清单

```
// 1. 数组越界检查
assert(index >= 0 && index < n);

// 2. 除零检查
if (divisor == 0) {
    cerr << "除零错误! " << "\n";
    return 0;
}

// 3. 整数溢出检查
if (a > LLONG_MAX / b) {
    cerr << "乘法溢出! " << "\n";
    return 0;
}

// 4. 递归深度检查
if (depth > 1000) {
    cerr << "递归深度过大! " << "\n";
    return 0;
}

// 5. 空指针/空容器检查
if (container.empty()) {
    return 0;  // 或适当处理
}

// 6. 输入合法性检查
i64 n;
cin >> n;
if (n <= 0) {
    cout << 0 << "\n";
    return 0;
}
```

测试用例设计

```cpp
// 小数据测试（自己设计）
void test_small() {
    vector<i64> nums = {1, 2, 3, 4, 5};
    i64 target = 9;
    i64 result = subarray_sum_equals_k(nums, target);
    cout << "小数据测试: " << result << "\n";
    assert(result == 2);
}

// 边界测试
void test_edge_cases() {
    // 空数组
    vector<i64> empty = {};
    assert(max_subarray_sum(empty) == LLONG_MIN);

    // 单个元素
    vector<i64> single = {5};
    assert(max_subarray_sum(single) == 5);

    // 全负数
    vector<i64> all_neg = {-1, -2, -3};
    assert(max_subarray_sum(all_neg) == -1);
}

// 随机测试
void test_random() {
    srand(time(0));
    for (int t = 0; t < 100; t++) {
        i64 n = rand() % 100 + 1;
        vector<i64> nums(n);
        for (i64 i = 0; i < n; i++) {
            nums[i] = rand() % 1000 - 500;
        }

        // 测试暴力解和优化解是否一致
        i64 brute_result = brute_force(nums);
        i64 optim_result = optimized_solution(nums);

        if (brute_result != optim_result) {
            cerr << "测试失败! " << "\n";
            print_vector(nums);
            cerr << "暴力解: " << brute_result << "\n";
            cerr << "优化解: " << optim_result << "\n";
            return;
        }
    }
    cout << "所有随机测试通过! " << "\n";
}
```

性能优化检查

```cpp
// 时间复杂度验证
auto start = chrono::high_resolution_clock::now();

// 执行代码
```

```
function_to_test();

auto end = chrono::high_resolution_clock::now();
auto duration = chrono::duration_cast<chrono::milliseconds>(end - start);

if (duration.count() > 1000) {  // 超过1秒可能需要优化
    cerr << "警告：执行时间 " << duration.count() << "ms，可能需要优化" << "\n";
}

// 内存使用检查
// 使用vector的capacity可以查看预分配内存
vector<i64> v;
v.reserve(1000000);  // 预分配内存，避免多次扩容
```

考核评分标准解读

详细评分标准（100分制）

评分维度	分值	具体标准	提分技巧
正确性	40分	通过所有测试用例	充分测试边界情况
时间复杂度	20分	算法复杂度最优	根据数据范围选择算法
代码规范	15分	结构清晰，命名规范，注释恰当	函数模块化，命名规范
边界处理	15分	处理空输入、极值等特殊情况	添加assert和条件检查
创新性	10分	优化或创新思路	在注释中说明优化思想

各难度题目目标分数

题目类型	分值范围	目标分数	策略
基础题	20-30分	28-30分	必须全对，仔细检查
核心题	30-40分	32-38分	争取大部分正确，部分优化
难题	30-40分	15-25分	争取基础分数，尝试优化

总分目标分析：

- **及格线**：60分（通过选拔）
- **良好线**：75分（表现良好）
- **优秀线**：85分（表现优秀）
- **顶尖线**：95分+（表现突出）

时间分配建议

题目难度	建议时间	检查时间	总分目标
简单题	15-20分钟	5分钟	满分

题目难度	建议时间	检查时间	总分目标
中等题	25-30分钟	5-10分钟	80%+分数
难题	30-40分钟	10分钟	50%+分数

📝 最后冲刺建议

考前一周复习计划

```
周一：递归与搜索（课程01，04）
  - 重点：DFS/BFS模板，记忆化搜索
  - 题目：LS1028，P1443，LS1117

周二：动态规划（课程10，12，17）
  - 重点：背包问题，线性DP，股票问题
  - 题目：LS1082，LS1016，LS1176-1179

周三：数据结构（课程03，07，13）
  - 重点：STL使用，BIT，线段树
  - 题目：P3374，P1908，LS1227

周四：图论与贪心（课程05，08，18）
  - 重点：最短路，最小生成树，双指针
  - 题目：P4779，LS1276，LS1256

周五：数学与高级算法（课程06，14-16，19-20）
  - 重点：数论分块，筛法，单调栈，前缀和
  - 题目：LS1261，LS1163，LS1199，LS1265

周六：全真模拟考试（3小时）
  - 模拟真实考试环境
  - 严格计时，独立完成
  - 考后分析错题

周日：错题回顾与查漏补缺
  - 复习错题本
  - 强化薄弱环节
  - 心理调整，放松心态心理调整技巧
```

1. **深呼吸法**：遇到难题时，深呼吸3次，冷静思考
2. **时间监控**：每30分钟检查一次进度
3. **先易后难**：按顺序做题，遇到难题先标记，回头再做
4. **检查清单**：最后15分钟按清单逐项检查
5. **积极暗示**：告诉自己"我能行"，保持自信

必备物品清单

- [ ] **证件**：身份证/学生证
- [ ] **文具**：黑色签字笔、铅笔、橡皮
- [ ] **草稿纸**：A4纸若干张

- [ ] **水和小零食**：补充能量
- [ ] **模板代码**：打印的关键模板
- [ ] **手表**：掌握时间
- [ ] **放松的心态**：最重要！

考前注意事项

1. **保证睡眠**：考前保证7-8小时睡眠
2. **合理饮食**：考前不吃油腻食物
3. **提前到场**：提前30分钟到达考场
4. **检查设备**：确认电脑、编译器正常
5. **保存备份**：代码及时保存，防止意外

🏆 成功秘诀总结

五大成功要素

1. **基础扎实**：熟练掌握所有基础算法模板
    - 递归/搜索、DP、数据结构、图论
    - 模板代码信手拈来
2. **思维灵活**：能根据题目特征快速识别算法
    - 观察数据范围
    - 识别问题模式
    - 选择合适算法
3. **代码熟练**：模板代码熟练，减少调试时间
    - 常用STL操作
    - 算法模板熟练
    - 调试技巧掌握
4. **心态稳定**：遇到难题不慌，合理分配时间
    - 时间管理
    - 压力应对
    - 积极心态
5. **检查细致**：最后一定留时间检查边界和格式
    - 边界条件
    - 输入输出格式
    - 常见错误

🚨 考场时间管理黄金法则

30-30-30法则：

- 第一个30分钟：完成基础题
- 第二个30分钟：完成核心题

- 第三个30分钟：攻克难题
- 最后30分钟：检查调试

5分钟原则：

- 如果5分钟没有思路，先跳过
- 如果5分钟调试不出错误，先标记
- 最后统一处理跳过和标记的题目

代码质量提升技巧

1. **模块化编程**：将功能分解为函数
2. **清晰命名**：变量名、函数名要有意义
3. **适当注释**：复杂逻辑添加注释
4. **错误处理**：考虑可能的错误情况
5. **测试充分**：多组数据验证

心态调整秘诀

1. **接受不完美**：不可能所有题目都完美解决
2. **关注过程**：享受解题过程，不只是结果
3. **积极思考**：每个错误都是学习机会
4. **保持自信**：相信自己的训练成果
5. **放松心态**：考试只是学习的一部分

🎉 最后寄语

亲爱的同学们：

信息营选拔不仅考察算法知识，更考察你的学习能力、解决问题的能力和心理素质。这是一次展示自己的机会，也是一次成长的过程。

记住：

- 每道题都是你展示能力的机会
- 每行代码都是你智慧的结晶
- 每次思考都是你成长的足迹

相信自己的训练成果，保持冷静，仔细读题，你一定能在考核中展现出最好的自己！

预祝各位同学在乐知六中信息营选拔中取得优异成绩，开启信息学竞赛的精彩旅程！ 🚀🌟

考核加油，未来可期！ 💪🎉

📋 快速查阅索引

按算法类型查找

算法类型	对应章节	关键模板
递归/搜索	一、1	DFS、BFS、记忆化
动态规划	二	背包、线性DP、股票
数据结构	三	BIT、线段树、STL
图论算法	四	Dijkstra、MST、并查集
数学数论	五、1-3	数论分块、筛法、快速幂
贡献思维	五、4	单调栈、贡献法
前缀优化	五、5	前缀和、差分
双指针	五、6	滑动窗口、相向指针
根号算法	五、7	分块、莫队

按课程编号查找

课程	主要内容	关键题目
01	递归函数	LS1028, LS1209
04	网格BFS	P1443, LS1217
10	背包九讲	LS1082-1088
12	经典DP	LS1016, LS1251
17	状态设计	LS1176-1179
07	树状数组	P3374, P1908
13	区间统计	LS1227, P1886
05	图论最短路	P4779, P4568
21	生成树	LS1276, P1396
15	数论分块	LS1261, P2261
14	筛法	LS1163, LS1164
16	贡献思维	LS1199, LS1244
20	前缀优化	LS1265, LS1264
18	双指针	LS1256, P1638
19	根号算法	P3901, P1494