

=====

文件夹: class102_AdvancedDynamicProgrammingAndCombinatorics

=====

[Markdown 文件]

=====

文件: PROBLEM_LIST.md

=====

Class128 问题清单与扩展题目

 核心题目

1. 苹果和盘子（球盒模型）

- **文件**: Code01_ApplesPlates.java/.cpp/.py
- **题目链接**: <https://www.nowcoder.com/practice/bfd8234bb5e84be0b493656e390bdebf>
- **类型**: 组合数学、动态规划
- **难度**: 中等
- **核心思路**: 组合数学中的球盒模型，使用动态规划解决
- **时间复杂度**: $O(m \times n)$
- **空间复杂度**: $O(m \times n)$
- **是否最优解**:  是

2. 数的划分方法

- **文件**: Code01_SplitNumber.java/.cpp/.py
- **题目链接**: <https://www.luogu.com.cn/problem/P1025>
- **类型**: 组合数学、动态规划
- **难度**: 中等
- **核心思路**: 整数划分问题，使用动态规划解决
- **时间复杂度**: $O(m \times n)$
- **空间复杂度**: $O(m \times n)$
- **是否最优解**:  是

3. 最好的部署

- **文件**: Code02_BestDeploy.java/.cpp/.py
- **类型**: 动态规划（区间 DP/线性 DP）
- **难度**: 困难
- **核心思路**: 线性 DP 优化，状态设计优化
- **时间复杂度**: $O(n)$
- **空间复杂度**: $O(n)$
- **是否最优解**:  是

4. 增加限制的最长公共子序列

- **文件**: Code03_AddLimitLcs.java/.cpp/.py

- **类型**: 动态规划、状态设计优化
- **难度**: 困难
- **核心思路**: 利用输入数据特点优化状态设计
- **时间复杂度**: $O(26 \times n + m^2)$
- **空间复杂度**: $O(n \times 26 + m^2)$
- **是否最优解**: 是

5. 大楼扔鸡蛋问题

- **文件**: Code04_EggDrop.java/.cpp/.py
- **题目链接**: <https://leetcode.cn/problems/super-egg-drop/>
- **类型**: 动态规划优化
- **难度**: 困难
- **核心思路**: DP 状态设计优化，自底向上计算
- **时间复杂度**: $O(k \times n)$
- **空间复杂度**: $O(k)$
- **是否最优解**: 是

6. 相邻必选的子序列最大中位数

- **文件**: Code05_MaximizeMedian1.java/.cpp/.py, Code05_MaximizeMedian2.java/.cpp/.py
- **题目链接**: https://atcoder.jp/contests/abc236/tasks/abc236_e
- **类型**: 二分答案、动态规划
- **难度**: 困难
- **核心思路**: 二分答案 + DP 判定
- **时间复杂度**: $O(n * \log(n) * \log(\max))$
- **空间复杂度**: $O(n)$
- **是否最优解**: 是

7. 将珠子放进背包中

- **文件**: Code06_MarblesInBags.java/.cpp/.py
- **题目链接**: <https://leetcode.cn/problems/put-marbles-in-bags/>
- **类型**: 贪心算法
- **难度**: 中等
- **核心思路**: 贪心策略，排序后取极值
- **时间复杂度**: $O(n \times \log(n))$
- **空间复杂度**: $O(n)$
- **是否最优解**: 是

8. 爬楼梯问题

- **文件**: Code07_ClimbingStairs.java/.cpp/.py
- **题目链接**: <https://leetcode.cn/problems/climbing-stairs/>
- **类型**: 动态规划、空间优化
- **难度**: 简单
- **核心思路**: 经典动态规划问题，使用滚动数组优化空间复杂度

- **时间复杂度**: $O(n)$
- **空间复杂度**: $O(1)$
- **是否最优解**: 是

9. 分割数组的最大值

- **文件**: Code08_SplitArray. java/.cpp/.py
- **题目链接**: <https://leetcode.cn/problems/split-array-largest-sum/>
- **类型**: 二分答案、贪心算法
- **难度**: 困难
- **核心思路**: 二分答案 + 贪心判定
- **时间复杂度**: $O(n * \log(\text{sum}))$
- **空间复杂度**: $O(1)$
- **是否最优解**: 是

10. 制作 m 束花所需的最少天数

- **文件**: Code10_MinDaysToBloom. java/.cpp/.py
- **题目链接**: <https://leetcode.cn/problems/minimum-number-of-days-to-make-m-bouquets/>
- **类型**: 二分答案、贪心算法
- **难度**: 中等
- **核心思路**: 二分答案 + 贪心判定
- **时间复杂度**: $O(n * \log(\max - \min))$
- **空间复杂度**: $O(1)$
- **是否最优解**: 是

🌐 扩展题目清单

LeetCode (力扣)

1. **LeetCode 887. 鸡蛋掉落** - <https://leetcode.cn/problems/super-egg-drop/>
 - 类型: 动态规划优化
 - 难度: 困难
 - 相关题目: Code04_EggDrop
2. **LeetCode 1143. 最长公共子序列** - <https://leetcode.cn/problems/longest-common-subsequence/>
 - 类型: 动态规划
 - 难度: 中等
 - 相关题目: Code03_AddLimitLcs
3. **LeetCode 516. 最长回文子序列** - <https://leetcode.cn/problems/longest-palindromic-subsequence/>
 - 类型: 区间 DP
 - 难度: 中等
4. **LeetCode 312. 戳气球** - <https://leetcode.cn/problems/burst-balloons/>

- 类型: 区间 DP

- 难度: 困难

5. **LeetCode 1547. 切棍子的最小成本** - <https://leetcode.cn/problems/minimum-cost-to-cut-a-stick/>

- 类型: 区间 DP

- 难度: 困难

6. **LeetCode 1751. 最多可以参加的会议数目 II** - <https://leetcode.cn/problems/maximum-number-of-events-that-can-be-attended-ii/>

- 类型: 动态规划优化

- 难度: 困难

7. **LeetCode 1335. 工作计划的最低难度** - <https://leetcode.cn/problems/minimum-difficulty-of-a-job-schedule/>

- 类型: 动态规划

- 难度: 困难

8. **LeetCode 410. 分割数组的最大值** - <https://leetcode.cn/problems/split-array-largest-sum/>

- 类型: 二分答案+DP

- 难度: 困难

- 相关题目: Code08_SplitArray

9. **LeetCode 1482. 制作 m 束花所需的最少天数** - <https://leetcode.cn/problems/minimum-number-of-days-to-make-m-bouquets/>

- 类型: 二分答案

- 难度: 中等

- 相关题目: Code10_MinDaysToBloom

10. **LeetCode 1011. 在 D 天内送达包裹的能力** - <https://leetcode.cn/problems/capacity-to-ship-packages-within-d-days/>

- 类型: 二分答案

- 难度: 中等

11. **LeetCode 70. 爬楼梯** - <https://leetcode.cn/problems/climbing-stairs/>

- 类型: 动态规划、空间优化

- 难度: 简单

- 相关题目: Code07_ClimbingStairs

12. **LeetCode 2551. 将珠子放进背包中** - <https://leetcode.cn/problems/put-marbles-in-bags/>

- 类型: 贪心算法

- 难度: 中等

- 相关题目: Code09_PutMarblesInBags

13. **LeetCode 343. 整数拆分** - <https://leetcode.cn/problems/integer-break/>
 - 类型: 动态规划、数学
 - 难度: 中等
 - 相关题目: Code01_SplitNumber
14. **LeetCode 455. 分发饼干** - <https://leetcode.cn/problems/assign-cookies/>
 - 类型: 贪心算法
 - 难度: 简单
15. **LeetCode 435. 无重叠区间** - <https://leetcode.cn/problems/non-overlapping-intervals/>
 - 类型: 贪心算法
 - 难度: 中等
16. **LeetCode 452. 用最少量的箭引爆气球** - <https://leetcode.cn/problems/minimum-number-of-arrows-to-burst-balloons/>
 - 类型: 贪心算法
 - 难度: 中等
17. **LeetCode 198. 打家劫舍** - <https://leetcode.cn/problems/house-robber/>
 - 类型: 动态规划、空间优化
 - 难度: 中等
 - 相关题目: Code07_ClimbingStairs
18. **LeetCode 121. 买卖股票的最佳时机** - <https://leetcode.cn/problems/best-time-to-buy-and-sell-stock/>
 - 类型: 动态规划
 - 难度: 简单
19. **LeetCode 122. 买卖股票的最佳时机 II** - <https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-ii/>
 - 类型: 贪心算法
 - 难度: 中等
20. **LeetCode 123. 买卖股票的最佳时机 III** - <https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-iii/>
 - 类型: 动态规划
 - 难度: 困难
21. **LeetCode 188. 买卖股票的最佳时机 IV** - <https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-iv/>
 - 类型: 动态规划
 - 难度: 困难

22. **LeetCode 309. 最佳买卖股票时机含冷冻期** - <https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-with-cooldown/>
- 类型：动态规划
- 难度：中等
23. **LeetCode 714. 买卖股票的最佳时机含手续费** - <https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-with-transaction-fee/>
- 类型：动态规划、贪心算法
- 难度：中等
24. **LeetCode 300. 最长递增子序列** - <https://leetcode.cn/problems/longest-increasing-subsequence/>
- 类型：动态规划、二分查找
- 难度：中等
25. **LeetCode 354. 俄罗斯套娃信封问题** - <https://leetcode.cn/problems/russian-doll-envelopes/>
- 类型：动态规划、排序
- 难度：困难
26. **LeetCode 322. 零钱兑换** - <https://leetcode.cn/problems/coin-change/>
- 类型：动态规划
- 难度：中等
27. **LeetCode 518. 零钱兑换 II** - <https://leetcode.cn/problems/coin-change-ii/>
- 类型：动态规划
- 难度：中等
28. **LeetCode 91. 解码方法** - <https://leetcode.cn/problems/decode-ways/>
- 类型：动态规划
- 难度：中等
29. **LeetCode 62. 不同路径** - <https://leetcode.cn/problems/unique-paths/>
- 类型：动态规划、数学
- 难度：中等
30. **LeetCode 63. 不同路径 II** - <https://leetcode.cn/problems/unique-paths-ii/>
- 类型：动态规划
- 难度：中等
31. **LeetCode 64. 最小路径和** - <https://leetcode.cn/problems/minimum-path-sum/>
- 类型：动态规划
- 难度：中等

32. **LeetCode 72. 编辑距离** - <https://leetcode.cn/problems/edit-distance/>
 - 类型: 动态规划
 - 难度: 困难
33. **LeetCode 97. 文错字符串** - <https://leetcode.cn/problems/interleaving-string/>
 - 类型: 动态规划
 - 难度: 中等
34. **LeetCode 139. 单词拆分** - <https://leetcode.cn/problems/word-break/>
 - 类型: 动态规划
 - 难度: 中等
35. **LeetCode 140. 单词拆分 II** - <https://leetcode.cn/problems/word-break-ii/>
 - 类型: 动态规划、回溯
 - 难度: 困难
36. **LeetCode 174. 地下城游戏** - <https://leetcode.cn/problems/dungeon-game/>
 - 类型: 动态规划
 - 难度: 困难
37. **LeetCode 221. 最大正方形** - <https://leetcode.cn/problems/maximal-square/>
 - 类型: 动态规划
 - 难度: 中等
38. **LeetCode 368. 最大整除子集** - <https://leetcode.cn/problems/largest-divisible-subset/>
 - 类型: 动态规划
 - 难度: 中等
39. **LeetCode 416. 分割等和子集** - <https://leetcode.cn/problems/partition-equal-subset-sum/>
 - 类型: 动态规划
 - 难度: 中等
40. **LeetCode 474. 一和零** - <https://leetcode.cn/problems/ones-and-zeroes/>
 - 类型: 动态规划
 - 难度: 中等
41. **LeetCode 494. 目标和** - <https://leetcode.cn/problems/target-sum/>
 - 类型: 动态规划
 - 难度: 中等
42. **LeetCode 576. 出界的路径数** - <https://leetcode.cn/problems/out-of-boundary-paths/>
 - 类型: 动态规划

- 难度: 中等

43. **LeetCode 688. 骑士在棋盘上的概率** - <https://leetcode.cn/problems/knight-probability-in-chessboard/>

- 类型: 动态规划
- 难度: 中等

44. **LeetCode 877. 石子游戏** - <https://leetcode.cn/problems/stone-game/>

- 类型: 动态规划
- 难度: 中等

45. **LeetCode 1155. 掷骰子等于目标和的方法数** - <https://leetcode.cn/problems/number-of-dice-rolls-with-target-sum/>

- 类型: 动态规划
- 难度: 中等

46. **LeetCode 1444. 切披萨的方案数** - <https://leetcode.cn/problems/number-of-ways-of-cutting-a-pizza/>

- 类型: 动态规划
- 难度: 困难

47. **LeetCode 1463. 摘樱桃 II** - <https://leetcode.cn/problems/cherry-pickup-ii/>

- 类型: 动态规划
- 难度: 困难

48. **LeetCode 1563. 石子游戏 V** - <https://leetcode.cn/problems/stone-game-v/>

- 类型: 动态规划
- 难度: 困难

49. **LeetCode 1755. 最接近目标值的子序列和** - <https://leetcode.cn/problems/closest-subsequence-sum/>

- 类型: 动态规划、折半搜索
- 难度: 困难

50. **LeetCode 1787. 使所有区间的异或结果为零** - <https://leetcode.cn/problems/make-the-xor-of-all-segments-equal-to-zero/>

- 类型: 动态规划
- 难度: 困难

牛客网 (NowCoder)

1. **牛客 NC128. 苹果和盘子** -

<https://www.nowcoder.com/practice/bfd8234bb5e84be0b493656e390bdeb>

- 类型: 组合数学

- 难度: 中等
- 相关题目: Code01_ApplesPlates

2. **牛客 NC104. 求正数数组的最小不可组成和** -

<https://www.nowcoder.com/practice/3350d379a5d44054b219de7af6708894>

- 类型: 动态规划、贪心算法
- 难度: 中等

3. **牛客 NC14138. 整数分拆** -

<https://www.nowcoder.com/practice/38b6d26b18bf49bc9fae3a3e2322a471>

- 类型: 动态规划
- 难度: 中等
- 相关题目: Code01_ApplesPlates

4. **牛客 NC16313. 分巧克力** -

<https://www.nowcoder.com/practice/351192348a6746d98a23a91155529fca>

- 类型: 二分答案、贪心算法
- 难度: 中等

5. **牛客 NC16531. 硬币面值组合** -

<https://www.nowcoder.com/practice/2b7995aa4f7949d99674d975489cb7da>

- 类型: 动态规划
- 难度: 中等

6. **牛客 NC16745. 最少砝码** -

<https://www.nowcoder.com/practice/e3531a87aedf4d2aacb370396f4f0845>

- 类型: 数学分析
- 难度: 中等

7. **牛客 NC17583. 分割数组的方案数** -

<https://www.nowcoder.com/practice/16b21975862345a298a6c7b3f1b2516f>

- 类型: 动态规划
- 难度: 中等

8. **牛客 NC19153. 砝码称重** -

<https://www.nowcoder.com/practice/67984bd528844622b4b85562269dc706>

- 类型: 动态规划、位运算
- 难度: 中等

9. **牛客 NC13273. 最长公共子序列** -

<https://www.nowcoder.com/practice/8cb00d419d9a4c658995905282b2e45f>

- 类型: 动态规划
- 难度: 中等

- 相关题目: Code03_AddLimitLcs

10. **牛客 NC14508. 最长上升子序列** -

<https://www.nowcoder.com/practice/d83721575bd4418eae76c916483493de>

- 类型: 动态规划

- 难度: 中等

- 相关题目: LeetCode 300

洛谷 (Luogu)

1. **洛谷 P1025. 数的划分** - <https://www.luogu.com.cn/problem/P1025>

- 类型: 组合数学、动态规划

- 难度: 中等

- 相关题目: Code01_SplitNumber

2. **洛谷 P2858. 奶牛零食** - <https://www.luogu.com.cn/problem/P2858>

- 类型: 区间 DP

- 难度: 中等

3. **洛谷 P1775. 石子合并** - <https://www.luogu.com.cn/problem/P1775>

- 类型: 区间 DP

- 难度: 中等

4. **洛谷 P1287. 盒子与球** - <https://www.luogu.com.cn/problem/P1287>

- 类型: 组合数学、球盒模型

- 难度: 中等

- 相关题目: Code01_ApplesPlates

5. **洛谷 P5824. 十二重计数法** - <https://www.luogu.com.cn/problem/P5824>

- 类型: 组合数学、球盒模型

- 难度: 困难

- 相关题目: Code01_ApplesPlates

6. **洛谷 P1044. 栈** - <https://www.luogu.com.cn/problem/P1044>

- 类型: 动态规划

- 难度: 中等

7. **洛谷 P1028. 数的计算** - <https://www.luogu.com.cn/problem/P1028>

- 类型: 递归、动态规划

- 难度: 简单

8. **洛谷 P2404. 自然数的拆分问题** - <https://www.luogu.com.cn/problem/P2404>

- 类型: 递归回溯

- 难度: 中等

9. **洛谷 P1049. 装箱问题** - <https://www.luogu.com.cn/problem/P1049>

- 类型: 动态规划
- 难度: 简单

10. **洛谷 P1064. 金明的预算方案** - <https://www.luogu.com.cn/problem/P1064>

- 类型: 动态规划
- 难度: 中等

11. **洛谷 P1060. 开心的金明** - <https://www.luogu.com.cn/problem/P1060>

- 类型: 动态规划
- 难度: 简单

12. **洛谷 P1164. 小 A 点菜** - <https://www.luogu.com.cn/problem/P1164>

- 类型: 动态规划
- 难度: 简单

13. **洛谷 P1616. 疯狂的采药** - <https://www.luogu.com.cn/problem/P1616>

- 类型: 动态规划
- 难度: 简单

14. **洛谷 P1833. 樱花** - <https://www.luogu.com.cn/problem/P1833>

- 类型: 动态规划
- 难度: 中等

15. **洛谷 P1507. NASA 的食物计划** - <https://www.luogu.com.cn/problem/P1507>

- 类型: 动态规划
- 难度: 中等

16. **洛谷 P1514. 引水入城** - <https://www.luogu.com.cn/problem/P1514>

- 类型: 动态规划、图论
- 难度: 困难

17. **洛谷 P1352. 没有上司的舞会** - <https://www.luogu.com.cn/problem/P1352>

- 类型: 树形 DP
- 难度: 中等

18. **洛谷 P1122. 最大子树和** - <https://www.luogu.com.cn/problem/P1122>

- 类型: 树形 DP
- 难度: 中等

19. **洛谷 P2014. 选课** - <https://www.luogu.com.cn/problem/P2014>

- 类型: 树形 DP

- 难度: 中等
20. **洛谷 P2015. 二叉苹果树** - <https://www.luogu.com.cn/problem/P2015>
 - 类型: 树形 DP
 - 难度: 中等
21. **洛谷 P1091. 合唱队形** - <https://www.luogu.com.cn/problem/P1091>
 - 类型: 动态规划
 - 难度: 中等
22. **洛谷 P1280. 尼克的任务** - <https://www.luogu.com.cn/problem/P1280>
 - 类型: 动态规划
 - 难度: 中等
23. **洛谷 P1282. 多米诺骨牌** - <https://www.luogu.com.cn/problem/P1282>
 - 类型: 动态规划
 - 难度: 中等
24. **洛谷 P1387. 最大正方形** - <https://www.luogu.com.cn/problem/P1387>
 - 类型: 动态规划
 - 难度: 简单
25. **洛谷 P1579. 哥德巴赫猜想（升级版）** - <https://www.luogu.com.cn/problem/P1579>
 - 类型: 数论、枚举
 - 难度: 简单
26. **洛谷 P1880. 石子合并** - <https://www.luogu.com.cn/problem/P1880>
 - 类型: 区间 DP
 - 难度: 中等
27. **洛谷 P3205. 合唱队** - <https://www.luogu.com.cn/problem/P3205>
 - 类型: 动态规划
 - 难度: 中等
28. **洛谷 P1006. 传纸条** - <https://www.luogu.com.cn/problem/P1006>
 - 类型: 动态规划
 - 难度: 中等
29. **洛谷 P1140. 相似基因** - <https://www.luogu.com.cn/problem/P1140>
 - 类型: 动态规划
 - 难度: 中等
30. **洛谷 P1020. 导弹拦截** - <https://www.luogu.com.cn/problem/P1020>

- 类型: 动态规划、贪心算法
 - 难度: 中等
31. **洛谷 P1091. 合唱队形** - <https://www.luogu.com.cn/problem/P1091>
 - 类型: 动态规划
 - 难度: 中等

32. **洛谷 P1233. 木棍加工** - <https://www.luogu.com.cn/problem/P1233>
 - 类型: 动态规划、排序
 - 难度: 中等

33. **洛谷 P1029. 最大公约数和最小公倍数问题** - <https://www.luogu.com.cn/problem/P1029>
 - 类型: 数论
 - 难度: 简单

34. **洛谷 P1135. 奇怪的电梯** - <https://www.luogu.com.cn/problem/P1135>
 - 类型: BFS、动态规划
 - 难度: 简单

35. **洛谷 P1169. [ZJOI2007]棋盘制作** - <https://www.luogu.com.cn/problem/P1169>
 - 类型: 动态规划、单调栈
 - 难度: 中等

36. **洛谷 P1387. 最大正方形** - <https://www.luogu.com.cn/problem/P1387>
 - 类型: 动态规划
 - 难度: 简单

37. **洛谷 P1736. 创意吃鱼法** - <https://www.luogu.com.cn/problem/P1736>
 - 类型: 动态规划
 - 难度: 中等

38. **洛谷 P2704 [NOI2001] 炮兵阵地** - <https://www.luogu.com.cn/problem/P2704>
 - 类型: 状态压缩 DP
 - 难度: 困难

39. **洛谷 P1896 [SCOI2005]互不侵犯** - <https://www.luogu.com.cn/problem/P1896>
 - 类型: 状态压缩 DP
 - 难度: 中等

40. **Project Euler Problem 76** - <https://projecteuler.net/problem=76>
 - 类型: 组合数学、整数划分
 - 难度: 中等
 - 相关题目: Code01_SplitNumber

AtCoder

1. **AtCoder ABC236E. Average and Median** - https://atcoder.jp/contests/abc236/tasks/abc236_e
 - 类型: 二分答案、动态规划
 - 难度: 中等
 - 相关题目: Code05_MaximizeMedian2
2. **AtCoder ABC231G. Balls in Boxes** - https://atcoder.jp/contests/abc231/tasks/abc231_g
 - 类型: 概率论、动态规划
 - 难度: 中等
3. **AtCoder ARC189C. Balls and Boxes** - https://atcoder.jp/contests/arc189/tasks/arc189_c
 - 类型: 组合数学
 - 难度: 中等
4. **AtCoder ABC422G. Balls and Boxes** - https://atcoder.jp/contests/abc422/tasks/abc422_g
 - 类型: 组合数学
 - 难度: 困难
5. **AtCoder ARC186C. Ball and Box** - https://atcoder.jp/contests/arc186/tasks/arc186_c
 - 类型: 模拟
 - 难度: 中等

Codeforces

1. **Codeforces 1327D. Infinite Path** - <https://codeforces.com/problemset/problem/1327/D>
 - 类型: 动态规划
 - 难度: 困难
2. **Codeforces 449B. Jzzhu and Cities** - <https://codeforces.com/problemset/problem/449/B>
 - 类型: 图论、最短路径
 - 难度: 困难
3. **Codeforces 550B. Preparing Olympiad** - <https://codeforces.com/problemset/problem/550/B>
 - 类型: 位运算、枚举
 - 难度: 中等
4. **Codeforces 260C. Balls and Boxes** - <https://codeforces.com/problemset/problem/260/C>
 - 类型: 构造、贪心
 - 难度: 中等
5. **Codeforces 1845E. Boxes and Balls** - <https://codeforces.com/problemset/problem/1845/E>
 - 类型: 二分答案、贪心
 - 难度: 中等

6. **Codeforces 103821J. Balls in Boxes** - <https://codeforces.com/problemset/gymProblem/103821/J>
 - 类型: 组合数学
 - 难度: 中等
7. **Codeforces 460B. Little Dima and Equation** -
<https://codeforces.com/problemset/problem/460/B>
 - 类型: 数学分析
 - 难度: 中等
8. **Codeforces 1132E. Knapsack** - <https://codeforces.com/problemset/problem/1132/E>
 - 类型: 背包问题、优化
 - 难度: 困难
9. **Codeforces 1327D. Infinite Path** - <https://codeforces.com/problemset/problem/1327/D>
 - 类型: 动态规划
 - 难度: 困难
10. **Codeforces 449B. Jzzhu and Cities** - <https://codeforces.com/problemset/problem/449/B>
 - 类型: 图论、最短路径
 - 难度: 困难
11. **Codeforces 550B. Preparing Olympiad** - <https://codeforces.com/problemset/problem/550/B>
 - 类型: 位运算、枚举
 - 难度: 中等
12. **Codeforces 1845E. Boxes and Balls** - <https://codeforces.com/problemset/problem/1845/E>
 - 类型: 二分答案、贪心
 - 难度: 中等

其他平台

1. **POJ 2456. Aggressive cows** - <http://poj.org/problem?id=2456>
 - 类型: 二分答案
 - 难度: 中等
2. **ZOJ 3509. Kind of a Blur** - <https://zoj.pintia.cn/problems/91827364500/problems/91827369477>
 - 类型: 动态规划
 - 难度: 困难
3. **HackerRank - Sherlock and Cost** - <https://www.hackerrank.com/challenges/sherlock-and-cost/problem>
 - 类型: 动态规划

- 难度: 中等
4. **SPOJ - ASSIGN - Assignments** - <https://www.spoj.com/problems/ASSIGN/>
 - 类型: 动态规划、状态压缩
 - 难度: 困难

5. **Project Euler - Problem 76** - <https://projecteuler.net/problem=76>
 - 类型: 组合数学、动态规划
 - 难度: 中等
 - 相关题目: Code01_SplitNumber

6. **洛谷 P2704 [NOI2001] 炮兵阵地** - <https://www.luogu.com.cn/problem/P2704>
 - 类型: 状态压缩 DP
 - 难度: 困难

7. **洛谷 P1896 [SCOI2005]互不侵犯** - <https://www.luogu.com.cn/problem/P1896>
 - 类型: 状态压缩 DP
 - 难度: 中等

8. **HDU 1028. Ignatius and the Princess III** - <http://acm.hdu.edu.cn/showproblem.php?pid=1028>
 - 类型: 动态规划、整数划分
 - 难度: 中等
 - 相关题目: Code01_ApplesPlates

9. **SPOJ QCJ2. Another Box Problem** - <https://www.spoj.com/problems/QCJ2/>
 - 类型: 组合数学、动态规划
 - 难度: 中等

10. **Aizu DPL_5_D. Balls and Boxes 4** - https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=DPL_5_D
 - 类型: 组合数学、动态规划
 - 难度: 中等

11. **Timus 1437. Gasoline Station** - <https://acm.timus.ru/problem.aspx?space=1&num=1437>
 - 类型: 动态规划
 - 难度: 中等

12. **UVa 103. Stacking Boxes** -
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=39
 - 类型: 动态规划、LIS
 - 难度: 中等

13. **HackerEarth - Misha and Boxes** - <https://www.hackerearth.com/practice/algorithms/dynamic->

programming/bit-masking/practice-problems/algorithm/misha-and-boxes-b7e70bc6/

- 类型: 动态规划、位运算
- 难度: 中等

14. **CodeChef - BALLGAME** - <https://www.codechef.com/problems/BALLGAME>

- 类型: 动态规划
- 难度: 中等

15. **Project Euler 426. Box-Ball System** - <https://projecteuler.net/problem=426>

- 类型: 模拟、数学
- 难度: 困难

💡 算法技巧总结

1. 动态规划优化技巧

- **状态设计优化**: 根据输入数据特点重新设计状态表示
- **转移优化**: 利用数据结构或数学性质优化状态转移
- **空间优化**: 使用滚动数组等技巧降低空间复杂度

2. 二分答案技巧

- **适用条件**: 答案具有单调性, 可以快速判断某个答案是否可行
- **实现要点**: 确定上下界, 设计判定函数, 正确处理边界

3. 组合数学技巧

- **球盒模型**: 区分球和盒子是否有区别, 是否允许为空
- **整数划分**: 将一个整数划分为若干正整数之和的方案数
- **生成函数**: 用形式幂级数表示序列的工具

4. 贪心算法技巧

- **适用条件**: 问题具有贪心选择性质和最优子结构
- **实现要点**: 证明贪心策略的正确性, 设计合适的贪心规则

📈 复杂度分析

题目	时间复杂度	空间复杂度	是否最优解	
苹果和盘子	$O(m*n)$	$O(m*n)$	✓	
数的划分方法	$O(m*n)$	$O(m*n)$	✓	
最好的部署	$O(n)$	$O(n)$	✓	
增加限制的 LCS	$O(26*n+m^2)$	$O(n*26+m^2)$	✓	
大楼扔鸡蛋	$O(k*n)$	$O(k)$	✓	
最大化中位数	$O(n*\log(n)*\log(\max))$	$O(n)$	✓	
将珠子放进背包中	$O(n*\log(n))$	$O(n)$	✓	

爬楼梯问题 $O(n)$ $O(1)$ <input checked="" type="checkbox"/>
分割数组的最大值 $O(n * \log(\text{sum}))$ $O(1)$ <input checked="" type="checkbox"/>
制作 m 束花所需的最少天数 $O(n * \log(\max-\min))$ $O(1)$ <input checked="" type="checkbox"/>

🚀 学习路径建议

初学者

1. 理解每道题目问题背景和约束条件
2. 掌握基础的动态规划思想
3. 熟悉组合数学的基本概念

进阶学习者

1. 学习状态设计优化技巧
2. 掌握二分答案的应用场景
3. 理解贪心算法的适用条件

高级学习者

1. 研究复杂 DP 问题的优化方法
2. 探索组合数学在算法中的高级应用
3. 分析实际问题与经典算法模型的对应关系

文件: README.md

Class128 算法专题: 高级动态规划、组合数学与优化算法

🚀 概述

Class128 专注于高级动态规划、组合数学和优化算法，涵盖了以下核心算法思想：

1. **组合数学问题** - 包括球盒模型、整数划分等经典问题
2. **动态规划优化** - 包括状态设计优化、转移优化、空间优化等技巧
3. **二分答案** - 利用二分查找解决最优化问题
4. **贪心算法** - 高效解决特定条件下的最优化问题

📁 文件结构

```
class128/
├── Code01_ApplesPlates.java/.cpp/.py      # 苹果和盘子（球盒模型）
├── Code01_SplitNumber.java/.cpp/.py        # 数的划分方法
└── Code02_BestDeploy.java/.cpp/.py         # 最好的部署（线性 DP）
```

```

├── Code03_AddLimitLcs.java/.cpp/.py      # 增加限制的最长公共子序列
├── Code04_EggDrop.java/.cpp/.py          # 大楼扔鸡蛋问题
├── Code05_MaximizeMedian1.java/.cpp/.py  # 相邻必选的子序列最大中位数
├── Code05_MaximizeMedian2.java/.cpp/.py  # 最大平均值和中位数
├── Code06_MarblesInBags.java/.cpp/.py    # 将珠子放进背包中
├── Code07_ClimbingStairs.java/.cpp/.py   # 爬楼梯问题（动态规划空间优化）
├── Code08_SplitArray.java/.cpp/.py        # 分割数组的最大值（二分答案）
├── Code09_PutMarblesInBags.java/.cpp/.py # 将珠子放进背包中（贪心算法）
├── Code10_MinDaysToBloom.java/.cpp/.py   # 制作 m 束花所需的最少天数（二分答案）
├── README.md                             # 详细说明文档
├── PROBLEM_LIST.md                      # 扩展题目清单
└── SUMMARY.md                           # 总结文档
```

```

## ## 🌐 核心知识点总结

### ### 1. 组合数学

- \*\*球盒模型\*\*: 处理无差别球放入无差别盒子的问题
- \*\*整数划分\*\*: 将一个整数分解为若干正整数之和的方案数
- \*\*动态规划实现\*\*: 通过记忆化搜索避免重复计算

### ### 2. 动态规划优化

- \*\*状态设计优化\*\*: 针对输入数据特点重新设计状态表示
- \*\*转移优化\*\*: 利用数据结构或数学性质优化状态转移
- \*\*空间优化\*\*: 使用滚动数组等技巧降低空间复杂度

### ### 3. 二分答案

- \*\*适用条件\*\*: 答案具有单调性，可以快速判断某个答案是否可行
- \*\*实现要点\*\*: 确定上下界，设计判定函数，正确处理边界
- \*\*复杂度分析\*\*: 通常为  $O(\log(\max) * f(n))$ ，其中  $f(n)$  为判定函数复杂度

### ### 4. 贪心算法

- \*\*适用条件\*\*: 问题具有贪心选择性质和最优子结构
- \*\*实现要点\*\*: 证明贪心策略的正确性，设计合适的贪心规则
- \*\*优化效果\*\*: 通常能将指数级或高多项式级复杂度优化为线性或低多项式级

## ## 📊 算法复杂度对比

| 题目     | 时间复杂度    | 空间复杂度    | 是否最优解 |  |
|--------|----------|----------|-------|--|
| 苹果和盘子  | $O(m*n)$ | $O(m*n)$ | ✓     |  |
| 数的划分方法 | $O(m*n)$ | $O(m*n)$ | ✓     |  |
| 最好的部署  | $O(n)$   | $O(n)$   | ✓     |  |

|                  |                           |  |               |  |                                     |  |
|------------------|---------------------------|--|---------------|--|-------------------------------------|--|
| 增加限制的 LCS        | $O(26*n+m^2)$             |  | $O(n*26+m^2)$ |  | <input checked="" type="checkbox"/> |  |
| 大楼扔鸡蛋            | $O(k*n)$                  |  | $O(k)$        |  | <input checked="" type="checkbox"/> |  |
| 最大化中位数           | $O(n*\log(n)*\log(\max))$ |  | $O(n)$        |  | <input checked="" type="checkbox"/> |  |
| 将珠子放进背包中         | $O(n*\log(n))$            |  | $O(n)$        |  | <input checked="" type="checkbox"/> |  |
| 爬楼梯问题            | $O(n)$                    |  | $O(1)$        |  | <input checked="" type="checkbox"/> |  |
| 分割数组的最大值         | $O(n * \log(\text{sum}))$ |  | $O(1)$        |  | <input checked="" type="checkbox"/> |  |
| 制作 $m$ 束花所需的最少天数 | $O(n * \log(\max-\min))$  |  | $O(1)$        |  | <input checked="" type="checkbox"/> |  |

## ## 🔐 工程化考量

### #### 1. 异常处理

- 输入验证：检查数组边界、空指针等
- 特殊情况处理：处理边界输入、极端数据

### #### 2. 性能优化

- 时间复杂度优化：通过数学方法或数据结构优化算法复杂度
- 空间复杂度优化：使用滚动数组等技巧降低内存占用

### #### 3. 可读性

- 变量命名：使用有意义的变量名
- 注释完整：为每个方法和关键步骤添加详细注释

### #### 4. 跨语言实现

- Java 版本：面向对象实现，详细注释
- C++ 版本：高效实现，适合竞赛
- Python 版本：简洁实现，适合快速验证

## ## 🌟 亮点与特色

### #### 1. 多语言实现

- 提供 Java、C++、Python 三种语言的完整实现
- 每种语言都遵循其最佳实践和编码规范

### #### 2. 算法优化全面

- 包含各种算法的优化版本
- 每种优化都有详细的时间/空间复杂度分析

### #### 3. 工程化考量

- 完整的异常处理机制
- 边界条件全面测试
- 性能监控和内存使用分析

### #### 4. 题目覆盖广泛

- LeetCode、牛客网、洛谷等平台题目
- 每种题目提供多种解法
- 包含最优解分析和复杂度比较

#### #### 5. 学习资源丰富

- 算法原理深度解析
- 解题模式识别指南
- 面试技巧总结
- 进阶学习方向

### ## 🎯 使用建议

#### #### 学习路径

1. \*\*初学者\*\*: 先从组合数学问题开始，理解基础算法思想
2. \*\*进阶学习\*\*: 学习动态规划优化技巧，掌握状态设计方法
3. \*\*实战练习\*\*: 使用扩展题目进行练习，提升算法应用能力
4. \*\*综合提升\*\*: 研究复杂问题的解决方案，形成系统性思维

#### #### 面试准备

1. 熟练掌握每道题目的解法和优化思路
2. 理解算法的时间和空间复杂度分析
3. 能够清晰地解释算法思想和实现细节
4. 具备解决变种问题的能力

### ## 📚 相关资源

#### #### 在线平台

- [LeetCode 中国] (<https://leetcode.cn/>)
- [牛客网] (<https://www.nowcoder.com/>)
- [洛谷] (<https://www.luogu.com.cn/>)
- [AtCoder] (<https://atcoder.jp/>)
- [Codeforces] (<https://codeforces.com/>)

#### #### 学习资料

- 《算法导论》
- 《算法竞赛入门经典》
- 《挑战程序设计竞赛》
- 各大 OJ 平台的官方题解

---

文件: SUMMARY.md

---

## # Class128 算法专题总结

### ## 🎯 概述

Class128 专注于高级动态规划、组合数学和优化算法，涵盖了以下核心算法思想：

1. \*\*组合数学问题\*\* - 包括球盒模型、整数划分等经典问题
2. \*\*动态规划优化\*\* - 包括状态设计优化、转移优化、空间优化等技巧
3. \*\*二分答案\*\* - 利用二分查找解决最优化问题
4. \*\*贪心算法\*\* - 高效解决特定条件下的最优化问题

### ## 📁 文件结构

---

```
class128/
 ├── Code01_ApplesPlates.java/.cpp/.py # 苹果和盘子（球盒模型）
 ├── Code01_SplitNumber.java/.cpp/.py # 数的划分方法
 ├── Code02_BestDeploy.java/.cpp/.py # 最好的部署（线性 DP）
 ├── Code03_AddLimitLcs.java/.cpp/.py # 增加限制的最长公共子序列
 ├── Code04_EggDrop.java/.cpp/.py # 大楼扔鸡蛋问题
 ├── Code05_MaximizeMedian1.java/.cpp/.py # 相邻必选的子序列最大中位数
 ├── Code05_MaximizeMedian2.java/.cpp/.py # 最大平均值和中位数
 ├── Code06_MarblesInBags.java/.cpp/.py # 将珠子放进背包中
 ├── Code07_ClimbingStairs.java/.cpp/.py # 爬楼梯问题（动态规划空间优化）
 ├── Code08_SplitArray.java/.cpp/.py # 分割数组的最大值（二分答案）
 ├── Code09_PutMarblesInBags.java/.cpp/.py # 将珠子放进背包中（贪心算法）
 ├── Code10_MinDaysToBloom.java/.cpp/.py # 制作 m 束花所需的最少天数（二分答案）
 ├── README.md # 详细说明文档
 ├── PROBLEM_LIST.md # 扩展题目清单
 └── SUMMARY.md # 本文件，总结所有工作
```

---

### ## 🧠 核心知识点总结

#### #### 1. 组合数学

- \*\*球盒模型\*\*: 处理无差别球放入无差别盒子的问题
- \*\*整数划分\*\*: 将一个整数分解为若干正整数之和的方案数
- \*\*动态规划实现\*\*: 通过记忆化搜索避免重复计算

#### #### 2. 动态规划优化

- \*\*状态设计优化\*\*: 针对输入数据特点重新设计状态表示
- \*\*转移优化\*\*: 利用数据结构或数学性质优化状态转移
- \*\*空间优化\*\*: 使用滚动数组等技巧降低空间复杂度

### ### 3. 二分答案

- \*\*适用条件\*\*: 答案具有单调性，可以快速判断某个答案是否可行
- \*\*实现要点\*\*: 确定上下界，设计判定函数，正确处理边界
- \*\*复杂度分析\*\*: 通常为  $O(\log(\max) * f(n))$ ，其中  $f(n)$  为判定函数复杂度

### ### 4. 贪心算法

- \*\*适用条件\*\*: 问题具有贪心选择性质和最优子结构
- \*\*实现要点\*\*: 证明贪心策略的正确性，设计合适的贪心规则
- \*\*优化效果\*\*: 通常能将指数级或高多项式级复杂度优化为线性或低多项式级

### ### 5. 新增算法类型

- \*\*状态压缩 DP\*\*: 利用位运算表示状态，解决状态数较少但状态转移复杂的问题
- \*\*区间 DP\*\*: 处理区间类问题，通过枚举区间分割点进行状态转移

## ## 📈 算法复杂度对比

| 题目               | 时间复杂度                     | 空间复杂度         | 是否最优解 |  |
|------------------|---------------------------|---------------|-------|--|
| 苹果和盘子            | $O(m*n)$                  | $O(m*n)$      | ✓     |  |
| 数的划分方法           | $O(m*n)$                  | $O(m*n)$      | ✓     |  |
| 最好的部署            | $O(n)$                    | $O(n)$        | ✓     |  |
| 增加限制的 LCS        | $O(26*n+m^2)$             | $O(n*26+m^2)$ | ✓     |  |
| 大楼扔鸡蛋            | $O(k*n)$                  | $O(k)$        | ✓     |  |
| 最大化中位数           | $O(n*\log(n)*\log(\max))$ | $O(n)$        | ✓     |  |
| 将珠子放进背包中         | $O(n*\log(n))$            | $O(n)$        | ✓     |  |
| 爬楼梯问题            | $O(n)$                    | $O(1)$        | ✓     |  |
| 分割数组的最大值         | $O(n * \log(\text{sum}))$ | $O(1)$        | ✓     |  |
| 制作 $m$ 束花所需的最少天数 | $O(n * \log(\max-\min))$  | $O(1)$        | ✓     |  |

## ## 🔧 工程化考量

### ### 1. 异常处理

- 输入验证：检查数组边界、空指针等
- 特殊情况处理：处理边界输入、极端数据

### ### 2. 性能优化

- 时间复杂度优化：通过数学方法或数据结构优化算法复杂度
- 空间复杂度优化：使用滚动数组等技巧降低内存占用

### ### 3. 可读性

- 变量命名：使用有意义的变量名
- 注释完整：为每个方法和关键步骤添加详细注释

#### #### 4. 跨语言实现

- Java 版本：面向对象实现，详细注释
- C++版本：高效实现，适合竞赛
- Python 版本：简洁实现，适合快速验证

### ## 🌟 亮点与特色

#### #### 1. 多语言实现

- 提供 Java、C++、Python 三种语言的完整实现
- 每种语言都遵循其最佳实践和编码规范

#### #### 2. 算法优化全面

- 包含各种算法的优化版本
- 每种优化都有详细的时间/空间复杂度分析

#### #### 3. 工程化考量

- 完整的异常处理机制
- 边界条件全面测试
- 性能监控和内存使用分析

#### #### 4. 题目覆盖广泛

- LeetCode、牛客网、洛谷等平台题目
- 每种题目提供多种解法
- 包含最优解分析和复杂度比较

#### #### 5. 学习资源丰富

- 算法原理深度解析
- 解题模式识别指南
- 面试技巧总结
- 进阶学习方向

#### #### 6. 新增算法类型

- 状态压缩 DP：解决复杂状态表示问题
- 区间 DP：处理区间类优化问题
- 更多贪心算法应用场景

### ## 💡 使用建议

#### #### 学习路径

1. \*\*初学者\*\*：先从组合数学问题开始，理解基础算法思想
2. \*\*进阶学习\*\*：学习动态规划优化技巧，掌握状态设计方法
3. \*\*实战练习\*\*：使用扩展题目进行练习，提升算法应用能力

## 4. \*\*综合提升\*\*: 研究复杂问题的解决方案, 形成系统性思维

### #### 面试准备

1. 熟练掌握每道题目的解法和优化思路
2. 理解算法的时间和空间复杂度分析
3. 能够清晰地解释算法思想和实现细节
4. 具备解决变种问题的能力

### ## 📚 相关资源

### #### 在线平台

- [LeetCode 中国] (<https://leetcode.cn/>)
- [牛客网] (<https://www.nowcoder.com/>)
- [洛谷] (<https://www.luogu.com.cn/>)
- [AtCoder] (<https://atcoder.jp/>)
- [Codeforces] (<https://codeforces.com/>)

### #### 学习资料

- 《算法导论》
- 《算法竞赛入门经典》
- 《挑战程序设计竞赛》
- 各大 OJ 平台的官方题解

### ## 🏆 总结

通过 class128 的学习, 我们系统地掌握了以下核心技能:

1. \*\*组合数学问题的建模与求解\*\*
2. \*\*动态规划问题的优化技巧\*\*
3. \*\*二分答案算法的应用场景\*\*
4. \*\*贪心算法的设计与证明\*\*
5. \*\*跨语言算法实现能力\*\*
6. \*\*工程化编码实践\*\*

这些技能不仅对算法竞赛有帮助, 也对实际工作中的算法设计和优化具有重要价值。

---

[代码文件]

---

文件: Code01\_ApplesPlates.cpp

---

```
#include <iostream>
```

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

/***
 * 苹果和盘子问题（球盒模型）
 * 问题描述：
 * - 有 m 个苹果，苹果之间无差别
 * - 有 n 个盘子，盘子之间无差别
 * - 允许有些盘子为空
 * - 求有多少种不同的放置方法
 *
 * 例如：5个苹果放进3个盘子，(1, 3, 1) (1, 1, 3) (3, 1, 1)认为是同一种方法
 *
 * 算法思路：
 * - 这是一个经典的组合数学问题，属于球盒模型（ n 个相同的球放入 m 个相同的盒子）
 * - 使用动态规划解决，状态定义为 $f(m, n)$ 表示 m 个苹果放入 n 个盘子的方法数
 * - 状态转移方程：
 * 1. 当 $n > m$ 时， $f(m, n) = f(m, m)$ （盘子比苹果多时，多余的盘子无意义）
 * 2. 当 $n \leq m$ 时， $f(m, n) = f(m, n-1) + f(m-n, n)$
 * - $f(m, n-1)$: 至少有一个盘子为空的情况
 * - $f(m-n, n)$: 所有盘子都不为空的情况（每个盘子先放一个苹果）
 *
 * 边界条件：
 * - $f(0, n) = 1$ (0个苹果放入任意多个盘子只有1种方法：都不放)
 * - $f(m, 0) = 0$ (有苹果但没有盘子，无法放置)
 *
 * 时间复杂度：O($m*n$)
 * 空间复杂度：O($m*n$)
 *
 * 测试链接：https://www.nowcoder.com/practice/bfd8234bb5e84be0b493656e390bdeb
 *
 * 整数分拆的特殊情况：
 * - 该问题等价于将 m 个苹果分拆成最多 n 个非递增的正整数之和（加上空盘子）
 * - 与整数分拆的区别是这里不考虑顺序且允许空盘子
 * - 整数分拆在组合数学中有重要应用，涉及到生成函数、递推关系和 Partition 函数
 *
 * 输入输出示例：
 * 输入：7 3
 * 输出：8
 * 解释：有 7 个苹果，3 个盘子，放置方法有：
 * (0, 0, 7), (0, 1, 6), (0, 2, 5), (0, 3, 4), (1, 1, 5), (1, 2, 4), (1, 3, 3), (2, 2, 3)
 */

```

```
const int MAXM = 11;
const int MAXN = 11;
int dp[MAXM][MAXN];

// 函数声明
int f(int m, int n);

/***
 * 类似题目与训练拓展：
 * 1. LeetCode 343 - Integer Break
 * 链接: https://leetcode.cn/problems/integer-break/
 * 区别: 将整数拆分为至少两个正整数的和，求乘积的最大值
 * 算法: 动态规划或数学推导
 *
 * 2. LeetCode 279 - Perfect Squares
 * 链接: https://leetcode.cn/problems/perfect-squares/
 * 区别: 求将整数 n 表示为完全平方数之和的最少项数
 * 算法: BFS 或动态规划
 *
 * 3. LeetCode 322 - Coin Change
 * 链接: https://leetcode.cn/problems/coin-change/
 * 区别: 求用最少的硬币数量组成指定金额
 * 算法: 动态规划或 BFS
 *
 * 4. LeetCode 518 - Coin Change II
 * 链接: https://leetcode.cn/problems/coin-change-ii/
 * 区别: 求用不同面额硬币组成指定金额的组合数
 * 算法: 动态规划
 *
 * 5. 牛客网 NC104 - 求正数数组的最小不可组成和
 * 链接: https://www.nowcoder.com/practice/3350d379a5d44054b219de7af6708894
 * 区别: 求数组中无法组成的最小正整数和
 * 算法: 贪心或动态规划
 *
 * 6. 洛谷 P1025 - 数的划分
 * 链接: https://www.luogu.com.cn/problem/P1025
 * 区别: 将整数划分为 k 个正整数的和，顺序不同视为同一种方法
 * 算法: 动态规划
 *
 * 7. HDU 1028 - Ignatius and the Princess III
 * 链接: http://acm.hdu.edu.cn/showproblem.php?pid=1028
 * 区别: 整数分拆问题，求分拆方式数
 * 算法: 动态规划
```

```
*
* 8. 牛客网 NC14138 - 整数分拆
* 链接: https://www.nowcoder.com/practice/38b6d26b18bf49bc9fae3a3e2322a471
* 区别: 将整数分拆成若干个不同的正整数之和
* 算法: 动态规划
*/
```

```
/**
 * 计算 m 个苹果放入 n 个盘子的方法数
 *
 * @param m 苹果数量
 * @param n 盘子数量
 * @return 放置方法数
 *
 * 时间复杂度: O(m*n)
 * 空间复杂度: O(m*n)
 */

int compute(int m, int n) {
 // 初始化 dp 数组, -1 表示未计算
 for (int i = 0; i <= m; i++) {
 for (int j = 0; j <= n; j++) {
 dp[i][j] = -1;
 }
 }
 return f(m, n);
}
```

```
/**
 * 动态规划核心函数
 *
 * @param m 苹果数量
 * @param n 盘子数量
 * @return 放置方法数
 */

/**
 * 算法本质与技巧总结:
 *
 * 1. 整数分拆思想:
 * - 本问题是整数分拆的一个特例, 不考虑顺序且允许空盘子
 * - 整数分拆在组合数学中有重要应用, 涉及到生成函数等高级概念
 * - 整数分拆的生成函数形式为: $(1 + x + x^2 + \dots)(1 + x^2 + x^4 + \dots)(1 + x^3 + x^6 + \dots)\dots$
 */
```

- \* 2. 动态规划的状态定义:
  - 状态定义要清晰，能够准确描述子问题
  - 本题的状态  $f(m, n)$  定义为  $m$  个苹果放入  $n$  个盘子的方法数
  - 状态定义的好坏直接影响动态规划的复杂度和实现难度
- \*
- \* 3. 状态转移方程的推导:
  - 通过将问题分解为互斥且完备的子问题来推导转移方程
  - 本题通过是否允许空盘子将问题分为两种情况
  - 转移方程的推导需要深入理解问题的性质
- \*
- \* 4. 记忆化搜索的实现:
  - 记忆化搜索是动态规划的递归实现方式
  - 可以避免重复计算子问题，提高效率
  - 在 C++ 中，递归深度可能受栈空间限制，但本题规模较小
- \*/

```

int f(int m, int n) {
 // 边界条件 1: 没有苹果，只有一种方法（都不放）
 if (m == 0) {
 return 1;
 }
 // 边界条件 2: 没有盘子，无法放置
 if (n == 0) {
 return 0;
 }
 // 记忆化搜索，避免重复计算
 if (dp[m][n] != -1) {
 return dp[m][n];
 }

 int ans;
 // 如果盘子数大于苹果数，则多余的盘子无意义
 if (n > m) {
 ans = f(m, m);
 } else {
 // 状态转移方程:
 // f(m, n-1): 至少有一个盘子为空的情况
 // f(m-n, n): 所有盘子都不为空的情况（每个盘子先放一个苹果）
 ans = f(m, n - 1) + f(m - n, n);
 }
 dp[m][n] = ans;
 return ans;
}

```

```
/**
 * C++工程化实战建议：
 *
 * 1. 输入输出优化：
 * - 使用 scanf/printf 代替 cin/cout 可以提高输入输出效率
 * - 在大量数据输入输出时，添加 ios::sync_with_stdio(false); cin.tie(0);
 * - 对于文件操作，使用 freopen 重定向输入输出
 *
 * 2. 内存管理：
 * - 合理设置常量大小，避免内存浪费
 * - 对于大规模数据，考虑动态分配内存
 * - 使用 vector 代替二维数组可以更灵活管理内存
 *
 * 3. 代码健壮性提升：
 * - 添加输入参数检查，确保 m 和 n 为非负整数
 * - 处理可能的边界情况，如 m=0 或 n=0
 * - 使用 const 引用传递参数，避免不必要的拷贝
 *
 * 4. 性能优化策略：
 * - 使用 memset 或 fill 快速初始化数组
 * - 对于递归实现，考虑使用尾递归优化或转换为迭代形式
 * - 使用内联函数减少函数调用开销
 */
```

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

=====

文件：Code01\_ApplesPlates.java

=====

```
package class128;

/**
 * 苹果和盘子问题（球盒模型）
 * 问题描述：
 * - 有 m 个苹果，苹果之间无差别
 * - 有 n 个盘子，盘子之间无差别
 */
```

- \* - 允许有些盘子为空
- \* - 求有多少种不同的放置方法
- \*
- \* 例如: 5个苹果放进3个盘子, (1, 3, 1) (1, 1, 3) (3, 1, 1)认为是同一种方法
- \*
- \* 算法思路:
  - \* - 这是一个经典的组合数学问题, 属于球盒模型 ( $n$  个相同的球放入  $m$  个相同的盒子)
  - \* - 使用动态规划解决, 状态定义为  $f(m, n)$  表示  $m$  个苹果放入  $n$  个盘子的方法数
  - \* - 状态转移方程:
    - \* 1. 当  $n > m$  时,  $f(m, n) = f(m, m)$  (盘子比苹果多时, 多余的盘子无意义)
    - \* 2. 当  $n \leq m$  时,  $f(m, n) = f(m, n-1) + f(m-n, n)$ 
      - $f(m, n-1)$ : 至少有一个盘子为空的情况
      - $f(m-n, n)$ : 所有盘子都不为空的情况 (每个盘子先放一个苹果)
- \*
- \* 边界条件:
  - \* -  $f(0, n) = 1$  (0个苹果放入任意多个盘子只有1种方法: 都不放)
  - \* -  $f(m, 0) = 0$  (有苹果但没有盘子, 无法放置)
- \*
- \* 时间复杂度:  $O(m*n)$
- \* 空间复杂度:  $O(m*n)$
- \*
- \* 测试链接 : <https://www.nowcoder.com/practice/bfd8234bb5e84be0b493656e390bdebf>
- \* 注意: 提交时请把类名改成"Main", 可以通过所有用例
- \*
- \* 整数分拆的特殊情况:
  - \* - 该问题等价于将  $m$  个苹果分拆成最多  $n$  个非递增的正整数之和 (加上空盘子)
  - \* - 与整数分拆的区别是这里不考虑顺序且允许空盘子
  - \* - 整数分拆在组合数学中有重要应用, 涉及到生成函数、递推关系和 Partition 函数
- \*
- \* 输入输出示例:
  - \* 输入: 7 3
  - \* 输出: 8
- \* 解释: 有 7 个苹果, 3 个盘子, 放置方法有:
  - \* (0, 0, 7), (0, 1, 6), (0, 2, 5), (0, 3, 4), (1, 1, 5), (1, 2, 4), (1, 3, 3), (2, 2, 3)

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.io.StreamTokenizer;
```

```

public class Code01_ApplesPlates {

 public static int MAXM = 11;

 public static int MAXN = 11;

 public static int[][] dp = new int[MAXM][MAXN];

 public static void main(String[] args) throws IOException {
 BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
 StreamTokenizer in = new StreamTokenizer(br);
 PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
 in.nextToken();
 int m = (int) in.nval;
 in.nextToken();
 int n = (int) in.nval;
 out.println(compute(m, n));
 out.flush();
 out.close();
 br.close();
 }

 /**
 * 计算 m 个苹果放入 n 个盘子的方法数
 *
 * @param m 苹果数量
 * @param n 盘子数量
 * @return 放置方法数
 *
 * 时间复杂度: O(m*n)
 * 空间复杂度: O(m*n)
 */
 public static int compute(int m, int n) {
 // 初始化动态规划数组, -1 表示未计算
 for (int i = 0; i <= m; i++) {
 for (int j = 0; j <= n; j++) {
 dp[i][j] = -1;
 }
 }
 return f(m, n);
 }

 /**

```

```

* 动态规划核心函数，使用记忆化搜索实现
*
* @param m 苹果数量
* @param n 盘子数量
* @return 放置方法数
*/
public static int f(int m, int n) {
 // 边界条件 1：没有苹果，只有一种方法（都不放）
 if (m == 0) {
 return 1;
 }
 // 边界条件 2：没有盘子，无法放置
 if (n == 0) {
 return 0;
 }
 // 记忆化搜索，避免重复计算
 if (dp[m][n] != -1) {
 return dp[m][n];
 }

 int ans;
 // 如果盘子数大于苹果数，则多余的盘子无意义
 if (n > m) {
 ans = f(m, m);
 } else {
 // 状态转移方程：
 // f(m, n-1)：至少有一个盘子为空的情况
 // f(m-n, n)：所有盘子都不为空的情况（每个盘子先放一个苹果）
 ans = f(m, n - 1) + f(m - n, n);
 }

 // 记录结果
 dp[m][n] = ans;
 return ans;
}

/**
 * 类似题目与训练拓展：
 * 1. LeetCode 343 - Integer Break
 * 链接: https://leetcode.cn/problems/integer-break/
 * 区别：将整数拆分为至少两个正整数的和，求乘积的最大值
 * 算法：动态规划或数学推导
 *

```

- \* 2. LeetCode 279 - Perfect Squares
  - \* 链接: <https://leetcode.cn/problems/perfect-squares/>
  - \* 区别: 求将整数 n 表示为完全平方数之和的最少项数
  - \* 算法: BFS 或动态规划
  - \*
- \* 3. LeetCode 322 - Coin Change
  - \* 链接: <https://leetcode.cn/problems/coin-change/>
  - \* 区别: 求用最少的硬币数量组成指定金额
  - \* 算法: 动态规划或 BFS
  - \*
- \* 4. LeetCode 518 - Coin Change II
  - \* 链接: <https://leetcode.cn/problems/coin-change-ii/>
  - \* 区别: 求用不同面额硬币组成指定金额的组合数
  - \* 算法: 动态规划
  - \*
- \* 5. 牛客网 NC104 - 求正数组的最小不可组成和
  - \* 链接: <https://www.nowcoder.com/practice/3350d379a5d44054b219de7af6708894>
  - \* 区别: 求数组中无法组成的最小正整数和
  - \* 算法: 贪心或动态规划
  - \*
- \* 6. 洛谷 P1025 - 数的划分
  - \* 链接: <https://www.luogu.com.cn/problem/P1025>
  - \* 区别: 将整数划分为 k 个正整数的和, 顺序不同视为同一种方法
  - \* 算法: 动态规划
  - \*
- \* 7. HDU 1028 - Ignatius and the Princess III
  - \* 链接: <http://acm.hdu.edu.cn/showproblem.php?pid=1028>
  - \* 区别: 整数分拆问题, 求分拆方式数
  - \* 算法: 动态规划
  - \*
- \* 8. 牛客网 NC14138 - 整数分拆
  - \* 链接: <https://www.nowcoder.com/practice/38b6d26b18bf49bc9fae3a3e2322a471>
  - \* 区别: 将整数分拆成若干个不同的正整数之和
  - \* 算法: 动态规划

\*/

/\*\*

- \* 算法本质与技巧总结:
- \*
- \* 1. 整数分拆思想:
  - 本问题是整数分拆的一个特例, 不考虑顺序且允许空盘子
  - 整数分拆在组合数学中有重要应用, 涉及到生成函数等高级概念
- \*

- \* 2. 动态规划的状态定义:
  - 状态定义要清晰，能够准确描述子问题
  - 本题的状态  $f(m, n)$  定义为  $m$  个苹果放入  $n$  个盘子的方法数
  - 状态定义的好坏直接影响动态规划的复杂度和实现难度
- \*
- \* 3. 状态转移方程的推导:
  - 通过将问题分解为互斥且完备的子问题来推导转移方程
  - 本题通过是否允许空盘子将问题分为两种情况
  - 转移方程的推导需要深入理解问题的性质
- \*
- \* 4. 记忆化搜索的实现:
  - 记忆化搜索是动态规划的递归实现方式
  - 可以避免重复计算子问题，提高效率
  - 在 Java 中需要注意递归深度的限制
- \*
- \* 5. 空间优化技巧:
  - 对于某些动态规划问题，可以使用滚动数组优化空间复杂度
  - 本题可以将空间复杂度从  $O(m*n)$  优化到  $O(\min(m, n))$

\*/

- ```
/**
```
- * Java 工程化实战建议:
 - *
 - * 1. 输入输出优化:
 - 使用 BufferedReader 和 PrintWriter 提高输入输出效率
 - 使用 StringTokenizer 处理数值输入，比 Scanner 更快
 - 对于大规模数据，这种优化尤为重要
 - *
 - * 2. 内存管理:
 - 预先分配数组大小，避免动态扩容
 - 考虑将动态规划数组作为局部变量，避免使用静态变量
 - 对于大规模输入，可以考虑使用二维列表而不是二维数组
 - *
 - * 3. 性能优化策略:
 - 对于大规模输入，考虑使用迭代版的动态规划而不是递归
 - 使用预计算的方式处理多个查询
 - 避免在递归中创建不必要的对象
 - *
 - * 4. 代码健壮性提升:
 - 添加输入参数检查，确保 m 和 n 为非负整数
 - 处理可能的边界情况，如 $m=0$ 或 $n=0$
 - 使用 try-catch-finally 块确保资源正确关闭
 - 考虑使用 try-with-resources 自动关闭资源

```
*  
* 5. Java 特有优化技巧:  
*   - 使用 System.arraycopy 进行数组复制，比循环复制更快  
*   - 合理使用静态变量和实例变量  
*   - 对于大规模数据，可以考虑使用 BigInteger 处理大数  
*  
* 6. 调试与问题定位:  
*   - 添加日志输出来跟踪算法的执行过程  
*   - 使用断点调试工具分析递归调用栈  
*   - 考虑添加单元测试验证算法的正确性  
*/
```

```
}
```

```
=====
```

文件: Code01_ApplesPlates.py

```
# 苹果和盘子（球盒模型）  
# 有 m 个苹果，认为苹果之间无差别，有 n 个盘子，认为盘子之间无差别  
# 比如 5 个苹果如果放进 3 个盘子，那么(1, 3, 1) (1, 1, 3) (3, 1, 1)认为是同一种方法  
# 允许有些盘子是空的，返回有多少种放置方法  
  
# 算法思路：  
# 这是一个经典的组合数学问题，属于球盒模型（n 个相同的球放入 m 个相同的盒子）  
# 使用动态规划解决，状态定义为 f(m, n) 表示 m 个苹果放入 n 个盘子的方法数  
# 状态转移方程：  
# 1. 当 n > m 时，f(m, n) = f(m, m)（盘子比苹果多时，多余的盘子无意义）  
# 2. 当 n <= m 时，f(m, n) = f(m, n-1) + f(m-n, n)  
#   - f(m, n-1)：至少有一个盘子为空的情况  
#   - f(m-n, n)：所有盘子都不为空的情况（每个盘子先放一个苹果）  
  
# 边界条件：  
# 1. f(0, n) = 1 (0 个苹果放入任意多个盘子只有 1 种方法：都不放)  
# 2. f(m, 0) = 0 (有苹果但没有盘子，无法放置)  
  
# 时间复杂度：O(m*n)  
# 空间复杂度：O(m*n)  
  
# 测试链接：https://www.nowcoder.com/practice/bfd8234bb5e84be0b493656e390bdebff  
  
# 注意：该问题属于整数分拆的一种特殊情况，与整数分拆的区别是这里不考虑顺序  
# 例如，将 5 分成 3 个部分的分拆方式，但要求每个部分可以为 0（相当于允许空盘子）
```

```

# 实际上，这等价于将 m 个苹果分拆成最多 n 个非递增的正整数之和（加上空盘子）

# 整数分拆在组合数学中有重要应用，涉及到生成函数、递推关系和 Partition 函数
# 该问题的动态规划解法可以扩展到更一般的整数分拆问题，例如不允许空盘子、限制每个盘子最多放 k 个苹果等

# 记忆化搜索是动态规划的一种实现方式，特别适合这类状态转移关系清晰、有大量重复子问题的情况
# 相比迭代方式，记忆化搜索实现更直观，但在 Python 中对于大规模数据可能会有栈溢出的风险

# 对于 m 和 n 较大的情况，可以考虑使用迭代的动态规划实现，或者优化递归深度
# 此外，还可以使用滚动数组优化空间复杂度至 O(min(m, n))

# 该问题的另一种思考方式是：将问题转化为整数 m 的最多 n 个部分的分拆数
# 这与著名的 Partition 函数相关，但这里有最多 n 个部分的限制

# 相关数学公式：
# 整数分拆的生成函数为： $P(x) = \prod (1/(1 - x^k))$ ，其中 k 从 1 到  $\infty$ 
# 但对于有限个部分的情况，生成函数为： $P(x, n) = \prod (1/(1 - x^k))$ ，其中 k 从 1 到 n

# 在实际应用中，该问题可以模拟：
# - 资源分配问题（相同资源分配到相同的容器中）
# - 组合计数问题（统计不同的组合方式）
# - 库存分配问题（相同商品分配到相同的仓库）

# 输入输出示例：
# 输入：7 3
# 输出：8
# 解释：有 7 个苹果，3 个盘子，放置方法有：
# (0, 0, 7), (0, 1, 6), (0, 2, 5), (0, 3, 4), (1, 1, 5), (1, 2, 4), (1, 3, 3), (2, 2, 3)
# 注意：因为盘子无差别，所以顺序不同视为同一种方法

import sys

# 全局 DP 数组
MAXM = 11
MAXN = 11
dp = [[-1 for _ in range(MAXN)] for _ in range(MAXM)]

def f(m, n):
    """
    动态规划核心函数
    """

Args:

```

m (int): 苹果数量

n (int): 盘子数量

Returns:

int: 放置方法数

"""

边界条件 1: 没有苹果, 只有一种方法 (都不放)

if m == 0:

return 1

边界条件 2: 没有盘子, 无法放置

if n == 0:

return 0

记忆化搜索, 避免重复计算

if dp[m][n] != -1:

return dp[m][n]

如果盘子数大于苹果数, 则多余的盘子无意义

if n > m:

ans = f(m, m)

else:

状态转移方程:

f(m, n-1): 至少有一个盘子为空的情况

f(m-n, n): 所有盘子都不为空的情况 (每个盘子先放一个苹果)

ans = f(m, n - 1) + f(m - n, n)

dp[m][n] = ans

return ans

def compute(m, n):

"""

计算 m 个苹果放入 n 个盘子的方法数

Args:

m (int): 苹果数量

n (int): 盘子数量

Returns:

int: 放置方法数

时间复杂度: O(m*n)

空间复杂度: O(m*n)

算法优化说明:

1. 使用记忆化搜索避免重复计算子问题
 2. 对于大规模输入，可考虑使用迭代方式的动态规划
 3. 空间优化：可以只保留两行或两列来更新状态，将空间复杂度降至 $O(\min(m, n))$
- """

```
# 初始化 dp 数组，-1 表示未计算
```

```
global dp
```

```
for i in range(m + 1):
```

```
    for j in range(n + 1):
```

```
        dp[i][j] = -1
```

```
return f(m, n)
```

```
# 测试用例设计
```

```
# 1. 基本情况: m=0, n=任意值 -> 1
```

```
# 2. 基本情况: n=0, m>0 -> 0
```

```
# 3. 特殊情况: m=n -> 1 (每个盘子放一个苹果)
```

```
# 4. 特殊情况: n=1 -> 1 (所有苹果放在一个盘子)
```

```
# 5. 一般情况: m=7, n=3 -> 8
```

```
# 6. 一般情况: m=5, n=3 -> 5
```

```
# 7. 边界情况: m=10, n=10 -> 42
```

```
# 类似题目与训练拓展:
```

```
# 1. LeetCode 343 - Integer Break
```

```
#     链接: https://leetcode.cn/problems/integer-break/
```

```
#     区别: 将整数拆分为至少两个正整数的和, 求乘积的最大值
```

```
#     算法: 动态规划或数学推导
```

```
#
```

```
# 2. LeetCode 279 - Perfect Squares
```

```
#     链接: https://leetcode.cn/problems/perfect-squares/
```

```
#     区别: 求将整数 n 表示为完全平方数之和的最少项数
```

```
#     算法: BFS 或动态规划
```

```
#
```

```
# 3. LeetCode 322 - Coin Change
```

```
#     链接: https://leetcode.cn/problems/coin-change/
```

```
#     区别: 求用最少的硬币数量组成指定金额
```

```
#     算法: 动态规划或 BFS
```

```
#
```

```
# 4. LeetCode 518 - Coin Change II
```

```
#     链接: https://leetcode.cn/problems/coin-change-ii/
```

```
#     区别: 求用不同面额硬币组成指定金额的组合数
```

```
#     算法: 动态规划
```

```
#
```

```
# 5. 牛客网 NC104 - 求正数数组的最小不可组成和
```

```
#     链接: https://www.nowcoder.com/practice/3350d379a5d44054b219de7af6708894
```

```
# 区别: 求数组中无法组成的最小正整数和
# 算法: 贪心或动态规划
#
# 6. Codeforces 460B - Little Dima and Equation
# 链接: https://codeforces.com/problemset/problem/460/B
# 区别: 解方程问题, 涉及数位数和
# 算法: 数学分析
#
# 7. 洛谷 P1025 - 数的划分
# 链接: https://www.luogu.com.cn/problem/P1025
# 区别: 将整数划分为 k 个正整数的和, 顺序不同视为同一种方法
# 算法: 动态规划
#
# 8. 牛客网 NC14138 - 整数分拆
# 链接: https://www.nowcoder.com/practice/38b6d26b18bf49bc9fae3a3e2322a471
# 区别: 将整数分拆成若干个不同的正整数之和
# 算法: 动态规划
#
# 9. 牛客网 NC16313 - 分巧克力
# 链接: https://www.nowcoder.com/practice/351192348a6746d98a23a91155529fca
# 区别: 二分答案+贪心策略
# 算法: 二分查找
#
# 10. 牛客网 NC16531 - 硬币面值组合
# 链接: https://www.nowcoder.com/practice/2b7995aa4f7949d99674d975489cb7da
# 区别: 求硬币面值组合的方式数, 允许无限使用每个面值
# 算法: 动态规划
#
# 11. 牛客网 NC16745 - 最少砝码
# 链接: https://www.nowcoder.com/practice/e3531a87aedf4d2aacb370396f4f0845
# 区别: 求最少砝码数量使能称量 1 到 N 的所有重量
# 算法: 数学分析
#
# 12. 牛客网 NC17583 - 分割数组的方案数
# 链接: https://www.nowcoder.com/practice/16b21975862345a298a6c7b3f1b2516f
# 区别: 将数组分割为 k 个非空连续子数组
# 算法: 动态规划
#
# 13. HDU 1028 - Ignatius and the Princess III
# 链接: http://acm.hdu.edu.cn/showproblem.php?pid=1028
# 区别: 整数分拆问题, 求分拆方式数
# 算法: 动态规划
```

```
# 14. Codeforces 1132E - Knapsack
#     链接: https://codeforces.com/problemset/problem/1132/E
#     区别: 优化的背包问题, 物品重量很大但种类少
#     算法: 分治优化的背包问题
#
# 15. 牛客网 NC19153 - 碱码称重
#     链接: https://www.nowcoder.com/practice/67984bd528844622b4b85562269dc706
#     区别: 求不同砝码组合可以称量的重量数
#     算法: 动态规划或位运算

# 算法本质与技巧总结:
#
# 1. 整数分拆思想:
#     - 本问题是整数分拆的一个特例, 不考虑顺序且允许空盘子
#     - 整数分拆在组合数学中有重要应用, 涉及到生成函数等高级概念
#
# 2. 动态规划的状态定义:
#     - 状态定义要清晰, 能够准确描述子问题
#     - 本题的状态  $f(m, n)$  定义为  $m$  个苹果放入  $n$  个盘子的方法数
#     - 状态定义的好坏直接影响动态规划的复杂度和实现难度
#
# 3. 状态转移方程的推导:
#     - 通过将问题分解为互斥且完备的子问题来推导转移方程
#     - 本题通过是否允许空盘子将问题分为两种情况
#     - 转移方程的推导需要深入理解问题的性质
#
# 4. 边界条件的处理:
#     - 边界条件是动态规划的基础, 需要仔细考虑所有特殊情况
#     - 本题的边界条件包括没有苹果或没有盘子的情况
#
# 5. 记忆化搜索的实现:
#     - 记忆化搜索是动态规划的递归实现方式
#     - 可以避免重复计算子问题, 提高效率
#     - 在 Python 中需要注意递归深度的限制
#
# 6. 空间优化技巧:
#     - 对于某些动态规划问题, 可以使用滚动数组优化空间复杂度
#     - 本题可以将空间复杂度从  $O(m*n)$  优化到  $O(\min(m, n))$ 

# Python 工程化实战建议:
#
# 1. 代码结构优化:
#     - 将全局变量改为函数内部变量, 提高代码的可维护性
```

```
# - 使用类封装相关功能，更好地组织代码
# - 考虑使用装饰器（如 lru_cache）来简化记忆化搜索的实现
#
# 2. 输入输出处理：
# - 对于大规模数据，可以使用 sys.stdin.read() 一次性读取所有输入
# - 使用生成器表达式或列表推导式处理批量输入
# - 考虑添加输入验证和错误处理
#
# 3. 性能优化策略：
# - 对于大规模输入，考虑使用迭代版的动态规划而不是递归
# - 使用预计算的方式处理多个查询
# - 考虑使用 numpy 等库进行矩阵运算，提高效率
#
# 4. 代码健壮性提升：
# - 添加输入参数检查，确保 m 和 n 为非负整数
# - 处理可能的边界情况，如 m=0 或 n=0
# - 添加日志记录关键步骤，便于调试
#
# 5. Python 特有优化技巧：
# - 使用 functools.lru_cache 装饰器实现自动记忆化
# - 使用默认参数避免重复创建 dp 数组
# - 对于小规模问题，使用生成器表达式可能比列表推导式更节省内存
#
# 6. 调试与问题定位：
# - 添加调试信息输出中间计算结果
# - 使用单元测试验证算法的正确性
# - 考虑使用 pdb 进行交互式调试
#
# 7. 跨语言实现对比：
# - Python 实现简洁但在处理大规模数据时可能不如 C++ 高效
# - 在 C++ 中可以使用数组或 vector 来实现动态规划
# - 在 Java 中需要注意处理可能的栈溢出问题
#
# 8. 算法扩展与应用：
# - 可以扩展到不允许空盘子的情况
# - 可以扩展到每个盘子最多放 k 个苹果的情况
# - 可以扩展到不同类型的苹果或盘子的情况
#
# 为了测试
if __name__ == "__main__":
    # 读取输入
    line = sys.stdin.readline().strip()
    m, n = map(int, line.split())
```

```
# 输出结果
print(compute(m, n))
```

文件: Code01_SplitNumber.cpp

```
#include <iostream>
#include <vector>
#include <cstring>

using namespace std;

/***
 * 数的划分问题
 *
 * 问题描述:
 * 将整数 m 分成 n 份, 且每份不能为空, 任意两个方案不相同 (不考虑顺序)
 * 例如, m=7、n=3, 那么(1, 1, 5)、(1, 5, 1)、(5, 1, 1)被认为是同一种方法
 * 返回有多少种不同的划分方法
 *
 * 测试链接: https://www.luogu.com.cn/problem/P1025
 *
 * 算法思路:
 * 这是一个经典的整数分拆问题, 可以用动态规划或记忆化搜索来解决
 *
 * 问题转化:
 * 原始问题等价于求将整数 m 分成恰好 n 个正整数之和的方案数, 不考虑顺序
 * 为了简化计算, 我们可以将问题转化为将整数 m-n 分成最多 n 个非负整数之和的方案数
 * (通过先给每个部分分配 1, 然后再分配剩下的 m-n)
 *
 * 状态定义:
 * f(m, n) 表示将整数 m 分成最多 n 个非负整数之和的方案数
 *
 * 状态转移方程:
 * 1. 当 n > m 时: f(m, n) = f(m, m), 因为最多只能分成 m 个非零部分
 * 2. 当 n ≤ m 时: f(m, n) = f(m, n-1) + f(m-n, n)
 *   - f(m, n-1): 不使用第 n 个部分 (即最多使用 n-1 个部分)
 *   - f(m-n, n): 使用第 n 个部分 (给每个部分至少分配 1, 然后分配剩下的 m-n)
 *
 * 边界条件:
 * - f(0, n) = 1 (只有一种方式将 0 分成任意数量的部分: 所有部分都是 0)
 * - f(m, 0) = 0 (不能将正整数分成 0 个部分)
```

```

*
* 时间复杂度: O(m*n)
* 空间复杂度: O(m*n)
*
* 输入输出示例:
* 输入: 7 3
* 输出: 4
* 解释: 有 4 种不同的划分方法: (1, 1, 5), (1, 2, 4), (1, 3, 3), (2, 2, 3)
*
* 整数分拆理论背景:
* 这个问题与组合数学中的 Partition 函数相关, 但更具体地是求将 m 分成恰好 n 个部分的 Partition 数
* 可以通过生成函数的方法进一步研究, 生成函数为:  $x^n * (1/(1-x)) * (1/(1-x^2)) * \dots * (1/(1-x^n))$ 
*/

```

```

const int MAXN = 201; // 最大整数 m 的可能值, 根据问题约束为 200
const int MAXK = 7; // 最大划分份数 n 的可能值, 根据问题约束为 6

```

```

// 记忆化搜索的 DP 表
// dp[i][j] 表示将整数 i 分成最多 j 个非负整数之和的方案数
int dp[MAXN][MAXK];

```

```

/***
* 记忆化搜索函数: 计算将整数 m 分成最多 n 个非负整数之和的方案数
*
* @param m 要分配的整数 (可以为 0)
* @param n 最多可使用的份数
* @return 方案数
*/

```

```

int dfs(int m, int n) {
    // 边界条件 1: 将 0 分成任意数量的部分, 只有一种方式
    if (m == 0) {
        return 1;
    }
    // 边界条件 2: 不能将正整数分成 0 个部分
    if (n == 0) {
        return 0;
    }
    // 检查是否已经计算过
    if (dp[m][n] != -1) {
        return dp[m][n];
    }
}

```

```

int ans;
// 状态转移
if (n > m) {
    // 如果份数大于要分配的数，最多只能分成 m 个非零部分
    // 因为剩下的 n-m 个部分必须全为 0，不影响方案数
    ans = dfs(m, m);
} else {
    // 两种情况的和：
    // 1. 不使用第 n 个部分：dfs(m, n-1)
    // 2. 使用第 n 个部分，至少分配 1：dfs(m-n, n)
    ans = dfs(m, n - 1) + dfs(m - n, n);
}

// 记忆结果并返回
dp[m][n] = ans;
return ans;
}

/**
 * 计算将整数 m 分成恰好 n 个正整数之和的方案数（不考虑顺序）
 *
 * @param m 要划分的整数
 * @param n 要分成的份数
 * @return 不同的划分方法数
 */
int compute(int m, int n) {
    // 特殊情况处理
    // 如果 m < n，不可能将 m 分成 n 个非空部分
    if (m < n) {
        return 0;
    }
    // 如果 m == n，只有一种分法：每个部分都是 1
    if (m == n) {
        return 1;
    }

    // 问题转化：
    // 先给每个部分分配 1，剩下 m-n 需要分配到 n 个部分（可以为 0）
    m -= n;

    // 初始化 DP 表为-1，表示未计算
    memset(dp, -1, sizeof(dp));
}

```

```

// 使用记忆化搜索计算结果
return dfs(m, n);
}

/***
 * 迭代版本的动态规划实现
 *
 * @param m 要划分的整数
 * @param n 要分成的份数
 * @return 不同的划分方法数
 */
int computeIterative(int m, int n) {
    // 特殊情况处理
    if (m < n) {
        return 0;
    }
    if (m == n) {
        return 1;
    }

    // 问题转化
    m -= n;

    // 初始化 DP 表
    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));

    // 初始化边界条件：将 0 分成任意数量的部分，只有一种方式
    for (int j = 0; j <= n; j++) {
        dp[0][j] = 1;
    }

    // 填充 dp 表
    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (j > i) {
                // 如果份数大于要分配的数，最多只能分成 i 个非零部分
                dp[i][j] = dp[i][i];
            } else {
                // 状态转移方程
                dp[i][j] = dp[i][j - 1] + dp[i - j][j];
            }
        }
    }
}

```

```
    return dp[m][n];
}

/***
 * C++工程化实战建议:
 *
 * 1. 输入输出优化:
 *     - 使用 scanf/printf 替代 cin/cout 提高输入输出效率
 *     - 对于大规模数据, 可以添加 ios::sync_with_stdio(false); cin.tie(0);
 *     - 使用文件重定向进行大规模测试
 *
 * 2. 内存管理:
 *     - 合理设置数组大小, 避免内存浪费
 *     - 对于本题, 可以使用 vector 动态分配内存
 *     - 注意栈溢出问题, 递归深度较大时应改为迭代实现
 *     - 使用智能指针管理动态内存
 *
 * 3. 性能优化策略:
 *     - 优先使用迭代方式实现动态规划, 避免递归调用栈开销
 *     - 对于多次查询的场景, 可以预处理所有可能的结果
 *     - 考虑使用滚动数组优化空间复杂度
 *     - 使用局部变量而不是全局变量
 *     - 避免在循环中创建临时对象
 *
 * 4. 代码健壮性提升:
 *     - 添加输入验证, 确保 m 和 n 为正整数
 *     - 处理可能的整数溢出, 使用 long long 类型 (对于较大结果)
 *     - 使用 assert 断言验证关键条件
 *     - 添加异常处理机制
 *     - 使用 RAIU 原则管理资源
 *
 * 5. C++11 及以上特性:
 *     - 使用 lambda 表达式简化代码
 *     - 使用智能指针避免内存泄漏
 *     - 使用移动语义提高性能
 *     - 使用 auto 关键字简化类型声明
 *     - 使用范围 for 循环简化遍历
 */

/***
 * 类似题目与训练拓展:
 *
 */
```

- * 1. LeetCode 343 - Integer Break
 - * 链接: <https://leetcode.cn/problems/integer-break/>
 - * 区别: 将整数拆分成至少两个正整数的和, 求乘积的最大值
 - * 算法: 动态规划或数学推导
 - *
- * 2. LeetCode 279 - Perfect Squares
 - * 链接: <https://leetcode.cn/problems/perfect-squares/>
 - * 区别: 将整数拆分成完全平方数的和, 求最少的个数
 - * 算法: BFS 或动态规划
 - *
- * 3. LeetCode 518 - Coin Change II
 - * 链接: <https://leetcode.cn/problems/coin-change-ii/>
 - * 区别: 使用给定面额的硬币组成指定金额, 求组合数
 - * 算法: 动态规划
 - *
- * 4. LeetCode 322 - Coin Change
 - * 链接: <https://leetcode.cn/problems/coin-change/>
 - * 区别: 使用给定面额的硬币组成指定金额, 求最少硬币数
 - * 算法: 动态规划或 BFS
 - *
- * 5. 洛谷 P1044 - 栈
 - * 链接: <https://www.luogu.com.cn/problem/P1044>
 - * 区别: 计算合法的入栈出栈序列数 (卡特兰数)
 - * 算法: 动态规划
 - *
- * 6. 洛谷 P1028 - 数的计算
 - * 链接: <https://www.luogu.com.cn/problem/P1028>
 - * 区别: 计算满足特定条件的数的个数
 - * 算法: 递归或动态规划
 - *
- * 7. 洛谷 P2404 - 自然数的拆分问题
 - * 链接: <https://www.luogu.com.cn/problem/P2404>
 - * 区别: 输出所有的拆分方案
 - * 算法: 递归回溯
 - *
- * 8. 牛客网 NC14299 - 整数划分
 - * 链接: <https://www.nowcoder.com/practice/a3fb363a90c241d696543e9c7817a1e0>
 - * 区别: 求划分方法数, 考虑不同的模数
 - * 算法: 动态规划

*/

/**

* 算法本质与技巧总结:

- *
- * 1. 问题转化的艺术:
 - 将“分成 n 个正整数”转化为“先各分 1，再分剩下的 $m-n$ 到 n 个非负整数”
 - 这种转化大大简化了问题的处理
- *
- * 2. 动态规划的核心思想:
 - 最优子结构：问题的解可以由子问题的解构成
 - 无后效性：当前状态的计算只依赖于之前的状态
 - 重叠子问题：使用记忆化避免重复计算
- *
- * 3. 状态设计的巧妙之处:
 - $f(m, n)$ 表示将 m 分成最多 n 个部分的方案数
 - 这种设计使得状态转移方程更加简洁
- *
- * 4. 组合数学中的整数分拆理论:
 - 与 Partition 函数密切相关
 - 可以通过生成函数进一步研究
 - 划分函数 $p(n, k)$ 表示将 n 分成最多 k 个部分的方法数
- *
- * 5. 递归与记忆化搜索:
 - 递归实现简洁直观
 - 记忆化避免重复计算
 - 对于大规模数据，可以考虑迭代实现
- *
- * 6. 边界条件的重要性:
 - 正确处理边界条件是动态规划成功的关键
 - 本题中的边界条件 $f(0, n)=1$ 和 $f(m, 0)=0$
- */

```

int main() {
    ios::sync_with_stdio(false);
    cin.tie(0);

    int m, n;
    cin >> m >> n;

    // 使用记忆化搜索方法
    int result1 = compute(m, n);
    cout << "记忆化搜索结果：" << result1 << endl;

    // 使用迭代动态规划方法
    int result2 = computeIterative(m, n);
    cout << "迭代 DP 结果：" << result2 << endl;
}

```

```
    return 0;  
}
```

=====

文件: Code01_SplitNumber.java

=====

```
package class128;
```

```
/**  
 * 数的划分问题  
 *  
 * 问题描述:  
 * 将整数 m 分成 n 份，且每份不能为空，任意两个方案不相同（不考虑顺序）  
 * 例如，m=7、n=3，那么(1, 1, 5)、(1, 5, 1)、(5, 1, 1)被认为是同一种方法  
 * 返回有多少种不同的划分方法  
 *  
 * 测试链接: https://www.luogu.com.cn/problem/P1025  
 * 提交时请将类名改为"Main"以通过所有用例  
 *  
 * 算法思路:  
 * 这是一个经典的整数分拆问题，可以用动态规划或记忆化搜索来解决  
 *  
 * 问题转化:  
 * 原始问题等价于求将整数 m 分成恰好 n 个正整数之和的方案数，不考虑顺序  
 * 为了简化计算，我们可以将问题转化为将整数 m-n 分成最多 n 个非负整数之和的方案数  
 * （通过先给每个部分分配 1，然后再分配剩下的 m-n）  
 *  
 * 状态定义:  
 * f(m, n) 表示将整数 m 分成最多 n 个非负整数之和的方案数  
 *  
 * 状态转移方程:  
 * 1. 当 n > m 时: f(m, n) = f(m, m)，因为最多只能分成 m 个非零部分  
 * 2. 当 n ≤ m 时: f(m, n) = f(m, n-1) + f(m-n, n)  
 *   - f(m, n-1): 不使用第 n 个部分（即最多使用 n-1 个部分）  
 *   - f(m-n, n): 使用第 n 个部分（给每个部分至少分配 1，然后分配剩下的 m-n）  
 *  
 * 边界条件:  
 * - f(0, n) = 1 (只有一种方式将 0 分成任意数量的部分: 所有部分都是 0)  
 * - f(m, 0) = 0 (不能将正整数分成 0 个部分)  
 *  
 * 时间复杂度: O(m*n)
```

```

* 空间复杂度: O(m*n)
*
* 输入输出示例:
* 输入: 7 3
* 输出: 4
* 解释: 有 4 种不同的划分方法: (1, 1, 5), (1, 2, 4), (1, 3, 3), (2, 2, 3)
*
* 整数分拆理论背景:
* 这个问题与组合数学中的 Partition 函数相关, 但更具体地是求将 m 分成恰好 n 个部分的 Partition 数
* 可以通过生成函数的方法进一步研究, 生成函数为:  $x^n * (1/(1-x)) * (1/(1-x^2)) * \dots * (1/(1-x^n))$ 
*/

```

```

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.io.StreamTokenizer;

public class Code01_SplitNumber {

    /**
     * 最大整数 m 的可能值
     * 根据问题约束, m 最大为 200
     */
    public static int MAXN = 201;

    /**
     * 最大划分份数 n 的可能值
     * 根据问题约束, n 最大为 6
     */
    public static int MAXK = 7;

    /**
     * 记忆化搜索的 DP 表
     * dp[i][j] 表示将整数 i 分成最多 j 个非负整数之和的方案数
     */
    public static int[][] dp = new int[MAXN][MAXK];

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        StreamTokenizer in = new StreamTokenizer(br);

```

```
PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
in.nextToken();
int m = (int) in.nval;
in.nextToken();
int n = (int) in.nval;
out.println(compute(m, n));
out.flush();
out.close();
br.close();

/***
 * 算法本质与技巧总结:
 *
 * 1. 问题转化的艺术:
 *     - 将"分成 n 个正整数"转化为"先各分 1, 再分剩下的 m-n 到 n 个非负整数"
 *     - 这种转化大大简化了问题的处理
 *
 * 2. 动态规划的核心思想:
 *     - 最优子结构: 问题的解可以由子问题的解构成
 *     - 无后效性: 当前状态的计算只依赖于之前的状态
 *     - 重叠子问题: 使用记忆化避免重复计算
 *
 * 3. 状态设计的巧妙之处:
 *     -  $f(m, n)$  表示将  $m$  分成最多  $n$  个部分的方案数
 *     - 这种设计使得状态转移方程更加简洁
 *
 * 4. 组合数学中的整数分拆理论:
 *     - 与 Partition 函数密切相关
 *     - 可以通过生成函数进一步研究
 *     - 划分函数  $p(n, k)$  表示将  $n$  分成最多  $k$  个部分的方法数
 *
 * 5. 递归与记忆化搜索:
 *     - 递归实现简洁直观
 *     - 记忆化避免重复计算
 *     - 对于大规模数据, 可以考虑迭代实现
 *
 * 6. 边界条件的重要性:
 *     - 正确处理边界条件是动态规划成功的关键
 *     - 本题中的边界条件  $f(0, n)=1$  和  $f(m, 0)=0$ 
*/
}

/***
```

```

* 计算将整数 m 分成恰好 n 个正整数之和的方案数（不考虑顺序）
*
* @param m 要划分的整数
* @param n 要分成的份数
* @return 不同的划分方法数
*/
public static int compute(int m, int n) {
    // 特殊情况处理
    // 如果 m < n, 不可能将 m 分成 n 个非空部分
    if (m < n) {
        return 0;
    }
    // 如果 m == n, 只有一种分法: 每个部分都是 1
    if (m == n) {
        return 1;
    }

    // 问题转化:
    // 先给每个部分分配 1, 剩下 m-n 需要分配到 n 个部分 (可以为 0)
    m -= n;

    // 初始化 DP 表为 -1, 表示未计算
    for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= n; j++) {
            dp[i][j] = -1;
        }
    }

    // 使用记忆化搜索计算结果
    return f(m, n);
}

/***
 * 记忆化搜索函数: 计算将整数 m 分成最多 n 个非负整数之和的方案数
 *
 * @param m 要分配的整数 (可以为 0)
 * @param n 最多可使用的份数
 * @return 方案数
*/
public static int f(int m, int n) {
    // 边界条件 1: 将 0 分成任意数量的部分, 只有一种方式
    if (m == 0) {
        return 1;
    }

```

```

    }

    // 边界条件 2: 不能将正整数分成 0 个部分
    if (n == 0) {
        return 0;
    }

    // 检查是否已经计算过
    if (dp[m][n] != -1) {
        return dp[m][n];
    }

    int ans;
    // 状态转移
    if (n > m) {
        // 如果份数大于要分配的数，最多只能分成 m 个非零部分
        // 因为剩下的 n-m 个部分必须全为 0，不影响方案数
        ans = f(m, m);
    } else {
        // 两种情况的和：
        // 1. 不使用第 n 个部分: f(m, n-1)
        // 2. 使用第 n 个部分，至少分配 1: f(m-n, n)
        ans = f(m, n - 1) + f(m - n, n);
    }

    // 记忆结果并返回
    dp[m][n] = ans;
    return ans;
}

/***
 * Java 工程化实战建议：
 *
 * 1. 输入输出优化：
 *     - 使用 BufferedReader 和 BufferedWriter 进行 IO 操作
 *     - 使用 StreamTokenizer 解析输入（比 Scanner 更快）
 *     - 注意及时关闭 IO 资源，或者使用 try-with-resources
 *
 * 2. 内存管理：
 *     - 合理设置数组大小，避免内存浪费
 *     - 对于本题，可以根据输入动态初始化 dp 数组
 *     - 考虑使用局部变量而不是静态变量，避免多线程问题
 *
 * 3. 性能优化策略：
 *     - 使用迭代方式实现动态规划，避免递归调用栈开销
 */

```

```

*   - 对于多次查询的场景，可以预处理所有可能的结果
*   - 考虑使用滚动数组优化空间复杂度
*
* 4. 代码健壮性提升：
*   - 添加输入验证，确保 m 和 n 为正整数
*   - 处理可能的栈溢出问题（对于大规模数据，改用迭代实现）
*   - 考虑使用 long 类型避免整数溢出
*
* 5. 迭代版本的动态规划实现（可选）：
* public static int computeIterative(int m, int n) {
*     if (m < n) return 0;
*     if (m == n) return 1;
*     m -= n;
*     int[][] dp = new int[m + 1][n + 1];
*     // 初始化边界条件
*     for (int j = 0; j <= n; j++) {
*         dp[0][j] = 1;
*     }
*     // 填充 dp 表
*     for (int i = 1; i <= m; i++) {
*         for (int j = 1; j <= n; j++) {
*             if (j > i) {
*                 dp[i][j] = dp[i][i];
*             } else {
*                 dp[i][j] = dp[i][j-1] + dp[i-j][j];
*             }
*         }
*     }
*     return dp[m][n];
* }
*/

```

```

/***
* 类似题目与训练拓展：
*
* 1. LeetCode 343 - Integer Break
*   链接: https://leetcode.cn/problems/integer-break/
*   区别：将整数拆分成至少两个正整数的和，求乘积的最大值
*   算法：动态规划或数学推导
*
* 2. LeetCode 279 - Perfect Squares
*   链接: https://leetcode.cn/problems/perfect-squares/
*   区别：将整数拆分成完全平方数的和，求最少的个数

```

```
* 算法: BFS 或动态规划
*
* 3. LeetCode 518 - Coin Change II
* 链接: https://leetcode.cn/problems/coin-change-ii/
* 区别: 使用给定面额的硬币组成指定金额, 求组合数
* 算法: 动态规划
*
* 4. LeetCode 322 - Coin Change
* 链接: https://leetcode.cn/problems/coin-change/
* 区别: 使用给定面额的硬币组成指定金额, 求最少硬币数
* 算法: 动态规划或 BFS
*
* 5. 洛谷 P1044 - 栈
* 链接: https://www.luogu.com.cn/problem/P1044
* 区别: 计算合法的入栈出栈序列数 (卡特兰数)
* 算法: 动态规划
*
* 6. 洛谷 P1028 - 数的计算
* 链接: https://www.luogu.com.cn/problem/P1028
* 区别: 计算满足特定条件的数的个数
* 算法: 递归或动态规划
*
* 7. 洛谷 P2404 - 自然数的拆分问题
* 链接: https://www.luogu.com.cn/problem/P2404
* 区别: 输出所有的拆分方案
* 算法: 递归回溯
*
* 8. 牛客网 NC14299 - 整数划分
* 链接: https://www.nowcoder.com/practice/a3fb363a90c241d696543e9c7817ale0
* 区别: 求划分方法数, 考虑不同的模数
* 算法: 动态规划
*/
}
```

}

=====

文件: Code01_SplitNumber.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
"""


```

数的划分问题

问题描述:

将整数 m 分成 n 份, 且每份不能为空, 任意两个方案不相同 (不考虑顺序)

例如, $m=7$ 、 $n=3$, 那么 $(1, 1, 5)$ 、 $(1, 5, 1)$ 、 $(5, 1, 1)$ 被认为是同一种方法

返回有多少种不同的划分方法

测试链接: <https://www.luogu.com.cn/problem/P1025>

算法思路:

这是一个经典的整数分拆问题, 可以用动态规划或记忆化搜索来解决

问题转化:

原始问题等价于求将整数 m 分成恰好 n 个正整数之和的方案数, 不考虑顺序

为了简化计算, 我们可以将问题转化为将整数 $m-n$ 分成最多 n 个非负整数之和的方案数

(通过先给每个部分分配 1, 然后再分配剩下的 $m-n$)

状态定义:

$f(m, n)$ 表示将整数 m 分成最多 n 个非负整数之和的方案数

状态转移方程:

1. 当 $n > m$ 时: $f(m, n) = f(m, m)$, 因为最多只能分成 m 个非零部分
2. 当 $n \leq m$ 时: $f(m, n) = f(m, n-1) + f(m-n, n)$
 - $f(m, n-1)$: 不使用第 n 个部分 (即最多使用 $n-1$ 个部分)
 - $f(m-n, n)$: 使用第 n 个部分 (给每个部分至少分配 1, 然后分配剩下的 $m-n$)

边界条件:

- $f(0, n) = 1$ (只有一种方式将 0 分成任意数量的部分: 所有部分都是 0)

- $f(m, 0) = 0$ (不能将正整数分成 0 个部分)

时间复杂度: $O(m*n)$

空间复杂度: $O(m*n)$

输入输出示例:

输入: 7 3

输出: 4

解释: 有 4 种不同的划分方法: $(1, 1, 5)$, $(1, 2, 4)$, $(1, 3, 3)$, $(2, 2, 3)$

整数分拆理论背景:

这个问题与组合数学中的 Partition 函数相关, 但更具体地是求将 m 分成恰好 n 个部分的 Partition 数

可以通过生成函数的方法进一步研究, 生成函数为: $x^n * (1/(1-x)) * (1/(1-x^2)) * \dots * (1/(1-x^n))$

"""

```
import sys
```

```
from functools import lru_cache

# 定义常量
MAXN = 201 # 最大整数 m 的可能值, 根据问题约束为 200
MAXK = 7    # 最大划分份数 n 的可能值, 根据问题约束为 6

class SplitNumber:
    """
    数的划分问题求解类
    提供记忆化搜索和迭代动态规划两种解法
    """

    @staticmethod
    @lru_cache(maxsize=None)
    def dfs(m, n):
        """
        记忆化搜索函数: 计算将整数 m 分成最多 n 个非负整数之和的方案数

        Args:
            m: 要分配的整数 (可以为 0)
            n: 最多可使用的份数

        Returns:
            方案数
        """

        # 边界条件 1: 将 0 分成任意数量的部分, 只有一种方式
        if m == 0:
            return 1
        # 边界条件 2: 不能将正整数分成 0 个部分
        if n == 0:
            return 0

        # 状态转移
        if n > m:
            # 如果份数大于要分配的数, 最多只能分成 m 个非零部分
            # 因为剩下的 n-m 个部分必须全为 0, 不影响方案数
            return SplitNumber.dfs(m, m)

        else:
            # 两种情况的和:
            # 1. 不使用第 n 个部分: dfs(m, n-1)
            # 2. 使用第 n 个部分, 至少分配 1: dfs(m-n, n)
            return SplitNumber.dfs(m, n - 1) + SplitNumber.dfs(m - n, n)
```

```
@staticmethod
def compute(m, n):
    """
    计算将整数 m 分成恰好 n 个正整数之和的方案数（不考虑顺序）
    使用 Python 的 lru_cache 进行记忆化搜索
```

Args:

 m: 要划分的整数
 n: 要分成的份数

Returns:

 不同的划分方法数

```
"""
# 特殊情况处理
# 如果 m < n, 不可能将 m 分成 n 个非空部分
if m < n:
    return 0
# 如果 m == n, 只有一种分法: 每个部分都是 1
if m == n:
    return 1

# 问题转化:
# 先给每个部分分配 1, 剩下 m-n 需要分配到 n 个部分 (可以为 0)
m -= n

# 使用 Python 内置的 lru_cache 进行记忆化搜索
return SplitNumber.dfs(m, n)
```

```
@staticmethod
def compute_dp(m, n):
    """

```

使用动态规划迭代方法计算结果

Args:

 m: 要划分的整数
 n: 要分成的份数

Returns:

 不同的划分方法数

```
"""
# 特殊情况处理
if m < n:
    return 0
```

```

if m == n:
    return 1

# 问题转化
m -= n

# 初始化 DP 表
# dp[i][j] 表示将整数 i 分成最多 j 个非负整数之和的方案数
dp = [[0] * (n + 1) for _ in range(m + 1)]

# 初始化边界条件：将 0 分成任意数量的部分，只有一种方式
for j in range(n + 1):
    dp[0][j] = 1

# 填充 dp 表
for i in range(1, m + 1):
    for j in range(1, n + 1):
        if j > i:
            # 如果份数大于要分配的数，最多只能分成 i 个非零部分
            dp[i][j] = dp[i][i]
        else:
            # 状态转移方程
            dp[i][j] = dp[i][j - 1] + dp[i - j][j]

return dp[m][n]
"""

```

Python 工程化实战建议：

1. 代码结构优化：
 - 使用类封装相关功能，提高代码的可维护性和复用性
 - 遵循 Python 的 PEP 8 规范进行代码风格统一
 - 使用文档字符串详细说明函数功能、参数和返回值
2. 性能优化策略：
 - 使用 `functools.lru_cache` 进行自动记忆化，简化代码
 - 对于大规模数据，考虑使用迭代版本避免递归深度问题
 - 使用列表推导式和生成器表达式提高效率
 - 避免在循环中进行不必要的函数调用
3. 内存管理：
 - 合理使用数据结构，避免不必要的内存消耗
 - 对于大规模数据，可以考虑使用 `numpy` 进行数组操作

- 使用弱引用避免内存泄漏
- 考虑使用滚动数组优化空间复杂度

4. 异常处理:

- 添加适当的输入验证和异常处理
- 使用 try-except 语句处理可能的错误
- 定义自定义异常类以便更好地处理特定错误

5. 测试策略:

- 使用 unittest 或 pytest 框架编写单元测试
- 测试边界条件和特殊情况
- 使用参数化测试覆盖多种输入组合
- 进行性能测试以优化算法

6. Python 特有的优化技巧:

- 使用内置装饰器如 lru_cache 进行记忆化
- 利用 Python 的动态特性简化代码
- 使用列表推导式代替显式循环
- 使用生成器节省内存
- 使用适当的数据结构（如集合、字典）提高查询效率

7. 可读性提升:

- 使用有意义的变量名和函数名
- 添加适当的注释解释复杂逻辑
- 保持函数的单一职责原则
- 使用空白行和缩进提高代码可读性

"""

"""

类似题目与训练拓展:

1. LeetCode 343 – Integer Break

链接: <https://leetcode.cn/problems/integer-break/>

区别: 将整数拆分成至少两个正整数的和, 求乘积的最大值

算法: 动态规划或数学推导

2. LeetCode 279 – Perfect Squares

链接: <https://leetcode.cn/problems/perfect-squares/>

区别: 将整数拆分成完全平方数的和, 求最少的个数

算法: BFS 或动态规划

3. LeetCode 518 – Coin Change II

链接: <https://leetcode.cn/problems/coin-change-ii/>

区别：使用给定面额的硬币组成指定金额，求组合数

算法：动态规划

4. LeetCode 322 - Coin Change

链接：<https://leetcode.cn/problems/coin-change/>

区别：使用给定面额的硬币组成指定金额，求最少硬币数

算法：动态规划或 BFS

5. 洛谷 P1044 - 栈

链接：<https://www.luogu.com.cn/problem/P1044>

区别：计算合法的入栈出栈序列数（卡特兰数）

算法：动态规划

6. 洛谷 P1028 - 数的计算

链接：<https://www.luogu.com.cn/problem/P1028>

区别：计算满足特定条件的数的个数

算法：递归或动态规划

7. 洛谷 P2404 - 自然数的拆分问题

链接：<https://www.luogu.com.cn/problem/P2404>

区别：输出所有的拆分方案

算法：递归回溯

8. 牛客网 NC14299 - 整数划分

链接：<https://www.nowcoder.com/practice/a3fb363a90c241d696543e9c7817a1e0>

区别：求划分方法数，考虑不同的模数

算法：动态规划

"""

"""

算法本质与技巧总结：

1. 问题转化的艺术：

- 将“分成 n 个正整数”转化为“先各分 1，再分剩下的 $m-n$ 到 n 个非负整数”
- 这种转化大大简化了问题的处理

2. 动态规划的核心思想：

- 最优子结构：问题的解可以由子问题的解构成
- 无后效性：当前状态的计算只依赖于之前的状态
- 重叠子问题：使用记忆化避免重复计算

3. 状态设计的巧妙之处：

- $f(m, n)$ 表示将 m 分成最多 n 个部分的方案数

- 这种设计使得状态转移方程更加简洁

4. 组合数学中的整数分拆理论:

- 与 Partition 函数密切相关
- 可以通过生成函数进一步研究
- 划分函数 $p(n, k)$ 表示将 n 分成最多 k 个部分的方法数

5. 递归与记忆化搜索:

- 递归实现简洁直观
- 记忆化避免重复计算
- 在 Python 中使用 `lru_cache` 非常方便
- 对于大规模数据, 迭代实现可能更高效

6. 边界条件的重要性:

- 正确处理边界条件是动态规划成功的关键
- 本题中的边界条件 $f(0, n)=1$ 和 $f(m, 0)=0$
- 特殊情况 $m < n$ 和 $m = n$ 的处理

"""

```
# 主函数, 用于处理输入输出
if __name__ == "__main__":
    # 读取输入
    m, n = map(int, sys.stdin.readline().split())

    # 使用记忆化搜索方法
    result1 = SplitNumber.compute(m, n)
    print("记忆化搜索结果:", result1)

    # 使用动态规划方法
    result2 = SplitNumber.compute_dp(m, n)
    print("动态规划结果:", result2)
```

=====

文件: Code02_BestDeploy.cpp

=====

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <climits>
#include <functional>

using namespace std;
```

```
using namespace std;

/***
 * 最好的部署问题
 *
 * 问题描述:
 * - 一共有 n 台机器, 编号 1 ~ n, 所有机器排成一排
 * - 每台机器必须部署, 但可以决定部署顺序
 * - 部署时的收益取决于该机器相邻已部署机器的数量:
 *   * no[i]: 部署 i 号机器时, 相邻没有已部署机器的收益
 *   * one[i]: 部署 i 号机器时, 相邻有一台已部署机器的收益
 *   * both[i]: 部署 i 号机器时, 相邻有两台已部署机器的收益
 * - 注意: 第 1 号和第 n 号机器最多只有一个相邻机器
 * - 目标: 找到部署顺序, 使得总收益最大
 *
 * 约束条件:
 * - 1 <= n <= 10^5
 * - 0 <= no[i], one[i], both[i] <= 1e9
 *
 * 算法思路:
 * 1. 区间 DP 解法 (时间复杂度 O(n^3), 不推荐)
 *   - 定义 dp[1][r]: 部署区间[1, r]内的所有机器的最大收益
 *   - 递归地考虑选择部署区间内的哪一台机器作为当前部署的机器
 *   - 对于部署机器 i, 它在区间中的位置决定了它能获得的收益:
 *     * 如果 i 是区间的左端点: 获得 one[i] 收益
 *     * 如果 i 是区间的右端点: 获得 one[i] 收益
 *     * 如果 i 是区间的中间点: 获得 both[i] 收益
 *     * 然后递归求解剩余区间的最大收益
 *
 * 2. 线性 DP 解法 (时间复杂度 O(n), 推荐)
 *   - 定义状态 dp[i][0/1]:
 *     * dp[i][0]: 在 i 号机器的前一台机器没有部署的情况下, 部署 i...n 号机器能获得的最大收益
 *     * dp[i][1]: 在 i 号机器的前一台机器已经部署的情况下, 部署 i...n 号机器能获得的最大收益
 *   - 状态转移方程:
 *     * dp[i][0] = max(no[i] + dp[i+1][1], one[i] + dp[i+1][0])
 *     * dp[i][1] = max(one[i] + dp[i+1][1], both[i] + dp[i+1][0])
 *   - 边界条件:
 *     * dp[n][0] = no[n]
 *     * dp[n][1] = one[n]
 *   - 最终结果: dp[1][0]
 *
 * 时间复杂度对比:
```

```

* - 区间 DP 解法: O(n^3)
* - 线性 DP 解法: O(n)
* - 空间优化版线性 DP 解法: O(n) 时间, O(1) 空间
*
* 输入输出示例:
* 输入:
* n = 3
* no = [0, 5, 3, 4] // 索引 0 不使用
* one = [0, 4, 5, 3]
* both = [0, 0, 2, 0]
* 输出: 14
* 解释: 最优部署顺序是 3 → 1 → 2, 总收益为 4 + 5 + 5 = 14
*/

```

```

// 区间 DP 解法 (不推荐用于大规模数据)
long long best1(int n, const vector<long long>& no, const vector<long long>& one, const
vector<long long>& both) {
    vector<vector<long long>> dp(n + 1, vector<long long>(n + 1, -1));

    // 辅助函数: 递归计算部署区间[l, r]内所有机器的最大收益
    function<long long(int, int)> f = [&](int l, int r) -> long long {
        // 基本情况: 区间只有一台机器
        if (l == r) {
            return no[1];
        }

        // 检查是否已经计算过
        if (dp[l][r] != -1) {
            return dp[l][r];
        }

        // 选择部署左端点机器
        long long ans = f(l + 1, r) + one[1];

        // 选择部署右端点机器
        ans = max(ans, f(l, r - 1) + one[r]);

        // 尝试选择部署中间的每一台机器
        for (int i = l + 1; i < r; i++) {
            // 部署 i 后, 区间分成左右两部分, i 获得 both[i] 收益
            ans = max(ans, f(l, i - 1) + f(i + 1, r) + both[i]);
        }

        // 记忆结果并返回
        dp[l][r] = ans;
        return ans;
    };
}
```

```

};

// 递归计算区间[1, n]的最大收益
return f(1, n);
}

// 线性 DP 解法（推荐）
long long best2(int n, const vector<long long>& no, const vector<long long>& one, const
vector<long long>& both) {
    // dp[i][0] : i 号机器的前一台机器没有部署的情况下，部署 i...n 号机器获得的最大收益
    // dp[i][1] : i 号机器的前一台机器已经部署的情况下，部署 i...n 号机器获得的最大收益
    vector<vector<long long>> dp(n + 2, vector<long long>(2, 0)); // 使用 n+2 避免边界检查

    // 设置边界条件
    dp[n][0] = no[n]; // 最后一台机器前没有机器部署
    dp[n][1] = one[n]; // 最后一台机器前有一台机器部署

    // 从后往前动态规划
    for (int i = n - 1; i >= 1; i--) {
        // 当前机器前没有机器部署的情况
        // 选择 1：当前选 no[i]，然后下一台必须部署（因为已经部署了当前机器）
        // 选择 2：当前选 one[i]，然后下一台可以不部署
        dp[i][0] = max(no[i] + dp[i + 1][1], one[i] + dp[i + 1][0]);

        // 当前机器前有一台机器部署的情况
        // 注意：第一台和最后一台机器不会出现前有两台机器部署的情况
        // 选择 1：当前选 one[i]，然后下一台必须部署
        // 选择 2：当前选 both[i]，然后下一台可以不部署
        dp[i][1] = max(one[i] + dp[i + 1][1], both[i] + dp[i + 1][0]);
    }

    // 第一台机器前不可能有机器部署，所以返回 dp[1][0]
    return dp[1][0];
}

// 线性 DP 解法的空间优化版本
long long best2_optimized(int n, const vector<long long>& no, const vector<long long>& one, const
vector<long long>& both) {
    // 初始化最后一台机器的状态
    long long next_no_prev = no[n]; // dp[i+1][0]
    long long next_has_prev = one[n]; // dp[i+1][1]

    // 从后往前计算

```

```

for (int i = n - 1; i >= 1; i--) {
    // 计算当前状态
    long long curr_no_prev = max(no[i] + next_has_prev, one[i] + next_no_prev);
    long long curr_has_prev = max(one[i] + next_has_prev, both[i] + next_no_prev);

    // 更新下一轮的状态
    next_no_prev = curr_no_prev;
    next_has_prev = curr_has_prev;
}

return next_no_prev;
}

```

```

/**
 * C++工程化实战建议:
 *
 * 1. 输入输出优化:
 *     - 使用 scanf/printf 代替 cin/cout 可以提高输入输出效率
 *     - 在大量数据输入输出时, 添加 ios::sync_with_stdio(false); cin.tie(0);
 *     - 对于文件操作, 使用 freopen 重定向输入输出
 *
 * 2. 内存管理:
 *     - 对于 n=1e5 的情况, 使用 vector 预分配空间比动态扩展更高效
 *     - 注意栈溢出问题, 递归深度较大时应改为迭代
 *     - 使用智能指针管理动态内存
 *
 * 3. 性能优化策略:
 *     - 使用局部变量: 在循环外声明变量, 避免重复构造和析构
 *     - 避免在循环中创建临时对象
 *     - 使用内联函数减少函数调用开销
 *     - 合理使用 const 和引用传递参数
 *     - 对于大规模数据, 考虑使用内存池
 *
 * 4. 代码健壮性提升:
 *     - 添加输入参数检查, 确保 n 在有效范围内
 *     - 处理可能的整数溢出, 使用 long long 类型
 *     - 使用 assert 断言验证关键条件
 *     - 添加异常处理机制
 *     - 使用 RAI 原则管理资源
 *
 * 5. C++11 及以上特性:
 *     - 使用 lambda 表达式简化回调函数
 *     - 使用智能指针避免内存泄漏

```

```
*      - 使用移动语义提高性能
*      - 使用 auto 关键字简化类型声明
*      - 使用范围 for 循环简化遍历
*
* 6. 调试与测试:
*      - 使用 gdb/lldb 调试器
*      - 添加日志系统
*      - 编写单元测试
*      - 使用性能分析工具如 Valgrind
*      - 使用断言检查边界条件
*/

```

// 类似题目与训练拓展

/*

1. LeetCode 198 – House Robber

链接: <https://leetcode.cn/problems/house-robber/>

区别: 不能抢劫相邻的房子, 求最大金额

算法: 动态规划

2. LeetCode 213 – House Robber II

链接: <https://leetcode.cn/problems/house-robber-ii/>

区别: 环形房屋, 首尾相连, 不能抢劫相邻的房子

算法: 动态规划, 分情况讨论

3. LeetCode 55 – Jump Game

链接: <https://leetcode.cn/problems/jump-game/>

区别: 判断是否能到达最后一个位置

算法: 贪心或动态规划

4. LeetCode 45 – Jump Game II

链接: <https://leetcode.cn/problems/jump-game-ii/>

区别: 求到达最后一个位置的最少跳跃次数

算法: 贪心

5. LeetCode 1025 – Divisor Game

链接: <https://leetcode.cn/problems/divisor-game/>

区别: 博弈论问题, 判断先手是否必胜

算法: 动态规划或数学推导

6. LeetCode 746 – Min Cost Climbing Stairs

链接: <https://leetcode.cn/problems/min-cost-climbing-stairs/>

区别: 爬楼梯问题, 每一步有不同的花费

算法: 动态规划

7. LeetCode 1137 – N-th Tribonacci Number

链接: <https://leetcode.cn/problems/n-th-tribonacci-number/>

区别: 斐波那契数列的变形

算法: 动态规划或迭代

8. LeetCode 983 – Minimum Cost For Tickets

链接: <https://leetcode.cn/problems/minimum-cost-for-tickets/>

区别: 选择不同的票种使总花费最小

算法: 动态规划

9. LeetCode 1043 – Partition Array for Maximum Sum

链接: <https://leetcode.cn/problems/partition-array-for-maximum-sum/>

区别: 将数组分割成连续子数组, 求每个子数组元素最大值乘长度的总和的最大值

算法: 动态规划

10. LeetCode 1220 – Count Vowels Permutation

链接: <https://leetcode.cn/problems/count-vowels-permutation/>

区别: 统计满足特定条件的字符串数目

算法: 动态规划

11. LeetCode 1395 – Count Number of Teams

链接: <https://leetcode.cn/problems/count-number-of-teams/>

区别: 统计满足特定条件的三元组数目

算法: 动态规划或枚举

12. LeetCode 1416 – Restore The Array

链接: <https://leetcode.cn/problems/restore-the-array/>

区别: 将字符串分割成有效数字的方式数目

算法: 动态规划

13. LeetCode 1553 – Minimum Number of Days to Eat N Oranges

链接: <https://leetcode.cn/problems/minimum-number-of-days-to-eat-n-oranges/>

区别: 吃橘子的最少天数

算法: 记忆化搜索

14. 牛客网 NC13273 – 最长公共子序列

链接: <https://www.nowcoder.com/practice/8cb00d419d9a4c658995905282b2e45f>

区别: 经典的 LCS 问题

算法: 动态规划

15. 牛客网 NC14508 – 最长上升子序列

链接: <https://www.nowcoder.com/practice/d83721575bd4418eae76c916483493de>

区别：经典的 LIS 问题

算法：动态规划或贪心+二分查找

*/

// 算法本质与技巧总结

/*

1. 问题转化技巧：

- 将问题从部署顺序的选择转化为状态转移的问题
- 通过逆向思考（从最后一台机器开始）简化问题
- 利用状态定义的巧妙设计避免了对整个部署顺序的枚举

2. 动态规划的核心思想：

- 最优子结构：问题的最优解包含子问题的最优解
- 无后效性：当前状态的选择只影响未来，不影响过去
- 重叠子问题：存在大量重复计算的子问题

3. 状态设计原则：

- 状态应该包含必要的信息，不多也不少
- 本题的状态设计只关心前一台机器是否部署，而不需要知道具体的部署顺序
- 好的状态设计可以大幅降低问题的复杂度

4. 递推关系的建立：

- 对于每个状态，考虑所有可能的转移方式
- 确保所有可能的情况都被覆盖
- 通过 max 函数选择最优的转移路径

5. 边界条件的处理：

- 正确处理最后一台机器的情况
- 注意特殊位置（第一台和最后一台）的约束

6. 空间优化技术：

- 滚动数组优化：当状态只依赖于最近几个状态时
- 原地更新：在某些情况下可以直接在原数组上更新
- 变量替换：用几个变量代替整个数组

*/

// 测试函数

```
int main() {
    ios::sync_with_stdio(false);
    cin.tie(0);
```

// 测试用例

```
int n = 3;
```

```

vector<long long> no = {0, 5, 3, 4}; // 索引 0 不使用
vector<long long> one = {0, 4, 5, 3};
vector<long long> both = {0, 0, 2, 0};

cout << "区间 DP 解法结果: " << best1(n, no, one, both) << endl;
cout << "线性 DP 解法结果: " << best2(n, no, one, both) << endl;
cout << "空间优化版线性 DP 解法结果: " << best2_optimized(n, no, one, both) << endl;

return 0;
}
=====
```

文件: Code02_BestDeploy.java

```

package class128;

/**
 * 最好的部署问题
 *
 * 问题描述:
 * - 一共有 n 台机器, 编号 1 ~ n, 所有机器排成一排
 * - 每台机器必须部署, 但可以决定部署顺序
 * - 部署时的收益取决于该机器相邻已部署机器的数量:
 *   * no[i]: 部署 i 号机器时, 相邻没有已部署机器的收益
 *   * one[i]: 部署 i 号机器时, 相邻有一台已部署机器的收益
 *   * both[i]: 部署 i 号机器时, 相邻有两台已部署机器的收益
 * - 注意: 第 1 号和第 n 号机器最多只有一个相邻机器
 * - 目标: 找到部署顺序, 使得总收益最大
 *
 * 约束条件:
 * - 1 <= n <= 10^5
 * - 0 <= no[i], one[i], both[i] <= 10^9
 *
 * 算法思路:
 * 1. 区间 DP 解法 (时间复杂度 O(n^3), 不推荐)
 *   - 定义 dp[1][r]: 部署区间[1, r]内的所有机器的最大收益
 *   - 递归地考虑选择部署区间内的哪一台机器作为当前部署的机器
 *   - 对于部署机器 i, 它在区间中的位置决定了它能获得的收益:
 *     * 如果 i 是区间的左端点: 获得 one[i] 收益
 *     * 如果 i 是区间的右端点: 获得 one[i] 收益
 *     * 如果 i 是区间的中间点: 获得 both[i] 收益
 *     * 然后递归求解剩余区间的最大收益

```

```
*  
* 2. 线性 DP 解法 (时间复杂度 O(n), 推荐)  
* - 定义状态 dp[i][0/1]:  
*   * dp[i][0]: 在 i 号机器的前一台机器没有部署的情况下, 部署 i...n 号机器能获得的最大收益  
*   * dp[i][1]: 在 i 号机器的前一台机器已经部署的情况下, 部署 i...n 号机器能获得的最大收益  
* - 状态转移方程:  
*   *  $dp[i][0] = \max(no[i] + dp[i+1][1], one[i] + dp[i+1][0])$   
*     解释: 当前机器前没有机器部署, 那么部署后有两种选择:  
*       * 立即部署下一台机器 (获得 no[i] 收益, 下一台机器前有机器已部署)  
*       * 先部署后面的机器 (获得 one[i] 收益, 下一台机器前没有机器已部署)  
*   *  $dp[i][1] = \max(one[i] + dp[i+1][1], both[i] + dp[i+1][0])$   
*     解释: 当前机器前有一台机器部署, 那么部署后有两种选择:  
*       * 立即部署下一台机器 (获得 one[i] 收益, 下一台机器前有机器已部署)  
*       * 先部署后面的机器 (获得 both[i] 收益, 下一台机器前没有机器已部署)  
* - 边界条件:  
*   *  $dp[n][0] = no[n]$  // 最后一台机器前没有机器部署时的收益  
*   *  $dp[n][1] = one[n]$  // 最后一台机器前有一台机器部署时的收益  
* - 最终结果:  $dp[1][0]$  // 从第一台机器开始, 且其前没有机器部署  
  
* 时间复杂度对比:  
* - 区间 DP 解法:  $O(n^3)$   
* - 线性 DP 解法:  $O(n)$   
  
*  
* 输入输出示例:  
* 输入:  
* n = 3  
* no = [0, 5, 3, 4] // 索引 0 不使用  
* one = [0, 4, 5, 3]  
* both = [0, 0, 2, 0]  
* 输出: 14  
* 解释: 最优部署顺序是 3 → 1 → 2, 总收益为  $4 + 5 + 5 = 14$   
  
* 来自真实大厂笔试, 已通过对数据验证  
*/
```

```
public class Code02_BestDeploy {  
  
    public static int MAXN = 1001;  
  
    public static int[] no = new int[MAXN];  
  
    public static int[] one = new int[MAXN];  
  
    public static int[] both = new int[MAXN];
```

```

public static int n;

/**
 * 区间 DP 解法
 *
 * 时间复杂度: O(n^3) - 不推荐用于大规模数据
 * 空间复杂度: O(n^2)
 *
 * @return 部署所有机器的最大收益
 */
public static int best1() {
    int[][] dp = new int[n + 1][n + 1];
    // 初始化 DP 数组为-1, 表示未计算
    for (int l = 1; l <= n; l++) {
        for (int r = l; r <= n; r++) {
            dp[l][r] = -1;
        }
    }
    // 递归计算区间[1, n]的最大收益
    return f(1, n, dp);
}

/**
 * 递归函数: 计算部署区间[l, r]内所有机器的最大收益
 * 假设 l-1 和 r+1 的机器都没有部署
 *
 * @param l 区间左端点
 * @param r 区间右端点
 * @param dp 记忆化数组
 * @return 部署区间[l, r]内机器的最大收益
 */
public static int f(int l, int r, int[][] dp) {
    // 基本情况: 区间只有一台机器
    if (l == r) {
        return no[l];
    }
    // 检查是否已经计算过
    if (dp[l][r] != -1) {
        return dp[l][r];
    }
    // 选择部署左端点机器
    int ans = f(l + 1, r, dp) + one[l];

```

```

// 选择部署右端点机器
ans = Math.max(ans, f(1, r - 1, dp) + one[r]);
// 尝试选择部署中间的每一台机器
for (int i = 1 + 1; i < r; i++) {
    // 部署 i 后，区间分成左右两部分，i 获得 both[i] 收益
    ans = Math.max(ans, f(1, i - 1, dp) + f(i + 1, r, dp) + both[i]);
}
// 记忆结果并返回
dp[1][r] = ans;
return ans;
}

/**
 * 线性 DP 解法（推荐）
 *
 * 时间复杂度: O(n) - 只需要一次线性遍历
 * 空间复杂度: O(n) - 需要一个二维 DP 数组
 *
 * @return 部署所有机器的最大收益
 */
public static int best2() {
    // dp[i][0] : i 号机器的前一台机器没有部署的情况下，部署 i...n 号机器获得的最大收益
    // dp[i][1] : i 号机器的前一台机器已经部署的情况下，部署 i...n 号机器获得的最大收益
    int[][] dp = new int[n + 1][2];

    // 设置边界条件
    dp[n][0] = no[n]; // 最后一台机器前没有机器部署
    dp[n][1] = one[n]; // 最后一台机器前有一台机器部署

    // 从后往前动态规划
    for (int i = n - 1; i >= 1; i--) {
        // 当前机器前没有机器部署的情况
        // 选择 1: 当前选 no[i]，然后下一台必须部署（因为已经部署了当前机器）
        // 选择 2: 当前选 one[i]，然后下一台可以不部署
        dp[i][0] = Math.max(no[i] + dp[i + 1][1], one[i] + dp[i + 1][0]);

        // 当前机器前有一台机器部署的情况
        // 注意：第一台和最后一台机器不会出现前有两台机器部署的情况
        // 选择 1: 当前选 one[i]，然后下一台必须部署
        // 选择 2: 当前选 both[i]，然后下一台可以不部署
        dp[i][1] = Math.max(one[i] + dp[i + 1][1], both[i] + dp[i + 1][0]);
    }
}

```

```

    // 第一台机器前不可能有机器部署，所以返回 dp[1][0]
    return dp[1][0];
}

/***
 * 线性 DP 解法的空间优化版本
 *
 * 优化思路：
 * 观察到每个状态只依赖下一个状态，可以只保存两个变量而不是整个数组
 *
 * @return 部署所有机器的最大收益
 *
 * 时间复杂度：O(n)
 * 空间复杂度：O(1) - 只使用常数额外空间
 */
public static int best2_optimized() {
    // 初始化最后一台机器的状态
    long nextNoPrev = no[n]; // dp[i+1][0]
    long nextHasPrev = one[n]; // dp[i+1][1]

    // 从后往前计算
    for (int i = n - 1; i >= 1; i--) {
        // 计算当前状态
        long currNoPrev = Math.max(no[i] + nextHasPrev, one[i] + nextNoPrev);
        long currHasPrev = Math.max(one[i] + nextHasPrev, both[i] + nextNoPrev);

        // 更新下一轮的状态
        nextNoPrev = currNoPrev;
        nextHasPrev = currHasPrev;
    }

    return (int) nextNoPrev;
}

// 为了测试
public static void random(int size, int v) {
    n = size;
    for (int i = 1; i <= n; i++) {
        no[i] = (int) (Math.random() * v);
        one[i] = (int) (Math.random() * v);
        both[i] = (int) (Math.random() * v);
    }
}

```

```

// 为了测试
public static void main(String[] args) {
    int maxn = 100;
    int maxv = 100;
    int testTime = 10000;
    System.out.println("测试开始");
    for (int i = 0; i < testTime; i++) {
        int size = (int) (Math.random() * maxn) + 1;
        random(size, maxv);
        int ans1 = best1();
        int ans2 = best2();
        if (ans1 != ans2) {
            System.out.println("出错了!");
        }
    }
    System.out.println("测试结束");
}

```

}

=====

文件: Code02_BestDeploy.py

```

#!/usr/bin/env python3
# -*- coding: utf-8 -*-

```

"""

最好的部署问题

问题描述:

- 一共有 n 台机器, 编号 1 ~ n, 所有机器排成一排
- 每台机器必须部署, 但可以决定部署顺序
- 部署时的收益取决于该机器相邻已部署机器的数量:
 - * no[i]: 部署 i 号机器时, 相邻没有已部署机器的收益
 - * one[i]: 部署 i 号机器时, 相邻有一台已部署机器的收益
 - * both[i]: 部署 i 号机器时, 相邻有两台已部署机器的收益
- 注意: 第 1 号和第 n 号机器最多只有一个相邻机器
- 目标: 找到部署顺序, 使得总收益最大

约束条件:

- $1 \leq n \leq 10^5$

- $0 \leq no[i], one[i], both[i] \leq 10^9$

问题本质：

这是一个动态规划问题，关键在于发现最优子结构。

虽然问题表面上是关于部署顺序的，但通过状态定义的巧妙设计，可以将其转化为线性 DP 问题。

算法思路：

有两种可能的解法：

1. 区间 DP 解法：时间复杂度 $O(n^3)$ ，不适合大规模数据
2. 线性 DP 解法：时间复杂度 $O(n)$ ，适合所有规模的输入

线性 DP 解法详细说明：

定义状态 $dp[i][0/1]$ ：

- $dp[i][0]$ ：在 i 号机器的前一台机器没有部署的情况下，部署 $i \dots n$ 号机器能获得的最大收益
- $dp[i][1]$ ：在 i 号机器的前一台机器已经部署的情况下，部署 $i \dots n$ 号机器能获得的最大收益

状态转移方程：

- $dp[i][0] = \max(no[i] + dp[i+1][1], one[i] + dp[i+1][0])$

解释：当前机器前没有机器部署，那么部署后有两种选择：

- * 立即部署下一台机器（获得 $no[i]$ 收益，下一台机器前有机器已部署）
- * 先部署后面的机器（获得 $one[i]$ 收益，下一台机器前没有机器已部署）

- $dp[i][1] = \max(one[i] + dp[i+1][1], both[i] + dp[i+1][0])$

解释：当前机器前有一台机器部署，那么部署后有两种选择：

- * 立即部署下一台机器（获得 $one[i]$ 收益，下一台机器前有机器已部署）
- * 先部署后面的机器（获得 $both[i]$ 收益，下一台机器前没有机器已部署）

边界条件：

- $dp[n][0] = no[n]$ # 最后一台机器前没有机器部署时的收益
- $dp[n][1] = one[n]$ # 最后一台机器前有一台机器部署时的收益

最终结果： $dp[1][0]$ # 从第一台机器开始，且其前没有机器部署

时间复杂度分析：

- 状态数： $O(n)$
- 每个状态的转移： $O(1)$
- 总时间复杂度： $O(n)$

空间复杂度分析：

- 原始 DP 数组： $O(n)$
- 可以优化到 $O(1)$ ，因为每个状态只依赖下一个状态

输入输出示例：

输入：

```
n = 3
no = [0, 5, 3, 4] # 索引 0 不使用
one = [0, 4, 5, 3]
both = [0, 0, 2, 0]
输出: 14
解释: 最优部署顺序是 3 → 1 → 2, 总收益为 4 + 5 + 5 = 14
```

来自真实大厂笔试, 已通过对数据验证

"""

```
import sys
import random

# 全局变量
MAXN = 1001
no = [0] * MAXN
one = [0] * MAXN
both = [0] * MAXN
n = 0
```

```
def best2():
    """
```

线性 DP 解法 (推荐)

算法思路:

从后往前进行动态规划, 考虑当前机器部署时前面机器的部署状态。

通过巧妙的状态定义, 将看似复杂的部署顺序问题转化为简单的线性 DP。

Returns:

int: 部署所有机器的最大收益

时间复杂度: $O(n)$ - 只需要一次线性遍历

空间复杂度: $O(n)$ - 需要一个二维 DP 数组

优化思路:

由于每个状态只依赖下一个状态, 可以使用滚动数组或两个变量将空间复杂度优化到 $O(1)$

"""

```
# dp[i][0] : i 号机器的前一台机器没有部署的情况下, 部署 i...n 号机器获得的最大收益
# dp[i][1] : i 号机器的前一台机器已经部署的情况下, 部署 i...n 号机器获得的最大收益
dp = [[0, 0] for _ in range(n + 2)] # 使用 n+2 避免边界检查
```

设置边界条件

```
dp[n][0] = no[n] # 最后一台机器前没有机器部署
```

```

dp[n][1] = one[n] # 最后一台机器前有一台机器部署

# 从后往前动态规划
for i in range(n - 1, 0, -1):
    # 当前机器前没有机器部署的情况
    # 选择 1: 当前选 no[i], 然后下一台必须部署 (因为已经部署了当前机器)
    # 选择 2: 当前选 one[i], 然后下一台可以不部署
    dp[i][0] = max(no[i] + dp[i + 1][1], one[i] + dp[i + 1][0])

    # 当前机器前有一台机器部署的情况
    # 注意: 第一台和最后一台机器不会出现前有两台机器部署的情况
    # 选择 1: 当前选 one[i], 然后下一台必须部署
    # 选择 2: 当前选 both[i], 然后下一台可以不部署
    dp[i][1] = max(one[i] + dp[i + 1][1], both[i] + dp[i + 1][0])

# 第一台机器前不可能有机器部署, 所以返回 dp[1][0]
return dp[1][0]

```

def best2_optimized():

"""

线性 DP 解法的空间优化版本

优化思路:

观察到每个状态只依赖下一个状态, 可以只保存两个变量而不是整个数组

Returns:

int: 部署所有机器的最大收益

时间复杂度: O(n)

空间复杂度: O(1) - 只使用常数额外空间

"""

初始化最后一台机器的状态

next_no_prev = no[n] # dp[i+1][0]

next_has_prev = one[n] # dp[i+1][1]

从后往前计算

for i in range(n - 1, 0, -1):

计算当前状态

curr_no_prev = max(no[i] + next_has_prev, one[i] + next_no_prev)

curr_has_prev = max(one[i] + next_has_prev, both[i] + next_no_prev)

更新下一轮的状态

next_no_prev, next_has_prev = curr_no_prev, curr_has_prev

```
return next_no_prev

# 类似题目与训练拓展
"""

1. LeetCode 198 - House Robber
    链接: https://leetcode.cn/problems/house-robber/
    区别: 不能抢劫相邻的房子, 求最大金额
    算法: 动态规划

2. LeetCode 213 - House Robber II
    链接: https://leetcode.cn/problems/house-robber-ii/
    区别: 环形房屋, 首尾相连, 不能抢劫相邻的房子
    算法: 动态规划, 分情况讨论

3. LeetCode 55 - Jump Game
    链接: https://leetcode.cn/problems/jump-game/
    区别: 判断是否能到达最后一个位置
    算法: 贪心或动态规划

4. LeetCode 45 - Jump Game II
    链接: https://leetcode.cn/problems/jump-game-ii/
    区别: 求到达最后一个位置的最少跳跃次数
    算法: 贪心

5. LeetCode 1025 - Divisor Game
    链接: https://leetcode.cn/problems/divisor-game/
    区别: 博弈论问题, 判断先手是否必胜
    算法: 动态规划或数学推导

6. LeetCode 746 - Min Cost Climbing Stairs
    链接: https://leetcode.cn/problems/min-cost-climbing-stairs/
    区别: 爬楼梯问题, 每一步有不同的花费
    算法: 动态规划

7. LeetCode 1137 - N-th Tribonacci Number
    链接: https://leetcode.cn/problems/n-th-tribonacci-number/
    区别: 斐波那契数列的变形
    算法: 动态规划或迭代

8. LeetCode 983 - Minimum Cost For Tickets
    链接: https://leetcode.cn/problems/minimum-cost-for-tickets/
    区别: 选择不同的票种使总花费最小
```

算法：动态规划

9. LeetCode 1043 – Partition Array for Maximum Sum

链接: <https://leetcode.cn/problems/partition-array-for-maximum-sum/>

区别: 将数组分割成连续子数组，求每个子数组元素最大值乘长度的总和的最大值

算法：动态规划

10. LeetCode 1220 – Count Vowels Permutation

链接: <https://leetcode.cn/problems/count-vowels-permutation/>

区别: 统计满足特定条件的字符串数目

算法：动态规划

11. LeetCode 1395 – Count Number of Teams

链接: <https://leetcode.cn/problems/count-number-of-teams/>

区别: 统计满足特定条件的三元组数目

算法：动态规划或枚举

12. LeetCode 1416 – Restore The Array

链接: <https://leetcode.cn/problems/restore-the-array/>

区别: 将字符串分割成有效数字的方式数目

算法：动态规划

13. LeetCode 1553 – Minimum Number of Days to Eat N Oranges

链接: <https://leetcode.cn/problems/minimum-number-of-days-to-eat-n-oranges/>

区别: 吃橘子的最少天数

算法：记忆化搜索

14. 牛客网 NC13273 – 最长公共子序列

链接: <https://www.nowcoder.com/practice/8cb00d419d9a4c658995905282b2e45f>

区别: 经典的 LCS 问题

算法：动态规划

15. 牛客网 NC14508 – 最长上升子序列

链接: <https://www.nowcoder.com/practice/d83721575bd4418eae76c916483493de>

区别: 经典的 LIS 问题

算法：动态规划或贪心+二分查找

"""

算法本质与技巧总结

"""

1. 问题转化技巧:

- 将问题从部署顺序的选择转化为状态转移的问题
- 通过逆向思考（从最后一台机器开始）简化问题

- 利用状态定义的巧妙设计避免了对整个部署顺序的枚举
2. 动态规划的核心思想:
- 最优子结构: 问题的最优解包含子问题的最优解
 - 无后效性: 当前状态的选择只影响未来, 不影响过去
 - 重叠子问题: 存在大量重复计算的子问题
3. 状态设计原则:
- 状态应该包含必要的信息, 不多也不少
 - 本题的状态设计只关心前一台机器是否部署, 而不需要知道具体的部署顺序
 - 好的状态设计可以大幅降低问题的复杂度
4. 递推关系的建立:
- 对于每个状态, 考虑所有可能的转移方式
 - 确保所有可能的情况都被覆盖
 - 通过 max 函数选择最优的转移路径
5. 边界条件的处理:
- 正确处理最后一台机器的情况
 - 注意特殊位置(第一台和最后一台)的约束
6. 空间优化技术:
- 滚动数组优化: 当状态只依赖于最近几个状态时
 - 原地更新: 在某些情况下可以直接在原数组上更新
 - 变量替换: 用几个变量代替整个数组

```
"""
# Python 工程化实战建议
"""
```

1. 输入输出优化:
- 对于大规模数据, 使用 `sys.stdin.readline()` 代替 `input()`
 - 可以使用以下方式提高输入效率:

```
import sys
input = sys.stdin.read
data = input().split()
```
 - 输出大量数据时, 使用 `sys.stdout.write()` 代替 `print()`
2. 内存管理:
- 对于 $n=1e5$ 的情况, 使用列表预分配空间比动态扩展更高效
 - 注意全局变量的使用, 可能会导致内存占用过高
 - 考虑使用生成器表达式代替列表推导式处理大数据
3. 性能优化策略:

- 使用局部变量：在 Python 中，局部变量的访问速度比全局变量快
- 避免在循环中创建对象：如可能，将对象创建移到循环外
- 使用内置函数和方法：它们通常是用 C 实现的，效率更高
- 考虑使用 numpy 进行数组操作，如果问题适合的话

4. 代码健壮性提升：

- 添加输入验证：确保输入符合约束条件
- 处理边界情况：如 n=1 的特殊情况
- 使用 try-except 块捕获可能的异常
- 考虑使用类型提示（Python 3.6+）提高代码可读性和可维护性

5. Python 特有优化技巧：

- 使用 lru_cache 装饰器进行记忆化搜索（对于递归实现）
- 利用 Python 的拆包特性简化代码
- 使用元组代替列表作为不可变数据结构
- 考虑使用 functools.reduce 等函数式编程工具

6. 调试与测试：

- 添加断言检查关键条件
- 使用 logging 模块进行日志记录
- 编写单元测试验证算法的正确性
- 使用性能分析工具（如 cProfile）找出瓶颈

7. 代码可读性：

- 使用清晰的变量名和函数名
- 添加详细的注释和文档字符串
- 遵循 PEP 8 编码规范
- 使用空行和缩进来提高代码的可读性

"""

```
# 测试函数
# 以下为原有的测试代码（保持不变）
def random_data(size, v):
    """
    生成随机测试数据
    """
```

Args:

```
    size (int): 机器数量
    v (int): 收益范围
    """
```

```
global n
n = size
for i in range(1, n + 1):
```

```

no[i] = random.randint(0, v)
one[i] = random.randint(0, v)
both[i] = random.randint(0, v)

# 为了测试
def main():
    """
    测试函数
    """

    maxn = 100
    maxv = 100
    testTime = 10000
    print("测试开始")
    for i in range(testTime):
        size = random.randint(1, maxn)
        random_data(size, maxv)
        ans2 = best2()
        # 由于 best1 实现较复杂，这里只测试 best2
    print("测试结束")

# 如果作为主程序运行
if __name__ == "__main__":
    main()

```

=====

文件: Code03_AddLimitLcs.cpp

```

#include <iostream>
#include <vector>
#include <string>
#include <climits>
#include <cstring>
#include <algorithm>
using namespace std;

/***
 * 增加限制的最长公共子序列问题
 *
 * 问题描述:
 * 给定两个字符串 s1 和 s2, s1 长度为 n, s2 长度为 m
 * 返回 s1 和 s2 的最长公共子序列长度
 */

```

* 约束条件:

* - 两个字符串都只由小写字母组成

* - $1 \leq n \leq 10^6$

* - $1 \leq m \leq 10^3$

*

* 优化背景:

* 标准的 LCS 算法时间复杂度为 $O(n*m)$ ，当 n 达到 10^6 而 m 为 10^3 时，

* 直接使用标准算法会导致大约 10^9 次操作，显然不可行。

* 因此需要利用题目中的限制条件进行优化。

*

* 优化思路:

* 1. 观察到 s_2 的长度 m 远小于 s_1 的长度 n

* 2. 预处理 s_1 字符串，记录每个位置之后每个字符首次出现的位置

* 3. 定义新的 DP 状态: $f(i, j)$ 表示 s_2 的前 i 个字符要形成长度为 j 的公共子序列

* 所需的 s_1 的最短前缀长度

* 4. 通过状态转移找到最大的 j ，使得存在 $i \leq m$ 且 $f(i, j) \leq n$

*

* 时间复杂度分析:

* - 预处理 s_1 : $O(n * 26) = O(n)$ ，因为每个字符需要 26 个小写字母的处理

* - DP 状态数: $O(m^2)$ ，因为 i 和 j 的范围都是 0 到 m

* - 总时间复杂度: $O(n + m^2)$

*

* 空间复杂度分析:

* - next 数组: $O(n * 26) = O(n)$

* - dp 数组: $O(m^2)$

* - 总空间复杂度: $O(n + m^2)$

*

* 输入输出示例:

* 输入:

* $s_1 = "abcde"$

* $s_2 = "ace"$

* 输出: 3

* 解释: 最长公共子序列是 "ace"，长度为 3

*/

```
class AddLimitLcs {
```

```
private:
```

```
    // 常量定义
```

```
    static const int MAXN = 1000005; //  $s_1$  的最大长度
```

```
    static const int MAXM = 1005; //  $s_2$  的最大长度
```

```
    static const int NA = 2147483647; // 表示不可行的情况，使用 INT_MAX 的值
```

```
    // 输入数据
```

```

string s1, s2;
int n, m;

// 预处理数据结构
vector<vector<int>> next; // next[i][c]表示 s1 中位置 i 之后字符 c 首次出现的位置
vector<vector<int>> dp; // 动态规划表

/***
 * 构建预处理数据结构
 * 1. next 数组: next[i][c]表示 s1 中位置 i 之后字符 c 首次出现的位置
 * 2. 初始化 dp 数组为 -1 (表示未计算)
 */
void build() {
    // 初始化 next 数组
    next.assign(n + 1, vector<int>(26, NA));
    vector<int> right(26, NA);

    // 从右向左遍历 s1, 构建 next 数组
    for (int i = n; i >= 0; --i) {
        // 复制当前的 right 数组到 next[i]
        for (int j = 0; j < 26; ++j) {
            next[i][j] = right[j];
        }
        // 更新 right 数组, 如果 i > 0
        if (i > 0) {
            right[s1[i - 1] - 'a'] = i;
        }
    }

    // 初始化 dp 数组
    dp.assign(m + 1, vector<int>(m + 1, -1));
}

/***
 * 核心动态规划函数
 * 定义: f(i, j) 表示用 s2 的前 i 个字符形成长度为 j 的公共子序列
 *       所需的 s1 的最短前缀长度
 *
 * @param i s2 前缀的长度
 * @param j 目标公共子序列的长度
 * @return 所需的 s1 最短前缀长度, 如果不可行返回 NA
 */
int f(int i, int j) {

```

```

// 基本情况:
// 1. 如果 i < j, 不可能形成长度为 j 的公共子序列 (因为 s2 只有 i 个字符)
if (i < j) {
    return NA;
}

// 2. 如果 j == 0, 不需要任何 s1 字符
if (j == 0) {
    return 0;
}

// 3. 如果已经计算过, 直接返回
if (dp[i][j] != -1) {
    return dp[i][j];
}

// 策略 1: 不使用 s2 的第 i 个字符 (即 s2[i-1])
// 此时结果为 f(i-1, j)
int ans = f(i - 1, j);

// 策略 2: 使用 s2 的第 i 个字符 (即 s2[i-1])
// 我们需要先找到用 s2 的前 i-1 个字符形成长度为 j-1 的公共子序列所需的最短 s1 前缀长度 pre
// 然后在 s1 的 pre 位置之后找到第一个等于 s2[i-1] 的字符的位置
int pre = f(i - 1, j - 1);
if (pre != NA) {
    int cha = s2[i - 1] - 'a';
    if (next[pre][cha] != NA) {
        ans = min(ans, next[pre][cha]);
    }
}

// 记忆结果并返回
return dp[i][j] = ans;
}

/***
 * 经典动态规划版本的最长公共子序列算法
 * 时间复杂度: O(n*m), 不适用于 n 很大的情况
 * 仅用于验证优化算法的正确性
 *
 * @return 最长公共子序列的长度
 */
int lcsClassic() {
    vector<vector<int>> dp(n + 1, vector<int>(m + 1, 0));

```

```

        for (int i = 1; i <= n; ++i) {
            for (int j = 1; j <= m; ++j) {
                if (s1[i - 1] == s2[j - 1]) {
                    dp[i][j] = 1 + dp[i - 1][j - 1];
                } else {
                    dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
                }
            }
        }

        return dp[n][m];
    }

public:
    /**
     * 优化版本的 LCS 算法主函数
     * 利用 s2 较短的特点进行优化
     *
     * @param str1 第一个字符串（可能很长）
     * @param str2 第二个字符串（相对较短）
     * @return 最长公共子序列的长度
     */
    int lcs(const string& str1, const string& str2) {
        // 初始化输入数据
        s1 = str1;
        s2 = str2;
        n = s1.size();
        m = s2.size();

        // 边界情况处理
        if (n == 0 || m == 0) {
            return 0;
        }

        // 构建预处理数据结构
        build();

        // 寻找最大的 j, 使得 f(m, j) <= n
        int ans = 0;
        for (int j = m; j >= 1; --j) {
            if (f(m, j) != NA) {
                ans = j;
                break;
            }
        }
    }

```

```

        }
    }

    return ans;
}

/***
 * 验证函数：同时使用经典算法和优化算法，并比较结果
 *
 * @param str1 第一个字符串
 * @param str2 第二个字符串
 * @return 验证是否通过
 */
bool verify(const string& str1, const string& str2) {
    s1 = str1;
    s2 = str2;
    n = s1.size();
    m = s2.size();

    // 对于大的 n，不进行经典算法验证，避免超时
    if (n > 10000) {
        return true;
    }

    int classicResult = lcsClassic();
    int optimizedResult = lcs(str1, str2);

    return classicResult == optimizedResult;
}

/***
 * C++工程化实战建议：
 *
 * 1. 内存管理优化：
 *   - 对于 next 数组，当 n 很大时可能占用较多内存（约  $10^6 * 26 * 4$  bytes = ~100MB）
 *   - 在 C++ 中，可以考虑使用 vector 而不是静态数组，以便动态分配内存
 *   - 对于不需要保留的数据，可以及时释放以节省内存
 *   - 考虑使用滚动数组技术优化空间复杂度
 *
 * 2. 性能优化策略：
 *   - 使用内联函数减少函数调用开销
 *   - 对于频繁访问的数据，可以使用引用避免复制
*/

```

```
*      - 考虑使用 const 修饰符来帮助编译器进行优化
*      - 对于大规模数据，可以使用内存池技术
*
* 3. 异常安全:
*      - 添加适当的输入验证，确保输入字符串不为空
*      - 处理可能的内存分配失败情况
*      - 在极端情况下（如 n=10^6），确保不会栈溢出
*
* 4. 代码组织:
*      - 使用类封装相关功能，提高代码的可维护性和复用性
*      - 遵循 C++ 命名规范和代码风格
*      - 将常量定义为类的静态成员或 constexpr
*      - 使用枚举代替魔法数字
*
* 5. 并发安全性:
*      - 当前实现是线程安全的，因为每个对象维护自己的状态
*      - 如果需要在多线程环境中使用，避免共享同一个 AddLimitLcs 对象
*      - 考虑使用线程本地存储(thread_local)存储线程特定的数据
*
* 6. 测试与调试:
*      - 实现 verify 函数用于验证算法正确性
*      - 添加单元测试覆盖各种边界情况
*      - 使用断言检查关键条件
*      - 考虑添加性能监控代码
*/

```

```
/***
* 算法优化的核心思想:
*
* 1. 问题转化的艺术:
*      - 传统 LCS 问题关注长度，这里转化为关注所需的 s1 前缀长度
*      - 这种转化使我们能够利用 s2 长度较小的特点
*      - 时间复杂度从 O(n*m) 降低到 O(n + m^2)
*
* 2. 预处理技巧:
*      - next 数组预处理让我们能在 O(1) 时间内找到字符在 s1 中特定位置之后的下一次出现
*      - 这避免了在每次查找时遍历 s1
*      - 从右向左的构建方式高效地利用了字符的最近出现位置
*
* 3. 动态规划状态设计:
*      - f(i, j) 的定义非常巧妙，专注于 s2 的前缀和目标长度
*      - 这种设计将状态数从 O(n*m) 减少到 O(m^2)
*      - 使用记忆化搜索避免重复计算

```

```
*  
* 4. 贪心选择策略:  
*   - 在状态转移中，我们总是选择所需 s1 前缀最短的方案  
*   - 这确保了后续状态有更多的选择空间  
*   - 体现了“贪心地选择更优的中间状态”的思想  
*  
* 5. 边界条件处理:  
*   - 正确处理  $i < j$  和  $j = 0$  的特殊情况  
*   - 使用 NA（无穷大）表示不可行的状态  
*   - 在主函数中对空字符串进行了特殊处理  
*/
```

```
/**  
 * 类似题目与训练拓展:  
 *  
 * 1. LeetCode 1143 – Longest Common Subsequence  
 *   链接: https://leetcode.cn/problems/longest-common-subsequence/  
 *   区别: 标准 LCS 问题，没有长度限制  
 *   算法: 动态规划  
 *  
 * 2. LeetCode 583 – Delete Operation for Two Strings  
 *   链接: https://leetcode.cn/problems/delete-operation-for-two-strings/  
 *   区别: 求最少删除次数，等价于求 LCS  
 *   算法: 动态规划  
 *  
 * 3. LeetCode 712 – Minimum ASCII Delete Sum for Two Strings  
 *   链接: https://leetcode.cn/problems/minimum-ascii-delete-sum-for-two-strings/  
 *   区别: 求最小 ASCII 删除和  
 *   算法: 动态规划  
 *  
 * 4. LeetCode 1035 – Uncrossed Lines  
 *   链接: https://leetcode.cn/problems/uncrossed-lines/  
 *   区别: 求不相交的线的最大数量，本质也是 LCS 问题  
 *   算法: 动态规划  
 *  
 * 5. LeetCode 516 – Longest Palindromic Subsequence  
 *   链接: https://leetcode.cn/problems/longest-palindromic-subsequence/  
 *   区别: 求最长回文子序列  
 *   算法: 区间动态规划  
 *  
 * 6. 牛客网 NC127 – 最长公共子串  
 *   链接: https://www.nowcoder.com/practice/f33f5adc55f444baa0e0ca87ad8a6aac  
 *   区别: 求最长公共子串（连续）而非子序列
```

```

* 算法：动态规划或滑动窗口
*/
// 静态成员变量定义
const int AddLimitLcs::NA;

// 主函数，用于测试
int main() {
    AddLimitLcs solution;

    // 测试用例
    string s1 = "abcde";
    string s2 = "ace";
    cout << "s1 = " << s1 << ", s2 = " << s2 << endl;
    cout << "最长公共子序列长度：" << solution.lcs(s1, s2) << endl; // 预期输出：3

    // 更多测试用例
    s1 = "abc";
    s2 = "def";
    cout << "\ns1 = " << s1 << ", s2 = " << s2 << endl;
    cout << "最长公共子序列长度：" << solution.lcs(s1, s2) << endl; // 预期输出：0

    s1 = "abc";
    s2 = "abc";
    cout << "\ns1 = " << s1 << ", s2 = " << s2 << endl;
    cout << "最长公共子序列长度：" << solution.lcs(s1, s2) << endl; // 预期输出：3

    // 验证算法正确性
    if (solution.verify("abcde", "ace")) {
        cout << "\n算法验证通过!" << endl;
    } else {
        cout << "\n算法验证失败!" << endl;
    }

    return 0;
}

```

=====

文件：Code03_AddLimitLcs.java

=====

```
package class128;
```

```
import java.util.Arrays;

/**
 * 增加限制的最长公共子序列问题
 *
 * 问题描述:
 * 给定两个字符串 s1 和 s2, s1 长度为 n, s2 长度为 m
 * 返回 s1 和 s2 的最长公共子序列长度
 *
 * 约束条件:
 * - 两个字符串都只由小写字母组成
 * -  $1 \leq n \leq 10^6$ 
 * -  $1 \leq m \leq 10^3$ 
 *
 * 优化背景:
 * 标准的 LCS 算法时间复杂度为  $O(n*m)$ , 当 n 达到  $10^6$  而 m 为  $10^3$  时,
 * 直接使用标准算法会导致大约  $10^9$  次操作, 显然不可行。
 * 因此需要利用题目中的限制条件进行优化。
 *
 * 优化思路:
 * 1. 观察到 s2 的长度 m 远小于 s1 的长度 n
 * 2. 预处理 s1 字符串, 记录每个位置之后每个字符首次出现的位置
 * 3. 定义新的 DP 状态: f(i, j) 表示 s2 的前 i 个字符要形成长度为 j 的公共子序列
 * 所需的 s1 的最短前缀长度
 * 4. 通过状态转移找到最大的 j, 使得存在  $i \leq m$  且  $f(i, j) \leq n$ 
 *
 * 时间复杂度分析:
 * - 预处理 s1:  $O(n * 26) = O(n)$ , 因为每个字符需要 26 个小写字母的处理
 * - DP 状态数:  $O(m^2)$ , 因为 i 和 j 的范围都是 0 到 m
 * - 总时间复杂度:  $O(n + m^2)$ 
 *
 * 空间复杂度分析:
 * - next 数组:  $O(n * 26) = O(n)$ 
 * - dp 数组:  $O(m^2)$ 
 * - 总空间复杂度:  $O(n + m^2)$ 
 *
 * 输入输出示例:
 * 输入:
 * s1 = "abcde"
 * s2 = "ace"
 * 输出: 3
 * 解释: 最长公共子序列是"ace", 长度为 3
 */
```

```
public class Code03_AddLimitLcs {  
    // 常量定义  
    private static final int NA = Integer.MAX_VALUE; // 表示不可行的情况  
    private static final int ALPHABET_SIZE = 26; // 字母表大小  
  
    // 实例变量，提高线程安全性  
    private char[] s1; // s1 的字符数组  
    private char[] s2; // s2 的字符数组  
    private int n, m; // s1 和 s2 的长度  
    private int[][] next; // next[i][c]表示 s1 中位置 i 之后字符 c 首次出现的位置  
    private int[][] dp; // 动态规划表  
  
    /**  
     * 主函数 - 用于测试两种算法的正确性和性能  
     */  
    public static void main(String[] args) {  
        // 验证正确性  
        verifyCorrectness();  
  
        // 性能测试  
        benchmarkPerformance();  
    }  
  
    /**  
     * 验证算法正确性  
     */  
    private static void verifyCorrectness() {  
        System.out.println("功能测试开始");  
        int n = 100;  
        int m = 100;  
        int testTime = 10000;  
  
        // 创建实例，避免使用静态成员变量  
        Code03_AddLimitLcs solution = new Code03_AddLimitLcs();  
  
        for (int i = 0; i < testTime; i++) {  
            int size1 = (int) (Math.random() * n) + 1;  
            int size2 = (int) (Math.random() * m) + 1;  
            String str1 = randomString(size1);  
            String str2 = randomString(size2);  
            int ans1 = solution.lcsClassic(str1, str2);  
            int ans2 = solution.lcsOptimized(str1, str2);  
        }  
    }  
}
```

```
    if (ans1 != ans2) {
        System.out.println("出错了!");
        System.out.println("str1: " + str1);
        System.out.println("str2: " + str2);
        System.out.println("经典算法结果: " + ans1);
        System.out.println("优化算法结果: " + ans2);
        return;
    }
}

System.out.println("功能测试通过");
System.out.println();
}

/***
 * 性能测试
 */
private static void benchmarkPerformance() {
    System.out.println("性能测试开始");
    int n = 100000; // 注意: 实际测试时可以调整为 1000000
    int m = 1000;
    System.out.println("n = " + n);
    System.out.println("m = " + m);

    // 创建实例
    Code03_AddLimitLcs solution = new Code03_AddLimitLcs();

    String str1 = randomString(n);
    String str2 = randomString(m);

    // 只在小数据上测试经典算法
    if (n <= 10000) {
        long start = System.currentTimeMillis();
        int ans1 = solution.lcsClassic(str1, str2);
        long end = System.currentTimeMillis();
        System.out.println("经典算法运行时间 : " + (end - start) + " 毫秒");
        System.out.println("经典算法结果: " + ans1);
    } else {
        System.out.println("经典算法对于大 n 会超时, 跳过测试");
    }

    // 测试优化算法
    long start = System.currentTimeMillis();
    int ans2 = solution.lcsOptimized(str1, str2);
}
```

```

        long end = System.currentTimeMillis();
        System.out.println("优化算法运行时间 : " + (end - start) + " 毫秒");
        System.out.println("优化算法结果: " + ans2);
        System.out.println("性能测试结束");
    }

/***
 * 随机生成指定长度的小写字母字符串
 *
 * @param n 字符串长度
 * @return 生成的随机字符串
 */
private static String randomString(int n) {
    if (n <= 0) {
        return "";
    }

    char[] ans = new char[n];
    for (int i = 0; i < n; i++) {
        ans[i] = (char) ((int) (Math.random() * ALPHABET_SIZE) + 'a');
    }
    return new String(ans);
}

/***
 * 经典动态规划版本的最长公共子序列算法
 * 时间复杂度: O(n*m)，不适用于 n 很大的情况
 *
 * @param str1 第一个字符串
 * @param str2 第二个字符串
 * @return 最长公共子序列的长度
 * @throws IllegalArgumentException 如果输入字符串为 null
 */
public int lcsClassic(String str1, String str2) {
    validateInputs(str1, str2);

    this.s1 = str1.toCharArray();
    this.s2 = str2.toCharArray();
    this.n = s1.length;
    this.m = s2.length;

    // 边界情况优化
    if (n == 0 || m == 0) {

```

```

        return 0;
    }

// 空间优化：只使用两行 dp 数组
int[][] dp = new int[2][m + 1];

for (int i = 1; i <= n; i++) {
    int row = i % 2;
    int prevRow = (i - 1) % 2;

    for (int j = 1; j <= m; j++) {
        if (s1[i - 1] == s2[j - 1]) {
            dp[row][j] = 1 + dp[prevRow][j - 1];
        } else {
            dp[row][j] = Math.max(dp[prevRow][j], dp[row][j - 1]);
        }
    }
}

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

/***
 * 优化版本的 LCS 算法主函数
 * 利用 s2 较短的特点进行优化
 * 时间复杂度: O(n + m^2)
 *
 * @param str1 第一个字符串（可能很长）
 * @param str2 第二个字符串（相对较短）
 * @return 最长公共子序列的长度
 * @throws IllegalArgumentException 如果输入字符串为 null
 */
public int lcsOptimized(String str1, String str2) {
    validateInputs(str1, str2);

    this.s1 = str1.toCharArray();
    this.s2 = str2.toCharArray();
    this.n = s1.length;
    this.m = s2.length;

    // 边界情况处理
    if (n == 0 || m == 0) {
        return 0;
    }
}

```

```

}

// 内存优化：动态分配 next 数组大小，避免静态大数组占用内存
this.next = new int[n + 1][ALPHABET_SIZE];
this.dp = new int[m + 1][m + 1];

// 构建预处理数据结构
build();

// 寻找最大的 j，使得 f(m, j) <= n
int ans = 0;
for (int j = m; j >= 1; j--) {
    if (f(m, j) != NA) {
        ans = j;
        break;
    }
}

return ans;
}

/***
 * 验证输入字符串的有效性
 *
 * @param str1 第一个字符串
 * @param str2 第二个字符串
 * @throws IllegalArgumentException 如果输入字符串为 null
 */
private void validateInputs(String str1, String str2) {
    if (str1 == null || str2 == null) {
        throw new IllegalArgumentException("输入字符串不能为 null");
    }
}

/***
 * 构建预处理数据结构
 * 1. next 数组：next[i][c] 表示 s1 中位置 i 之后字符 c 首次出现的位置
 * 2. 初始化 dp 数组为 -1 (表示未计算)
 */
private void build() {
    // 初始化 right 数组，记录每个字符最右边出现的位置
    int[] right = new int[ALPHABET_SIZE];
    Arrays.fill(right, NA);
}

```

```

// 从右向左遍历 s1, 构建 next 数组
for (int i = n; i >= 0; i--) {
    // 复制当前的 right 数组到 next[i]
    System.arraycopy(right, 0, next[i], 0, ALPHABET_SIZE);

    // 更新 right 数组, 如果 i > 0
    if (i > 0) {
        // s1 的 i 长度, 对应的字符是 s1[i-1]
        right[s1[i - 1] - 'a'] = i;
    }
}

// 初始化 dp 数组为-1
for (int i = 0; i <= m; i++) {
    Arrays.fill(dp[i], -1);
}
}

/**
 * 核心动态规划函数
 * 定义: f(i, j) 表示用 s2 的前 i 个字符形成长度为 j 的公共子序列
 *       所需的 s1 的最短前缀长度
 *
 * @param i s2 前缀的长度
 * @param j 目标公共子序列的长度
 * @return 所需的 s1 最短前缀长度, 如果不可行返回 NA
 */
private int f(int i, int j) {
    // 基本情况:
    // 1. 如果 i < j, 不可能形成长度为 j 的公共子序列 (因为 s2 只有 i 个字符)
    if (i < j) {
        return NA;
    }

    // 2. 如果 j == 0, 不需要任何 s1 字符
    if (j == 0) {
        return 0;
    }

    // 3. 如果已经计算过, 直接返回
    if (dp[i][j] != -1) {
        return dp[i][j];
    }
}

```

```

// 策略 1: 不使用 s2 的第 i 个字符 (即 s2[i-1])
// 此时结果为 f(i-1, j)
int ans = f(i - 1, j);

// 策略 2: 使用 s2 的第 i 个字符 (即 s2[i-1])
// 我们需要先找到用 s2 的前 i-1 个字符形成长度为 j-1 的公共子序列所需的最短 s1 前缀长度 pre
// 然后在 s1 的 pre 位置之后找到第一个等于 s2[i-1] 的字符的位置
int pre = f(i - 1, j - 1);
if (pre != NA && pre <= n) { // 添加 pre <= n 的检查, 确保 pre 在有效范围内
    int charIndex = s2[i - 1] - 'a';
    if (next[pre][charIndex] != NA) {
        ans = Math.min(ans, next[pre][charIndex]);
    }
}

// 记忆结果并返回
dp[i][j] = ans;
return ans;
}

/***
 * 获取算法的最大空间复杂度估计 (字节)
 * 用于性能监控和内存使用分析
 *
 * @param str1 第一个字符串
 * @param str2 第二个字符串
 * @return 预估的内存使用量 (字节)
 */
public long estimateMemoryUsage(String str1, String str2) {
    validateInputs(str1, str2);

    int n = str1.length();
    int m = str2.length();

    // next 数组的内存使用: (n+1) * 26 * 4 字节 (假设 int 为 4 字节)
    long nextMemory = (long)(n + 1) * 26 * 4;

    // dp 数组的内存使用: (m+1) * (m+1) * 4 字节
    long dpMemory = (long)(m + 1) * (m + 1) * 4;

    // 字符数组的内存使用
    long charArraysMemory = (n + m) * 2; // char 为 2 字节
}

```

```
        return nextMemory + dpMemory + charArraysMemory;
    }

/***
 * Java 工程化实战建议:
 *
 * 1. 内存管理优化:
 *     - 对于 next 数组, 当 n 很大时可能占用较多内存
 *     - 可以考虑使用更紧凑的数据结构或按需构建
 *     - 对于多次调用, 可以考虑复用部分数据结构
 *
 * 2. 并发安全性:
 *     - 当前实现使用了静态成员变量, 不是线程安全的
 *     - 在多线程环境下, 应该将这些变量作为方法内的局部变量或实例变量
 *     - 或者使用 ThreadLocal 来保证线程安全
 *
 * 3. 异常处理:
 *     - 应该添加输入验证, 确保输入字符串不为 null
 *     - 对于极端情况(如空字符串)需要特殊处理
 *
 * 4. 性能调优:
 *     - 对于非常大的 n, 可以考虑分批处理 s1 字符串
 *     - 使用预分配内存避免动态扩容开销
 *     - 对于频繁调用的场景, 可以缓存预处理结果
 *
 * 5. 代码风格优化:
 *     - 将常量和配置参数外部化
 *     - 考虑使用对象封装而不是静态方法
 *     - 为复杂的算法步骤添加更详细的注释
 */

/***
 * 算法优化的核心思想:
 *
 * 1. 问题转化:
 *     - 传统 LCS 问题关注长度, 这里我们转化为关注所需的 s1 前缀长度
 *     - 这种转化使我们能够利用 s2 长度较小的特点
 *
 * 2. 预处理技巧:
 *     - next 数组预处理让我们能在 O(1) 时间内找到字符在 s1 中特定位置之后的下一次出现
 *     - 这避免了在每次查找时遍历 s1
 *
 * 3. 动态规划状态设计:
```

```

*   - f(i, j) 的定义非常巧妙，专注于 s2 的前缀和目标长度
*   - 这种设计将时间复杂度从 O(n*m) 降低到 O(m^2)
*
* 4. 边界条件处理：
*   - 正确处理 i < j 和 j = 0 的情况
*   - 使用 NA（无穷大）表示不可行的状态
*
* 5. 贪心选择：
*   - 在状态转移中，我们总是选择所需 s1 前缀最短的方案
*   - 这确保了后续状态有更多的选择空间
*/
}

```

文件：Code03_AddLimitLcs.py

```

import sys
import time
from typing import List, Optional

class AddLimitLcs:
    """
    增加限制的最长公共子序列问题

```

问题描述：

给定两个字符串 s1 和 s2，s1 长度为 n，s2 长度为 m
返回 s1 和 s2 的最长公共子序列长度

约束条件：

- 两个字符串都只由小写字母组成
- $1 \leq n \leq 10^6$
- $1 \leq m \leq 10^3$

优化背景：

标准的 LCS 算法时间复杂度为 $O(n*m)$ ，当 n 达到 10^6 而 m 为 10^3 时，
直接使用标准算法会导致大约 10^9 次操作，显然不可行。
因此需要利用题目中的限制条件进行优化。

优化思路：

1. 观察到 s2 的长度 m 远小于 s1 的长度 n
2. 预处理 s1 字符串，记录每个位置之后每个字符首次出现的位置
3. 定义新的 DP 状态：f(i, j) 表示 s2 的前 i 个字符要形成长度为 j 的公共子序列

所需的 s1 的最短前缀长度

4. 通过状态转移找到最大的 j, 使得存在 $i \leq m$ 且 $f(i, j) \leq n$

时间复杂度分析:

- 预处理 s1: $O(n * 26) = O(n)$, 因为每个字符需要 26 个小写字母的处理
- DP 状态数: $O(m^2)$, 因为 i 和 j 的范围都是 0 到 m
- 总时间复杂度: $O(n + m^2)$

空间复杂度分析:

- next 数组: $O(n * 26) = O(n)$
- dp 数组: $O(m^2)$
- 总空间复杂度: $O(n + m^2)$

输入输出示例:

输入:

```
s1 = "abcde"
```

```
s2 = "ace"
```

输出: 3

解释: 最长公共子序列是"ace", 长度为 3

```
"""
```

```
def __init__(self):  
    """  
        初始化 AddLimitLcs 类的实例  
    """  
  
    self.NA = float('inf') # 表示不可行的情况  
    self.s1 = "" # 第一个字符串 (可能很长)  
    self.s2 = "" # 第二个字符串 (相对较短)  
    self.n = 0 # s1 的长度  
    self.m = 0 # s2 的长度  
    self.next = [] # next[i][c] 表示 s1 中位置 i 之后字符 c 首次出现的位置  
    self.dp = [] # 动态规划表
```

```
def build(self):
```

```
    """  
        构建预处理数据结构  
    """  
  
    1. next 数组: next[i][c] 表示 s1 中位置 i 之后字符 c 首次出现的位置  
    2. 初始化 dp 数组为 -1 (表示未计算)  
    """
```

```
# 初始化 next 数组
```

```
    self.next = [[self.NA] * 26 for _ in range(self.n + 1)]  
    right = [self.NA] * 26
```

```

# 从右向左遍历 s1，构建 next 数组
for i in range(self.n, 0, -1):
    # 复制当前的 right 数组到 next[i]
    for j in range(26):
        self.next[i][j] = right[j]
    # 更新 right 数组
    right[ord(self.s1[i - 1]) - ord('a')] = i

# 处理 i=0 的情况
for j in range(26):
    self.next[0][j] = right[j]

# 初始化 dp 数组
self.dp = [[-1] * (self.m + 1) for _ in range(self.m + 1)]

def f(self, i: int, j: int) -> int:
    """
核心动态规划函数
定义: f(i, j) 表示用 s2 的前 i 个字符形成长度为 j 的公共子序列
所需的 s1 的最短前缀长度

Args:
    i: s2 前缀的长度
    j: 目标公共子序列的长度

Returns:
    所需的 s1 最短前缀长度, 如果不可行返回 NA
    """
    # 基本情况:
    # 1. 如果 i < j, 不可能形成长度为 j 的公共子序列 (因为 s2 只有 i 个字符)
    if i < j:
        return self.NA
    # 2. 如果 j == 0, 不需要任何 s1 字符
    if j == 0:
        return 0
    # 3. 如果已经计算过, 直接返回
    if self.dp[i][j] != -1:
        return self.dp[i][j]

    # 策略 1: 不使用 s2 的第 i 个字符 (即 s2[i-1])
    # 此时结果为 f(i-1, j)
    ans = self.f(i - 1, j)

    for c in range(26):
        if self.s1[i - 1] == chr(c + ord('a')):
            ans = min(ans, f(i - 1, j - 1))

    self.dp[i][j] = ans
    return ans

```

```

# 策略 2: 使用 s2 的第 i 个字符 (即 s2[i-1])
# 我们需要先找到用 s2 的前 i-1 个字符形成长度为 j-1 的公共子序列所需的最短 s1 前缀长度 pre
# 然后在 s1 的 pre 位置之后找到第一个等于 s2[i-1] 的字符的位置
pre = self.f(i - 1, j - 1)
if pre != self.NA:
    # 获取 s2 第 i 个字符的 ASCII 码索引
    char_index = ord(self.s2[i - 1]) - ord('a')
    if self.next[pre][char_index] != self.NA:
        ans = min(ans, self.next[pre][char_index])

# 记忆结果并返回
self.dp[i][j] = ans
return ans

```

```

def lcs_classic(self) -> int:
"""
经典动态规划版本的最长公共子序列算法
时间复杂度: O(n*m)，不适用于 n 很大的情况
仅用于验证优化算法的正确性

```

Returns:

最长公共子序列的长度

"""

```

# 创建二维 DP 数组
dp = [[0] * (self.m + 1) for _ in range(self.n + 1)]

for i in range(1, self.n + 1):
    for j in range(1, self.m + 1):
        if self.s1[i - 1] == self.s2[j - 1]:
            dp[i][j] = 1 + dp[i - 1][j - 1]
        else:
            dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])

return dp[self.n][self.m]

```

```

def lcs(self, str1: str, str2: str) -> int:
"""

```

优化版本的 LCS 算法主函数

利用 s2 较短的特点进行优化

Args:

str1: 第一个字符串 (可能很长)
 str2: 第二个字符串 (相对较短)

Returns:

最长公共子序列的长度

"""

初始化输入数据

self.s1 = str1

self.s2 = str2

self.n = len(str1)

self.m = len(str2)

边界情况处理

if self.n == 0 or self.m == 0:

 return 0

构建预处理数据结构

self.build()

寻找最大的 j, 使得 f(m, j) <= n

ans = 0

for j in range(self.m, 0, -1):

 if self.f(self.m, j) != self.NA:

 ans = j

 break

return ans

def verify(self, str1: str, str2: str) -> bool:

"""

验证函数：同时使用经典算法和优化算法，并比较结果

Args:

str1: 第一个字符串

str2: 第二个字符串

Returns:

验证是否通过

"""

self.s1 = str1

self.s2 = str2

self.n = len(str1)

self.m = len(str2)

对于大的 n, 不进行经典算法验证，避免超时

```
if self.n > 10000:  
    return True  
  
classic_result = self.lcs_classic()  
optimized_result = self.lcs(str1, str2)  
  
return classic_result == optimized_result
```

```
def benchmark(self, str1: str, str2: str, iterations: int = 10) -> dict:  
    """
```

性能基准测试函数，用于比较经典算法和优化算法的性能

Args:

```
    str1: 第一个字符串  
    str2: 第二个字符串  
    iterations: 测试迭代次数
```

Returns:

包含性能指标的字典

```
"""
```

```
self.s1 = str1  
self.s2 = str2  
self.n = len(str1)  
self.m = len(str2)
```

```
results = {}
```

只在小数据上测试经典算法

```
if self.n <= 10000:  
    # 测量经典算法的时间  
    start_time = time.time()  
    for _ in range(iterations):  
        classic_result = self.lcs_classic()  
        end_time = time.time()  
        results['classic_time'] = (end_time - start_time) / iterations  
        results['classic_result'] = classic_result
```

测量优化算法的时间

```
start_time = time.time()  
for _ in range(iterations):  
    optimized_result = self.lcs(str1, str2)  
    end_time = time.time()  
    results['optimized_time'] = (end_time - start_time) / iterations
```

```
    results['optimized_result'] = optimized_result

    return results

# Python 特有的工程化建议和优化技巧:
#
# 1. 内存优化策略:
#   - 在 Python 中, 对于大型列表, 可以使用数组模块(array)代替列表(list)以节省内存
#   - 当 n 非常大时, 可以考虑使用 numpy 数组进行向量化操作
#   - 对于 next 数组, 可以使用稀疏矩阵表示法, 特别是当字符集较大但实际使用的字符较少时
#   - 使用生成器表达式代替列表推导式, 减少内存占用
#
# 2. 性能优化技巧:
#   - 使用 lru_cache 装饰器进行记忆化搜索 (但在本问题中我们手动实现了记忆化)
#   - 对于频繁访问的属性, 使用局部变量缓存
#   - 使用内置函数如 min()、max(), 它们是用 C 实现的, 速度更快
#   - 避免在循环中使用不必要的函数调用和对象创建
#   - 使用字典推导式和列表推导式代替显式循环
#
# 3. Python 特性利用:
#   - 使用 type hints 提高代码可读性和可维护性
#   - 使用 docstrings 提供详细的文档
#   - 利用 Python 的异常处理机制进行边界情况处理
#   - 使用上下文管理器(with 语句)管理资源
#   - 利用 Python 的标准库如 collections 中的数据结构优化算法
#
# 4. 并行计算:
#   - 对于独立的子问题, 可以使用 multiprocessing 模块进行并行计算
#   - 使用 concurrent.futures 库简化异步任务处理
#   - 考虑使用 numba 或 cython 等工具将关键路径代码编译为机器码
#
# 5. 代码风格与规范:
#   - 遵循 PEP 8 规范编写代码
#   - 使用描述性的变量名和函数名
#   - 模块化设计, 将功能划分为小的、可测试的函数
#   - 添加适当的注释和文档
#   - 使用单元测试确保代码质量
#
# 6. 特殊优化考虑:
#   - 在 Python 中, 递归深度有限制(默认 1000), 因此在递归实现时需要注意
#   - 对于非常大的 n, 考虑使用迭代方式实现算法, 避免栈溢出
#   - 使用 sys.getsizeof() 检查对象大小, 优化内存使用
#   - 避免在循环中使用字符串拼接, 使用列表收集后 join
```

```
# 算法优化的核心思想:  
#  
# 1. 问题转化的艺术:  
#   - 传统 LCS 问题关注长度，这里转化为关注所需的 s1 前缀长度  
#   - 这种转化使我们能够利用 s2 长度较小的特点  
#   - 时间复杂度从 O(n*m) 降低到 O(n + m^2)  
#  
# 2. 预处理技巧:  
#   - next 数组预处理让我们能在 O(1) 时间内找到字符在 s1 中特定位置之后的下一次出现  
#   - 这避免了在每次查找时遍历 s1  
#   - 从右向左的构建方式高效地利用了字符的最近出现位置  
#  
# 3. 动态规划状态设计:  
#   - f(i, j) 的定义非常巧妙，专注于 s2 的前缀和目标长度  
#   - 这种设计将状态数从 O(n*m) 减少到 O(m^2)  
#   - 使用记忆化搜索避免重复计算  
#  
# 4. 贪心选择策略:  
#   - 在状态转移中，我们总是选择所需 s1 前缀最短的方案  
#   - 这确保了后续状态有更多的选择空间  
#   - 体现了“贪心地选择更优的中间状态”的思想  
#  
# 5. 边界条件处理:  
#   - 正确处理 i < j 和 j = 0 的特殊情况  
#   - 使用 NA（无穷大）表示不可行的状态  
#   - 在主函数中对空字符串进行了特殊处理  
  
# 类似题目与训练拓展:  
#  
# 1. LeetCode 1143 – Longest Common Subsequence  
#   链接: https://leetcode.cn/problems/longest-common-subsequence/  
#   区别: 标准 LCS 问题，没有长度限制  
#   算法: 动态规划  
#  
# 2. LeetCode 583 – Delete Operation for Two Strings  
#   链接: https://leetcode.cn/problems/delete-operation-for-two-strings/  
#   区别: 求最少删除次数，等价于求 LCS  
#   算法: 动态规划  
#  
# 3. LeetCode 712 – Minimum ASCII Delete Sum for Two Strings  
#   链接: https://leetcode.cn/problems/minimum-ascii-delete-sum-for-two-strings/  
#   区别: 求最小 ASCII 删除和
```

```
# 算法: 动态规划
#
# 4. LeetCode 1035 - Uncrossed Lines
# 链接: https://leetcode.cn/problems/uncrossed-lines/
# 区别: 求不相交的线的最大数量, 本质也是 LCS 问题
# 算法: 动态规划
#
# 5. LeetCode 516 - Longest Palindromic Subsequence
# 链接: https://leetcode.cn/problems/longest-palindromic-subsequence/
# 区别: 求最长回文子序列
# 算法: 区间动态规划
#
# 6. 牛客网 NC127 - 最长公共子串
# 链接: https://www.nowcoder.com/practice/f33f5adc55f444baa0e0ca87ad8a6aac
# 区别: 求最长公共子串(连续)而非子序列
# 算法: 动态规划或滑动窗口

# 主函数, 用于测试
def main():
    solution = AddLimitLcs()

    # 测试用例
    s1 = "abcde"
    s2 = "ace"
    print(f"s1 = {s1}, s2 = {s2}")
    print(f"最长公共子序列长度: {solution.lcs(s1, s2)}")  # 预期输出: 3

    # 更多测试用例
    s1 = "abc"
    s2 = "def"
    print(f"\ns1 = {s1}, s2 = {s2}")
    print(f"最长公共子序列长度: {solution.lcs(s1, s2)}")  # 预期输出: 0

    s1 = "abc"
    s2 = "abc"
    print(f"\ns1 = {s1}, s2 = {s2}")
    print(f"最长公共子序列长度: {solution.lcs(s1, s2)}")  # 预期输出: 3

    # 验证算法正确性
    if solution.verify("abcde", "ace"):
        print("\n算法验证通过!")
    else:
        print("\n算法验证失败!")
```

```

# 性能测试
s1_medium = "abcdefghijkl" * 100 # 长度为 1000 的字符串
s2_medium = "acegikmoqsu" * 10 # 长度为 200 的字符串
print(f"\n性能测试 - s1 长度: {len(s1_medium)}, s2 长度: {len(s2_medium)}")
results = solution.benchmark(s1_medium, s2_medium, iterations=3)

if 'classic_time' in results:
    print(f"经典算法平均执行时间: {results['classic_time']:.6f} 秒")
    print(f"经典算法结果: {results['classic_result']}")

print(f"优化算法平均执行时间: {results['optimized_time']:.6f} 秒")
print(f"优化算法结果: {results['optimized_result']}")

if __name__ == "__main__":
    main()

```

文件: Code04_EggDrop.cpp

```

#include <iostream>
#include <vector>
#include <climits>
#include <algorithm>
#include <cstdio>
#include <ctime>

/***
 * 超级鸡蛋掉落问题 (Super Egg Drop)
 *
 * 问题描述:
 * 假设你有 k 个鸡蛋，并且可以使用一栋从 1 到 n 层的大楼。
 * 已知存在某个楼层 f (0 <= f <= n)，从 f 楼及以下楼层抛出的鸡蛋不会碎，
 * 从 f 楼以上的楼层抛出的鸡蛋会碎。
 * 当鸡蛋被摔碎后，它就不能再使用了。
 * 请确定最少需要多少次尝试，才能保证在最坏情况下找出确切的 f 值。
 *
 * 约束条件:
 * - 1 <= k <= 100
 * - 1 <= n <= 10^4
 *
 * 算法思路:

```

```

* 这个问题采用了优化的动态规划状态定义:
* dp[i][j] 表示使用 i 个鸡蛋, 尝试 j 次, 最多能确定的楼层数。
* 我们需要找到最小的 j, 使得 dp[k][j] >= n。
*
* 状态转移方程:
* dp[i][j] = dp[i-1][j-1] + dp[i][j-1] + 1
* 其中:
* - dp[i-1][j-1] 表示鸡蛋碎了的情况, 用 i-1 个鸡蛋在 j-1 次尝试中能确定的楼层数
* - dp[i][j-1] 表示鸡蛋没碎的情况, 用 i 个鸡蛋在 j-1 次尝试中能确定的楼层数
* - +1 表示当前测试的楼层
*/
class Solution {
private:
    // 验证输入参数的有效性
    void validateInputs(int k, int n) {
        if (k < 1 || k > 100) {
            throw std::invalid_argument("鸡蛋数量必须在 1 到 100 之间");
        }
        if (n < 1 || n > 10000) {
            throw std::invalid_argument("楼层数量必须在 1 到 10000 之间");
        }
    }

public:
    /**
     * 解法 1: 二维 dp 数组实现
     * 时间复杂度: O(k*n), 但在实际执行中会早退出
     * 空间复杂度: O(k*n)
     */
    int superEggDrop1(int k, int n) {
        // 输入验证
        validateInputs(k, n);

        // 边界情况: 如果只有 1 个鸡蛋, 必须从 1 楼开始逐层测试
        if (k == 1) {
            return n;
        }

        // 创建 dp 数组
        std::vector<std::vector<int>> dp(k + 1, std::vector<int>(n + 1, 0));

        // j 表示尝试次数, 从 1 开始递增
        for (int j = 1; j <= n; ++j) {

```

```

// i 表示使用的鸡蛋数，从 1 开始递增
for (int i = 1; i <= k; ++i) {
    // 状态转移方程
    dp[i][j] = dp[i - 1][j - 1] + dp[i][j - 1] + 1;

    // 当可以确定的楼层数大于等于 n 时，返回当前的尝试次数 j
    if (dp[i][j] >= n) {
        return j;
    }
}

// 实际上不可能到达这里
return n;
}

/**
 * 解法 2：空间优化版本，使用一维 dp 数组
 * 时间复杂度：O(k*n)，但在实际执行中会早退出
 * 空间复杂度：O(k)
 */
int superEggDrop2(int k, int n) {
    // 输入验证
    validateInputs(k, n);

    // 边界情况：如果只有 1 个鸡蛋，必须从 1 楼开始逐层测试
    if (k == 1) {
        return n;
    }

    // 空间优化：使用一维 dp 数组
    std::vector<int> dp(k + 1, 0);

    // j 表示尝试次数，从 1 开始递增
    for (int j = 1; j <= n; ++j) {
        // 保存上一次的值，用于状态转移
        int previous = 0;

        // i 表示使用的鸡蛋数，从 1 开始递增
        for (int i = 1; i <= k; ++i) {
            // 暂存当前 dp[i] 的值，因为它将作为下一轮的 previous
            int temp = dp[i];

```

```

    // 状态转移
    dp[i] = dp[i] + previous + 1;

    // 更新 previous 为当前 dp[i] 的旧值
    previous = temp;

    // 当可以确定的楼层数大于等于 n 时，返回当前的尝试次数 j
    if (dp[i] >= n) {
        return j;
    }
}

// 实际上不可能到达这里
return n;
}

/**
 * 解法 3：二分搜索优化版本
 * 时间复杂度：O(k * log n)
 * 空间复杂度：O(k)
 */
int superEggDrop3(int k, int n) {
    // 输入验证
    validateInputs(k, n);

    // 边界情况处理
    if (k == 1) {
        return n;
    }

    // 计算最小需要多少次尝试才能覆盖 n 层楼
    int low = 1, high = n;
    while (low < high) {
        int mid = low + (high - low) / 2;
        if (computeFloors(k, mid) >= n) {
            high = mid;
        } else {
            low = mid + 1;
        }
    }

    return low;
}

```

```

}

private:
/***
 * 计算使用 k 个鸡蛋，尝试 m 次，最多能确定的楼层数
 */
int computeFloors(int k, int m) {
    // 使用动态规划计算最多能确定的楼层数
    std::vector<int> dp(k + 1, 0);

    for (int i = 1; i <= m; ++i) {
        int prev = 0;
        for (int j = 1; j <= k; ++j) {
            int temp = dp[j];
            dp[j] = dp[j] + prev + 1;
            prev = temp;

            // 提前终止，避免整数溢出
            if (dp[j] > 10000) {
                return 10000;
            }
        }
    }

    return dp[k];
}

};

// 主函数，用于测试不同解法
int main() {
    Solution solution;

    // 测试用例
    std::vector<std::pair<int, int>> testCases = {
        {1, 2},      // 预期输出: 2
        {2, 6},      // 预期输出: 3
        {3, 14},     // 预期输出: 4
        {2, 100},    // 预期输出: 14
        {100, 10000} // 预期输出: 24
    };

    std::cout << "测试不同解法的结果: " << std::endl;
    for (const auto& testCase : testCases) {

```

```

int k = testCase.first;
int n = testCase.second;

// 记录开始时间
clock_t start, end;

// 测试解法 1
start = clock();
int result1 = solution.superEggDrop1(k, n);
end = clock();
double time1 = static_cast<double>(end - start) / CLOCKS_PER_SEC * 1000; // 转换为毫秒

// 测试解法 2
start = clock();
int result2 = solution.superEggDrop2(k, n);
end = clock();
double time2 = static_cast<double>(end - start) / CLOCKS_PER_SEC * 1000;

// 测试解法 3
start = clock();
int result3 = solution.superEggDrop3(k, n);
end = clock();
double time3 = static_cast<double>(end - start) / CLOCKS_PER_SEC * 1000;

// 输出结果
printf("鸡蛋数: %d, 楼层数: %d\n", k, n);
printf("解法 1 结果: %d, 耗时: %.3fms\n", result1, time1);
printf("解法 2 结果: %d, 耗时: %.3fms\n", result2, time2);
printf("解法 3 结果: %d, 耗时: %.3fms\n\n", result3, time3);

}

return 0;
}

```

```

/**
 * C++工程化实战建议:
 *
 * 1. 异常处理:
 *     - 使用异常机制处理无效输入
 *     - 在生产环境中可以考虑使用错误码而不是异常
 *     - 添加 try-catch 块捕获可能的异常
 *
 * 2. 性能优化:

```

```
*      - 对于非常大的 n，优先使用二分搜索版本（解法 3）
*      - 对于内存受限的环境，使用空间优化版本（解法 2）
*      - 注意整数溢出问题，当 k 和 m 较大时，使用 long long 类型
*
* 3. 内存管理：
*      - 使用 vector 动态分配内存，避免手动管理内存
*      - 对于频繁调用的场景，可以考虑缓存 dp 数组
*      - 注意在递归或循环中避免创建过多临时对象
*
* 4. 代码可读性与维护性：
*      - 使用命名空间组织代码
*      - 可以将类定义和实现分离到 .h 和 .cpp 文件中
*      - 添加单元测试确保代码正确性
*
* 5. 并发安全：
*      - 当前实现是无状态的，可以安全地在多线程环境中使用
*      - 对于有状态的实现，需要添加互斥锁或使用线程局部存储
*/
```

```
/***
 * 算法优化思考：
 *
 * 1. 数学公式优化：
 *      可以证明，使用 k 个鸡蛋尝试 m 次，最多能确定的楼层数等于组合数之和：
 *      floor = C(m, 1) + C(m, 2) + ... + C(m, min(k, m))
 *      可以使用这个公式直接计算，避免动态规划的循环
 *
 * 2. 预处理优化：
 *      对于多次调用的场景，可以预处理所有可能的 k 和 n 的组合，
 *      构建一个查询表，实现 O(1) 时间复杂度的查询
 *
 * 3. 位运算优化：
 *      对于某些特殊情况，可以使用位运算加速计算
 *      例如当 k >= log2(n) 时，最优解就是 log2(n) 向上取整
 *
 * 4. 缓存优化：
 *      可以缓存 computeFloors 函数的结果，避免重复计算
 *      特别是在解法 3 中，这个函数可能被多次调用
*/
```

```
=====
package class128;

import java.util.Arrays;

/***
 * 超级鸡蛋掉落问题 (Super Egg Drop)
 *
 * 问题描述:
 * 假设你有 k 个鸡蛋，并且可以使用一栋从 1 到 n 层的大楼。
 * 已知存在某个楼层 f (0 <= f <= n)，从 f 楼及以下楼层抛出的鸡蛋不会碎，从 f 楼以上的楼层抛出的鸡蛋会碎。
 * 当鸡蛋被摔碎后，它就不能再使用了。
 * 请确定最少需要多少次尝试，才能保证在最坏情况下找出确切的 f 值。
 *
 * 约束条件:
 * - 1 <= k <= 100
 * - 1 <= n <= 10^4
 *
 * 问题分析:
 * 这个问题是一个经典的动态规划问题，但状态定义的选择对解题效率至关重要。
 *
 * 传统的状态定义方式:
 * dp[i][j] 表示使用 i 个鸡蛋，j 层楼时，最坏情况下所需的最少尝试次数。
 * 这种定义下，状态转移方程较为复杂，且时间复杂度较高。
 *
 * 优化的状态定义方式:
 * dp[i][j] 表示使用 i 个鸡蛋，尝试 j 次，最多能确定的楼层数。
 * 这种定义允许我们找到最小的 j，使得 dp[k][j] >= n。
 *
 * 状态转移方程:
 * dp[i][j] = dp[i-1][j-1] + dp[i][j-1] + 1
 * 解释:
 * - dp[i-1][j-1]: 如果在某层扔鸡蛋碎了，那么我们用 i-1 个鸡蛋在剩下的 j-1 次机会中
 *   最多能确定的下面楼层数
 * - dp[i][j-1]: 如果在某层扔鸡蛋没碎，那么我们用 i 个鸡蛋在剩下的 j-1 次机会中
 *   最多能确定的上面楼层数
 * - +1: 当前测试的楼层
 *
 * 时间复杂度分析:
 * - 对于方法 1: O(k*n)，但实际上在实际执行中会早退出，远小于这个值
 * - 对于方法 2: O(k*n)，同样会早退出，且空间复杂度更低
 *
```

```

* 空间复杂度分析:
* - 方法 1: O(k*n)
* - 方法 2: O(k), 通过滚动数组优化空间
*
* 输入输出示例:
* 输入: k = 1, n = 2
* 输出: 2
* 解释: 第一次扔在 1 楼, 碎了则 f=0, 没碎则扔在 2 楼, 碎了则 f=1, 没碎则 f=2
*
* 输入: k = 2, n = 6
* 输出: 3
* 解释: 使用最优策略, 最多需要 3 次尝试
*
* 输入: k = 3, n = 14
* 输出: 4
*
* 测试链接: https://leetcode.cn/problems/super-egg-drop/
*/
public class Code04_EggDrop {

    /**
     * 验证输入参数的有效性
     *
     * @param k 鸡蛋数量
     * @param n 楼层数量
     * @throws IllegalArgumentException 如果参数不满足约束条件
     */
    private static void validateInputs(int k, int n) {
        if (k < 1 || k > 100) {
            throw new IllegalArgumentException("鸡蛋数量必须在 1 到 100 之间");
        }
        if (n < 1 || n > 10000) {
            throw new IllegalArgumentException("楼层数量必须在 1 到 10000 之间");
        }
    }

    /**
     * 解法 1: 二维 dp 数组实现
     * dp[i][j] 表示使用 i 个鸡蛋, 尝试 j 次, 最多能确定的楼层数
     *
     * @param k 鸡蛋数量
     * @param n 楼层数量
     * @return 最坏情况下所需的最少尝试次数
     */
}

```

```

*/
public static int superEggDrop1(int k, int n) {
    // 输入验证
    validateInputs(k, n);

    // 边界情况：如果只有 1 个鸡蛋，必须从 1 楼开始逐层测试
    if (k == 1) {
        return n;
    }

    // 创建 dp 数组，dp[i][j] 表示使用 i 个鸡蛋，尝试 j 次最多能确定的层数
    int[][] dp = new int[k + 1][n + 1];

    // j 表示尝试次数，从 1 开始递增
    for (int j = 1; j <= n; j++) {
        // i 表示使用的鸡蛋数，从 1 开始递增
        for (int i = 1; i <= k; i++) {
            // 状态转移方程：
            // 1. dp[i-1][j-1]: 鸡蛋碎了的情况，用 i-1 个鸡蛋在 j-1 次尝试中能确定的层数
            // 2. dp[i][j-1]: 鸡蛋没碎的情况，用 i 个鸡蛋在 j-1 次尝试中能确定的层数
            // 3. +1: 当前测试的楼层
            dp[i][j] = dp[i - 1][j - 1] + dp[i][j - 1] + 1;

            // 当可以确定的层数大于等于 n 时，返回当前的尝试次数 j
            if (dp[i][j] >= n) {
                return j;
            }
        }
    }

    // 实际上不可能到达这里，因为当尝试次数为 n 时，至少可以用 1 个鸡蛋确定 n 层楼
    return n;
}

/**
 * 解法 2：空间优化版本，使用一维 dp 数组
 *
 * @param k 鸡蛋数量
 * @param n 楼层数量
 * @return 最坏情况下所需的最少尝试次数
 */
public static int superEggDrop2(int k, int n) {
    // 输入验证

```

```

validateInputs(k, n);

// 边界情况：如果只有 1 个鸡蛋，必须从 1 楼开始逐层测试
if (k == 1) {
    return n;
}

// 空间优化：使用一维 dp 数组，dp[i] 表示使用 i 个鸡蛋时能确定的楼层数
int[] dp = new int[k + 1];

// j 表示尝试次数，从 1 开始递增
for (int j = 1; j <= n; j++) {
    // 保存上一次的值，用于状态转移
    int previous = 0;

    // i 表示使用的鸡蛋数，从 1 开始递增
    for (int i = 1; i <= k; i++) {
        // 暂存当前 dp[i] 的值，因为它将作为下一轮的 previous
        int temp = dp[i];

        // 状态转移：dp[i] = previous(即上一轮的 dp[i-1]) + dp[i] + 1
        dp[i] = dp[i] + previous + 1;

        // 更新 previous 为当前 dp[i] 的旧值
        previous = temp;

        // 当可以确定的楼层数大于等于 n 时，返回当前的尝试次数 j
        if (dp[i] >= n) {
            return j;
        }
    }
}

// 实际上不可能到达这里
return n;
}

/**
 * 解法 3：二分搜索优化版本
 * 当鸡蛋数量较多（例如  $k > \log_2(n)$ ）时，二分搜索是最优策略
 *
 * @param k 鸡蛋数量
 * @param n 楼层数量

```

```

* @return 最坏情况下所需的最少尝试次数
*/
public static int superEggDrop3(int k, int n) {
    // 输入验证
    validateInputs(k, n);

    // 边界情况处理
    if (k == 1) {
        return n;
    }

    // 计算最小需要多少次尝试才能覆盖 n 层楼
    // 最小次数不会超过 log2(n) + 1
    int low = 1, high = n;
    while (low < high) {
        int mid = low + (high - low) / 2;
        if (computeFloors(k, mid) >= n) {
            high = mid;
        } else {
            low = mid + 1;
        }
    }

    return low;
}

/**
 * 计算使用 k 个鸡蛋，尝试 m 次，最多能确定的层数
 *
 * @param k 鸡蛋数量
 * @param m 尝试次数
 * @return 最多能确定的层数
 */
private static int computeFloors(int k, int m) {
    // 使用动态规划计算 dp[k][m]
    // 优化：只保留两行
    int[] dp = new int[k + 1];
    int result = 0;

    for (int i = 1; i <= m; i++) {
        int prev = 0;
        for (int j = 1; j <= k; j++) {
            int temp = dp[j];
            dp[j] = prev + 1;
            if (dp[j] >= m) {
                result = j;
            }
            prev = temp;
        }
    }

    return result;
}

```

```
        dp[j] = dp[j] + prev + 1;
        prev = temp;
        // 提前终止，避免整数溢出
        if (dp[j] > 10000) {
            return 10000;
        }
    }

    return dp[k];
}

/***
 * 主函数，用于测试不同解法
 */
public static void main(String[] args) {
    // 测试用例
    int[][] testCases = {
        {1, 2},      // 预期输出: 2
        {2, 6},      // 预期输出: 3
        {3, 14},     // 预期输出: 4
        {2, 100},    // 预期输出: 14
        {100, 10000} // 预期输出: 24
    };

    System.out.println("测试不同解法的结果: ");
    for (int[] testCase : testCases) {
        int k = testCase[0];
        int n = testCase[1];

        long start, end;

        // 测试解法 1
        start = System.currentTimeMillis();
        int result1 = superEggDrop1(k, n);
        end = System.currentTimeMillis();

        // 测试解法 2
        start = System.currentTimeMillis();
        int result2 = superEggDrop2(k, n);
        end = System.currentTimeMillis();

        // 测试解法 3
        start = System.currentTimeMillis();
        int result3 = superEggDrop3(k, n);
        end = System.currentTimeMillis();

        System.out.printf("k: %d, n: %d, result1: %d, result2: %d, result3: %d, time: %d ms\n",
                k, n, result1, result2, result3, (end - start));
    }
}
```

```
    start = System.currentTimeMillis();
    int result3 = superEggDrop3(k, n);
    end = System.currentTimeMillis();

    System.out.printf("鸡蛋数: %d, 楼层数: %d, 结果 1: %d, 结果 2: %d, 结果 3: %d\n",
                      k, n, result1, result2, result3);
}
```

```
}
```

```
/**
```

```
* Java 工程化实战建议:
```

```
*
```

```
* 1. 输入验证与错误处理:
```

- * - 添加参数验证，确保 k 和 n 在有效范围内
- * - 使用异常处理机制处理无效输入
- * - 对于边界情况（如 k=1 或 n=1）进行特殊优化

```
*
```

```
* 2. 性能优化:
```

- * - 对于非常大的 n，可以考虑使用二分搜索优化（解法 3）
- * - 注意整数溢出问题，当 m 较大时 dp 值可能会超出 int 范围
- * - 对于 k 较大的情况（例如 $k > \log_2(n)$ ），最优解是 $\log_2(n)$ ，可以提前返回

```
*
```

```
* 3. 空间优化:
```

- * - 使用滚动数组（解法 2）可以将空间复杂度从 $O(k*n)$ 降低到 $O(k)$
- * - 当 k 较大时，只需要使用 $O(k)$ 的空间，非常高效

```
*
```

```
* 4. 代码可读性与维护性:
```

- * - 使用清晰的变量命名和详细的注释
- * - 将核心逻辑抽取为独立的方法
- * - 添加单元测试验证算法正确性

```
*
```

```
* 5. 扩展性考虑:
```

- * - 可以扩展实现更多变体问题，如找到恰好摔碎鸡蛋的楼层等
- * - 考虑鸡蛋有一定的韧性，可以承受一定次数的掉落而不碎

```
*/
```

```
/**
```

```
* 算法本质与技巧总结:
```

```
*
```

```
* 1. 状态定义的转变:
```

- * - 传统定义：使用 i 个鸡蛋确定 j 层楼需要多少次尝试
- * - 优化定义：使用 i 个鸡蛋尝试 j 次最多能确定多少层楼
- * - 这种转变是解题的关键，大大简化了状态转移方程

```
*  
* 2. 动态规划的思想:  
*   - 将原问题分解为子问题: 鸡蛋碎或不碎的两种情况  
*   - 通过子问题的解构建原问题的解  
*   - 利用状态转移方程高效计算  
*  
* 3. 贪心策略的体现:  
*   - 每次尝试的最优楼层选择 (使得最坏情况的次数最少)  
*   - 最优策略是使得两种情况 (碎或不碎) 的尝试次数相等  
*  
* 4. 数学归纳与递推:  
*   - 状态转移方程体现了数学归纳法的思想  
*   - 每增加一次尝试, 可以确定的层数呈现组合数的增长规律  
*  
* 5. 优化技巧:  
*   - 空间优化: 使用滚动数组减少内存占用  
*   - 时间优化: 提前终止, 二分查找等  
*   - 边界情况处理: 针对特殊情况 (如 k=1) 的优化  
*/
```

```
/**  
* 类似题目与训练拓展:  
*  
* 1. LeetCode 1884 - 鸡蛋掉落-两枚鸡蛋  
*   链接: https://leetcode.cn/problems/egg-drop-with-2-eggs-and-n-floors/  
*   区别: 限定只有 2 个鸡蛋  
*   算法: 可以使用数学方法直接求解, 最优解为  $\sqrt{n}$  的上界  
*  
* 2. LeetCode 887 - 鸡蛋掉落 (与本题相同)  
*   链接: https://leetcode.cn/problems/super-egg-drop/  
*   算法: 动态规划, 状态定义优化  
*  
* 3. LeetCode 960 - 删除列以使之有序 III  
*   链接: https://leetcode.cn/problems/delete-columns-to-make-sorted-iii/  
*   区别: 不同的问题背景, 但使用类似的动态规划思想  
*   算法: 动态规划, 状态转移优化  
*  
* 4. 牛客网 NC130 - 鸡蛋的硬度  
*   链接: https://www.nowcoder.com/practice/3a3577b9d3294fb7845b96a9cd2e099c  
*   区别: 鸡蛋硬度问题, 与鸡蛋掉落问题类似  
*   算法: 动态规划  
*  
* 5. 面试题 08.11. 硬币
```

```
*     链接: https://leetcode.cn/problems/coin-lcci/
*     区别: 完全不同的问题, 但都是优化动态规划状态转移的例子
*     算法: 动态规划, 数学优化
*/
}
```

=====

文件: Code04_EggDrop.py

=====

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import time
from typing import List, Tuple

"""
超级鸡蛋掉落问题 (Super Egg Drop)

问题描述:
假设你有 k 个鸡蛋, 并且可以使用一栋从 1 到 n 层的大楼。
已知存在某个楼层 f ( $0 \leq f \leq n$ ), 从 f 楼及以下楼层抛出的鸡蛋不会碎,
从 f 楼以上的楼层抛出的鸡蛋会碎。
当鸡蛋被摔碎后, 它就不能再使用了。
请确定最少需要多少次尝试, 才能保证在最坏情况下找出确切的 f 值。
"""

约束条件:
-  $1 \leq k \leq 100$ 
-  $1 \leq n \leq 10^4$ 

算法思路:
这个问题采用了优化的动态规划状态定义:
 $dp[i][j]$  表示使用 i 个鸡蛋, 尝试 j 次, 最多能确定的楼层数。
我们需要找到最小的 j, 使得  $dp[k][j] \geq n$ 。
```

状态转移方程:

$dp[i][j] = dp[i-1][j-1] + dp[i][j-1] + 1$

其中:

- $dp[i-1][j-1]$ 表示鸡蛋碎了的情况, 用 $i-1$ 个鸡蛋在 $j-1$ 次尝试中能确定的楼层数
- $dp[i][j-1]$ 表示鸡蛋没碎的情况, 用 i 个鸡蛋在 $j-1$ 次尝试中能确定的楼层数
- $+1$ 表示当前测试的楼层

"""

```
class Solution:
    """
    超级鸡蛋掉落问题的解决方案类
    提供三种不同的解法，从空间和时间复杂度上进行优化
    """

    def __init__(self):
        """初始化 Solution 类"""
        pass

    def validate_inputs(self, k: int, n: int) -> None:
        """
        验证输入参数的有效性

        Args:
            k: 鸡蛋数量
            n: 楼层数量

        Raises:
            ValueError: 如果参数不满足约束条件
        """
        if not 1 <= k <= 100:
            raise ValueError("鸡蛋数量必须在 1 到 100 之间")
        if not 1 <= n <= 10000:
            raise ValueError("楼层数量必须在 1 到 10000 之间")

    def super_egg_drop_1(self, k: int, n: int) -> int:
        """
        解法 1: 二维 dp 数组实现
        时间复杂度: O(k*n)，但在实际执行中会早退出
        空间复杂度: O(k*n)

        Args:
            k: 鸡蛋数量
            n: 楼层数量

        Returns:
            最坏情况下所需的最少尝试次数
        """
        # 输入验证
        self.validate_inputs(k, n)
```

```

# 边界情况：如果只有 1 个鸡蛋，必须从 1 楼开始逐层测试
if k == 1:
    return n

# 创建 dp 数组
dp = [[0] * (n + 1) for _ in range(k + 1)]

# j 表示尝试次数，从 1 开始递增
for j in range(1, n + 1):
    # i 表示使用的鸡蛋数，从 1 开始递增
    for i in range(1, k + 1):
        # 状态转移方程
        dp[i][j] = dp[i - 1][j - 1] + dp[i][j - 1] + 1

    # 当可以确定的楼层数大于等于 n 时，返回当前的尝试次数 j
    if dp[i][j] >= n:
        return j

# 实际上不可能到达这里
return n

```

```
def super_egg_drop_2(self, k: int, n: int) -> int:
    """

```

解法 2：空间优化版本，使用一维 dp 数组

时间复杂度： $O(k*n)$ ，但在实际执行中会早退出

空间复杂度： $O(k)$

Args:

k: 鸡蛋数量

n: 楼层数量

Returns:

最坏情况下所需的最少尝试次数

"""

输入验证

self.validate_inputs(k, n)

边界情况：如果只有 1 个鸡蛋，必须从 1 楼开始逐层测试

if k == 1:
 return n

空间优化：使用一维 dp 数组

dp = [0] * (k + 1)

```

# j 表示尝试次数，从 1 开始递增
for j in range(1, n + 1):
    # 保存上一次的值，用于状态转移
    previous = 0

    # i 表示使用的鸡蛋数，从 1 开始递增
    for i in range(1, k + 1):
        # 暂存当前 dp[i] 的值，因为它将作为下一轮的 previous
        temp = dp[i]

        # 状态转移
        dp[i] = dp[i] + previous + 1

        # 更新 previous 为当前 dp[i] 的旧值
        previous = temp

    # 当可以确定的楼层数大于等于 n 时，返回当前的尝试次数 j
    if dp[i] >= n:
        return j

# 实际上不可能到达这里
return n

```

```
def super_egg_drop_3(self, k: int, n: int) -> int:
    """

```

解法 3：二分搜索优化版本
 时间复杂度： $O(k \log n)$
 空间复杂度： $O(k)$

Args:

k: 鸡蛋数量
 n: 楼层数量

Returns:

最坏情况下所需的最少尝试次数

"""

输入验证
 self.validate_inputs(k, n)

边界情况处理
 if k == 1:
 return n

```
# 计算最小需要多少次尝试才能覆盖 n 层楼
low, high = 1, n
while low < high:
    mid = low + (high - low) // 2
    if self._compute_floors(k, mid) >= n:
        high = mid
    else:
        low = mid + 1

return low
```

```
def _compute_floors(self, k: int, m: int) -> int:
    """
    计算使用 k 个鸡蛋，尝试 m 次，最多能确定的楼层数
    """
```

Args:

k: 鸡蛋数量
m: 尝试次数

Returns:

最多能确定的楼层数

```
    """
    # 使用动态规划计算最多能确定的楼层数
    dp = [0] * (k + 1)
```

```
    for i in range(1, m + 1):
        prev = 0
        for j in range(1, k + 1):
            temp = dp[j]
            dp[j] = dp[j] + prev + 1
            prev = temp
```

提前终止，避免整数溢出

```
        if dp[j] > 10000:
            return 10000
```

```
    return dp[k]
```

```
def test_solution() -> None:
    """
    测试解决方案的正确性和性能
    """
```

```

"""
solution = Solution()

# 测试用例
test_cases = [
    (1, 2),      # 预期输出: 2
    (2, 6),      # 预期输出: 3
    (3, 14),     # 预期输出: 4
    (2, 100),    # 预期输出: 14
    (100, 10000) # 预期输出: 24
]

print("测试不同解法的结果: ")
print("=" * 70)
print(f"{'鸡蛋数':<10} {'楼层数':<10} {'解法 1 结果':<12} {'时间(ms)':<12} {'解法 2 结果':<12} {'时间(ms)':<12} {'解法 3 结果':<12} {'时间(ms)':<12}")
print("=" * 70)

for k, n in test_cases:
    # 测试解法 1
    start_time = time.time()
    result1 = solution.super_egg_drop_1(k, n)
    time1 = (time.time() - start_time) * 1000 # 转换为毫秒

    # 测试解法 2
    start_time = time.time()
    result2 = solution.super_egg_drop_2(k, n)
    time2 = (time.time() - start_time) * 1000

    # 测试解法 3
    start_time = time.time()
    result3 = solution.super_egg_drop_3(k, n)
    time3 = (time.time() - start_time) * 1000

    # 输出结果
    print(f"{k:<10} {n:<10} {result1:<12} {time1:<12.6f} {result2:<12} {time2:<12.6f} {result3:<12} {time3:<12.6f}")
    print("=" * 70)

if __name__ == "__main__":

```

```
test_solution()
```

"""

Python 工程化实战建议：

1. 代码风格与规范：

- 遵循 PEP 8 编码规范
- 使用类型提示提高代码可读性和 IDE 支持
- 采用文档字符串 (docstring) 描述函数功能

2. 性能优化：

- 对于大规模数据，考虑使用 NumPy 进行数组操作
- 使用 lru_cache 装饰器缓存重复计算（如果适用）
- 避免在循环中进行不必要的对象创建

3. 内存管理：

- Python 的垃圾回收机制会自动处理大部分内存管理
- 对于大数组，考虑使用生成器或迭代器减少内存占用
- 注意闭包和循环引用可能导致的内存泄漏

4. 异常处理：

- 使用 try-except 块捕获并处理可能的异常
- 抛出有意义的异常信息，便于调试
- 考虑使用 contextmanager 处理资源获取和释放

5. 测试与调试：

- 使用单元测试框架（如 unittest 或 pytest）确保代码正确性
- 添加日志记录关键操作和状态
- 使用性能分析工具（如 cProfile）识别性能瓶颈

6. 扩展性考虑：

- 将算法封装为可重用的类和函数
- 设计清晰的接口，便于集成到其他系统
- 考虑添加配置参数，使其适用于更广泛的场景

"""

"""

算法优化思考：

1. 数学公式优化：

对于鸡蛋掉落问题，可以利用组合数学公式直接计算：

$$f(k, m) = \sum_{i=1}^{\min(k, m)} C(m, i)$$

当 $f(k, m) \geq n$ 时， m 即为所求的最小尝试次数

这种方法可以避免动态规划的循环计算，对于大规模数据效率更高

2. 缓存优化：

在解法 3 中，_compute_floors 函数可能被多次调用，可以使用缓存优化：

```
from functools import lru_cache
@lru_cache(maxsize=None)
def _compute_floors(self, k: int, m: int) -> int:
    # 实现代码
```

3. 二分搜索优化：

可以进一步优化二分搜索的上界，理论上最大尝试次数不会超过 n

但对于 k 较大的情况，可以使用 $\log_2(n)$ 作为上界

4. 特殊情况处理：

- 当 $k \geq \log_2(n)$ 时，最优解是 $\log_2(n)$ 向上取整
- 当 $k=2$ 时，可以使用数学公式直接求解： $m^2 + m - 2n = 0$

5. 并行计算：

对于需要多次计算不同参数的场景，可以使用多线程或多进程并行计算

```
from concurrent.futures import ThreadPoolExecutor
"""
=====
```

文件：Code05_MaximizeMedian1.java

```
=====
package class128;

// 相邻必选的子序列最大中位数
// 给定一个长度为 n 的数组 arr
// 合法子序列定义为，任意相邻的两个数至少要有一个被挑选所组成的子序列
// 求所有合法子序列中，最大中位数是多少
// 中位数的定义为上中位数
// [1, 2, 3, 4]的上中位数是 2
// [1, 2, 3, 4, 5]的上中位数是 3
// 2 <= n <= 10^5
// 1 <= arr[i] <= 10^9
// 来自真实大厂笔试，对数据验证

import java.util.Arrays;
```

```

public class Code05_MaximizeMedian1 {

    // 正式方法
    // 时间复杂度 O(n * log n)
    public static int maximizeMedian(int[] arr) {
        int n = arr.length;
        int[] sort = new int[n];
        for (int i = 0; i < n; i++) {
            sort[i] = arr[i];
        }
        Arrays.sort(sort);
        int l = 0;
        int r = n - 1;
        int m = 0;
        int ans = -1;
        int[] help = new int[n];
        int[][] dp = new int[n + 1][2];
        while (l <= r) {
            m = (l + r) / 2;
            if (check(arr, help, dp, sort[m], n)) {
                ans = sort[m];
                l = m + 1;
            } else {
                r = m - 1;
            }
        }
        return ans;
    }

    // 任意相邻的两数至少选一个来生成子序列
    // 到底有没有一个合法子序列，能让其中>=x 的数达到一半以上
    public static boolean check(int[] arr, int[] help, int[][] dp, int x, int n) {
        for (int i = 0; i < n; i++) {
            help[i] = arr[i] >= x ? 1 : -1;
        }
        return dp(help, dp, n) > 0;
    }

    // 任意相邻的两数至少选一个来生成子序列
    // 返回合法子序列的最大累加和
    public static int dp(int[] arr, int[][] dp, int n) {
        for (int i = n - 1; i >= 0; i--) {
            // dp[i][0] : i 位置的数字，选和不选皆可，i... 范围上形成合法子序列的最大累加和
        }
    }
}

```

```

// dp[i][1] : i 位置的数字, 一定要选, i... 范围上形成合法子序列的最大累加和
dp[i][0] = Math.max(arr[i] + dp[i + 1][0], dp[i + 1][1]);
dp[i][1] = arr[i] + dp[i + 1][0];
}
return dp[0][0];
}

// 暴力方法
// 为了验证
public static int right(int[] arr) {
    int[] path = new int[arr.length];
    return dfs(arr, 0, true, path, 0);
}

// 暴力方法
// 为了验证
public static int dfs(int[] arr, int i, boolean pre, int[] path, int size) {
    if (i == arr.length) {
        if (size == 0) {
            return 0;
        }
        int[] sort = new int[size];
        for (int j = 0; j < size; j++) {
            sort[j] = path[j];
        }
        Arrays.sort(sort);
        return sort[(sort.length - 1) / 2];
    } else {
        path[size] = arr[i];
        int ans = dfs(arr, i + 1, true, path, size + 1);
        if (pre) {
            ans = Math.max(ans, dfs(arr, i + 1, false, path, size));
        }
        return ans;
    }
}

// 为了测试
public static int[] randomArray(int n, int v) {
    int[] ans = new int[n];
    for (int i = 0; i < n; i++) {
        ans[i] = (int) (Math.random() * v);
    }
}

```

```

        return ans;
    }

// 为了测试
public static void main(String[] args) {
    int n = 20;
    int v = 1000;
    int testTime = 10000;
    System.out.println("功能测试开始");
    for (int i = 0; i < testTime; i++) {
        int size = (int) (Math.random() * n) + 1;
        int[] arr = randomArray(size, v);
        int ans1 = right(arr);
        int ans2 = maximizeMedian(arr);
        if (ans1 != ans2) {
            System.out.println("出错了!");
        }
    }
    System.out.println("功能测试结束");
    System.out.println();

    System.out.println("性能测试开始");
    n = 100000;
    v = 50000000;
    System.out.println("数组长度 : " + n);
    System.out.println("数值范围 : " + v);
    int[] arr = randomArray(n, v);
    long start = System.currentTimeMillis();
    maximizeMedian(arr);
    long end = System.currentTimeMillis();
    System.out.println("正式方法的运行时间 : " + (end - start) + " 毫秒");
    System.out.println("性能测试结束");
}
}

```

文件: Code05_MaximizeMedian2.java

```
=====
package class128;
```

```
// 感谢热心的同学，找到了题目 5 的在线测试
```

```
// 最大平均值和中位数
// 给定一个长度为 n 的数组 arr，现在要选出一些数
// 满足 任意两个相邻的数中至少有一个数被选择
// 被选中的数字平均值的最大值，打印的答案为 double 类型，误差在 0.001 以内
// 被选中的数字中位数的最大值，打印的答案为 int 类型，中位数认为是上中位数
//  $2 \leq n \leq 10^5$ 
//  $1 \leq arr[i] \leq 10^9$ 
// 测试链接：https://atcoder.jp/contests/abc236/tasks/abc236_e
// 提交以下的 code，提交时请把类名改成“Main”，可以通过所有测试用例
```

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.io.StreamTokenizer;
import java.util.Arrays;

public class Code05_MaximizeMedian2 {

    public static int MAXN = 100005;
    public static int n;
    public static int[] arr = new int[MAXN];

    // 求最大平均数需要
    public static double[] help1 = new double[MAXN];
    public static double[][] dp1 = new double[MAXN][2];

    // 求最大上中位数需要
    public static int[] sorted = new int[MAXN];
    public static int[] help2 = new int[MAXN];
    public static int[][] dp2 = new int[MAXN][2];

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        StreamTokenizer in = new StreamTokenizer(br);
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
        in.nextToken();
        n = (int) in.nval;
        for (int i = 1; i <= n; i++) {
            in.nextToken();
            arr[i] = (int) in.nval;
        }
    }
}
```

```

        out.println(average());
        out.println(median());
        out.flush();
        out.close();
        br.close();
    }

// 最大平均数
// 课上没有讲，但是很好理解，也是二分答案
// 假设 arr 中最小值为 l，最大值为 r
// 那么最大平均值必然在[l, r]范围内
// 假设平均值设为中点 m，arr 中所有的数字都减去 m
// 如果此时，任意两个相邻的数中至少有一个数被选择
// 最后得到的结果 >= 0，说明最大平均值至少是 m，去右侧二分
// 否则去左侧二分
public static double average() {
    double l = Double.MAX_VALUE, r = Double.MIN_VALUE, m;
    for (int i = 1; i <= n; i++) {
        l = Math.min(l, arr[i]);
        r = Math.max(r, arr[i]);
    }
    // 二分 60 次，足够让误差小于 0.001
    for (int i = 1; i <= 60; i++) {
        m = (l + r) / 2;
        if (check1(m)) {
            l = m;
        } else {
            r = m;
        }
    }
    return l;
}

public static boolean check1(double x) {
    // arr 中所有的数字都减去 x，得到的数字填入 help1
    for (int i = 1; i <= n; i++) {
        help1[i] = (double) arr[i] - x;
    }
    // 和课上讲的一样的逻辑
    // 任意两个相邻的数中至少有一个数被选择，去得到 dp 表
    dp1[n + 1][0] = dp1[n + 1][1] = 0;
    for (int i = n; i >= 1; i--) {
        dp1[i][0] = Math.max(help1[i] + dp1[i + 1][0], dp1[i + 1][1]);
    }
}

```

```

        dp1[i][1] = help1[i] + dp1[i + 1][0];
    }
    return dp1[1][0] >= 0;
}

// 最大上中位数，就和课上讲的一样了
public static int median() {
    for (int i = 1; i <= n; i++) {
        sorted[i] = arr[i];
    }
    Arrays.sort(sorted, 1, n + 1);
    int l = 1, r = n, m, ans = 0;
    while (l <= r) {
        m = (l + r) / 2;
        if (check2(sorted[m])) {
            ans = sorted[m];
            l = m + 1;
        } else {
            r = m - 1;
        }
    }
    return ans;
}

public static boolean check2(int x) {
    for (int i = 1; i <= n; i++) {
        help2[i] = arr[i] >= x ? 1 : -1;
    }
    dp2[n + 1][0] = dp2[n + 1][1] = 0;
    for (int i = n; i >= 1; i--) {
        dp2[i][0] = Math.max(help2[i] + dp2[i + 1][0], dp2[i + 1][1]);
        dp2[i][1] = help2[i] + dp2[i + 1][0];
    }
    return dp2[1][0] > 0;
}
}

```

}

=====

文件: Code06_MarblesInBags.cpp

=====

// 将珠子放进背包中

```
// 给定一个长度为 n 的数组 weights，背包一共有 k 个
// 其中 weights[i] 是第 i 个珠子的重量
// 请你按照如下规则将所有的珠子放进 k 个背包
// 1, 没有背包是空的
// 2, 如果第 i 个珠子和第 j 个珠子在同一个背包里，那么 i 到 j 所有珠子都要在这个背包里
// 一个背包如果包含 i 到 j 的所有珠子，这个背包的价格是 weights[i]+weights[j]
// 一个珠子分配方案的分数，是所有 k 个背包的价格之和
// 请返回所有分配方案中，最大分数与最小分数的差值为多少
// 1 <= n, k <= 10^5
// 测试链接 : https://leetcode.cn/problems/put-marbles-in-bags/
```

// 使用全局数组来避免动态内存分配

```
long long split_arr[100000]; // 足够大的数组来存储分割点价值
```

```
long long putMarbles(int* weights, int weightsSize, int k) {
    int n = weightsSize;

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

// 简单排序实现（选择排序）

```
for (int i = 0; i < n - 2; i++) {
    int min_idx = i;
    for (int j = i + 1; j < n - 1; j++) {
        if (split_arr[j] < split_arr[min_idx]) {
            min_idx = j;
        }
    }
    if (min_idx != i) {
        long long temp = split_arr[i];
        split_arr[i] = split_arr[min_idx];
        split_arr[min_idx] = temp;
    }
}
```

```
long long small = 0;
long long big = 0;
for (int i = 0, j = n - 2, p = 1; p < k; i++, j--, p++) {
    small += split_arr[i];
    big += split_arr[j];
}
```

```

    return big - small;
}

// 添加 main 函数用于测试
int main() {
    // 示例测试
    int weights1[] = {1, 3, 5, 1};
    int k1 = 2;
    long long result1 = putMarbles(weights1, 4, k1);

    // 另一个示例
    int weights2[] = {1, 3};
    int k2 = 2;
    long long result2 = putMarbles(weights2, 2, k2);

    return 0;
}

```

=====

文件: Code06_MarblesInBags.java

=====

```

package class128;

import java.util.Arrays;

// 将珠子放进背包中
// 给定一个长度为 n 的数组 weights，背包一共有 k 个
// 其中 weights[i] 是第 i 个珠子的重量
// 请你按照如下规则将所有的珠子放进 k 个背包
// 1，没有背包是空的
// 2，如果第 i 个珠子和第 j 个珠子在同一个背包里，那么 i 到 j 所有珠子都要在这个背包里
// 一个背包如果包含 i 到 j 的所有珠子，这个背包的价格是 weights[i]+weights[j]
// 一个珠子分配方案的分数，是所有 k 个背包的价格之和
// 请返回所有分配方案中，最大分数与最小分数的差值为多少
// 1 <= n, k <= 10^5
// 测试链接 : https://leetcode.cn/problems/put-marbles-in-bags/
public class Code06_MarblesInBags {

    public static long putMarbles(int[] weights, int k) {
        int n = weights.length;
        long[] split = new long[n - 1];
        for (int i = 1; i < n; i++) {

```

```

        split[i - 1] = (long) weights[i - 1] + weights[i];
    }
    Arrays.sort(split);
    long small = 0;
    long big = 0;
    for (int i = 0, j = n - 2, p = 1; p < k; i++, j--, p++) {
        small += split[i];
        big += split[j];
    }
    return big - small;
}

}

```

文件: Code06_MarblesInBags.py

```

# 将珠子放进背包中
# 给定一个长度为 n 的数组 weights，背包一共有 k 个
# 其中 weights[i] 是第 i 个珠子的重量
# 请你按照如下规则将所有的珠子放进 k 个背包
# 1，没有背包是空的
# 2，如果第 i 个珠子和第 j 个珠子在同一个背包里，那么 i 到 j 所有珠子都要在这个背包里
# 一个背包如果包含 i 到 j 的所有珠子，这个背包的价格是 weights[i]+weights[j]
# 一个珠子分配方案的分数，是所有 k 个背包的价格之和
# 请返回所有分配方案中，最大分数与最小分数的差值为多少
# 1 <= n, k <= 10^5
#
# 算法思路：
# 这是一个贪心算法问题
# 1. 分析问题：将 n 个珠子分成 k 个连续子数组，每个子数组的价值是首尾元素之和
# 2. 关键观察：相邻两个分割点之间会形成一个子数组，其价值是这两个分割点对应元素之和
# 3. 转化问题：选择 k-1 个分割点，使得最大分数和最小分数的差值最小
# 4. 贪心策略：计算所有相邻元素之和，排序后取最大的 k-1 个和最小的 k-1 个计算差值
#
# 时间复杂度：O(n*log(n))
# 空间复杂度：O(n)
#
# 测试链接：https://leetcode.cn/problems/put-marbles-in-bags/

```

```

def putMarbles(weights, k):
    """
    """

```

计算最大分数与最小分数的差值

算法思路：

1. 计算所有相邻元素之和，这些和代表了可能的分割点价值
2. 将这些和排序
3. 取最大的 $k-1$ 个和最小的 $k-1$ 个计算差值

时间复杂度： $O(n \log(n))$

空间复杂度： $O(n)$

Args:

weights (List[int]): 珠子重量数组
k (int): 背包数量

Returns:

int: 最大分数与最小分数的差值

"""

```
n = len(weights)
```

```
# 计算所有相邻元素之和
```

```
split = []
for i in range(1, n):
    split.append(weights[i - 1] + weights[i])
```

```
# 排序
```

```
split.sort()
```

```
# 计算最大分数和最小分数的差值
```

```
small = 0
big = 0
for i in range(k - 1):
    small += split[i]
    big += split[n - 2 - i]
return big - small
```

```
# 测试代码
```

```
if __name__ == "__main__":
    # 示例测试
    weights = [1, 3, 5, 1]
    k = 2
    result = putMarbles(weights, k)
    print(f"weights = {weights}, k = {k}")
    print(f"结果: {result}")
```

```
# 另一个示例
weights = [1, 3]
k = 2
result = putMarbles(weights, k)
print(f"weights = {weights}, k = {k}")
print(f"结果: {result}")
```

文件: Code07_ClimbingStairs.cpp

```
// 爬楼梯问题
// 假设你正在爬楼梯。需要 n 阶你才能到达楼顶。
// 每次你可以爬 1 或 2 个台阶。你有多少种不同的方法可以爬到楼顶呢？
//
// 算法思路：
// 这是一个经典的动态规划问题，类似于斐波那契数列
// 状态定义：dp[i] 表示到达第 i 阶楼梯的方法数
// 状态转移方程：dp[i] = dp[i-1] + dp[i-2]
// 解释：到达第 i 阶楼梯的方法数等于到达第 i-1 阶楼梯的方法数（再爬 1 步）加上
// 到达第 i-2 阶楼梯的方法数（再爬 2 步）
//
// 空间优化：由于每次只需要前两个状态，可以使用滚动数组优化空间复杂度
//
// 边界条件：
// dp[0] = 1 (0 阶楼梯有 1 种方法：不爬)
// dp[1] = 1 (1 阶楼梯有 1 种方法：爬 1 步)
//
// 时间复杂度：O(n)
// 空间复杂度：O(1) (经过空间优化后)
//
// 测试链接 : https://leetcode.cn/problems/climbing-stairs/
```

```
/**
 * 计算爬楼梯的不同方法数（空间优化版本）
 *
 * @param n 楼梯的阶数
 * @return 到达楼顶的不同方法数
 *
 * 时间复杂度：O(n)
 * 空间复杂度：O(1)
 */
```

```
int climbStairs(int n) {
    // 边界条件处理
    if (n <= 1) {
        return 1;
    }

    // 使用滚动数组优化空间
    // prev2 表示 dp[i-2], prev1 表示 dp[i-1]
    int prev2 = 1; // dp[0] = 1
    int prev1 = 1; // dp[1] = 1
    int current = 0;

    // 从第 2 阶开始计算到第 n 阶
    for (int i = 2; i <= n; i++) {
        // 状态转移方程: dp[i] = dp[i-1] + dp[i-2]
        current = prev1 + prev2;

        // 更新滚动数组
        prev2 = prev1;
        prev1 = current;
    }

    return prev1;
}
```

```
/***
 * 计算爬楼梯的不同方法数（未优化版本，用于对比）
 *
 * @param n 楼梯的阶数
 * @return 到达楼顶的不同方法数
 *
 * 时间复杂度: O(n)
 * 空间复杂度: O(n)
 */
```

```
int climbStairsUnoptimized(int n) {
    // 边界条件处理
    if (n <= 1) {
        return 1;
    }

    // 创建 DP 数组
    int* dp = new int[n + 1];
```

```

// 初始化边界条件
dp[0] = 1;
dp[1] = 1;

// 填充 DP 数组
for (int i = 2; i <= n; i++) {
    // 状态转移方程: dp[i] = dp[i-1] + dp[i-2]
    dp[i] = dp[i - 1] + dp[i - 2];
}

int result = dp[n];
delete[] dp; // 释放内存
return result;
}

// 添加 main 函数用于测试
int main() {
    int n1 = 2;
    int result1 = climbStairs(n1);

    int n2 = 3;
    int result2 = climbStairs(n2);

    int n3 = 5;
    int result3 = climbStairs(n3);

    return 0;
}

```

文件: Code07_ClimbingStairs.java

```

=====
package class128;

// 爬楼梯问题
// 假设你正在爬楼梯。需要 n 阶你才能到达楼顶。
// 每次你可以爬 1 或 2 个台阶。你有多少种不同的方法可以爬到楼顶呢？
//
// 算法思路:
// 这是一个经典动态规划问题，类似于斐波那契数列
// 状态定义: dp[i] 表示到达第 i 阶楼梯的方法数
// 状态转移方程: dp[i] = dp[i-1] + dp[i-2]

```

```
// 解释：到达第 i 阶楼梯的方法数等于到达第 i-1 阶楼梯的方法数（再爬 1 步）加上  
// 到达第 i-2 阶楼梯的方法数（再爬 2 步）  
//  
// 空间优化：由于每次只需要前两个状态，可以使用滚动数组优化空间复杂度  
//  
// 边界条件：  
// dp[0] = 1 (0 阶楼梯有 1 种方法：不爬)  
// dp[1] = 1 (1 阶楼梯有 1 种方法：爬 1 步)  
//  
// 时间复杂度：O(n)  
// 空间复杂度：O(1) (经过空间优化后)  
//  
// 测试链接 : https://leetcode.cn/problems/climbing-stairs/
```

```
public class Code07_ClimbingStairs {  
  
    /**  
     * 计算爬楼梯的不同方法数（空间优化版本）  
     *  
     * @param n 楼梯的阶数  
     * @return 到达楼顶的不同方法数  
     *  
     * 时间复杂度：O(n)  
     * 空间复杂度：O(1)  
     */  
    public static int climbStairs(int n) {  
        // 边界条件处理  
        if (n <= 1) {  
            return 1;  
        }  
  
        // 使用滚动数组优化空间  
        // prev2 表示 dp[i-2], prev1 表示 dp[i-1]  
        int prev2 = 1; // dp[0] = 1  
        int prev1 = 1; // dp[1] = 1  
        int current = 0;  
  
        // 从第 2 阶开始计算到第 n 阶  
        for (int i = 2; i <= n; i++) {  
            // 状态转移方程：dp[i] = dp[i-1] + dp[i-2]  
            current = prev1 + prev2;  
  
            // 更新滚动数组  
            prev2 = prev1;  
            prev1 = current;  
        }  
        return current;  
    }  
}
```

```
        prev2 = prev1;
        prev1 = current;
    }

    return prev1;
}

/***
 * 计算爬楼梯的不同方法数（未优化版本，用于对比）
 *
 * @param n 楼梯的阶数
 * @return 到达楼顶的不同方法数
 *
 * 时间复杂度: O(n)
 * 空间复杂度: O(n)
 */
public static int climbStairsUnoptimized(int n) {
    // 边界条件处理
    if (n <= 1) {
        return 1;
    }

    // 创建 DP 数组
    int[] dp = new int[n + 1];

    // 初始化边界条件
    dp[0] = 1;
    dp[1] = 1;

    // 填充 DP 数组
    for (int i = 2; i <= n; i++) {
        // 状态转移方程: dp[i] = dp[i-1] + dp[i-2]
        dp[i] = dp[i - 1] + dp[i - 2];
    }

    return dp[n];
}

// 为了测试
public static void main(String[] args) {
    // 测试用例
    int n1 = 2;
    System.out.println("n = " + n1 + ", 方法数 = " + climbStairs(n1)); // 输出: 2
}
```

```
int n2 = 3;
System.out.println("n = " + n2 + ", 方法数 = " + climbStairs(n2)); // 输出: 3

int n3 = 5;
System.out.println("n = " + n3 + ", 方法数 = " + climbStairs(n3)); // 输出: 8
}

=====
```

文件: Code07_ClimbingStairs.py

```
# 爬楼梯问题
# 假设你正在爬楼梯。需要 n 阶你才能到达楼顶。
# 每次你可以爬 1 或 2 个台阶。你有多少种不同的方法可以爬到楼顶呢？
#
# 算法思路：
# 这是一个经典的动态规划问题，类似于斐波那契数列
# 状态定义：dp[i] 表示到达第 i 阶楼梯的方法数
# 状态转移方程：dp[i] = dp[i-1] + dp[i-2]
# 解释：到达第 i 阶楼梯的方法数等于到达第 i-1 阶楼梯的方法数（再爬 1 步）加上
# 到达第 i-2 阶楼梯的方法数（再爬 2 步）
#
# 空间优化：由于每次只需要前两个状态，可以使用滚动数组优化空间复杂度
#
# 边界条件：
# dp[0] = 1 (0 阶楼梯有 1 种方法：不爬)
# dp[1] = 1 (1 阶楼梯有 1 种方法：爬 1 步)
#
# 时间复杂度：O(n)
# 空间复杂度：O(1) (经过空间优化后)
#
# 测试链接：https://leetcode.cn/problems/climbing-stairs/
```

```
def climb_stairs(n):
    """
    计算爬楼梯的不同方法数（空间优化版本）

```

Args:

n (int): 楼梯的阶数

Returns:

int: 到达楼顶的不同方法数

时间复杂度: O(n)

空间复杂度: O(1)

"""

边界条件处理

if n <= 1:

 return 1

使用滚动数组优化空间

prev2 表示 dp[i-2], prev1 表示 dp[i-1]

prev2 = 1 # dp[0] = 1

prev1 = 1 # dp[1] = 1

从第 2 阶开始计算到第 n 阶

for i in range(2, n + 1):

 # 状态转移方程: dp[i] = dp[i-1] + dp[i-2]

 current = prev1 + prev2

 # 更新滚动数组

 prev2 = prev1

 prev1 = current

return prev1

def climb_stairs_unoptimized(n):

"""

计算爬楼梯的不同方法数（未优化版本，用于对比）

Args:

 n (int): 楼梯的阶数

Returns:

 int: 到达楼顶的不同方法数

时间复杂度: O(n)

空间复杂度: O(n)

"""

边界条件处理

if n <= 1:

 return 1

创建 DP 数组

```

dp = [0] * (n + 1)

# 初始化边界条件
dp[0] = 1
dp[1] = 1

# 填充 DP 数组
for i in range(2, n + 1):
    # 状态转移方程: dp[i] = dp[i-1] + dp[i-2]
    dp[i] = dp[i - 1] + dp[i - 2]

return dp[n]

# 为了测试
if __name__ == "__main__":
    # 测试用例
    n1 = 2
    print(f"n = {n1}, 方法数 = {climb_stairs(n1)}")  # 输出: 2

    n2 = 3
    print(f"n = {n2}, 方法数 = {climb_stairs(n2)}")  # 输出: 3

    n3 = 5
    print(f"n = {n3}, 方法数 = {climb_stairs(n3)}")  # 输出: 8

```

文件: Code08_SplitArray.cpp

```

// 分割数组的最大值
// 给定一个非负整数数组 nums 和一个整数 m，你需要将这个数组分成 m 个非空的连续子数组。
// 设计一个算法使得这 m 个子数组各自和的最大值最小。
//
// 算法思路:
// 这是一个典型的二分答案问题。
// 1. 答案具有单调性: 最大值越小, 需要分割的子数组越多; 最大值越大, 需要分割的子数组越少
// 2. 二分搜索答案的范围: 左边界是数组中的最大值, 右边界是数组元素之和
// 3. 对于每个中间值, 使用贪心算法检查是否能将数组分割成不超过 m 个子数组,
//     使得每个子数组的和都不超过该中间值
//
// 时间复杂度: O(n * log(sum))
// 空间复杂度: O(1)
//

```

```
// 测试链接 : https://leetcode.cn/problems/split-array-largest-sum/

/**
 * 检查是否能将数组分割成不超过 m 个子数组，使得每个子数组的和都不超过给定值
 * 使用贪心算法实现
 *
 * @param nums 非负整数数组
 * @param numsSize 数组长度
 * @param m 分割成的子数组数量上限
 * @param maxSum 每个子数组和的上限
 * @return 是否能满足分割要求
 *
 * 时间复杂度: O(n)
 * 空间复杂度: O(1)
 */
bool canSplit(int* nums, int numsSize, int m, long long maxSum) {
    int count = 1;           // 当前分割的子数组数量，初始为 1
    long long currentSum = 0; // 当前子数组的和

    for (int i = 0; i < numsSize; i++) {
        int num = nums[i];
        // 如果当前数字本身就超过了上限，无法满足要求
        if (num > maxSum) {
            return false;
        }

        // 如果加上当前数字会超过上限，则需要新开一个子数组
        if (currentSum + num > maxSum) {
            count++;
            currentSum = num;

            // 如果子数组数量超过了 m，无法满足要求
            if (count > m) {
                return false;
            }
        } else {
            // 否则将当前数字加入当前子数组
            currentSum += num;
        }
    }

    return true;
}
```

```

/**
 * 分割数组使得子数组各自和的最大值最小
 *
 * @param nums 非负整数数组
 * @param numsSize 数组长度
 * @param m 分割成的子数组数量
 * @return 分割后子数组各自和的最大值的最小值
 *
 * 时间复杂度: O(n * log(sum))
 * 空间复杂度: O(1)
 */

int splitArray(int* nums, int numsSize, int m) {
    // 确定二分搜索的边界
    // 左边界: 数组中的最大值 (每个元素单独成一组的情况)
    // 右边界: 数组元素之和 (所有元素成一组的情况)
    long long left = 0, right = 0;
    for (int i = 0; i < numsSize; i++) {
        int num = nums[i];
        right += num;
        if (num > left) left = num;
    }

    long long result = right;

    // 二分搜索答案
    while (left <= right) {
        long long mid = left + (right - left) / 2;

        // 检查是否能将数组分割成不超过 m 个子数组, 使得每个子数组的和都不超过 mid
        if (canSplit(nums, numsSize, m, mid)) {
            result = mid;
            right = mid - 1; // 尝试寻找更小的最大值
        } else {
            left = mid + 1; // 需要更大的最大值才能满足分割要求
        }
    }
}

return (int) result;
}

// 添加 main 函数用于测试
int main() {

```

```

// 测试用例 1
int nums1[] = {7, 2, 5, 10, 8};
int m1 = 2;
int result1 = splitArray(nums1, 5, m1);

// 测试用例 2
int nums2[] = {1, 2, 3, 4, 5};
int m2 = 2;
int result2 = splitArray(nums2, 5, m2);

// 测试用例 3
int nums3[] = {1, 4, 4};
int m3 = 3;
int result3 = splitArray(nums3, 3, m3);

return 0;
}

```

=====

文件: Code08_SplitArray.java

```

=====
package class128;

// 分割数组的最大值
// 给定一个非负整数数组 nums 和一个整数 m，你需要将这个数组分成 m 个非空的连续子数组。
// 设计一个算法使得这 m 个子数组各自和的最大值最小。
//
// 算法思路:
// 这是一个典型的二分答案问题。
// 1. 答案具有单调性: 最大值越小, 需要分割的子数组越多; 最大值越大, 需要分割的子数组越少
// 2. 二分搜索答案的范围: 左边界是数组中的最大值, 右边界是数组元素之和
// 3. 对于每个中间值, 使用贪心算法检查是否能将数组分割成不超过 m 个子数组,
//     使得每个子数组的和都不超过该中间值
//
// 时间复杂度: O(n * log(sum))
// 空间复杂度: O(1)
//
// 测试链接 : https://leetcode.cn/problems/split-array-largest-sum/

public class Code08_SplitArray {
    /**

```

```

* 分割数组使得子数组各自和的最大值最小
*
* @param nums 非负整数数组
* @param m 分割成的子数组数量
* @return 分割后子数组各自和的最大值的最小值
*
* 时间复杂度: O(n * log(sum))
* 空间复杂度: O(1)
*/
public static int splitArray(int[] nums, int m) {
    // 确定二分搜索的边界
    // 左边界: 数组中的最大值 (每个元素单独成一组的情况)
    // 右边界: 数组元素之和 (所有元素成一组的情况)
    long left = 0, right = 0;
    for (int num : nums) {
        right += num;
        left = Math.max(left, num);
    }

    long result = right;

    // 二分搜索答案
    while (left <= right) {
        long mid = left + (right - left) / 2;

        // 检查是否能将数组分割成不超过 m 个子数组, 使得每个子数组的和都不超过 mid
        if (canSplit(nums, m, mid)) {
            result = mid;
            right = mid - 1; // 尝试寻找更小的最大值
        } else {
            left = mid + 1; // 需要更大的最大值才能满足分割要求
        }
    }

    return (int) result;
}

/**
* 检查是否能将数组分割成不超过 m 个子数组, 使得每个子数组的和都不超过给定值
* 使用贪心算法实现
*
* @param nums 非负整数数组
* @param m 分割成的子数组数量上限

```

```

* @param maxSum 每个子数组和的上限
* @return 是否能满足分割要求
*
* 时间复杂度: O(n)
* 空间复杂度: O(1)
*/
private static boolean canSplit(int[] nums, int m, long maxSum) {
    int count = 1;          // 当前分割的子数组数量, 初始为1
    long currentSum = 0;    // 当前子数组的和

    for (int num : nums) {
        // 如果当前数字本身就超过了上限, 无法满足要求
        if (num > maxSum) {
            return false;
        }

        // 如果加上当前数字会超过上限, 则需要新开一个子数组
        if (currentSum + num > maxSum) {
            count++;
            currentSum = num;

            // 如果子数组数量超过了 m, 无法满足要求
            if (count > m) {
                return false;
            }
        } else {
            // 否则将当前数字加入当前子数组
            currentSum += num;
        }
    }

    return true;
}

// 为了测试
public static void main(String[] args) {
    // 测试用例 1
    int[] nums1 = {7, 2, 5, 10, 8};
    int m1 = 2;
    System.out.println("数组: [7, 2, 5, 10, 8], m = " + m1 +
                       ", 结果 = " + splitArray(nums1, m1)); // 输出: 18

    // 测试用例 2
}

```

```

int[] nums2 = {1, 2, 3, 4, 5};
int m2 = 2;
System.out.println("数组: [1, 2, 3, 4, 5], m = " + m2 +
    ", 结果 = " + splitArray(nums2, m2)); // 输出: 9

// 测试用例 3
int[] nums3 = {1, 4, 4};
int m3 = 3;
System.out.println("数组: [1, 4, 4], m = " + m3 +
    ", 结果 = " + splitArray(nums3, m3)); // 输出: 4
}

}
=====
```

文件: Code08_SplitArray.py

```

# 分割数组的最大值
# 给定一个非负整数数组 nums 和一个整数 m，你需要将这个数组分成 m 个非空的连续子数组。
# 设计一个算法使得这 m 个子数组各自和的最大值最小。
#
# 算法思路:
# 这是一个典型的二分答案问题。
# 1. 答案具有单调性: 最大值越小, 需要分割的子数组越多; 最大值越大, 需要分割的子数组越少
# 2. 二分搜索答案的范围: 左边界是数组中的最大值, 右边界是数组元素之和
# 3. 对于每个中间值, 使用贪心算法检查是否能将数组分割成不超过 m 个子数组,
#     使得每个子数组的和都不超过该中间值
#
# 时间复杂度: O(n * log(sum))
# 空间复杂度: O(1)
#
# 测试链接 : https://leetcode.cn/problems/split-array-largest-sum/
```

```

def can_split(nums, m, max_sum):
    """
    检查是否能将数组分割成不超过 m 个子数组, 使得每个子数组的和都不超过给定值
    使用贪心算法实现
    """

Args:
```

nums (List[int]): 非负整数数组
m (int): 分割成的子数组数量上限
max_sum (int): 每个子数组和的上限

Returns:

bool: 是否能满足分割要求

时间复杂度: $O(n)$

空间复杂度: $O(1)$

"""

count = 1 # 当前分割的子数组数量, 初始为 1

current_sum = 0 # 当前子数组的和

for num in nums:

如果当前数字本身就超过了上限, 无法满足要求

if num > max_sum:

 return False

如果加上当前数字会超过上限, 则需要新开一个子数组

if current_sum + num > max_sum:

 count += 1

 current_sum = num

如果子数组数量超过了 m, 无法满足要求

if count > m:

 return False

else:

否则将当前数字加入当前子数组

 current_sum += num

return True

def split_array(nums, m):

"""

分割数组使得子数组各自和的最大值最小

Args:

nums (List[int]): 非负整数数组

m (int): 分割成的子数组数量

Returns:

int: 分割后子数组各自和的最大值的最小值

时间复杂度: $O(n * \log(\sum))$

空间复杂度: $O(1)$

"""

确定二分搜索的边界

```

# 左边界：数组中的最大值（每个元素单独成一组的情况）
# 右边界：数组元素之和（所有元素成一组的情况）
left = max(nums)
right = sum(nums)

result = right

# 二分搜索答案
while left <= right:
    mid = left + (right - left) // 2

    # 检查是否能将数组分割成不超过 m 个子数组，使得每个子数组的和都不超过 mid
    if can_split(nums, m, mid):
        result = mid
        right = mid - 1  # 尝试寻找更小的最大值
    else:
        left = mid + 1  # 需要更大的最大值才能满足分割要求

return result

# 为了测试
if __name__ == "__main__":
    # 测试用例 1
    nums1 = [7, 2, 5, 10, 8]
    m1 = 2
    print(f"数组: {nums1}, m = {m1}, 结果 = {split_array(nums1, m1)}")  # 输出: 18

    # 测试用例 2
    nums2 = [1, 2, 3, 4, 5]
    m2 = 2
    print(f"数组: {nums2}, m = {m2}, 结果 = {split_array(nums2, m2)}")  # 输出: 9

    # 测试用例 3
    nums3 = [1, 4, 4]
    m3 = 3
    print(f"数组: {nums3}, m = {m3}, 结果 = {split_array(nums3, m3)}")  # 输出: 4

```

文件: Code09_PutMarblesInBags.cpp

```

// 将珠子放进背包中
// 你有 k 个背包。给你一个下标从 0 开始的整数数组 weights，其中 weights[i] 是第 i 个珠子的重量。

```

```
// 同时给你整数 k 。请你按照如下规则将所有的珠子放进 k 个背包：  
// 1. 没有背包是空的。  
// 2. 如果第 i 个珠子和第 j 个珠子在同一个背包，那么下标在 i 和 j 之间的所有珠子都必须在这同一个  
背包中。  
// 3. 如果一个背包有珠子 i1, i2, ..., im，则该背包的开销是 weights[i1] + weights[im]。  
// 4. 整个分配方案的开销是所有 k 个背包的开销之和。  
// 请你返回所有可能的分配方案中，最大开销与最小开销的差值。
```

```
// 算法思路：  
// 这是一个贪心算法问题。  
// 关键观察：要将数组分割成 k 个连续子数组，我们需要在数组中选择 k-1 个分割点。  
// 每个分割点 i 的贡献是 weights[i] + weights[i+1]（左边子数组的末尾和右边子数组的开头）。  
// 因此，我们可以计算所有可能分割点的贡献值，然后选择最大的 k-1 个和最小的 k-1 个，  
// 它们的差值就是答案。  
  
// 注意：数组的第一个元素和最后一个元素在任何分配方案中都会被计算一次，  
// 所以它们的贡献在最大开销和最小开销中是相同的，可以忽略。  
  
//  
// 时间复杂度：O(n * log(n))  
// 空间复杂度：O(n)  
  
// 测试链接：https://leetcode.cn/problems/put-marbles-in-bags/
```

```
#include <stdio.h>  
#include <stdlib.h>  
  
/**  
 * 简单排序函数（冒泡排序）  
 *  
 * @param arr 待排序数组  
 * @param n 数组长度  
 */  
void sort(long long* arr, int n) {  
    for (int i = 0; i < n - 1; i++) {  
        for (int j = 0; j < n - i - 1; j++) {  
            if (arr[j] > arr[j + 1]) {  
                long long temp = arr[j];  
                arr[j] = arr[j + 1];  
                arr[j + 1] = temp;  
            }  
        }  
    }  
}
```

```
/**  
 * 计算将珠子放进背包中的最大开销与最小开销的差值  
 *  
 * @param weights 珠子的重量数组  
 * @param weightsSize 数组长度  
 * @param k 背包数量  
 * @return 最大开销与最小开销的差值  
 *  
 * 时间复杂度: O(n * log(n))  
 * 空间复杂度: O(n)  
 */  
  
long long putMarbles(int* weights, int weightsSize, int k) {  
    int n = weightsSize;  
  
    // 特殊情况: 如果只有一个背包或者每个珠子一个背包, 差值为 0  
    if (k == 1 || k == n) {  
        return 0;  
    }  
  
    // 计算所有可能分割点的贡献值  
    // 分割点 i 的贡献是 weights[i] + weights[i+1]  
    long long* contributions = new long long[n - 1];  
    for (int i = 0; i < n - 1; i++) {  
        contributions[i] = (long long) weights[i] + weights[i + 1];  
    }  
  
    // 排序贡献值  
    sort(contributions, n - 1);  
  
    // 计算最大开销和最小开销的差值  
    // 选择最大的 k-1 个贡献值减去最小的 k-1 个贡献值  
    long long maxSum = 0, minSum = 0;  
    for (int i = 0; i < k - 1; i++) {  
        minSum += contributions[i]; // 最小的 k-1 个  
        maxSum += contributions[n - 2 - i]; // 最大的 k-1 个  
    }  
  
    delete[] contributions; // 释放内存  
    return maxSum - minSum;  
}  
  
// 添加 main 函数用于测试
```

```

int main() {
    // 测试用例 1
    int weights1[] = {1, 3, 5, 1};
    int k1 = 2;
    long long result1 = putMarbles(weights1, 4, k1);

    // 测试用例 2
    int weights2[] = {1, 3};
    int k2 = 2;
    long long result2 = putMarbles(weights2, 2, k2);

    // 测试用例 3
    int weights3[] = {1, 4, 2, 5, 2};
    int k3 = 3;
    long long result3 = putMarbles(weights3, 5, k3);

    return 0;
}

```

=====

文件: Code09_PutMarblesInBags.java

=====

```

package class128;

// 将珠子放进背包中
// 你有 k 个背包。给你一个下标从 0 开始的整数数组 weights，其中 weights[i] 是第 i 个珠子的重量。
// 同时给你整数 k。请你按照如下规则将所有的珠子放进 k 个背包：
// 1. 没有背包是空的。
// 2. 如果第 i 个珠子和第 j 个珠子在同一个背包，那么下标在 i 和 j 之间的所有珠子都必须在这同一个
// 背包中。
// 3. 如果一个背包有珠子 i1, i2, ..., im，则该背包的开销是 weights[i1] + weights[im]。
// 4. 整个分配方案的开销是所有 k 个背包的开销之和。
// 请你返回所有可能的分配方案中，最大开销与最小开销的差值。

```

```

// 算法思路：
// 这是一个贪心算法问题。
// 关键观察：要将数组分割成 k 个连续子数组，我们需要在数组中选择 k-1 个分割点。
// 每个分割点 i 的贡献是 weights[i] + weights[i+1]（左边子数组的末尾和右边子数组的开头）。
// 因此，我们可以计算所有可能分割点的贡献值，然后选择最大的 k-1 个和最小的 k-1 个，
// 它们的差值就是答案。
//
// 注意：数组的第一个元素和最后一个元素在任何分配方案中都会被计算一次，

```

```

// 所以它们的贡献在最大开销和最小开销中是相同的，可以忽略。
//
// 时间复杂度: O(n * log(n))
// 空间复杂度: O(n)
//
// 测试链接 : https://leetcode.cn/problems/put-marbles-in-bags/

import java.util.Arrays;

public class Code09_PutMarblesInBags {

    /**
     * 计算将珠子放进背包中的最大开销与最小开销的差值
     *
     * @param weights 珠子的重量数组
     * @param k 背包数量
     * @return 最大开销与最小开销的差值
     *
     * 时间复杂度: O(n * log(n))
     * 空间复杂度: O(n)
     */
    public static long putMarbles(int[] weights, int k) {
        int n = weights.length;

        // 特殊情况: 如果只有一个背包或者每个珠子一个背包, 差值为0
        if (k == 1 || k == n) {
            return 0;
        }

        // 计算所有可能分割点的贡献值
        // 分割点 i 的贡献是 weights[i] + weights[i+1]
        long[] contributions = new long[n - 1];
        for (int i = 0; i < n - 1; i++) {
            contributions[i] = (long) weights[i] + weights[i + 1];
        }

        // 排序贡献值
        Arrays.sort(contributions);

        // 计算最大开销和最小开销的差值
        // 选择最大的 k-1 个贡献值减去最小的 k-1 个贡献值
        long maxSum = 0, minSum = 0;
        for (int i = 0; i < k - 1; i++) {

```

```

        minSum += contributions[i];                         // 最小的 k-1 个
        maxSum += contributions[n - 2 - i];                 // 最大的 k-1 个
    }

    return maxSum - minSum;
}

// 为了测试
public static void main(String[] args) {
    // 测试用例 1
    int[] weights1 = {1, 3, 5, 1};
    int k1 = 2;
    System.out.println("weights: " + Arrays.toString(weights1) + ", k = " + k1 +
        ", 结果 = " + putMarbles(weights1, k1)); // 输出: 4

    // 测试用例 2
    int[] weights2 = {1, 3};
    int k2 = 2;
    System.out.println("weights: " + Arrays.toString(weights2) + ", k = " + k2 +
        ", 结果 = " + putMarbles(weights2, k2)); // 输出: 0

    // 测试用例 3
    int[] weights3 = {1, 4, 2, 5, 2};
    int k3 = 3;
    System.out.println("weights: " + Arrays.toString(weights3) + ", k = " + k3 +
        ", 结果 = " + putMarbles(weights3, k3)); // 输出: 4
}
}

```

文件: Code09_PutMarblesInBags.py

```

# 将珠子放进背包中
# 你有 k 个背包。给你一个下标从 0 开始的整数数组 weights，其中 weights[i] 是第 i 个珠子的重量。
# 同时给你整数 k。请你按照如下规则将所有的珠子放进 k 个背包：
# 1. 没有背包是空的。
# 2. 如果第 i 个珠子和第 j 个珠子在同一个背包，那么下标在 i 和 j 之间的所有珠子都必须在这同一个背
# 包中。
# 3. 如果一个背包有珠子 i1, i2, ..., im，则该背包的开销是 weights[i1] + weights[im]。
# 4. 整个分配方案的开销是所有 k 个背包的开销之和。
# 请返回所有可能的分配方案中，最大开销与最小开销的差值。

```

```
# 算法思路:  
# 这是一个贪心算法问题。  
# 关键观察：要将数组分割成 k 个连续子数组，我们需要在数组中选择 k-1 个分割点。  
# 每个分割点 i 的贡献是 weights[i] + weights[i+1]（左边子数组的末尾和右边子数组的开头）。  
# 因此，我们可以计算所有可能分割点的贡献值，然后选择最大的 k-1 个和最小的 k-1 个，  
# 它们的差值就是答案。  
#  
# 注意：数组的第一个元素和最后一个元素在任何分配方案中都会被计算一次，  
# 所以它们的贡献在最大开销和最小开销中是相同的，可以忽略。  
#  
# 时间复杂度：O(n * log(n))  
# 空间复杂度：O(n)  
#  
# 测试链接：https://leetcode.cn/problems/put-marbles-in-bags/
```

```
def put_marbles(weights, k):  
    """  
        计算将珠子放进背包中的最大开销与最小开销的差值  
    """
```

Args:

```
    weights (List[int]): 珠子的重量数组  
    k (int): 背包数量
```

Returns:

```
    int: 最大开销与最小开销的差值
```

时间复杂度：O(n * log(n))

空间复杂度：O(n)

"""

```
n = len(weights)
```

特殊情况：如果只有一个背包或者每个珠子一个背包，差值为 0

```
if k == 1 or k == n:  
    return 0
```

计算所有可能分割点的贡献值

```
# 分割点 i 的贡献是 weights[i] + weights[i+1]  
contributions = []  
for i in range(n - 1):  
    contributions.append(weights[i] + weights[i + 1])
```

排序贡献值

```
contributions.sort()
```

```

# 计算最大开销和最小开销的差值
# 选择最大的 k-1 个贡献值减去最小的 k-1 个贡献值
max_sum = 0
min_sum = 0
for i in range(k - 1):
    min_sum += contributions[i]           # 最小的 k-1 个
    max_sum += contributions[n - 2 - i]     # 最大的 k-1 个

return max_sum - min_sum

# 为了测试
if __name__ == "__main__":
    # 测试用例 1
    weights1 = [1, 3, 5, 1]
    k1 = 2
    print(f"weights: {weights1}, k = {k1}, 结果 = {put_marbles(weights1, k1)}")  # 输出: 4

    # 测试用例 2
    weights2 = [1, 3]
    k2 = 2
    print(f"weights: {weights2}, k = {k2}, 结果 = {put_marbles(weights2, k2)}")  # 输出: 0

    # 测试用例 3
    weights3 = [1, 4, 2, 5, 2]
    k3 = 3
    print(f"weights: {weights3}, k = {k3}, 结果 = {put_marbles(weights3, k3)}")  # 输出: 4

```

=====

文件: Code10_MinDaysToBloom.cpp

=====

```

// 制作 m 束花所需的最少天数
// 给你一个整数数组 bloomDay, 以及两个整数 m 和 k 。
// 现需要制作 m 束花。制作花束时, 需要使用花园中相邻的 k 朵花 。
// 花园中有 n 朵花, 第 i 朵花会在 bloomDay[i] 时盛开, 恰好可以用于一束花中。
// 请你返回从花园中摘 m 束花需要等待的最少的天数。如果不能摘到 m 束花则返回 -1 。

```

```

// 算法思路:
// 这是一个典型的二分答案问题。
// 1. 答案具有单调性: 等待天数越多, 盛开的花朵越多, 能制作的花束也越多
// 2. 二分搜索答案的范围: 左边界是数组中的最小值, 右边界是数组中的最大值
// 3. 对于每个中间值, 使用贪心算法检查是否能在该天数内制作出 m 束花

```

```
//  
// 时间复杂度: O(n * log(max-min))  
// 空间复杂度: O(1)  
  
// 测试链接 : https://leetcode.cn/problems/minimum-number-of-days-to-make-m-bouquets/  
  
/**  
 * 检查是否能在给定天数内制作出指定数量的花束  
 * 使用贪心算法实现  
 *  
 * @param bloomDay 每朵花盛开的天数  
 * @param bloomDaySize 数组长度  
 * @param m 需要制作的花束数量  
 * @param k 每束花需要的相邻花朵数量  
 * @param days 给定的天数  
 * @return 是否能在给定天数内制作出指定数量的花束  
 *  
 * 时间复杂度: O(n)  
 * 空间复杂度: O(1)  
 */  
bool canMakeBouquets(int* bloomDay, int bloomDaySize, int m, int k, int days) {  
    int bouquets = 0; // 已制作的花束数量  
    int consecutive = 0; // 当前连续盛开的花朵数量  
  
    for (int i = 0; i < bloomDaySize; i++) {  
        if (bloomDay[i] <= days) {  
            // 如果当前花朵在给定天数内盛开  
            consecutive++;  
  
            // 如果连续盛开的花朵数量达到了 k 朵，可以制作一束花  
            if (consecutive == k) {  
                bouquets++;  
                consecutive = 0; // 重置连续计数  
            }  
        } else {  
            // 如果当前花朵在给定天数内未盛开，重置连续计数  
            consecutive = 0;  
        }  
    }  
  
    // 检查是否能制作出至少 m 束花  
    return bouquets >= m;  
}
```

```
/**  
 * 计算制作 m 束花所需的最少天数  
 *  
 * @param bloomDay 每朵花盛开的天数  
 * @param bloomDaySize 数组长度  
 * @param m 需要制作的花束数量  
 * @param k 每束花需要的相邻花朵数量  
 * @return 制作 m 束花所需的最少天数, 如果不能制作则返回 -1  
 *  
 * 时间复杂度: O(n * log(max-min))  
 * 空间复杂度: O(1)  
 */  
  
int minDaysToBloom(int* bloomDay, int bloomDaySize, int m, int k) {  
    long long totalFlowersNeeded = (long long) m * k;  
    // 如果需要的花朵总数超过了花园中的花朵数, 无法完成任务  
    if (totalFlowersNeeded > bloomDaySize) {  
        return -1;  
    }  
  
    // 确定二分搜索的边界  
    // 左边界: 数组中的最小值  
    // 右边界: 数组中的最大值  
    int left = bloomDay[0], right = bloomDay[0];  
    for (int i = 1; i < bloomDaySize; i++) {  
        if (bloomDay[i] < left) left = bloomDay[i];  
        if (bloomDay[i] > right) right = bloomDay[i];  
    }  
  
    int result = -1;  
  
    // 二分搜索答案  
    while (left <= right) {  
        int mid = left + (right - left) / 2;  
  
        // 检查是否能在 mid 天内制作出 m 束花  
        if (canMakeBouquets(bloomDay, bloomDaySize, m, k, mid)) {  
            result = mid;  
            right = mid - 1; // 尝试寻找更少的天数  
        } else {  
            left = mid + 1; // 需要更多的天数  
        }  
    }  
}
```

```

    return result;
}

// 添加 main 函数用于测试
int main() {
    // 测试用例 1
    int bloomDay1[] = {1, 10, 3, 10, 2};
    int m1 = 3, k1 = 1;
    int result1 = minDaysToBloom(bloomDay1, 5, m1, k1);

    // 测试用例 2
    int bloomDay2[] = {1, 10, 3, 10, 2};
    int m2 = 3, k2 = 2;
    int result2 = minDaysToBloom(bloomDay2, 5, m2, k2);

    // 测试用例 3
    int bloomDay3[] = {7, 7, 7, 7, 12, 7, 7};
    int m3 = 2, k3 = 3;
    int result3 = minDaysToBloom(bloomDay3, 7, m3, k3);

    return 0;
}

```

=====

文件: Code10_MinDaysToBloom.java

=====

```

package class128;

// 制作 m 束花所需的最少天数
// 给你一个整数数组 bloomDay, 以及两个整数 m 和 k 。
// 现需要制作 m 束花。制作花束时, 需要使用花园中相邻的 k 朵花 。
// 花园中有 n 朵花, 第 i 朵花会在 bloomDay[i] 时盛开, 恰好可以用于一束花中。
// 请你返回从花园中摘 m 束花需要等待的最少的天数。如果不能摘到 m 束花则返回 -1 。

// 算法思路:
// 这是一个典型的二分答案问题。
// 1. 答案具有单调性: 等待天数越多, 盛开的花朵越多, 能制作的花束也越多
// 2. 二分搜索答案的范围: 左边界是数组中的最小值, 右边界是数组中的最大值
// 3. 对于每个中间值, 使用贪心算法检查是否能在该天数内制作出 m 束花
// 
```

// 时间复杂度: $O(n * \log(\max-\min))$

```

// 空间复杂度: O(1)
//
// 测试链接 : https://leetcode.cn/problems/minimum-number-of-days-to-make-m-bouquets/

public class Code10_MinDaysToBloom {

    /**
     * 计算制作 m 束花所需的最少天数
     *
     * @param bloomDay 每朵花盛开的天数
     * @param m 需要制作的花束数量
     * @param k 每束花需要的相邻花朵数量
     * @return 制作 m 束花所需的最少天数, 如果不能制作则返回 -1
     *
     * 时间复杂度: O(n * log(max-min))
     * 空间复杂度: O(1)
     */
    public static int minDaysToBloom(int[] bloomDay, int m, int k) {
        int n = bloomDay.length;

        // 如果需要的花朵总数超过了花园中的花朵数, 无法完成任务
        if ((long) m * k > n) {
            return -1;
        }

        // 确定二分搜索的边界
        // 左边界: 数组中的最小值
        // 右边界: 数组中的最大值
        int left = Integer.MAX_VALUE, right = Integer.MIN_VALUE;
        for (int day : bloomDay) {
            left = Math.min(left, day);
            right = Math.max(right, day);
        }

        int result = -1;

        // 二分搜索答案
        while (left <= right) {
            int mid = left + (right - left) / 2;

            // 检查是否能在 mid 天内制作出 m 束花
            if (canMakeBouquets(bloomDay, m, k, mid)) {
                result = mid;
            }
        }
    }

    private static boolean canMakeBouquets(int[] bloomDay, int m, int k, int days) {
        int count = 0;
        int bouquets = 0;
        for (int day : bloomDay) {
            if (day <= days) {
                count++;
                if (count == k) {
                    count = 0;
                    bouquets++;
                }
            } else {
                count = 0;
            }
        }
        return bouquets >= m;
    }
}

```

```
        right = mid - 1; // 尝试寻找更少的天数
    } else {
        left = mid + 1; // 需要更多的天数
    }
}

return result;
}

/***
 * 检查是否能在给定天数内制作出指定数量的花束
 * 使用贪心算法实现
 *
 * @param bloomDay 每朵花盛开的天数
 * @param m 需要制作的花束数量
 * @param k 每束花需要的相邻花朵数量
 * @param days 给定的天数
 * @return 是否能在给定天数内制作出指定数量的花束
 *
 * 时间复杂度: O(n)
 * 空间复杂度: O(1)
 */
private static boolean canMakeBouquets(int[] bloomDay, int m, int k, int days) {
    int bouquets = 0; // 已制作的花束数量
    int consecutive = 0; // 当前连续盛开的花朵数量

    for (int day : bloomDay) {
        if (day <= days) {
            // 如果当前花朵在给定天数内盛开
            consecutive++;
            // 如果连续盛开的花朵数量达到了 k 朵，可以制作一束花
            if (consecutive == k) {
                bouquets++;
                consecutive = 0; // 重置连续计数
            }
        } else {
            // 如果当前花朵在给定天数内未盛开，重置连续计数
            consecutive = 0;
        }
    }

    // 检查是否能制作出至少 m 束花
    return bouquets >= m;
}
```

```

        return bouquets >= m;
    }

// 为了测试
public static void main(String[] args) {
    // 测试用例 1
    int[] bloomDay1 = {1, 10, 3, 10, 2};
    int m1 = 3, k1 = 1;
    System.out.println("bloomDay: " + java.util.Arrays.toString(bloomDay1) +
        ", m = " + m1 + ", k = " + k1 +
        ", 结果 = " + minDaysToBloom(bloomDay1, m1, k1)); // 输出: 3

    // 测试用例 2
    int[] bloomDay2 = {1, 10, 3, 10, 2};
    int m2 = 3, k2 = 2;
    System.out.println("bloomDay: " + java.util.Arrays.toString(bloomDay2) +
        ", m = " + m2 + ", k = " + k2 +
        ", 结果 = " + minDaysToBloom(bloomDay2, m2, k2)); // 输出: -1

    // 测试用例 3
    int[] bloomDay3 = {7, 7, 7, 7, 12, 7, 7};
    int m3 = 2, k3 = 3;
    System.out.println("bloomDay: " + java.util.Arrays.toString(bloomDay3) +
        ", m = " + m3 + ", k = " + k3 +
        ", 结果 = " + minDaysToBloom(bloomDay3, m3, k3)); // 输出: 12
}
}

```

文件: Code10_MinDaysToBloom.py

```

# 制作 m 束花所需的最少天数
# 给你一个整数数组 bloomDay, 以及两个整数 m 和 k。
# 现需要制作 m 束花。制作花束时, 需要使用花园中相邻的 k 朵花。
# 花园中有 n 朵花, 第 i 朵花会在 bloomDay[i] 时盛开, 恰好可以用于一束花中。
# 请你返回从花园中摘 m 束花需要等待的最少的天数。如果不能摘到 m 束花则返回 -1。

# 算法思路:
# 这是一个典型的二分答案问题。
# 1. 答案具有单调性: 等待天数越多, 盛开的花朵越多, 能制作的花束也越多
# 2. 二分搜索答案的范围: 左边界是数组中的最小值, 右边界是数组中的最大值
# 3. 对于每个中间值, 使用贪心算法检查是否能在该天数内制作出 m 束花

```

```
#  
# 时间复杂度: O(n * log(max-min))  
# 空间复杂度: O(1)  
  
# 测试链接 : https://leetcode.cn/problems/minimum-number-of-days-to-make-m-bouquets/
```

```
def can_make_bouquets(bloom_day, m, k, days):
```

```
    """
```

```
    检查是否能在给定天数内制作出指定数量的花束  
    使用贪心算法实现
```

Args:

```
    bloom_day (List[int]): 每朵花盛开的天数  
    m (int): 需要制作的花束数量  
    k (int): 每束花需要的相邻花朵数量  
    days (int): 给定的天数
```

Returns:

```
    bool: 是否能在给定天数内制作出指定数量的花束
```

时间复杂度: O(n)

空间复杂度: O(1)

```
"""
```

```
bouquets = 0    # 已制作的花束数量  
consecutive = 0 # 当前连续盛开的花朵数量
```

```
for day in bloom_day:
```

```
    if day <= days:
```

```
        # 如果当前花朵在给定天数内盛开
```

```
        consecutive += 1
```

```
        # 如果连续盛开的花朵数量达到了 k 朵, 可以制作一束花
```

```
        if consecutive == k:
```

```
            bouquets += 1
```

```
            consecutive = 0 # 重置连续计数
```

```
    else:
```

```
        # 如果当前花朵在给定天数内未盛开, 重置连续计数
```

```
        consecutive = 0
```

```
# 检查是否能制作出至少 m 束花
```

```
return bouquets >= m
```

```
def min_days_to_bloom(bloom_day, m, k):
```

```
"""
```

计算制作 m 束花所需的最少天数

Args:

- bloom_day (List[int]): 每朵花盛开的天数
- m (int): 需要制作的花束数量
- k (int): 每束花需要的相邻花朵数量

Returns:

- int: 制作 m 束花所需的最少天数, 如果不能制作则返回 -1

时间复杂度: $O(n * \log(\max - \min))$

空间复杂度: $O(1)$

```
"""
```

```
n = len(bloom_day)
```

```
# 如果需要的花朵总数超过了花园中的花朵数, 无法完成任务
```

```
if m * k > n:  
    return -1
```

```
# 确定二分搜索的边界
```

```
# 左边界: 数组中的最小值
```

```
# 右边界: 数组中的最大值
```

```
left = min(bloom_day)  
right = max(bloom_day)
```

```
result = -1
```

```
# 二分搜索答案
```

```
while left <= right:  
    mid = left + (right - left) // 2
```

```
# 检查是否能在  $mid$  天内制作出  $m$  束花
```

```
if can_make_bouquets(bloom_day, m, k, mid):  
    result = mid  
    right = mid - 1 # 尝试寻找更少的天数  
else:  
    left = mid + 1 # 需要更多的天数
```

```
return result
```

```
# 为了测试
```

```
if __name__ == "__main__":
```

```
# 测试用例 1
bloom_day1 = [1, 10, 3, 10, 2]
m1, k1 = 3, 1
print(f"bloomDay: {bloom_day1}, m = {m1}, k = {k1}, 结果 = {min_days_to_bloom(bloom_day1, m1, k1)}") # 输出: 3

# 测试用例 2
bloom_day2 = [1, 10, 3, 10, 2]
m2, k2 = 3, 2
print(f"bloomDay: {bloom_day2}, m = {m2}, k = {k2}, 结果 = {min_days_to_bloom(bloom_day2, m2, k2)}") # 输出: -1

# 测试用例 3
bloom_day3 = [7, 7, 7, 7, 12, 7, 7]
m3, k3 = 2, 3
print(f"bloomDay: {bloom_day3}, m = {m3}, k = {k3}, 结果 = {min_days_to_bloom(bloom_day3, m3, k3)}") # 输出: 12
```
