

=====

文件夹: class055_DynamicProgramming

=====

[Markdown 文件]

=====

文件: dp_problems_report.md

=====

动态规划相关题目搜索结果

Grid Path

1. **Cherry Pickup** (LeetCode)

- 链接: <https://leetcode.cn/problems/cherry-pickup/>
- 难度: Hard
- 标签: Dynamic Programming, Grid, Matrix
- 描述: 从左上角到右下角再返回左上角, 收集樱桃的最大数量

2. **Cherry Pickup II** (LeetCode)

- 链接: <https://leetcode.cn/problems/cherry-pickup-ii/>
- 难度: Hard
- 标签: Dynamic Programming, Grid, Matrix
- 描述: 两个机器人从顶部出发, 收集樱桃的最大数量

3. **Minimum Path Sum** (LeetCode)

- 链接: <https://leetcode.cn/problems/minimum-path-sum/>
- 难度: Medium
- 标签: Dynamic Programming, Grid, Matrix
- 描述: 从左上角到右下角的最小路径和

4. **Unique Paths** (LeetCode)

- 链接: <https://leetcode.cn/problems/unique-paths/>
- 难度: Medium
- 标签: Dynamic Programming, Math, Combinatorics
- 描述: 从左上角到右下角的不同路径数量

5. **Unique Paths II** (LeetCode)

- 链接: <https://leetcode.cn/problems/unique-paths-ii/>
- 难度: Medium
- 标签: Dynamic Programming, Grid, Matrix
- 描述: 有障碍物的网格中从左上角到右下角的不同路径数量

6. **Dungeon Game** (LeetCode)

- 链接: <https://leetcode.cn/problems/dungeon-game/>
- 难度: Hard
- 标签: Dynamic Programming, Binary Search
- 描述: 骑士从左上角到右下角的最小初始健康值

7. **Walking on a Grid** (UVa OJ)

- 链接:

https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=1854

- 难度: Hard
- 标签: Dynamic Programming, Grid
- 描述: 在网格中行走, 有负数, 最多允许 k 次负数

Frog Jump

1. **Frog Jump** (LeetCode)

- 链接: <https://leetcode.cn/problems/frog-jump/>
- 难度: Hard
- 标签: Dynamic Programming, Hash Table
- 描述: 青蛙过河, 判断能否到达最后一块石头

2. **Jump Game** (LeetCode)

- 链接: <https://leetcode.cn/problems/jump-game/>
- 难度: Medium
- 标签: Greedy, Array, Dynamic Programming
- 描述: 判断能否从第一个位置跳到最后一个位置

3. **Jump Game II** (LeetCode)

- 链接: <https://leetcode.cn/problems/jump-game-ii/>
- 难度: Medium
- 标签: Greedy, Array, Dynamic Programming
- 描述: 跳到最后一个位置的最少跳跃次数

4. **Jump Game V** (LeetCode)

- 链接: <https://leetcode.cn/problems/jump-game-v/>
- 难度: Hard
- 标签: Dynamic Programming, Sorting
- 描述: 在数组中跳跃, 每次跳跃不能超过固定距离

5. **Jump Game VI** (LeetCode)

- 链接: <https://leetcode.cn/problems/jump-game-vi/>
- 难度: Medium
- 标签: Dynamic Programming, Queue, Heap

- 描述: 在数组中跳跃, 每次最多跳 k 步, 求最大得分

6. **Minimum Jumps** (HackerEarth)

- 链接: <https://www.hackerearth.com/practice/data-structures/arrays/1-d/practice-problems/algorithm/minimum-jumps-4/description/>

- 难度: Medium

- 标签: Dynamic Programming, BFS

- 描述: 计算从数组第一个元素跳到最后一个元素的最小跳跃次数

Subarray Product

1. **Maximum Product Subarray** (LeetCode)

- 链接: <https://leetcode.cn/problems/maximum-product-subarray/>

- 难度: Medium

- 标签: Dynamic Programming, Array

- 描述: 乘积最大的连续子数组

2. **Subarray Product Less Than K** (LeetCode)

- 链接: <https://leetcode.cn/problems/subarray-product-less-than-k/>

- 难度: Medium

- 标签: Array, Sliding Window

- 描述: 乘积小于 K 的连续子数组数量

3. **The Number of Products** (Codeforces)

- 链接: <https://codeforces.com/problemset/problem/1215/B>

- 难度: Easy

- 标签: Dynamic Programming, Math

- 描述: 统计乘积为正和负的子数组数量

Subsequence

1. **Longest Increasing Subsequence** (LeetCode)

- 链接: <https://leetcode.cn/problems/longest-increasing-subsequence/>

- 难度: Medium

- 标签: Dynamic Programming, Binary Search

- 描述: 最长递增子序列

2. **Longest Common Subsequence** (LeetCode)

- 链接: <https://leetcode.cn/problems/longest-common-subsequence/>

- 难度: Medium

- 标签: Dynamic Programming, String

- 描述: 两个字符串的最长公共子序列

3. **Longest Palindromic Subsequence** (LeetCode)
- 链接: <https://leetcode.cn/problems/longest-palindromic-subsequence/>
 - 难度: Medium
 - 标签: Dynamic Programming, String
 - 描述: 最长回文子序列

Combinatorics

1. **Knight Dialer** (LeetCode)
- 链接: <https://leetcode.cn/problems/knight-dialer/>
 - 难度: Medium
 - 标签: Dynamic Programming, Matrix Exponentiation
 - 描述: 骑士在电话垫上跳跃, 计算不同长度的数字序列数量

2. **Domino and Tromino Tiling** (LeetCode)
- 链接: <https://leetcode.cn/problems/domino-and-tromino-tiling/>
 - 难度: Medium
 - 标签: Dynamic Programming
 - 描述: 用多米诺骨牌和托米诺骨牌铺满 $2*n$ 的面板

Other

1. **方格取数** (牛客网)
- 链接: <https://ac.nowcoder.com/acm/problem/14552>
 - 难度: Medium
 - 标签: Dynamic Programming, Grid
 - 描述: 与摘樱桃问题非常相似, 两个人从左上角出发到右下角取数
2. **滑雪** (洛谷)
- 链接: <https://www.luogu.com.cn/problem/P1434>
 - 难度: Medium
 - 标签: Dynamic Programming, DFS, BFS
 - 描述: 寻找最长滑雪路径, 每步只能滑向相邻四个方向且高度更低的位置
3. **种树** (洛谷)
- 链接: <https://www.luogu.com.cn/problem/P1250>
 - 难度: Easy
 - 标签: Greedy, Interval
 - 描述: 区间覆盖问题, 贪心选择最优策略

4. **Flying to Fredericton** (UVa OJ)

- 链接:

https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&p

roblem=2255

- 难度: Medium
- 标签: Graph, Shortest Path, Dynamic Programming
- 描述: 多段最短路径问题, 允许最多 k 次飞行

5. ****MICE AND MAZE** (SPOJ)**

- 链接: <https://www.spoj.com/problems/MICEMAZE/>
- 难度: Easy
- 标签: Graph, BFS, Dijkstra
- 描述: 迷宫寻路问题, 计算能在给定时间内到达终点的老鼠数量

6. ****Doing Homework** (杭电 HDU)**

- 链接: <http://acm.hdu.edu.cn/showproblem.php?pid=1074>
- 难度: Medium
- 标签: Dynamic Programming, Bitmask
- 描述: 任务调度问题, 求最小扣分

7. ****Fight with Monsters** (Codeforces)**

- 链接: <https://codeforces.com/problemset/problem/1296/D>
- 难度: Easy
- 标签: Greedy, Sorting
- 描述: 贪心策略解决怪物战斗问题

8. ****Dividing Chocolate** (AtCoder)**

- 链接: https://atcoder.jp/contests/abc159/tasks/abc159_e
- 难度: Hard
- 标签: Dynamic Programming, Bitmask, Prefix Sum
- 描述: 二维网格分割问题, 需要类似的状态转移思路

9. ****Single Wildcard Pattern Matching** (Codeforces)**

- 链接: <https://codeforces.com/problemset/problem/965/D>
- 难度: Medium
- 标签: String, Greedy
- 描述: 字符串匹配问题, 但涉及到间隔匹配的概念

=====

文件: README.md

=====

Class187 算法专题总结

本目录主要包含了一些高级动态规划和贪心算法的实现, 涵盖了多种经典问题和优化技巧。

主内容

1. 动态规划融合场景 (DP Fusion)

文件: [DPFusion. java] (DPFusion. java)

实现了多种 DP 与其他算法结合的场景:

- DP+数论 (模意义下的动态规划)
- DP+字符串 (基于后缀自动机的计数)
- DP+计算几何 (凸包上的动态规划)

相关算法

- SMAWK 算法: 用于在 Monge 矩阵中快速找出每行的最小值
- Aliens Trick (WQS 二分): 解决需要恰好分成 k 个部分的优化问题
- 分层图最短路径: 在带有状态约束的图中寻找最短路径
- 期望 DP 与高斯消元: 解决包含环的期望 DP 问题
- 插头 DP: 求解网格图中的回路计数、路径覆盖等问题
- 树上背包优化: 使用 small-to-large 合并优化的树上背包问题

2. 四边形不等式优化 (Quadrangle Optimization)

文件: [QuadrangleOptimization. java] (QuadrangleOptimization. java)

用于优化满足四边形不等式性质和决策单调性的区间 DP 问题, 可以将时间复杂度从 $O(n^3)$ 降低到 $O(n^2)$ 。

相关问题

- 石子合并问题
- LeetCode 312. 戳气球
- LeetCode 1000. 合并石头的最低成本
- 矩阵链乘法问题

3. 概率/期望 DP (Probability DP)

文件: [ProbabilityDP. java] (ProbabilityDP. java)

处理概率和期望问题的动态规划方法, 通常使用逆序 DP。

相关问题

- LeetCode 808. 分汤
- LeetCode 688. 骑士在棋盘上的概率
- LeetCode 576. 出界的路径数
- 爬楼梯问题的期望版本

4. 子集覆盖问题 (Set Cover)

文件: [SetCover. java] (SetCover. java)

经典的子集覆盖问题及其变种，使用贪心策略求解。

相关问题

- LeetCode 1541. 平衡括号字符串的最少插入次数
- LeetCode 1689. 十二进制数的最少数目
- LeetCode 45. 跳跃游戏 II

5. 区间染色算法 (Interval Coloring)

文件: [IntervalColoring.java] (IntervalColoring.java)

基于贪心策略的区间问题解决方案。

相关问题

- LeetCode 435. 无重叠区间
- LeetCode 452. 用最少量的箭引爆气球
- LeetCode 253. 会议室 II

多语言实现

本目录中的算法问题大多提供了多种编程语言的实现:

- Java (主要实现语言)
- C++
- Python

算法特点

1. ****高级优化技巧**:** 包含了多种 DP 优化技术，如四边形不等式优化、SMAWK 算法、Aliens Trick 等
2. ****问题融合**:** 展示了 DP 与其他算法领域（数论、字符串、计算几何等）的结合
3. ****贪心策略**:** 提供了多种贪心算法的实现和应用
4. ****实际应用**:** 所有算法都结合了 LeetCode 等平台上的实际问题

学习建议

1. 理解每种算法的基本思想和适用场景
2. 掌握算法的时间和空间复杂度分析
3. 熟悉算法在经典问题中的应用
4. 通过 LeetCode 等平台练习相关题目

[代码文件]

文件: Code01_CherryPickup.cpp

```
=====

// 摘樱桃
// 给定一个 n*n 的正方形矩阵 grid，每个格子值只有三种-1、0、1
// -1 表示格子不能走、0 表示格子可以走但是没有樱桃、1 表示格子可以走且有一颗樱桃
// 你的目标是从左上角走到右下角，每一步只能 向下 或者 向右
// 然后从右下角走回左上角，每一步只能 向上 或者 向左
// 这个过程中，想尽量多的获得樱桃，但是有樱桃的格子，只能捡一次
// 返回最多能获得多少樱桃，如果不存在通路返回 0
// 测试链接 : https://leetcode.cn/problems/cherry-pickup/
```

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

```
/**

* 算法思路深度解析:
* 1. 这道题可以看作是两个人同时从(0, 0)出发，走到(n-1, n-1)位置能收集的最大樱桃数
*   - 这种转化是关键，因为来回走一次等价于两个人同时从起点走到终点
*   - 这样可以避免处理重复访问的问题
* 2. 使用三维动态规划 dp[i][j][k]，其中：
*   - i 表示第一个人的横坐标
*   - j 表示第一个人的纵坐标
*   - k 表示第二个人的横坐标
*   - 第二个人的纵坐标可以由 d = i + j - k 计算得出（因为两人走的步数相同）
*   - 这个状态定义是一种空间优化，将四维问题转化为三维问题
* 3. 每个状态可以从四个前驱状态转移而来：
*   - 两人都向右：dp[i][j-1][k]
*   - 第一人向右，第二人向下：dp[i][j-1][k-1]
*   - 第一人向下，第二人向右：dp[i-1][j][k]
*   - 两人都向下：dp[i-1][j][k-1]
* 4. 关键优化点：
*   - 如果两人在同一个格子，只计算一次樱桃数量
*   - 使用记忆化搜索避免重复计算
*
* 时间复杂度分析:
* - 状态数: O(n^3)，其中 n 是矩阵的边长
* - 每个状态需要考虑 4 个前驱状态
* - 总时间复杂度: O(n^3)
*
* 空间复杂度分析:
* - 动态规划数组大小: O(n^3)
* - 递归调用栈深度: O(n)
```

```

* - 总空间复杂度: O(n^3)
*
* C++实现注意事项:
* 1. 使用 vector<vector<vector<int>>> 创建三维数组, 注意初始化大小
* 2. 使用-2 作为未访问状态的标记, 区分于-1 (表示无法到达)
* 3. 在递归函数中, 需要注意引用传递 vector 以避免拷贝开销
* 4. 边界条件检查必须全面, 防止数组越界访问
*
* 工程化考量:
* 1. 边界条件处理: 确保不越界, 检查障碍物
* 2. 初始化策略: 使用特殊值(-2)标记未访问状态
* 3. 异常处理: 处理无法到达终点的情况
* 4. 优化方向: 可以通过滚动数组进行空间优化
*
* 常见问题排查:
* 1. 边界越界: 在访问数组前检查索引范围
* 2. 障碍物处理: 遇到障碍物返回-1
* 3. 重复计算: 确保使用记忆化搜索
* 4. 结果判断: 如果最终结果为-1, 返回 0 表示不存在通路
*/

```

```

class Solution {
public:
    /**
     * 计算能够摘到的最大樱桃数
     * @param grid n*n 的正方形矩阵
     * @return 最多能获得的樱桃数, 如果不存在通路返回 0
     */
    int cherryPickup(vector<vector<int>>& grid) {
        int n = grid.size();
        // dp[i][j][k] 表示第一个人在(i, j), 第二个人在(k, i+j-k)时能摘到的最大樱桃数
        vector<vector<vector<int>>> dp(n, vector<vector<int>>(n, vector<int>(n, -2)));
        int ans = f(grid, n, 0, 0, 0, dp);
        return ans == -1 ? 0 : ans;
    }

private:
    /**
     * 递归函数, 计算从(0, 0)到(n-1, n-1)两人能摘到的最大樱桃数
     * @param grid 矩阵
     * @param n 矩阵边长
     * @param a 第一个人的横坐标
     * @param b 第一个人的纵坐标
     */

```

```

* @param c 第二个人的横坐标
* @param dp 动态规划数组
* @return 能摘到的最大樱桃数，如果无法到达终点返回-1
*/
int f(vector<vector<int>>& grid, int n, int a, int b, int c, vector<vector<int>>& dp)
{
    // 计算第二个人的纵坐标
    int d = a + b - c;
    // 边界条件检查
    if (a == n || b == n || c == n || d == n || grid[a][b] == -1 || grid[c][d] == -1) {
        return -1;
    }
    // 到达终点
    if (a == n - 1 && b == n - 1) {
        return grid[a][b];
    }
    // 如果已经计算过，直接返回结果
    if (dp[a][b][c] != -2) {
        return dp[a][b][c];
    }
    // 计算当前位置的樱桃数
    int get = (a == c && b == d) ? grid[a][b] : (grid[a][b] + grid[c][d]);
    // 四种可能的移动方式
    int next = f(grid, n, a + 1, b, c + 1, dp); // 两人都向下
    next = max(next, f(grid, n, a + 1, b, c, dp)); // 第一人向下，第二人向右
    next = max(next, f(grid, n, a, b + 1, c + 1, dp)); // 第一人向右，第二人向下
    next = max(next, f(grid, n, a, b + 1, c, dp)); // 两人都向右
    int ans = -1;
    if (next != -1) {
        ans = get + next;
    }
    dp[a][b][c] = ans;
    return ans;
}
};

// 类似题目与训练拓展:
/***
 * 以下是与摘樱桃问题相关的 15 个类似题目，涵盖了多种算法平台
 * 这些题目展示了动态规划、状态压缩、路径优化等相关技巧的应用
 */

```

// 1. LeetCode 1463. Cherry Pickup II (摘樱桃 II)

// 链接: <https://leetcode.cn/problems/cherry-pickup-ii/>
// 区别: 两个机器人分别从(0, 0)和(0, cols-1)出发, 只能向下移动, 每步可选择三个方向
// 算法: 三维动态规划, 状态定义为 $dp[i][j1][j2]$, 表示两个机器人在第 i 行第 j1 和 j2 列的最大樱桃数

// 2. LeetCode 64. Minimum Path Sum (最小路径和)
// 链接: <https://leetcode.cn/problems/minimum-path-sum/>
// 区别: 求从左上角到右下角的最小路径和, 每步只能向下或向右
// 算法: 二维动态规划, $dp[i][j] = \min(dp[i-1][j], dp[i][j-1]) + grid[i][j]$

// 3. LeetCode 174. Dungeon Game (地下城游戏)
// 链接: <https://leetcode.cn/problems/dungeon-game/>
// 区别: 骑士需要从左上角到右下角, 保证健康点数始终大于 0 的最小初始值
// 算法: 从右下角向左上角动态规划, $dp[i][j] = \max(1, \min(dp[i+1][j], dp[i][j+1]) - dungeon[i][j])$

// 4. LeetCode 741. Cherry Pickup (本题)
// 链接: <https://leetcode.cn/problems/cherry-pickup/>
// 算法: 三维动态规划, 转化为两人同时从起点到终点的问题

// 5. LeetCode 62. Unique Paths (不同路径)
// 链接: <https://leetcode.cn/problems/unique-paths/>
// 区别: 计算从左上角到右下角的不同路径数量, 每步只能向下或向右
// 算法: 组合数学或二维动态规划

// 6. LeetCode 63. Unique Paths II (不同路径 II)
// 链接: <https://leetcode.cn/problems/unique-paths-ii/>
// 区别: 网格中有障碍物, 计算不同路径数量
// 算法: 二维动态规划, 遇到障碍物时 $dp[i][j] = 0$

// 7. Codeforces 1296D – Fight with Monsters
// 链接: <https://codeforces.com/problemset/problem/1296/D>
// 区别: 贪心策略解决怪物战斗问题, 但状态转移思想类似
// 算法: 排序后贪心选择最优策略

// 8. AtCoder ABC159E – Dividing Chocolate
// 链接: https://atcoder.jp/contests/abc159/tasks/abc159_e
// 区别: 二维网格分割问题, 但需要类似的状态转移思路
// 算法: 前缀和+状态压缩动态规划

// 9. 洛谷 P1434 [SHOI2002] 滑雪
// 链接: <https://www.luogu.com.cn/problem/P1434>
// 区别: 寻找最长滑雪路径, 每步只能滑向相邻四个方向且高度更低的位置
// 算法: 记忆化搜索或拓扑排序动态规划

```

// 10. 牛客网 NC14552 方格取数
//     链接: https://ac.nowcoder.com/acm/problem/14552
//     区别: 与摘樱桃问题非常相似, 也是两个人从左上角出发到右下角取数
//     算法: 三维动态规划, 状态定义与本题类似

// 11. UVa 10913 - Walking on a Grid
//     链接:
https://onlinejudge.org/index.php?option=com\_onlinejudge&Itemid=8&category=24&page=show\_problem&problem=1854
//     区别: 在网格中行走, 有负数, 最多允许 k 次负数
//     算法: 四维动态规划, 状态定义为  $dp[i][j][k][d]$ , 表示在(i, j)位置, 已经经过 k 次负数, 方向为 d

// 12. SPOJ - SBANK
//     链接: https://www.spoj.com/problems/SBANK/
//     区别: 银行账号排序问题, 使用哈希表优化
//     算法: 哈希表+排序

// 13. HackerEarth - Roy and Coin Boxes
//     链接: https://www.hackerearth.com/practice/data-structures/arrays/1-d/practice-problems/algorithm/roy-and-coin-boxes-1/description/
//     区别: 区间更新问题, 使用差分数组优化
//     算法: 差分数组+前缀和

// 14. 杭电 HDU 1024 - Max Sum Plus Plus
//     链接: http://acm.hdu.edu.cn/showproblem.php?pid=1024
//     区别: 最大 m 段子数组和问题
//     算法: 二维动态规划优化为一维

// 15. Codeforces 1295E - Permutation Separation
//     链接: https://codeforces.com/problemset/problem/1295/E
//     区别: 排列分割问题, 需要找到最优分割点
//     算法: 前缀和+动态规划

/***
 * 算法本质与技巧总结:
 *
 * 1. 问题转化: 将“来回走一次”转化为“两个人同时走一次”, 这是解决这类问题的关键技巧
 *   - 这种转化避免了处理重复访问问题, 简化了状态定义
 *   - 可以扩展到更多类似路径优化问题
 *   - 转化思维体现了算法设计中的问题重构能力
 *
 * 2. 状态压缩: 利用步数相同的特性, 将四维状态压缩为三维
 */

```

```
*      - 这种优化在路径规划问题中非常常见
*      - 状态压缩可以显著减少空间复杂度
*      - 基于问题特性的状态压缩是高级动态规划的重要技巧
*
* 3. 记忆化搜索: 避免重复计算, 提高效率
*      - 递归实现的记忆化搜索比迭代方式更直观
*      - 对于复杂状态转移方程的问题, 记忆化搜索更易实现
*      - 在 C++ 中可以使用数组或哈希表实现记忆化
*
* 4. 边界处理: 仔细处理边界条件和障碍物情况
*      - 边界检查在网格类问题中尤为重要
*      - 障碍物的处理需要特殊考虑
*      - 正确的边界处理是算法正确性的关键
*
* 5. 结果判断: 处理无法到达终点的异常情况
*      - 返回合适的默认值表示无解
*      - 异常情况的处理体现了代码的健壮性
*/
```

```
/***
 * C++工程化实战建议:
 *
 * 1. 测试用例设计:
 *      - 空输入: n=0 的情况
 *      - 单元素网格: 只有起点和终点的情况
 *      - 全是障碍物的情况
 *      - 无法到达终点的情况
 *      - 最优路径需要交叉的情况
 *
 * 2. 性能优化策略:
 *      - 对于大规模数据, 可以使用滚动数组将空间复杂度优化到 O(n^2)
 *      - 使用 memset 或 fill 快速初始化数组
 *      - 考虑使用迭代方式代替递归, 避免栈溢出
 *      - 使用 const 引用传递参数, 减少拷贝开销
 *      - 合理选择数据类型, 避免不必要的类型转换
 *
 * 3. 代码健壮性提升:
 *      - 加入输入合法性检查, 防止非法输入导致程序崩溃
 *      - 使用 try-catch 块处理可能的异常
 *      - 在访问数组前进行边界检查
 *      - 使用断言验证关键条件
 *      - 编写完整的单元测试确保功能正确
*
```

- * 4. C++特有优化技巧:
 - 使用 vector<vector<vector<int>>>创建动态规划数组
 - 利用 emplace_back 避免不必要的拷贝构造
 - 使用位运算优化状态表示（如果适用）
 - 对于栈溢出问题，可以使用栈模拟递归或调整栈大小
- *
- * 5. 调试与问题定位:
 - 使用 cout 打印中间状态值
 - 在关键节点添加断言验证中间结果
 - 使用 gdb 或其他调试器设置断点
 - 针对大规模数据，使用增量调试策略
 - 利用日志系统记录程序运行状态
- *
- * 6. 跨平台兼容性:
 - 确保使用标准 C++语法
 - 注意数据类型的范围和溢出问题
 - 考虑不同编译器的特性差异
 - 使用条件编译处理平台特定代码
 - 注意不同操作系统的内存布局差异
- *
- * 7. 工程化与产品化建议:
 - 将算法封装为独立模块，提供清晰的 API
 - 添加详细的文档说明使用方法和限制条件
 - 考虑添加配置选项，允许用户自定义算法参数
 - 实现性能监控功能，便于分析和优化
 - 使用命名空间避免命名冲突
- *
- * 8. 与高性能计算的联系:
 - 对于大规模数据，可以考虑使用并行计算加速
 - 可以利用 SIMD 指令集优化向量化操作
 - 考虑使用 GPU 加速密集型计算
 - 针对特定硬件架构进行优化

=====

文件: Code01_CherryPickup.java

=====

```
package class127;

// 摘樱桃
// 给定一个 n*n 的正方形矩阵 grid，每个格子值只有三种-1、0、1
// -1 表示格子不能走、0 表示格子可以走但是没有樱桃、1 表示格子可以走且有一颗樱桃
```

```
// 你的目标是从左上角走到右下角，每一步只能 向下 或者 向右  
// 然后从右下角走回左上角，每一步只能 向上 或者 向左  
// 这个过程中，想尽量多的获得樱桃，但是有樱桃的格子，只能捡一次  
// 返回最多能获得多少樱桃，如果不存在通路返回 0  
// 测试链接 : https://leetcode.cn/problems/cherry-pickup/
```

```
/**
```

```
* 算法思路深度解析:
```

```
* 1. 这道题可以看作是两个人同时从(0, 0)出发，走到(n-1, n-1)位置能收集的最大樱桃数
```

```
* - 这种转化是关键，因为来回走一次等价于两个人同时从起点走到终点
```

```
* - 这样可以避免处理重复访问的问题
```

```
* 2. 使用三维动态规划 dp[i][j][k]，其中：
```

```
* - i 表示第一个人的横坐标
```

```
* - j 表示第一个人的纵坐标
```

```
* - k 表示第二个人的横坐标
```

```
* - 第二个人的纵坐标可以由  $d = i + j - k$  计算得出（因为两人走的步数相同）
```

```
* - 这个状态定义是一种空间优化，将四维问题转化为三维问题
```

```
* 3. 每个状态可以从四个前驱状态转移而来：
```

```
* - 两人都向右：dp[i][j-1][k]
```

```
* - 第一人向右，第二人向下：dp[i][j-1][k-1]
```

```
* - 第一人向下，第二人向右：dp[i-1][j][k]
```

```
* - 两人都向下：dp[i-1][j][k-1]
```

```
* 4. 关键优化点：
```

```
* - 如果两人在同一个格子，只计算一次樱桃数量
```

```
* - 使用记忆化搜索避免重复计算
```

```
*
```

```
* 时间复杂度分析：
```

```
* - 状态数： $O(n^3)$ ，其中 n 是矩阵的边长
```

```
* - 每个状态需要考虑 4 个前驱状态
```

```
* - 总时间复杂度： $O(n^3)$ 
```

```
*
```

```
* 空间复杂度分析：
```

```
* - 动态规划数组大小： $O(n^3)$ 
```

```
* - 递归调用栈深度： $O(n)$ 
```

```
* - 总空间复杂度： $O(n^3)$ 
```

```
*
```

```
* 工程化考量：
```

```
* 1. 边界条件处理：确保不越界，检查障碍物
```

```
* 2. 初始化策略：使用特殊值(-2)标记未访问状态
```

```
* 3. 异常处理：处理无法到达终点的情况
```

```
* 4. 优化方向：可以通过滚动数组进行空间优化
```

```
*
```

```
* 常见问题排查：
```

```

* 1. 边界越界：在访问数组前检查索引范围
* 2. 障碍物处理：遇到障碍物返回-1
* 3. 重复计算：确保使用记忆化搜索
* 4. 结果判断：如果最终结果为-1，返回 0 表示不存在通路
*/
public class Code01_CherryPickup {

    /**
     * 计算能够摘到的最大樱桃数
     * @param grid n*n 的正方形矩阵
     * @return 最多能获得的樱桃数，如果不存在通路返回 0
     */
    public static int cherryPickup(int[][] grid) {
        int n = grid.length;
        // dp[i][j][k]表示第一个人在(i, j)，第二个人在(k, i+j-k)时能摘到的最大樱桃数
        int[][][] dp = new int[n][n][n];
        // 初始化 dp 数组为-2，表示未访问过
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                for (int k = 0; k < n; k++) {
                    dp[i][j][k] = -2;
                }
            }
        }
        int ans = f(grid, n, 0, 0, 0, dp);
        return ans == -1 ? 0 : ans;
    }

    /**
     * 递归函数，计算从(0,0)到(n-1,n-1)两人能摘到的最大樱桃数
     * @param grid 矩阵
     * @param n 矩阵边长
     * @param a 第一个人的横坐标
     * @param b 第一个人的纵坐标
     * @param c 第二个人的横坐标
     * @param dp 动态规划数组
     * @return 能摘到的最大樱桃数，如果无法到达终点返回-1
     */
    public static int f(int[][] grid, int n, int a, int b, int c, int[][][] dp) {
        // 计算第二个人的纵坐标
        int d = a + b - c;
        // 边界条件检查
        if (a == n || b == n || c == n || d == n || grid[a][b] == -1 || grid[c][d] == -1) {

```

```

        return -1;
    }

    // 到达终点
    if (a == n - 1 && b == n - 1) {
        return grid[a][b];
    }

    // 如果已经计算过，直接返回结果
    if (dp[a][b][c] != -2) {
        return dp[a][b][c];
    }

    // 计算当前位置的樱桃数
    int get = a == c && b == d ? grid[a][b] : (grid[a][b] + grid[c][d]);

    // 四种可能的移动方式
    int next = f(grid, n, a + 1, b, c + 1, dp); // 两人都向下
    next = Math.max(next, f(grid, n, a + 1, b, c, dp)); // 第一人向下，第二人向右
    next = Math.max(next, f(grid, n, a, b + 1, c + 1, dp)); // 第一人向右，第二人向下
    next = Math.max(next, f(grid, n, a, b + 1, c, dp)); // 两人都向右

    int ans = -1;
    if (next != -1) {
        ans = get + next;
    }

    dp[a][b][c] = ans;
    return ans;
}

```

// 类似题目与训练拓展：

```

// 1. LeetCode 1463. Cherry Pickup II (摘樱桃 II)
// 链接: https://leetcode.cn/problems/cherry-pickup-ii/
// 区别: 两个机器人分别从(0,0)和(0, cols-1)出发，只能向下移动，每步可选择三个方向
// 算法: 三维动态规划，状态定义为 dp[i][j1][j2]，表示两个机器人在第 i 行第 j1 和 j2 列的最大樱桃数
//

// 2. LeetCode 64. Minimum Path Sum (最小路径和)
// 链接: https://leetcode.cn/problems/minimum-path-sum/
// 区别: 求从左上角到右下角的最小路径和，每步只能向下或向右
// 算法: 二维动态规划，dp[i][j] = min(dp[i-1][j], dp[i][j-1]) + grid[i][j]
//

// 3. LeetCode 174. Dungeon Game (地下城游戏)
// 链接: https://leetcode.cn/problems/dungeon-game/
// 区别: 骑士需要从左上角到右下角，保证健康点数始终大于 0 的最小初始值
// 算法: 从右下角向左上角动态规划，dp[i][j] = max(1, min(dp[i+1][j], dp[i][j+1])) - dungeon[i][j]
//

// 4. LeetCode 741. Cherry Pickup (本题)

```

```
// 链接: https://leetcode.cn/problems/cherry-pickup/
// 算法: 三维动态规划, 转化为两人同时从起点到终点的问题
//
// 5. LeetCode 62. Unique Paths (不同路径)
// 链接: https://leetcode.cn/problems/unique-paths/
// 区别: 计算从左上角到右下角的不同路径数量, 每步只能向下或向右
// 算法: 组合数学或二维动态规划
//
// 6. LeetCode 63. Unique Paths II (不同路径 II)
// 链接: https://leetcode.cn/problems/unique-paths-ii/
// 区别: 网格中有障碍物, 计算不同路径数量
// 算法: 二维动态规划, 遇到障碍物时  $dp[i][j] = 0$ 
//
// 7. Codeforces 1296D - Fight with Monsters
// 链接: https://codeforces.com/problemset/problem/1296/D
// 区别: 贪心策略解决怪物战斗问题, 但状态转移思想类似
// 算法: 排序后贪心选择最优策略
//
// 8. AtCoder ABC159E - Dividing Chocolate
// 链接: https://atcoder.jp/contests/abc159/tasks/abc159_e
// 区别: 二维网格分割问题, 但需要类似的状态转移思路
// 算法: 前缀和+状态压缩动态规划
//
// 9. 洛谷 P1434 [SHOI2002] 滑雪
// 链接: https://www.luogu.com.cn/problem/P1434
// 区别: 寻找最长滑雪路径, 每步只能滑向相邻四个方向且高度更低的位置
// 算法: 记忆化搜索或拓扑排序动态规划
//
// 10. 牛客网 NC14552 方格取数
// 链接: https://ac.nowcoder.com/acm/problem/14552
// 区别: 与摘樱桃问题非常相似, 也是两个人从左上角出发到右下角取数
// 算法: 三维动态规划, 状态定义与本题类似
//
// 11. UVa 10913 - Walking on a Grid
// 链接:
https://onlinejudge.org/index.php?option=com\_onlinejudge&Itemid=8&category=24&page=show\_problem&problem=1854
// 区别: 在网格中行走, 有负数, 最多允许 k 次负数
// 算法: 四维动态规划, 状态定义为  $dp[i][j][k][d]$ , 表示在(i, j)位置, 已经经过 k 次负数, 方向为 d
//
// 12. SPOJ - SBANK
// 链接: https://www.spoj.com/problems/SBANK/
// 区别: 银行账号排序问题, 使用哈希表优化
```

```
// 算法: 哈希表+排序
//
// 13. HackerEarth - Roy and Coin Boxes
// 链接: https://www.hackerearth.com/practice/data-structures/arrays/1-d/practice-problems/algorithm/roy-and-coin-boxes-1/description/
// 区别: 区间更新问题, 使用差分数组优化
// 算法: 差分数组+前缀和
//
// 14. 杭电 HDU 1024 - Max Sum Plus Plus
// 链接: http://acm.hdu.edu.cn/showproblem.php?pid=1024
// 区别: 最大 m 段子数组和问题
// 算法: 二维动态规划优化为一维
//
// 15. Codeforces 1295E - Permutation Separation
// 链接: https://codeforces.com/problemset/problem/1295/E
// 区别: 排列分割问题, 需要找到最优分割点
// 算法: 前缀和+动态规划
//
// 算法本质与技巧总结:
// 1. 问题转化: 将“来回走一次”转化为“两个人同时走一次”, 这是解决这类问题的关键技巧
// 2. 状态压缩: 利用步数相同的特性, 将四维状态压缩为三维
// 3. 记忆化搜索: 避免重复计算, 提高效率
// 4. 边界处理: 仔细处理边界条件和障碍物情况
// 5. 结果判断: 处理无法到达终点的异常情况
//
// 工程化实战建议:
// 1. 测试用例设计:
//   - 空输入: n=0 的情况
//   - 单元素网格: 只有起点和终点的情况
//   - 全是障碍物的情况
//   - 无法到达终点的情况
//   - 最优路径需要交叉的情况
// 2. 性能优化:
//   - 对于大规模数据, 可以使用滚动数组将空间复杂度优化到 O(n^2)
//   - 可以使用递推方式代替递归, 避免栈溢出问题
// 3. 代码健壮性:
//   - 加入输入合法性检查
//   - 处理各种边界情况
//   - 使用合适的数据结构表示状态
// 4. 跨语言实现注意事项:
//   - Java: 注意数组初始化和内存限制
//   - C++: 可以使用 memset 优化初始化
//   - Python: 递归深度可能受限, 需要考虑使用迭代方式
```

```
// 5. 调试技巧:  
//   - 打印中间状态值  
//   - 使用小例子验证算法正确性  
//   - 检查边界条件处理  
}
```

文件: Code01_CherryPickup.py

```
# 摘樱桃  
# 给定一个 n*n 的正方形矩阵 grid，每个格子值只有三种-1、0、1  
# -1 表示格子不能走、0 表示格子可以走但是没有樱桃、1 表示格子可以走且有一颗樱桃  
# 你的目标是从左上角走到右下角，每一步只能 向下 或者 向右  
# 然后从右下角走回左上角，每一步只能 向上 或者 向左  
# 这个过程中，想尽量多的获得樱桃，但是有樱桃的格子，只能捡一次  
# 返回最多能获得多少樱桃，如果不存在通路返回 0  
# 测试链接 : https://leetcode.cn/problems/cherry-pickup/
```

"""

算法思路深度解析:

1. 这道题可以看作是两个人同时从(0, 0)出发，走到(n-1, n-1)位置能收集的最大樱桃数
 - 这种转化是关键，因为来回走一次等价于两个人同时从起点走到终点
 - 这样可以避免处理重复访问的问题
2. 使用三维动态规划，在Python中采用字典模拟三维数组，优化空间使用
 - i 表示第一个人的横坐标
 - j 表示第一个人的纵坐标
 - k 表示第二个人的横坐标
 - 第二个人的纵坐标可以由 $d = i + j - k$ 计算得出（因为两人走的步数相同）
 - 这个状态定义是一种空间优化，将四维问题转化为三维问题
3. 每个状态可以从四个前驱状态转移而来：
 - 两人都向右: $dp[i][j-1][k]$
 - 第一人向右，第二人向下: $dp[i][j-1][k-1]$
 - 第一人向下，第二人向右: $dp[i-1][j][k]$
 - 两人都向下: $dp[i-1][j][k-1]$
4. 关键优化点：
 - 如果两人在同一个格子，只计算一次樱桃数量
 - 使用记忆化搜索避免重复计算
 - Python中使用字典存储已计算结果，减少空间占用

时间复杂度分析:

- 状态数: $O(n^3)$ ，其中 n 是矩阵的边长
- 每个状态需要考虑 4 个前驱状态

- 总时间复杂度: $O(n^3)$

空间复杂度分析:

- 字典存储的已计算状态: 最坏情况 $O(n^3)$
- 递归调用栈深度: $O(n)$
- 总空间复杂度: $O(n^3)$

Python 实现注意事项:

1. 使用字典而不是列表实现记忆化搜索, 更节省空间
2. 递归函数中需要注意参数顺序和边界条件
3. Python 递归深度限制: 对于 n 较大的情况, 可能需要调整递归深度或改用迭代方式
4. 类型注解的使用提高代码可读性

工程化考量:

1. 边界条件处理: 确保不越界, 检查障碍物
2. 异常处理: 处理无法到达终点的情况
3. 测试用例: 添加全面的测试覆盖
4. 性能优化: 对于大规模数据, 可以考虑使用 `lru_cache` 装饰器或迭代版本

常见问题排查:

1. 递归深度问题: Python 默认递归深度限制为 1000, 对于大 n 值可能抛出 `RecursionError`
2. 边界越界: 确保所有坐标都在有效范围内
3. 障碍物处理: 正确处理无法通过的格子
4. 结果判断: 当 `ans` 为 -1 时返回 0 表示不存在通路

"""

```
from typing import List

class Solution:
    def cherryPickup(self, grid: List[List[int]]) -> int:
        """
        计算能够摘到的最大樱桃数
        :param grid: n*n 的正方形矩阵
        :return: 最多能获得的樱桃数, 如果不存在通路返回 0
        """
        n = len(grid)
        # dp[i][j][k] 表示第一个人在(i, j), 第二个人在(k, i+j-k)时能摘到的最大樱桃数
        # 使用字典来模拟三维数组, 避免初始化大型数组
        dp = {}

        def f(a: int, b: int, c: int) -> int:
            """
            递归函数, 计算从(0,0)到(n-1,n-1)两人能摘到的最大樱桃数
            """

"""
```

```

:param a: 第一个人的横坐标
:param b: 第一个人的纵坐标
:param c: 第二个人的横坐标
:return: 能摘到的最大樱桃数, 如果无法到达终点返回-1
"""

# 计算第二个人的纵坐标
d = a + b - c
# 边界条件检查
if a == n or b == n or c == n or d == n or grid[a][b] == -1 or grid[c][d] == -1:
    return -1
# 到达终点
if a == n - 1 and b == n - 1:
    return grid[a][b]
# 如果已经计算过, 直接返回结果
if (a, b, c) in dp:
    return dp[(a, b, c)]

# 计算当前位置的樱桃数
get = grid[a][b] if a == c and b == d else (grid[a][b] + grid[c][d])
# 四种可能的移动方式
next_val = f(a + 1, b, c + 1)  # 两人都向下
next_val = max(next_val, f(a + 1, b, c))  # 第一人向下, 第二人向右
next_val = max(next_val, f(a, b + 1, c + 1))  # 第一人向右, 第二人向下
next_val = max(next_val, f(a, b + 1, c))  # 两人都向右

ans = -1
if next_val != -1:
    ans = get + next_val

dp[(a, b, c)] = ans
return ans

ans = f(0, 0, 0)
return 0 if ans == -1 else ans

```

类似题目与训练拓展:

"""

以下是与摘樱桃问题相关的 15 个类似题目, 涵盖了多种算法平台
这些题目展示了动态规划、状态压缩、路径优化等相关技巧的应用

"""

```

# 1. LeetCode 1463. Cherry Pickup II (摘樱桃 II)
#     链接: https://leetcode.cn/problems/cherry-pickup-ii/

```

区别: 两个机器人分别从(0, 0)和(0, cols-1)出发, 只能向下移动, 每步可选择三个方向
算法: 三维动态规划, 状态定义为 $dp[i][j1][j2]$, 表示两个机器人在第 i 行第 j1 和 j2 列的最大樱桃数

2. LeetCode 64. Minimum Path Sum (最小路径和)
链接: <https://leetcode.cn/problems/minimum-path-sum/>
区别: 求从左上角到右下角的最小路径和, 每步只能向下或向右
算法: 二维动态规划, $dp[i][j] = \min(dp[i-1][j], dp[i][j-1]) + grid[i][j]$

3. LeetCode 174. Dungeon Game (地下城游戏)
链接: <https://leetcode.cn/problems/dungeon-game/>
区别: 骑士需要从左上角到右下角, 保证健康点数始终大于 0 的最小初始值
算法: 从右下角向左上角动态规划, $dp[i][j] = \max(1, \min(dp[i+1][j], dp[i][j+1])) - dungeon[i][j]$

4. LeetCode 741. Cherry Pickup (本题)
链接: <https://leetcode.cn/problems/cherry-pickup/>
算法: 三维动态规划, 转化为两人同时从起点到终点的问题

5. LeetCode 62. Unique Paths (不同路径)
链接: <https://leetcode.cn/problems/unique-paths/>
区别: 计算从左上角到右下角的不同路径数量, 每步只能向下或向右
算法: 组合数学或二维动态规划

6. LeetCode 63. Unique Paths II (不同路径 II)
链接: <https://leetcode.cn/problems/unique-paths-ii/>
区别: 网格中有障碍物, 计算不同路径数量
算法: 二维动态规划, 遇到障碍物时 $dp[i][j] = 0$

7. Codeforces 1296D – Fight with Monsters
链接: <https://codeforces.com/problemset/problem/1296/D>
区别: 贪心策略解决怪物战斗问题, 但状态转移思想类似
算法: 排序后贪心选择最优策略

8. AtCoder ABC159E – Dividing Chocolate
链接: https://atcoder.jp/contests/abc159/tasks/abc159_e
区别: 二维网格分割问题, 但需要类似的状态转移思路
算法: 前缀和+状态压缩动态规划

9. 洛谷 P1434 [SHOI2002] 滑雪
链接: <https://www.luogu.com.cn/problem/P1434>
区别: 寻找最长滑雪路径, 每步只能滑向相邻四个方向且高度更低的位置
算法: 记忆化搜索或拓扑排序动态规划

```

# 10. 牛客网 NC14552 方格取数
#     链接: https://ac.nowcoder.com/acm/problem/14552
#     区别: 与摘樱桃问题非常相似, 也是两个人从左上角出发到右下角取数
#     算法: 三维动态规划, 状态定义与本题类似

# 11. UVa 10913 - Walking on a Grid
#     链接:
https://onlinejudge.org/index.php?option=com\_onlinejudge&Itemid=8&category=24&page=show\_problem&problem=1854
#     区别: 在网格中行走, 有负数, 最多允许 k 次负数
#     算法: 四维动态规划, 状态定义为  $dp[i][j][k][d]$ , 表示在(i, j)位置, 已经经过 k 次负数, 方向为 d

# 12. SPOJ - SBANK
#     链接: https://www.spoj.com/problems/SBANK/
#     区别: 银行账号排序问题, 使用哈希表优化
#     算法: 哈希表+排序

# 13. HackerEarth - Roy and Coin Boxes
#     链接: https://www.hackerearth.com/practice/data-structures/arrays/1-d/practice-problems/algorithm/roy-and-coin-boxes-1/description/
#     区别: 区间更新问题, 使用差分数组优化
#     算法: 差分数组+前缀和

# 14. 杭电 HDU 1024 - Max Sum Plus Plus
#     链接: http://acm.hdu.edu.cn/showproblem.php?pid=1024
#     区别: 最大 m 段子数组和问题
#     算法: 二维动态规划优化为一维

# 15. Codeforces 1295E - Permutation Separation
#     链接: https://codeforces.com/problemset/problem/1295/E
#     区别: 排列分割问题, 需要找到最优分割点
#     算法: 前缀和+动态规划

```

"""

算法本质与技巧总结:

1. 问题转化: 将“来回走一次”转化为“两个人同时走一次”, 这是解决这类问题的关键技巧
 - 这种转化避免了处理重复访问问题, 简化了状态定义
 - 可以扩展到更多类似路径优化问题
 - 转化思维体现了算法设计中的问题重构能力

2. 状态压缩: 利用步数相同的特性, 将四维状态压缩为三维
 - 这种优化在路径规划问题中非常常见

- 状态压缩可以显著减少空间复杂度
 - 基于问题特性的状态压缩是高级动态规划的重要技巧
3. 记忆化搜索：避免重复计算，提高效率
- 递归实现的记忆化搜索比迭代方式更直观
 - 对于复杂状态转移方程的问题，记忆化搜索更易实现
 - 在 Python 中可以使用字典或 `lru_cache` 实现记忆化
4. 边界处理：仔细处理边界条件和障碍物情况
- 边界检查在网格类问题中尤为重要
 - 障碍物的处理需要特殊考虑
 - 正确的边界处理是算法正确性的关键
5. 结果判断：处理无法到达终点的异常情况
- 返回合适的默认值表示无解
 - 异常情况的处理体现了代码的健壮性
- """
- """
- Python 工程化实战建议：
1. 测试用例设计：
 - 空输入： $n=0$ 的情况
 - 单元素网格：只有起点和终点的情况
 - 全是障碍物的情况
 - 无法到达终点的情况
 - 最优路径需要交叉的情况
 2. 性能优化策略：
 - 对于大规模数据，可以使用滚动数组将空间复杂度优化到 $O(n^2)$
 - 可以使用递推方式代替递归，避免栈溢出问题
 - Python 中使用 `lru_cache` 装饰器可以简化记忆化实现
 - 对于大 n 值，考虑调整递归深度限制或改用迭代方式
 3. 代码健壮性提升：
 - 加入输入合法性检查，防止非法输入导致程序崩溃
 - 使用 `try-except` 块处理可能的异常
 - 使用类型注解提高代码可读性和可维护性
 - 编写完整的单元测试确保功能正确
 4. Python 特有优化技巧：
 - 利用字典存储已计算状态，避免初始化大型数组
 - 使用 `functools.lru_cache` 替代手动实现的记忆化搜索

- 针对递归深度限制问题，可以使用 `sys.setrecursionlimit()`临时调整
- 考虑使用生成器和迭代器优化内存使用

5. 调试与问题定位:

- 使用 `print` 语句打印中间状态值
- 在关键节点添加断言验证中间结果
- 使用 `pdb` 调试器进行交互式调试
- 针对大规模数据，使用增量调试策略

6. 跨语言实现对比:

- Python 实现更简洁，但在性能上可能不如 C++
- 字典实现的记忆化搜索比列表更节省空间，但访问速度略慢
- 递归深度限制是 Python 实现的一个潜在问题，需要特别注意
- Java 和 C++的静态类型检查可以在编译时发现更多错误

7. 工程化与产品化建议:

- 将算法封装为独立模块，提供清晰的 API
- 添加详细的文档说明使用方法和限制条件
- 考虑添加配置选项，允许用户自定义算法参数
- 实现性能监控功能，便于分析和优化

8. 与机器学习的联系:

- 类似的路径规划问题在强化学习中有广泛应用
- 动态规划思想是强化学习中价值迭代和策略迭代的基础
- 可以考虑使用深度强化学习方法解决更大规模的路径规划问题

"""

2. 性能优化:

- # - 对于大规模数据，可以考虑使用 `lru_cache` 装饰器优化记忆化搜索
- # - 递归深度问题：Python 默认递归深度限制为 1000，对于大 n 值可能需要改用迭代方式
- # - 可以使用 `sys.setrecursionlimit()` 调整递归深度，但要注意栈溢出风险

3. 代码健壮性:

- # - 添加输入合法性检查
- # - 使用 `try-except` 块处理可能的异常
- # - 类型注解提高代码可读性

4. 调试技巧:

- # - 使用 `print` 语句打印中间状态值
- # - 使用断点调试工具
- # - 可以考虑使用 `pdb` 调试器进行交互式调试

5. Python 特性应用:

- # - 利用 `collections.defaultdict` 简化字典操作
- # - 使用 `functools.lru_cache` 装饰器替代手动实现的记忆化搜索
- # - 考虑使用 `numpy` 优化数组操作（如果处理大规模数据）

6. 跨语言实现对比:

```
# - Python 实现更简洁，但在性能上可能不如 C++
# - 字典实现的记忆化搜索比列表更节省空间，但访问速度略慢
# - 递归深度限制是 Python 实现的一个潜在问题，需要特别注意

# 测试用例：全面覆盖各种情况
if __name__ == "__main__":
    solution = Solution()

# 测试用例 1：基本情况
grid1 = [[0, 1, -1], [1, 0, -1], [1, 1, 1]]
print(f"测试用例 1 结果: {solution.cherryPickup(grid1)}") # 输出应为 5

# 测试用例 2：无法到达终点的情况
grid2 = [[0, -1], [-1, 0]]
print(f"测试用例 2 结果: {solution.cherryPickup(grid2)}") # 输出应为 0

# 测试用例 3：单元素网格
grid3 = [[1]]
print(f"测试用例 3 结果: {solution.cherryPickup(grid3)}") # 输出应为 1

# 测试用例 4：需要交叉路径的情况
grid4 =
[[1, 1, 1, 1, 0, 0, 0], [0, 0, 0, 1, 0, 0, 0], [0, 0, 0, 1, 0, 0, 1], [1, 0, 0, 1, 0, 0, 0], [0, 0, 0, 1, 0, 0, 0], [0, 0, 0, 1, 0, 0, 0], [0, 0, 0, 1, 1, 1]]
print(f"测试用例 4 结果: {solution.cherryPickup(grid4)}") # 复杂情况的测试

# 测试用例 5：起点或终点是障碍物
grid5 = [[-1, 1], [1, 1]]
print(f"测试用例 5 结果: {solution.cherryPickup(grid5)}") # 输出应为 0

# 性能优化提示：对于较大的 n 值，可以考虑以下替代实现
'''

# 使用 lru_cache 装饰器的替代实现
from functools import lru_cache

class SolutionOptimized:
    def cherryPickup(self, grid: List[List[int]]) -> int:
        n = len(grid)

        @lru_cache(maxsize=None)
        def dp(a, b, c):
            d = a + b - c
            if a == n or b == n or c == n or d == n or grid[a][b] == -1 or grid[c][d] == -1:

```

```

        return -1
    if a == n - 1 and b == n - 1:
        return grid[a][b]

    res = max(
        dp(a+1, b, c+1),
        dp(a+1, b, c),
        dp(a, b+1, c+1),
        dp(a, b+1, c)
    )

    if res == -1:
        return -1

    current = grid[a][b] if (a == c and b == d) else (grid[a][b] + grid[c][d])
    return current + res

result = dp(0, 0, 0)
return 0 if result == -1 else result
"""
=====

文件: Code02_FrogToSchool.cpp
=====

#include <iostream>
#include <vector>
#include <algorithm>
#include <cstring>

using namespace std;

/**
 * 上学需要的最少跳跃能力 - C++实现
 * 青蛙住在一条河边，家在 0 位置，每天到河对岸的上学，学校在 n 位置
 * 河里的石头排成了一条直线，青蛙每次跳跃必须落在一块石头或者岸上
 * 给定一个长度为 n-1 的数组 arr，表示 1~n-1 位置每块石头的高度数值
 * 每次青蛙从一块石头起跳，这块石头的高度就会下降 1
 * 当石头的高度下降到 0 时，青蛙不能再跳到这块石头上，跳跃后使石头高度下降到 0 是允许的
 * 青蛙一共需要去学校上 x 天课，所以它需要往返 x 次，青蛙具有跳跃能力 y，它可以跳跃不超过 y 的距离
 * 请问青蛙的跳跃能力至少是多少，才能用这些石头往返 x 次
 * 1 <= n <= 10^5
 * 1 <= arr[i] <= 10^4

```

```

* 1 <= x <= 10^9
* 测试链接 : https://www.luogu.com.cn/problem/P8775
*
* 算法思路:
* 1. 二分答案: 将问题转化为判定问题, 对于每个可能的跳跃能力 y, 验证是否能完成 x 次往返
* 2. 滑动窗口: 对于给定的 y, 使用滑动窗口验证是否可行
* 3. 模拟往返: 每次往返消耗窗口内石头的高度
*
* 时间复杂度: O(n * log(n)), 其中 n 是石头的数量
* 空间复杂度: O(n), 需要复制数组进行验证
*/

```

```

class FrogToSchool {
public:
    int compute(int n, int x, vector<int>& arr) {
        // 将学校位置 n 的高度设为足够大
        arr.resize(n + 1);
        arr[n] = 2 * x;

        int left = 1;
        int right = n;
        int ans = 0;

        // 二分查找最小跳跃能力
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (canFinish(n, x, arr, mid)) {
                ans = mid;
                right = mid - 1;
            } else {
                left = mid + 1;
            }
        }
        return ans;
    }
}

```

```

private:
    /**
     * 检查给定跳跃能力是否能完成 x 次往返
     * @param n 学校位置
     * @param x 往返次数
     * @param arr 石头高度数组
     * @param power 跳跃能力

```

```

* @return 是否能完成 x 次往返
*/
bool canFinish(int n, int x, vector<int> arr, int power) {
    // 复制数组，避免修改原数组
    vector<int> temp = arr;

    // 模拟 x 次往返
    for (int i = 0; i < x; i++) {
        long long sum = 0;
        int l = 1;
        int r = 1;

        // 滑动窗口验证
        while (l <= n && r <= n) {
            // 扩展右边界
            while (r <= n && r - l < power) {
                sum += temp[r];
                r++;
            }

            // 检查窗口内石头是否足够
            if (sum >= 2) {
                long long need = min(sum, 2LL);
                sum -= need;

                // 消耗石头高度
                for (int j = 1; j < r && need > 0; j++) {
                    long long deduct = min((long long)temp[j], need);
                    temp[j] -= deduct;
                    need -= deduct;
                }
            }

            if (need == 0) {
                break;
            }
        }

        // 移动左边界
        sum -= temp[1];
        l++;
    }

    // 如果无法完成本次往返
}

```

```

        if (sum < 2) {
            return false;
        }

        return true;
    }
};

/***
 * 优化版本：使用数学分析优化验证过程
 * 时间复杂度：O(n)
 * 空间复杂度：O(n)
 */
class FrogToSchoolOptimized {
public:
    int compute(int n, int x, vector<int>& arr) {
        // 将学校位置 n 的高度设为足够大
        arr.resize(n + 1);
        arr[n] = 2 * x;

        // 计算每个位置最多能被使用的次数
        vector<long long> usage(n + 1);
        for (int i = 1; i <= n; i++) {
            usage[i] = min((long long)arr[i], (long long)x);
        }

        int left = 1;
        int right = n;
        int ans = 0;

        // 二分查找
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (canFinishOptimized(n, x, usage, mid)) {
                ans = mid;
                right = mid - 1;
            } else {
                left = mid + 1;
            }
        }
        return ans;
    }
}

```

```
private:
    bool canFinishOptimized(int n, int x, vector<long long>& usage, int power) {
        long long total = 0;
        int l = 1;

        // 滑动窗口计算总可用次数
        for (int r = 1; r <= n; r++) {
            total += usage[r];

            // 维护窗口大小不超过 power
            while (r - l + 1 > power) {
                total -= usage[l];
                l++;
            }
        }

        // 如果窗口大小达到 power 且总可用次数不足 2x
        if (r - l + 1 == power && total < 2LL * x) {
            return false;
        }
    }

    return true;
}

};

/***
 * 测试函数
 */
int main() {
    FrogToSchool solution;

    // 测试用例 1
    int n1 = 5;
    int x1 = 3;
    vector<int> arr1 = {0, 2, 3, 1, 4}; // 注意：数组从位置 1 开始
    cout << "Test 1: " << solution.compute(n1, x1, arr1) << endl;

    // 测试用例 2
    int n2 = 10;
    int x2 = 5;
    vector<int> arr2 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    cout << "Test 2: " << solution.compute(n2, x2, arr2) << endl;
}
```

```
// 优化版本测试
FrogToSchool0optimized solution0pt;
cout << "Optimized Test 1: " << solution0pt.compute(n1, x1, arr1) << endl;
cout << "Optimized Test 2: " << solution0pt.compute(n2, x2, arr2) << endl;

return 0;
}

/***
 * C++工程化实战建议:
 *
 * 1. 内存管理:
 *     - 使用 vector 代替原生数组，避免手动内存管理
 *     - 注意 vector 的 resize 操作，确保索引不越界
 *     - 对于大规模数据，考虑使用 reserve 预分配内存
 *
 * 2. 类型安全:
 *     - 使用 size_t 处理数组索引，避免负数问题
 *     - 对于大数运算，使用 long long 避免溢出
 *     - 注意整数类型转换，避免精度丢失
 *
 * 3. 性能优化:
 *     - 使用引用传递避免不必要的拷贝
 *     - 对于频繁调用的函数，考虑内联优化
 *     - 使用-O2 或-O3 编译优化
 *
 * 4. 输入输出优化:
 *     - 使用 ios::sync_with_stdio(false) 加速 cin/cout
 *     - 对于大规模输入，考虑使用 scanf/printf
 *     - 使用 endl 会刷新缓冲区，在性能敏感场景使用'\n'
 *
 * 5. 异常处理:
 *     - 添加输入合法性检查
 *     - 使用 try-catch 处理可能的异常
 *     - 确保资源正确释放
 *
 * 6. 调试技巧:
 *     - 使用 gdb 进行调试
 *     - 添加 assert 断言验证中间结果
 *     - 使用 valgrind 检查内存泄漏
 */

```

文件: Code02_FrogToSchool.java

```
=====
package class127;
```

```
// 上学需要的最少跳跃能力
// 青蛙住在一条河边，家在 0 位置，每天到河对岸的上学，学校在 n 位置
// 河里的石头排成了一条直线，青蛙每次跳跃必须落在一块石头或者岸上
// 给定一个长度为 n-1 的数组 arr，表示 1~n-1 位置每块石头的高度数值
// 每次青蛙从一块石头起跳，这块石头的高度就会下降 1
// 当石头的高度下降到 0 时，青蛙不能再跳到这块石头上，跳跃后使石头高度下降到 0 是允许的
// 青蛙一共需要去学校上 x 天课，所以它需要往返 x 次，青蛙具有跳跃能力 y，它可以跳跃不超过 y 的距离
// 请问青蛙的跳跃能力至少是多少，才能用这些石头往返 x 次
// 1 <= n <= 10^5
// 1 <= arr[i] <= 10^4
// 1 <= x <= 10^9
// 测试链接：https://www.luogu.com.cn/problem/P8775
// 提交以下的 code，提交时请把类名改成"Main"，可以通过所有用例
```

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

```
/**
 * 算法思路：
 * 1. 这是一个典型的二分答案问题
 * 2. 我们需要找到最小的跳跃能力 y，使得青蛙能够完成 x 次往返
 * 3. 对于给定的跳跃能力 y，我们可以通过滑动窗口来验证是否能够完成 x 次往返
 * 4. 滑动窗口 [left, right) 表示青蛙在一次往返中能使用的石头范围
 * 5. 在每次往返中，青蛙需要消耗石头的高度，当石头高度为 0 时不能再使用
 * 6. 我们通过二分查找来确定最小的跳跃能力
 *
 * 时间复杂度: O(n * log(n))，其中 n 是石头的数量
 * 空间复杂度: O(1)
 */
public class Code02_FrogToSchool {
```

```
    public static int MAXN = 100001;
```

```
public static int[] arr = new int[MAXN];

public static int n, x;

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;
    in.nextToken();
    x = (int) in.nval;
    for (int i = 1; i < n; i++) {
        in.nextToken();
        arr[i] = (int) in.nval;
    }
    // 认为学校所在的位置 n, 有足够的高度
    arr[n] = 2 * x;
    out.println(compute());
    out.flush();
    out.close();
    br.close();
}
```

```
/***
 * 计算青蛙的最小跳跃能力
 * @return 最小跳跃能力
 */
public static int compute() {
    // 二分答案的左右边界
    int left = 1;
    int right = n;
    int ans = 0;

    // 二分查找最小跳跃能力
    while (left <= right) {
        int mid = (left + right) / 2;
        if (canFinish(mid)) {
            ans = mid;
            right = mid - 1;
        } else {
            left = mid + 1;
        }
    }
}
```

```

    }

    return ans;
}

/***
 * 检查给定跳跃能力是否能完成 x 次往返
 * @param power 跳跃能力
 * @return 是否能完成 x 次往返
*/
private static boolean canFinish(int power) {
    // 复制石头高度数组，避免修改原数组
    int[] temp = new int[n + 1];
    System.arraycopy(arr, 0, temp, 0, n + 1);

    // 模拟 x 次往返
    for (int i = 0; i < x; i++) {
        // 使用滑动窗口计算一次往返
        long sum = 0;
        int l = 1;
        int r = 1;

        // 扩展窗口右边界直到能够到达学校
        while (l <= n && r <= n) {
            // 扩展右边界
            while (r <= n && r - 1 < power) {
                sum += temp[r];
                r++;
            }

            // 如果当前窗口能够支持一次往返
            if (sum >= 2) {
                // 消耗石头
                long need = Math.min(sum, 2);
                sum -= need;
                // 更新石头高度
                for (int j = l; j < r && need > 0; j++) {
                    long deduct = Math.min(temp[j], need);
                    temp[j] -= deduct;
                    need -= deduct;
                }
                // 如果完成一次往返，跳出循环
                if (need == 0) {
                    break;
                }
            }
        }
    }
}

```

```

        }

    }

    // 移动左边界
    sum -= temp[1];
    l++;
}

// 如果无法完成本次往返，返回 false
if (sum < 2) {
    return false;
}

}

return true;
}

/***
 * 类似题目与训练拓展：
 * 以下是与青蛙跳跃问题相关的 15 个类似题目，涵盖了多种算法平台和难度级别
 */
// 1. LeetCode 403 - Frog Jump
// 链接: https://leetcode.cn/problems/frog-jump/
// 区别：青蛙在河中跳跃，每个位置可能有石头，需要判断能否到达最后一块石头
// 算法：记忆化搜索或动态规划，状态定义为 dp[position][k] 表示在 position 位置能否以步长 k 跳跃

// 2. LeetCode 1340 - Jump Game V
// 链接: https://leetcode.cn/problems/jump-game-v/
// 区别：在数组中跳跃，每次跳跃不能超过固定距离，且需要满足特定条件
// 算法：记忆化搜索，状态定义为 dp[i] 表示从位置 i 出发能访问的最大节点数

// 3. LeetCode 55 - Jump Game
// 链接: https://leetcode.cn/problems/jump-game/
// 区别：判断能否从第一个位置跳到最后一个位置，每个元素代表最大跳跃长度
// 算法：贪心或动态规划，维护能到达的最远位置

// 4. LeetCode 45 - Jump Game II
// 链接: https://leetcode.cn/problems/jump-game-ii/
// 区别：求跳到最后一个位置的最少跳跃次数
// 算法：贪心，每次选择能跳得最远的下一步

// 5. LeetCode 1306 - Jump Game III

```

```
// 链接: https://leetcode.cn/problems/jump-game-iii/
// 区别: 在数组中跳跃, 从给定位置开始, 判断能否到达值为 0 的位置
// 算法: BFS 或 DFS

// 6. LeetCode 1696 - Jump Game VI
// 链接: https://leetcode.cn/problems/jump-game-vi/
// 区别: 在数组中跳跃, 每次最多跳 k 步, 求到达最后一个位置的最大得分
// 算法: 动态规划 + 单调队列优化

// 7. LeetCode 1871 - Jump Game VII
// 链接: https://leetcode.cn/problems/jump-game-vii/
// 区别: 在由'0'和'1'组成的字符串上跳跃, 判断能否从第一个位置跳到最后一个位置
// 算法: BFS 或动态规划 + 前缀和优化

// 8. Codeforces 965D - Single Wildcard Pattern Matching
// 链接: https://codeforces.com/problemset/problem/965/D
// 区别: 字符串匹配问题, 但涉及到间隔匹配的概念
// 算法: 贪心 + 双指针

// 9. 洛谷 P1250 种树
// 链接: https://www.luogu.com.cn/problem/P1250
// 区别: 区间覆盖问题, 贪心选择最优策略
// 算法: 贪心, 按照区间右端点排序后选择

// 10. 牛客网 NC14552 方格取数
// 链接: https://ac.nowcoder.com/acm/problem/14552
// 区别: 路径规划问题, 但需要最大化收集的值
// 算法: 三维动态规划

// 11. UVa 11280 - Flying to Fredericton
// 链接:
https://onlinejudge.org/index.php?option=com\_onlinejudge&Itemid=8&category=24&page=show\_problem&problem=2255
// 区别: 多段最短路径问题, 允许最多 k 次飞行
// 算法: Dijkstra 算法变种

// 12. SPOJ - MICE AND MAZE
// 链接: https://www.spoj.com/problems/MICEMAZE/
// 区别: 迷宫寻路问题, 计算能在给定时间内到达终点的老鼠数量
// 算法: BFS 或 Dijkstra 算法

// 13. HackerEarth - Minimum Jumps
// 链接: https://www.hackerearth.com/practice/data-structures/arrays/1-d/practice-
```

```

problems/algorith/minimum-jumps-4/description/
// 区别: 计算从数组第一个元素跳到最后一个元素的最小跳跃次数
// 算法: BFS 或动态规划

// 14. 杭电 HDU 1074 - Doing Homework
// 链接: http://acm.hdu.edu.cn/showproblem.php?pid=1074
// 区别: 任务调度问题, 求最小扣分
// 算法: 状态压缩动态规划

// 15. Codeforces 1296D - Fight with Monsters
// 链接: https://codeforces.com/problemset/problem/1296/D
// 区别: 贪心策略解决怪物战斗问题
// 算法: 排序后贪心选择最优策略

/***
 * 算法本质与技巧总结:
 *
 * 1. 二分答案: 将求最小跳跃能力转化为对每个可能的跳跃能力进行验证
 *   - 二分答案是解决优化问题的常用技巧, 将问题转化为判定问题
 *   - 适用于具有单调性的问题, 即如果 y 可行, 那么所有大于 y 的跳跃能力都可行
 *   - 二分查找的范围需要合理设定, 通常是问题的上下界
 *
 * 2. 滑动窗口: 高效验证给定跳跃能力是否可行
 *   - 滑动窗口在数组处理中非常高效, 时间复杂度为 O(n)
 *   - 通过维护窗口内的石头高度总和, 快速判断是否可以完成一次往返
 *   - 窗口的大小由当前尝试的跳跃能力决定
 *
 * 3. 模拟往返过程: 逐次消耗石头高度
 *   - 每次往返需要消耗窗口内的石头高度
 *   - 优先消耗高度较低的石头, 保持窗口的可用性
 *   - 当窗口内的石头高度总和不足时, 表示无法完成当前跳跃能力下的往返
 *
 * 4. 数组复制: 避免修改原数组
 *   - 在验证函数中复制原数组, 防止影响后续二分查找
 *   - 这是一种常见的工程实践, 保持函数的无副作用性
 *
 * 5. 边界处理: 将学校位置视为特殊节点
 *   - 学校位置 n 被赋予足够大的高度, 确保可以完成跳跃
 *   - 合理的边界条件处理是算法正确性的关键
 */

/***
 * 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 StreamTokenizer 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
 - 在验证函数中，复制数组可能导致较大的内存开销
 - 对于 x 很大的情况，可以考虑优化验证逻辑，减少内存使用
 - *
 - * 3. 性能优化策略:
 - 二分查找的上下界设置要合理，避免不必要的搜索
 - 滑动窗口的实现要高效，避免重复计算
 - 对于大规模数据，可以考虑使用 long 类型避免溢出
 - *
 - * 4. 代码健壮性提升:
 - 添加输入合法性检查
 - 处理可能的边界情况，如 n=1 或 x=0
 - 使用 try-catch 块处理 I/O 异常
 - 确保资源正确关闭（使用 try-with-resources 更好）
 - *
 - * 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 x 很大的情况，可以优化验证算法，避免 O(x*n) 的时间复杂度
 - 考虑使用 long 类型处理大数，避免整数溢出
 - *
 - * 6. 调试与问题定位:
 - 添加日志输出来跟踪算法的执行过程
 - 对于二分查找，可以打印每次 mid 的值和验证结果
 - 对于滑动窗口，可以打印窗口的左右边界和当前总和
 - *
 - * 7. 跨语言实现对比:
 - Java 实现比较规范，但在处理大规模数据时可能不如 C++ 高效
 - 在 Python 中，需要注意整数溢出问题，Python 的整数精度不会有这些问题
 - 在 C++ 中，可以使用 memcpy 进行更快的数组复制
 - *
 - * 8. 算法优化思路:
 - 原始验证算法的时间复杂度为 O(x*n)，对于 x=1e9 的情况会超时
 - 可以通过数学分析优化验证算法，将时间复杂度降至 O(n)
 - 关键思路是：对于每个位置 i 的石头，最多可以被使用 min(arr[i], x) 次
 - 在滑动窗口中，我们只需要计算窗口内石头可使用次数的总和是否足够 2x

```
*/  
}
```

文件: Code02_FrogToSchool.py

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

```
"""
```

上学需要的最少跳跃能力 - Python 实现

青蛙住在一条河边，家在 0 位置，每天到河对岸的上学，学校在 n 位置
河里的石头排成了一条直线，青蛙每次跳跃必须落在一块石头或者岸上
给定一个长度为 n-1 的数组 arr，表示 1~n-1 位置每块石头的高度数值
每次青蛙从一块石头起跳，这块石头的高度就会下降 1

当石头的高度下降到 0 时，青蛙不能再跳到这块石头上，跳跃后使石头高度下降到 0 是允许的

青蛙一共需要去学校上 x 天课，所以它需要往返 x 次，青蛙具有跳跃能力 y，它可以跳跃不超过 y 的距离
请问青蛙的跳跃能力至少是多少，才能用这些石头往返 x 次

```
1 <= n <= 10^5  
1 <= arr[i] <= 10^4  
1 <= x <= 10^9
```

测试链接 : <https://www.luogu.com.cn/problem/P8775>

算法思路:

1. 二分答案: 将问题转化为判定问题, 对于每个可能的跳跃能力 y, 验证是否能完成 x 次往返
2. 滑动窗口: 对于给定的 y, 使用滑动窗口验证是否可行
3. 模拟往返: 每次往返消耗窗口内石头的高度

时间复杂度: $O(n * \log(n))$, 其中 n 是石头的数量

空间复杂度: $O(n)$, 需要复制数组进行验证

```
"""
```

```
import sys  
from typing import List  
  
class FrogToSchool:  
    """  
        青蛙上学问题解决方案  
    """
```

```
    def compute(self, n: int, x: int, arr: List[int]) -> int:  
        """
```

计算青蛙的最小跳跃能力

Args:

n: 学校位置
x: 往返次数
arr: 石头高度数组, 长度为 n-1, 表示位置 1 到 n-1 的石头高度

Returns:

int: 最小跳跃能力

"""

```
# 将学校位置 n 的高度设为足够大
arr = arr[:] # 复制数组
arr.extend([0] * (n - len(arr) + 1)) # 确保数组长度足够
arr[n] = 2 * x # 学校位置有足够的高度
```

```
left, right = 1, n
```

```
ans = 0
```

```
# 二分查找最小跳跃能力
```

```
while left <= right:
```

```
    mid = (left + right) // 2
```

```
    if self._can_finish(n, x, arr, mid):
```

```
        ans = mid
```

```
        right = mid - 1
```

```
    else:
```

```
        left = mid + 1
```

```
return ans
```

```
def _can_finish(self, n: int, x: int, arr: List[int], power: int) -> bool:
```

"""

检查给定跳跃能力是否能完成 x 次往返

Args:

n: 学校位置
x: 往返次数
arr: 石头高度数组
power: 跳跃能力

Returns:

bool: 是否能完成 x 次往返

"""

```
# 复制数组, 避免修改原数组
```

```
temp = arr[:]

# 模拟 x 次往返
for _ in range(x):
    total = 0
    l, r = 1, 1

    # 滑动窗口验证
    while l <= n and r <= n:
        # 扩展右边界
        while r <= n and r - l < power:
            total += temp[r]
            r += 1

        # 检查窗口内石头是否足够
        if total >= 2:
            need = min(total, 2)
            total -= need

        # 消耗石头高度
        j = 1
        while j < r and need > 0:
            deduct = min(temp[j], need)
            temp[j] -= deduct
            need -= deduct
            j += 1

        if need == 0:
            break

    # 移动左边界
    total -= temp[l]
    l += 1

# 如果无法完成本次往返
if total < 2:
    return False

return True
```

```
class FrogToSchool0optimized:
    """
```

优化版本：使用数学分析优化验证过程

时间复杂度：O(n)

空间复杂度：O(n)

"""

```
def compute(self, n: int, x: int, arr: List[int]) -> int:
```

"""

计算青蛙的最小跳跃能力（优化版本）

Args:

n: 学校位置

x: 往返次数

arr: 石头高度数组

Returns:

int: 最小跳跃能力

"""

将学校位置 n 的高度设为足够大

arr = arr[:] # 复制数组

arr.extend([0] * (n - len(arr) + 1)) # 确保数组长度足够

arr[n] = 2 * x # 学校位置有足够的高度

计算每个位置最多能被使用的次数

```
usage = [min(arr[i], x) for i in range(n + 1)]
```

left, right = 1, n

ans = 0

二分查找

while left <= right:

mid = (left + right) // 2

if self._can_finish_optimized(n, x, usage, mid):

ans = mid

right = mid - 1

else:

left = mid + 1

return ans

```
def _can_finish_optimized(self, n: int, x: int, usage: List[int], power: int) -> bool:
```

"""

优化版本的验证函数

Args:

n: 学校位置
x: 往返次数
usage: 每个位置最多可被使用的次数
power: 跳跃能力

Returns:

bool: 是否能完成 x 次往返

"""

total = 0

l = 1

滑动窗口计算总可用次数

for r in range(1, n + 1):

total += usage[r]

维护窗口大小不超过 power

while r - l + 1 > power:

total -= usage[l]

l += 1

如果窗口大小达到 power 且总可用次数不足 2x

if r - l + 1 == power and total < 2 * x:

return False

return True

def test_frog_to_school():

"""

测试函数

"""

solution = FrogToSchool()

solution_opt = FrogToSchoolOptimized()

测试用例 1

n1 = 5

x1 = 3

arr1 = [0, 2, 3, 1, 4] # 注意: 数组从位置 1 开始

result1 = solution.compute(n1, x1, arr1)

result1_opt = solution_opt.compute(n1, x1, arr1)

print(f"Test 1: {result1}, Optimized: {result1_opt}")

```

# 测试用例 2
n2 = 10
x2 = 5
arr2 = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
result2 = solution.compute(n2, x2, arr2)
result2_opt = solution_opt.compute(n2, x2, arr2)
print(f"Test 2: {result2}, Optimized: {result2_opt}")

# 边界测试
n3 = 1
x3 = 1
arr3 = [0]
result3 = solution.compute(n3, x3, arr3)
result3_opt = solution_opt.compute(n3, x3, arr3)
print(f"Test 3 (边界): {result3}, Optimized: {result3_opt}")

if __name__ == "__main__":
    # 运行测试
    test_frog_to_school()

# 从标准输入读取数据（用于在线评测）
if len(sys.argv) > 1 and sys.argv[1] == "--online":
    data = sys.stdin.read().strip().split()
    if len(data) >= 2:
        n = int(data[0])
        x = int(data[1])
        arr = [0] # 位置 0 不使用
        for i in range(1, n):
            arr.append(int(data[i + 1]))

        solution = FrogToSchoolOptimized()
        result = solution.compute(n, x, arr)
        print(result)

"""

```

Python 工程化实战建议：

1. 类型注解：
 - 使用 `typing` 模块提供类型提示，提高代码可读性
 - 类型注解有助于 IDE 的智能提示和代码检查

2. 内存管理:

- Python 有自动垃圾回收，但要注意循环引用
- 对于大规模数据，使用生成器表达式节省内存
- 注意列表的深拷贝和浅拷贝区别

3. 性能优化:

- 使用局部变量访问比全局变量更快
- 避免在循环中重复计算相同的表达式
- 使用列表推导式比循环更快

4. 异常处理:

- 使用 try-except 处理可能的异常
- 添加输入合法性检查
- 使用 logging 模块记录错误信息

5. 调试技巧:

- 使用 pdb 进行调试
- 添加 assert 断言验证中间结果
- 使用 timeit 模块测试性能

6. Python 特有优化:

- 使用 numpy 处理数值计算（如果允许）
- 使用 functools.lru_cache 进行记忆化
- 注意 Python 的 GIL 对多线程的影响

7. 算法优化思路:

- 原始验证算法的时间复杂度为 $O(x*n)$ ，对于 $x=1e9$ 的情况会超时
- 优化版本通过数学分析将时间复杂度降至 $O(n)$
- 关键思路是：对于每个位置 i 的石头，最多可以被使用 $\min(\text{arr}[i], x)$ 次

8. 跨语言对比:

- Python 代码更简洁，但运行速度较慢
- 在 Python 中需要注意整数溢出问题，Python 的整数精度不会有大问题
- 对于大规模数据，C++ 版本通常更快

9. 笔试面试技巧:

- 理解二分答案的单调性原理
- 掌握滑动窗口的模板代码
- 能够分析时间复杂度和空间复杂度
- 能够处理边界情况和极端输入

10. 相关题目扩展:

- LeetCode 403: Frog Jump

- LeetCode 55: Jump Game
- LeetCode 45: Jump Game II
- 这些题目都涉及跳跃问题的不同变种，可以对比学习

文件: Code03_MultiplyPositiveNegative1.cpp

```
#include <iostream>
#include <vector>
#include <random>
#include <chrono>

using namespace std;

/***
 * 相乘为正或负的子数组数量 - C++实现
 * 给定一个长度为 n 的数组 arr，其中所有值都不是 0
 * 返回有多少个子数组相乘的结果是正
 * 返回有多少个子数组相乘的结果是负
 * 1 <= n <= 10^6
 * -10^9 <= arr[i] <= +10^9, arr[i]一定不是 0
 * 来自真实大厂笔试，对数据验证
 *
 * 算法思路：
 * 1. 使用前缀和思想，维护当前位置之前正数和负数的子数组数量
 * 2. 遍历数组，维护一个变量 cur 表示到当前位置的累积符号（0 表示正，1 表示负）
 * 3. cnt[0]表示累积符号为正的前缀数量，cnt[1]表示累积符号为负的前缀数量
 * 4. 对于当前位置 i：
 *     - 如果 arr[i]为正数，符号不变，cur 保持不变
 *     - 如果 arr[i]为负数，符号改变，cur ^= 1
 * 5. 如果当前累积符号为 cur，那么：
 *     - 与之前累积符号为 cur 的前缀组合，乘积为正数
 *     - 与之前累积符号为 cur^1 的前缀组合，乘积为负数
 * 6. 更新 cnt 数组
 *
 * 时间复杂度：O(n)，只需要遍历一次数组
 * 空间复杂度：O(1)，只使用了常数额外空间
 */

class MultiplyPositiveNegative {
public:
```

```
    class MultiplyPositiveNegative {  
public:
```

```

vector<int> num(vector<int>& arr) {
    // cnt[0]: 累积符号为正的前缀数量
    // cnt[1]: 累积符号为负的前缀数量
    vector<int> cnt(2, 0);
    // 初始化, 空数组乘积为正数
    cnt[0] = 1;
    cnt[1] = 0;

    int ans1 = 0; // 正数子数组数量
    int ans2 = 0; // 负数子数组数量
    int cur = 0; // 当前累积符号, 0 表示正, 1 表示负

    for (int i = 0; i < arr.size(); i++) {
        // 如果当前元素为负数, 改变符号
        cur ^= (arr[i] > 0) ? 0 : 1;

        // 与之前相同符号的前缀组合, 乘积为正数
        ans1 += cnt[cur];
        // 与之前不同符号的前缀组合, 乘积为负数
        ans2 += cnt[cur ^ 1];

        // 更新 cnt 数组
        cnt[cur]++;
    }

    return {ans1, ans2};
}

```

```

/***
 * 暴力方法 - 用于验证
 * 时间复杂度: O(n^2)
 * 空间复杂度: O(1)
 */

```

```

vector<int> right(vector<int>& arr) {
    int n = arr.size();
    int ans1 = 0;
    int ans2 = 0;

    for (int i = 0; i < n; i++) {
        long long cur = 1;
        for (int j = i; j < n; j++) {
            cur = cur * arr[j];
            if (cur > 0) {

```

```

        ans1++;
    } else {
        ans2++;
    }
}

return {ans1, ans2};
}

/***
 * 生成随机数组用于测试
 * @param n 数组长度
 * @param v 数值范围[-v, v]，但不包含 0
 * @return 随机数组
 */
vector<int> randomArray(int n, int v) {
    vector<int> ans(n);
    random_device rd;
    mt19937 gen(rd());
    uniform_int_distribution<> dis(-v, v);

    for (int i = 0; i < n; i++) {
        int num;
        do {
            num = dis(gen);
        } while (num == 0);
        ans[i] = num;
    }

    return ans;
}

/***
 * 运行测试
 */
void test() {
    int n = 20;
    int v = 10;
    int testTime = 10000;

    cout << "测试开始" << endl;
    auto start = chrono::high_resolution_clock::now();

```

```

for (int i = 0; i < testTime; i++) {
    int size = rand() % n;
    vector<int> arr = randomArray(size, v);
    vector<int> ans1 = num(arr);
    vector<int> ans2 = right(arr);

    if (ans1[0] != ans2[0] || ans1[1] != ans2[1]) {
        cout << "出错了!" << endl;
        cout << "数组: ";
        for (int num : arr) cout << num << " ";
        cout << endl;
        cout << "算法结果: " << ans1[0] << ", " << ans1[1] << endl;
        cout << "暴力结果: " << ans2[0] << ", " << ans2[1] << endl;
        break;
    }
}

auto end = chrono::high_resolution_clock::now();
auto duration = chrono::duration_cast<chrono::milliseconds>(end - start);

cout << "测试结束, 耗时: " << duration.count() << "ms" << endl;
}

/***
 * 性能测试: 大规模数据
 */
void performanceTest() {
    int n = 1000000; // 100 万数据
    int v = 1000;

    cout << "生成大规模测试数据..." << endl;
    vector<int> arr = randomArray(n, v);

    cout << "开始性能测试..." << endl;
    auto start = chrono::high_resolution_clock::now();

    vector<int> result = num(arr);

    auto end = chrono::high_resolution_clock::now();
    auto duration = chrono::duration_cast<chrono::milliseconds>(end - start);

    cout << "性能测试结果:" << endl;
}

```

```

    cout << "正数子数组数量: " << result[0] << endl;
    cout << "负数子数组数量: " << result[1] << endl;
    cout << "耗时: " << duration.count() << "ms" << endl;
}

};

/***
 * 优化版本: 使用更简洁的代码实现
 */
class MultiplyPositiveNegativeOptimized {
public:
    vector<int> num(vector<int>& arr) {
        int cnt[2] = {1, 0}; // cnt[0]:正, cnt[1]:负
        int cur = 0, ans1 = 0, ans2 = 0;

        for (int num : arr) {
            cur ^= (num < 0); // 负数时异或 1, 正数时异或 0
            ans1 += cnt[cur];
            ans2 += cnt[cur ^ 1];
            cnt[cur]++;
        }

        return {ans1, ans2};
    }
};

int main() {
    MultiplyPositiveNegative solution;

    // 基础测试
    cout << "==== 基础测试 ===" << endl;
    vector<int> test1 = {1, -2, 3, -4, 5};
    vector<int> result1 = solution.num(test1);
    cout << "测试数组: 1, -2, 3, -4, 5" << endl;
    cout << "正数子数组数量: " << result1[0] << endl;
    cout << "负数子数组数量: " << result1[1] << endl;

    // 运行测试
    cout << "\n==== 正确性测试 ===" << endl;
    solution.test();

    // 性能测试
    cout << "\n==== 性能测试 ===" << endl;
}

```

```
    solution.performanceTest();

    // 优化版本测试
    cout << "\n==== 优化版本测试 ===" << endl;
    MultiplyPositiveNegativeOptimized solution0pt;
    vector<int> result0pt = solution0pt.num(test1);
    cout << "优化版本结果: " << result0pt[0] << ", " << result0pt[1] << endl;

    return 0;
}

/***
 * C++工程化实战建议:
 *
 * 1. 内存管理:
 *     - 使用 vector 代替原生数组，避免手动内存管理
 *     - 注意 vector 的初始化方式，确保正确初始化
 *     - 对于大规模数据，考虑使用 reserve 预分配内存
 *
 * 2. 类型安全:
 *     - 使用 size_t 处理数组索引，避免负数问题
 *     - 对于大数运算，使用 long long 避免溢出
 *     - 注意整数类型转换，避免精度丢失
 *
 * 3. 性能优化:
 *     - 使用引用传递避免不必要的拷贝
 *     - 对于频繁调用的函数，考虑内联优化
 *     - 使用-O2 或-O3 编译优化
 *
 * 4. 随机数生成:
 *     - 使用 C++11 的随机数库，比 rand() 更安全
 *     - 注意随机数种子的设置
 *     - 确保生成的随机数不包含 0
 *
 * 5. 时间测量:
 *     - 使用 chrono 库进行精确的时间测量
 *     - 注意时间单位的转换
 *     - 多次测量取平均值以获得更准确的结果
 *
 * 6. 调试技巧:
 *     - 使用 gdb 进行调试
 *     - 添加 assert 断言验证中间结果
 *     - 使用 valgrind 检查内存泄漏
 */
```

```
*  
* 7. 算法优化思路:  
*   - 原始暴力方法的时间复杂度为  $O(n^2)$ , 无法处理大规模数据  
*   - 优化方法利用前缀和思想, 将时间复杂度降至  $O(n)$   
*   - 关键思路是维护累积符号的前缀数量  
*   - 这种方法避免了重复计算, 大大提高了效率  
  
*  
* 8. 相关题目扩展:  
*   - LeetCode 152: Maximum Product Subarray  
*   - LeetCode 53: Maximum Subarray  
*   - Codeforces 1215B: The Number of Products  
*   - 这些题目都涉及子数组乘积或和的统计, 可以对比学习  
*/
```

=====

文件: Code03_MultiplyPositiveNegative1.java

=====

```
package class127;  
  
// 相乘为正或负的子数组数量  
// 给定一个长度为 n 的数组 arr, 其中所有值都不是 0  
// 返回有多少个子数组相乘的结果是正  
// 返回有多少个子数组相乘的结果是负  
//  $1 \leq n \leq 10^6$   
//  $-10^9 \leq arr[i] \leq +10^9$ , arr[i]一定不是 0  
// 来自真实大厂笔试, 对数器验证  
  
/**  
 * 算法思路:  
 * 1. 使用前缀和思想, 维护当前位置之前正数和负数的子数组数量  
 * 2. 遍历数组, 维护一个变量 cur 表示到当前位置的累积符号 (0 表示正, 1 表示负)  
 * 3. cnt[0]表示累积符号为正的前缀数量, cnt[1]表示累积符号为负的前缀数量  
 * 4. 对于当前位置 i:  
 *   - 如果 arr[i]为正数, 符号不变, cur 保持不变  
 *   - 如果 arr[i]为负数, 符号改变, cur ^= 1  
 * 5. 如果当前累积符号为 cur, 那么:  
 *   - 与之前累积符号为 cur 的前缀组合, 乘积为正数  
 *   - 与之前累积符号为 cur^1 的前缀组合, 乘积为负数  
 * 6. 更新 cnt 数组  
  
 * 时间复杂度:  $O(n)$ , 只需要遍历一次数组  
 * 空间复杂度:  $O(1)$ , 只使用了常数额外空间
```

```

*/
public class Code03_MultiplyPositiveNegative1 {

    // 正式方法
    public static int[] num(int[] arr) {
        // 0 : 正状态
        // 1 : 负状态
        // ^ : 乘法运算
        int[] cnt = new int[2];
        // 初始化, 空数组乘积为正数
        cnt[0] = 1;
        cnt[1] = 0;
        int ans1 = 0; // 正数子数组数量
        int ans2 = 0; // 负数子数组数量
        // cur 表示当前累积符号, 0 表示正, 1 表示负
        for (int i = 0, cur = 0; i < arr.length; i++) {
            // 如果当前元素为负数, 改变符号
            cur ^= arr[i] > 0 ? 0 : 1;
            // 与之前相同符号的前缀组合, 乘积为正数
            ans1 += cnt[cur];
            // 与之前不同符号的前缀组合, 乘积为负数
            ans2 += cnt[cur ^ 1];
            // 更新 cnt 数组
            cnt[cur]++;
        }
        return new int[] { ans1, ans2 };
    }

    // 暴力方法
    // 为了验证
    public static int[] right(int[] arr) {
        int n = arr.length;
        int ans1 = 0;
        int ans2 = 0;
        for (int i = 0; i < n; i++) {
            long cur = 1;
            for (int j = i; j < n; j++) {
                cur = cur * arr[j];
                if (cur > 0) {
                    ans1++;
                } else {
                    ans2++;
                }
            }
        }
        return new int[] { ans1, ans2 };
    }
}

```

```

    }
}

return new int[] { ans1, ans2 };
}

// 为了验证
public static int[] randomArray(int n, int v) {
    int[] ans = new int[n];
    for (int i = 0; i < n; i++) {
        do {
            ans[i] = (int) (Math.random() * v * 2) - v;
        } while (ans[i] == 0);
    }
    return ans;
}

public static void main(String[] args) {
    // 正式方法无所谓，怎么都正确
    // 但是对数器方法是暴力乘起来，所以 n 和 v 不要太大，防止溢出
    int n = 20;
    int v = 10;
    int testTime = 10000;
    System.out.println("测试开始");
    for (int i = 0; i < testTime; i++) {
        int size = (int) (Math.random() * n);
        int[] arr = randomArray(size, v);
        int[] ans1 = num(arr);
        int[] ans2 = right(arr);
        if (ans1[0] != ans2[0] || ans1[1] != ans2[1]) {
            System.out.println("出错了!");
        }
    }
    System.out.println("测试结束");
}

```

```

// 相关题目：
// 1. LeetCode 152 – Maximum Product Subarray (乘积最大子数组)
//     链接: https://leetcode.cn/problems/maximum-product-subarray/
//     区别：求乘积最大的连续子数组
// 2. LeetCode 53 – Maximum Subarray (最大子数组和)
//     链接: https://leetcode.cn/problems/maximum-subarray/
//     区别：求和最大的连续子数组
// 3. LeetCode 918 – Maximum Sum Circular Subarray (环形子数组的最大和)

```

```
//    链接: https://leetcode.cn/problems/maximum-sum-circular-subarray/
//    区别: 数组是环形的, 求和最大的连续子数组
// 4. Codeforces 1215B - The Number of Products
//    链接: https://codeforces.com/problemset/problem/1215/B
//    区别: 与本题相同, 统计乘积为正和负的子数组数量
}
```

=====

文件: Code03_MultiplyPositiveNegative1.py

=====

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

"""

相乘为正或负的子数组数量 - Python 实现

给定一个长度为 n 的数组 arr, 其中所有值都不是 0

返回有多少个子数组相乘的结果是正

返回有多少个子数组相乘的结果是负

$1 \leq n \leq 10^6$

$-10^9 \leq arr[i] \leq +10^9$, arr[i]一定不是 0

来自真实大厂笔试, 对数据验证

算法思路:

1. 使用前缀和思想, 维护当前位置之前正数和负数的子数组数量
2. 遍历数组, 维护一个变量 cur 表示到当前位置的累积符号 (0 表示正, 1 表示负)
3. cnt[0]表示累积符号为正的前缀数量, cnt[1]表示累积符号为负的前缀数量
4. 对于当前位置 i:
 - 如果 arr[i]为正数, 符号不变, cur 保持不变
 - 如果 arr[i]为负数, 符号改变, cur ^= 1
5. 如果当前累积符号为 cur, 那么:
 - 与之前累积符号为 cur 的前缀组合, 乘积为正数
 - 与之前累积符号为 cur^1 的前缀组合, 乘积为负数
6. 更新 cnt 数组

时间复杂度: O(n), 只需要遍历一次数组

空间复杂度: O(1), 只使用了常数额外空间

"""

```
import random
import time
from typing import List
```

```

class MultiplyPositiveNegative:
    """
    相乘为正或负的子数组数量解决方案
    """

    def num(self, arr: List[int]) -> List[int]:
        """
        计算正数和负数子数组的数量

        Args:
            arr: 输入数组, 所有元素都不为 0

        Returns:
            List[int]: [正数子数组数量, 负数子数组数量]
        """
        # cnt[0]: 累积符号为正的前缀数量
        # cnt[1]: 累积符号为负的前缀数量
        cnt = [1, 0]  # 初始化, 空数组乘积为正数

        ans1 = 0  # 正数子数组数量
        ans2 = 0  # 负数子数组数量
        cur = 0  # 当前累积符号, 0 表示正, 1 表示负

        for num in arr:
            # 如果当前元素为负数, 改变符号
            cur ^= 0 if num > 0 else 1

            # 与之前相同符号的前缀组合, 乘积为正数
            ans1 += cnt[cur]
            # 与之前不同符号的前缀组合, 乘积为负数
            ans2 += cnt[cur ^ 1]

            # 更新 cnt 数组
            cnt[cur] += 1

        return [ans1, ans2]

    def right(self, arr: List[int]) -> List[int]:
        """
        暴力方法 - 用于验证
        时间复杂度: O(n^2)
        空间复杂度: O(1)
        """

```

Args:

arr: 输入数组

Returns:

List[int]: [正数子数组数量, 负数子数组数量]

"""

```
n = len(arr)
```

```
ans1 = 0
```

```
ans2 = 0
```

```
for i in range(n):
```

```
    cur = 1
```

```
    for j in range(i, n):
```

```
        cur = cur * arr[j]
```

```
        if cur > 0:
```

```
            ans1 += 1
```

```
        else:
```

```
            ans2 += 1
```

```
return [ans1, ans2]
```

```
def random_array(self, n: int, v: int) -> List[int]:
```

"""

生成随机数组用于测试

Args:

n: 数组长度

v: 数值范围[-v, v], 但不包含 0

Returns:

List[int]: 随机数组

"""

```
arr = []
```

```
for _ in range(n):
```

```
    num = 0
```

```
    while num == 0:
```

```
        num = random.randint(-v, v)
```

```
    arr.append(num)
```

```
return arr
```

```
def test(self, n: int = 20, v: int = 10, test_time: int = 10000):
```

"""

运行正确性测试

Args:

n: 最大数组长度
v: 数值范围
test_time: 测试次数

"""

```
print("测试开始")
start_time = time.time()

for _ in range(test_time):
    size = random.randint(0, n)
    arr = self.random_array(size, v)
    ans1 = self.num(arr)
    ans2 = self.right(arr)

    if ans1 != ans2:
        print("出错了!")
        print(f"数组: {arr}")
        print(f"算法结果: {ans1}")
        print(f"暴力结果: {ans2}")

    return

end_time = time.time()
print(f"测试结束, 耗时: {end_time - start_time:.2f}秒")
```

def performance_test(self, n: int = 1000000, v: int = 1000):

"""

性能测试: 大规模数据

Args:

n: 数组长度
v: 数值范围

"""

```
print("生成大规模测试数据...")
arr = self.random_array(n, v)

print("开始性能测试...")
start_time = time.time()

result = self.num(arr)

end_time = time.time()
```

```
print("性能测试结果:")
print(f"正数子数组数量: {result[0]}")
print(f"负数子数组数量: {result[1]}")
print(f"耗时: {end_time - start_time:.2f}秒")
```

```
class MultiplyPositiveNegativeOptimized:
```

```
    """
```

```
    优化版本: 使用更简洁的代码实现
```

```
    """
```

```
def num(self, arr: List[int]) -> List[int]:
```

```
    """
```

```
    优化版本的实现
```

```
Args:
```

```
    arr: 输入数组
```

```
Returns:
```

```
    List[int]: [正数子数组数量, 负数子数组数量]
```

```
    """
```

```
cnt = [1, 0] # cnt[0]:正, cnt[1]:负
```

```
cur = ans1 = ans2 = 0
```

```
for num in arr:
```

```
    cur ^= (num < 0) # 负数时异或 1, 正数时异或 0
```

```
    ans1 += cnt[cur]
```

```
    ans2 += cnt[cur ^ 1]
```

```
    cnt[cur] += 1
```

```
return [ans1, ans2]
```

```
def test_basic():
```

```
    """基础测试"""

```

```
    solution = MultiplyPositiveNegative()
```

```
# 测试用例 1
```

```
test1 = [1, -2, 3, -4, 5]
```

```
result1 = solution.num(test1)
```

```
print(f"测试数组: {test1}")
```

```
print(f"正数子数组数量: {result1[0]})")
```

```
print(f"负数子数组数量: {result1[1]})")
```

```
# 测试用例 2: 全正数
test2 = [1, 2, 3, 4, 5]
result2 = solution.num(test2)
print(f"\n测试数组(全正): {test2}")
print(f"正数子数组数量: {result2[0]}")
print(f"负数子数组数量: {result2[1]}")

# 测试用例 3: 全负数
test3 = [-1, -2, -3, -4, -5]
result3 = solution.num(test3)
print(f"\n测试数组(全负): {test3}")
print(f"正数子数组数量: {result3[0]}")
print(f"负数子数组数量: {result3[1]}")

def test_edge_cases():
    """边界测试"""
    solution = MultiplyPositiveNegative()

    # 空数组
    test1 = []
    result1 = solution.num(test1)
    print(f"空数组结果: {result1}")

    # 单个元素
    test2 = [1]
    result2 = solution.num(test2)
    print(f"单个正数结果: {result2}")

    test3 = [-1]
    result3 = solution.num(test3)
    print(f"单个负数结果: {result3}")

if __name__ == "__main__":
    solution = MultiplyPositiveNegative()

    # 基础测试
    print("== 基础测试 ==")
    test_basic()

    # 边界测试
```

```
print("\n==== 边界测试 ===")
test_edge_cases()

# 正确性测试
print("\n==== 正确性测试 ===")
solution.test()

# 性能测试
print("\n==== 性能测试 ===")
solution.performance_test()

# 优化版本测试
print("\n==== 优化版本测试 ===")
solution_opt = MultiplyPositiveNegativeOptimized()
test1 = [1, -2, 3, -4, 5]
result_opt = solution_opt.num(test1)
print(f"优化版本结果: {result_opt}")

"""


```

Python 工程化实战建议：

1. 类型注解:

- 使用 typing 模块提供类型提示，提高代码可读性
- 类型注解有助于 IDE 的智能提示和代码检查

2. 内存管理:

- Python 有自动垃圾回收，但要注意循环引用
- 对于大规模数据，使用生成器表达式节省内存
- 注意列表的深拷贝和浅拷贝区别

3. 性能优化:

- 使用局部变量访问比全局变量更快
- 避免在循环中重复计算相同的表达式
- 使用列表推导式比循环更快

4. 异常处理:

- 使用 try-except 处理可能的异常
- 添加输入合法性检查
- 使用 logging 模块记录错误信息

5. 调试技巧:

- 使用 pdb 进行调试

- 添加 assert 断言验证中间结果

- 使用 timeit 模块测试性能

6. Python 特有优化:

- 使用 numpy 处理数值计算（如果允许）
- 使用 functools.lru_cache 进行记忆化
- 注意 Python 的 GIL 对多线程的影响

7. 算法优化思路:

- 原始暴力方法的时间复杂度为 $O(n^2)$ ，无法处理大规模数据
- 优化方法利用前缀和思想，将时间复杂度降至 $O(n)$
- 关键思路是维护累积符号的前缀数量
- 这种方法避免了重复计算，大大提高了效率

8. 相关题目扩展:

- LeetCode 152: Maximum Product Subarray
- LeetCode 53: Maximum Subarray
- Codeforces 1215B: The Number of Products
- 这些题目都涉及子数组乘积或和的统计，可以对比学习

9. 笔试面试技巧:

- 理解前缀和思想的应用场景
- 掌握符号累积的巧妙用法
- 能够分析时间复杂度和空间复杂度
- 能够处理边界情况和极端输入

10. 数学原理:

- 两个正数相乘为正，两个负数相乘为正
- 正数和负数相乘为负
- 利用异或运算可以高效地切换符号状态

11. 工程化考量:

- 代码的可读性和可维护性
- 异常处理和边界条件检查
- 性能优化和内存管理
- 测试用例的覆盖度

12. 跨语言对比:

- Python 代码更简洁，但运行速度较慢
- 在 Python 中需要注意整数溢出问题，Python 的整数精度不会有问题
- 对于大规模数据，C++版本通常更快

"""

=====

文件: Code03_MultiplyPositiveNegative2.java

=====

```
package class127;
```

```
// 感谢热心的同学，找到了题目3的在线测试  
// 测试链接：https://codeforces.com/problemset/problem/1215/B  
// 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;
```

```
public class Code03_MultiplyPositiveNegative2 {
```

```
    public static int MAXN = 200001;
```

```
    public static int n;
```

```
    public static int[] arr = new int[MAXN];
```

```
    // 结果可能很大，所以用long类型
```

```
    public static long ans1, ans2;
```

```
    public static void compute() {
```

```
        int[] cnt = new int[2];  
        cnt[0] = 1;  
        cnt[1] = 0;  
        ans1 = ans2 = 0;  
        for (int i = 1, cur = 0; i <= n; i++) {  
            cur ^= arr[i] > 0 ? 0 : 1;  
            ans1 += cnt[cur];  
            ans2 += cnt[cur ^ 1];  
            cnt[cur]++;  
        }
```

```
}
```

```
    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;
}
compute();
out.println(ans2 + " " + ans1);
out.flush();
out.close();
br.close();
}

}

```

文件: Code04_LongestAddNotZero.cpp

```

#include <iostream>
#include <vector>
#include <algorithm>
#include <cstring>

using namespace std;

/***
 * 相邻与结果不为 0 的最长子序列 - C++实现
 * 给定一个长度为 n 的数组 arr，你可以随意选择数字组成子序列
 * 但是要求任意相邻的两个数&的结果不能是 0，这样的子序列才是合法的
 * 返回最长合法子序列的长度
 * 1 <= n <= 10^5
 * 0 <= arr[i] <= 10^9
 * 测试链接 : https://www.luogu.com.cn/problem/P4310
 *
 * 算法思路:
 * 1. 这是一个动态规划问题
 * 2. 对于每个数，我们关注它的二进制表示中为 1 的位
 * 3. dp[i] 表示以第 i 位为结尾的最长子序列长度
 * 4. 对于当前数 num，我们找出它二进制表示中为 1 的位 j
 * 5. 找到所有以位 j 结尾的最长子序列长度，取最大值+1 作为新的长度

```

```

* 6. 更新所有 num 中为 1 的位 j 对应的 dp[j]
*
* 时间复杂度: O(n * 31) = O(n), 其中 n 是数组长度
* 空间复杂度: O(1), 只使用了固定大小的数组
*/

```

```

class LongestAddNotZero {
public:
    int compute(vector<int>& arr) {
        int n = arr.size();
        // pre 数组存储以每个二进制位结尾的最长子序列长度
        vector<int> pre(32, 0);

        for (int i = 0; i < n; i++) {
            int num = arr[i];
            int cur = 1; // 当前数字可以单独构成一个子序列

            // 第一遍遍历: 找到当前数之前, 以 num 中任意为 1 的位结尾的最长子序列长度的最大值
            for (int j = 0; j < 31; j++) {
                // 如果 num 的第 j 位为 1
                if ((num >> j) & 1) {
                    // 更新 cur 为以第 j 位结尾的最长子序列长度+1 的最大值
                    cur = max(cur, pre[j] + 1);
                }
            }
        }

        // 第二遍遍历: 更新 pre 数组
        for (int j = 0; j < 31; j++) {
            // 如果 num 的第 j 位为 1
            if ((num >> j) & 1) {
                // 更新以第 j 位结尾的最长子序列长度
                pre[j] = max(pre[j], cur);
            }
        }
    }

    // 找到所有位结尾的最长子序列长度的最大值
    int ans = 0;
    for (int j = 0; j < 31; j++) {
        ans = max(ans, pre[j]);
    }
    return ans;
}

```

```

/***
 * 暴力方法 - 用于验证
 * 时间复杂度: O(2^n), 指数级复杂度, 仅用于小规模测试
 */
int bruteForce(vector<int>& arr) {
    int n = arr.size();
    int maxLen = 0;

    // 枚举所有子序列
    for (int mask = 1; mask < (1 << n); mask++) {
        vector<int> seq;
        for (int i = 0; i < n; i++) {
            if (mask & (1 << i)) {
                seq.push_back(arr[i]);
            }
        }
    }

    // 检查子序列是否合法
    bool valid = true;
    for (int i = 1; i < seq.size(); i++) {
        if ((seq[i-1] & seq[i]) == 0) {
            valid = false;
            break;
        }
    }

    if (valid) {
        maxLen = max(maxLen, (int)seq.size());
    }
}

return maxLen;
}

```

```

/***
 * 优化版本: 使用更简洁的代码实现
 */
int computeOptimized(vector<int>& arr) {
    int n = arr.size();
    vector<int> dp(32, 0);

    for (int num : arr) {

```

```

int cur = 1;

// 找到当前数可以接在哪些位后面
for (int j = 0; j < 31; j++) {
    if ((num >> j) & 1) {
        cur = max(cur, dp[j] + 1);
    }
}

// 更新所有为 1 的位
for (int j = 0; j < 31; j++) {
    if ((num >> j) & 1) {
        dp[j] = max(dp[j], cur);
    }
}

return *max_element(dp.begin(), dp.end());
}

/***
 * 测试函数
 */
void test() {
    // 测试用例 1
    vector<int> test1 = {1, 2, 3, 4, 5};
    int result1 = compute(test1);
    int result10pt = computeOptimized(test1);
    cout << "Test 1: " << result1 << ", Optimized: " << result10pt << endl;

    // 测试用例 2: 包含 0 的情况
    vector<int> test2 = {0, 1, 2, 0, 4};
    int result2 = compute(test2);
    int result20pt = computeOptimized(test2);
    cout << "Test 2: " << result2 << ", Optimized: " << result20pt << endl;

    // 测试用例 3: 大规模数据
    vector<int> test3;
    for (int i = 0; i < 1000; i++) {
        test3.push_back(i);
    }
    int result3 = compute(test3);
    cout << "Test 3 (1000 elements): " << result3 << endl;
}

```

```

// 暴力方法验证（小规模）
vector<int> test4 = {1, 2, 3};
int result4 = compute(test4);
int result4Brute = bruteForce(test4);
cout << "Test 4 - Validation: " << result4 << ", Brute Force: " << result4Brute << endl;
}

/***
 * 性能测试
 */
void performanceTest() {
    int n = 100000; // 10 万数据
    vector<int> arr;

    cout << "生成测试数据..." << endl;
    for (int i = 0; i < n; i++) {
        arr.push_back(rand() % 1000000000);
    }

    cout << "开始性能测试..." << endl;
    auto start = chrono::high_resolution_clock::now();

    int result = compute(arr);

    auto end = chrono::high_resolution_clock::now();
    auto duration = chrono::duration_cast<chrono::milliseconds>(end - start);

    cout << "性能测试结果:" << endl;
    cout << "最长子序列长度: " << result << endl;
    cout << "耗时: " << duration.count() << "ms" << endl;
}

};

int main() {
    LongestAddNotZero solution;

    // 基础测试
    cout << "==== 基础测试 ===" << endl;
    solution.test();

    // 性能测试
    cout << "\n==== 性能测试 ===" << endl;
}

```

```
    solution.performanceTest();  
  
    return 0;  
}  
  
/**/  
 * C++工程化实战建议:  
 *  
 * 1. 内存管理:  
 *     - 使用 vector 代替原生数组，避免手动内存管理  
 *     - 注意 vector 的初始化方式，确保正确初始化  
 *     - 对于大规模数据，考虑使用 reserve 预分配内存  
 *  
 * 2. 位运算优化:  
 *     - 使用位运算检查二进制位，比除法取模更高效  
 *     - 注意位运算的优先级，使用括号确保正确性  
 *     - 对于 32 位整数，只需要检查 0-30 位  
 *  
 * 3. 性能优化:  
 *     - 使用引用传递避免不必要的拷贝  
 *     - 对于频繁调用的函数，考虑内联优化  
 *     - 使用-02 或-03 编译优化  
 *  
 * 4. 算法优化思路:  
 *     - 原始暴力方法的时间复杂度为  $O(2^n)$ ，无法处理大规模数据  
 *     - 优化方法利用二进制位状态，将时间复杂度降至  $O(n)$   
 *     - 关键思路是维护以每个二进制位结尾的最长子序列长度  
 *     - 这种方法避免了枚举所有子序列，大大提高了效率  
 *  
 * 5. 边界情况处理:  
 *     - 处理数组为空的情况  
 *     - 处理数组中包含 0 的情况  
 *     - 处理所有数字都为 0 的情况  
 *     - 处理大数的情况  
 *  
 * 6. 调试技巧:  
 *     - 使用 gdb 进行调试  
 *     - 添加 assert 断言验证中间结果  
 *     - 使用 valgrind 检查内存泄漏  
 *  
 * 7. 相关题目扩展:  
 *     - LeetCode 300: Longest Increasing Subsequence  
 *     - LeetCode 128: Longest Consecutive Sequence
```

```
*      - LeetCode 152: Maximum Product Subarray
*      - 这些题目都涉及子序列或子数组的统计，可以对比学习
*
* 8. 数学原理:
*      - 两个数相与不为 0，意味着它们至少有一个相同的二进制位为 1
*      - 子序列中相邻数字的约束条件可以转化为二进制位的约束
*      - 利用二进制位状态可以高效地维护子序列信息
*
* 9. 工程化考量:
*      - 代码的可读性和可维护性
*      - 异常处理和边界条件检查
*      - 性能优化和内存管理
*      - 测试用例的覆盖度
*/
=====
```

文件: Code04_LongestAddNotZero.java

```
=====
package class127;

// 相邻与结果不为 0 的最长子序列
// 给定一个长度为 n 的数组 arr，你可以随意选择数字组成子序列
// 但是要求任意相邻的两个数&的结果不能是 0，这样的子序列才是合法的
// 返回最长合法子序列的长度
// 1 <= n <= 10^5
// 0 <= arr[i] <= 10^9
// 测试链接 : https://www.luogu.com.cn/problem/P4310
// 提交以下的 code，提交时请把类名改成"Main"，可以通过所有用例
```

```
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;
```

```
/**
 * 算法思路:
 * 1. 这是一个动态规划问题
 * 2. 对于每个数，我们关注它的二进制表示中为 1 的位
 * 3. dp[i] 表示以第 i 位为结尾的最长子序列长度
```

```

* 4. 对于当前数 num，我们找出它二进制表示中为 1 的位 j
* 5. 找到所有以位 j 结尾的最长子序列长度，取最大值+1 作为新的长度
* 6. 更新所有 num 中为 1 的位 j 对应的 dp[j]
*
* 时间复杂度: O(n * 31) = O(n)，其中 n 是数组长度
* 空间复杂度: O(1)，只使用了固定大小的数组
*/
public class Code04_LongestAddNotZero {

    public static int MAXN = 100001;

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

    public static int[] pre = new int[32];

    public static int n;

    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 = 0; i < n; i++) {
            in.nextToken();
            arr[i] = (int) in.nval;
        }
        out.println(compute());
        out.flush();
        out.close();
        br.close();
    }

    /**
     * 计算最长合法子序列的长度
     * @return 最长合法子序列的长度
     */
    public static int compute() {
        // 初始化 pre 数组
        Arrays.fill(pre, 0);
        // 遍历每个数
        for (int i = 0, num, cur; i < n; i++) {
            num = arr[i];

```

```

cur = 1;
// 找到当前数之前，以 num 中任意为 1 的位结尾的最长子序列长度的最大值
for (int j = 0; j < 31; j++) {
    // 如果 num 的第 j 位为 1
    if (((num >> j) & 1) == 1) {
        // 更新 cur 为以第 j 位结尾的最长子序列长度+1 的最大值
        cur = Math.max(cur, pre[j] + 1);
    }
}
// 更新 pre 数组
for (int j = 0; j < 31; j++) {
    // 如果 num 的第 j 位为 1
    if (((num >> j) & 1) == 1) {
        // 更新以第 j 位结尾的最长子序列长度
        pre[j] = Math.max(pre[j], cur);
    }
}
}
// 找到所有位结尾的最长子序列长度的最大值
int ans = 0;
for (int j = 0; j < 31; j++) {
    ans = Math.max(ans, pre[j]);
}
return ans;
}

```

// 相关题目：

```

// 1. LeetCode 152 - Maximum Product Subarray (乘积最大子数组)
//     链接: https://leetcode.cn/problems/maximum-product-subarray/
//     区别: 求连续子数组的最大乘积
// 2. LeetCode 53 - Maximum Subarray (最大子数组和)
//     链接: https://leetcode.cn/problems/maximum-subarray/
//     区别: 求连续子数组的最大和
// 3. LeetCode 300 - Longest Increasing Subsequence (最长递增子序列)
//     链接: https://leetcode.cn/problems/longest-increasing-subsequence/
//     区别: 求最长递增子序列
// 4. LeetCode 128 - Longest Consecutive Sequence (最长连续序列)
//     链接: https://leetcode.cn/problems/longest-consecutive-sequence/
//     区别: 求最长连续数字序列
}
=====
```

文件: Code04_LongestAddNotZero.py

```
=====
```

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

```
"""
```

相邻与结果不为 0 的最长子序列 - Python 实现

给定一个长度为 n 的数组 arr, 你可以随意选择数字组成子序列

但是要求任意相邻的两个数&的结果不能是 0, 这样的子序列才是合法的

返回最长合法子序列的长度

```
1 <= n <= 10^5
```

```
0 <= arr[i] <= 10^9
```

测试链接 : <https://www.luogu.com.cn/problem/P4310>

算法思路:

1. 这是一个动态规划问题
2. 对于每个数, 我们关注它的二进制表示中为 1 的位
3. $dp[i]$ 表示以第 i 位为结尾的最长子序列长度
4. 对于当前数 num , 我们找出它二进制表示中为 1 的位 j
5. 找到所有以位 j 结尾的最长子序列长度, 取最大值+1 作为新的长度
6. 更新所有 num 中为 1 的位 j 对应的 $dp[j]$

时间复杂度: $O(n * 31) = O(n)$, 其中 n 是数组长度

空间复杂度: $O(1)$, 只使用了固定大小的数组

```
"""
```

```
import random
import time
from typing import List
```

```
class LongestAddNotZero:
```

```
"""
```

相邻与结果不为 0 的最长子序列解决方案

```
"""
```

```
def compute(self, arr: List[int]) -> int:
```

```
"""
```

计算最长合法子序列的长度

Args:

arr: 输入数组

Returns:

```

int: 最长合法子序列的长度
"""

if not arr:
    return 0

n = len(arr)
# pre 数组存储以每个二进制位结尾的最长子序列长度
pre = [0] * 32

for num in arr:
    cur = 1 # 当前数字可以单独构成一个子序列

    # 第一遍遍历: 找到当前数之前, 以 num 中任意为 1 的位结尾的最长子序列长度的最大值
    for j in range(31):
        # 如果 num 的第 j 位为 1
        if (num >> j) & 1:
            # 更新 cur 为以第 j 位结尾的最长子序列长度+1 的最大值
            cur = max(cur, pre[j] + 1)

    # 第二遍遍历: 更新 pre 数组
    for j in range(31):
        # 如果 num 的第 j 位为 1
        if (num >> j) & 1:
            # 更新以第 j 位结尾的最长子序列长度
            pre[j] = max(pre[j], cur)

    # 找到所有位结尾的最长子序列长度的最大值
return max(pre)

def compute_optimized(self, arr: List[int]) -> int:
"""

优化版本: 使用更简洁的代码实现

Args:
    arr: 输入数组

Returns:
    int: 最长合法子序列的长度
"""

if not arr:
    return 0

dp = [0] * 32

```

```

for num in arr:
    cur = 1

    # 找到当前数可以接在哪些位后面
    for j in range(31):
        if (num >> j) & 1:
            cur = max(cur, dp[j] + 1)

    # 更新所有为 1 的位
    for j in range(31):
        if (num >> j) & 1:
            dp[j] = max(dp[j], cur)

return max(dp)

```

```

def brute_force(self, arr: List[int]) -> int:
    """
    暴力方法 - 用于验证
    时间复杂度: O(2^n)，指数级复杂度，仅用于小规模测试

```

Args:

arr: 输入数组

Returns:

int: 最长合法子序列的长度

"""

n = len(arr)

max_len = 0

枚举所有子序列

for mask in range(1, 1 << n):

seq = []

for i in range(n):

if mask & (1 << i):

seq.append(arr[i])

检查子序列是否合法

valid = True

for i in range(1, len(seq)):

if (seq[i-1] & seq[i]) == 0:

valid = False

break

```
if valid:
    max_len = max(max_len, len(seq))

return max_len

def test_basic(self):
    """基础测试"""
    # 测试用例 1
    test1 = [1, 2, 3, 4, 5]
    result1 = self.compute(test1)
    result1_opt = self.compute_optimized(test1)
    print(f"测试数组: {test1}")
    print(f"结果: {result1}, 优化版本: {result1_opt}")

    # 测试用例 2: 包含 0 的情况
    test2 = [0, 1, 2, 0, 4]
    result2 = self.compute(test2)
    result2_opt = self.compute_optimized(test2)
    print(f"测试数组(含 0): {test2}")
    print(f"结果: {result2}, 优化版本: {result2_opt}")

    # 测试用例 3: 全 0 的情况
    test3 = [0, 0, 0, 0]
    result3 = self.compute(test3)
    result3_opt = self.compute_optimized(test3)
    print(f"测试数组(全 0): {test3}")
    print(f"结果: {result3}, 优化版本: {result3_opt}")

    # 测试用例 4: 单个元素
    test4 = [1]
    result4 = self.compute(test4)
    result4_opt = self.compute_optimized(test4)
    print(f"测试数组(单个): {test4}")
    print(f"结果: {result4}, 优化版本: {result4_opt}")

def test_validation(self):
    """正确性验证(小规模)"""
    print("\n==== 正确性验证 ====")

    # 小规模测试用例
    test_cases = [
        [1, 2, 3],
```

```
[1, 3, 7],  
[2, 4, 8],  
[1, 2, 4, 8]  
]  
  
for i, test_case in enumerate(test_cases):  
    result = self.compute(test_case)  
    result_opt = self.compute_optimized(test_case)  
    result_brute = self.brute_force(test_case)  
  
    print(f"测试用例 {i+1}: {test_case}")  
    print(f"算法结果: {result}, 优化版本: {result_opt}, 暴力结果: {result_brute}")  
  
    if result == result_brute and result_opt == result_brute:  
        print("✓ 结果正确")  
    else:  
        print("✗ 结果错误")  
  
def performance_test(self, n: int = 100000):  
    """性能测试"""  
    print("\n==== 性能测试 ===")  
  
    # 生成测试数据  
    print("生成测试数据...")  
    arr = [random.randint(0, 10**9) for _ in range(n)]  
  
    # 测试标准版本  
    print("测试标准版本...")  
    start_time = time.time()  
    result1 = self.compute(arr)  
    end_time = time.time()  
    time1 = end_time - start_time  
  
    # 测试优化版本  
    print("测试优化版本...")  
    start_time = time.time()  
    result2 = self.compute_optimized(arr)  
    end_time = time.time()  
    time2 = end_time - start_time  
  
    print(f"测试数据规模: {n}")  
    print(f"标准版本结果: {result1}, 耗时: {time1:.4f}秒")  
    print(f"优化版本结果: {result2}, 耗时: {time2:.4f}秒")
```

```
if result1 == result2:  
    print("✓ 两个版本结果一致")  
else:  
    print("✗ 两个版本结果不一致")  
  
  
def analyze_algorithm():  
    """算法分析"""  
    print("\n==== 算法分析 ===")  
  
    print("1. 算法思路: ")  
    print("    - 利用二进制位状态来维护子序列信息")  
    print("    - 每个二进制位记录以该位结尾的最长子序列长度")  
    print("    - 对于每个数字, 找到可以接在哪些位后面")  
  
    print("\n2. 时间复杂度分析: ")  
    print("    - 遍历数组: O(n)")  
    print("    - 每个数字检查 31 个二进制位: O(31)")  
    print("    - 总时间复杂度: O(n * 31) = O(n)")  
  
    print("\n3. 空间复杂度分析: ")  
    print("    - 使用固定大小的数组存储二进制位状态: O(32) = O(1)")  
  
    print("\n4. 关键技巧: ")  
    print("    - 将子序列约束条件转化为二进制位约束")  
    print("    - 利用位运算高效检查二进制位")  
    print("    - 维护每个二进制位的最优解")  
  
  
if __name__ == "__main__":  
    solution = LongestAddNotZero()  
  
    # 基础测试  
    print("==== 基础测试 ===")  
    solution.test_basic()  
  
    # 正确性验证  
    solution.test_validation()  
  
    # 性能测试  
    solution.performance_test(100000)
```

```
# 算法分析
analyze_algorithm()
```

"""

Python 工程化实战建议:

1. 类型注解:

- 使用 `typing` 模块提供类型提示，提高代码可读性
- 类型注解有助于 IDE 的智能提示和代码检查

2. 内存管理:

- Python 有自动垃圾回收，但要注意循环引用
- 对于大规模数据，使用生成器表达式节省内存
- 注意列表的深拷贝和浅拷贝区别

3. 性能优化:

- 使用局部变量访问比全局变量更快
- 避免在循环中重复计算相同的表达式
- 使用列表推导式比循环更快

4. 位运算技巧:

- 使用位运算检查二进制位，比除法取模更高效
- 注意 Python 的整数是任意精度的，但位运算只关注 32 位
- 使用 `(num >> j) & 1` 检查第 `j` 位是否为 1

5. 异常处理:

- 使用 `try-except` 处理可能的异常
- 添加输入合法性检查
- 使用 `logging` 模块记录错误信息

6. 调试技巧:

- 使用 `pdb` 进行调试
- 添加 `assert` 断言验证中间结果
- 使用 `timeit` 模块测试性能

7. 算法优化思路:

- 原始暴力方法的时间复杂度为 $O(2^n)$ ，无法处理大规模数据
- 优化方法利用二进制位状态，将时间复杂度降至 $O(n)$
- 关键思路是维护以每个二进制位结尾的最长子序列长度
- 这种方法避免了枚举所有子序列，大大提高了效率

8. 相关题目扩展:

- LeetCode 300: Longest Increasing Subsequence
- LeetCode 128: Longest Consecutive Sequence
- LeetCode 152: Maximum Product Subarray
- 这些题目都涉及子序列或子数组的统计，可以对比学习

9. 数学原理:

- 两个数相与不为 0，意味着它们至少有一个相同的二进制位为 1
- 子序列中相邻数字的约束条件可以转化为二进制位的约束
- 利用二进制位状态可以高效地维护子序列信息

10. 工程化考量:

- 代码的可读性和可维护性
- 异常处理和边界条件检查
- 性能优化和内存管理
- 测试用例的覆盖度

11. 跨语言对比:

- Python 代码更简洁，但运行速度较慢
- 在 Python 中需要注意整数溢出问题，Python 的整数精度不会有问题
- 对于大规模数据，C++版本通常更快

12. 笔试面试技巧:

- 理解二进制位状态的应用场景
- 掌握位运算的巧妙用法
- 能够分析时间复杂度和空间复杂度
- 能够处理边界情况和极端输入

=====

文件: Code05_WaysOfArrangePlates.java

```
=====
package class127;

// 摆盘子的方法
// 一共有 n 个盘子 k 种菜，所有盘子排成一排，每个盘子只能放一种菜
// 要求最多连续两个盘子菜品一样，更长的重复出现是不允许的
// 返回摆盘的方法数，答案对 1000000007 取模
// 1 <= n <= 1000
// 1 <= k <= 1000
// 来自真实大厂笔试，对数据验证

/**
```

- * 算法思路:
- * 1. 这是一个动态规划问题
- * 2. 定义状态 $f(i)$ 表示长度为 i 的摆盘方法数
- * 3. 状态转移方程:
 - $f(0) = 1$ (空序列)
 - $f(1) = k$ (第一种菜有 k 种选择)
 - 对于 $i \geq 2$, 考虑最后两个盘子:
 - * 如果最后两个盘子菜品不同, 方法数为 $f(i-1) * (k-1)$
 - * 如果最后两个盘子菜品相同, 倒数第三个盘子必须不同, 方法数为 $f(i-2) * (k-1)$
 - 所以 $f(i) = (f(i-1) + f(i-2)) * (k-1)$
- * 4. 特殊情况: 当 $n=1$ 时, 答案为 k
- * 5. 优化: 可以使用矩阵快速幂优化到 $O(\log n)$ 时间复杂度
- *
- * 时间复杂度:
 - 普通动态规划: $O(n)$
 - 矩阵快速幂优化: $O(\log n)$
- * 空间复杂度: $O(1)$
- */

```
public class Code05_WaysOfArrangePlates {
```

```
    public static int MOD = 1000000007;
```

```
// 正式方法的尝试思路
```

```
    public static int dp1(int n, int k) {
        if (n == 1) {
            return k;
        }
        return (int) (((long) f1(n - 1, k) + f1(n - 2, k)) * k) % MOD;
    }
```

```
    public static int f1(int i, int k) {
        if (i == 0) {
            return 1;
        }
        if (i == 1) {
            return k - 1;
        }
        long p1 = f1(i - 1, k);
        long p2 = f1(i - 2, k);
        return (int) (((p1 + p2) * (k - 1)) % MOD);
    }
```

```
// 正式方法的普通动态规划版本
```

```

// 时间复杂度 O(n)
public static int dp2(int n, int k) {
    if (n == 1) {
        return k;
    }
    int[] dp = new int[n];
    dp[0] = 1;
    dp[1] = k - 1;
    for (int i = 2; i < n; i++) {
        long p1 = dp[i - 1];
        long p2 = dp[i - 2];
        dp[i] = (int) (((p1 + p2) * (k - 1)) % MOD);
    }
    return (int) (((long) dp[n - 1] + dp[n - 2]) * k) % MOD;
}

```

```

// 最优解的版本，动态规划 + 矩阵快速幂优化
// 时间复杂度 O(log n)
// 不会的同学看讲解 098
public static int dp3(int n, int k) {
    if (n == 1) {
        return k;
    }
    int s = k - 1;
    int[][] start = { { 1, s } };
    int[][] base = { { 0, s }, { 1, s } };
    int[][] ans = multiply(start, power(base, n - 2));
    return (int) (((long) ans[0][0] + ans[0][1]) * k) % MOD;
}

```

```

public static int[][] multiply(int[][] a, int[][] b) {
    int n = a.length;
    int m = b[0].length;
    int k = a[0].length;
    int[][] ans = new int[n][m];
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            for (int c = 0; c < k; c++) {
                ans[i][j] = (int) (((long) ans[i][j] + (long) a[i][c] * b[c][j])) % MOD;
            }
        }
    }
    return ans;
}

```

```
}

public static int[][] power(int[][] m, int p) {
    int n = m.length;
    int[][] ans = new int[n][n];
    for (int i = 0; i < n; i++) {
        ans[i][i] = 1;
    }
    for (; p != 0; p >>= 1) {
        if ((p & 1) != 0) {
            ans = multiply(ans, m);
        }
        m = multiply(m, m);
    }
    return ans;
}
```

```
// 暴力方法
// 为了验证
public static int right(int n, int k) {
    int[] path = new int[n];
    return dfs(path, 0, k);
}

// 暴力方法
// 为了验证
public static int dfs(int[] path, int i, int k) {
    if (i == path.length) {
        int len = 1;
        for (int j = 1; j < path.length; j++) {
            if (path[j - 1] == path[j]) {
                len++;
            } else {
                len = 1;
            }
            if (len > 2) {
                return 0;
            }
        }
        return len > 2 ? 0 : 1;
    } else {
        int ans = 0;
        for (int j = 0; j < k; j++) {
```

```

        path[i] = j;
        ans += dfs(path, i + 1, k);
    }
    return ans;
}

}

// 对数据
// 为了验证
public static void main(String[] args) {
    System.out.println("功能测试开始");
    for (int n = 1; n <= 8; n++) {
        for (int k = 1; k <= 8; k++) {
            int ans1 = dp1(n, k);
            int ans2 = dp2(n, k);
            int ans3 = dp3(n, k);
            int ans4 = right(n, k);
            if (ans1 != ans2 || ans1 != ans3 || ans1 != ans4) {
                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");
    int n = 505060123;
    int k = 303060123;
    System.out.println("数据量 : n = " + n + ", k = " + k);

    long start, end;
    start = System.currentTimeMillis();
    System.out.println("dp2 方法结果(已经取模) : " + dp2(n, k));
    end = System.currentTimeMillis();
    System.out.println("dp2 方法用时(毫秒) : " + (end - start));

    start = System.currentTimeMillis();
    System.out.println("dp3 方法结果(已经取模) : " + dp3(n, k));
    end = System.currentTimeMillis();
    System.out.println("dp3 方法用时(毫秒) : " + (end - start));
    System.out.println("性能测试结束");
}

```

```
// 相关题目：  
// 1. LeetCode 935 - Knight Dialer (骑士拨号器)  
//   链接: https://leetcode.cn/problems/knight-dialer/  
//   区别: 骑士在电话垫上跳跃, 计算不同长度的数字序列数量  
// 2. LeetCode 790 - Domino and Tromino Tiling (多米诺和托米诺平铺)  
//   链接: https://leetcode.cn/problems/domino-and-tromino-tiling/  
//   区别: 用多米诺骨牌和托米诺骨牌铺满  $2 \times n$  的面板  
// 3. LeetCode 123 - Best Time to Buy and Sell Stock III (买卖股票的最佳时机 III)  
//   链接: https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-iii/  
//   区别: 最多完成两笔交易, 计算最大利润  
}
```

=====

文件: Code06_FrogCrossRiver.cpp

=====

```
#include <iostream>  
#include <vector>  
#include <algorithm>  
#include <climits>  
#include <cstring>  
  
using namespace std;  
  
/**  
 * 青蛙过河问题 - C++实现  
 * 算法思路: 动态规划 + 距离压缩  
 * 时间复杂度:  $O(n * (t-s+1))$   
 * 空间复杂度:  $O(n)$   
 *  
 * 核心优化:  
 * 1. 当  $s < t$  时, 通过实验确定安全距离, 超过这个距离后就不会再遇到新的石子  
 * 2. 对石子位置进行压缩, 减少动态规划的范围  
 * 3. 使用滑动窗口思想优化状态转移  
 */
```

```
const int MAXN = 101;  
const int MAXL = 100001;  
const int MAXK = 201;
```

```
int arr[MAXN];  
int where[MAXN];  
int dp[MAXL];
```

```

bool stone[MAXL];
bool reach[MAXK];
int n, s, t, m, safe;

/***
 * 计算安全距离
 * 一旦 s 和 t 定了，那么距离多远就可以缩减呢？
 * 通过实验确定安全距离
 */
int reduce(int s, int t) {
    memset(reach, false, sizeof(reach));
    int cnt = 0;
    int ans = 0;
    for (int i = 0; i < MAXK; i++) {
        for (int j = i + s; j < min(i + t + 1, MAXK); j++) {
            reach[j] = true;
        }
        if (!reach[i]) {
            cnt = 0;
        } else {
            cnt++;
        }
        if (cnt == t) {
            ans = i;
            break;
        }
    }
    return ans;
}

/***
 * 计算青蛙过河最少踩到的石子数
 * @return 最少踩到的石子数
 */
int compute() {
    // 对石子位置进行排序
    sort(arr + 1, arr + m + 1);

    // 特殊情况: s == t
    if (s == t) {
        int ans = 0;
        for (int i = 1; i <= m; i++) {
            if (arr[i] % s == 0) {

```

```

        ans++;
    }
}

return ans;
} else { // s < t
    // 计算安全距离
    safe = reduce(s, t);

    // 重新计算石子位置
    where[0] = 0;
    for (int i = 1; i <= m; i++) {
        where[i] = where[i - 1] + min(arr[i] - arr[i - 1], safe);
        stone[where[i]] = true;
    }

    // 更新桥的长度
    n = where[m] + safe;

    // 初始化 dp 数组
    for (int i = 1; i <= n; i++) {
        dp[i] = MAXN;
    }
    dp[0] = 0;

    // 动态规划
    for (int i = 1; i <= n; i++) {
        for (int j = max(i - t, 0); j <= i - s; j++) {
            dp[i] = min(dp[i], dp[j] + (stone[i] ? 1 : 0));
        }
    }

    // 找到最小值
    int ans = MAXN;
    for (int i = where[m] + 1; i <= n; i++) {
        ans = min(ans, dp[i]);
    }
    return ans;
}

/***
 * 单元测试函数
 */

```

```
void test() {
    // 测试用例 1: 基础测试
    n = 10;
    s = 2;
    t = 3;
    m = 2;
    arr[1] = 2;
    arr[2] = 3;

    int result1 = compute();
    cout << "Test 1 - Basic: " << result1 << endl;

    // 测试用例 2: 边界测试 - s == t
    n = 10;
    s = 2;
    t = 2;
    m = 3;
    arr[1] = 2;
    arr[2] = 4;
    arr[3] = 6;

    int result2 = compute();
    cout << "Test 2 - s == t: " << result2 << endl;

    // 测试用例 3: 无石子
    n = 10;
    s = 2;
    t = 3;
    m = 0;

    int result3 = compute();
    cout << "Test 3 - No stones: " << result3 << endl;
}

int main() {
    // 从标准输入读取数据
    cin >> n >> s >> t >> m;
    for (int i = 1; i <= m; i++) {
        cin >> arr[i];
    }

    cout << compute() << endl;
```

```
// 运行单元测试
// test();

return 0;
}

/*
 * 相关题目扩展:
 *
 * 1. LeetCode 403 - Frog Jump (青蛙跳)
 *   链接: https://leetcode.cn/problems/frog-jump/
 *   区别: 青蛙在河中跳跃, 每个位置可能有石头, 需要判断能否到达最后一块石头
 *   解法: 使用哈希表记录每个位置可以跳跃的距离
 *
 * 2. LeetCode 1340 - Jump Game V (跳跃游戏 V)
 *   链接: https://leetcode.cn/problems/jump-game-v/
 *   区别: 在数组中跳跃, 每次跳跃不能超过固定距离, 且需要满足特定条件
 *   解法: 动态规划 + 单调栈
 *
 * 3. LeetCode 1306 - Jump Game III (跳跃游戏 III)
 *   链接: https://leetcode.cn/problems/jump-game-iii/
 *   区别: 在数组中跳跃, 从起始位置开始, 判断能否到达值为 0 的位置
 *   解法: BFS 或 DFS
 *
 * 4. Codeforces 965D - Single Wildcard Pattern Matching
 *   链接: https://codeforces.com/problemset/problem/965/D
 *   区别: 青蛙在河中跳跃, 河中有一些石头, 需要计算能否到达对岸
 *
 * 5. LeetCode 45 - Jump Game II (跳跃游戏 II)
 *   链接: https://leetcode.cn/problems/jump-game-ii/
 *   区别: 计算到达数组末尾的最少跳跃次数
 *   解法: 贪心算法
 *
 * 6. LeetCode 55 - Jump Game (跳跃游戏)
 *   链接: https://leetcode.cn/problems/jump-game/
 *   区别: 判断能否到达数组末尾
 *   解法: 贪心算法
 *
 * 7. LeetCode 1696 - Jump Game VI (跳跃游戏 VI)
 *   链接: https://leetcode.cn/problems/jump-game-vi/
 *   区别: 在数组中跳跃, 每次跳跃有最大距离限制, 需要最大化得分
 *   解法: 动态规划 + 单调队列
 *
```

- * 8. LeetCode 1871 - Jump Game VII (跳跃游戏 VII)
 - * 链接: <https://leetcode.cn/problems/jump-game-vii/>
 - * 区别: 在二进制字符串中跳跃, 需要判断能否到达末尾
 - * 解法: BFS 或滑动窗口
 - *
 - * 9. LeetCode 1345 - Jump Game IV (跳跃游戏 IV)
 - * 链接: <https://leetcode.cn/problems/jump-game-iv/>
 - * 区别: 在数组中跳跃, 可以跳到相同值的位置
 - * 解法: BFS + 哈希表
 - *
 - * 10. LeetCode 1346 - Jump Game V (跳跃游戏 V)
 - * 链接: <https://leetcode.cn/problems/jump-game-v/>
 - * 区别: 在数组中跳跃, 有最大跳跃距离限制
 - * 解法: 动态规划 + 排序
 - *
- * 算法技巧总结:
- * 1. 动态规划是解决跳跃类问题的核心方法
 - * 2. 当数据规模较大时, 需要考虑距离压缩等优化技巧
 - * 3. 对于特殊边界情况 (如 $s==t$) 需要单独处理
 - * 4. 滑动窗口思想可以优化状态转移过程
 - * 5. 实验法确定安全距离是本题的关键优化点
 - *
- * 工程化考量:
- * 1. 异常处理: 输入数据合法性验证
 - * 2. 性能优化: 避免重复计算, 使用合适的数据结构
 - * 3. 内存管理: 合理分配数组大小, 避免内存泄漏
 - * 4. 测试覆盖: 包含边界测试、性能测试、异常测试
- */
-
-

文件: Code06_FrogCrossRiver.java

```
package class127;

// 过河踩过的最少石子数
// 在河上有一座独木桥, 一只青蛙想沿着独木桥从河的一侧跳到另一侧
// 在桥上有一些石子, 青蛙很讨厌踩在这些石子上
// 我们可以把独木桥上青蛙可能到达的点看成数轴上的一串整点 0...n
// 其中 n 是桥的长度, 坐标为 0 的点表示桥的起点, 坐标为 n 的点表示桥的终点
// 青蛙从桥的起点开始, 不停的向终点方向跳跃, 一次跳跃的距离是 [s, t] 之间的任意正整数
// 当青蛙跳到或跳过坐标为 n 的点时, 就算青蛙已经跳出了独木桥
// 题目给出独木桥的长度 n, 青蛙跳跃的距离范围 s、t, 题目还给定 m 个桥上石子的位置
```

```
// 你的任务是确定青蛙要想过河，最少需要踩到的石子数
// 1 <= n <= 10^7
// 1 <= s <= t <= 10
// 1 <= m <= 100
// 测试链接 : https://www.luogu.com.cn/problem/P1052
// 提交以下的 code，提交时请把类名改成"Main"，可以通过所有用例

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;

/**
 * 算法思路:
 * 1. 这是一个动态规划问题
 * 2. dp[i]表示到达位置 i 时最少踩到的石子数
 * 3. 状态转移方程:
 *      dp[i] = min(dp[j] + stone[i] ? 1 : 0) for j in [i-t, i-s]
 * 4. 优化: 当 s < t 时, 可以对距离进行压缩, 因为青蛙可以跳到足够远的位置
 * 通过实验确定安全距离, 超过这个距离后就不会再遇到新的石子
 *
 * 时间复杂度: O(n * (t-s+1))
 * 空间复杂度: O(n)
 */
public class Code06_FrogCrossRiver {

    public static int MAXN = 101;

    public static int MAXL = 100001;

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

    public static int[] where = new int[MAXN];

    public static int[] dp = new int[MAXL];

    public static boolean[] stone = new boolean[MAXL];

    public static int n, s, t, m, safe;
```

```
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;
    in.nextToken();
    s = (int) in.nval;
    in.nextToken();
    t = (int) in.nval;
    in.nextToken();
    m = (int) in.nval;
    for (int i = 1; i <= m; i++) {
        in.nextToken();
        arr[i] = (int) in.nval;
    }
    out.println(compute());
    out.flush();
    out.close();
    br.close();
}
```

```
/**
```

```
* 计算青蛙过河最少踩到的石子数
```

```
* @return 最少踩到的石子数
```

```
*/
```

```
public static int compute() {
    // 对石子位置进行排序
    Arrays.sort(arr, 1, m + 1);
    // 特殊情况: s == t
    if (s == t) {
        int ans = 0;
        for (int i = 1; i <= m; i++) {
            if (arr[i] % s == 0) {
                ans++;
            }
        }
        return ans;
    } else { // s < t
        // 计算安全距离
        safe = reduce(s, t);
        // 重新计算石子位置
        for (int i = 1; i <= m; i++) {
```

```

        where[i] = where[i - 1] + Math.min(arr[i] - arr[i - 1], safe);
        stone[where[i]] = true;
    }
    // 更新桥的长度
    n = where[m] + safe;
    // 初始化 dp 数组
    Arrays.fill(dp, 1, n + 1, MAXN);
    dp[0] = 0;
    // 动态规划
    for (int i = 1; i <= n; i++) {
        for (int j = Math.max(i - t, 0); j <= i - s; j++) {
            dp[i] = Math.min(dp[i], dp[j] + (stone[i] ? 1 : 0));
        }
    }
    // 找到最小值
    int ans = MAXN;
    for (int i = where[m] + 1; i <= n; i++) {
        ans = Math.min(ans, dp[i]);
    }
    return ans;
}
}

```

```

public static int MAXK = 201;

public static boolean[] reach = new boolean[MAXK];

```

```

// 一旦 s 和 t 定了，那么距离多远就可以缩减呢？
// 做实验！
public static int reduce(int s, int t) {
    Arrays.fill(reach, false);
    int cnt = 0;
    int ans = 0;
    for (int i = 0; i < MAXK; i++) {
        for (int j = i + s; j < Math.min(i + t + 1, MAXK); j++) {
            reach[j] = true;
        }
        if (!reach[i]) {
            cnt = 0;
        } else {
            cnt++;
        }
        if (cnt == t) {

```

```

        ans = i;
        break;
    }
}

return ans;
}

// 相关题目:
// 1. LeetCode 403 - Frog Jump
//     链接: https://leetcode.cn/problems/frog-jump/
//     区别: 青蛙在河中跳跃, 每个位置可能有石头, 需要判断能否到达最后一块石头
// 2. Codeforces 965D - Single Wildcard Pattern Matching
//     链接: https://codeforces.com/problemset/problem/965/D
//     区别: 青蛙在河中跳跃, 河中有一些石头, 需要计算能否到达对岸
// 3. LeetCode 1340 - Jump Game V
//     链接: https://leetcode.cn/problems/jump-game-v/
//     区别: 在数组中跳跃, 每次跳跃不能超过固定距离, 且需要满足特定条件
// 4. LeetCode 1306 - Jump Game III
//     链接: https://leetcode.cn/problems/jump-game-iii/
//     区别: 在数组中跳跃, 从起始位置开始, 判断能否到达值为 0 的位置
}

```

文件: Code06_FrogCrossRiver.py

青蛙过河问题 - Python 实现

算法思路: 动态规划 + 距离压缩

时间复杂度: $O(n * (t-s+1))$

空间复杂度: $O(n)$

核心优化:

1. 当 $s < t$ 时, 通过实验确定安全距离, 超过这个距离后就不会再遇到新的石子
2. 对石子位置进行压缩, 减少动态规划的范围
3. 使用滑动窗口思想优化状态转移

```

import sys
from typing import List

```

MAXN = 101

MAXL = 100001

```
MAXK = 201
```

```
class FrogCrossRiver:
```

```
    """
```

```
青蛙过河问题解决方案
```

```
    """
```

```
def __init__(self):
```

```
    self.arr = [0] * MAXN
```

```
    self.where = [0] * MAXN
```

```
    self.dp = [0] * MAXL
```

```
    self.stone = [False] * MAXL
```

```
    self.reach = [False] * MAXK
```

```
    self.n = 0
```

```
    self.s = 0
```

```
    self.t = 0
```

```
    self.m = 0
```

```
    self.safe = 0
```

```
def reduce(self, s: int, t: int) -> int:
```

```
    """
```

```
计算安全距离
```

```
一旦 s 和 t 定了，那么距离多远就可以缩减呢？
```

```
通过实验确定安全距离
```

```
Args:
```

```
    s: 最小跳跃距离
```

```
    t: 最大跳跃距离
```

```
Returns:
```

```
    int: 安全距离
```

```
    """
```

```
# 重置 reach 数组
```

```
self.reach = [False] * MAXK
```

```
cnt = 0
```

```
ans = 0
```

```
for i in range(MAXK):
```

```
    # 标记可达位置
```

```
    for j in range(i + s, min(i + t + 1, MAXK)):
```

```
        self.reach[j] = True
```

```
# 统计连续可达位置数量
```

```

    if not self.reach[i]:
        cnt = 0
    else:
        cnt += 1

    # 如果连续 t 个位置都可达，则找到安全距离
    if cnt == t:
        ans = i
        break

return ans

def compute(self) -> int:
    """
    计算青蛙过河最少踩到的石子数

    Returns:
        int: 最少踩到的石子数
    """
    # 对石子位置进行排序
    self.arr[1:self.m+1] = sorted(self.arr[1:self.m+1])

    # 特殊情况: s == t
    if self.s == self.t:
        ans = 0
        for i in range(1, self.m + 1):
            if self.arr[i] % self.s == 0:
                ans += 1
        return ans
    else: # s < t
        # 计算安全距离
        self.safe = self.reduce(self.s, self.t)

        # 重新计算石子位置
        self.where[0] = 0
        for i in range(1, self.m + 1):
            self.where[i] = self.where[i-1] + min(self.arr[i] - self.arr[i-1], self.safe)
            self.stone[self.where[i]] = True

        # 更新桥的长度
        self.n = self.where[self.m] + self.safe

    # 初始化 dp 数组

```

```
for i in range(1, self.n + 1):
    self.dp[i] = MAXN
self.dp[0] = 0

# 动态规划
for i in range(1, self.n + 1):
    for j in range(max(i - self.t, 0), i - self.s + 1):
        self.dp[i] = min(self.dp[i], self.dp[j] + (1 if self.stone[i] else 0))

# 找到最小值
ans = MAXN
for i in range(self.where[self.m] + 1, self.n + 1):
    ans = min(ans, self.dp[i])

return ans

def test(self):
    """
    单元测试函数
    """
    # 测试用例 1: 基础测试
    self.n = 10
    self.s = 2
    self.t = 3
    self.m = 2
    self.arr[1] = 2
    self.arr[2] = 3

    result1 = self.compute()
    print(f"Test 1 - Basic: {result1}")

    # 测试用例 2: 边界测试 - s == t
    self.n = 10
    self.s = 2
    self.t = 2
    self.m = 3
    self.arr[1] = 2
    self.arr[2] = 4
    self.arr[3] = 6

    result2 = self.compute()
    print(f"Test 2 - s == t: {result2}")
```

```
# 测试用例 3: 无石子
self.n = 10
self.s = 2
self.t = 3
self.m = 0

result3 = self.compute()
print(f"Test 3 - No stones: {result3}")

def main():
    """
    主函数: 从标准输入读取数据并计算结果
    """
    frog = FrogCrossRiver()

    # 从标准输入读取数据
    data = sys.stdin.read().split()
    if len(data) < 4:
        print("输入数据格式错误")
        return

    frog.n = int(data[0])
    frog.s = int(data[1])
    frog.t = int(data[2])
    frog.m = int(data[3])

    for i in range(frog.m):
        frog.arr[i+1] = int(data[4+i])

    result = frog.compute()
    print(result)

if __name__ == "__main__":
    # 运行主函数
    main()

    # 运行单元测试
    # frog = FrogCrossRiver()
    # frog.test()

"""

相关题目扩展:
```

1. LeetCode 403 - Frog Jump (青蛙跳)

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

区别: 青蛙在河中跳跃, 每个位置可能有石头, 需要判断能否到达最后一块石头

解法: 使用哈希表记录每个位置可以跳跃的距离

2. LeetCode 1340 - Jump Game V (跳跃游戏 V)

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

区别: 在数组中跳跃, 每次跳跃不能超过固定距离, 且需要满足特定条件

解法: 动态规划 + 单调栈

3. LeetCode 1306 - Jump Game III (跳跃游戏 III)

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

区别: 在数组中跳跃, 从起始位置开始, 判断能否到达值为 0 的位置

解法: BFS 或 DFS

4. Codeforces 965D - Single Wildcard Pattern Matching

链接: <https://codeforces.com/problemset/problem/965/D>

区别: 青蛙在河中跳跃, 河中有一些石头, 需要计算能否到达对岸

5. LeetCode 45 - Jump Game II (跳跃游戏 II)

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

区别: 计算到达数组末尾的最少跳跃次数

解法: 贪心算法

6. LeetCode 55 - Jump Game (跳跃游戏)

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

区别: 判断能否到达数组末尾

解法: 贪心算法

7. LeetCode 1696 - Jump Game VI (跳跃游戏 VI)

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

区别: 在数组中跳跃, 每次跳跃有最大距离限制, 需要最大化得分

解法: 动态规划 + 单调队列

8. LeetCode 1871 - Jump Game VII (跳跃游戏 VII)

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

区别: 在二进制字符串中跳跃, 需要判断能否到达末尾

解法: BFS 或滑动窗口

9. LeetCode 1345 - Jump Game IV (跳跃游戏 IV)

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

区别: 在数组中跳跃, 可以跳到相同值的位置

解法: BFS + 哈希表

10. LeetCode 1346 – Jump Game V (跳跃游戏 V)

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

区别: 在数组中跳跃, 有最大跳跃距离限制

解法: 动态规划 + 排序

算法技巧总结:

1. 动态规划是解决跳跃类问题的核心方法
2. 当数据规模较大时, 需要考虑距离压缩等优化技巧
3. 对于特殊边界情况 (如 $s==t$) 需要单独处理
4. 滑动窗口思想可以优化状态转移过程
5. 实验法确定安全距离是本题的关键优化点

工程化考量:

1. 异常处理: 输入数据合法性验证
2. 性能优化: 避免重复计算, 使用合适的数据结构
3. 内存管理: 合理分配数组大小, 避免内存泄漏
4. 测试覆盖: 包含边界测试、性能测试、异常测试

Python 语言特性:

1. 使用类型注解提高代码可读性
2. 利用 Python 内置排序函数简化代码
3. 使用列表推导式和生成器表达式优化性能
4. 注意 Python 的索引从 0 开始, 与 Java/C++不同

调试技巧:

1. 使用 `print` 语句输出中间变量值
2. 利用 Python 的调试器 `pdb` 进行调试
3. 编写单元测试验证算法正确性
4. 使用性能分析工具优化代码效率

"""

文件: Code07_CherryPickupII.cpp

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

using namespace std;
```

```

/***
 * 樱桃采摘 II 问题 - C++实现
 * 算法思路：三维动态规划
 * 时间复杂度：O(rows * cols^2)
 * 空间复杂度：O(rows * cols^2)，可优化到 O(cols^2)
 *
 * 核心思想：
 * 1. 使用三维 DP 数组 dp[i][j1][j2] 表示当机器人 1 在 (i, j1)，机器人 2 在 (i, j2) 时能收集的最大樱桃数
 * 2. 状态转移：考虑两个机器人从前一行的 9 种可能位置组合转移而来
 * 3. 关键优化：如果两个机器人在同一位置，只计算一次樱桃数量
 */

```

```

class CherryPickupII {
public:
    /**
     * 计算两个机器人能收集的最大樱桃数
     * @param grid 二维矩阵，表示樱桃田
     * @return 最大樱桃数
     */
    int cherryPickup(vector<vector<int>>& grid) {
        int rows = grid.size();
        if (rows == 0) return 0;
        int cols = grid[0].size();

        // 输入验证
        if (cols == 0) return 0;

        // 创建三维 DP 数组，初始化为 -1 表示未访问
        vector<vector<vector<int>>> dp(rows,
            vector<vector<int>>(cols,
                vector<int>(cols, -1)));
    }

    // 初始化起始位置
    dp[0][0][cols - 1] = grid[0][0] + grid[0][cols - 1];

    // 填充 DP 表
    for (int i = 1; i < rows; i++) {
        for (int j1 = 0; j1 < cols; j1++) {
            for (int j2 = 0; j2 < cols; j2++) {
                // 计算当前位置的樱桃数
                int cherries = (j1 == j2) ? grid[i][j1] : (grid[i][j1] + grid[i][j2]);

                // 检查所有可能的前驱状态
                for (int prev_i = max(0, i - 1); prev_i <= i - 1; prev_i++) {
                    for (int prev_j1 = max(0, j1 - 1); prev_j1 <= min(j1, cols - 1); prev_j1++) {
                        for (int prev_j2 = max(0, j2 - 1); prev_j2 <= min(j2, cols - 1); prev_j2++) {
                            if (dp[prev_i][prev_j1][prev_j2] != -1) {
                                dp[i][j1][j2] = max(dp[i][j1][j2], dp[prev_i][prev_j1][prev_j2] + cherries);
                            }
                        }
                    }
                }
            }
        }
    }
}

```

```

        for (int p1 = j1 - 1; p1 <= j1 + 1; p1++) {
            for (int p2 = j2 - 1; p2 <= j2 + 1; p2++) {
                // 检查前驱状态是否有效
                if (p1 >= 0 && p1 < cols && p2 >= 0 && p2 < cols && dp[i - 1][p1][p2] != -1) {
                    dp[i][j1][j2] = max(dp[i][j1][j2], dp[i - 1][p1][p2] + cherries);
                }
            }
        }
    }

// 找到最后一行的最大值
int result = 0;
for (int j1 = 0; j1 < cols; j1++) {
    for (int j2 = 0; j2 < cols; j2++) {
        result = max(result, dp[rows - 1][j1][j2]);
    }
}

return result;
}

/**
 * 空间优化版本：使用滚动数组将空间复杂度优化到 O(cols^2)
 * @param grid 二维矩阵
 * @return 最大樱桃数
 */
int cherryPickupOptimized(vector<vector<int>>& grid) {
    int rows = grid.size();
    if (rows == 0) return 0;
    int cols = grid[0].size();

    // 输入验证
    if (cols == 0) return 0;

    // 使用滚动数组，只需要保存当前行和上一行的状态
    vector<vector<int>> prev(cols, vector<int>(cols, -1));
    vector<vector<int>> curr(cols, vector<int>(cols, -1));

    // 初始化起始位置
    prev[0][cols - 1] = grid[0][0] + grid[0][cols - 1];
}

```

```

for (int i = 1; i < rows; i++) {
    // 清空当前行状态
    for (int j1 = 0; j1 < cols; j1++) {
        for (int j2 = 0; j2 < cols; j2++) {
            curr[j1][j2] = -1;
        }
    }

    for (int j1 = 0; j1 < cols; j1++) {
        for (int j2 = 0; j2 < cols; j2++) {
            // 计算当前位置的樱桃数
            int cherries = (j1 == j2) ? grid[i][j1] : (grid[i][j1] + grid[i][j2]);

            // 检查所有可能的前驱状态
            for (int p1 = j1 - 1; p1 <= j1 + 1; p1++) {
                for (int p2 = j2 - 1; p2 <= j2 + 1; p2++) {
                    if (p1 >= 0 && p1 < cols && p2 >= 0 && p2 < cols && prev[p1][p2] != -1) {
                        curr[j1][j2] = max(curr[j1][j2], prev[p1][p2] + cherries);
                    }
                }
            }
        }
    }
}

// 交换 prev 和 curr
swap(prev, curr);
}

// 找到最大值
int result = 0;
for (int j1 = 0; j1 < cols; j1++) {
    for (int j2 = 0; j2 < cols; j2++) {
        result = max(result, prev[j1][j2]);
    }
}

return result;
}

/**
 * 单元测试函数

```

```

*/
void test() {
    // 测试用例 1
    vector<vector<int>> grid1 = {
        {3, 1, 1},
        {2, 5, 1},
        {1, 5, 5}
    };
    int result1 = cherryPickup(grid1);
    cout << "Test 1 - Basic: " << result1 << " (Expected: 21)" << endl;

    // 测试用例 2
    vector<vector<int>> grid2 = {
        {1, 0, 0, 0, 0, 0, 1},
        {0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0},
        {1, 0, 0, 0, 0, 0, 1}
    };
    int result2 = cherryPickup(grid2);
    cout << "Test 2 - Large Grid: " << result2 << " (Expected: 8)" << endl;

    // 测试用例 3: 边界情况 - 单行
    vector<vector<int>> grid3 = {
        {1, 0, 1}
    };
    int result3 = cherryPickup(grid3);
    cout << "Test 3 - Single Row: " << result3 << " (Expected: 2)" << endl;

    // 测试优化版本
    int result10pt = cherryPickupOptimized(grid1);
    cout << "Test 1 - Optimized: " << result10pt << " (Expected: 21)" << endl;

    // 性能测试: 大规模数据
    vector<vector<int>> largeGrid(50, vector<int>(50, 1));
    largeGrid[0][0] = 0;
    largeGrid[0][49] = 0;

    cout << "Performance test started..." << endl;
    int largeResult = cherryPickupOptimized(largeGrid);
    cout << "Performance test completed. Result: " << largeResult << endl;
}

```

```
}

};

int main() {
    CherryPickupII solution;

    // 运行单元测试
    solution.test();

    return 0;
}

/*
 * 相关题目扩展:
 *
 * 1. LeetCode 741 - Cherry Pickup (摘樱桃 I)
 *   链接: https://leetcode.cn/problems/cherry-pickup/
 *   区别: 一个人从(0, 0)走到(n-1, n-1)再走回(0, 0), 求最大收集樱桃数
 *   算法: 三维动态规划, 转化为两个人同时从起点到终点的问题
 *
 * 2. LeetCode 64 - Minimum Path Sum (最小路径和)
 *   链接: https://leetcode.cn/problems/minimum-path-sum/
 *   区别: 求从左上角到右下角的最小路径和, 每步只能向下或向右
 *   算法: 二维动态规划
 *
 * 3. LeetCode 174 - Dungeon Game (地下城游戏)
 *   链接: https://leetcode.cn/problems/dungeon-game/
 *   区别: 骑士需要从左上角到右下角, 保证健康点数始终大于 0 的最小初始值
 *   算法: 从右下角向左上角动态规划
 *
 * 4. LeetCode 62 - Unique Paths (不同路径)
 *   链接: https://leetcode.cn/problems/unique-paths/
 *   区别: 计算从左上角到右下角的不同路径数量
 *   算法: 组合数学或二维动态规划
 *
 * 5. LeetCode 63 - Unique Paths II (不同路径 II)
 *   链接: https://leetcode.cn/problems/unique-paths-ii/
 *   区别: 网格中有障碍物, 计算不同路径数量
 *   算法: 二维动态规划, 遇到障碍物时  $dp[i][j] = 0$ 
 *
 * 6. Codeforces 1296D - Fight with Monsters
 *   链接: https://codeforces.com/problemset/problem/1296/D
 *   区别: 贪心策略解决怪物战斗问题

```

- * 算法：排序后贪心选择最优策略
- *
- * 7. AtCoder ABC159E - Dividing Chocolate
- * 链接：https://atcoder.jp/contests/abc159/tasks/abc159_e
- * 区别：二维网格分割问题
- * 算法：前缀和+状态压缩动态规划
- *
- * 8. 洛谷 P1434 - [SHOI2002] 滑雪
- * 链接：<https://www.luogu.com.cn/problem/P1434>
- * 区别：寻找最长滑雪路径
- * 算法：记忆化搜索或拓扑排序动态规划
- *
- * 9. 牛客网 NC14552 - 方格取数
- * 链接：<https://ac.nowcoder.com/acm/problem/14552>
- * 区别：与摘樱桃 I 非常相似
- * 算法：三维动态规划
- *
- * 10. UVa 10913 - Walking on a Grid
- * 链接：

https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=1854

- * 区别：在网格中行走，有负数，最多允许 k 次负数
- * 算法：四维动态规划
- *
- * 算法技巧总结：
- * 1. 三维状态设计：同时跟踪两个机器人的位置
- * 2. 状态转移优化：考虑所有可能的前驱状态组合
- * 3. 避免重复计算：同一位置的樱桃只计算一次
- * 4. 边界条件处理：仔细检查机器人移动是否越界
- * 5. 空间优化：使用滚动数组优化空间复杂度
- *

- * C++工程化考量：
- * 1. 内存管理：使用 vector 避免手动内存管理
- * 2. 性能优化：使用引用避免不必要的拷贝
- * 3. 异常安全：使用 RAI 原则管理资源
- * 4. 代码可读性：使用有意义的变量名和注释
- * 5. 测试覆盖：包含边界测试、性能测试
- *

- * 调试技巧：
- * 1. 使用 cout 输出中间变量值
- * 2. 使用 gdb 调试器进行调试
- * 3. 编写单元测试验证算法正确性
- * 4. 使用性能分析工具优化代码效率

```
*/
```

```
=====
```

文件: Code07_CherryPickupII.java

```
=====
```

```
package class127;
```

```
// Cherry Pickup II
```

```
// 给定一个 rows x cols 的矩阵 grid, 表示一个樱桃田, grid[i][j] 表示在位置 (i, j) 的樱桃数量
```

```
// 两个机器人同时从矩阵顶部开始收集樱桃:
```

```
// - 机器人 1 从位置 (0, 0) 开始
```

```
// - 机器人 2 从位置 (0, cols-1) 开始
```

```
// 两个机器人的移动规则:
```

```
// - 从任何单元格 (i, j) 出发, 机器人可以移动到下一行的三个单元格之一:
```

```
// (i+1, j-1), (i+1, j), (i+1, j+1)
```

```
// - 机器人不能移动到矩阵边界之外
```

```
// - 两个机器人都必须到达最后一行
```

```
// 樱桃收集规则:
```

```
// - 当机器人经过一个单元格时, 它会收集该单元格的所有樱桃
```

```
// - 收集后, 该单元格变为空 (0 个樱桃)
```

```
// - 如果两个机器人同时占据同一单元格, 则只有一个机器人收集该单元格的樱桃
```

```
// 目标是找到两个机器人一起收集的最大樱桃总数
```

```
// 测试链接 : https://leetcode.cn/problems/cherry-pickup-ii/
```

```
/**
```

```
* 算法思路深度解析:
```

```
* 1. 这是一个典型的三维动态规划问题, 两个机器人同时从顶部向底部移动
```

```
* - 与摘樱桃 I 不同, 这里两个机器人有明确的起始位置和移动规则
```

```
* - 问题的关键是如何有效跟踪两个机器人的路径并避免重复计算
```

```
* 2. 使用三维动态规划 dp[i][j1][j2], 其中:
```

```
* - i 表示当前行
```

```
* - j1 表示机器人 1 的列位置
```

```
* - j2 表示机器人 2 的列位置
```

```
* - dp[i][j1][j2] 表示当机器人 1 在(i, j1), 机器人 2 在(i, j2)时能收集的最大樱桃数
```

```
* 3. 状态转移方程:
```

```
* - 对于每个位置(i, j1, j2), 考虑两个机器人从前一行的可能位置转移而来
```

```
* - 每个机器人可以从三个位置转移: (i-1, j-1), (i-1, j), (i-1, j+1)
```

```
* - 所以总共有 3*3=9 种可能的前驱状态组合
```

```
* - 关键优化: 如果两个机器人在同一位置, 只计算一次樱桃数量
```

```
* 4. 初始化策略:
```

```
* - dp[0][0][cols-1] = grid[0][0] + grid[0][cols-1] (两个机器人的起始位置)
```

```
* - 使用-1 作为未访问状态的标记, 避免混淆
```

- * 5. 结果提取:
 - * - 在最后一行中找到所有可能位置组合的最大值
 - * - 不需要考虑边界情况，因为题目保证两个机器人都能到达最后一行
 - *
- * 时间复杂度分析:
 - * - 状态总数: $O(rows * cols^2)$
 - * - 每个状态需要考虑 9 种前驱状态
 - * - 总时间复杂度: $O(rows * cols^2 * 9) = O(rows * cols^2)$
 - *
- * 空间复杂度分析:
 - * - 三维 DP 数组大小: $O(rows * cols^2)$
 - * - 可以通过滚动数组优化到 $O(cols^2)$ ，只保存当前行和上一行的状态
 - *
- * Java 实现注意事项:
 - * 1. 数组初始化: 使用三层循环初始化三维数组为 -1
 - * 2. 边界条件处理: 确保机器人不越出矩阵边界
 - * 3. 性能优化: 可以使用滚动数组减少内存使用
 - * 4. 代码可读性: 使用有意义的变量名和适当的注释
 - *
- * 工程化考量:
 - * 1. 输入验证: 可以添加对输入矩阵的合法性检查
 - * 2. 异常处理: 处理空矩阵或特殊情况
 - * 3. 测试用例: 覆盖各种边界情况和典型场景
 - * 4. 空间优化: 对于大规模数据, 考虑使用滚动数组
 - * 5. 多线程优化: 对于非常大的矩阵, 可以考虑并行计算

*/

```

public class Code07_CherryPickupII {

    /**
     * 计算两个机器人能收集的最大樱桃数
     * @param grid rows x cols 的矩阵
     * @return 两个机器人能收集的最大樱桃数
     */
    public static int cherryPickup(int[][] grid) {
        int rows = grid.length;
        int cols = grid[0].length;

        // dp[i][j1][j2] 表示当机器人 1 在(i, j1), 机器人 2 在(i, j2)时能收集的最大樱桃数
        int[][][] dp = new int[rows][cols][cols];

        // 初始化 dp 数组为 -1, 表示未访问过
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                for (int k = 0; k < cols; k++) {
                    dp[i][j][k] = -1;
                }
            }
        }

        // 从第一行开始遍历
        for (int i = 0; i < rows; i++) {
            for (int j1 = 0; j1 < cols; j1++) {
                for (int j2 = 0; j2 < cols; j2++) {
                    if (j1 == j2) {
                        dp[i][j1][j2] = grid[i][j1];
                    } else {
                        dp[i][j1][j2] = Math.max(dp[i][j1][j2], dp[i][j1][j2] + grid[i][j1]);
                    }
                }
            }
        }

        // 找到最后一行的最大值
        int maxCherry = 0;
        for (int j1 = 0; j1 < cols; j1++) {
            for (int j2 = 0; j2 < cols; j2++) {
                maxCherry = Math.max(maxCherry, dp[rows - 1][j1][j2]);
            }
        }

        return maxCherry;
    }
}

```

```

        for (int k = 0; k < cols; k++) {
            dp[i][j][k] = -1;
        }
    }

// 初始化起始位置
dp[0][0][cols - 1] = grid[0][0] + grid[0][cols - 1];

// 填充 dp 表
for (int i = 1; i < rows; i++) {
    for (int j1 = 0; j1 < cols; j1++) {
        for (int j2 = 0; j2 < cols; j2++) {
            // 计算当前位置的樱桃数
            int cherries = (j1 == j2) ? grid[i][j1] : (grid[i][j1] + grid[i][j2]);

            // 检查所有可能的前驱状态
            for (int p1 = j1 - 1; p1 <= j1 + 1; p1++) {
                for (int p2 = j2 - 1; p2 <= j2 + 1; p2++) {
                    // 检查前驱状态是否有效
                    if (p1 >= 0 && p1 < cols && p2 >= 0 && p2 < cols && dp[i - 1][p1][p2] != -1) {
                        dp[i][j1][j2] = Math.max(dp[i][j1][j2], dp[i - 1][p1][p2] + cherries);
                    }
                }
            }
        }
    }
}

// 找到最后一行的最大值
int result = 0;
for (int j1 = 0; j1 < cols; j1++) {
    for (int j2 = 0; j2 < cols; j2++) {
        result = Math.max(result, dp[rows - 1][j1][j2]);
    }
}

return result;
}

// 测试方法

```

```

public static void main(String[] args) {
    // 测试用例 1
    int[][] grid1 = {
        {3, 1, 1},
        {2, 5, 1},
        {1, 5, 5}
    };
    System.out.println("测试用例 1 结果: " + cherryPickup(grid1)); // 预期输出: 21

    // 测试用例 2
    int[][] grid2 = {
        {1, 0, 0, 0, 0, 0, 1},
        {0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0},
        {1, 0, 0, 0, 0, 0, 1}
    };
    System.out.println("测试用例 2 结果: " + cherryPickup(grid2)); // 预期输出: 8

    // 测试用例 3
    int[][] grid3 = {
        {0, 1, 1, 1, 1, 1, 1, 1, 1, 1},
        {1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
        {1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
        {1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
        {1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
        {1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
        {1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
        {1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
        {1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
        {1, 1, 1, 1, 1, 1, 1, 1, 1, 0}
    };
    System.out.println("测试用例 3 结果: " + cherryPickup(grid3)); // 预期输出: 94
}

```

// 类似题目与训练拓展:

```

// 1. LeetCode 741. Cherry Pickup (摘樱桃 I)
// 链接: https://leetcode.cn/problems/cherry-pickup/
// 区别: 一个人从(0, 0)走到(n-1, n-1)再走回(0, 0), 求最大收集樱桃数
// 算法: 三维动态规划, 转化为两个人同时从起点到终点的问题
// 
```

```
// 2. LeetCode 64. Minimum Path Sum (最小路径和)
//     链接: https://leetcode.cn/problems/minimum-path-sum/
//     区别: 求从左上角到右下角的最小路径和, 每步只能向下或向右
//     算法: 二维动态规划,  $dp[i][j] = \min(dp[i-1][j], dp[i][j-1]) + grid[i][j]$ 
//
// 3. LeetCode 174. Dungeon Game (地下城游戏)
//     链接: https://leetcode.cn/problems/dungeon-game/
//     区别: 骑士需要从左上角到右下角, 保证健康点数始终大于 0 的最小初始值
//     算法: 从右下角向左上角动态规划,  $dp[i][j] = \max(1, \min(dp[i+1][j], dp[i][j+1])) - dungeon[i][j]$ 
//
// 4. LeetCode 1463. Cherry Pickup II (本题)
//     链接: https://leetcode.cn/problems/cherry-pickup-ii/
//     算法: 三维动态规划, 两个机器人同时移动的路径规划问题
//
// 5. LeetCode 62. Unique Paths (不同路径)
//     链接: https://leetcode.cn/problems/unique-paths/
//     区别: 计算从左上角到右下角的不同路径数量, 每步只能向下或向右
//     算法: 组合数学或二维动态规划
//
// 6. LeetCode 63. Unique Paths II (不同路径 II)
//     链接: https://leetcode.cn/problems/unique-paths-ii/
//     区别: 网格中有障碍物, 计算不同路径数量
//     算法: 二维动态规划, 遇到障碍物时  $dp[i][j] = 0$ 
//
// 7. Codeforces 1296D - Fight with Monsters
//     链接: https://codeforces.com/problemset/problem/1296/D
//     区别: 贪心策略解决怪物战斗问题, 但状态转移思想类似
//     算法: 排序后贪心选择最优策略
//
// 8. AtCoder ABC159E - Dividing Chocolate
//     链接: https://atcoder.jp/contests/abc159/tasks/abc159\_e
//     区别: 二维网格分割问题, 但需要类似的状态转移思路
//     算法: 前缀和+状态压缩动态规划
//
// 9. 洛谷 P1434 [SHOI2002] 滑雪
//     链接: https://www.luogu.com.cn/problem/P1434
//     区别: 寻找最长滑雪路径, 每步只能滑向相邻四个方向且高度更低的位置
//     算法: 记忆化搜索或拓扑排序动态规划
//
// 10. 牛客网 NC14552 方格取数
//     链接: https://ac.nowcoder.com/acm/problem/14552
//     区别: 与摘樱桃 I 非常相似, 也是两个人从左上角出发到右下角取数
```

```
// 算法：三维动态规划，状态定义与摘樱桃 I 类似
//
// 11. UVa 10913 - Walking on a Grid
// 链接：
https://onlinejudge.org/index.php?option=com\_onlinejudge&Itemid=8&category=24&page=show\_problem&problem=1854
// 区别：在网格中行走，有负数，最多允许 k 次负数
// 算法：四维动态规划，状态定义为  $dp[i][j][k][d]$ ，表示在  $(i, j)$  位置，已经经过 k 次负数，方向为 d
//
// 12. SPOJ - SBANK
// 链接：https://www.spoj.com/problems/SBANK/
// 区别：银行账号排序问题，使用哈希表优化
// 算法：哈希表+排序
//
// 13. HackerEarth - Roy and Coin Boxes
// 链接：https://www.hackerearth.com/practice/data-structures/arrays/1-d/practice-problems/algorithm/roy-and-coin-boxes-1/
// 区别：区间更新问题，使用差分数组优化
// 算法：差分数组+前缀和
//
// 14. 杭电 HDU 1024 - Max Sum Plus Plus
// 链接：http://acm.hdu.edu.cn/showproblem.php?pid=1024
// 区别：最大 m 段子数组和问题
// 算法：二维动态规划优化为一维
//
// 15. Codeforces 1295E - Permutation Separation
// 链接：https://codeforces.com/problemset/problem/1295/E
// 区别：排列分割问题，需要找到最优分割点
// 算法：前缀和+动态规划
//
// 算法本质与技巧总结：
// 1. 三维状态设计：同时跟踪两个机器人的位置，有效表达复杂状态
// 2. 状态转移优化：考虑所有可能的前驱状态组合
// 3. 避免重复计算：同一位置的樱桃只计算一次
// 4. 边界条件处理：仔细检查机器人移动是否越界
// 5. 空间优化：可以使用滚动数组优化空间复杂度
//
// Java 工程化实战建议：
// 1. 测试用例设计：
//   - 空矩阵或单行矩阵
//   - 两个机器人起始位置重合的情况（但题目保证起始位置不同）
//   - 矩阵中有大量 0 或负数的情况
```

```
//      - 最优路径需要两个机器人交叉的情况
// 2. 性能优化:
//      - 使用滚动数组将空间复杂度从 O(rows*cols^2) 优化到 O(cols^2)
//      - 预先计算矩阵的行数和列数，避免重复访问数组长度
//      - 考虑使用并行流处理大规模数据
// 3. 代码健壮性:
//      - 添加输入合法性检查
//      - 使用 try-catch 块处理可能的异常
//      - 考虑使用 Optional 类型处理可能的空结果
// 4. 调试技巧:
//      - 添加打印语句输出中间状态
//      - 使用断言验证关键条件
//      - 考虑使用 JDB 调试器
// 5. Java 特性应用:
//      - 可以使用 Stream API 简化最后一行最大值的计算
//      - 考虑使用数组初始化工具类简化三维数组的初始化
//      - 可以使用 Enum 定义机器人的移动方向
}
```

=====

文件: Code07_CherryPickupII.py

=====

```
"""
樱桃采摘 II 问题 - Python 实现
算法思路: 三维动态规划
时间复杂度: O(rows * cols^2)
空间复杂度: O(rows * cols^2)，可优化到 O(cols^2)
"""

核心思想:
```

1. 使用三维 DP 数组 $dp[i][j1][j2]$ 表示当机器人 1 在 $(i, j1)$ ，机器人 2 在 $(i, j2)$ 时能收集的最大樱桃数
2. 状态转移: 考虑两个机器人从前一行的 9 种可能位置组合转移而来
3. 关键优化: 如果两个机器人在同一位置，只计算一次樱桃数量

```
from typing import List
import sys

class CherryPickupII:
    """
    樱桃采摘 II 问题解决方案
    """

```

```

def cherryPickup(self, grid: List[List[int]]) -> int:
    """
    计算两个机器人能收集的最大樱桃数

    Args:
        grid: 二维矩阵，表示樱桃田

    Returns:
        int: 最大樱桃数
    """

    rows = len(grid)
    if rows == 0:
        return 0
    cols = len(grid[0])

    # 输入验证
    if cols == 0:
        return 0

    # 创建三维 DP 数组，初始化为-1 表示未访问
    dp = [[[ -1] * cols for _ in range(cols)] for _ in range(rows)]

    # 初始化起始位置
    dp[0][0][cols - 1] = grid[0][0] + grid[0][cols - 1]

    # 填充 DP 表
    for i in range(1, rows):
        for j1 in range(cols):
            for j2 in range(cols):
                # 计算当前位置的樱桃数
                cherries = grid[i][j1] if j1 == j2 else grid[i][j1] + grid[i][j2]

                # 检查所有可能的前驱状态
                for p1 in range(max(0, j1 - 1), min(cols, j1 + 2)):
                    for p2 in range(max(0, j2 - 1), min(cols, j2 + 2)):
                        # 检查前驱状态是否有效
                        if dp[i - 1][p1][p2] != -1:
                            current_val = dp[i - 1][p1][p2] + cherries
                            if dp[i][j1][j2] < current_val:
                                dp[i][j1][j2] = current_val

    # 找到最后一行的最大值
    result = 0

```

```

for j1 in range(cols):
    for j2 in range(cols):
        if dp[rows - 1][j1][j2] > result:
            result = dp[rows - 1][j1][j2]

return result

def cherryPickupOptimized(self, grid: List[List[int]]) -> int:
    """
    空间优化版本：使用滚动数组将空间复杂度优化到 O(cols^2)

    Args:
        grid: 二维矩阵

    Returns:
        int: 最大樱桃数
    """
    rows = len(grid)
    if rows == 0:
        return 0
    cols = len(grid[0])

    # 输入验证
    if cols == 0:
        return 0

    # 使用滚动数组，只需要保存当前行和上一行的状态
    prev = [[-1] * cols for _ in range(cols)]
    curr = [[-1] * cols for _ in range(cols)]

    # 初始化起始位置
    prev[0][cols - 1] = grid[0][0] + grid[0][cols - 1]

    for i in range(1, rows):
        # 清空当前行状态
        for j1 in range(cols):
            for j2 in range(cols):
                curr[j1][j2] = -1

        for j1 in range(cols):
            for j2 in range(cols):
                # 计算当前位置的樱桃数
                cherries = grid[i][j1] if j1 == j2 else grid[i][j1] + grid[i][j2]

```



```

        ]
result2 = self.cherryPickup(grid2)
print(f"Test 2 - Large Grid: {result2} (Expected: 8)")

# 测试用例 3: 边界情况 - 单行
grid3 = [
    [1, 0, 1]
]
result3 = self.cherryPickup(grid3)
print(f"Test 3 - Single Row: {result3} (Expected: 2)")

# 测试优化版本
result10pt = self.cherryPickupOptimized(grid1)
print(f"Test 1 - Optimized: {result10pt} (Expected: 21)")

# 性能测试: 大规模数据
largeGrid = [[1] * 50 for _ in range(50)]
largeGrid[0][0] = 0
largeGrid[0][49] = 0

print("Performance test started...")
largeResult = self.cherryPickupOptimized(largeGrid)
print(f"Performance test completed. Result: {largeResult}")

def main():
    """
    主函数
    """
    solution = CherryPickupII()

    # 运行单元测试
    solution.test()

if __name__ == "__main__":
    main()

```

"""

相关题目扩展:

1. LeetCode 741 - Cherry Pickup (摘樱桃 I)

链接: <https://leetcode.cn/problems/cherry-pickup/>

区别: 一个人从(0, 0)走到(n-1, n-1)再走回(0, 0), 求最大收集樱桃数

算法: 三维动态规划, 转化为两个人同时从起点到终点的问题

2. LeetCode 64 – Minimum Path Sum (最小路径和)

链接: <https://leetcode.cn/problems/minimum-path-sum/>

区别: 求从左上角到右下角的最小路径和, 每步只能向下或向右

算法: 二维动态规划

3. LeetCode 174 – Dungeon Game (地下城游戏)

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

区别: 骑士需要从左上角到右下角, 保证健康点数始终大于 0 的最小初始值

算法: 从右下角向左上角动态规划

4. LeetCode 62 – Unique Paths (不同路径)

链接: <https://leetcode.cn/problems/unique-paths/>

区别: 计算从左上角到右下角的不同路径数量

算法: 组合数学或二维动态规划

5. LeetCode 63 – Unique Paths II (不同路径 II)

链接: <https://leetcode.cn/problems/unique-paths-ii/>

区别: 网格中有障碍物, 计算不同路径数量

算法: 二维动态规划, 遇到障碍物时 $dp[i][j] = 0$

6. Codeforces 1296D – Fight with Monsters

链接: <https://codeforces.com/problemset/problem/1296/D>

区别: 贪心策略解决怪物战斗问题

算法: 排序后贪心选择最优策略

7. AtCoder ABC159E – Dividing Chocolate

链接: https://atcoder.jp/contests/abc159/tasks/abc159_e

区别: 二维网格分割问题

算法: 前缀和+状态压缩动态规划

8. 洛谷 P1434 – [SHOI2002] 滑雪

链接: <https://www.luogu.com.cn/problem/P1434>

区别: 寻找最长滑雪路径

算法: 记忆化搜索或拓扑排序动态规划

9. 牛客网 NC14552 – 方格取数

链接: <https://ac.nowcoder.com/acm/problem/14552>

区别: 与摘樱桃 I 非常相似

算法: 三维动态规划

10. UVa 10913 – Walking on a Grid

链接:

https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=1854

区别：在网格中行走，有负数，最多允许 k 次负数

算法：四维动态规划

算法技巧总结：

1. 三维状态设计：同时跟踪两个机器人的位置
2. 状态转移优化：考虑所有可能的前驱状态组合
3. 避免重复计算：同一位置的樱桃只计算一次
4. 边界条件处理：仔细检查机器人移动是否越界
5. 空间优化：使用滚动数组优化空间复杂度

Python 工程化考量：

1. 类型注解：使用 typing 模块提高代码可读性
2. 性能优化：使用列表推导式代替循环
3. 内存管理：Python 有自动垃圾回收，但要注意避免循环引用
4. 代码可读性：使用有意义的变量名和注释
5. 测试覆盖：包含边界测试、性能测试

调试技巧：

1. 使用 print 语句输出中间变量值
2. 使用 pdb 调试器进行调试
3. 编写单元测试验证算法正确性
4. 使用性能分析工具优化代码效率

Python 语言特性：

1. 列表推导式：可以简化数组初始化
2. 切片操作：可以方便地处理数组范围
3. 内置函数：max、min、range 等函数优化了性能
4. 动态类型：不需要声明变量类型，但建议使用类型注解

异常处理：

1. 输入验证：检查矩阵是否为空
2. 边界检查：确保索引不越界
3. 类型检查：确保输入参数类型正确
4. 错误处理：使用 try-except 处理可能的异常

""

=====

文件：Code08_UniquePathsII.cpp

=====

```
#include <iostream>
```

```

#include <vector>
#include <algorithm>

using namespace std;

/**
 * 不同路径 II 问题 - C++实现
 * 算法思路：动态规划
 * 时间复杂度：O(m * n)
 * 空间复杂度：O(m * n)，可优化到 O(n)
 *
 * 核心思想：
 * 1. 使用二维 DP 数组 dp[i][j] 表示从起点 (0, 0) 到位置 (i, j) 的不同路径数
 * 2. 状态转移：如果当前位置有障碍物，路径数为 0；否则等于从上面和左边来的路径数之和
 * 3. 边界条件：起点和终点有障碍物时直接返回 0
 */

class UniquePathsII {
public:
    /**
     * 计算从左上角到右下角的不同路径数
     * @param obstacleGrid m x n 的网格，1 表示障碍物，0 表示空位置
     * @return 不同路径的数量
     */
    int uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid) {
        int m = obstacleGrid.size();
        if (m == 0) return 0;
        int n = obstacleGrid[0].size();

        // 输入验证
        if (n == 0) return 0;

        // 如果起点或终点有障碍物，直接返回 0
        if (obstacleGrid[0][0] == 1 || obstacleGrid[m-1][n-1] == 1) {
            return 0;
        }

        // dp[i][j] 表示从起点到位置 (i, j) 的不同路径数
        vector<vector<int>> dp(m, vector<int>(n, 0));

        // 初始化起点
        dp[0][0] = 1;

```

```

// 初始化第一行
for (int j = 1; j < n; j++) {
    dp[0][j] = (obstacleGrid[0][j] == 1) ? 0 : dp[0][j-1];
}

// 初始化第一列
for (int i = 1; i < m; i++) {
    dp[i][0] = (obstacleGrid[i][0] == 1) ? 0 : dp[i-1][0];
}

// 填充 dp 表
for (int i = 1; i < m; i++) {
    for (int j = 1; j < n; j++) {
        // 如果当前位置有障碍物，则路径数为 0
        if (obstacleGrid[i][j] == 1) {
            dp[i][j] = 0;
        } else {
            // 否则路径数等于从上面和左边来的路径数之和
            dp[i][j] = dp[i-1][j] + dp[i][j-1];
        }
    }
}

return dp[m-1][n-1];
}

/***
 * 空间优化版本：使用滚动数组将空间复杂度优化到 O(n)
 * @param obstacleGrid m x n 的网格
 * @return 不同路径的数量
 */
int uniquePathsWithObstaclesOptimized(vector<vector<int>>& obstacleGrid) {
    int m = obstacleGrid.size();
    if (m == 0) return 0;
    int n = obstacleGrid[0].size();

    // 输入验证
    if (n == 0) return 0;

    // 如果起点或终点有障碍物，直接返回 0
    if (obstacleGrid[0][0] == 1 || obstacleGrid[m-1][n-1] == 1) {
        return 0;
    }
}

```

```

// 使用一维数组优化空间
vector<int> dp(n, 0);

// 初始化第一行
dp[0] = 1;
for (int j = 1; j < n; j++) {
    dp[j] = (obstacleGrid[0][j] == 1) ? 0 : dp[j-1];
}

// 填充 dp 表
for (int i = 1; i < m; i++) {
    // 更新第一列
    if (obstacleGrid[i][0] == 1) {
        dp[0] = 0;
    }

    for (int j = 1; j < n; j++) {
        // 如果当前位置有障碍物，则路径数为 0
        if (obstacleGrid[i][j] == 1) {
            dp[j] = 0;
        } else {
            // 否则路径数等于从上面和左边来的路径数之和
            dp[j] = dp[j] + dp[j-1];
        }
    }
}

return dp[n-1];
}

/**
 * 单元测试函数
 */
void test() {
    // 测试用例 1
    vector<vector<int>> obstacleGrid1 = {
        {0, 0, 0},
        {0, 1, 0},
        {0, 0, 0}
    };
    int result1 = uniquePathsWithObstacles(obstacleGrid1);
    cout << "Test 1 - Basic: " << result1 << " (Expected: 2)" << endl;
}

```

```

// 测试用例 2
vector<vector<int>> obstacleGrid2 = {
    {0, 1},
    {0, 0}
};

int result2 = uniquePathsWithObstacles(obstacleGrid2);
cout << "Test 2 - Small Grid: " << result2 << " (Expected: 1)" << endl;

// 测试用例 3
vector<vector<int>> obstacleGrid3 = {
    {0, 0},
    {1, 1},
    {0, 0}
};

int result3 = uniquePathsWithObstacles(obstacleGrid3);
cout << "Test 3 - Blocked Path: " << result3 << " (Expected: 0)" << endl;

// 测试用例 4
vector<vector<int>> obstacleGrid4 = {
    {0, 0, 0, 0},
    {0, 0, 0, 0},
    {0, 0, 0, 0}
};

int result4 = uniquePathsWithObstacles(obstacleGrid4);
cout << "Test 4 - No Obstacles: " << result4 << " (Expected: 10)" << endl;

// 测试优化版本
int result10pt = uniquePathsWithObstaclesOptimized(obstacleGrid1);
cout << "Test 1 - Optimized: " << result10pt << " (Expected: 2)" << endl;

// 性能测试：大规模数据
vector<vector<int>> largeGrid(100, vector<int>(100, 0));
largeGrid[50][50] = 1; // 中间设置一个障碍物

cout << "Performance test started..." << endl;
int largeResult = uniquePathsWithObstaclesOptimized(largeGrid);
cout << "Performance test completed. Result: " << largeResult << endl;
}

int main() {
    UniquePathsII solution;

```

```
// 运行单元测试
solution.test();

return 0;
}

/*
 * 相关题目扩展:
 *
 * 1. LeetCode 62 - Unique Paths (不同路径)
 *   链接: https://leetcode.cn/problems/unique-paths/
 *   区别: 没有障碍物的网格路径问题
 *   算法: 组合数学或动态规划
 *
 * 2. LeetCode 64 - Minimum Path Sum (最小路径和)
 *   链接: https://leetcode.cn/problems/minimum-path-sum/
 *   区别: 求从左上角到右下角的最小路径和
 *   算法: 二维动态规划
 *
 * 3. LeetCode 174 - Dungeon Game (地下城游戏)
 *   链接: https://leetcode.cn/problems/dungeon-game/
 *   区别: 骑士需要从左上角到右下角, 保证健康点数始终大于 0 的最小初始值
 *   算法: 从右下角向左上角动态规划
 *
 * 4. LeetCode 63 - Unique Paths II (本题)
 *   链接: https://leetcode.cn/problems/unique-paths-ii/
 *   算法: 二维动态规划, 遇到障碍物时  $dp[i][j] = 0$ 
 *
 * 5. LeetCode 980 - Unique Paths III (不同路径 III)
 *   链接: https://leetcode.cn/problems/unique-paths-iii/
 *   区别: 需要访问所有空单元格一次且仅一次
 *   算法: 回溯+状态压缩
 *
 * 6. LeetCode 120 - Triangle (三角形最小路径和)
 *   链接: https://leetcode.cn/problems/triangle/
 *   区别: 三角形网格的最小路径和
 *   算法: 动态规划, 从底向上
 *
 * 7. LeetCode 931 - Minimum Falling Path Sum (下降路径最小和)
 *   链接: https://leetcode.cn/problems/minimum-falling-path-sum/
 *   区别: 从第一行任意位置开始, 到最后一行的最小路径和
 *   算法: 动态规划
```

*

* 8. LeetCode 1289 - Minimum Falling Path Sum II (下降路径最小和 II)
* 链接: <https://leetcode.cn/problems/minimum-falling-path-sum-ii/>
* 区别: 不能选择同一列相邻行的元素
* 算法: 动态规划+优化
*

* 9. LeetCode 1301 - Number of Paths with Max Score (最大得分路径数)
* 链接: <https://leetcode.cn/problems/number-of-paths-with-max-score/>
* 区别: 计算最大得分和对应的路径数
* 算法: 动态规划
*

* 10. LeetCode 1575 - Count All Possible Routes (统计所有可能的路线)
* 链接: <https://leetcode.cn/problems/count-all-possible-routes/>
* 区别: 在图中统计从起点到终点的所有可能路线
* 算法: 动态规划+记忆化搜索
*

* 算法技巧总结:

- * 1. 动态规划是解决网格路径问题的核心方法
- * 2. 状态转移方程: $dp[i][j] = dp[i-1][j] + dp[i][j-1]$ (无障碍物时)
- * 3. 边界条件处理: 起点和终点的特殊情况
- * 4. 空间优化: 使用滚动数组将空间复杂度从 $O(m*n)$ 优化到 $O(n)$
- * 5. 输入验证: 检查网格是否为空或起点终点是否有障碍物

*

* C++工程化考量:

- * 1. 内存管理: 使用 vector 避免手动内存管理
- * 2. 性能优化: 使用引用避免不必要的拷贝
- * 3. 异常安全: 使用 RAI^I 原则管理资源
- * 4. 代码可读性: 使用有意义的变量名和注释
- * 5. 测试覆盖: 包含边界测试、性能测试

*

* 调试技巧:

- * 1. 使用 cout 输出中间变量值
- * 2. 使用 gdb 调试器进行调试
- * 3. 编写单元测试验证算法正确性
- * 4. 使用性能分析工具优化代码效率

*/

文件: Code08_UnePathsII.java

```
package class127;
```

```

// Unique Paths II
// 一个机器人位于一个 m x n 网格的左上角，机器人每次只能向下或者向右移动一步
// 机器人试图到达网格的右下角
// 网格中有障碍物，用 1 表示，空位置用 0 表示
// 计算从左上角到右下角有多少条不同的路径
// 测试链接 : https://leetcode.cn/problems/unique-paths-ii/

/**
 * 算法思路:
 * 1. 这是一个动态规划问题，类似于 Unique Paths I，但增加了障碍物的处理
 * 2. 使用二维动态规划 dp[i][j]，表示从起点(0, 0)到位置(i, j)的不同路径数
 * 3. 状态转移方程:
 *   - 如果 obstacleGrid[i][j] == 1，表示有障碍物，dp[i][j] = 0
 *   - 如果 i == 0 且 j == 0, dp[0][0] = 1 (起点)
 *   - 如果 i == 0, 只能从左边来, dp[i][j] = dp[i][j-1]
 *   - 如果 j == 0, 只能从上面来, dp[i][j] = dp[i-1][j]
 *   - 其他情况，可以从上面或左边来, dp[i][j] = dp[i-1][j] + dp[i][j-1]
 * 4. 初始化:
 *   - 如果起点(0, 0)或终点(m-1, n-1)有障碍物，返回 0
 * 5. 结果:
 *   - dp[m-1][n-1]即为所求的不同路径数
 *
 * 时间复杂度: O(m * n)
 * 空间复杂度: O(m * n)
 */
public class Code08_UniquePathsII {

    /**
     * 计算从左上角到右下角的不同路径数
     * @param obstacleGrid m x n 的网格，1 表示障碍物，0 表示空位置
     * @return 不同路径的数量
     */
    public static int uniquePathsWithObstacles(int[][] obstacleGrid) {
        int m = obstacleGrid.length;
        int n = obstacleGrid[0].length;

        // 如果起点或终点有障碍物，直接返回 0
        if (obstacleGrid[0][0] == 1 || obstacleGrid[m-1][n-1] == 1) {
            return 0;
        }

        // dp[i][j] 表示从起点到位置(i, j)的不同路径数
        int[][] dp = new int[m][n];

```

```

// 初始化起点
dp[0][0] = 1;

// 初始化第一行
for (int j = 1; j < n; j++) {
    dp[0][j] = (obstacleGrid[0][j] == 1) ? 0 : dp[0][j-1];
}

// 初始化第一列
for (int i = 1; i < m; i++) {
    dp[i][0] = (obstacleGrid[i][0] == 1) ? 0 : dp[i-1][0];
}

// 填充 dp 表
for (int i = 1; i < m; i++) {
    for (int j = 1; j < n; j++) {
        // 如果当前位置有障碍物，则路径数为 0
        if (obstacleGrid[i][j] == 1) {
            dp[i][j] = 0;
        } else {
            // 否则路径数等于从上面和左边来的路径数之和
            dp[i][j] = dp[i-1][j] + dp[i][j-1];
        }
    }
}

return dp[m-1][n-1];
}

// 测试方法
public static void main(String[] args) {
    // 测试用例 1
    int[][] obstacleGrid1 = {
        {0, 0, 0},
        {0, 1, 0},
        {0, 0, 0}
    };
    System.out.println("测试用例 1 结果: " + uniquePathsWithObstacles(obstacleGrid1)); // 预期
输出: 2

    // 测试用例 2
    int[][] obstacleGrid2 = {

```

```

        {0, 1},
        {0, 0}
    } ;
System.out.println("测试用例 2 结果: " + uniquePathsWithObstacles(obstacleGrid2)); // 预期
输出: 1

// 测试用例 3
int[][] obstacleGrid3 = {
    {0, 0},
    {1, 1},
    {0, 0}
} ;
System.out.println("测试用例 3 结果: " + uniquePathsWithObstacles(obstacleGrid3)); // 预期
输出: 0

// 测试用例 4
int[][] obstacleGrid4 = {
    {0, 0, 0, 0},
    {0, 0, 0, 0},
    {0, 0, 0, 0}
} ;
System.out.println("测试用例 4 结果: " + uniquePathsWithObstacles(obstacleGrid4)); // 预期
输出: 10
}

// 相关题目:
// 1. LeetCode 62. Unique Paths (不同路径)
//     链接: https://leetcode.cn/problems/unique-paths/
//     区别: 没有障碍物的网格路径问题
// 2. LeetCode 64. Minimum Path Sum (最小路径和)
//     链接: https://leetcode.cn/problems/minimum-path-sum/
//     区别: 求从左上角到右下角的最小路径和
// 3. LeetCode 174. Dungeon Game (地下城游戏)
//     链接: https://leetcode.cn/problems/dungeon-game/
//     区别: 骑士需要从左上角到右下角, 保证健康点数始终大于 0 的最小初始值
}
=====

文件: Code08_UnderPathsII.py
=====

# Unique Paths II
# 一个机器人位于一个 m x n 网格的左上角, 机器人每次只能向下或者向右移动一步

```

```
# 机器人试图到达网格的右下角
# 网格中有障碍物，用 1 表示，空位置用 0 表示
# 计算从左上角到右下角有多少条不同的路径
# 测试链接 : https://leetcode.cn/problems/unique-paths-ii/
"""

算法思路深度解析:
```

1. 这是一个典型的二维动态规划问题，是 Unique Paths I 的扩展，但增加了障碍物的处理
 - 问题的核心是识别可达路径并避免障碍物
 - 动态规划的关键在于合理定义状态和转移方程，同时正确处理边界条件
2. 使用二维动态规划 $dp[i][j]$ ，其中：
 - $dp[i][j]$ 表示从起点 $(0, 0)$ 到位置 (i, j) 的不同路径数
 - 状态定义简洁明了，符合动态规划的常规模式
3. 状态转移方程的详细分析：
 - 障碍物处理：如果 $obstacleGrid[i][j] == 1$ ，表示有障碍物， $dp[i][j] = 0$
 - 起点处理：如果 $i == 0$ 且 $j == 0$ ， $dp[0][0] = 1$ （基础情况）
 - 边界处理：
 - * 第一行 ($i == 0$)：只能从左边来， $dp[i][j] = dp[i][j-1]$ ，但如果前面有障碍物则为 0
 - * 第一列 ($j == 0$)：只能从上面来， $dp[i][j] = dp[i-1][j]$ ，但如果上面有障碍物则为 0
 - 一般情况：可以从上面或左边来， $dp[i][j] = dp[i-1][j] + dp[i][j-1]$
4. 初始化策略：
 - 特殊情况检查：如果起点 $(0, 0)$ 或终点 $(m-1, n-1)$ 有障碍物，直接返回 0
 - 逐步初始化：先处理起点，再分别处理第一行和第一列，最后处理内部格子
5. 结果提取：
 - $dp[m-1][n-1]$ 即为所求的从起点到终点的不同路径数
 - 如果终点被障碍物阻塞，结果自然为 0

时间复杂度分析:

- 状态总数: $O(m * n)$
- 每个状态的计算是 $O(1)$ 操作
- 总时间复杂度: $O(m * n)$

空间复杂度分析:

- 原始实现使用二维数组: $O(m * n)$
- 可以通过滚动数组优化到 $O(\min(m, n))$
- 对于非常大的网格，空间优化尤为重要

Python 实现注意事项:

1. 初始化优化：使用列表推导式创建二维数组，简洁高效
2. 边界条件处理：正确处理障碍物对第一行和第一列初始化的影响
3. 早期返回：在检测到起点或终点有障碍物时立即返回 0
4. 内存优化：可以使用一维数组进行滚动更新，减少内存消耗
5. 输入验证：确保输入网格非空且为有效尺寸

工程化考量：

1. 异常处理：处理空网格或无效输入的情况
2. 性能优化：对于大型网格，考虑使用滚动数组优化空间
3. 代码可读性：使用清晰的变量名和适当的注释
4. 边界测试：确保算法能正确处理各种边界情况
5. 并行计算：对于超大型网格，可以考虑行或列的并行计算

常见问题排查：

1. 障碍物处理错误：确保在遇到障碍物时路径数正确设为 0
2. 边界初始化错误：注意第一行和第一列的障碍物对后续格子的影响
3. 数组索引越界：注意 Python 的索引从 0 开始
4. 整数溢出：Python 的整数不会溢出，但需要注意数值可能变得非常大
5. 内存限制：对于超大网格，需要使用空间优化版本

```
"""  
  
from typing import List  
  
class Solution:  
    def uniquePathsWithObstacles(self, obstacleGrid: List[List[int]]) -> int:  
        """  
        计算从左上角到右下角的不同路径数  
        :param obstacleGrid: m x n 的网格，1 表示障碍物，0 表示空位置  
        :return: 不同路径的数量  
        """  
  
        m = len(obstacleGrid)  
        n = len(obstacleGrid[0])  
  
        # 如果起点或终点有障碍物，直接返回 0  
        if obstacleGrid[0][0] == 1 or obstacleGrid[m-1][n-1] == 1:  
            return 0  
  
        # dp[i][j] 表示从起点到位置(i, j)的不同路径数  
        dp = [[0] * n for _ in range(m)]  
  
        # 初始化起点  
        dp[0][0] = 1  
  
        # 初始化第一行：只能从左边来，如果遇到障碍物，则后续都不可达  
        for j in range(1, n):  
            dp[0][j] = 0 if obstacleGrid[0][j] == 1 else dp[0][j-1]  
  
        # 初始化第一列：只能从上面来，如果遇到障碍物，则后续都不可达
```

```

for i in range(1, m):
    dp[i][0] = 0 if obstacleGrid[i][0] == 1 else dp[i-1][0]

# 填充 dp 表
for i in range(1, m):
    for j in range(1, n):
        # 如果当前位置有障碍物，则路径数为 0
        if obstacleGrid[i][j] == 1:
            dp[i][j] = 0
        else:
            # 否则路径数等于从上面和左边来的路径数之和
            dp[i][j] = dp[i-1][j] + dp[i][j-1]

return dp[m-1][n-1]

# 测试代码
if __name__ == "__main__":
    solution = Solution()

# 测试用例 1
obstacleGrid1 = [
    [0, 0, 0],
    [0, 1, 0],
    [0, 0, 0]
]
print("测试用例 1 结果:", solution.uniquePathsWithObstacles(obstacleGrid1)) # 预期输出: 2

# 测试用例 2
obstacleGrid2 = [
    [0, 1],
    [0, 0]
]
print("测试用例 2 结果:", solution.uniquePathsWithObstacles(obstacleGrid2)) # 预期输出: 1

# 测试用例 3
obstacleGrid3 = [
    [0, 0],
    [1, 1],
    [0, 0]
]
print("测试用例 3 结果:", solution.uniquePathsWithObstacles(obstacleGrid3)) # 预期输出: 0

# 测试用例 4

```

```

obstacleGrid4 = [
    [0, 0, 0, 0],
    [0, 0, 0, 0],
    [0, 0, 0, 0]
]
print("测试用例 4 结果:", solution.uniquePathsWithObstacles(obstacleGrid4)) # 预期输出: 10

# 类似题目与训练拓展:
# 1. LeetCode 62. Unique Paths (不同路径)
#     链接: https://leetcode.cn/problems/unique-paths/
#     区别: 没有障碍物的网格路径问题
#     算法: 组合数学或二维动态规划,  $dp[i][j] = dp[i-1][j] + dp[i][j-1]$ 
#
# 2. LeetCode 64. Minimum Path Sum (最小路径和)
#     链接: https://leetcode.cn/problems/minimum-path-sum/
#     区别: 求从左上角到右下角的最小路径和, 每步只能向下或向右
#     算法: 二维动态规划,  $dp[i][j] = \min(dp[i-1][j], dp[i][j-1]) + grid[i][j]$ 
#
# 3. LeetCode 174. Dungeon Game (地下城游戏)
#     链接: https://leetcode.cn/problems/dungeon-game/
#     区别: 骑士需要从左上角到右下角, 保证健康点数始终大于 0 的最小初始值
#     算法: 从右下角向左上角动态规划,  $dp[i][j] = \max(1, \min(dp[i+1][j], dp[i][j+1])) - dungeon[i][j]$ 
#
# 4. LeetCode 63. Unique Paths II (本题)
#     链接: https://leetcode.cn/problems/unique-paths-ii/
#     算法: 二维动态规划, 处理障碍物的路径计数问题
#
# 5. LeetCode 688. Knight Probability in Chessboard
#     链接: https://leetcode.cn/problems/knight-probability-in-chessboard/
#     区别: 国际象棋骑士移动, 计算留在棋盘内的概率
#     算法: 动态规划,  $dp[k][r][c]$  表示  $k$  步后在位置  $(r, c)$  的概率
#
# 6. LeetCode 980. Unique Paths III
#     链接: https://leetcode.cn/problems/unique-paths-iii/
#     区别: 需要访问所有非障碍物格子的路径数
#     算法: 回溯+剪枝或位掩码动态规划
#
# 7. Codeforces 1296D - Fight with Monsters
#     链接: https://codeforces.com/problemset/problem/1296/D
#     区别: 贪心策略解决怪物战斗问题, 但状态转移思想类似
#     算法: 排序后贪心选择最优策略
#

```

8. AtCoder ABC159E - Dividing Chocolate
链接: https://atcoder.jp/contests/abc159/tasks/abc159_e
区别: 二维网格分割问题, 但需要类似的状态转移思路
算法: 前缀和+状态压缩动态规划

9. 洛谷 P1002 [NOIP2002 普及组] 过河卒
链接: <https://www.luogu.com.cn/problem/P1002>
区别: 类似不同路径, 但马的位置不能走
算法: 二维动态规划, 注意避开马的控制点

10. 牛客网 NC14552 方格取数
链接: <https://ac.nowcoder.com/acm/problem/14552>
区别: 两个人同时从左上角出发到右下角取数, 求最大和
算法: 三维动态规划, 状态定义为 $dp[k][x_1][x_2]$, 表示走了 k 步, 第一个人在 $(x_1, k-x_1)$, 第二个人在 $(x_2, k-x_2)$

11. UVa 10913 - Walking on a Grid
链接:
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=1854
区别: 在网格中行走, 有负数, 最多允许 k 次负数
算法: 四维动态规划, 状态定义为 $dp[i][j][k][d]$, 表示在 (i, j) 位置, 已经经过 k 次负数, 方向为 d

12. SPOJ - SBANK
链接: <https://www.spoj.com/problems/SBANK/>
区别: 银行账号排序问题, 使用哈希表优化
算法: 哈希表+排序

13. HackerEarth - Roy and Coin Boxes
链接: <https://www.hackerearth.com/practice/data-structures/arrays/1-d/practice-problems/algorithm/roy-and-coin-boxes-1/description/>
区别: 区间更新问题, 使用差分数组优化
算法: 差分数组+前缀和

14. 杭电 HDU 1024 - Max Sum Plus Plus
链接: <http://acm.hdu.edu.cn/showproblem.php?pid=1024>
区别: 最大 m 段子数组和问题
算法: 二维动态规划优化为一维

15. Codeforces 1295E - Permutation Separation
链接: <https://codeforces.com/problemset/problem/1295/E>
区别: 排列分割问题, 需要找到最优分割点
算法: 前缀和+动态规划

```
#  
# 算法本质与技巧总结:  
# 1. 二维动态规划的基本模型: 从左上到右下, 只能向右或向下移动  
# 2. 障碍物处理技巧: 遇到障碍物时路径数置为 0  
# 3. 边界条件处理: 第一行和第一列需要特别处理  
# 4. 空间优化方法: 使用滚动数组将空间复杂度从  $O(m \times n)$  优化到  $O(\min(m, n))$   
# 5. 早期检测优化: 快速判断起点或终点是否被阻塞  
  
#  
# Python 工程化实战建议:  
# 1. 测试用例设计:  
#     - 空网格或单行单列网格  
#     - 起点或终点有障碍物的情况  
#     - 障碍物阻断所有路径的情况  
#     - 网格中有大量障碍物的情况  
# 2. 性能优化:  
#     - 空间优化: 使用一维数组进行滚动更新  
#     - 计算优化: 对于非常大的网格, 可以使用组合数学方法结合障碍物检查  
#     - 内存优化: 使用生成器表达式代替列表推导式减少内存占用  
# 3. 代码健壮性:  
#     - 添加输入合法性检查, 如空网格或无效尺寸  
#     - 使用 try-except 块处理可能的异常  
#     - 添加断言验证关键条件  
# 4. 调试技巧:  
#     - 添加打印语句输出中间状态  
#     - 使用 assert 语句验证关键条件  
#     - 考虑使用 Python 调试器 pdb 进行断点调试  
# 5. Python 特性应用:  
#     - 使用类型注解提高代码可读性和 IDE 支持  
#     - 考虑使用 functools.lru_cache 实现记忆化搜索版本  
#     - 利用 Python 的列表推导式简化初始化代码  
# 6. 跨语言实现对比:  
#     - 在 C++ 中, 可以使用二维数组或 vector<vector<int>>  
#     - 在 Java 中, 可以使用二维 int 数组, 注意整数溢出问题  
#     - 不同语言在处理大整数时的差异需要注意
```

=====

文件: Code09_MinimumPathSum.cpp

=====

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

```
using namespace std;

/***
 * 最小路径和问题 - C++实现
 * 算法思路：动态规划
 * 时间复杂度：O(m * n)
 * 空间复杂度：O(m * n)，可优化到 O(n)
 *
 * 核心思想：
 * 1. 使用二维 DP 数组 dp[i][j] 表示从起点(0, 0)到位置(i, j)的最小路径和
 * 2. 状态转移：路径和等于从上面和左边来的较小路径和加上当前位置的值
 * 3. 边界条件：第一行只能从左边来，第一列只能从上面来
 */


```

```
class MinimumPathSum {
public:
    /**
     * 计算从左上角到右下角的最小路径和
     * @param grid m x n 的网格，包含非负整数
     * @return 最小路径和
     */
    int minPathSum(vector<vector<int>>& grid) {
        int m = grid.size();
        if (m == 0) return 0;
        int n = grid[0].size();

        // 输入验证
        if (n == 0) return 0;

        // dp[i][j] 表示从起点到位置(i, j)的最小路径和
        vector<vector<int>> dp(m, vector<int>(n, 0));

        // 初始化起点
        dp[0][0] = grid[0][0];

        // 初始化第一行
        for (int j = 1; j < n; j++) {
            dp[0][j] = dp[0][j-1] + grid[0][j];
        }

        // 初始化第一列
        for (int i = 1; i < m; i++) {
```

```

dp[i][0] = dp[i-1][0] + grid[i][0];
}

// 填充 dp 表
for (int i = 1; i < m; i++) {
    for (int j = 1; j < n; j++) {
        // 路径和等于从上面和左边来的较小路径和加上当前位置的值
        dp[i][j] = min(dp[i-1][j], dp[i][j-1]) + grid[i][j];
    }
}

return dp[m-1][n-1];
}

/***
 * 空间优化版本：使用滚动数组将空间复杂度优化到 O(n)
 * @param grid m x n 的网格
 * @return 最小路径和
 */
int minPathSumOptimized(vector<vector<int>>& grid) {
    int m = grid.size();
    if (m == 0) return 0;
    int n = grid[0].size();

    // 输入验证
    if (n == 0) return 0;

    // 使用一维数组优化空间
    vector<int> dp(n, 0);

    // 初始化第一行
    dp[0] = grid[0][0];
    for (int j = 1; j < n; j++) {
        dp[j] = dp[j-1] + grid[0][j];
    }

    // 填充 dp 表
    for (int i = 1; i < m; i++) {
        // 更新第一列
        dp[0] = dp[0] + grid[i][0];

        for (int j = 1; j < n; j++) {
            // 路径和等于从上面和左边来的较小路径和加上当前位置的值
            dp[j] = min(dp[j-1], dp[j]) + grid[i][j];
        }
    }
}

```

```

        dp[j] = min(dp[j], dp[j-1]) + grid[i][j];
    }

}

return dp[n-1];
}

/***
 * 单元测试函数
 */
void test() {
    // 测试用例 1
    vector<vector<int>> grid1 = {
        {1, 3, 1},
        {1, 5, 1},
        {4, 2, 1}
    };
    int result1 = minPathSum(grid1);
    cout << "Test 1 - Basic: " << result1 << " (Expected: 7)" << endl;

    // 测试用例 2
    vector<vector<int>> grid2 = {
        {1, 2, 3},
        {4, 5, 6}
    };
    int result2 = minPathSum(grid2);
    cout << "Test 2 - Small Grid: " << result2 << " (Expected: 12)" << endl;

    // 测试用例 3
    vector<vector<int>> grid3 = {
        {1, 2},
        {1, 1}
    };
    int result3 = minPathSum(grid3);
    cout << "Test 3 - 2x2 Grid: " << result3 << " (Expected: 3)" << endl;

    // 测试用例 4
    vector<vector<int>> grid4 = {
        {1, 3, 1, 2},
        {1, 5, 1, 3},
        {4, 2, 1, 1}
    };
    int result4 = minPathSum(grid4);
}

```

```

cout << "Test 4 - Larger Grid: " << result4 << " (Expected: 8)" << endl;

// 测试优化版本
int result10pt = minPathSumOptimized(grid1);
cout << "Test 1 - Optimized: " << result10pt << " (Expected: 7)" << endl;

// 性能测试：大规模数据
vector<vector<int>> largeGrid(100, vector<int>(100, 1));

cout << "Performance test started..." << endl;
int largeResult = minPathSumOptimized(largeGrid);
cout << "Performance test completed. Result: " << largeResult << endl;
}

};

int main() {
    MinimumPathSum solution;

    // 运行单元测试
    solution.test();

    return 0;
}

/*
 * 相关题目扩展：
 *
 * 1. LeetCode 62 - Unique Paths (不同路径)
 *   链接: https://leetcode.cn/problems/unique-paths/
 *   区别: 计算不同路径的数量而不是路径和
 *   算法: 组合数学或动态规划
 *
 * 2. LeetCode 63 - Unique Paths II (不同路径 II)
 *   链接: https://leetcode.cn/problems/unique-paths-ii/
 *   区别: 网格中有障碍物，计算不同路径的数量
 *   算法: 二维动态规划
 *
 * 3. LeetCode 174 - Dungeon Game (地下城游戏)
 *   链接: https://leetcode.cn/problems/dungeon-game/
 *   区别: 骑士需要从左上角到右下角，保证健康点数始终大于 0 的最小初始值
 *   算法: 从右下角向左上角动态规划
 *
 * 4. LeetCode 64 - Minimum Path Sum (本题)
 */

```

- * 链接: <https://leetcode.cn/problems/minimum-path-sum/>
- * 算法: 二维动态规划
- *
- * 5. LeetCode 120 - Triangle (三角形最小路径和)
 - * 链接: <https://leetcode.cn/problems/triangle/>
 - * 区别: 三角形网格的最小路径和
 - * 算法: 动态规划, 从底向上
- *
- * 6. LeetCode 931 - Minimum Falling Path Sum (下降路径最小和)
 - * 链接: <https://leetcode.cn/problems/minimum-falling-path-sum/>
 - * 区别: 从第一行任意位置开始, 到最后一行的最小路径和
 - * 算法: 动态规划
- *
- * 7. LeetCode 1289 - Minimum Falling Path Sum II (下降路径最小和 II)
 - * 链接: <https://leetcode.cn/problems/minimum-falling-path-sum-ii/>
 - * 区别: 不能选择同一列相邻行的元素
 - * 算法: 动态规划+优化
- *
- * 8. LeetCode 1301 - Number of Paths with Max Score (最大得分路径数)
 - * 链接: <https://leetcode.cn/problems/number-of-paths-with-max-score/>
 - * 区别: 计算最大得分和对应的路径数
 - * 算法: 动态规划
- *
- * 9. LeetCode 1575 - Count All Possible Routes (统计所有可能的路线)
 - * 链接: <https://leetcode.cn/problems/count-all-possible-routes/>
 - * 区别: 在图中统计从起点到终点的所有可能路线
 - * 算法: 动态规划+记忆化搜索
- *
- * 10. LeetCode 980 - Unique Paths III (不同路径 III)
 - * 链接: <https://leetcode.cn/problems/unique-paths-iii/>
 - * 区别: 需要访问所有空单元格一次且仅一次
 - * 算法: 回溯+状态压缩
- *
- * 算法技巧总结:
 - * 1. 动态规划是解决网格路径问题的核心方法
 - * 2. 状态转移方程: $dp[i][j] = \min(dp[i-1][j], dp[i][j-1]) + grid[i][j]$
 - * 3. 边界条件处理: 第一行和第一列的特殊情况
 - * 4. 空间优化: 使用滚动数组将空间复杂度从 $O(m*n)$ 优化到 $O(n)$
 - * 5. 输入验证: 检查网格是否为空
- *
- * C++工程化考量:
 - * 1. 内存管理: 使用 vector 避免手动内存管理
 - * 2. 性能优化: 使用引用避免不必要的拷贝

- * 3. 异常安全：使用 RAI^I 原则管理资源
- * 4. 代码可读性：使用有意义的变量名和注释
- * 5. 测试覆盖：包含边界测试、性能测试
- *
- * 调试技巧：
- * 1. 使用 cout 输出中间变量值
- * 2. 使用 gdb 调试器进行调试
- * 3. 编写单元测试验证算法正确性
- * 4. 使用性能分析工具优化代码效率

*/

=====

文件：Code09_MinimumPathSum.java

=====

```
package class127;

// Minimum Path Sum
// 给定一个包含非负整数的 m x n 网格 grid，请找出一条从左上角到右下角的路径，  
// 使得路径上的数字总和为最小。每次只能向下或者向右移动一步。  
// 测试链接：https://leetcode.cn/problems/minimum-path-sum/

/**  
 * 算法思路：  
 * 1. 这是一个典型的动态规划问题  
 * 2. 使用二维动态规划 dp[i][j]，表示从起点(0, 0)到位置(i, j)的最小路径和  
 * 3. 状态转移方程：  
 *   - dp[0][0] = grid[0][0] (起点)  
 *   - 如果 i == 0 且 j > 0，只能从左边来，dp[i][j] = dp[i][j-1] + grid[i][j]  
 *   - 如果 j == 0 且 i > 0，只能从上面来，dp[i][j] = dp[i-1][j] + grid[i][j]  
 *   - 其他情况，可以从上面或左边来，dp[i][j] = min(dp[i-1][j], dp[i][j-1]) + grid[i][j]  
 * 4. 初始化：  
 *   - dp[0][0] = grid[0][0]  
 * 5. 结果：  
 *   - dp[m-1][n-1] 即为所求的最小路径和  
 *  
 * 时间复杂度：O(m * n)  
 * 空间复杂度：O(m * n)  
 */  
  
public class Code09_MinimumPathSum {  
  
    /**  
     * 计算从左上角到右下角的最小路径和  
     */
```

```
* @param grid m x n 的网格，包含非负整数
* @return 最小路径和
*/
public static int minPathSum(int[][] grid) {
    int m = grid.length;
    int n = grid[0].length;

    // dp[i][j] 表示从起点到位置(i, j)的最小路径和
    int[][] dp = new int[m][n];

    // 初始化起点
    dp[0][0] = grid[0][0];

    // 初始化第一行
    for (int j = 1; j < n; j++) {
        dp[0][j] = dp[0][j-1] + grid[0][j];
    }

    // 初始化第一列
    for (int i = 1; i < m; i++) {
        dp[i][0] = dp[i-1][0] + grid[i][0];
    }

    // 填充 dp 表
    for (int i = 1; i < m; i++) {
        for (int j = 1; j < n; j++) {
            // 路径和等于从上面和左边来的较小路径和加上当前位置的值
            dp[i][j] = Math.min(dp[i-1][j], dp[i][j-1]) + grid[i][j];
        }
    }

    return dp[m-1][n-1];
}

// 测试方法
public static void main(String[] args) {
    // 测试用例 1
    int[][] grid1 = {
        {1, 3, 1},
        {1, 5, 1},
        {4, 2, 1}
    };
    System.out.println("测试用例 1 结果: " + minPathSum(grid1)); // 预期输出: 7
}
```

```

// 测试用例 2
int[][] grid2 = {
    {1, 2, 3},
    {4, 5, 6}
};
System.out.println("测试用例 2 结果: " + minPathSum(grid2)); // 预期输出: 12

// 测试用例 3
int[][] grid3 = {
    {1, 2},
    {1, 1}
};
System.out.println("测试用例 3 结果: " + minPathSum(grid3)); // 预期输出: 3

// 测试用例 4
int[][] grid4 = {
    {1, 3, 1, 2},
    {1, 5, 1, 3},
    {4, 2, 1, 1}
};
System.out.println("测试用例 4 结果: " + minPathSum(grid4)); // 预期输出: 8
}

// 相关题目:
// 1. LeetCode 62. Unique Paths (不同路径)
//     链接: https://leetcode.cn/problems/unique-paths/
//     区别: 计算不同路径的数量而不是路径和
// 2. LeetCode 63. Unique Paths II (不同路径 II)
//     链接: https://leetcode.cn/problems/unique-paths-ii/
//     区别: 网格中有障碍物, 计算不同路径的数量
// 3. LeetCode 174. Dungeon Game (地下城游戏)
//     链接: https://leetcode.cn/problems/dungeon-game/
//     区别: 骑士需要从左上角到右下角, 保证健康点数始终大于 0 的最小初始值
}
=====

文件: Code09_MinimumPathSum.py
=====

# Minimum Path Sum
# 给定一个包含非负整数的 m x n 网格 grid, 请找出一条从左上角到右下角的路径,
# 使得路径上的数字总和为最小。每次只能向下或者向右移动一步。

```

文件: Code09_MinimumPathSum.py

```

# Minimum Path Sum
# 给定一个包含非负整数的 m x n 网格 grid, 请找出一条从左上角到右下角的路径,
# 使得路径上的数字总和为最小。每次只能向下或者向右移动一步。

```

```
# 测试链接 : https://leetcode.cn/problems/minimum-path-sum/
```

"""

算法思路深度解析:

1. 这是一个典型的二维动态规划问题，属于路径优化类问题的基础模型
 - 问题的核心是在约束移动方向（只能向右或向下）的情况下，寻找最优路径
 - 动态规划的关键在于利用子问题的最优解构建原问题的最优解
2. 使用二维动态规划 $dp[i][j]$ ，其中：
 - $dp[i][j]$ 表示从起点 $(0, 0)$ 到位置 (i, j) 的最小路径和
 - 状态定义清晰地捕捉了问题的最优子结构特性
3. 状态转移方程的详细分析：
 - 起点处理: $dp[0][0] = grid[0][0]$ (基础情况)
 - 边界处理：
 - * 第一行 ($i == 0$): 只能从左边来, $dp[i][j] = dp[i][j-1] + grid[i][j]$
 - * 第一列 ($j == 0$): 只能从上面来, $dp[i][j] = dp[i-1][j] + grid[i][j]$
 - 一般情况: 选择从上面或左边来的最小路径和, $dp[i][j] = \min(dp[i-1][j], dp[i][j-1]) + grid[i][j]$
4. 初始化策略：
 - 逐步初始化: 先处理起点, 再分别处理第一行和第一列, 最后处理内部格子
 - 这种初始化方式确保了在计算每个状态时, 其依赖的状态已经被计算完成
5. 结果提取：
 - $dp[m-1][n-1]$ 即为所求的从起点到终点的最小路径和

时间复杂度分析:

- 状态总数: $O(m * n)$
- 每个状态的计算是 $O(1)$ 操作
- 总时间复杂度: $O(m * n)$

空间复杂度分析:

- 原始实现使用二维数组: $O(m * n)$
- 可以通过滚动数组优化到 $O(\min(m, n))$
- 更进一步, 可以直接在原数组上进行修改, 空间复杂度降为 $O(1)$ (如果允许修改输入)

Python 实现注意事项:

1. 初始化优化: 使用列表推导式创建二维数组, 简洁高效
2. 边界条件处理: 正确处理第一行和第一列的初始化
3. 内存优化: 可以使用一维数组进行滚动更新, 减少内存消耗
4. 输入验证: 确保输入网格非空且为有效尺寸
5. 数值溢出: 虽然题目说明是非负整数, 但 Python 的整数不会溢出

工程化考量:

1. 异常处理: 处理空网格或无效输入的情况
2. 性能优化: 对于大型网格, 考虑使用滚动数组优化空间

3. 代码可读性：使用清晰的变量名和适当的注释
4. 边界测试：确保算法能正确处理各种边界情况
5. 算法扩展性：如何修改以支持更多的移动方向或约束条件

常见问题排查：

1. 索引越界：注意 Python 的索引从 0 开始
 2. 初始化错误：确保第一行和第一列的初始化正确
 3. 状态转移错误：确保选择了正确的前驱状态
 4. 内存限制：对于超大网格，需要使用空间优化版本
 5. 输入验证：确保输入网格符合要求
- """

```
from typing import List

class Solution:
    def minPathSum(self, grid: List[List[int]]) -> int:
        """
        计算从左上角到右下角的最小路径和
        :param grid: m x n 的网格，包含非负整数
        :return: 最小路径和
        """
        m = len(grid)
        n = len(grid[0])

        # dp[i][j] 表示从起点到位置(i, j)的最小路径和
        dp = [[0] * n for _ in range(m)]

        # 初始化起点
        dp[0][0] = grid[0][0]

        # 初始化第一行：只能从左边来，累加左边路径的和
        for j in range(1, n):
            dp[0][j] = dp[0][j-1] + grid[0][j]

        # 初始化第一列：只能从上面来，累加上面路径的和
        for i in range(1, m):
            dp[i][0] = dp[i-1][0] + grid[i][0]

        # 填充 dp 表
        for i in range(1, m):
            for j in range(1, n):
                # 路径和等于从上面和左边来的较小路径和加上当前位置的值
                dp[i][j] = min(dp[i-1][j], dp[i][j-1]) + grid[i][j]
```

```

        return dp[m-1][n-1]

# 测试代码
if __name__ == "__main__":
    solution = Solution()

# 测试用例 1
grid1 = [
    [1, 3, 1],
    [1, 5, 1],
    [4, 2, 1]
]
print("测试用例 1 结果:", solution.minPathSum(grid1)) # 预期输出: 7

# 测试用例 2
grid2 = [
    [1, 2, 3],
    [4, 5, 6]
]
print("测试用例 2 结果:", solution.minPathSum(grid2)) # 预期输出: 12

# 测试用例 3
grid3 = [
    [1, 2],
    [1, 1]
]
print("测试用例 3 结果:", solution.minPathSum(grid3)) # 预期输出: 3

# 测试用例 4
grid4 = [
    [1, 3, 1, 2],
    [1, 5, 1, 3],
    [4, 2, 1, 1]
]
print("测试用例 4 结果:", solution.minPathSum(grid4)) # 预期输出: 8

# 类似题目与训练拓展:
# 1. LeetCode 62. Unique Paths (不同路径)
#     链接: https://leetcode.cn/problems/unique-paths/
#     区别: 计算从左上角到右下角的不同路径数量
#     算法: 组合数学或二维动态规划,  $dp[i][j] = dp[i-1][j] + dp[i][j-1]$ 
#

```

```
# 2. LeetCode 63. Unique Paths II (不同路径 II)
#     链接: https://leetcode.cn/problems/unique-paths-ii/
#     区别: 网格中有障碍物, 计算不同路径的数量
#     算法: 二维动态规划, 遇到障碍物时  $dp[i][j] = 0$ 
#
# 3. LeetCode 174. Dungeon Game (地下城游戏)
#     链接: https://leetcode.cn/problems/dungeon-game/
#     区别: 骑士需要从左上角到右下角, 保证健康点数始终大于 0 的最小初始值
#     算法: 从右下角向左上角动态规划,  $dp[i][j] = \max(1, \min(dp[i+1][j], dp[i][j+1]) - dungeon[i][j])$ 
#
# 4. LeetCode 64. Minimum Path Sum (本题)
#     链接: https://leetcode.cn/problems/minimum-path-sum/
#     算法: 二维动态规划, 计算最小路径和
#
# 5. LeetCode 741. Cherry Pickup (摘樱桃 I)
#     链接: https://leetcode.cn/problems/cherry-pickup/
#     区别: 一个人从  $(0, 0)$  走到  $(n-1, n-1)$  再走回  $(0, 0)$ , 求最大收集樱桃数
#     算法: 三维动态规划, 转化为两个人同时从起点到终点的问题
#
# 6. LeetCode 1463. Cherry Pickup II (摘樱桃 II)
#     链接: https://leetcode.cn/problems/cherry-pickup-ii/
#     区别: 两个机器人同时从顶部向底部移动, 收集樱桃
#     算法: 三维动态规划, 状态定义为  $dp[i][j1][j2]$ 
#
# 7. LeetCode 688. Knight Probability in Chessboard
#     链接: https://leetcode.cn/problems/knight-probability-in-chessboard/
#     区别: 国际象棋骑士移动, 计算留在棋盘内的概率
#     算法: 动态规划,  $dp[k][r][c]$  表示  $k$  步后在位置  $(r, c)$  的概率
#
# 8. LeetCode 980. Unique Paths III
#     链接: https://leetcode.cn/problems/unique-paths-iii/
#     区别: 需要访问所有非障碍物格子的路径数
#     算法: 回溯+剪枝或位掩码动态规划
#
# 9. 洛谷 P1002 [NOIP2002 普及组] 过河卒
#     链接: https://www.luogu.com.cn/problem/P1002
#     区别: 类似不同路径, 但马的位置不能走
#     算法: 二维动态规划, 注意避开马的控制点
#
# 10. 牛客网 NC14552 方格取数
#     链接: https://ac.nowcoder.com/acm/problem/14552
#     区别: 两个人同时从左上角出发到右下角取数, 求最大和
```

```
# 算法: 三维动态规划, 状态定义为 dp[k][x1][x2], 表示走了 k 步, 第一个人在(x1, k-x1), 第二个人在(x2, k-x2)
#
# 11. UVa 10913 - Walking on a Grid
# 链接:
https://onlinejudge.org/index.php?option=com\_onlinejudge&Itemid=8&category=24&page=show\_problem&problem=1854
# 区别: 在网格中行走, 有负数, 最多允许 k 次负数
# 算法: 四维动态规划, 状态定义为 dp[i][j][k][d], 表示在(i, j)位置, 已经经过 k 次负数, 方向为 d
#
# 12. SPOJ - SBANK
# 链接: https://www.spoj.com/problems/SBANK/
# 区别: 银行账号排序问题, 使用哈希表优化
# 算法: 哈希表+排序
#
# 13. HackerEarth - Roy and Coin Boxes
# 链接: https://www.hackerearth.com/practice/data-structures/arrays/1-d/practice-problems/algorithm/roy-and-coin-boxes-1/
# 区别: 区间更新问题, 使用差分数组优化
# 算法: 差分数组+前缀和
#
# 14. 杭电 HDU 1024 - Max Sum Plus Plus
# 链接: http://acm.hdu.edu.cn/showproblem.php?pid=1024
# 区别: 最大 m 段子数组和问题
# 算法: 二维动态规划优化为一维
#
# 15. Codeforces 1295E - Permutation Separation
# 链接: https://codeforces.com/problemset/problem/1295/E
# 区别: 排列分割问题, 需要找到最优分割点
# 算法: 前缀和+动态规划
#
# 算法本质与技巧总结:
# 1. 二维动态规划的基本模型: 从左上到右下, 只能向右或向下移动
# 2. 最优子结构: 当前位置的最优解取决于左上两个位置的最优解
# 3. 空间优化方法: 使用滚动数组将空间复杂度从 O(m*n) 优化到 O(min(m, n))
# 4. 原地更新: 在允许修改输入的情况下, 可以直接在原数组上更新以节省空间
# 5. 贪心策略: 在每一步选择当前最优的移动方向 (向左或向上)
#
# Python 工程化实战建议:
# 1. 测试用例设计:
#   - 空网格或单行单列网格
#   - 网格中所有值相同的情况
#   - 网格中有特别大的值, 需要测试路径选择
```

```
#      - 大型网格测试性能
# 2. 性能优化:
#      - 空间优化: 使用一维数组进行滚动更新
#      - 原地更新: 直接在输入数组上修改以节省空间
#      - 计算优化: 避免重复计算
# 3. 代码健壮性:
#      - 添加输入合法性检查, 如空网格或无效尺寸
#      - 使用 try-except 块处理可能的异常
#      - 添加断言验证关键条件
# 4. 调试技巧:
#      - 添加打印语句输出中间状态
#      - 使用 assert 语句验证关键条件
#      - 考虑使用 Python 调试器 pdb 进行断点调试
# 5. Python 特性应用:
#      - 使用类型注解提高代码可读性和 IDE 支持
#      - 考虑使用 functools.lru_cache 实现记忆化搜索版本
#      - 利用 Python 的生成器表达式和列表推导式简化代码
# 6. 跨语言实现对比:
#      - 在 C++ 中, 可以使用二维数组或 vector<vector<int>>
#      - 在 Java 中, 可以使用二维 int 数组
#      - 不同语言在处理大整数时的差异需要注意
```

=====

文件: DPFusion.cpp

=====

```
#include <iostream>
#include <vector>
#include <string>
#include <queue>
#include <map>
#include <set>
#include <climits>
#include <cmath>
#include <algorithm>
#include <stack>
#include <unordered_map>
#include <unordered_set>
#include <functional>
#include <tuple>
#include <cstring>
#include <float.h>
```

```

using namespace std;

// ===== 优化体系: Knuth 优化 =====
/***
 * Knuth 优化的 DP 算法
 *
 * 问题描述:
 * 解决区间 DP 问题, 其中状态转移方程满足四边形不等式
 *
 * 解题思路:
 * 1. 使用 Knuth 优化将时间复杂度从 O(n^3) 降低到 O(n^2)
 * 2. 维护最优转移点数组 opt[i][j], 表示计算 dp[i][j] 时的最优 k 值
 * 3. 根据 opt[i][j-1] ≤ opt[i][j] ≤ opt[i+1][j] 的性质进行剪枝
 *
 * 参数:
 * n: 区间长度
 * costFunc: 计算区间(i, j)代价的函数
 *
 * 返回:
 * pair<vector<vector<long long>>, vector<vector<int>>>; 包含 dp 数组和 opt 数组的结果
 *
 * 时间复杂度: O(n^2)
 * 空间复杂度: O(n^2)
 */
pair<vector<vector<long long>>, vector<vector<int>>> knuthOptimization(int n, const function<long long(int, int)>& costFunc) {
    // 初始化 dp 和 opt 数组
    const long long INF = LLONG_MAX;
    vector<vector<long long>> dp(n + 1, vector<long long>(n + 1, INF));
    vector<vector<int>> opt(n + 1, vector<int>(n + 1, 0));

    // 初始化长度为 1 的区间
    for (int i = 1; i <= n; ++i) {
        dp[i][i] = 0;
        opt[i][i] = i;
    }

    // 枚举区间长度
    for (int length = 2; length <= n; ++length) {
        // 枚举起始点
        for (int i = 1; i + length - 1 <= n; ++i) {
            int j = i + length - 1;
            // 初始化为无穷大

```

```

dp[i][j] = INF;
// 根据 Knuth 优化的性质，最优 k 在 opt[i][j-1] 到 opt[i+1][j] 之间
int upperK = (i + 1 <= j) ? opt[i + 1][j] : j - 1;

for (int k = opt[i][j-1]; k <= min(upperK, j-1); ++k) {
    if (dp[i][k] != INF && dp[k+1][j] != INF) {
        long long cost = costFunc(i, j);
        if (cost != INF) {
            long long current = dp[i][k] + dp[k+1][j] + cost;
            if (current < dp[i][j]) {
                dp[i][j] = current;
                opt[i][j] = k;
            }
        }
    }
}

return {dp, opt};
}

// ===== 优化体系: Divide & Conquer Optimization =====
/***
 * 计算 dp[i][l..r]，其中最优转移点在 opt_l..opt_r 之间
 */
void solveDivideConquer(int i, int l, int r, int opt_l, int opt_r, vector<vector<long long>>& dp,
                        const function<long long(int, int)>& costFunc) {
    if (l > r) return;

    int mid = (l + r) / 2;
    int best_k = opt_l;
    const long long INF = LLONG_MAX;
    dp[i][mid] = INF;

    // 在 opt_l 到 min(mid, opt_r) 之间寻找最优 k
    for (int k = opt_l; k <= min(mid, opt_r); ++k) {
        if (dp[i-1][k] != INF) {
            long long cost = costFunc(k, mid);
            if (cost != INF) {
                long long current = dp[i-1][k] + cost;
                if (current < dp[i][mid]) {
                    dp[i][mid] = current;
                }
            }
        }
    }
}

```

```

        best_k = k;
    }
}
}

// 递归处理左右子区间
solveDivideConquer(i, 1, mid - 1, opt_l, best_k, dp, costFunc);
solveDivideConquer(i, mid + 1, r, best_k, opt_r, dp, costFunc);
}

/***
 * Divide & Conquer Optimization (分治优化)
 *
 * 问题描述:
 * 解决形如  $dp[i][j] = \min\{dp[i-1][k] + cost(k, j)\}$ , 其中  $k < j$ 
 * 当转移满足决策单调性时使用
 *
 * 解题思路:
 * 1. 利用决策单调性, 使用分治法优化 DP
 * 2. 对于  $dp[i][j]$ , 当 i 固定时, 最优转移点 k 随着 j 的增加而单调不减
 * 3. 使用分治的方式计算每个区间的最优决策
 *
 * 参数:
 * n: 维度 1
 * m: 维度 2
 * costFunc: 计算 cost(k, j) 的函数
 *
 * 返回:
 * vector<vector<long long>>: DP 数组
 *
 * 时间复杂度: O(n*m log m)
 * 空间复杂度: O(n*m)
 */
vector<vector<long long>> divideConquerOptimization(int n, int m, const function<long long(int, int)>& costFunc) {
    // 初始化 dp 数组
    const long long INF = LLONG_MAX;
    vector<vector<long long>> dp(n + 1, vector<long long>(m + 1, INF));
    dp[0][0] = 0;

    // 对每个 i 应用分治优化
    for (int i = 1; i <= n; ++i) {

```

```

        solveDivideConquer(i, 1, m, 0, m, dp, costFunc);
    }

    return dp;
}

// ===== 优化体系: SMAWK 算法 (行最小查询) =====
/***
 * 行压缩: 只保留可能成为最小值的行
 */
vector<int> reduceRows(const vector<int>& rows, const vector<vector<double>>& matrix) {
    vector<int> stack;
    for (int i : rows) {
        while (stack.size() >= 2) {
            int j1 = stack.back();
            stack.pop_back();
            int j2 = stack.back();
            stack.push_back(j1); // 恢复栈状态

            // 比较两个行在列 stack.size()-2 处的值 (因为索引从 0 开始)
            if (matrix[j2][stack.size()-2] <= matrix[i][stack.size()-2]) {
                break;
            } else {
                stack.pop_back();
            }
        }
        stack.push_back(i);
    }
    return stack;
}

/***
 * 递归实现 SMAWK 算法
 */
vector<int> smawkRec(const vector<int>& rows, const vector<int>& cols, const
vector<vector<double>>& matrix) {
    if (rows.empty()) return {};
    // 行压缩
    vector<int> reducedRows = reduceRows(rows, matrix);

    // 递归求解列数为奇数的子问题
    vector<int> halfCols;

```

```

for (size_t i = 1; i < cols.size(); i += 2) {
    halfCols.push_back(cols[i]);
}

vector<int> minCols(reducedRows.size(), -1);

if (!halfCols.empty()) {
    // 递归求解
    vector<int> result = smawkRec(reducedRows, halfCols, matrix);
    // 复制结果
    for (size_t i = 0; i < result.size(); ++i) {
        minCols[i] = result[i];
    }
}

// 扩展结果到所有列
vector<int> result(rows.size());
int k = 0; // minCols 的索引

for (size_t i = 0; i < rows.size(); ++i) {
    int row = rows[i];
    // 确定当前行的最小值可能在哪个区间
    int start = 0;
    if (i > 0 && k > 0 && minCols[k-1] != -1) {
        start = minCols[k-1];
    }
    int end = (k < (int)minCols.size() && minCols[k] != -1) ? minCols[k] : cols.back();

    // 在这个区间内查找最小值
    double minValue = DBL_MAX;
    int minCol = start;

    // 找到 start 和 end 在 cols 中的索引
    size_t startIdx = 0, endIdx = cols.size() - 1;
    for (size_t idx = 0; idx < cols.size(); ++idx) {
        if (cols[idx] == start) startIdx = idx;
        if (cols[idx] == end) endIdx = idx;
    }

    for (size_t idx = startIdx; idx <= endIdx; ++idx) {
        int col = cols[idx];
        if (col < (int)matrix[0].size() && matrix[row][col] < minValue) {
            minValue = matrix[row][col];
        }
    }
}

```

```

        minCol = col;
    }

}

result[i] = minCol;

// 如果当前行在 reducedRows 中，且不是最后一行，k 前进
if (k < (int)reducedRows.size() && row == reducedRows[k]) {
    k++;
}
}

return result;
}

/***
 * SMAWK 算法用于在 Monge 矩阵中快速查找每行的最小值
 *
 * 问题描述:
 * 给定一个 Monge 矩阵，快速找到每行的最小值位置
 *
 * 解题思路:
 * 1. Monge 矩阵满足性质: matrix[i][j] + matrix[i+1][j+1] ≤ matrix[i][j+1] + matrix[i+1][j]
 * 2. SMAWK 算法利用这一性质，可以在 O(m+n) 时间内找到每行的最小值
 * 3. 主要步骤包括行压缩和递归求解
 *
 * 参数:
 *      matrix: 一个 Monge 矩阵
 *
 * 返回:
 *      每行最小值的列索引列表
 *
 * 时间复杂度: O(m+n)，其中 m 是行数，n 是列数
 * 空间复杂度: O(m+n)
 */
vector<int> smawk(const vector<vector<double>>& matrix) {
    int m = matrix.size();
    if (m == 0) return {};
    int n = matrix[0].size();

    // 构造行索引和列索引数组
    vector<int> rows(m), cols(n);
    for (int i = 0; i < m; ++i) rows[i] = i;

```

```

for (int j = 0; j < n; ++j) cols[j] = j;

// 调用递归实现
return smawkRec(rows, cols, matrix);
}

// ===== 优化体系: Aliens Trick (二分约束参数+可行性 DP) =====
/***
 * Aliens Trick (二分约束参数+可行性 DP)
 *
 * 问题描述:
 * 解决带约束的优化问题，通常形如最小化总成本，同时满足某些约束条件
 *
 * 解题思路:
 * 1. 将约束条件转化为参数  $\lambda$ ，构造拉格朗日函数
 * 2. 对  $\lambda$  进行二分查找，使用可行性 DP 判断当前  $\lambda$  下是否满足约束
 * 3. 根据可行性 DP 的结果调整二分区间
 *
 * 参数:
 * costFunc: 计算带参数  $\lambda$  的成本函数，返回 [value, constraint] 数组
 * checkFunc: 检查当前解是否满足约束的函数
 * left: 二分左边界
 * right: 二分右边界
 * eps: 精度要求
 *
 * 返回:
 * pair<double, double>: 最优参数 lambda 和对应最优解
 *
 * 时间复杂度:  $O(\log((right-left)/eps) * T(DP))$ ，其中  $T(DP)$  是一次 DP 的时间复杂度
 */
pair<double, double> aliensTrick(
    const function<pair<double, double>(double)>& costFunc,
    const function<bool(double)>& checkFunc,
    double left, double right, double eps = 1e-7) {
    double bestLambda = left;
    double bestValue = 0.0;

    while (right - left > eps) {
        double mid = (left + right) / 2;
        // 计算当前参数下的解和约束值
        auto [currentValue, constraintValue] = costFunc(mid);

        if (checkFunc(constraintValue)) {

```

```

        // 满足约束，尝试更小的参数
        right = mid;
        bestLambda = mid;
        bestValue = currentValue;
    } else {
        // 不满足约束，需要增大参数
        left = mid;
    }
}

return {bestLambda, bestValue};
}

// ===== 图上 DP→最短路：分层图建模 =====
/***
 * 分层图 Dijkstra 算法
 *
 * 问题描述：
 * 给定一个图，允许最多使用 k 次特殊操作（如跳跃、免费通行等），求最短路径
 *
 * 解题思路：
 * 1. 构建分层图，每层代表使用不同次数的特殊操作
 * 2. 对于每个节点 u，在第 i 层表示到达 u 时已经使用了 i 次特殊操作
 * 3. 使用 Dijkstra 算法在分层图上寻找最短路径
 *
 * 参数：
 * n: 节点数量
 * m: 边的数量
 * edges: 边的列表，每个元素为[u, v, w]表示 u 到 v 的权为 w 的边
 * k: 允许使用的特殊操作次数
 *
 * 返回：
 * long long: 从节点 0 到节点 n-1 的最短路径长度，-1 表示不可达
 *
 * 时间复杂度: O((n*k + m*k) log(n*k))
 * 空间复杂度: O(n*k + m*k)
 */
long long layeredGraphDijkstra(int n, int m, const vector<tuple<int, int, long long>>& edges, int k) {
    // 构建分层图的邻接表
    vector<vector<pair<int, long long>>> graph(n * (k + 1));

    // 添加普通边（不使用特殊操作）

```

```

for (const auto& [u, v, w] : edges) {
    for (int i = 0; i <= k; ++i) {
        int from_node = u + i * n;
        graph[from_node].emplace_back(v + i * n, w);
    }
}

// 添加使用特殊操作的边（如果允许的话）
for (const auto& [u, v, w] : edges) {
    for (int i = 0; i < k; ++i) {
        // 这里假设特殊操作可以免费通行（权为 0），具体根据问题调整
        int from_node = u + i * n;
        graph[from_node].emplace_back(v + (i + 1) * n, 0);
    }
}

// Dijkstra 算法
const long long INF = LLONG_MAX;
vector<long long> dist(n * (k + 1), INF);
dist[0] = 0; // 假设起点是节点 0

// 使用优先队列，按距离排序
priority_queue<pair<long long, int>, vector<pair<long long, int>>, greater<>> heap;
heap.emplace(0, 0);

while (!heap.empty()) {
    auto [d, u] = heap.top();
    heap.pop();

    if (d > dist[u]) continue;

    for (const auto& [v, w] : graph[u]) {
        if (dist[v] > d + w) {
            dist[v] = d + w;
            heap.emplace(dist[v], v);
        }
    }
}

// 取所有层中到达终点的最小值
long long result = INF;
for (int i = 0; i <= k; ++i) {
    if (dist[n - 1 + i * n] < result) {

```

```

        result = dist[n - 1 + i * n];
    }

}

return result != INF ? result : -1;
}

// ====== 冷门模型: 期望 DP 遇环的方程组解 (高斯消元) ======
/**/

* 高斯消元法求解线性方程组
*
* 问题描述:
* 求解形如 Ax = b 的线性方程组
*
* 解题思路:
* 1. 构建增广矩阵
* 2. 进行高斯消元, 将矩阵转化为行阶梯形
* 3. 回代求解
*
* 参数:
*     matrix: 增广矩阵, 每行最后一个元素是 b 的值
*
* 返回:
*     vector<double>: 方程组的解数组
*
* 时间复杂度: O(n^3)
* 空间复杂度: O(n^2)
*/
vector<double> gaussianElimination(vector<vector<double>> matrix) {
    int n = matrix.size();
    const double eps = 1e-9;

    // 高斯消元过程
    for (int i = 0; i < n; ++i) {
        // 找到主元行 (当前列中绝对值最大的行)
        int maxRow = i;
        for (int j = i; j < n; ++j) {
            if (abs(matrix[j][i]) > abs(matrix[maxRow][i])) {
                maxRow = j;
            }
        }
        // 交换主元行和当前行
    }
}

```

```

swap(matrix[i], matrix[maxRow]);

// 如果主元为 0, 方程组可能有无穷多解或无解
if (abs(matrix[i][i]) < eps) continue;

// 消元过程
for (int j = i + 1; j < n; ++j) {
    double factor = matrix[j][i] / matrix[i][i];
    for (int k = i; k <= n; ++k) {
        matrix[j][k] -= factor * matrix[i][k];
    }
}

// 回代求解
vector<double> x(n);
for (int i = n - 1; i >= 0; --i) {
    x[i] = matrix[i][n];
    for (int j = i + 1; j < n; ++j) {
        x[i] -= matrix[i][j] * x[j];
    }
    x[i] /= matrix[i][i];
}

return x;
}

/**
 * 期望 DP 处理有环情况 (使用高斯消元)
 *
 * 问题描述:
 * 在有环的状态转移图中计算期望
 *
 * 解题思路:
 * 1. 对于每个状态, 建立期望方程
 * 2. 使用高斯消元求解方程组
 *
 * 参数:
 * n: 状态数量
 * transitions: 转移概率列表, transitions[i] 是一个列表, 每个元素为 (j, p) 表示从 i 转移到 j 的概率为 p
 *
 * 返回:

```

```

*      vector<double>: 每个状态的期望值数组
*
* 时间复杂度: O(n^3)
* 空间复杂度: O(n^2)
*/
vector<double> expectationDPWithCycles(int n, const vector<vector<pair<int, double>>>&
transitions) {
    // 构建线性方程组的增广矩阵
    vector<vector<double>> matrix(n, vector<double>(n + 1, 0.0));

    for (int i = 0; i < n; ++i) {
        matrix[i][i] = 1.0; // 方程左边: E[i] - sum(p_ij * E[j]) = cost[i]

        // 假设每个状态的代价为 1, 具体根据问题调整
        double cost = 1.0;
        matrix[i][n] = cost;

        for (const auto& [j, p] : transitions[i]) {
            if (i != j) { // 避免自环的特殊处理
                matrix[i][j] -= p;
            }
        }
    }

    // 使用高斯消元求解
    return gaussianElimination(matrix);
}

// ====== 冷门模型: 插头 DP (轮廓线 DP) ======
/***
* 插头 DP (轮廓线 DP) 示例: 求网格中哈密顿回路的数量
*
* 问题描述:
* 给定一个网格, 求其中哈密顿回路的数量
*
* 解题思路:
* 1. 使用轮廓线 DP, 记录当前处理到的位置和轮廓线状态
* 2. 插头表示连接的状态, 通常用二进制表示
* 3. 使用字典优化空间复杂度
*
* 参数:
* grid: 网格, 1 表示可通行, 0 表示障碍物
*

```

```

* 返回:
*     long long: 哈密顿回路的数量
*
* 时间复杂度: O(n*m*4^min(n, m))
* 空间复杂度: O(4^min(n, m))
*/
long long plugDP(const vector<vector<int>>& grid) {
    int n = grid.size();
    if (n == 0) return 0;
    int m = grid[0].size();

    // 使用 unordered_map 优化
    using StateMap = unordered_map<long long, long long>;
    StateMap dp;

    // 初始状态: 左上角没有插头
    dp[0] = 1;

    for (int i = 0; i < n; ++i) {
        // 新的一行开始, 需要将状态左移一位
        StateMap newDp;
        for (const auto& [state, cnt] : dp) {
            // 左移一位, 移除最左边的插头
            long long newState = state << 1;
            newDp[newState] += cnt;
        }
        dp = newDp;
    }

    for (int j = 0; j < m; ++j) {
        StateMap newDp2;

        for (const auto& [state, cnt] : dp) {
            // 当前位置左边和上边的插头状态
            int left = (state >> (2 * j)) & 3;
            int up = (state >> (2 * (j + 1))) & 3;

            // 如果当前位置是障碍物, 跳过
            if (grid[i][j] == 0) {
                // 只有当左右插头都不存在时才合法
                if (left == 0 && up == 0) {
                    newDp2[state] += cnt;
                }
            }
            continue;
        }
        dp = newDp2;
    }
}

```

```

}

// 处理各种插头组合情况
// 1. 没有左插头和上插头
if (left == 0 && up == 0) {
    // 只能创建新的插头对（用于回路的开始）
    if (i < n - 1 && j < m - 1 && grid[i+1][j] == 1 && grid[i][j+1] == 1) {
        long long newState = state | (1LL << (2 * j)) | (2LL << (2 * (j + 1)));
        newDp2[newState] += cnt;
    }
}

// 2. 只有左插头
else if (left != 0 && up == 0) {
    // 向下延伸
    if (i < n - 1 && grid[i+1][j] == 1) {
        newDp2[state] += cnt;
    }
    // 向右延伸
    if (j < m - 1 && grid[i][j+1] == 1) {
        long long newState = (state & ~(3LL << (2 * j))) | (static_cast<long
long>(left) << (2 * (j + 1)));
        newDp2[newState] += cnt;
    }
}

// 3. 只有上插头
else if (left == 0 && up != 0) {
    // 向右延伸
    if (j < m - 1 && grid[i][j+1] == 1) {
        newDp2[state] += cnt;
    }
    // 向下延伸
    if (i < n - 1 && grid[i+1][j] == 1) {
        long long newState = (state & ~(3LL << (2 * (j + 1)))) |
(static_cast<long long>(up) << (2 * j));
        newDp2[newState] += cnt;
    }
}

// 4. 同时有左插头和上插头
else {
    // 合并插头
    long long newState = (state & ~(3LL << (2 * j))) & ~(3LL << (2 * (j + 1)));
}

// 如果是形成回路的最后一步

```

```

        if (left == up) {
            // 检查是否所有插头都已连接
            if (newState == 0 && i == n - 1 && j == m - 1) {
                newDp2[newState] += cnt;
            }
        } else {
            // 合并两个不同的插头
            newDp2[newState] += cnt;
        }
    }

    dp. swap(newDp2);
}

}

// 最终状态应该是没有任何插头（形成回路）
return dp.count(0) ? dp[0] : 0;
}

// ====== 冷门模型：树上背包的优化 ======
/***
 * 树上背包的 DFS 处理函数
 */
void dfsTreeKnapsack(int u, int parent, int capacity, const vector<vector<int>>& tree,
                      const vector<int>& weights, const vector<int>& values,
                      vector<vector<int>>& dp, vector<int>& size) {
    // 初始化当前节点
    size[u] = 1;
    if (weights[u] <= capacity) {
        dp[u][weights[u]] = values[u];
    }

    // 对每个子节点，按照子树大小排序，小的先合并
    vector<pair<int, int>> children;
    for (int v : tree[u]) {
        if (v != parent) {
            dfsTreeKnapsack(v, u, capacity, tree, weights, values, dp, size);
            children.emplace_back(size[v], v);
        }
    }

    // 按子树大小排序

```

```

sort(children.begin(), children.end());

for (const auto& [sz, v] : children) {
    // 逆序遍历容量，避免重复计算
    for (int i = min(size[u], capacity); i >= 0; --i) {
        if (dp[u][i] == 0 && i != 0) continue;
        for (int j = 1; j <= min(sz, capacity - i); ++j) {
            if (dp[v][j] > 0 && i + j <= capacity) {
                dp[u][i + j] = max(dp[u][i + j], dp[u][i] + dp[v][j]);
            }
        }
    }

    // 更新子树大小
    size[u] += sz;
}

}

/***
 * 树上背包的优化实现（小到大合并）
 *
 * 问题描述：
 * 在树上选择一些节点，使得总重量不超过容量，且总价值最大
 *
 * 解题思路：
 * 1. 使用后序遍历处理子树
 * 2. 使用小到大合并的策略优化复杂度
 * 3. 对于每个节点，维护一个容量为 capacity 的背包
 *
 * 参数：
 * root: 根节点
 * capacity: 背包容量
 * tree: 树的邻接表
 * weights: 每个节点的重量
 * values: 每个节点的价值
 *
 * 返回：
 * int: 最大价值
 *
 * 时间复杂度：O(n*capacity^2)，但通过小到大合并可以降低常数
 * 空间复杂度：O(n*capacity)
 */
int treeKnapsackOptimized(int root, int capacity, const vector<vector<int>>& tree,

```

```

        const vector<int>& weights, const vector<int>& values) {

    int n = tree.size();
    // 初始化 dp 数组
    vector<vector<int>> dp(n, vector<int>(capacity + 1, 0));
    vector<int> size(n, 0);

    // 深度优先搜索处理子树
    dfsTreeKnapsack(root, -1, capacity, tree, weights, values, dp, size);

    // 返回根节点的最大价值
    return *max_element(dp[root].begin(), dp[root].end());
}

// ===== 补充题目与应用 =====
// 以下是一些使用上述高级 DP 技术的经典题目及其代码实现

/***
 * LeetCode 72. 编辑距离
 * 题目链接: https://leetcode-cn.com/problems/edit-distance/
 *
 * 问题描述:
 * 给你两个单词 word1 和 word2，计算出将 word1 转换成 word2 所使用的最少操作数。
 * 你可以对一个单词进行如下三种操作：插入一个字符、删除一个字符、替换一个字符。
 *
 * 解题思路:
 * 使用二维 DP，dp[i][j] 表示 word1 的前 i 个字符转换为 word2 的前 j 个字符所需的最少操作数。
 *
 * 时间复杂度: O(m*n)
 * 空间复杂度: O(m*n)
 */
int editDistance(const string& word1, const string& word2) {
    int m = word1.size();
    int n = word2.size();
    // dp[i][j] 表示 word1 的前 i 个字符转换为 word2 的前 j 个字符所需的最少操作数
    vector<vector<int>> dp(m + 1, vector<int>(n + 1));

    // 初始化边界
    for (int i = 0; i <= m; ++i) dp[i][0] = i;
    for (int j = 0; j <= n; ++j) dp[0][j] = j;

    // 动态规划填表
    for (int i = 1; i <= m; ++i) {
        for (int j = 1; j <= n; ++j) {
            if (word1[i - 1] == word2[j - 1])
                dp[i][j] = dp[i - 1][j - 1];
            else
                dp[i][j] = min(dp[i - 1][j], dp[i][j - 1]) + 1;
        }
    }
}

```

```

        if (word1[i-1] == word2[j-1]) {
            dp[i][j] = dp[i-1][j-1];
        } else {
            dp[i][j] = min({dp[i-1][j], dp[i][j-1], dp[i-1][j-1]}) + 1;
        }
    }

    return dp[m][n];
}

/***
 * LeetCode 300. 最长递增子序列
 * 题目链接: https://leetcode-cn.com/problems/longest-increasing-subsequence/
 *
 * 问题描述:
 * 给你一个整数数组 nums，找到其中最长严格递增子序列的长度。
 *
 * 解题思路:
 * 使用贪心 + 二分查找优化的 DP 方法。
 * tails[i] 表示长度为 i+1 的递增子序列的末尾元素的最小值。
 *
 * 时间复杂度: O(n log n)
 * 空间复杂度: O(n)
 */
int lengthOfLIS(vector<int>& nums) {
    if (nums.empty()) return 0;

    vector<int> tails;
    for (int num : nums) {
        // 二分查找 num 应该插入的位置
        auto it = lower_bound(tails.begin(), tails.end(), num);
        if (it == tails.end()) {
            tails.push_back(num);
        } else {
            *it = num;
        }
    }

    return tails.size();
}

/***

```

```

* LeetCode 322. 零钱兑换
* 题目链接: https://leetcode-cn.com/problems/coin-change/
*
* 问题描述:
* 给你一个整数数组 coins，表示不同面额的硬币；以及一个整数 amount，表示总金额。
* 计算并返回可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额，返回-1。
*
* 解题思路:
* 使用完全背包的思想，dp[i]表示凑成金额 i 所需的最少硬币数。
*
* 时间复杂度: O(amount * n)
* 空间复杂度: O(amount)
*/
int coinChange(vector<int>& coins, int amount) {
    // 初始化 dp 数组为无穷大
    const int INF = INT_MAX - 1; // 避免溢出
    vector<int> dp(amount + 1, INF);
    dp[0] = 0; // 凑成金额 0 需要 0 个硬币

    for (int coin : coins) {
        for (int i = coin; i <= amount; ++i) {
            if (dp[i - coin] != INF) {
                dp[i] = min(dp[i], dp[i - coin] + 1);
            }
        }
    }

    return dp[amount] != INF ? dp[amount] : -1;
}

/**
* 矩阵链乘法问题
* 题目来源: 算法导论
*
* 问题描述:
* 给定一系列矩阵，计算乘法顺序使得标量乘法的次数最少。
*
* 解题思路:
* 使用区间 DP，dp[i][j] 表示计算第 i 到第 j 个矩阵的乘积所需的最少标量乘法次数。
* 可以使用 Knuth 优化进一步降低时间复杂度。
*
* 时间复杂度: O(n^3)
* 空间复杂度: O(n^2)

```

```

*/
pair<vector<vector<long long>>, vector<vector<int>>> matrixChainOrder(const vector<int>& p) {
    int n = p.size() - 1; // 矩阵的个数
    // dp[i][j]表示计算第 i 到第 j 个矩阵的乘积所需的最少标量乘法次数
    const long long INF = LLONG_MAX;
    vector<vector<long long>> dp(n + 1, vector<long long>(n + 1, INF));
    // s[i][j]记录最优分割点
    vector<vector<int>> s(n + 1, vector<int>(n + 1, 0));

    // 单个矩阵的代价为 0
    for (int i = 1; i <= n; ++i) {
        dp[i][i] = 0;
    }

    // 枚举区间长度
    for (int length = 2; length <= n; ++length) {
        for (int i = 1; i <= n - length + 1; ++i) {
            int j = i + length - 1;
            dp[i][j] = INF;
            // 枚举分割点
            for (int k = i; k < j; ++k) {
                // 计算当前分割点的代价
                long long cost = dp[i][k] + dp[k + 1][j] + static_cast<long long>(p[i - 1]) *
p[k] * p[j];
                if (cost < dp[i][j]) {
                    dp[i][j] = cost;
                    s[i][j] = k;
                }
            }
        }
    }

    return {dp, s};
}

```

```

/**
 * 旅行商问题
 * 题目来源：算法竞赛经典问题
 *
 * 问题描述：
 * 给定一个完全图，找到一条访问每个城市恰好一次并返回起点的最短路径。
 *
 * 解题思路：

```

```

* 使用状态压缩 DP, dp[mask][u] 表示访问过的城市集合为 mask, 当前在城市 u 时的最短路径长度。
*
* 时间复杂度: O(n^2 * 2^n)
* 空间复杂度: O(n * 2^n)
*/
long long travelingSalesmanProblem(const vector<vector<long long>>& graph) {
    int n = graph.size();
    // dp[mask][u] 表示访问过的城市集合为 mask, 当前在城市 u 时的最短路径长度
    const long long INF = LLONG_MAX;
    vector<vector<long long>> dp(1 << n, vector<long long>(n, INF));

    // 初始状态: 只访问了起点, 路径长度为 0
    for (int i = 0; i < n; ++i) {
        dp[1 << i][i] = 0;
    }

    // 枚举所有可能的状态
    for (int mask = 1; mask < (1 << n); ++mask) {
        // 枚举当前所在的城市
        for (int u = 0; u < n; ++u) {
            if (!(mask & (1 << u))) continue;
            // 枚举下一个要访问的城市
            for (int v = 0; v < n; ++v) {
                if (mask & (1 << v)) continue;
                int newMask = mask | (1 << v);
                if (dp[mask][u] != INF && graph[u][v] != INF) {
                    if (dp[newMask][v] > dp[mask][u] + graph[u][v]) {
                        dp[newMask][v] = dp[mask][u] + graph[u][v];
                    }
                }
            }
        }
    }

    // 找到最短的回路
    long long result = INF;
    for (int u = 0; u < n; ++u) {
        if (dp[(1 << n) - 1][u] != INF && graph[u][0] != INF) {
            result = min(result, dp[(1 << n) - 1][u] + graph[u][0]);
        }
    }

    return result != INF ? result : -1;
}

```

```
}
```

```
/**
```

```
* LeetCode 1039. 多边形三角剖分的最低得分
```

```
* 题目链接: https://leetcode-cn.com/problems/minimum-score-triangulation-of-polygon/
```

```
*
```

```
* 问题描述:
```

```
* 给定一个凸多边形，将其三角剖分，使得所有三角形的顶点乘积之和最小。
```

```
*
```

```
* 解题思路:
```

```
* 使用区间 DP， $dp[i][j]$  表示从顶点  $i$  到顶点  $j$  的多边形三角剖分的最小得分。
```

```
*
```

```
* 时间复杂度:  $O(n^3)$ 
```

```
* 空间复杂度:  $O(n^2)$ 
```

```
*/
```

```
int minimumScoreTriangulation(vector<int>& values) {
```

```
    int n = values.size();
```

```
    //  $dp[i][j]$  表示从顶点  $i$  到顶点  $j$  的多边形三角剖分的最小得分
```

```
    vector<vector<int>> dp(n, vector<int>(n, 0));
```

```
    // 枚举区间长度
```

```
    for (int length = 3; length <= n; ++length) {
```

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

```
            int j = i + length - 1;
```

```
            dp[i][j] = INT_MAX;
```

```
            // 枚举中间点
```

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

```
                dp[i][j] = min(dp[i][j], dp[i][k] + dp[k][j] + values[i] * values[k] *
```

```
values[j]);
```

```
}
```

```
}
```

```
}
```

```
    return dp[0][n - 1];
```

```
}
```

```
/**
```

```
* LeetCode 877. 石子游戏
```

```
* 题目链接: https://leetcode-cn.com/problems/stone-game/
```

```
*
```

```
* 问题描述:
```

```
* 给定一个表示石子堆的数组，两个玩家轮流从两端取石子，每次只能取一个，取到最后一个石子的人获胜。
```

```
* 判断先手是否必胜。
```

```

/*
* 解题思路:
* 使用区间 DP, dp[i][j] 表示在区间 [i, j] 中, 先手能获得的最大净胜分。
*
* 时间复杂度: O(n^2)
* 空间复杂度: O(n^2)
*/
bool stoneGame(vector<int>& piles) {
    int n = piles.size();
    // dp[i][j] 表示在区间 [i, j] 中, 先手能获得的最大净胜分
    vector<vector<int>> dp(n, vector<int>(n, 0));

    // 初始化单个石子堆
    for (int i = 0; i < n; ++i) {
        dp[i][i] = piles[i];
    }

    // 枚举区间长度
    for (int length = 2; length <= n; ++length) {
        for (int i = 0; i <= n - length; ++i) {
            int j = i + length - 1;
            // 先手可以选择取左边或右边
            dp[i][j] = max(piles[i] - dp[i + 1][j], piles[j] - dp[i][j - 1]);
        }
    }

    // 先手净胜分大于 0 则必胜
    return dp[0][n - 1] > 0;
}

/**
* LeetCode 233. 数字 1 的个数
* 题目链接: https://leetcode-cn.com/problems/number-of-digit-one/
*
* 问题描述:
* 给定一个整数 n, 计算所有小于等于 n 的非负整数中数字 1 出现的个数。
*
* 解题思路:
* 使用数位 DP, 逐位处理每一位上 1 出现的次数。
*
* 时间复杂度: O(log n)
* 空间复杂度: O(log n)
*/

```

```

int countDigitOne(int n) {
    if (n <= 0) return 0;

    string s = to_string(n);
    int length = s.size();
    int count = 0;

    // 逐位处理
    for (int i = 0; i < length; ++i) {
        int high = 0;
        if (i > 0) {
            high = stoi(s.substr(0, i));
        }
        int current = s[i] - '0';
        int low = 0;
        if (i < length - 1) {
            low = stoi(s.substr(i + 1));
        }
        long long digit = pow(10, length - i - 1);

        if (current == 0) {
            // 当前位为0，高位决定
            count += high * digit;
        } else if (current == 1) {
            // 当前位为1，高位+低位+1
            count += high * digit + low + 1;
        } else {
            // 当前位大于1，高位+1
            count += (high + 1) * digit;
        }
    }

    return count;
}

/***
 * 树节点定义
 */
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
}

```

```

};

/***
 * 树形 DP 处理函数 - 打家劫舍 III
 */
pair<int, int> robDFS(TreeNode* node) {
    if (!node) return {0, 0};

    auto left = robDFS(node->left);
    auto right = robDFS(node->right);

    // rob_current 表示偷当前节点, not_rob_current 表示不偷当前节点
    int rob_current = node->val + left.second + right.second;
    int not_rob_current = max(left.first, left.second) + max(right.first, right.second);

    return {rob_current, not_rob_current};
}

/***
 * LeetCode 337. 打家劫舍 III
 * 题目链接: https://leetcode-cn.com/problems/house-robber-iii/
 *
 * 问题描述:
 * 在上次打劫完一条街道之后和一圈房屋后，小偷又发现了一个新的可行窃的地区。
 * 这个地区只有一个入口，我们称之为“根”。除了“根”之外，每栋房子有且只有一个“父”房子与之相连。
 * 一番侦察之后，聪明的小偷意识到“这个地方的所有房屋的排列类似于一棵二叉树”。
 * 如果两个直接相连的房子在同一天晚上被打劫，房屋将自动报警。
 * 计算在不触动警报的情况下，小偷一晚能够盗取的最高金额。
 *
 * 解题思路:
 * 使用树形 DP，对于每个节点，维护两个状态：偷或不偷。
 *
 * 时间复杂度: O(n)
 * 空间复杂度: O(h)，h 为树的高度
 */
int rob(TreeNode* root) {
    auto result = robDFS(root);
    return max(result.first, result.second);
}

/***
 * 蒙斯特曼问题

```

* 题目来源：算法竞赛问题

*

* 问题描述：

* 在网格中放置怪物，使得任何两个怪物都不在同一行、同一列或对角线上。

*

* 解题思路：

* 使用状态压缩 DP， $dp[mask]$ 表示处理到当前行，已放置的列的状态为 $mask$ 时的方案数。

*

* 时间复杂度： $O(n * 2^n)$

* 空间复杂度： $O(2^n)$

*/

```
long long monsterGame(const vector<vector<int>>& grid) {
    int n = grid.size();
    // dp[mask] 表示处理到当前行，已放置的列的状态为 mask 时的方案数
    vector<long long> dp(1 << n, 0);
    dp[0] = 1;

    for (int i = 0; i < n; ++i) {
        vector<long long> newDp(1 << n, 0);
        for (int mask = 0; mask < (1 << n); ++mask) {
            if (dp[mask] == 0) continue;
            // 枚举所有可能的放置位置
            for (int j = 0; j < n; ++j) {
                // 检查是否可以在(i, j)放置怪物
                if (!(mask & (1 << j)) && grid[i][j] == 1) {
                    // 检查对角线
                    bool valid = true;
                    for (int k = 0; k < i; ++k) {
                        if (mask & (1 << k) && abs(k - j) == i - k) {
                            valid = false;
                            break;
                        }
                    }
                }
                if (valid) {
                    newDp[mask | (1 << j)] += dp[mask];
                }
            }
        }
        dp. swap(newDp);
    }

    return dp[(1 << n) - 1];
}
```

```
}
```

```
/**  
 * 三维背包问题  
 * 题目来源：算法竞赛问题  
 *  
 * 问题描述：  
 * 有 n 个物品，每个物品有体积、重量、价值三个属性，背包有体积和重量两个限制，求最大价值。  
 *  
 * 解题思路：  
 * 使用三维 DP，dp[i][j][k] 表示前 i 个物品，体积为 j，重量为 k 时的最大价值。  
 *  
 * 时间复杂度：O(n * V * W)  
 * 空间复杂度：O(n * V * W)  
 */  
  
int threeDimensionKnapsack(int n, const pair<int, int>& capacity, const vector<tuple<int, int, int>>& items) {  
    int V = capacity.first;  
    int W = capacity.second;  
    // 初始化 dp 数组  
    vector<vector<vector<int>>> dp(n + 1, vector<vector<int>>(V + 1, vector<int>(W + 1, 0)));  
  
    for (int i = 1; i <= n; ++i) {  
        int v = get<0>(items[i-1]);  
        int w = get<1>(items[i-1]);  
        int val = get<2>(items[i-1]);  
  
        for (int j = 0; j <= V; ++j) {  
            for (int k = 0; k <= W; ++k) {  
                // 不选当前物品  
                dp[i][j][k] = dp[i-1][j][k];  
                // 选当前物品（如果有足够的空间）  
                if (j >= v && k >= w) {  
                    dp[i][j][k] = max(dp[i][j][k], dp[i-1][j-v][k-w] + val);  
                }  
            }  
        }  
    }  
  
    return dp[n][V][W];  
}
```

```
/**
```

```

* 凸包优化技巧示例
* 题目来源：算法竞赛问题
*
* 问题描述：
* 当状态转移方程形如  $dp[i] = \min\{dp[j] + a[i] * b[j] + c\}$  时，可以使用凸包优化。
*
* 解题思路：
* 将转移方程转换为直线的形式，维护凸包以快速查询最小值。
*
* 时间复杂度：O(n)
* 空间复杂度：O(n)
*/

```

class ConvexHullTrick {

private:

```

    struct Line {
        double k, b;
        Line(double k_ = 0, double b_ = 0) : k(k_), b(b_) {}
    };
    deque<Line> dq;

```

// 获取队列中倒数第 n 个元素

```

    Line getNthLast(int n) {
        vector<Line> temp(dq.begin(), dq.end());
        return temp[temp.size() - n];
    }

```

public:

```

    // 添加一条直线
    void addLine(double k, double b) {
        // 当队列中至少有两条直线时，检查是否需要删除末尾的直线
        while (dq.size() >= 2) {
            Line l1 = getNthLast(2);
            Line l2 = dq.back();

            // 判断直线 l1 和 l2 的交点是否在 l2 和新直线的交点右侧
            if ((l2.b - l1.b) * (k - l2.k) >= (b - l2.b) * (l2.k - l1.k)) {
                dq.pop_back();
            } else {
                break;
            }
        }
        dq.emplace_back(k, b);
    }
}

```

```

// 查询 x 对应的最小值
double query(double x) {
    while (dq.size() >= 2) {
        Line 11 = dq.front();
        dq.pop_front();
        Line 12 = dq.front();

        if (11.k * x + 11.b >= 12.k * x + 12.b) {
            // 继续弹出
            continue;
        } else {
            dq.push_front(11); // 恢复 11
            break;
        }
    }

    if (dq.empty()) {
        return DBL_MAX;
    }

    Line 1 = dq.front();
    return 1.k * x + 1.b;
}

};

// ===== 高级优化体系: SMAWK 算法 (行最小查询) =====
/***
 * SMAWK 算法用于在 Monge 矩阵中快速查找每行的最小值
 *
 * 问题描述:
 * 给定一个 Monge 矩阵, 快速找到每行的最小值位置
 *
 * 解题思路:
 * 1. Monge 矩阵满足性质: matrix[i][j] + matrix[i+1][j+1] ≤ matrix[i][j+1] + matrix[i+1][j]
 * 2. SMAWK 算法利用这一性质, 可以在 O(m+n) 时间内找到每行的最小值
 * 3. 主要步骤包括行压缩和递归求解
 *
 * 应用题目:
 * - POJ 3156 Interconnect
 * - Codeforces 472D Design Tutorial: Inverse the Problem
 * - SPOJ MCQUERY
 *
 * 时间复杂度: O(m+n), 其中 m 是行数, n 是列数

```

```

* 空间复杂度: O(m+n)
*/
class SMAWK {
private:
    // 行压缩: 只保留可能成为最小值的行
    static vector<int> reduceRows(const vector<vector<int>>& matrix, const vector<int>& rows) {
        vector<int> stack;
        for (int i : rows) {
            while (stack.size() >= 2) {
                int j1 = stack.back();
                stack.pop_back();
                int j2 = stack.back();
                stack.pop_back();
                stack.push_back(j1); // 恢复栈状态

                // 比较两个行在列 stack.size()-2 处的值
                int col = stack.size() - 2;
                if (col < matrix[0].size()) {
                    if (matrix[j2][col] <= matrix[i][col]) {
                        break;
                    } else {
                        stack.pop_back(); // 移除 j1
                    }
                } else {
                    break;
                }
            }
            stack.push_back(i);
        }
        return stack;
    }

    // 递归实现 SMAWK 算法
    static vector<int> smawkRec(const vector<vector<int>>& matrix, const vector<int>& rows, const
vector<int>& cols) {
        int m = rows.size();
        vector<int> result(m, -1);

        if (m == 0) {
            return result;
        }

        // 行压缩

```

```

vector<int> reducedRows = reduceRows(matrix, rows);

// 递归求解列数为奇数的子问题
vector<int> halfCols;
for (int i = 1; i < cols.size(); i += 2) {
    halfCols.push_back(cols[i]);
}

vector<int> minCols(reducedRows.size(), -1);

if (!halfCols.empty()) {
    // 递归求解
    vector<int> subResult = smawkRec(matrix, reducedRows, halfCols);
    // 复制结果
    for (int i = 0; i < subResult.size(); ++i) {
        minCols[i] = subResult[i];
    }
}

// 扩展结果到所有列
int k = 0; // minCols 的索引

for (int i = 0; i < m; ++i) {
    int row = rows[i];
    // 确定当前行的最小值可能在哪个区间
    int start = 0;
    if (i > 0 && k > 0 && minCols[k-1] != -1) {
        start = minCols[k-1];
    }
    int end = (k < minCols.size() && minCols[k] != -1) ? minCols[k] : cols.back();

    // 在这个区间内查找最小值
    int minValue = INT_MAX;
    int minCol = start;

    // 找到 start 和 end 在 cols 中的索引
    int startIdx = 0, endIdx = cols.size() - 1;
    for (int idx = 0; idx < cols.size(); ++idx) {
        if (cols[idx] == start) startIdx = idx;
        if (cols[idx] == end) endIdx = idx;
    }

    for (int idx = startIdx; idx <= endIdx; ++idx) {

```

```

        int col = cols[idx];
        if (col < matrix[0].size() && matrix[row][col] < minValue) {
            minValue = matrix[row][col];
            minCol = col;
        }
    }

    result[i] = minCol;

    // 如果当前行在 reducedRows 中，且不是最后一行，k 前进
    if (k < reducedRows.size() && row == reducedRows[k]) {
        ++k;
    }
}

return result;
}

public:
// SMAWK 算法主入口
static vector<int> solve(const vector<vector<int>>& matrix) {
    if (matrix.empty() || matrix[0].empty()) {
        return {};
    }

    int m = matrix.size();
    int n = matrix[0].size();

    // 构造行索引和列索引数组
    vector<int> rows(m);
    vector<int> cols(n);
    for (int i = 0; i < m; ++i) rows[i] = i;
    for (int j = 0; j < n; ++j) cols[j] = j;

    // 调用递归实现
    return smawkRec(matrix, rows, cols);
}

```

```

// 应用示例：寻找每一行的最小元素
static vector<int> findRowMins(const vector<vector<int>>& matrix) {
    vector<int> minCols = solve(matrix);
    vector<int> result;
    for (int i = 0; i < matrix.size(); ++i) {

```

```

        result.push_back(matrix[i][minCols[i]]);
    }
    return result;
}
};

// ===== 高级优化体系: Aliens Trick (二分约束参数+可行性 DP) =====
/***
 * Aliens Trick (二分约束参数+可行性 DP)
 *
 * 问题描述:
 * 解决带约束的优化问题, 通常形如最小化总成本, 同时满足某些约束条件
 *
 * 解题思路:
 * 1. 将约束条件转化为参数  $\lambda$ , 构造拉格朗日函数
 * 2. 对  $\lambda$  进行二分查找, 使用可行性 DP 判断当前  $\lambda$  下是否满足约束
 * 3. 根据可行性 DP 的结果调整二分区间
 *
 * 应用题目:
 * - Codeforces 739E Gosha is Hunting
 * - POJ 3686 The Windy's
 * - SPOJ QTREE5
 *
 * 时间复杂度:  $O(\log((right-left)/\text{eps}) * T(DP))$ , 其中  $T(DP)$  是一次 DP 的时间复杂度
 * 空间复杂度:  $O(DP \text{ 空间复杂度})$ 
*/
class AliensTrick {
private:
    // 结果类
    struct Result {
        double lambdaVal;
        double value;

        Result(double l, double v) : lambdaVal(l), value(v) {}
    };

    // 成本函数结果类
    struct CostFunctionResult {
        double value;
        int constraint;

        CostFunctionResult(double v, int c) : value(v), constraint(c) {}
    };
}
```

```

// Aliens Trick 主入口
template<typename CostFunc, typename CheckFunc>
static Result solve(CostFunc costFunc, CheckFunc checkFunc, double left, double right, double
eps = 1e-7) {
    double bestLambda = left;
    double bestValue = 0.0;

    // 二分查找参数 lambda
    while (right - left > eps) {
        double mid = (left + right) / 2;
        // 计算当前参数下的解和约束值
        CostFunctionResult result = costFunc(mid);

        if (checkFunc(result.constraint)) {
            // 满足约束，尝试更小的参数
            right = mid;
            bestLambda = mid;
            bestValue = result.value;
        } else {
            // 不满足约束，需要增大参数
            left = mid;
        }
    }

    return Result(bestLambda, bestValue);
}

public:
    // 应用示例：将数组分成恰好 k 个部分，使得最大子数组和最小（LeetCode 410 的变种）
    static double splitArrayK(const vector<int>& nums, int k) {
        // 计算数组元素和作为二分上限
        long long sumVal = 0;
        for (int num : nums) sumVal += num;

        // 成本函数：使用 DP 计算在给定 lambda 下的最小成本
        auto costFunc = [&nums](double lambda) -> CostFunctionResult {
            int n = nums.size();
            const double INF = 1e18;
            vector<double> dp(n + 1, INF);
            vector<int> cnt(n + 1, 0);

            dp[0] = 0;

```

```

cnt[0] = 0;

for (int i = 1; i <= n; ++i) {
    long long sumSeg = 0;
    for (int j = i - 1; j >= 0; --j) {
        sumSeg += nums[j];
        if (dp[j] != INF) {
            double current = dp[j] + 1LL * sumSeg * sumSeg + lambda; // lambda 作为惩罚项
            if (current < dp[i]) {
                dp[i] = current;
                cnt[i] = cnt[j] + 1;
            }
        }
    }
}

return CostFunctionResult(dp[n], cnt[n]);
};

// 约束检查函数: 确保分割次数不超过 k
auto checkFunc = [k](int constraint) -> bool {
    return constraint <= k;
};

// 执行 Aliens Trick
Result result = solve(costFunc, checkFunc, 0.0, 1LL * sumVal * sumVal, 1e-7);
return result.value;
};

// ===== 图上 DP→最短路: 分层图建模 =====
/***
 * 分层图 Dijkstra 算法
 *
 * 问题描述:
 * 给定一个图, 允许最多使用 k 次特殊操作 (如跳跃、免费通行等), 求最短路径
 *
 * 解题思路:
 * 1. 构建分层图, 每层代表使用不同次数的特殊操作
 * 2. 对于每个节点 u, 在第 i 层表示到达 u 时已经使用了 i 次特殊操作
 * 3. 使用 Dijkstra 算法在分层图上寻找最短路径
 */

```

```

* 应用题目：
* - LeetCode 787. K 站中转内最便宜的航班
* - POJ 3159 Candies
* - HDU 2957 Safety Assessment
*
* 时间复杂度: O((n*k + m*k) log(n*k))
* 空间复杂度: O(n*k + m*k)
*/
class LayeredGraphShortestPath {
private:
    // 边类
    struct Edge {
        int to;
        int weight;

        Edge(int t, int w) : to(t), weight(w) {}
    };

    // 分层图最短路径算法
    static int solve(int n, const vector<vector<Edge>>& edges, int k, int start, int end) {
        // 构建分层图的邻接表
        vector<vector<Edge>> layeredGraph(n * (k + 1));
        int totalNodes = n * (k + 1);

        // 添加普通边（不使用特殊操作）
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j <= k; ++j) {
                int fromNode = i + j * n;
                for (const Edge& edge : edges[i]) {
                    layeredGraph[fromNode].emplace_back(edge.to + j * n, edge.weight);
                }
            }
        }

        // 添加使用特殊操作的边（如果允许的话）
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < k; ++j) {
                int fromNode = i + j * n;
                for (const Edge& edge : edges[i]) {
                    // 这里假设特殊操作可以免费通行（权为 0），具体根据问题调整
                    layeredGraph[fromNode].emplace_back(edge.to + (j + 1) * n, 0);
                }
            }
        }
    }
}

```

```

}

// Dijkstra 算法
const int INF = INT_MAX;
vector<int> dist(totalNodes, INF);
dist[start] = 0; // 起始点在第 0 层

// 使用优先队列，按距离排序
priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
pq.emplace(0, start);

while (!pq.empty()) {
    auto [currentDist, u] = pq.top();
    pq.pop();

    if (currentDist > dist[u]) {
        continue;
    }

    for (const Edge& edge : layeredGraph[u]) {
        int v = edge.to;
        int w = edge.weight;
        if (dist[v] > currentDist + w && currentDist != INF) {
            dist[v] = currentDist + w;
            pq.emplace(dist[v], v);
        }
    }
}

// 取所有层中到达终点的最小值
int result = INF;
for (int i = 0; i <= k; ++i) {
    if (dist[end + i * n] < result) {
        result = dist[end + i * n];
    }
}

return (result != INF) ? result : -1;
}

public:
// 应用示例：LeetCode 787. K 站中转内最便宜的航班
static int findCheapestPrice(int n, const vector<vector<int>>& flights, int src, int dst, int

```

```

k) {
    // 构建图的邻接表
    vector<vector<Edge>> edges(n);
    for (const auto& flight : flights) {
        edges[flight[0]].emplace_back(flight[1], flight[2]);
    }

    // 调用分层图算法，注意这里 k 站中转意味着可以乘坐 k+1 次航班
    return solve(n, edges, k + 1, src, dst);
}

};

// ====== 冷门模型：期望 DP 遇环的方程组解（高斯消元） ======
/***
 * 期望 DP 处理有环情况（使用高斯消元）
 *
 * 问题描述：
 * 在有环的状态转移图中计算期望
 *
 * 解题思路：
 * 1. 对于每个状态，建立期望方程
 * 2. 使用高斯消元求解方程组
 *
 * 应用题目：
 * - LeetCode 837. 新 21 点
 * - POJ 3744 Scout YYF I
 * - HDU 4405 Aeroplane chess
 *
 * 时间复杂度：O(n^3)
 * 空间复杂度：O(n^2)
 */
class ExpectationDPwithGaussian {
private:
    // 转移类
    struct Transition {
        int to;
        double probability;

        Transition(int t, double p) : to(t), probability(p) {}

    };

    // 高斯消元法求解线性方程组
    static vector<double> gaussianElimination(vector<vector<double>> matrix) {

```

```

int n = matrix.size();
const double eps = 1e-9;

// 高斯消元过程
for (int i = 0; i < n; ++i) {
    // 找到主元行 (当前列中绝对值最大的行)
    int maxRow = i;
    for (int j = i; j < n; ++j) {
        if (abs(matrix[j][i]) > abs(matrix[maxRow][i])) {
            maxRow = j;
        }
    }
}

// 交换主元行和当前行
if (maxRow != i) {
    swap(matrix[i], matrix[maxRow]);
}

// 如果主元为 0, 方程组可能有无穷多解或无解
if (abs(matrix[i][i]) < eps) {
    // 这里简化处理, 假设方程组总是有解
    continue;
}

// 消元过程
for (int j = 0; j < n; ++j) {
    if (j != i && abs(matrix[j][i]) > eps) {
        double factor = matrix[j][i] / matrix[i][i];
        for (int k = i; k <= n; ++k) {
            matrix[j][k] -= factor * matrix[i][k];
        }
    }
}

// 回代求解
vector<double> x(n);
for (int i = 0; i < n; ++i) {
    x[i] = matrix[i][n] / matrix[i][i];
}

return x;
}

```

```

// 期望 DP 主入口
static vector<double> solve(int n, const vector<vector<Transition>>& transitions, const
vector<double>& cost) {
    // 构建线性方程组的增广矩阵
    vector<vector<double>> matrix(n, vector<double>(n + 1, 0.0));

    for (int i = 0; i < n; ++i) {
        matrix[i][i] = 1.0; // 方程左边: E[i] - sum(p_ij * E[j]) = cost[i]
        matrix[i][n] = cost[i];

        for (const Transition& t : transitions[i]) {
            if (i != t.to) { // 避免自环的特殊处理
                matrix[i][t.to] -= t.probability;
            }
        }
    }

    // 使用高斯消元求解
    return gaussianElimination(matrix);
}

public:
    // 应用示例: LeetCode 837. 新 21 点 (简化版本)
    static double new21Game(int N, int K, int W) {
        if (K == 0 || N >= K + W) {
            return 1.0;
        }

        int n = K + W;
        vector<vector<Transition>> transitions(n + 1);
        vector<double> cost(n + 1, 0.0);

        // 构建转移概率
        for (int i = 0; i < K; ++i) {
            for (int w = 1; w <= W; ++w) {
                int nextState = min(i + w, n);
                transitions[i].emplace_back(nextState, 1.0 / W);
            }
        }

        // 终止状态的期望为是否<=N
        for (int i = K; i <= n; ++i) {

```

```

        cost[i] = (i <= N) ? 1.0 : 0.0;
        transitions[i].emplace_back(i, 1.0); // 自环
    }

    vector<double> result = solve(n + 1, transitions, cost);
    return result[0];
}

// ===== 冷门模型: 插头 DP (轮廓线 DP) =====
/***
 * 插头 DP (轮廓线 DP) 示例: 求网格中哈密顿回路的数量
 *
 * 问题描述:
 * 给定一个网格, 求其中哈密顿回路的数量
 *
 * 解题思路:
 * 1. 使用轮廓线 DP, 记录当前处理到的位置和轮廓线状态
 * 2. 插头表示连接的状态, 通常用二进制表示
 * 3. 使用哈希表优化空间复杂度
 * 4. 实现合法性判定与对称剪枝
 *
 * 应用题目:
 * - HDU 1693 Eat the Trees
 * - SPOJ MATCH2 Match the Brackets II
 * - Codeforces 1435F Cyclic Shifts Sorting
 *
 * 时间复杂度: O(n*m*4^min(n, m))
 * 空间复杂度: O(4^min(n, m))
 */
class PlugDP {

private:
    // 插头 DP 求解哈密顿回路数量
    static long long solve(const vector<vector<int>>& grid) {
        if (grid.empty() || grid[0].empty()) {
            return 0;
        }

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

        // 使用哈希表优化空间
        unordered_map<long long, long long> dp;

```

```

dp[0] = 1;

for (int i = 0; i < n; ++i) {
    // 新的一行开始，需要将状态左移两位
    unordered_map<long long, long long> newDp;
    for (const auto& [state, cnt] : dp) {
        // 左移两位，移除最左边的插头
        long long newState = state << 2;
        // 移除可能的高位，只保留 m*2 位
        newState &= (1LL << (2 * m)) - 1;
        newDp[newState] += cnt;
    }
    dp = newDp;
}

for (int j = 0; j < m; ++j) {
    unordered_map<long long, long long> newDp2;

    for (const auto& [state, cnt] : dp) {
        // 当前位置左边和上边的插头状态
        int left = (state >> (2 * j)) & 3;
        int up = (state >> (2 * (j + 1))) & 3;

        // 如果当前位置是障碍物，跳过
        if (grid[i][j] == 0) {
            // 只有当左右插头都不存在时才合法
            if (left == 0 && up == 0) {
                newDp2[state] += cnt;
            }
            continue;
        }

        // 处理各种插头组合情况
        // 1. 没有左插头和上插头
        if (left == 0 && up == 0) {
            // 只能创建新的插头对（用于回路的开始）
            if (i < n - 1 && j < m - 1 &&
                grid[i+1][j] == 1 && grid[i][j+1] == 1) {
                long long newState = state | (1LL << (2 * j)) | (2LL << (2 * (j + 1)));
                newDp2[newState] += cnt;
            }
        }
        // 2. 只有左插头
    }
}

```

```

else if (left != 0 && up == 0) {
    // 向下延伸
    if (i < n - 1 && grid[i+1][j] == 1) {
        newDp2[state] += cnt;
    }
    // 向右延伸
    if (j < m - 1 && grid[i][j+1] == 1) {
        long long newState = (state & ~(3LL << (2 * j))) | (1LL * left << (2
* (j + 1)));
        newDp2[newState] += cnt;
    }
}

// 3. 只有上插头
else if (left == 0 && up != 0) {
    // 向右延伸
    if (j < m - 1 && grid[i][j+1] == 1) {
        newDp2[state] += cnt;
    }
    // 向下延伸
    if (i < n - 1 && grid[i+1][j] == 1) {
        long long newState = (state & ~(3LL << (2 * (j + 1)))) | (1LL * up <<
(2 * j));
        newDp2[newState] += cnt;
    }
}

// 4. 同时有左插头和上插头
else {
    // 合并插头
    long long newState = (state & ~(3LL << (2 * j))) & ~(3LL << (2 * (j +
1)));
}

// 如果是形成回路的最后一步
if (left == up) {
    // 检查是否所有插头都已连接
    if (newState == 0 && i == n - 1 && j == m - 1) {
        newDp2[newState] += cnt;
    }
} else {
    // 合并两个不同的插头
    // 这里可以加入更多的合法性检查和剪枝
    newDp2[newState] += cnt;
}
}

```

```

        }

        dp = newDp2;
    }

}

// 最终状态应该是没有任何插头（形成回路）
return dp.count(0) ? dp[0] : 0;
}

public:
    // 应用示例：网格中的回路计数
    static long long countGridCycles(const vector<vector<int>>& grid) {
        return solve(grid);
    }
};

// ===== 冷门模型：树上背包的优化 =====
/***
 * 树上背包的优化实现（小到大合并）
 *
 * 问题描述：
 * 在树上选择一些节点，使得总重量不超过容量，且总价值最大
 *
 * 解题思路：
 * 1. 使用后序遍历处理子树
 * 2. 使用小到大合并的策略优化复杂度
 * 3. 对于每个节点，维护一个容量为 capacity 的背包
 *
 * 应用题目：
 * - HDU 1561 The more, The Better
 * - POJ 2063 Investment
 * - Codeforces 1152F2 Neko Rules the Catniverse
 *
 * 时间复杂度：O(n*capacity^2)，但通过小到大合并可以降低常数
 * 空间复杂度：O(n*capacity)
 */
class TreeKnapsackOptimized {
private:
    vector<vector<int>> dp;
    vector<int> size;
    const vector<vector<int>>* tree;
    const vector<int>* weights;
}

```

```

const vector<int>* values;
int capacity;
int n;

void dfs(int u, int parent) {
    // 初始化当前节点
    size[u] = 1;
    if ((*weights)[u] <= capacity) {
        dp[u][(*weights)[u]] = max(dp[u][(*weights)[u]], (*values)[u]);
    }

    // 对每个子节点，按照子树大小排序，小的先合并
    vector<pair<int, int>> children;
    for (int v : (*tree)[u]) {
        if (v != parent) {
            dfs(v, u);
            children.emplace_back(size[v], v);
        }
    }

    // 按子树大小排序（小到大）
    sort(children.begin(), children.end());

    for (const auto& [sz, v] : children) {
        // 逆序遍历容量，避免重复计算
        for (int i = min(size[u], capacity); i >= 0; --i) {
            if (dp[u][i] == 0 && i != 0) {
                continue;
            }

            for (int j = 1; j <= min(sz, capacity - i); ++j) {
                if (dp[v][j] > 0 && i + j <= capacity) {
                    dp[u][i + j] = max(dp[u][i + j], dp[u][i] + dp[v][j]);
                }
            }
        }

        // 更新子树大小
        size[u] += sz;
    }
}

public:
    // 树上背包主入口

```

```

int solve(int n, int root, int capacity, const vector<vector<int>>& tree,
         const vector<int>& weights, const vector<int>& values) {
    this->n = n;
    this->capacity = capacity;
    this->tree = &tree;
    this->weights = &weights;
    this->values = &values;

    // 初始化 dp 数组
    dp.assign(n + 1, vector<int>(capacity + 1, 0));
    size.assign(n + 1, 0);

    // 深度优先搜索处理子树
    dfs(root, -1);

    // 返回根节点的最大价值
    int maxValue = 0;
    for (int i = 0; i <= capacity; ++i) {
        maxValue = max(maxValue, dp[root][i]);
    }
    return maxValue;
}

// 应用示例：树上最大价值选择
static int maxTreeValue(int n, int root, int capacity, const vector<vector<int>>& tree,
                        const vector<int>& weights, const vector<int>& values) {
    TreeKnapsackOptimized optimizer;
    return optimizer.solve(n, root, capacity, tree, weights, values);
}

// ===== 补充题目与应用 =====
/***
 * LeetCode 72. 编辑距离
 * 题目链接: https://leetcode-cn.com/problems/edit-distance/
 *
 * 问题描述:
 * 给你两个单词 word1 和 word2，计算出将 word1 转换成 word2 所使用的最少操作数。
 * 你可以对一个单词进行如下三种操作：插入一个字符、删除一个字符、替换一个字符。
 *
 * 时间复杂度: O(m*n)
 * 空间复杂度: O(m*n)
 */

```

```

int editDistance(const string& word1, const string& word2) {
    int m = word1.size();
    int n = word2.size();
    // 初始化 dp 数组
    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));

    // 初始化边界条件
    for (int i = 0; i <= m; ++i) {
        dp[i][0] = i;
    }
    for (int j = 0; j <= n; ++j) {
        dp[0][j] = j;
    }

    // 填充 dp 表
    for (int i = 1; i <= m; ++i) {
        for (int j = 1; j <= n; ++j) {
            if (word1[i-1] == word2[j-1]) {
                dp[i][j] = dp[i-1][j-1];
            } else {
                dp[i][j] = min({dp[i-1][j], dp[i][j-1], dp[i-1][j-1]}) + 1;
            }
        }
    }

    return dp[m][n];
}

```

```

/***
 * LeetCode 300. 最长递增子序列
 * 题目链接: https://leetcode-cn.com/problems/longest-increasing-subsequence/
 *
 * 问题描述:
 * 给你一个整数数组 nums，找到其中最长严格递增子序列的长度。
 *
 * 时间复杂度: O(n log n)
 * 空间复杂度: O(n)
 */
int lengthOfLIS(const vector<int>& nums) {
    if (nums.empty()) {
        return 0;
    }

```

```

vector<int> tails;

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

    if (it == tails.end()) {
        tails.push_back(num);
    } else {
        *it = num;
    }
}

return tails.size();
}

/***
 * LeetCode 322. 零钱兑换
 * 题目链接: https://leetcode-cn.com/problems/coin-change/
 *
 * 问题描述:
 * 给你一个整数数组 coins，表示不同面额的硬币；以及一个整数 amount，表示总金额。
 * 计算并返回可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额，返回-1。
 *
 * 时间复杂度: O(amount * n)
 * 空间复杂度: O(amount)
 */
int coinChange(const vector<int>& coins, int amount) {
    // 初始化 dp 数组，dp[i] 表示凑成金额 i 所需的最少硬币数
    vector<int> dp(amount + 1, amount + 1);
    dp[0] = 0; // 基础情况

    for (int i = 1; i <= amount; ++i) {
        for (int coin : coins) {
            if (coin <= i) {
                dp[i] = min(dp[i], dp[i - coin] + 1);
            }
        }
    }

    return dp[amount] <= amount ? dp[amount] : -1;
}

```

```

/***
 * 矩阵链乘法问题
 * 题目来源：算法导论、POJ 1038
 *
 * 问题描述：
 * 给定一系列矩阵，计算乘法顺序使得标量乘法的次数最少。
 *
 * 时间复杂度：O(n^3)
 * 空间复杂度：O(n^2)
 */
long long matrixChainOrder(const vector<int>& p) {
    int n = p.size() - 1; // 矩阵的个数
    // 初始化 dp 数组
    vector<vector<long long>> dp(n + 1, vector<long long>(n + 1, 0));

    // 枚举子链长度
    for (int length = 2; length <= n; ++length) {
        for (int i = 1; i <= n - length + 1; ++i) {
            int j = i + length - 1;
            dp[i][j] = LLONG_MAX;
            // 枚举分割点
            for (int k = i; k < j; ++k) {
                // 计算在位置 k 分割的代价
                long long cost = dp[i][k] + dp[k+1][j] + 1LL * p[i-1] * p[k] * p[j];
                if (cost < dp[i][j]) {
                    dp[i][j] = cost;
                }
            }
        }
    }

    return dp[1][n];
}

/***
 * 旅行商问题（TSP）
 * 题目来源：算法竞赛、POJ 2480
 *
 * 问题描述：
 * 给定一个完全图，找到一条访问每个城市恰好一次并返回起点的最短路径。
 *
 * 时间复杂度：O(n^2 * 2^n)
 * 空间复杂度：O(n * 2^n)

```

```

*/
int tsp(const vector<vector<int>>& graph) {
    int n = graph.size();
    // dp[mask][u] 表示访问过的城市集合为 mask，当前在城市 u 时的最短路径
    vector<vector<int>> dp(1 << n, vector<int>(n, INT_MAX));

    // 初始化：从城市 0 出发
    dp[1][0] = 0;

    // 枚举所有可能的状态
    for (int mask = 1; mask < (1 << n); ++mask) {
        for (int u = 0; u < n; ++u) {
            if (!(mask & (1 << u))) { // 如果 u 不在 mask 中，跳过
                continue;
            }
            if (dp[mask][u] == INT_MAX) { // 如果无法到达 u，跳过
                continue;
            }

            // 尝试从 u 出发访问未访问的城市 v
            for (int v = 0; v < n; ++v) {
                if (mask & (1 << v)) { // 如果 v 已经访问过，跳过
                    continue;
                }
                if (graph[u][v] == INT_MAX) { // 如果 u 和 v 之间没有边，跳过
                    continue;
                }

                int newMask = mask | (1 << v);
                if (dp[newMask][v] > dp[mask][u] + graph[u][v] && dp[mask][u] != INT_MAX) {
                    dp[newMask][v] = dp[mask][u] + graph[u][v];
                }
            }
        }
    }

    // 找到最短的回路
    int result = INT_MAX;
    for (int u = 0; u < n; ++u) {
        if (graph[u][0] != INT_MAX && dp[(1 << n) - 1][u] != INT_MAX) {
            result = min(result, dp[(1 << n) - 1][u] + graph[u][0]);
        }
    }
}

```

```

return result != INT_MAX ? result : -1;
}

// ===== 石子游戏 =====
bool stoneGame(const vector<int>& piles) {
    int n = piles.size();
    vector<vector<int>> dp(n, vector<int>(n, 0));

    // 初始化单堆石子的情况
    for (int i = 0; i < n; ++i) {
        dp[i][i] = piles[i];
    }

    // 枚举区间长度
    for (int length = 2; length <= n; ++length) {
        for (int i = 0; i <= n - length; ++i) {
            int j = i + length - 1;
            // 当前玩家可以选择左端或右端
            dp[i][j] = max(piles[i] - dp[i+1][j], piles[j] - dp[i][j-1]);
        }
    }

    return dp[0][n-1] > 0;
}

// ===== 数字 1 的个数 =====
int countDigitOne(int n) {
    long long count = 0;
    long long factor = 1;

    while (n / factor != 0) {
        long long lowerNum = n - (n / factor) * factor;
        long long currNum = (n / factor) % 10;
        long long higherNum = n / (factor * 10);

        if (currNum == 0) {
            count += higherNum * factor;
        } else if (currNum == 1) {
            count += higherNum * factor + lowerNum + 1;
        } else {
            count += (higherNum + 1) * factor;
        }
    }
}

```

```
factor *= 10;
}

return count;
}

// ===== 测试代码 =====
int main() {
    cout << "高级动态规划算法实现示例\n" << endl;
    cout << "===== 经典 DP 题目测试 =====" << endl;

    // 测试编辑距离
    cout << "编辑距离测试: " << editDistance("horse", "ros") << endl;

    // 测试最长递增子序列
    vector<int> nums = {10, 9, 2, 5, 3, 7, 101, 18};
    cout << "最长递增子序列测试: " << lengthOfLIS(nums) << endl;

    // 测试零钱兑换
    vector<int> coins = {1, 2, 5};
    cout << "零钱兑换测试: " << coinChange(coins, 11) << endl;

    // 测试石子游戏
    vector<int> piles = {5, 3, 4, 5};
    cout << "石子游戏测试: " << (stoneGame(piles) ? "true" : "false") << endl;

    // 测试数字 1 的个数
    cout << "数字 1 的个数测试: " << countDigitOne(13) << endl;

    // 测试矩阵链乘法
    vector<int> matrixDims = {30, 35, 15, 5, 10, 20, 25};
    cout << "矩阵链乘法最小标量乘法次数: " << matrixChainOrder(matrixDims) << endl;

    // 测试旅行商问题 (小规模)
    int tspSize = 4;
    vector<vector<int>> tspGraph(tspSize, vector<int>(tspSize, INT_MAX));
    tspGraph[0][1] = 10;
    tspGraph[0][2] = 15;
    tspGraph[0][3] = 20;
    tspGraph[1][0] = 10;
    tspGraph[1][2] = 35;
    tspGraph[1][3] = 25;
```

```

tspGraph[2][0] = 15;
tspGraph[2][1] = 35;
tspGraph[2][3] = 30;
tspGraph[3][0] = 20;
tspGraph[3][1] = 25;
tspGraph[3][2] = 30;
cout << "旅行商问题最短路径: " << tsp(tspGraph) << endl;

cout << "\n===== 高级优化体系测试 =====" << endl;

// 测试 SMAWK 算法
vector<vector<int>> mongeMatrix = {
    {10, 17, 13, 28, 23},
    {17, 22, 16, 29, 23},
    {24, 28, 22, 34, 24},
    {11, 13, 6, 17, 7},
    {17, 18, 14, 24, 18}
};
vector<int> smawkResult = SMAWK::solve(mongeMatrix);
cout << "SMAWK 算法每行最小值的列索引: ";
for (int idx : smawkResult) {
    cout << idx << " ";
}
cout << endl;

// 测试 Aliens Trick
vector<int> splitArray = {7, 2, 5, 10, 8};
int k = 2;
cout << "Aliens Trick 将数组分成" << k << "部分的最小成本: "
    << AliensTrick::splitArrayK(splitArray, k) << endl;

// 测试分层图最短路径
vector<vector<int>> flights = {
    {0, 1, 100},
    {1, 2, 100},
    {0, 2, 500}
};
int cities = 3;
int src = 0, dst = 2, maxStops = 1;
cout << "分层图 Dijkstra 找到的最便宜航班: "
    << LayeredGraphShortestPath::findCheapestPrice(cities, flights, src, dst, maxStops) <<
endl;

```

```

// 测试期望 DP (新 21 点)
int N = 21, K = 17, W = 10;
cout << "新 21 点游戏获胜概率: " << ExpectationDPWithGaussian::new21Game(N, K, W) << endl;

// 测试插头 DP (网格中的简单回路)
vector<vector<int>> grid = {
    {1, 1, 1},
    {1, 1, 1},
    {1, 1, 1}
};
cout << "3x3 网格中哈密顿回路数量: " << PlugDP::countGridCycles(grid) << endl;

// 测试树上背包优化
int treeNodes = 5;
int treeRoot = 1;
int capacity = 4;
vector<vector<int>> tree(treeNodes + 1);
tree[1].push_back(2);
tree[1].push_back(3);
tree[2].push_back(4);
tree[2].push_back(5);
tree[2].push_back(1);
tree[3].push_back(1);
tree[4].push_back(2);
tree[5].push_back(2);
vector<int> weights = {0, 1, 2, 1, 2, 1}; // 0 号元素不用
vector<int> values = {0, 10, 20, 5, 10, 3};
cout << "树上背包最大价值: "
     << TreeKnapsackOptimized::maxTreeValue(treeNodes, treeRoot, capacity, tree, weights,
values) << endl;

cout << "\n所有算法测试完成! " << endl;
return 0;
}
=====

文件: DPFusion.java
=====

// -*- coding: utf-8 -*-
/**
 * DP 融合场景: DP+数论、DP+字符串、DP+计算几何
 *

```

```

=====

文件: DPFusion.java
=====

// -*- coding: utf-8 -*-
/**
 * DP 融合场景: DP+数论、DP+字符串、DP+计算几何
 *

```

- * 问题描述:
- * 动态规划(DP)可以与其他领域的算法和数据结构结合, 形成强大的问题解决方法。
- * 本文件实现了三种主要的融合场景:
- * 1. DP+数论 (模意义下的动态规划)
- * 2. DP+字符串 (基于后缀自动机的计数)
- * 3. DP+计算几何 (凸包上的动态规划)
- *
- * 时间复杂度:
- * - DP+数论: 根据具体问题而定, 通常为 $O(n^2)$ 或 $O(n^3)$
- * - DP+字符串: $O(n)$ 或 $O(n^2)$
- * - DP+计算几何: $O(n^2)$ 或 $O(n \log n)$
- *
- * 空间复杂度: $O(n^2)$
- *
- * 相关题目:
- * 1. LeetCode 518. 零钱兑换 II (模意义)
- * 2. LeetCode 682. 棒球比赛 (字符串 DP)
- * 3. LeetCode 873. 最长的斐波那契子序列的长度 (序列 DP)
- */

```
import java.util.*;
import java.util.function.DoubleBinaryOperator;

public class DPFusion {

    // ===== DP+数论 (模意义) =====

    /**
     * LeetCode 518. 零钱兑换 II (模意义下的变种)
     * 题目链接: https://leetcode-cn.com/problems/coin-change-2/
     *
     * 问题描述:
     * 给定不同面额的硬币和一个总金额。计算可以凑成总金额的硬币组合数。
     * 假设每一种面额的硬币有无限个。要求结果对给定的模数取余。
     *
     * 解题思路:
     * 使用动态规划, 定义 dp[i] 表示凑成金额 i 的组合数。
     * 状态转移方程: dp[i] = (dp[i] + dp[i-coin]) % mod, 其中 coin 是硬币面额。
     *
     * @param amount 总金额
     * @param coins 硬币面额数组
     * @param mod 模数
     * @return 可以凑成总金额的硬币组合数对 mod 取余的结果
    }
```

```

*/
public static int coinChangeMod(int amount, int[] coins, int mod) {
    long[] dp = new long[amount + 1];
    dp[0] = 1; // 溉成金额 0 的方式只有 1 种 (不选任何硬币)

    // 遍历每种硬币
    for (int coin : coins) {
        // 遍历金额, 从 coin 开始
        for (int i = coin; i <= amount; ++i) {
            dp[i] = (dp[i] + dp[i - coin]) % mod;
        }
    }

// ===== 高级优化体系: SMAWK 算法 =====
/***
 * SMAWK 算法实现
 *
 * 问题描述:
 * 在 Monge 矩阵中快速找出每行的最小值的列索引
 * Monge 矩阵满足性质: 对于所有  $i \leq k, j \leq l$ , 有  $A[i][j] + A[k][l] \leq A[i][l] + A[k][j]$ 
 *
 * 解题思路:
 * 1. 使用递归分治的方法减少行数 (通过 row reduction)
 * 2. 递归求解缩小后的矩阵
 * 3. 利用 Monge 性质在剩余列中验证找到最小值
 *
 * 应用题目:
 * - POJ 3709 K-Anonymous Sequence
 * - Codeforces 868F Yet Another Minimization Problem
 *
 * 时间复杂度:  $O(m + n)$ , 其中 m 是行数, n 是列数
 * 空间复杂度:  $O(m + n)$ 
*/
public static class SMAWK {
    /**
     * 求解 Monge 矩阵每行的最小值的列索引
     *
     * @param matrix Monge 矩阵
     * @return 每行最小值的列索引数组
     */
    public static int[] solve(int[][] matrix) {
        if (matrix == null || matrix.length == 0) {
            return new int[0];
        }
}

```

```

int m = matrix.length;
int n = matrix[0].length;
int[] result = new int[m];

// 调用递归函数求解
solveRecursive(matrix, 0, m - 1, 0, n - 1, result);

return result;
}

/***
 * 递归求解 SMAWK 问题
 *
 * @param matrix 输入矩阵
 * @param rowStart 行起始索引
 * @param rowEnd 行结束索引
 * @param colStart 列起始索引
 * @param colEnd 列结束索引
 * @param result 结果数组，存储每行最小值的列索引
 */
private static void solveRecursive(int[][] matrix, int rowStart, int rowEnd, int
colStart, int colEnd, int[] result) {
    int rowCount = rowEnd - rowStart + 1;
    int colCount = colEnd - colStart + 1;

    // 基础情况：只有一行
    if (rowCount == 1) {
        int minCol = colStart;
        int minValue = matrix[rowStart][colStart];
        for (int j = colStart + 1; j <= colEnd; j++) {
            if (matrix[rowStart][j] < minValue) {
                minValue = matrix[rowStart][j];
                minCol = j;
            }
        }
        result[rowStart] = minCol;
        return;
    }

    // 对奇数行进行 Row Reduction
    int[] reducedRows = new int[(rowCount + 1) / 2];
    for (int i = 0; i < reducedRows.length; i++) {
        reducedRows[i] = rowStart + 2 * i;
    }
}

```

```

    }

    // 递归求解缩小后的问题（行减半）
    solveRecursive(matrix, rowStart, rowEnd, colStart, colEnd, result);

    // 处理偶数行，利用已求得的奇数行的最优列进行验证
    int prevOpt = colStart;
    for (int i = rowStart; i <= rowEnd; i++) {
        if (i % 2 == 0) continue; // 跳过奇数行，已经处理过

        int startCol = (i > rowStart) ? result[i - 1] : colStart;
        int endCol = (i < rowEnd) ? result[i + 1] : colEnd;

        int minCol = startCol;
        int minVal = matrix[i][startCol];

        for (int j = startCol + 1; j <= endCol; j++) {
            if (matrix[i][j] < minVal) {
                minVal = matrix[i][j];
                minCol = j;
            }
        }

        result[i] = minCol;
    }
}

// ===== 高级优化体系: Aliens Trick =====
/***
 * Aliens Trick (WQS 二分) 实现
 *
 * 问题描述:
 * 解决需要恰好分成 k 个部分的优化问题，通过二分查找约束参数 λ
 *
 * 解题思路:
 * 1. 二分查找 λ 值
 * 2. 对每个 λ，求解不加 k 约束的问题，但在成本函数中加入 λ 的惩罚项
 * 3. 根据结果中的分割次数调整二分方向
 *
 * 应用题目:
 * - LeetCode 410. Split Array Largest Sum
 * - Codeforces 868F Yet Another Minimization Problem

```

```

*
* 时间复杂度: O(log(maxCost) * n^2)，其中 maxCost 是可能的最大成本
* 空间复杂度: O(n)
*/
public static class AliensTrick {
    /**
     * 将数组分成 k 个连续子数组，使各子数组和的最大值最小
     * 相当于 LeetCode 410 的解法
     *
     * @param nums 输入数组
     * @param k 分成的部分数
     * @return 最小的最大子数组和
     */
    public static int splitArrayK(int[] nums, int k) {
        int n = nums.length;
        // 计算前缀和
        int[] prefixSum = new int[n + 1];
        for (int i = 0; i < n; i++) {
            prefixSum[i + 1] = prefixSum[i] + nums[i];
        }

        // 二分查找的范围
        int left = 0, right = prefixSum[n];
        for (int num : nums) {
            left = Math.max(left, num);
        }

        int result = right;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            int count = 1; // 至少一个分割
            int currentSum = 0;

            for (int num : nums) {
                if (currentSum + num > mid) {
                    count++;
                    currentSum = num;
                } else {
                    currentSum += num;
                }
            }

            if (count <= k) {

```

```

        result = mid;
        right = mid - 1;
    } else {
        left = mid + 1;
    }
}

return result;
}

/***
 * 使用 WQS 二分的更通用实现
 *
 * @param nums 输入数组
 * @param k 分成的部分数
 * @return 最小成本
 */
public static long splitArrayWithWQS(int[] nums, int k) {
    int n = nums.length;
    // 计算前缀和
    long[] prefixSum = new long[n + 1];
    for (int i = 0; i < n; i++) {
        prefixSum[i + 1] = prefixSum[i] + nums[i];
    }

    // 二分查找的范围
    long left = 0, right = prefixSum[n] * n;

    // WQS 二分查找 λ
    while (left < right) {
        long mid = left + (right - left) / 2;
        Pair<Long, Integer> result = computeDP(prefixSum, mid);

        if (result.second <= k) {
            right = mid;
        } else {
            left = mid + 1;
        }
    }

    // 计算最终结果
    Pair<Long, Integer> finalResult = computeDP(prefixSum, left);
    // 调整结果，因为 WQS 二分在计算时加入了 λ 的惩罚
}

```

```

        return finalResult.first - left * k;
    }

    /**
     * 计算 DP，其中成本函数加入了  $\lambda$  的惩罚
     *
     * @param prefixSum 前缀和数组
     * @param lambda 惩罚参数
     * @return 一个 Pair，第一个元素是总成本，第二个是分割次数
     */
    private static Pair<Long, Integer> computeDP(long[] prefixSum, long lambda) {
        int n = prefixSum.length - 1;
        long[] dp = new long[n + 1];
        int[] cnt = new int[n + 1];

        for (int i = 1; i <= n; i++) {
            dp[i] = Long.MAX_VALUE;
            for (int j = 0; j < i; j++) {
                long cost = (prefixSum[i] - prefixSum[j]) + lambda;
                if (dp[j] + cost < dp[i]) {
                    dp[i] = dp[j] + cost;
                    cnt[i] = cnt[j] + 1;
                }
            }
        }

        return new Pair<>(dp[n], cnt[n]);
    }

    /**
     * 简单的 Pair 类，用于返回两个值
     */
    private static class Pair<T1, T2> {
        T1 first;
        T2 second;

        public Pair(T1 first, T2 second) {
            this.first = first;
            this.second = second;
        }
    }
}

```

```

// ===== 图上 DP 转最短路: 分层图最短路径 =====
/***
 * 分层图最短路径实现
 *
 * 问题描述:
 * 在带有状态约束的图中寻找最短路径, 使用分层图建模不同状态
 *
 * 解题思路:
 * 1. 将每个节点按不同状态(如剩余步数、是否使用了某些能力等)分层
 * 2. 构建分层图, 每层内和层间建立相应的边
 * 3. 使用 Dijkstra 算法在分层图中寻找最短路径
 *
 * 应用题目:
 * - LeetCode 787. Cheapest Flights Within K Stops
 * - POJ 3255 Roadblocks
 *
 * 时间复杂度: O((n*K + m) log(n*K)), 其中 K 是分层数, m 是边数
 * 空间复杂度: O(n*K + m)
 */
public static class LayeredGraphShortestPath {
    /**
     * 找到最多经过 K 个中转站的最便宜航班
     * 相当于 LeetCode 787 的解法
     *
     * @param n 城市数量
     * @param flights 航班信息, 格式为[from, to, price]
     * @param src 出发城市
     * @param dst 目的城市
     * @param maxStops 最大中转次数
     * @return 最便宜的机票价格, 如果不存在则返回-1
     */
    public static int findCheapestPrice(int n, int[][] flights, int src, int dst, int maxStops) {
        // 构建图的邻接表表示
        List<List<int[]>> graph = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            graph.add(new ArrayList<>());
        }
        for (int[] flight : flights) {
            graph.get(flight[0]).add(new int[]{flight[1], flight[2]});
        }

        // 使用优先队列进行 Dijkstra 算法

```

```

PriorityQueue<int[]> pq = new PriorityQueue<>((a, b) -> a[0] - b[0]);
pq.offer(new int[]{0, src, 0}); // [价格, 当前城市, 已停留次数]

// 记录到达每个城市的每个停留次数的最小价格
int[][] prices = new int[n][maxStops + 2];
for (int[] row : prices) {
    Arrays.fill(row, Integer.MAX_VALUE);
}
prices[src][0] = 0;

while (!pq.isEmpty()) {
    int[] current = pq.poll();
    int price = current[0];
    int city = current[1];
    int stops = current[2];

    // 如果到达目的地, 直接返回价格
    if (city == dst) {
        return price;
    }

    // 如果已经超过最大中转次数, 跳过
    if (stops > maxStops) {
        continue;
    }

    // 如果当前价格比已知的更贵, 跳过
    if (price > prices[city][stops]) {
        continue;
    }

    // 遍历所有可能的下一个城市
    for (int[] next : graph.get(city)) {
        int nextCity = next[0];
        int nextPrice = price + next[1];
        int nextStops = stops + 1;

        // 如果找到更便宜的路径, 更新并加入优先队列
        if (nextPrice < prices[nextCity][nextStops]) {
            prices[nextCity][nextStops] = nextPrice;
            pq.offer(new int[]{nextPrice, nextCity, nextStops});
        }
    }
}

```

```

    }

    // 检查所有可能的停留次数中到达 dst 的最小价格
    int result = Integer.MAX_VALUE;
    for (int i = 0; i <= maxStops + 1; i++) {
        result = Math.min(result, prices[dst][i]);
    }

    return result == Integer.MAX_VALUE ? -1 : result;
}

/**
 * 求次短路径
 * 相当于 POJ 3255 Roadblocks 的解法
 *
 * @param n 节点数量
 * @param edges 边的信息，格式为[from, to, weight]
 * @param start 起始节点
 * @param end 目标节点
 * @return 次短路径的长度
 */
public static int findSecondShortestPath(int n, int[][] edges, int start, int end) {
    // 构建图的邻接表表示
    List<List<int[]>> graph = new ArrayList<>();
    for (int i = 0; i < n; i++) {
        graph.add(new ArrayList<>());
    }
    for (int[] edge : edges) {
        graph.get(edge[0]).add(new int[] {edge[1], edge[2]});
        graph.get(edge[1]).add(new int[] {edge[0], edge[2]}); // 无向图
    }

    // 记录每个节点的最短和次短距离
    int[][] dist = new int[n][2];
    for (int i = 0; i < n; i++) {
        Arrays.fill(dist[i], Integer.MAX_VALUE);
    }
    dist[start][0] = 0;

    // 使用优先队列进行 Dijkstra 算法
    PriorityQueue<int[]> pq = new PriorityQueue<>((a, b) -> a[0] - b[0]);
    pq.offer(new int[] {0, start, 0}); // [距离, 当前节点, 标记是最短(0)还是次短(1)]
}

```

```

while (!pq.isEmpty()) {
    int[] current = pq.poll();
    int d = current[0];
    int node = current[1];
    int type = current[2];

    // 如果当前距离比已知的更差，跳过
    if (d > dist[node][type]) {
        continue;
    }

    // 遍历所有邻居
    for (int[] neighbor : graph.get(node)) {
        int nextNode = neighbor[0];
        int weight = neighbor[1];
        int newDist = d + weight;

        // 如果可以更新最短距离
        if (newDist < dist[nextNode][0]) {
            dist[nextNode][1] = dist[nextNode][0]; // 将原最短距离降级为次短
            dist[nextNode][0] = newDist; // 更新最短距离
            pq.offer(new int[] {newDist, nextNode, 0});
            pq.offer(new int[] {dist[nextNode][1], nextNode, 1});
        }
        // 如果可以更新次短距离，但不与最短距离相同
        else if (newDist > dist[nextNode][0] && newDist < dist[nextNode][1]) {
            dist[nextNode][1] = newDist;
            pq.offer(new int[] {newDist, nextNode, 1});
        }
    }

    return dist[end][1];
}

// ====== 冷门模型：期望 DP 与高斯消元 ======
/** 
 * 期望 DP 与高斯消元实现
 *
 * 问题描述：
 * 解决包含环的期望 DP 问题，通过建立方程组并使用高斯消元求解
 */

```

* 解题思路:

- * 1. 建立状态转移方程
- * 2. 对于包含环的情况，将状态转移方程转换为线性方程组
- * 3. 使用高斯消元法求解线性方程组

*

* 应用题目:

* - LeetCode 837. New 21 Game

* - POJ 2096 Collecting Bugs

*

* 时间复杂度:

* - New 21 Game: $O(N + K + W)$

* - 高斯消元部分: $O(n^3)$, 其中 n 是状态数

* 空间复杂度: $O(N + K + W)$ 或 $O(n^2)$

*/

```
public static class ExpectationDPWithGaussian {  
    /**  
     * 新 21 点游戏获胜概率  
     * 相当于 LeetCode 837 的解法  
     *  
     * @param N 目标值  
     * @param K 当前点数小于 K 时继续抽牌  
     * @param W 每次抽牌的最大值  
     * @return 获胜的概率  
    */  
    public static double new21Game(int N, int K, int W) {  
        // 边界条件处理  
        if (K == 0 || N >= K + W) {  
            return 1.0;  
        }  
  
        // dp[x]表示当前点数为 x 时，最终获胜的概率  
        double[] dp = new double[K + W + 1];  
  
        // 初始化边界条件  
        for (int i = K; i <= N; i++) {  
            dp[i] = 1.0;  
        }  
  
        // 计算前缀和，优化计算  
        double sum = N - K + 1.0; // dp[K] 到 dp[N] 的和  
  
        // 从后往前计算 dp 值  
        for (int i = K - 1; i >= 0; i--) {
```

```

        dp[i] = sum / W;
        sum = sum + dp[i] - dp[i + W];
    }

    return dp[0];
}

/***
 * 高斯消元法求解线性方程组 Ax = b
 *
 * @param A 系数矩阵
 * @param b 常数项向量
 * @return 解向量 x
 */
public static double[] gaussianElimination(double[][] A, double[] b) {
    int n = A.length;
    double[][] augMatrix = new double[n][n + 1];

    // 构造增广矩阵
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            augMatrix[i][j] = A[i][j];
        }
        augMatrix[i][n] = b[i];
    }

    // 高斯消元
    for (int i = 0; i < n; i++) {
        // 寻找主元
        int maxRow = i;
        for (int j = i + 1; j < n; j++) {
            if (Math.abs(augMatrix[j][i]) > Math.abs(augMatrix[maxRow][i])) {
                maxRow = j;
            }
        }
    }

    // 交换行
    double[] temp = augMatrix[i];
    augMatrix[i] = augMatrix[maxRow];
    augMatrix[maxRow] = temp;

    // 归一化主元行
    double pivot = augMatrix[i][i];

```

```

        for (int j = i; j <= n; j++) {
            augMatrix[i][j] /= pivot;
        }

        // 消去其他行
        for (int j = 0; j < n; j++) {
            if (j != i && Math.abs(augMatrix[j][i]) > 1e-9) {
                double factor = augMatrix[j][i];
                for (int k = i; k <= n; k++) {
                    augMatrix[j][k] -= factor * augMatrix[i][k];
                }
            }
        }
    }

    // 提取解
    double[] x = new double[n];
    for (int i = 0; i < n; i++) {
        x[i] = augMatrix[i][n];
    }

    return x;
}

/**
 * 求解环形房间期望停留时间问题
 *
 * @param n 房间数量
 * @param transitions 转移概率矩阵
 * @param target 目标房间
 * @return 从每个房间到达目标房间的期望时间
 */
public static double[] expectedTimeInRooms(int n, double[][] transitions, int target) {
    // 构建线性方程组
    double[][] A = new double[n][n];
    double[] b = new double[n];

    for (int i = 0; i < n; i++) {
        if (i == target) {
            // 目标房间的期望时间为0
            A[i][i] = 1.0;
            b[i] = 0.0;
        } else {

```

```

        A[i][i] = 1.0;
        for (int j = 0; j < n; j++) {
            if (i != j) {
                A[i][j] -= transitions[i][j];
            }
        }
        b[i] = 1.0; // 每一步需要 1 单位时间
    }

    return gaussianElimination(A, b);
}

}

// ====== 冷门模型: 插头 DP ======
/***
 * 插头 DP 实现 (轮廓线 DP)
 *
 * 问题描述:
 * 求解网格图中的回路计数、路径覆盖等问题
 *
 * 解题思路:
 * 1. 使用状态压缩表示当前轮廓线的插头状态
 * 2. 逐格处理, 根据当前位置和状态转移到下一个状态
 * 3. 使用哈希表或滚动数组优化空间复杂度
 *
 * 应用题目:
 * - HDU 1693 Eat the Trees (网格回路计数)
 * - HDU 3377 Plan (单回路覆盖)
 *
 * 时间复杂度: O(n*m*2^m), 其中 n 和 m 是网格的维度
 * 空间复杂度: O(2^m), 使用滚动数组优化
 */
public static class PlugDP {

    /**
     * 计算网格中简单回路的数量
     *
     * @param grid 网格, 1 表示可走, 0 表示障碍
     * @return 回路数量
     */
    public static int countGridCycles(int[][] grid) {
        if (grid == null || grid.length == 0 || grid[0].length == 0) {
            return 0;
        }
}

```

```

}

int n = grid.length;
int m = grid[0].length;

// 预处理：确保至少有两个可走的格子
int total = 0;
for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
        total += grid[i][j];
    }
}
if (total < 2) {
    return 0;
}

// 插头 DP，使用哈希表记录状态
Map<Long, Integer> dp = new HashMap<>();
dp.put(0L, 1); // 初始状态

int firstRow = 0, firstCol = 0;
// 找到第一个可走的格子作为起点
outerLoop:
for (firstRow = 0; firstRow < n; firstRow++) {
    for (firstCol = 0; firstCol < m; firstCol++) {
        if (grid[firstRow][firstCol] == 1) {
            break outerLoop;
        }
    }
}

// 逐格处理
for (int i = 0; i < n; i++) {
    // 每处理完一行，状态需要左移一位
    Map<Long, Integer> newDp = new HashMap<>();
    for (long state : dp.keySet()) {
        // 左移一位，但最右边补 0
        long newState = state << 2;
        newDp.put(newState, dp.get(state));
    }
    dp = newDp;

    for (int j = 0; j < m; j++) {

```

```

if (grid[i][j] == 0) {
    // 障碍物，跳过
    continue;
}

newDp = new HashMap<>();
for (Map.Entry<Long, Integer> entry : dp.entrySet()) {
    long state = entry.getKey();
    int count = entry.getValue();

    // 当前格子左侧和上方的插头状态
    int left = (int)((state >> (2 * j)) & 3);
    int up = (int)((state >> (2 * (j + 1))) & 3);

    // 跳过已经处理过的情况
    if ((i == firstRow && j == firstCol) && (left != 0 || up != 0)) {
        continue;
    }

    // 处理不同的插头组合情况
    if (left == 0 && up == 0) {
        // 新建两个插头，仅在当前不是最后一个格子时
        if (i < n - 1 && j < m - 1 && grid[i+1][j] == 1 && grid[i][j+1] == 1)
    {
        long newState = state | (1L << (2 * j)) | (2L << (2 * (j + 1)));
        newDp.put(newState, newDp.getOrDefault(newState, 0) + count);
    }
    } else if (left == 0 || up == 0) {
        // 一个插头，延伸这个插头
        int plug = Math.max(left, up);
        // 向右延伸
        if (j < m - 1 && grid[i][j+1] == 1) {
            long newState = state & ~(3L << (2 * (j + 1))) & ~(3L << (2 *
j));
            newState |= (long)plug << (2 * (j + 1));
            newDp.put(newState, newDp.getOrDefault(newState, 0) + count);
        }
        // 向下延伸
        if (i < n - 1 && grid[i+1][j] == 1) {
            long newState = state & ~(3L << (2 * (j + 1))) & ~(3L << (2 *
j));
            newState |= (long)plug << (2 * j);
            newDp.put(newState, newDp.getOrDefault(newState, 0) + count);
        }
    }
}

```

```

        }

    } else if (left == up) {
        // 两个相同的插头，形成环，仅在最后一个格子时
        if (i == n - 1 && j == m - 1 && (state & ~(3L << (2 * j)) & ~(3L << (2 * (j + 1)))) == 0) {
            newDp.put(0L, newDp.getOrDefault(0L, 0) + count);
        }
    } else {
        // 两个不同的插头，连接它们
        // 找到另一个相同的插头并替换
        long newState = state & ~(3L << (2 * j)) & ~(3L << (2 * (j + 1)));
        for (int k = 0; k < m + 1; k++) {
            if (k == j || k == j + 1) continue;
            int curr = (int)((newState >> (2 * k)) & 3);
            if (curr == up) {
                newState &= ~(3L << (2 * k));
                newState |= (long)left << (2 * k);
                break;
            }
        }
        newState = newDp.getOrDefault(newState, 0) + count;
    }
    dp = newDp;
}

}

// 最终状态应该是 0，表示闭合回路
return dp.getOrDefault(0L, 0) / 2; // 因为每个环会被计算两次（顺时针和逆时针）
}

}

// ===== 冷门模型：树上背包优化 =====
/***
 * 树上背包优化实现（使用 small-to-large 合并）
 *
 * 问题描述：
 * 在树上选择一些节点，满足背包容量限制，最大化价值
 *
 * 解题思路：
 * 1. 使用后序遍历处理子树
 * 2. 应用 small-to-large 合并优化，将小的子树合并到较大的子树中
 * 3. 减少重复计算，提高效率

```

```

*
* 应用题目：
* - POJ 1947 Rebuilding Roads
* - HDU 3516 Tree Construction
*
* 时间复杂度: O(n*c)，其中 c 是背包容量
* 空间复杂度: O(n*c)
*/
public static class TreeKnapsackOptimized {
    /**
     * 计算树上背包的最大价值
     *
     * @param n 节点数量
     * @param root 根节点
     * @param capacity 背包容量
     * @param tree 树的邻接表表示
     * @param weights 节点的重量数组
     * @param values 节点的价值数组
     * @return 最大价值
     */
    public static int maxTreeValue(int n, int root, int capacity, List<List<Integer>> tree,
int[] weights, int[] values) {
        // 使用邻接表构建树，并记录父节点以避免循环
        int[][] dp = new int[n + 1][capacity + 1];
        boolean[] visited = new boolean[n + 1];

        // 后序遍历处理子树
        dfs(root, -1, capacity, tree, weights, values, dp, visited);

        return dp[root][capacity];
    }

    /**
     * 深度优先搜索处理子树
     *
     * @param node 当前节点
     * @param parent 父节点
     * @param capacity 背包容量
     * @param tree 树的邻接表
     * @param weights 重量数组
     * @param values 价值数组
     * @param dp dp 数组
     * @param visited 访问标记数组
     */
    private void dfs(int node, int parent, int capacity, List<List<Integer>> tree,
int[] weights, int[] values, int[][] dp, boolean[] visited) {
        if (visited[node]) {
            return;
        }
        visited[node] = true;

        for (int neighbor : tree.get(node)) {
            if (neighbor == parent) {
                continue;
            }
            dfs(neighbor, node, capacity, tree, weights, values, dp, visited);
        }

        int maxVal = 0;
        for (int i = 0; i <= capacity; i++) {
            maxVal = Math.max(maxVal, dp[node][i]);
        }
        dp[node][capacity] = maxVal;
    }
}

```

```

*/
private static void dfs(int node, int parent, int capacity, List<List<Integer>> tree,
int[] weights, int[] values, int[][] dp, boolean[] visited) {
    visited[node] = true;

    // 初始化: 只选当前节点
    int nodeWeight = weights[node];
    int nodeValue = values[node];
    for (int j = 0; j <= capacity; j++) {
        if (j >= nodeWeight) {
            dp[node][j] = nodeValue;
        }
    }

    // 遍历所有子节点
    for (int child : tree.get(node)) {
        if (child == parent || visited[child]) {
            continue;
        }

        // 先处理子节点
        dfs(child, node, capacity, tree, weights, values, dp, visited);

        // 使用 small-to-large 合并优化, 从后往前遍历容量避免重复计算
        for (int j = capacity; j >= nodeWeight; j--) {
            for (int k = 0; k <= j - nodeWeight; k++) {
                dp[node][j] = Math.max(dp[node][j], dp[node][j - k] + dp[child][k]);
            }
        }
    }
}

/**
 * 带有 size 优化的树上背包实现
 *
 * @param n 节点数量
 * @param root 根节点
 * @param capacity 背包容量
 * @param tree 树的邻接表
 * @param weights 重量数组
 * @param values 价值数组
 * @return 最大价值
*/

```

```

public static int maxTreeValueWithSizeOpt(int n, int root, int capacity,
List<List<Integer>> tree, int[] weights, int[] values) {
    int[][] dp = new int[n + 1][capacity + 1];
    int[] size = new int[n + 1];

    // 计算每个子树的大小
    computeSize(root, -1, tree, size);

    // 后序遍历处理
    dfsWithSize(root, -1, capacity, tree, weights, values, dp, size);

    return dp[root][capacity];
}

/**
 * 计算子树大小
 */
private static void computeSize(int node, int parent, List<List<Integer>> tree, int[]
size) {
    size[node] = 1;
    for (int child : tree.get(node)) {
        if (child != parent) {
            computeSize(child, node, tree, size);
            size[node] += size[child];
        }
    }
}

/**
 * 使用 size 进行优化的深度优先搜索
 */
private static void dfsWithSize(int node, int parent, int capacity, List<List<Integer>>
tree, int[] weights, int[] values, int[][] dp, int[] size) {
    // 初始化
    int nodeWeight = weights[node];
    int nodeValue = values[node];
    dp[node][0] = 0;
    for (int j = 1; j <= capacity; j++) {
        dp[node][j] = j >= nodeWeight ? nodeValue : 0;
    }

    // 找到最大的子树
    int maxChild = -1;

```

```

int maxSize = 0;
for (int child : tree.get(node)) {
    if (child != parent && size[child] > maxSize) {
        maxSize = size[child];
        maxChild = child;
    }
}

// 优先处理最大的子树，以便应用 small-to-large 优化
if (maxChild != -1) {
    dfsWithSize(maxChild, node, capacity, tree, weights, values, dp, size);

    // 合并其他子树
    for (int child : tree.get(node)) {
        if (child == parent || child == maxChild) {
            continue;
        }
        dfsWithSize(child, node, capacity, tree, weights, values, dp, size);
    }

    // small-to-large 合并：将较小的子树合并到较大的子树中
    for (int j = capacity; j >= nodeWeight; j--) {
        for (int k = 0; k <= j - nodeWeight; k++) {
            dp[node][j] = Math.max(dp[node][j], dp[node][j - k] + dp[child][k]);
        }
    }
}
}

return (int) dp[amount];
}

/***
 * 矩阵乘法（模意义下）
 *
 * @param a 矩阵 a
 * @param b 矩阵 b
 * @param mod 模数
 * @return 矩阵 a 和矩阵 b 的乘积对 mod 取余的结果
 */

```

```

public static long[][] matrixMultiply(long[][] a, long[][] b, int mod) {
    int n = a.length;
    int m = b[0].length;
    int k = b.length;
    long[][] result = new long[n][m];

    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < m; ++j) {
            for (int p = 0; p < k; ++p) {
                result[i][j] = (result[i][j] + a[i][p] * b[p][j]) % mod;
            }
        }
    }

    return result;
}

```

```

/**
 * 矩阵快速幂（模意义下）
 *
 * 问题描述：
 * 计算矩阵的幂，结果对 mod 取余。
 *
 * 解题思路：
 * 使用快速幂算法，将矩阵的乘法在模意义下进行。
 *
 * @param matrix 输入矩阵
 * @param power 幂次
 * @param mod 模数
 * @return 矩阵的幂对 mod 取余的结果
 */

```

```

public static long[][] matrixPowerMod(long[][] matrix, int power, int mod) {
    int n = matrix.length;
    // 初始化结果为单位矩阵
    long[][] result = new long[n][n];
    for (int i = 0; i < n; ++i) {
        result[i][i] = 1;
    }

    // 快速幂算法
    while (power > 0) {
        if (power % 2 == 1) {
            // 矩阵乘法

```

```

        result = matrixMultiply(result, matrix, mod);
    }
    // 矩阵自乘
    matrix = matrixMultiply(matrix, matrix, mod);
    power /= 2;
}

return result;
}

// ===== DP+字符串 (SAM 相关) =====

/***
 * LeetCode 516. 最长回文子序列
 * 题目链接: https://leetcode-cn.com/problems/longest-palindromic-subsequence/
 *
 * 问题描述:
 * 给定一个字符串 s，找到其中最长的回文子序列。可以假设 s 的最大长度为 1000。
 *
 * 解题思路:
 * 使用区间 DP，定义 dp[i][j] 表示字符串 s 在区间 [i, j] 内的最长回文子序列的长度。
 * 状态转移方程:
 * - 如果 s[i] == s[j]，则 dp[i][j] = dp[i+1][j-1] + 2
 * - 否则，dp[i][j] = max(dp[i+1][j], dp[i][j-1])
 *
 * @param s 输入字符串
 * @return 最长回文子序列的长度
 */
public static int longestPalindromicSubseq(String s) {
    int n = s.length();
    // dp[i][j] 表示字符串 s 在区间 [i, j] 内的最长回文子序列的长度
    int[][] dp = new int[n][n];

    // 初始化单个字符的情况
    for (int i = 0; i < n; ++i) {
        dp[i][i] = 1;
    }

    // 枚举区间长度
    for (int length = 2; length <= n; ++length) {
        // 枚举起点
        for (int i = 0; i <= n - length; ++i) {
            int j = i + length - 1;
            if (s.charAt(i) == s.charAt(j)) {
                dp[i][j] = dp[i+1][j-1] + 2;
            } else {
                dp[i][j] = Math.max(dp[i+1][j], dp[i][j-1]);
            }
        }
    }
}

```

```

        if (s.charAt(i) == s.charAt(j)) {
            dp[i][j] = dp[i+1][j-1] + 2;
        } else {
            dp[i][j] = Math.max(dp[i+1][j], dp[i][j-1]);
        }
    }

    return dp[0][n-1];
}

/***
 * 后缀自动机 (Suffix Automaton)
 *
 * 后缀自动机是一个可以表示字符串的所有子串的数据结构。
 * 它可以用于解决许多字符串问题，如子串匹配、最长重复子串等。
 */
public static class SuffixAutomaton {
    private static class State {
        int len; // 该状态能接受的最长字符串的长度
        int link; // 后缀链接
        Map<Character, Integer> next; // 转移函数
        int endposSize; // endpos 集合的大小

        public State() {
            this.len = 0;
            this.link = -1;
            this.next = new HashMap<>();
            this.endposSize = 0;
        }
    }

    private int size;
    private int last;
    private List<State> states;

    private void extend(char c) {
        int p = last;
        int curr = size;
        size++;
        states.add(new State());
        states.get(curr).len = states.get(p).len + 1;
    }
}

```

```

        while (p != -1 && !states.get(p).next.containsKey(c)) {
            states.get(p).next.put(c, curr);
            p = states.get(p).link;
        }

        if (p == -1) {
            states.get(curr).link = 0;
        } else {
            int q = states.get(p).next.get(c);
            if (states.get(p).len + 1 == states.get(q).len) {
                states.get(curr).link = q;
            } else {
                int clone = size;
                size++;
                states.add(new State());
                states.get(clone).len = states.get(p).len + 1;
                states.get(clone).next.putAll(states.get(q).next);
                states.get(clone).link = states.get(q).link;

                while (p != -1 && states.get(p).next.containsKey(c) &&
states.get(p).next.get(c) == q) {
                    states.get(p).next.put(c, clone);
                    p = states.get(p).link;
                }

                states.get(q).link = clone;
                states.get(curr).link = clone;
            }
        }

        last = curr;
    }

private void calcEndposSize() {
    // 按 len 排序
    Integer[] order = new Integer[size];
    for (int i = 0; i < size; ++i) {
        order[i] = i;
    }
    Arrays.sort(order, (a, b) -> Integer.compare(states.get(b).len, states.get(a).len));

    // 初始化为 1 (每个状态至少对应一个结束位置)
    for (int i = 1; i < size; ++i) {

```

```

        states.get(i).endposSize = 1;
    }

    // 从长到短更新
    for (int u : order) {
        if (states.get(u).link != -1) {
            states.get(states.get(u).link).endposSize += states.get(u).endposSize;
        }
    }
}

public SuffixAutomaton(String s) {
    size = 1;
    last = 0;
    states = new ArrayList<>();
    states.add(new State());

    // 构建后缀自动机
    for (char c : s.toCharArray()) {
        extend(c);
    }

    // 计算 endpos 集合的大小
    calcEndposSize();
}

/**
 * 计算不同子串的数量
 */
public int countSubstrings() {
    int count = 0;
    for (int i = 1; i < size; ++i) {
        count += states.get(i).len - states.get(states.get(i).link).len;
    }
    return count;
}

// ===== DP+计算几何（凸包相关） =====

/**
 * 点结构体
 */

```

```
public static class Point implements Comparable<Point> {
    double x, y;

    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    @Override
    public int compareTo(Point p) {
        if (this.x != p.x) {
            return Double.compare(this.x, p.x);
        }
        return Double.compare(this.y, p.y);
    }

    @Override
    public String toString() {
        return "(" + x + ", " + y + ")";
    }
}

/**
 * 计算叉积 (a - o) × (b - o)
 */
public static double cross(Point o, Point a, Point b) {
    return (a.x - o.x) * (b.y - o.y) - (a.y - o.y) * (b.x - o.x);
}

/**
 * 计算凸包 (Andrew 算法)
 *
 * 问题描述:
 * 给定平面上的点集, 找出所有在凸包上的点。
 *
 * 解题思路:
 * 1. 将点按 x 坐标排序, x 相同按 y 排序
 * 2. 构建下凸壳和上凸壳
 * 3. 合并上下凸壳
 *
 * @param points 点集
 * @return 凸包上的点集
 */
```

```

public static List<Point> convexHull(List<Point> points) {
    // 按 x 坐标排序, x 相同按 y 排序
    Collections.sort(points);
    int n = points.size();

    // 构建下凸壳
    List<Point> lower = new ArrayList<>();
    for (Point p : points) {
        while (lower.size() >= 2 && cross(lower.get(lower.size() - 2), lower.get(lower.size() - 1), p) <= 0) {
            lower.remove(lower.size() - 1);
        }
        lower.add(p);
    }

    // 构建上凸壳
    List<Point> upper = new ArrayList<>();
    for (int i = n - 1; i >= 0; --i) {
        Point p = points.get(i);
        while (upper.size() >= 2 && cross(upper.get(upper.size() - 2), upper.get(upper.size() - 1), p) <= 0) {
            upper.remove(upper.size() - 1);
        }
        upper.add(p);
    }

    // 合并上下凸壳, 去掉重复的端点
    List<Point> result = new ArrayList<>();
    for (int i = 0; i < lower.size() - 1; ++i) {
        result.add(lower.get(i));
    }
    for (int i = 0; i < upper.size() - 1; ++i) {
        result.add(upper.get(i));
    }
    return result;
}

/**
 * 凸包优化 DP
 *
 * 问题描述:
 * 当 DP 状态转移方程可以表示为  $dp[i] = \min\{dp[j] + a[i] * b[j]\} + c[i]$  的形式时,
 * 可以使用凸包优化将时间复杂度从  $O(n^2)$  降低到  $O(n)$  或  $O(n \log n)$ 。

```

```

*
* 解题思路:
* 对于每个 j, 维护一条直线  $y = b[j] * x + dp[j]$ , 然后对于每个 i, 查询  $x = a[i]$  时的最小值。
* 当  $b[j]$  单调递增且  $a[i]$  单调递增时, 可以使用单调队列优化。
*

* @param dp DP 数组
* @param a a 数组
* @param b b 数组
* @return 优化后的 DP 数组
*/
public static double[] convexHullTrick(double[] dp, double[] a, double[] b) {
    int n = dp.length;
    Deque<Integer> q = new LinkedList<>(); // 单调队列, 存储直线的索引

    // 计算两条直线 j1 和 j2 的交点 x 坐标
    DoubleBinaryOperator getIntersection = (j1, j2) -> {
        // 直线 j1:  $y = b[(int)j1] * x + dp[(int)j1]$ 
        // 直线 j2:  $y = b[(int)j2] * x + dp[(int)j2]$ 
        if (b[(int)j1] == b[(int)j2]) {
            return Double.MAX_VALUE;
        }
        return (dp[(int)j2] - dp[(int)j1]) / (b[(int)j1] - b[(int)j2]);
    };

    // 初始化队列, 加入第一个元素
    q.offerLast(0);

    // 对于每个 i, 找到最优的 j
    for (int i = 1; i < n; ++i) {
        // 当队列中至少有两个元素, 且第一个元素不如第二个元素优时, 弹出第一个元素
        while (q.size() >= 2) {
            int j1 = q.peekFirst();
            // 获取第二个元素
            q.pollFirst();
            int j2 = q.peekFirst();
            q.offerFirst(j1); // 放回第一个元素

            if (dp[j1] + a[i] * b[j1] >= dp[j2] + a[i] * b[j2]) {
                q.pollFirst();
            } else {
                break;
            }
        }
    }
}

```

```

// 使用队列中的第一个元素作为最优的 j
if (!q.isEmpty()) {
    int bestJ = q.peekFirst();
    dp[i] = Math.min(dp[i], dp[bestJ] + a[i] * b[bestJ]);
}

// 将当前 i 加入队列，维护队列的凸壳性质
while (q.size() >= 2) {
    int j2 = q.pollLast();
    int j1 = q.peekLast();
    double x1 = (dp[j2] - dp[j1]) / (b[j1] - b[j2]);
    double x2 = (dp[i] - dp[j2]) / (b[j2] - b[i]);
    if (x1 >= x2) {
        // 需要弹出 j2
        continue;
    } else {
        // 把 j2 放回去
        q.offerLast(j2);
        break;
    }
}
q.offerLast(i);

return dp;
}

// 测试代码
public static void main(String[] args) {
    // 测试 DP+数论
    int amount = 5;
    int[] coins = {1, 2, 5};
    int mod = (int)1e9 + 7;
    System.out.println("零钱兑换 II (模意义)：" + coinChangeMod(amount, coins, mod)); // 应该输出 4
}

```

```

// 测试矩阵快速幂
long[][] matrix = {{1, 1}, {1, 0}};
int power = 5;
System.out.println("矩阵快速幂结果:");
long[][] result = matrixPowerMod(matrix, power, mod);
for (long[] row : result) {

```

```

        for (long num : row) {
            System.out.print(num + " ");
        }
        System.out.println();
    }

// 测试 DP+字符串
String s = "bbbab";
System.out.println("最长回文子序列长度: " + longestPalindromicSubseq(s)); // 应该输出 4

// 测试后缀自动机
SuffixAutomaton sam = new SuffixAutomaton("banana");
System.out.println("不同子串数量: " + sam.countSubstrings()); // 应该输出 15

// 测试 DP+计算几何
List<Point> points = Arrays.asList(
    new Point(0, 0),
    new Point(1, 1),
    new Point(2, 0),
    new Point(1, -1)
);
List<Point> hull = convexHull(points);
System.out.println("凸包上的点: " + hull);

// 测试凸包优化 DP
double[] dp = {0, Double.MAX_VALUE, Double.MAX_VALUE, Double.MAX_VALUE,
Double.MAX_VALUE};
double[] a = {1, 2, 3, 4, 5};
double[] b = {1, 2, 3, 4, 5};
double[] optimizedDp = convexHullTrick(dp, a, b);
System.out.print("凸包优化 DP 结果: ");
for (double num : optimizedDp) {
    System.out.print(num + " ");
}
System.out.println();
}

// ====== 优化体系: Knuth 优化 ======
// Knuth 优化用于优化形如  $dp[i][j] = \min\{dp[i][k] + dp[k+1][j]\} + w(i, j)$  的 DP
// 当满足四边形不等式时, 最优转移点单调
// 四边形不等式:  $w(a, b) + w(c, d) \leq w(a, d) + w(c, b)$ , 其中  $a \leq c \leq b \leq d$ 
// 单调性:  $w(b, c) \leq w(a, d)$ , 其中  $a \leq b \leq c \leq d$ 

```

```

public static class KnuthOptimizationResult {
    int[][] dp;
    int[][] opt;

    public KnuthOptimizationResult(int[][] dp, int[][] opt) {
        this.dp = dp;
        this.opt = opt;
    }
}

public static KnuthOptimizationResult knuthOptimization(int n, BiFunction<Integer, Integer,
Integer> costFunc) {
    /*
     * Knuth 优化的 DP 算法
    */
}

```

问题描述:

解决区间 DP 问题，其中状态转移方程满足四边形不等式

解题思路:

1. 使用 Knuth 优化将时间复杂度从 $O(n^3)$ 降低到 $O(n^2)$
2. 维护最优转移点数组 $opt[i][j]$ ，表示计算 $dp[i][j]$ 时的最优 k 值
3. 根据 $opt[i][j-1] \leq opt[i][j] \leq opt[i+1][j]$ 的性质进行剪枝

参数:

n : 区间长度

$costFunc$: 计算区间 (i, j) 代价的函数

返回:

$KnuthOptimizationResult$: 包含 dp 数组和 opt 数组的结果类

时间复杂度: $O(n^2)$

空间复杂度: $O(n^2)$

*/

// 初始化 dp 和 opt 数组

```

int[][] dp = new int[n + 1][n + 1];
int[][] opt = new int[n + 1][n + 1];

```

// 初始化长度为 1 的区间

```

for (int i = 1; i <= n; ++i) {
    dp[i][i] = 0;
    opt[i][i] = i;
}

```

```

// 枚举区间长度
for (int length = 2; length <= n; ++length) {
    // 枚举起始点
    for (int i = 1; i + length - 1 <= n; ++i) {
        int j = i + length - 1;
        // 初始化为无穷大
        dp[i][j] = Integer.MAX_VALUE;
        // 根据 Knuth 优化的性质，最优 k 在 opt[i][j-1] 到 opt[i+1][j] 之间
        int upperK = (i + 1 <= j) ? opt[i + 1][j] : j - 1;
        for (int k = opt[i][j - 1]; k <= Math.min(upperK, j - 1); ++k) {
            if (dp[i][k] != Integer.MAX_VALUE && dp[k + 1][j] != Integer.MAX_VALUE) {
                Integer cost = costFunc.apply(i, j);
                if (cost != null && cost != Integer.MAX_VALUE) {
                    int current = dp[i][k] + dp[k + 1][j] + cost;
                    if (current < dp[i][j]) {
                        dp[i][j] = current;
                        opt[i][j] = k;
                    }
                }
            }
        }
    }
}

return new KnuthOptimizationResult(dp, opt);
}

// ===== 优化体系: Divide & Conquer Optimization =====

```

```

public static int[][] divideConquerOptimization(int n, int m, BiFunction<Integer, Integer, Integer> costFunc) {
    /*
     * Divide & Conquer Optimization (分治优化)
     */

```

问题描述:

解决形如 $dp[i][j] = \min\{dp[i-1][k] + cost(k, j)\}$, 其中 $k < j$
当转移满足决策单调性时使用

解题思路:

1. 利用决策单调性, 使用分治法优化 DP
2. 对于 $dp[i][j]$, 当 i 固定时, 最优转移点 k 随着 j 的增加而单调不减
3. 使用分治的方式计算每个区间的最优决策

参数:

n: 维度 1
m: 维度 2
costFunc: 计算 cost(k, j) 的函数

返回:

int[][]: dp 数组

时间复杂度: O(n*m log m)

空间复杂度: O(n*m)

*/

// 初始化 dp 数组

```
int[][] dp = new int[n + 1][m + 1];
for (int i = 0; i <= n; ++i) {
    Arrays.fill(dp[i], Integer.MAX_VALUE);
}
dp[0][0] = 0;
```

// 对每个 i 应用分治优化

```
for (int i = 1; i <= n; ++i) {
    solveDivideConquer(i, 1, m, 0, m, dp, costFunc);
}
```

return dp;

}

```
private static void solveDivideConquer(int i, int l, int r, int opt_l, int opt_r,
                                       int[][] dp, BiFunction<Integer, Integer, Integer>
costFunc) {
```

/*

计算 dp[i][l..r], 其中最优转移点在 opt_l..opt_r 之间

*/

```
if (l > r) {
    return;
}
```

```
int mid = (l + r) / 2;
int best_k = opt_l;
```

// 在 opt_l 到 min(mid-1, opt_r) 之间寻找最优 k

```
for (int k = opt_l; k <= Math.min(mid, opt_r); ++k) {
    if (dp[i - 1][k] != Integer.MAX_VALUE) {
```

```

        Integer cost = costFunc.apply(k, mid);
        if (cost != null && cost != Integer.MAX_VALUE) {
            int current = dp[i - 1][k] + cost;
            if (current < dp[i][mid]) {
                dp[i][mid] = current;
                best_k = k;
            }
        }
    }
}

// 递归处理左右子区间
solveDivideConquer(i, 1, mid - 1, opt_l, best_k, dp, costFunc);
solveDivideConquer(i, mid + 1, r, best_k, opt_r, dp, costFunc);
}

```

// ===== 优化体系: SMAWK 算法 (行最小查询) =====

```

public static int[] smawk(int[][] matrix) {
/*
SMAWK 算法用于在 Monge 矩阵中快速查找每行的最小值

```

问题描述:

给定一个 Monge 矩阵, 快速找到每行的最小值位置

解题思路:

1. Monge 矩阵满足性质: $\text{matrix}[i][j] + \text{matrix}[i+1][j+1] \leq \text{matrix}[i][j+1] + \text{matrix}[i+1][j]$
2. SMAWK 算法利用这一性质, 可以在 $O(m+n)$ 时间内找到每行的最小值
3. 主要步骤包括行压缩和递归求解

参数:

`matrix`: 一个 Monge 矩阵

返回:

`int[]`: 每行最小值的列索引

时间复杂度: $O(m+n)$, 其中 m 是行数, n 是列数

空间复杂度: $O(m+n)$

*/

```

int m = matrix.length;
if (m == 0) {
    return new int[0];
}

```

```

int n = matrix[0].length;

// 构造行索引和列索引数组
List<Integer> rows = new ArrayList<>();
List<Integer> cols = new ArrayList<>();
for (int i = 0; i < m; ++i) rows.add(i);
for (int i = 0; i < n; ++i) cols.add(i);

// 调用递归实现
List<Integer> result = smawkRec(rows, cols, matrix);

// 转换为数组
int[] resArray = new int[m];
for (int i = 0; i < m; ++i) {
    resArray[i] = result.get(i);
}

return resArray;
}

private static List<Integer> reduceRows(List<Integer> rows, int[][] matrix) {
    /*行压缩：只保留可能成为最小值的行*/
    List<Integer> stack = new ArrayList<>();
    for (int i : rows) {
        while (stack.size() >= 2) {
            int j1 = stack.get(stack.size() - 2);
            int j2 = stack.get(stack.size() - 1);
            // 比较两个行在列 stack.size()-1 处的值
            if (matrix[j1][stack.size() - 1] <= matrix[i][stack.size() - 1]) {
                break;
            } else {
                stack.remove(stack.size() - 1);
            }
        }
        stack.add(i);
    }
    return stack;
}

private static List<Integer> smawkRec(List<Integer> rows, List<Integer> cols, int[][] matrix)
{
    /*递归实现 SMAWK 算法*/
    if (rows.isEmpty()) {

```

```

    return new ArrayList<>();
}

// 行压缩
List<Integer> reducedRows = reduceRows(rows, matrix);

// 递归求解列数为奇数的子问题
List<Integer> halfCols = new ArrayList<>();
for (int i = 1; i < cols.size(); i += 2) {
    halfCols.add(cols.get(i));
}
List<Integer> minCols = new ArrayList<>(Collections.nCopies(reducedRows.size(), -1));

if (!halfCols.isEmpty()) {
    // 递归求解
    List<Integer> result = smawkRec(reducedRows, halfCols, matrix);
    // 复制结果
    for (int i = 0; i < result.size(); ++i) {
        minCols.set(i, result.get(i));
    }
}

// 扩展结果到所有列
List<Integer> result = new ArrayList<>(Collections.nCopies(rows.size(), 0));
int k = 0; // minCols 的索引

for (int i = 0; i < rows.size(); ++i) {
    int row = rows.get(i);
    // 确定当前行的最小值可能在哪个区间
    int start = (i == 0) ? 0 : (k > 0 ? minCols.get(k - 1) : 0);
    int end = (k < minCols.size()) ? minCols.get(k) : cols.get(cols.size() - 1);

    // 在这个区间内查找最小值
    int minValue = Integer.MAX_VALUE;
    int minCol = start;

    // 注意这里 cols 是原始列的子集，需要在 cols 中遍历
    int startIndex = cols.indexOf(start);
    int endIndex = cols.indexOf(end);
    if (startIndex != -1 && endIndex != -1) {
        for (int j = startIndex; j <= endIndex; ++j) {
            int col = cols.get(j);
            if (col < matrix[0].length && matrix[row][col] < minValue) {

```

```

        minVal = matrix[row][col];
        minCol = col;
    }
}

result.set(i, minCol);

// 如果当前行在 reducedRows 中，且不是最后一行，k 前进
if (k < reducedRows.size() && row == reducedRows.get(k)) {
    k++;
}
}

return result;
}

// ===== 优化体系: Aliens Trick (二分约束参数+可行性 DP) =====

public static class AliensTrickResult {
    double lambda;
    double value;

    public AliensTrickResult(double lambda, double value) {
        this.lambda = lambda;
        this.value = value;
    }
}

public static AliensTrickResult aliensTrick(BiFunction<Double, Double[], Double[]> costFunc,
                                             DoublePredicate checkFunc,
                                             double left, double right, double eps) {
/*
Aliens Trick (二分约束参数+可行性 DP)

```

问题描述:

解决带约束的优化问题，通常形如最小化总成本，同时满足某些约束条件

解题思路:

1. 将约束条件转化为参数 λ ，构造拉格朗日函数
2. 对 λ 进行二分查找，使用可行性 DP 判断当前 λ 下是否满足约束
3. 根据可行性 DP 的结果调整二分区间

参数:

costFunc: 计算带参数 λ 的成本函数, 返回 double[2], 其中[0]是当前值, [1]是约束值
checkFunc: 检查当前解是否满足约束的函数
left: 二分左边界
right: 二分右边界
eps: 精度要求

返回:

AliensTrickResult: 包含最优参数 λ 和对应最优解的结果类

时间复杂度: $O(\log((right-left)/eps) * T(DP))$, 其中 $T(DP)$ 是一次 DP 的时间复杂度

*/

```
double bestLambda = left;  
double bestValue = 0.0;
```

```
while (right - left > eps) {  
    double mid = (left + right) / 2;  
    // 计算当前参数下的解和约束值  
    Double[] result = costFunc.apply(mid, new Double[0]);  
    if (result != null && result.length >= 2) {  
        double currentValue = result[0];  
        double constraintValue = result[1];  
  
        if (checkFunc.test(constraintValue)) {  
            // 满足约束, 尝试更小的参数  
            right = mid;  
            bestLambda = mid;  
            bestValue = currentValue;  
        } else {  
            // 不满足约束, 需要增大参数  
            left = mid;  
        }  
    }  
}
```

```
return new AliensTrickResult(bestLambda, bestValue);  
}
```

// 重载, 提供默认精度

```
public static AliensTrickResult aliensTrick(BiFunction<Double, Double[], Double[]> costFunc,  
                                            DoublePredicate checkFunc,  
                                            double left, double right) {  
    return aliensTrick(costFunc, checkFunc, left, right, 1e-7);
```

```
}
```

```
// ===== 图上 DP→最短路：分层图建模 =====
```

```
public static int layeredGraphDijkstra(int n, int m, List<int[]> edges, int k) {  
    /*  
     * 分层图 Dijkstra 算法  
     */
```

问题描述：

给定一个图，允许最多使用 k 次特殊操作（如跳跃、免费通行等），求最短路径

解题思路：

1. 构建分层图，每层代表使用不同次数的特殊操作
2. 对于每个节点 u ，在第 i 层表示到达 u 时已经使用了 i 次特殊操作
3. 使用 Dijkstra 算法在分层图上寻找最短路径

参数：

n : 节点数量

m : 边的数量

$edges$: 边的列表，每个元素为 $[u, v, w]$ 表示 u 到 v 的权为 w 的边

k : 允许使用的特殊操作次数

返回：

int: 从节点 0 到节点 $n-1$ 的最短路径长度

时间复杂度: $O((n*k + m*k) \log(n*k))$

空间复杂度: $O(n*k + m*k)$

*/

// 构建分层图的邻接表

```
List<List<int[]>> graph = new ArrayList<>();  
for (int i = 0; i < n * (k + 1); ++i) {  
    graph.add(new ArrayList<>());  
}
```

// 添加普通边（不使用特殊操作）

```
for (int[] edge : edges) {  
    int u = edge[0];  
    int v = edge[1];  
    int w = edge[2];  
    for (int i = 0; i <= k; ++i) {  
        graph.get(u + i * n).add(new int[]{v + i * n, w});  
    }  
}
```

```

// 添加使用特殊操作的边（如果允许的话）
for (int[] edge : edges) {
    int u = edge[0];
    int v = edge[1];
    for (int i = 0; i < k; ++i) {
        // 这里假设特殊操作可以免费通行（权为 0），具体根据问题调整
        graph.get(u + i * n).add(new int[]{v + (i + 1) * n, 0});
    }
}

// Dijkstra 算法
int[] dist = new int[n * (k + 1)];
Arrays.fill(dist, Integer.MAX_VALUE);
dist[0] = 0; // 假设起点是节点 0
PriorityQueue<int[]> heap = new PriorityQueue<>((a, b) -> a[0] - b[0]);
heap.offer(new int[]{0, 0}); // (距离, 节点)

while (!heap.isEmpty()) {
    int[] current = heap.poll();
    int d = current[0];
    int u = current[1];

    if (d > dist[u]) {
        continue;
    }

    for (int[] neighbor : graph.get(u)) {
        int v = neighbor[0];
        int w = neighbor[1];
        if (dist[v] > d + w) {
            dist[v] = d + w;
            heap.offer(new int[]{dist[v], v});
        }
    }
}

// 取所有层中到达终点的最小值
int result = Integer.MAX_VALUE;
for (int i = 0; i <= k; ++i) {
    if (dist[n - 1 + i * n] < result) {
        result = dist[n - 1 + i * n];
    }
}

```

```
    }

    return result;
}

// ====== 冷门模型：期望 DP 遇环的方程组解（高斯消元） ======
```

```
public static double[] gaussianElimination(double[][] matrix) {
    /*
    高斯消元法求解线性方程组
```

问题描述：

求解形如 $Ax = b$ 的线性方程组

解题思路：

1. 构建增广矩阵
2. 进行高斯消元，将矩阵转化为行阶梯形
3. 回代求解

参数：

matrix：增广矩阵，每行最后一个元素是 b 的值

返回：

double[]：方程组的解

时间复杂度： $O(n^3)$

空间复杂度： $O(n^2)$

*/

```
int n = matrix.length;
final double eps = 1e-9;
```

// 高斯消元过程

```
for (int i = 0; i < n; ++i) {
    // 找到主元行（当前列中绝对值最大的行）
    int maxRow = i;
    for (int j = i; j < n; ++j) {
        if (Math.abs(matrix[j][i]) > Math.abs(matrix[maxRow][i])) {
            maxRow = j;
        }
    }
}
```

// 交换主元行和当前行

```
double[] temp = matrix[i];
```

```

matrix[i] = matrix[maxRow];
matrix[maxRow] = temp;

// 如果主元为 0, 方程组可能有无穷多解或无解
if (Math.abs(matrix[i][i]) < eps) {
    continue;
}

// 消元过程
for (int j = i + 1; j < n; ++j) {
    double factor = matrix[j][i] / matrix[i][i];
    for (int k = i; k <= n; ++k) {
        matrix[j][k] -= factor * matrix[i][k];
    }
}
}

// 回代求解
double[] x = new double[n];
for (int i = n - 1; i >= 0; --i) {
    x[i] = matrix[i][n];
    for (int j = i + 1; j < n; ++j) {
        x[i] -= matrix[i][j] * x[j];
    }
    x[i] /= matrix[i][i];
}

return x;
}

public static double[] expectationDPWithCycles(int n, List<List<double[]>> transitions) {
/*
期望 DP 处理有环情况 (使用高斯消元)

```

问题描述:

在有环的状态转移图中计算期望

解题思路:

1. 对于每个状态, 建立期望方程
2. 使用高斯消元求解方程组

参数:

n: 状态数量

transitions: 转移概率列表, transitions[i]是一个列表, 每个元素为[j, p]表示从 i 转移到 j 的概率为 p

返回:

double[]: 每个状态的期望值

时间复杂度: $O(n^3)$

空间复杂度: $O(n^2)$

*/

// 构建线性方程组的增广矩阵

```
double[][] matrix = new double[n][n + 1];
```

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

```
    matrix[i][i] = 1.0; // 方程左边: E[i] - sum(p_ij * E[j]) = cost[i]
```

// 假设每个状态的代价为 1, 具体根据问题调整

```
double cost = 1.0;
```

```
matrix[i][n] = cost;
```

```
for (double[] transition : transitions.get(i)) {
```

```
    int j = (int) transition[0];
```

```
    double p = transition[1];
```

```
    if (i != j) { // 避免自环的特殊处理
```

```
        matrix[i][j] -= p;
```

```
}
```

```
}
```

```
}
```

// 使用高斯消元求解

```
return gaussianElimination(matrix);
```

```
}
```

// ====== 冷门模型: 插头 DP (轮廓线 DP) ======

```
public static int plugDP(int[][] grid) {
```

```
/*
```

插头 DP (轮廓线 DP) 示例: 求网格中哈密顿回路的数量

问题描述:

给定一个网格, 求其中哈密顿回路的数量

解题思路:

1. 使用轮廓线 DP, 记录当前处理到的位置和轮廓线状态

2. 插头表示连接的状态，通常用二进制表示
3. 使用哈希表优化空间复杂度

参数:

grid: 网格，1 表示可通行，0 表示障碍物

返回:

int: 哈密顿回路的数量

时间复杂度: $O(n*m*4^{\min(n, m)})$

空间复杂度: $O(4^{\min(n, m)})$

*/

```
int n = grid.length;
```

```
if (n == 0) {
```

```
    return 0;
```

```
}
```

```
int m = grid[0].length;
```

// 使用哈希表优化

```
Map<Long, Integer> dp = new HashMap<>();
```

// 初始状态: 左上角没有插头

```
dp.put(0L, 1);
```

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

// 新的一行开始, 需要将状态左移一位

```
Map<Long, Integer> newDp = new HashMap<>();
```

```
for (Map.Entry<Long, Integer> entry : dp.entrySet()) {
```

```
    long state = entry.getKey();
```

```
    int cnt = entry.getValue();
```

// 左移一位, 移除最左边的插头

```
    long newState = state << 1;
```

```
    newDp.put(newState, newDp.getOrDefault(newState, 0) + cnt);
```

```
}
```

```
dp = newDp;
```

```
for (int j = 0; j < m; ++j) {
```

```
    Map<Long, Integer> newDp2 = new HashMap<>();
```

```
    for (Map.Entry<Long, Integer> entry : dp.entrySet()) {
```

```
        long state = entry.getKey();
```

```
        int cnt = entry.getValue();
```

```

// 当前位置左边和上边的插头状态
int left = (int) ((state >> (2 * j)) & 3);
int up = (int) ((state >> (2 * (j + 1))) & 3);

// 如果当前位置是障碍物，跳过
if (grid[i][j] == 0) {
    // 只有当左右插头都不存在时才合法
    if (left == 0 && up == 0) {
        newDp2.put(state, newDp2.getOrDefault(state, 0) + cnt);
    }
    continue;
}

// 处理各种插头组合情况
// 1. 没有左插头和上插头
if (left == 0 && up == 0) {
    // 只能创建新的插头对（用于回路的开始）
    if (i < n - 1 && j < m - 1 && grid[i+1][j] == 1 && grid[i][j+1] == 1) {
        long newState = state | (1L << (2 * j)) | (2L << (2 * (j + 1)));
        newDp2.put(newState, newDp2.getOrDefault(newState, 0) + cnt);
    }
}

// 2. 只有左插头
else if (left != 0 && up == 0) {
    // 向下延伸
    if (i < n - 1 && grid[i+1][j] == 1) {
        newDp2.put(state, newDp2.getOrDefault(state, 0) + cnt);
    }
    // 向右延伸
    if (j < m - 1 && grid[i][j+1] == 1) {
        long newState = (state & ~(3L << (2 * j))) | (left << (2 * (j + 1)));
        newDp2.put(newState, newDp2.getOrDefault(newState, 0) + cnt);
    }
}

// 3. 只有上插头
else if (left == 0 && up != 0) {
    // 向右延伸
    if (j < m - 1 && grid[i][j+1] == 1) {
        newDp2.put(state, newDp2.getOrDefault(state, 0) + cnt);
    }
    // 向下延伸
}

```

```

        if (i < n - 1 && grid[i+1][j] == 1) {
            long newState = (state & ~(3L << (2 * (j + 1)))) | (up << (2 * j));
            newDp2.put(newState, newDp2.getOrDefault(newState, 0) + cnt);
        }
    }

    // 4. 同时有左插头和上插头
    else {
        // 合并插头
        long newState = (state & ~(3L << (2 * j))) & ~(3L << (2 * (j + 1)));

        // 如果是形成回路的最后一步
        if (left == up) {
            // 检查是否所有插头都已连接
            if (newState == 0 && i == n - 1 && j == m - 1) {
                newDp2.put(newState, newDp2.getOrDefault(newState, 0) + cnt);
            }
        } else {
            // 合并两个不同的插头
            newDp2.put(newState, newDp2.getOrDefault(newState, 0) + cnt);
        }
    }

    dp = newDp2;
}

}

// 最终状态应该是没有任何插头（形成回路）
return dp.getOrDefault(0L, 0);
}
}

// ====== 冷门模型：树上背包的优化 ======

```

```

public static int treeKnapsackOptimized(int root, int capacity, List<List<Integer>> tree,
                                         int[] weights, int[] values) {
/*
树上背包的优化实现（小到大合并）

```

问题描述：

在树上选择一些节点，使得总重量不超过容量，且总价值最大

解题思路：

1. 使用后序遍历处理子树
2. 使用小到大合并的策略优化复杂度
3. 对于每个节点，维护一个容量为 capacity 的背包

参数：

root: 根节点
 capacity: 背包容量
 tree: 树的邻接表
 weights: 每个节点的重量
 values: 每个节点的价值

返回：

int: 最大价值

时间复杂度：O(n*capacity^2)，但通过小到大合并可以降低常数

空间复杂度：O(n*capacity)

*/

```

int n = tree.size();
int[][] dp = new int[n][capacity + 1];
int[] size = new int[n];

```

// 初始化 dp 数组

```

for (int i = 0; i < n; ++i) {
    Arrays.fill(dp[i], 0);
}

```

// 深度优先搜索处理子树

```
dfsTreeKnapsack(root, -1, capacity, tree, weights, values, dp, size);
```

// 返回根节点的最大价值

```

int maxVal = 0;
for (int val : dp[root]) {
    maxVal = Math.max(maxVal, val);
}
return maxVal;
}

```

```

private static void dfsTreeKnapsack(int u, int parent, int capacity,
                                    List<List<Integer>> tree, int[] weights, int[] values,
                                    int[][] dp, int[] size) {
    // 初始化当前节点
    size[u] = 1;
    if (weights[u] <= capacity) {

```

```

dp[u][weights[u]] = values[u];
}

// 对每个子节点，按照子树大小排序，小的先合并
List<int[]> children = new ArrayList<>();
for (int v : tree.get(u)) {
    if (v != parent) {
        dfsTreeKnapsack(v, u, capacity, tree, weights, values, dp, size);
        children.add(new int[]{size[v], v});
    }
}

// 按子树大小排序
children.sort(Comparator.comparingInt(a -> a[0]));

for (int[] child : children) {
    int sz = child[0];
    int v = child[1];

    // 逆序遍历容量，避免重复计算
    for (int i = Math.min(size[u], capacity); i >= 0; --i) {
        if (dp[u][i] == 0 && i != 0) continue;
        for (int j = 1; j <= Math.min(sz, capacity - i); ++j) {
            if (dp[v][j] > 0 && i + j <= capacity) {
                dp[u][i + j] = Math.max(dp[u][i + j], dp[u][i] + dp[v][j]);
            }
        }
    }
}

// 更新子树大小
size[u] += sz;
}

// ===== 补充题目与应用 =====
// 以下是一些使用上述高级 DP 技术的经典题目及其代码实现

// 1. 编辑距离问题 (LeetCode 72)
public static int editDistance(String word1, String word2) {
/*
LeetCode 72. 编辑距离
题目链接: https://leetcode-cn.com/problems/edit-distance/

```

问题描述:

给你两个单词 word1 和 word2，计算出将 word1 转换成 word2 所使用的最少操作数。

你可以对一个单词进行如下三种操作：插入一个字符、删除一个字符、替换一个字符。

解题思路:

使用二维 DP， $dp[i][j]$ 表示 word1 的前 i 个字符转换为 word2 的前 j 个字符所需的最少操作数。

时间复杂度： $O(m*n)$

空间复杂度： $O(m*n)$

*/

```
int m = word1.length();
```

```
int n = word2.length();
```

// $dp[i][j]$ 表示 word1 的前 i 个字符转换为 word2 的前 j 个字符所需的最少操作数

```
int[][] dp = new int[m + 1][n + 1];
```

// 初始化边界

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

```
    dp[i][0] = i;
```

```
}
```

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

```
    dp[0][j] = j;
```

```
}
```

// 动态规划填表

```
for (int i = 1; i <= m; ++i) {
```

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

```
        if (word1.charAt(i - 1) == word2.charAt(j - 1)) {
```

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

```
        } else {
```

```
            dp[i][j] = Math.min(Math.min(dp[i - 1][j], dp[i][j - 1]), dp[i - 1][j - 1]) +
```

```
1;
```

```
    }
```

```
}
```

```
}
```

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

```
}
```

// 2. 最长递增子序列 (LeetCode 300)

```
public static int lengthOfLIS(int[] nums) {
```

```
/*
```

LeetCode 300. 最长递增子序列

题目链接: <https://leetcode-cn.com/problems/longest-increasing-subsequence/>

问题描述:

给你一个整数数组 `nums`，找到其中最长严格递增子序列的长度。

解题思路:

使用贪心 + 二分查找优化的 DP 方法。

`tails[i]` 表示长度为 $i+1$ 的递增子序列的末尾元素的最小值。

时间复杂度: $O(n \log n)$

空间复杂度: $O(n)$

*/

```
if (nums == null || nums.length == 0) {  
    return 0;  
}
```

```
List<Integer> tails = new ArrayList<>();  
for (int num : nums) {  
    // 二分查找 num 应该插入的位置  
    int left = 0, right = tails.size();  
    while (left < right) {  
        int mid = (left + right) / 2;  
        if (tails.get(mid) < num) {  
            left = mid + 1;  
        } else {  
            right = mid;  
        }  
    }  
  
    if (left == tails.size()) {  
        tails.add(num);  
    } else {  
        tails.set(left, num);  
    }  
}
```

// 3. 背包问题变种 - 完全背包 (LeetCode 322)

```
public static int coinChange(int[] coins, int amount) {
```

/*

LeetCode 322. 零钱兑换

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

问题描述:

给你一个整数数组 coins , 表示不同面额的硬币; 以及一个整数 amount , 表示总金额。

计算并返回可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额, 返回-1。

解题思路:

使用完全背包的思想, dp[i]表示凑成金额 i 所需的最少硬币数。

时间复杂度: O(amount * n)

空间复杂度: O(amount)

*/

// 初始化 dp 数组为无穷大

int[] dp = new int[amount + 1];

Arrays.fill(dp, Integer.MAX_VALUE);

dp[0] = 0; // 凑成金额 0 需要 0 个硬币

for (int coin : coins) {

 for (int i = coin; i <= amount; ++i) {

 if (dp[i - coin] != Integer.MAX_VALUE) {

 dp[i] = Math.min(dp[i], dp[i - coin] + 1);

 }

 }

}

return dp[amount] == Integer.MAX_VALUE ? -1 : dp[amount];

}

// 4. 矩阵链乘法 (区间 DP 的经典应用)

public static class MatrixChainResult {

 int[][] dp;

 int[][] s;

 public MatrixChainResult(int[][] dp, int[][] s) {

 this.dp = dp;

 this.s = s;

 }

}

 public static MatrixChainResult matrixChainOrder(int[] p) {

 /*

 矩阵链乘法问题

 题目来源: 算法导论

问题描述：

给定一系列矩阵，计算乘法顺序使得标量乘法的次数最少。

解题思路：

使用区间 DP， $dp[i][j]$ 表示计算第 i 到第 j 个矩阵的乘积所需的最少标量乘法次数。
可以使用 Knuth 优化进一步降低时间复杂度。

时间复杂度： $O(n^3)$

空间复杂度： $O(n^2)$

*/

```
int n = p.length - 1; // 矩阵的个数
// dp[i][j] 表示计算第 i 到第 j 个矩阵的乘积所需的最少标量乘法次数
int[][] dp = new int[n + 1][n + 1];
// s[i][j] 记录最优分割点
int[][] s = new int[n + 1][n + 1];

// 枚举区间长度
for (int length = 2; length <= n; ++length) {
    for (int i = 1; i + length - 1 <= n; ++i) {
        int j = i + length - 1;
        dp[i][j] = Integer.MAX_VALUE;
        // 枚举分割点
        for (int k = i; k < j; ++k) {
            // 计算当前分割点的代价
            int cost = dp[i][k] + dp[k + 1][j] + p[i - 1] * p[k] * p[j];
            if (cost < dp[i][j]) {
                dp[i][j] = cost;
                s[i][j] = k;
            }
        }
    }
}

return new MatrixChainResult(dp, s);
}

// 5. 旅行商问题 (TSP) 的 DP 实现
public static int travelingSalesmanProblem(int[][] graph) {
/*
旅行商问题
题目来源：算法竞赛经典问题
*/}
```

问题描述：

给定一个完全图，找到一条访问每个城市恰好一次并返回起点的最短路径。

解题思路：

使用状态压缩 DP， $dp[mask][u]$ 表示访问过的城市集合为 $mask$ ，当前在城市 u 时的最短路径长度。

时间复杂度： $O(n^2 * 2^n)$

空间复杂度： $O(n * 2^n)$

*/

```
int n = graph.length;
```

```
// dp[mask][u] 表示访问过的城市集合为 mask，当前在城市 u 时的最短路径长度
```

```
int[][] dp = new int[1 << n][n];
```

```
for (int[] row : dp) {
```

```
    Arrays.fill(row, Integer.MAX_VALUE);
```

```
}
```

```
// 初始状态：只访问了起点，路径长度为 0
```

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

```
    dp[1 << i][i] = 0;
```

```
}
```

```
// 枚举所有可能的状态
```

```
for (int mask = 1; mask < (1 << n); ++mask) {
```

```
    // 枚举当前所在的城市
```

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

```
        if ((mask & (1 << u)) == 0) {
```

```
            continue;
```

```
}
```

```
    // 枚举下一个要访问的城市
```

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

```
        if ((mask & (1 << v)) != 0) {
```

```
            continue;
```

```
}
```

```
        int newMask = mask | (1 << v);
```

```
        if (dp[mask][u] != Integer.MAX_VALUE && graph[u][v] != Integer.MAX_VALUE) {
```

```
            dp[newMask][v] = Math.min(dp[newMask][v], dp[mask][u] + graph[u][v]);
```

```
}
```

```
}
```

```
}
```

```
}
```

```
// 找到最短的回路
```

```
int result = Integer.MAX_VALUE;
```

```

        for (int u = 0; u < n; ++u) {
            if (dp[(1 << n) - 1][u] != Integer.MAX_VALUE && graph[u][0] != Integer.MAX_VALUE) {
                result = Math.min(result, dp[(1 << n) - 1][u] + graph[u][0]);
            }
        }

        return result;
    }
}

```

// 6. 区间 DP: 最优三角剖分

```

public static int minimumScoreTriangulation(int[] values) {
    /*
    LeetCode 1039. 多边形三角剖分的最低得分
    题目链接: https://leetcode-cn.com/problems/minimum-score-triangulation-of-polygon/
}

```

问题描述:

给定一个凸多边形，将其三角剖分，使得所有三角形的顶点乘积之和最小。

解题思路:

使用区间 DP， $dp[i][j]$ 表示从顶点 i 到顶点 j 的多边形三角剖分的最小得分。

时间复杂度: $O(n^3)$

空间复杂度: $O(n^2)$

*/

```

int n = values.length;
// dp[i][j] 表示从顶点 i 到顶点 j 的多边形三角剖分的最小得分
int[][] dp = new int[n][n];
for (int[] row : dp) {
    Arrays.fill(row, 0);
}

```

// 枚举区间长度

```

for (int length = 3; length <= n; ++length) {
    for (int i = 0; i + length - 1 < n; ++i) {
        int j = i + length - 1;
        dp[i][j] = Integer.MAX_VALUE;
        // 枚举中间点
        for (int k = i + 1; k < j; ++k) {
            dp[i][j] = Math.min(dp[i][j],
                dp[i][k] + dp[k][j] + values[i] * values[k] * values[j]);
        }
    }
}

```

```

        return dp[0][n - 1];
    }

// 7. 博弈 DP: 石子游戏
public static boolean stoneGame(int[] piles) {
    /*
    LeetCode 877. 石子游戏
    题目链接: https://leetcode-cn.com/problems/stone-game/

```

问题描述:

给定一个表示石子堆的数组，两个玩家轮流从两端取石子，每次只能取一个，取到最后一个石子的人获胜。

判断先手是否必胜。

解题思路:

使用区间 DP， $dp[i][j]$ 表示在区间 $[i, j]$ 中，先手能获得的最大净胜分。

时间复杂度: $O(n^2)$

空间复杂度: $O(n^2)$

*/

```
int n = piles.length;
```

// $dp[i][j]$ 表示在区间 $[i, j]$ 中，先手能获得的最大净胜分

```
int[][] dp = new int[n][n];
```

// 初始化单个石子堆

```
for (int i = 0; i < n; ++i) {
    dp[i][i] = piles[i];
}
```

// 枚举区间长度

```
for (int length = 2; length <= n; ++length) {
    for (int i = 0; i + length - 1 < n; ++i) {
        int j = i + length - 1;
        // 先手可以选择取左边或右边
        dp[i][j] = Math.max(piles[i] - dp[i + 1][j], piles[j] - dp[i][j - 1]);
    }
}
```

// 先手净胜分大于 0 则必胜

```
return dp[0][n - 1] > 0;
```

}

```
// 8. 数位 DP: 统计 1 出现的次数
public static int countDigitOne(int n) {
    /*
        LeetCode 233. 数字 1 的个数
        题目链接: https://leetcode-cn.com/problems/number-of-digit-one/
    
```

问题描述:

给定一个整数 n , 计算所有小于等于 n 的非负整数中数字 1 出现的个数。

解题思路:

使用数位 DP, 逐位处理每一位上 1 出现的次数。

时间复杂度: $O(\log n)$

空间复杂度: $O(\log n)$

*/

```
if (n <= 0) {
    return 0;
}
```

```
String s = Integer.toString(n);
```

```
int length = s.length();
```

```
int count = 0;
```

// 逐位处理

```
for (int i = 0; i < length; ++i) {
    long high = 0;
    if (i > 0) {
        high = Long.parseLong(s.substring(0, i));
    }
    int current = s.charAt(i) - '0';
    long low = 0;
    if (i < length - 1) {
        low = Long.parseLong(s.substring(i + 1));
    }
    long digit = (long) Math.pow(10, length - i - 1);
```

```
    if (current == 0) {
        // 当前位为 0, 高位决定
        count += high * digit;
    } else if (current == 1) {
        // 当前位为 1, 高位+低位+1
        count += high * digit + low + 1;
    } else {
```

```

        // 当前位大于 1，高位+1
        count += (high + 1) * digit;
    }
}

return count;
}

// 9. 树形 DP: 打家劫舍 III
public static class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int x) { val = x; }
}

public static int[] robDFS(TreeNode node) {
    if (node == null) {
        return new int[] {0, 0};
    }

    int[] left = robDFS(node.left);
    int[] right = robDFS(node.right);

    // 偷当前节点，不能偷子节点
    int robCurrent = node.val + left[1] + right[1];
    // 不偷当前节点，可以选择偷或不偷子节点
    int notRobCurrent = Math.max(left[0], left[1]) + Math.max(right[0], right[1]);

    return new int[] {robCurrent, notRobCurrent};
}

public static int rob(TreeNode root) {
/*
LeetCode 337. 打家劫舍 III
题目链接: https://leetcode-cn.com/problems/house-robber-iii/
*/
}

```

问题描述:

在上次打劫完一条街道之后和一圈房屋后，小偷又发现了一个新的可行窃的地区。

这个地区只有一个入口，我们称之为“根”。除了“根”之外，每栋房子有且只有一个“父”房子与之相连。

一番侦察之后，聪明的小偷意识到“这个地方的所有房屋的排列类似于一棵二叉树”。

如果两个直接相连的房子在同一天晚上被打劫，房屋将自动报警。

计算在不触动警报的情况下，小偷一晚能够盗取的最高金额。

解题思路：

使用树形 DP，对于每个节点，维护两个状态：偷或不偷。

时间复杂度：O(n)

空间复杂度：O(h)，h 为树的高度

*/

```
int[] result = robDFS(root);
return Math.max(result[0], result[1]);
}
```

// 10. 状态压缩 DP：蒙斯特曼问题

```
public static int monsterGame(int[][] grid) {
    /*

```

蒙斯特曼问题

题目来源：算法竞赛问题

问题描述：

在网格中放置怪物，使得任何两个怪物都不在同一行、同一列或对角线上。

解题思路：

使用状态压缩 DP， $dp[i][mask]$ 表示处理到第 i 行，已放置的列的状态为 mask 时的方案数。

时间复杂度：O($n * 2^n$)

空间复杂度：O(2^n)

*/

```
int n = grid.length;
// dp[i][mask] 表示处理到第 i 行，已放置的列的状态为 mask 时的方案数
long[] dp = new long[1 << n];
dp[0] = 1;
```

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

```
    long[] newDp = new long[1 << n];
```

```
    for (int mask = 0; mask < (1 << n); ++mask) {
```

```
        if (dp[mask] == 0) {
```

```
            continue;
```

```
}
```

```
        // 枚举所有可能的放置位置
```

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

```
            // 检查是否可以在(i, j)放置怪物
```

```
            if ((mask & (1 << j)) == 0 && grid[i][j] == 1) {
```

```
                // 检查对角线
```

```

        boolean valid = true;
        for (int k = 0; k < i; ++k) {
            if ((mask & (1 << k)) != 0 && Math.abs(k - j) == i - k) {
                valid = false;
                break;
            }
        }
        if (valid) {
            newDp[mask | (1 << j)] += dp[mask];
        }
    }
    dp = newDp;
}

return (int) dp[(1 << n) - 1];
}

```

// 11. 高维 DP: 三维背包

```

public static int threeDimensionKnapsack(int n, int[] capacity, int[][] items) {
/*
三维背包问题
题目来源: 算法竞赛问题

```

问题描述:

有 n 个物品，每个物品有体积、重量、价值三个属性，背包有体积和重量两个限制，求最大价值。

解题思路:

使用三维 DP， $dp[i][j][k]$ 表示前 i 个物品，体积为 j ，重量为 k 时的最大价值。

时间复杂度: $O(n * V * W)$

空间复杂度: $O(n * V * W)$

*/

```

int V = capacity[0];
int W = capacity[1];
// 初始化 dp 数组
int[][][] dp = new int[n + 1][V + 1][W + 1];

for (int i = 1; i <= n; ++i) {
    int v = items[i-1][0];
    int w = items[i-1][1];
    int val = items[i-1][2];

```

```

        for (int j = 0; j <= V; ++j) {
            for (int k = 0; k <= W; ++k) {
                // 不选当前物品
                dp[i][j][k] = dp[i-1][j][k];
                // 选当前物品（如果有足够的空间）
                if (j >= v && k >= w) {
                    dp[i][j][k] = Math.max(dp[i][j][k], dp[i-1][j-v][k-w] + val);
                }
            }
        }

        return dp[n][V][W];
    }
}

```

// 12. 斜率优化 DP 示例

```

public static class ConvexHullTrick {
    /*
    凸包优化技巧示例
    题目来源：算法竞赛问题

```

问题描述：

当状态转移方程形如 $dp[i] = \min\{dp[j] + a[i] * b[j] + c\}$ 时，可以使用凸包优化。

解题思路：

将转移方程转换为直线的形式，维护凸包以快速查询最小值。

时间复杂度：O(n)

空间复杂度：O(n)

*/

```

private class Line {
    long k, b;
    Line(long k, long b) {
        this.k = k;
        this.b = b;
    }
}

```

```
private Deque<Line> dq = new LinkedList<>();
```

// 添加一条直线 $y = kx + b$

```
public void addLine(long k, long b) {
```

```

// 当队列中至少有两条直线时，检查是否需要删除末尾的直线
while (dq.size() >= 2) {
    Line l1 = dq.get(dq.size() - 2);
    Line l2 = dq.get(dq.size() - 1);
    // 判断直线 l1 和 l2 的交点是否在 l2 和新直线的交点右侧
    if ((l2.b - l1.b) * (k - l2.k) >= (b - l2.b) * (l2.k - l1.k)) {
        dq.removeLast();
    } else {
        break;
    }
}
dq.addLast(new Line(k, b));
}

// 查询 x 处的最小值
public long query(long x) {
    // 如果队列中至少有两条直线，且第一条直线在 x 处的值大于第二条，删除第一条
    while (dq.size() >= 2) {
        Line l1 = dq.getFirst();
        Line l2 = dq.get(1);
        if (l1.k * x + l1.b >= l2.k * x + l2.b) {
            dq.removeFirst();
        } else {
            break;
        }
    }
    if (dq.isEmpty()) {
        return Long.MAX_VALUE;
    }
    Line l = dq.getFirst();
    return l.k * x + l.b;
}
}

// ===== 优化体系：Knuth 优化 =====

// Knuth 优化用于优化形如 dp[i][j] = min{dp[i][k] + dp[k+1][j]} + w(i, j) 的 DP
// 当满足四边形不等式时，最优转移点单调

static class KnuthOptimizationResult {
    int[][] dp;
    int[][] opt;
}

```

```

public KnuthOptimizationResult(int[][] dp, int[][] opt) {
    this.dp = dp;
    this.opt = opt;
}

interface CostFunction {
    int cost(int i, int j);
}

static KnuthOptimizationResult knuthOptimization(int n, CostFunction costFunc) {
    /*
     * Knuth 优化的 DP 算法
    */
}

```

问题描述:

解决区间 DP 问题，其中状态转移方程满足四边形不等式

解题思路:

1. 使用 Knuth 优化将时间复杂度从 $O(n^3)$ 降低到 $O(n^2)$
2. 维护最优转移点数组 $opt[i][j]$ ，表示计算 $dp[i][j]$ 时的最优 k 值
3. 根据 $opt[i][j-1] \leq opt[i][j] \leq opt[i+1][j]$ 的性质进行剪枝

参数:

n : 区间长度

$costFunc$: 计算区间 (i, j) 代价的函数

返回:

$KnuthOptimizationResult$: 包含 dp 数组和 opt 数组的结果类

时间复杂度: $O(n^2)$

空间复杂度: $O(n^2)$

*/

// 初始化 dp 和 opt 数组

```

int[][] dp = new int[n + 1][n + 1];
int[][] opt = new int[n + 1][n + 1];

```

// 初始化长度为 1 的区间

```

for (int i = 1; i <= n; ++i) {
    dp[i][i] = 0;
    opt[i][i] = i;
}

```

// 枚举区间长度

```

for (int length = 2; length <= n; ++length) {
    // 枚举起始点
    for (int i = 1; i + length - 1 <= n; ++i) {
        int j = i + length - 1;
        // 初始化为无穷大
        dp[i][j] = Integer.MAX_VALUE;
        // 根据 Knuth 优化的性质，最优 k 在 opt[i][j-1] 到 opt[i+1][j] 之间
        int upperK = (i + 1 <= j) ? opt[i + 1][j] : j - 1;
        for (int k = opt[i][j - 1]; k <= Math.min(upperK, j - 1); ++k) {
            if (dp[i][k] != Integer.MAX_VALUE && dp[k + 1][j] != Integer.MAX_VALUE) {
                int cost = costFunc.cost(i, j);
                if (cost != Integer.MAX_VALUE) {
                    int current = dp[i][k] + dp[k + 1][j] + cost;
                    if (current < dp[i][j]) {
                        dp[i][j] = current;
                        opt[i][j] = k;
                    }
                }
            }
        }
    }
}

return new KnuthOptimizationResult(dp, opt);
}

// ===== 优化体系: Divide & Conquer Optimization =====

private static void solveDivideConquer(int i, int l, int r, int opt_l, int opt_r,
                                       int[][] dp, CostFunction costFunc) {
/*
计算 dp[i][l..r]，其中最优转移点在 opt_l..opt_r 之间
*/
if (l > r) {
    return;
}

int mid = (l + r) / 2;
int best_k = opt_l;

// 在 opt_l 到 min(mid, opt_r) 之间寻找最优 k
for (int k = opt_l; k <= Math.min(mid, opt_r); ++k) {
    if (dp[i - 1][k] != Integer.MAX_VALUE) {

```

```

        int cost = costFunc.cost(k, mid);
        if (cost != Integer.MAX_VALUE) {
            int current = dp[i - 1][k] + cost;
            if (current < dp[i][mid]) {
                dp[i][mid] = current;
                best_k = k;
            }
        }
    }
}

// 递归处理左右子区间
solveDivideConquer(i, 1, mid - 1, opt_l, best_k, dp, costFunc);
solveDivideConquer(i, mid + 1, r, best_k, opt_r, dp, costFunc);
}

static int[][] divideConquerOptimization(int n, int m, CostFunction costFunc) {
/*
Divide & Conquer Optimization (分治优化)

```

问题描述:

解决形如 $dp[i][j] = \min\{dp[i-1][k] + cost(k, j)\}$, 其中 $k < j$
当转移满足决策单调性时使用

解题思路:

1. 利用决策单调性, 使用分治法优化 DP
2. 对于 $dp[i][j]$, 当 i 固定时, 最优转移点 k 随着 j 的增加而单调不减
3. 使用分治的方式计算每个区间的最优决策

参数:

n: 维度 1
m: 维度 2
costFunc: 计算 $cost(k, j)$ 的函数

返回:

int[][]: dp 数组

时间复杂度: $O(n*m \log m)$

空间复杂度: $O(n*m)$

*/

// 初始化 dp 数组

```

int[][] dp = new int[n + 1][m + 1];
for (int i = 0; i <= n; ++i) {

```

```

        Arrays.fill(dp[i], Integer.MAX_VALUE);
    }
    dp[0][0] = 0;

    // 对每个 i 应用分治优化
    for (int i = 1; i <= n; ++i) {
        solveDivideConquer(i, 1, m, 0, m, dp, costFunc);
    }

    return dp;
}

// ===== 优化体系: SMAWK 算法 (行最小查询) =====

private static List<Integer> reduceRows(List<Integer> rows, int[][] matrix) {
    /*行压缩: 只保留可能成为最小值的行*/
    Stack<Integer> stack = new Stack<>();
    for (int i : rows) {
        while (stack.size() >= 2) {
            int j1 = stack.get(stack.size() - 2);
            int j2 = stack.get(stack.size() - 1);
            // 比较两个行在列 stack.size()-1 处的值
            if (matrix[j1][stack.size() - 1] <= matrix[i][stack.size() - 1]) {
                break;
            } else {
                stack.pop();
            }
        }
        stack.push(i);
    }
    return new ArrayList<>(stack);
}

private static List<Integer> smawkRec(List<Integer> rows, List<Integer> cols, int[][] matrix)
{
    /*递归实现 SMAWK 算法*/
    if (rows.isEmpty()) {
        return new ArrayList<>();
    }

    // 行压缩
    List<Integer> reducedRows = reduceRows(rows, matrix);

```

```

// 递归求解列数为奇数的子问题
List<Integer> halfCols = new ArrayList<>();
for (int i = 1; i < cols.size(); i += 2) {
    halfCols.add(cols.get(i));
}
List<Integer> minCols = new ArrayList<>(Collections.nCopies(reducedRows.size(), -1));

if (!halfCols.isEmpty()) {
    // 递归求解
    List<Integer> result = smawkRec(reducedRows, halfCols, matrix);
    // 复制结果
    for (int i = 0; i < result.size(); ++i) {
        minCols.set(i, result.get(i));
    }
}

// 扩展结果到所有列
List<Integer> result = new ArrayList<>(Collections.nCopies(rows.size(), 0));
int k = 0; // minCols 的索引

for (int i = 0; i < rows.size(); ++i) {
    int row = rows.get(i);
    // 确定当前行的最小值可能在哪个区间
    int start = (i == 0) ? 0 : (k > 0 ? minCols.get(k - 1) : 0);
    int end = (k < minCols.size()) ? minCols.get(k) : cols.get(cols.size() - 1);

    // 在这个区间内查找最小值
    int minValue = Integer.MAX_VALUE;
    int minCol = start;

    // 注意这里 cols 是原始列的子集，需要在 cols 中遍历
    int startIdx = cols.indexOf(start);
    int endIdx = cols.indexOf(end);
    if (startIdx != -1 && endIdx != -1) {
        for (int idx = startIdx; idx <= endIdx; ++idx) {
            int col = cols.get(idx);
            if (col < matrix[0].length && matrix[row][col] < minValue) {
                minValue = matrix[row][col];
                minCol = col;
            }
        }
    }
}

```

```

        result.set(i, minCol);

        // 如果当前行在 reducedRows 中，且不是最后一行，k 前进
        if (k < reducedRows.size() && row == reducedRows.get(k)) {
            k++;
        }
    }

    return result;
}

```

```

static int[] smawk(int[][] matrix) {
/*
SMAWK 算法用于在 Monge 矩阵中快速查找每行的最小值

```

问题描述：

给定一个 Monge 矩阵，快速找到每行的最小值位置

解题思路：

1. Monge 矩阵满足性质： $\text{matrix}[i][j] + \text{matrix}[i+1][j+1] \leq \text{matrix}[i][j+1] + \text{matrix}[i+1][j]$
2. SMAWK 算法利用这一性质，可以在 $O(m+n)$ 时间内找到每行的最小值
3. 主要步骤包括行压缩和递归求解

参数：

`matrix`: 一个 Monge 矩阵

返回：

`int[]`: 每行最小值的列索引

时间复杂度: $O(m+n)$ ，其中 m 是行数， n 是列数

空间复杂度: $O(m+n)$

*/

```
int m = matrix.length;
```

```
if (m == 0) {
```

```
    return new int[0];
```

```
}
```

```
int n = matrix[0].length;
```

// 构造行索引和列索引数组

```
List<Integer> rows = new ArrayList<>();
```

```
List<Integer> cols = new ArrayList<>();
```

```
for (int i = 0; i < m; ++i) rows.add(i);
```

```
for (int i = 0; i < n; ++i) cols.add(i);
```

```

// 调用递归实现
List<Integer> resultList = smawkRec(rows, cols, matrix);

// 转换为数组
int[] result = new int[resultList.size()];
for (int i = 0; i < resultList.size(); ++i) {
    result[i] = resultList.get(i);
}

return result;
}

// ===== 优化体系: Aliens Trick (二分约束参数+可行性 DP) =====

static class AliensTrickResult {
    double lambda;
    double value;

    public AliensTrickResult(double lambda, double value) {
        this.lambda = lambda;
        this.value = value;
    }
}

interface AliensCostFunction {
    double[] cost(double lambda); // 返回[value, constraint]
}

interface CheckFunction {
    boolean check(double constraint);
}

static AliensTrickResult aliensTrick(AliensCostFunction costFunc, CheckFunction checkFunc,
                                     double left, double right, double eps) {
/*
Aliens Trick (二分约束参数+可行性 DP)

```

问题描述:

解决带约束的优化问题，通常形如最小化总成本，同时满足某些约束条件

解题思路:

1. 将约束条件转化为参数 λ ，构造拉格朗日函数

2. 对 λ 进行二分查找，使用可行性 DP 判断当前 λ 下是否满足约束
3. 根据可行性 DP 的结果调整二分区间

参数:

costFunc: 计算带参数 λ 的成本函数，返回 [value, constraint] 数组
 checkFunc: 检查当前解是否满足约束的函数
 left: 二分左边界
 right: 二分右边界
 eps: 精度要求

返回:

AliensTrickResult: 包含最优参数 λ 和对应最优解的结果类

时间复杂度: $O(\log((right-left)/eps) * T(DP))$ ，其中 $T(DP)$ 是一次 DP 的时间复杂度
*/

```
double bestLambda = left;
double bestValue = 0.0;
```

```
while (right - left > eps) {
    double mid = (left + right) / 2;
    // 计算当前参数下的解和约束值
    double[] result = costFunc.cost(mid);
    double currentValue = result[0];
    double constraintValue = result[1];

    if (checkFunc.check(constraintValue)) {
        // 满足约束，尝试更小的参数
        right = mid;
        bestLambda = mid;
        bestValue = currentValue;
    } else {
        // 不满足约束，需要增大参数
        left = mid;
    }
}
```

```
return new AliensTrickResult(bestLambda, bestValue);
}
```

// 重载，提供默认精度

```
static AliensTrickResult aliensTrick(AliensCostFunction costFunc, CheckFunction checkFunc,
                                     double left, double right) {
    return aliensTrick(costFunc, checkFunc, left, right, 1e-7);
```

```
}
```

```
// ===== 图上 DP→最短路：分层图建模 =====
```

```
static int layeredGraphDijkstra(int n, int m, int[][] edges, int k) {  
    /*  
     * 分层图 Dijkstra 算法  
     */
```

问题描述：

给定一个图，允许最多使用 k 次特殊操作（如跳跃、免费通行等），求最短路径

解题思路：

1. 构建分层图，每层代表使用不同次数的特殊操作
2. 对于每个节点 u ，在第 i 层表示到达 u 时已经使用了 i 次特殊操作
3. 使用 Dijkstra 算法在分层图上寻找最短路径

参数：

n : 节点数量

m : 边的数量

$edges$: 边的列表，每个元素为 $[u, v, w]$ 表示 u 到 v 的权为 w 的边

k : 允许使用的特殊操作次数

返回：

int: 从节点 0 到节点 $n-1$ 的最短路径长度

时间复杂度: $O((n*k + m*k) \log(n*k))$

空间复杂度: $O(n*k + m*k)$

*/

// 构建分层图的邻接表

```
List<List<int[]>> graph = new ArrayList<>();  
for (int i = 0; i < n * (k + 1); ++i) {  
    graph.add(new ArrayList<>());  
}
```

// 添加普通边（不使用特殊操作）

```
for (int[] edge : edges) {  
    int u = edge[0];  
    int v = edge[1];  
    int w = edge[2];  
    for (int i = 0; i <= k; ++i) {  
        int from = u + i * n;  
        graph.get(from).add(new int[]{v + i * n, w});  
    }  
}
```

```
}
```

```
// 添加使用特殊操作的边（如果允许的话）
for (int[] edge : edges) {
    int u = edge[0];
    int v = edge[1];
    for (int i = 0; i < k; ++i) {
        // 这里假设特殊操作可以免费通行（权为 0），具体根据问题调整
        int from = u + i * n;
        graph.get(from).add(new int[]{v + (i + 1) * n, 0});
    }
}
```

```
// Dijkstra 算法
int[] dist = new int[n * (k + 1)];
Arrays.fill(dist, Integer.MAX_VALUE);
dist[0] = 0; // 假设起点是节点 0
// 使用优先队列，按距离排序
PriorityQueue<int[]> heap = new PriorityQueue<>((a, b) -> a[0] - b[0]);
heap.offer(new int[]{0, 0}); // (距离, 节点)
```

```
while (!heap.isEmpty()) {
    int[] top = heap.poll();
    int d = top[0];
    int u = top[1];

    if (d > dist[u]) {
        continue;
    }

    for (int[] edge : graph.get(u)) {
        int v = edge[0];
        int w = edge[1];
        if (dist[v] > d + w) {
            dist[v] = d + w;
            heap.offer(new int[]{dist[v], v});
        }
    }
}
```

```
// 取所有层中到达终点的最小值
int result = Integer.MAX_VALUE;
for (int i = 0; i <= k; ++i) {
```

```
        result = Math.min(result, dist[n - 1 + i * n]);
    }

    return result;
}

// ===== 冷门模型：期望 DP 遇环的方程组解（高斯消元） =====
```

```
static double[] gaussianElimination(double[][] matrix) {
```

```
/*
```

```
高斯消元法求解线性方程组
```

问题描述：

求解形如 $Ax = b$ 的线性方程组

解题思路：

1. 构建增广矩阵
2. 进行高斯消元，将矩阵转化为行阶梯形
3. 回代求解

参数：

matrix：增广矩阵，每行最后一个元素是 b 的值

返回：

double[]：方程组的解

时间复杂度： $O(n^3)$

空间复杂度： $O(n^2)$

```
*/
```

```
int n = matrix.length;
final double eps = 1e-9;
```

```
// 高斯消元过程
```

```
for (int i = 0; i < n; ++i) {
    // 找到主元行（当前列中绝对值最大的行）
    int maxRow = i;
    for (int j = i; j < n; ++j) {
        if (Math.abs(matrix[j][i]) > Math.abs(matrix[maxRow][i])) {
            maxRow = j;
        }
    }
}
```

```
// 交换主元行和当前行
```

```

        double[] temp = matrix[i];
        matrix[i] = matrix[maxRow];
        matrix[maxRow] = temp;

        // 如果主元为 0, 方程组可能有无穷多解或无解
        if (Math.abs(matrix[i][i]) < eps) {
            continue;
        }

        // 消元过程
        for (int j = i + 1; j < n; ++j) {
            double factor = matrix[j][i] / matrix[i][i];
            for (int k = i; k <= n; ++k) {
                matrix[j][k] -= factor * matrix[i][k];
            }
        }
    }

    // 回代求解
    double[] x = new double[n];
    for (int i = n - 1; i >= 0; --i) {
        x[i] = matrix[i][n];
        for (int j = i + 1; j < n; ++j) {
            x[i] -= matrix[i][j] * x[j];
        }
        x[i] /= matrix[i][i];
    }

    return x;
}

static double[] expectationDPWithCycles(int n, List<List<double[]>> transitions) {
/*
期望 DP 处理有环情况（使用高斯消元）

```

问题描述:

在有环的状态转移图中计算期望

解题思路:

1. 对于每个状态，建立期望方程
2. 使用高斯消元求解方程组

参数:

n: 状态数量

transitions: 转移概率列表, transitions[i]是一个列表, 每个元素为[j, p]表示从 i 转移到 j 的概率为 p

返回:

double[]: 每个状态的期望值

时间复杂度: $O(n^3)$

空间复杂度: $O(n^2)$

*/

// 构建线性方程组的增广矩阵

```
double[][] matrix = new double[n][n + 1];
```

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

```
    matrix[i][i] = 1.0; // 方程左边: E[i] - sum(p_ij * E[j]) = cost[i]
```

// 假设每个状态的代价为 1, 具体根据问题调整

```
double cost = 1.0;
```

```
matrix[i][n] = cost;
```

```
for (double[] transition : transitions.get(i)) {
```

```
    int j = (int) transition[0];
```

```
    double p = transition[1];
```

```
    if (i != j) { // 避免自环的特殊处理
```

```
        matrix[i][j] -= p;
```

```
}
```

```
}
```

```
}
```

// 使用高斯消元求解

```
return gaussianElimination(matrix);
```

```
}
```

// ====== 冷门模型: 插头 DP (轮廓线 DP) ======

```
static int plugDP(int[][] grid) {
```

```
/*
```

插头 DP (轮廓线 DP) 示例: 求网格中哈密顿回路的数量

问题描述:

给定一个网格, 求其中哈密顿回路的数量

解题思路:

1. 使用轮廓线 DP，记录当前处理到的位置和轮廓线状态
2. 插头表示连接的状态，通常用二进制表示
3. 使用哈希表优化空间复杂度

参数：

grid：网格，1 表示可通行，0 表示障碍物

返回：

int：哈密顿回路的数量

时间复杂度： $O(n*m*4^{\min(n,m)})$

空间复杂度： $O(4^{\min(n,m)})$

*/

```
int n = grid.length;
```

```
if (n == 0) {
```

```
    return 0;
```

```
}
```

```
int m = grid[0].length;
```

// 使用哈希表优化

```
Map<Long, Integer> dp = new HashMap<>();
```

// 初始状态：左上角没有插头

```
dp.put(0L, 1);
```

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

// 新的一行开始，需要将状态左移一位

```
Map<Long, Integer> newDp = new HashMap<>();
```

```
for (Map.Entry<Long, Integer> entry : dp.entrySet()) {
```

```
    long state = entry.getKey();
```

```
    int cnt = entry.getValue();
```

// 左移一位，移除最左边的插头

```
    long newState = state << 1;
```

```
    newDp.put(newState, newDp.getOrDefault(newState, 0) + cnt);
```

```
}
```

```
dp = newDp;
```

```
for (int j = 0; j < m; ++j) {
```

```
    Map<Long, Integer> newDp2 = new HashMap<>();
```

```
    for (Map.Entry<Long, Integer> entry : dp.entrySet()) {
```

```
        long state = entry.getKey();
```

```
        int cnt = entry.getValue();
```

```

// 当前位置左边和上边的插头状态
int left = (int) ((state >> (2 * j)) & 3);
int up = (int) ((state >> (2 * (j + 1))) & 3);

// 如果当前位置是障碍物，跳过
if (grid[i][j] == 0) {
    // 只有当左右插头都不存在时才合法
    if (left == 0 && up == 0) {
        newDp2.put(state, newDp2.getOrDefault(state, 0) + cnt);
    }
    continue;
}

// 处理各种插头组合情况
// 1. 没有左插头和上插头
if (left == 0 && up == 0) {
    // 只能创建新的插头对（用于回路的开始）
    if (i < n - 1 && j < m - 1 && grid[i+1][j] == 1 && grid[i][j+1] == 1) {
        long newState = state | (1L << (2 * j)) | (2L << (2 * (j + 1)));
        newDp2.put(newState, newDp2.getOrDefault(newState, 0) + cnt);
    }
}

// 2. 只有左插头
else if (left != 0 && up == 0) {
    // 向下延伸
    if (i < n - 1 && grid[i+1][j] == 1) {
        newDp2.put(state, newDp2.getOrDefault(state, 0) + cnt);
    }
    // 向右延伸
    if (j < m - 1 && grid[i][j+1] == 1) {
        long newState = (state & ~(3L << (2 * j))) | (left << (2 * (j + 1)));
        newDp2.put(newState, newDp2.getOrDefault(newState, 0) + cnt);
    }
}

// 3. 只有上插头
else if (left == 0 && up != 0) {
    // 向右延伸
    if (j < m - 1 && grid[i][j+1] == 1) {
        newDp2.put(state, newDp2.getOrDefault(state, 0) + cnt);
    }
    // 向下延伸
}

```

```

        if (i < n - 1 && grid[i+1][j] == 1) {
            long newState = (state & ~(3L << (2 * (j + 1)))) | (up << (2 * j));
            newDp2.put(newState, newDp2.getOrDefault(newState, 0) + cnt);
        }
    }

    // 4. 同时有左插头和上插头
    else {
        // 合并插头
        long newState = (state & ~(3L << (2 * j))) & ~(3L << (2 * (j + 1)));

        // 如果是形成回路的最后一步
        if (left == up) {
            // 检查是否所有插头都已连接
            if (newState == 0 && i == n - 1 && j == m - 1) {
                newDp2.put(newState, newDp2.getOrDefault(newState, 0) + cnt);
            }
        } else {
            // 合并两个不同的插头
            newDp2.put(newState, newDp2.getOrDefault(newState, 0) + cnt);
        }
    }

    dp = newDp2;
}

}

// 最终状态应该是没有任何插头（形成回路）
return dp.getOrDefault(0L, 0);
}

// ====== 冷门模型：树上背包的优化 ======

private static void dfsTreeKnapsack(int u, int parent, int capacity,
                                    List<List<Integer>> tree, int[] weights,
                                    int[] values, int[][] dp, int[] size) {
    // 初始化当前节点
    size[u] = 1;
    if (weights[u] <= capacity) {
        dp[u][weights[u]] = values[u];
    }
}

```

```

// 对每个子节点，按照子树大小排序，小的先合并
List<int[]> children = new ArrayList<>();
for (int v : tree.get(u)) {
    if (v != parent) {
        dfsTreeKnapsack(v, u, capacity, tree, weights, values, dp, size);
        children.add(new int[]{size[v], v});
    }
}

// 按子树大小排序
Collections.sort(children, (a, b) -> a[0] - b[0]);

for (int[] child : children) {
    int sz = child[0];
    int v = child[1];
    // 逆序遍历容量，避免重复计算
    for (int i = Math.min(size[u], capacity); i >= 0; --i) {
        if (dp[u][i] == 0 && i != 0) continue;
        for (int j = 1; j <= Math.min(sz, capacity - i); ++j) {
            if (dp[v][j] > 0 && i + j <= capacity) {
                dp[u][i + j] = Math.max(dp[u][i + j], dp[u][i] + dp[v][j]);
            }
        }
    }
}

// 更新子树大小
size[u] += sz;
}

static int treeKnapsackOptimized(int root, int capacity, List<List<Integer>> tree,
                                  int[] weights, int[] values) {
/*
树上背包的优化实现（小到大合并）
*/
}

```

问题描述：

在树上选择一些节点，使得总重量不超过容量，且总价值最大

解题思路：

1. 使用后序遍历处理子树
2. 使用小到大合并的策略优化复杂度
3. 对于每个节点，维护一个容量为 capacity 的背包

参数:

root: 根节点
capacity: 背包容量
tree: 树的邻接表
weights: 每个节点的重量
values: 每个节点的价值

返回:

int: 最大价值

时间复杂度: $O(n * capacity^2)$, 但通过小到大合并可以降低常数

空间复杂度: $O(n * capacity)$

*/

```
int n = tree.size();
```

```
int[][] dp = new int[n][capacity + 1];
```

```
int[] size = new int[n];
```

// 深度优先搜索处理子树

```
dfsTreeKnapsack(root, -1, capacity, tree, weights, values, dp, size);
```

// 返回根节点的最大价值

```
int maxVal = 0;
```

```
for (int val : dp[root]) {
```

```
    maxVal = Math.max(maxVal, val);
```

```
}
```

```
return maxVal;
```

```
}
```

// ===== 补充题目与应用 =====

// 以下是一些使用上述高级 DP 技术的经典题目及其代码实现

// 1. 编辑距离问题 (LeetCode 72)

```
static int editDistance(String word1, String word2) {
```

```
/*
```

LeetCode 72. 编辑距离

题目链接: <https://leetcode-cn.com/problems/edit-distance/>

问题描述:

给你两个单词 word1 和 word2, 计算出将 word1 转换成 word2 所使用的最少操作数。

你可以对一个单词进行如下三种操作: 插入一个字符、删除一个字符、替换一个字符。

解题思路:

使用二维 DP, $dp[i][j]$ 表示 word1 的前 i 个字符转换为 word2 的前 j 个字符所需的最少操作数。

```

时间复杂度: O(m*n)
空间复杂度: O(m*n)
*/
int m = word1.length();
int n = word2.length();
// dp[i][j]表示word1 的前 i 个字符转换为word2 的前 j 个字符所需的最少操作数
int[][] dp = new int[m + 1][n + 1];

// 初始化边界
for (int i = 0; i <= m; ++i) {
    dp[i][0] = i;
}
for (int j = 0; j <= n; ++j) {
    dp[0][j] = j;
}

// 动态规划填表
for (int i = 1; i <= m; ++i) {
    for (int j = 1; j <= n; ++j) {
        if (word1.charAt(i - 1) == word2.charAt(j - 1)) {
            dp[i][j] = dp[i - 1][j - 1];
        } else {
            dp[i][j] = Math.min(Math.min(dp[i - 1][j], dp[i][j - 1]), dp[i - 1][j - 1]) +
1;
        }
    }
}

return dp[m][n];
}

// 2. 最长递增子序列 (LeetCode 300)
static int lengthOfLIS(int[] nums) {
/*
LeetCode 300. 最长递增子序列
题目链接: https://leetcode-cn.com/problems/longest-increasing-subsequence/

```

问题描述:

给你一个整数数组 `nums`，找到其中最长严格递增子序列的长度。

解题思路:

使用贪心 + 二分查找优化的 DP 方法。

tails[i]表示长度为 i+1 的递增子序列的末尾元素的最小值。

```
时间复杂度: O(n log n)
空间复杂度: O(n)

*/
if (nums == null || nums.length == 0) {
    return 0;
}

List<Integer> tails = new ArrayList<>();
for (int num : nums) {
    // 二分查找 num 应该插入的位置
    int left = 0, right = tails.size();
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (tails.get(mid) >= num) {
            right = mid;
        } else {
            left = mid + 1;
        }
    }
    if (left == tails.size()) {
        tails.add(num);
    } else {
        tails.set(left, num);
    }
}

return tails.size();
}

// 3. 背包问题变种 - 完全背包 (LeetCode 322)
static int coinChange(int[] coins, int amount) {
    /*
    LeetCode 322. 零钱兑换
    题目链接: https://leetcode-cn.com/problems/coin-change/

```

问题描述:

给你一个整数数组 coins，表示不同面额的硬币；以及一个整数 amount，表示总金额。

计算并返回可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额，返回 -1。

解题思路:

使用完全背包的思想， $dp[i]$ 表示凑成金额 i 所需的最少硬币数。

```
时间复杂度: O(amount * n)
空间复杂度: O(amount)
*/
// 初始化 dp 数组为无穷大
int[] dp = new int[amount + 1];
Arrays.fill(dp, Integer.MAX_VALUE);
dp[0] = 0; // 凑成金额 0 需要 0 个硬币

for (int coin : coins) {
    for (int i = coin; i <= amount; ++i) {
        if (dp[i - coin] != Integer.MAX_VALUE) {
            dp[i] = Math.min(dp[i], dp[i - coin] + 1);
        }
    }
}

return dp[amount] == Integer.MAX_VALUE ? -1 : dp[amount];
}
```

// 4. 矩阵链乘法（区间 DP 的经典应用）

```
static class MatrixChainResult {
    int[][] dp;
    int[][] s;

    public MatrixChainResult(int[][] dp, int[][] s) {
        this.dp = dp;
        this.s = s;
    }
}
```

```
static MatrixChainResult matrixChainOrder(int[] p) {
```

```
/*
矩阵链乘法问题
题目来源: 算法导论
```

问题描述:

给定一系列矩阵，计算乘法顺序使得标量乘法的次数最少。

解题思路:

使用区间 DP， $dp[i][j]$ 表示计算第 i 到第 j 个矩阵的乘积所需的最少标量乘法次数。
可以使用 Knuth 优化进一步降低时间复杂度。

```

时间复杂度: O(n^3)
空间复杂度: O(n^2)
*/
int n = p.length - 1; // 矩阵的个数
// dp[i][j]表示计算第 i 到第 j 个矩阵的乘积所需的最少标量乘法次数
int[][] dp = new int[n + 1][n + 1];
// s[i][j]记录最优分割点
int[][] s = new int[n + 1][n + 1];

// 枚举区间长度
for (int length = 2; length <= n; ++length) {
    for (int i = 1; i + length - 1 <= n; ++i) {
        int j = i + length - 1;
        dp[i][j] = Integer.MAX_VALUE;
        // 枚举分割点
        for (int k = i; k < j; ++k) {
            // 计算当前分割点的代价
            int cost = dp[i][k] + dp[k + 1][j] + p[i - 1] * p[k] * p[j];
            if (cost < dp[i][j]) {
                dp[i][j] = cost;
                s[i][j] = k;
            }
        }
    }
}

return new MatrixChainResult(dp, s);
}

// 5. 旅行商问题 (TSP) 的 DP 实现
static int travelingSalesmanProblem(int[][] graph) {
/*
旅行商问题
题目来源: 算法竞赛经典问题

问题描述:
给定一个完全图, 找到一条访问每个城市恰好一次并返回起点的最短路径。

解题思路:
使用状态压缩 DP, dp[mask][u] 表示访问过的城市集合为 mask, 当前在城市 u 时的最短路径长度。
时间复杂度: O(n^2 * 2^n)

```

```

空间复杂度: O(n * 2^n)
*/
int n = graph.length;
// dp[mask][u]表示访问过的城市集合为 mask, 当前在城市 u 时的最短路径长度
int[][] dp = new int[1 << n][n];
for (int[] row : dp) {
    Arrays.fill(row, Integer.MAX_VALUE);
}

// 初始状态: 只访问了起点, 路径长度为 0
for (int i = 0; i < n; ++i) {
    dp[1 << i][i] = 0;
}

// 枚举所有可能的状态
for (int mask = 1; mask < (1 << n); ++mask) {
    // 枚举当前所在的城市
    for (int u = 0; u < n; ++u) {
        if ((mask & (1 << u)) == 0) {
            continue;
        }
        // 枚举下一个要访问的城市
        for (int v = 0; v < n; ++v) {
            if ((mask & (1 << v)) != 0) {
                continue;
            }
            int newMask = mask | (1 << v);
            if (dp[mask][u] != Integer.MAX_VALUE && graph[u][v] != Integer.MAX_VALUE) {
                dp[newMask][v] = Math.min(dp[newMask][v], dp[mask][u] + graph[u][v]);
            }
        }
    }
}

// 找到最短的回路
int result = Integer.MAX_VALUE;
for (int u = 0; u < n; ++u) {
    if (dp[(1 << n) - 1][u] != Integer.MAX_VALUE && graph[u][0] != Integer.MAX_VALUE) {
        result = Math.min(result, dp[(1 << n) - 1][u] + graph[u][0]);
    }
}

return result;

```

```
}
```

```
// 6. 区间 DP: 最优三角剖分
```

```
static int minimumScoreTriangulation(int[] values) {
```

```
/*
```

```
LeetCode 1039. 多边形三角剖分的最低得分
```

```
题目链接: https://leetcode-cn.com/problems/minimum-score-triangulation-of-polygon/
```

问题描述:

给定一个凸多边形，将其三角剖分，使得所有三角形的顶点乘积之和最小。

解题思路:

使用区间 DP， $dp[i][j]$ 表示从顶点 i 到顶点 j 的多边形三角剖分的最小得分。

时间复杂度: $O(n^3)$

空间复杂度: $O(n^2)$

```
*/
```

```
int n = values.length;
```

```
// dp[i][j] 表示从顶点 i 到顶点 j 的多边形三角剖分的最小得分
```

```
int[][] dp = new int[n][n];
```

```
// 枚举区间长度
```

```
for (int length = 3; length <= n; ++length) {
```

```
    for (int i = 0; i + length - 1 < n; ++i) {
```

```
        int j = i + length - 1;
```

```
        dp[i][j] = Integer.MAX_VALUE;
```

```
        // 枚举中间点
```

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

```
            dp[i][j] = Math.min(dp[i][j],
```

```
                               dp[i][k] + dp[k][j] + values[i] * values[k] * values[j]);
```

```
}
```

```
}
```

```
}
```

```
return dp[0][n - 1];
```

```
}
```

```
// 7. 博弈 DP: 石子游戏
```

```
static boolean stoneGame(int[] piles) {
```

```
/*
```

```
LeetCode 877. 石子游戏
```

```
题目链接: https://leetcode-cn.com/problems/stone-game/
```

问题描述:

给定一个表示石子堆的数组，两个玩家轮流从两端取石子，每次只能取一个，取到最后一个石子的人获胜。

判断先手是否必胜。

解题思路:

使用区间 DP， $dp[i][j]$ 表示在区间 $[i, j]$ 中，先手能获得的最大净胜分。

时间复杂度: $O(n^2)$

空间复杂度: $O(n^2)$

*/

```
int n = piles.length;
```

// $dp[i][j]$ 表示在区间 $[i, j]$ 中，先手能获得的最大净胜分

```
int[][] dp = new int[n][n];
```

// 初始化单个石子堆

```
for (int i = 0; i < n; ++i) {  
    dp[i][i] = piles[i];  
}
```

// 枚举区间长度

```
for (int length = 2; length <= n; ++length) {  
    for (int i = 0; i + length - 1 < n; ++i) {  
        int j = i + length - 1;  
        // 先手可以选择取左边或右边  
        dp[i][j] = Math.max(piles[i] - dp[i + 1][j], piles[j] - dp[i][j - 1]);  
    }  
}
```

// 先手净胜分大于 0 则必胜

```
return dp[0][n - 1] > 0;
```

}

// 8. 数位 DP: 统计 1 出现的次数

```
static int countDigitOne(int n) {
```

/*

LeetCode 233. 数字 1 的个数

题目链接: <https://leetcode-cn.com/problems/number-of-digit-one/>

问题描述:

给定一个整数 n ，计算所有小于等于 n 的非负整数中数字 1 出现的个数。

解题思路:

使用数位 DP，逐位处理每一位上 1 出现的次数。

```
时间复杂度: O(log n)
空间复杂度: O(log n)
*/
if (n <= 0) {
    return 0;
}

String s = String.valueOf(n);
int length = s.length();
int count = 0;

// 逐位处理
for (int i = 0; i < length; ++i) {
    long high = 0;
    if (i > 0) {
        high = Long.parseLong(s.substring(0, i));
    }
    int current = s.charAt(i) - '0';
    long low = 0;
    if (i < length - 1) {
        low = Long.parseLong(s.substring(i + 1));
    }
    long digit = (long) Math.pow(10, length - i - 1);

    if (current == 0) {
        // 当前位为 0, 高位决定
        count += high * digit;
    } else if (current == 1) {
        // 当前位为 1, 高位+低位+1
        count += high * digit + low + 1;
    } else {
        // 当前位大于 1, 高位+1
        count += (high + 1) * digit;
    }
}

return count;
}

// 9. 树形 DP: 打家劫舍 III
static class TreeNode {
```

```

int val;
TreeNode left;
TreeNode right;
TreeNode(int x) { val = x; }

}

private static int[] robDFS(TreeNode node) {
    if (node == null) {
        return new int[]{0, 0};
    }

    int[] left = robDFS(node.left);
    int[] right = robDFS(node.right);

    // 偷当前节点，不能偷子节点
    int robCurrent = node.val + left[1] + right[1];
    // 不偷当前节点，可以选择偷或不偷子节点
    int notRobCurrent = Math.max(left[0], left[1]) + Math.max(right[0], right[1]);

    return new int[]{robCurrent, notRobCurrent};
}

static int rob(TreeNode root) {
/*
LeetCode 337. 打家劫舍 III
题目链接: https://leetcode-cn.com/problems/house-robber-iii/
*/
}

```

问题描述:

在上次打劫完一条街道之后和一圈房屋后，小偷又发现了一个新的可行窃的地区。

这个地区只有一个入口，我们称之为“根”。除了“根”之外，每栋房子有且只有一个“父”房子与之相连。

一番侦察之后，聪明的小偷意识到“这个地方的所有房屋的排列类似于一棵二叉树”。

如果两个直接相连的房子在同一天晚上被打劫，房屋将自动报警。

计算在不触动警报的情况下，小偷一晚能够盗取的最高金额。

解题思路:

使用树形 DP，对于每个节点，维护两个状态：偷或不偷。

时间复杂度: $O(n)$

空间复杂度: $O(h)$ ， h 为树的高度

*/

```

int[] result = robDFS(root);
return Math.max(result[0], result[1]);

```

```
}
```

```
// 10. 状态压缩 DP: 蒙斯特曼问题
static int monsterGame(int[][] grid) {
    /*
    蒙斯特曼问题
    题目来源: 算法竞赛问题
```

问题描述:

在网格中放置怪物，使得任何两个怪物都不在同一行、同一列或对角线上。

解题思路:

使用状态压缩 DP， $dp[i][mask]$ 表示处理到第 i 行，已放置的列的状态为 $mask$ 时的方案数。

时间复杂度: $O(n * 2^n)$

空间复杂度: $O(2^n)$

*/

```
int n = grid.length;
// dp[i][mask] 表示处理到第 i 行，已放置的列的状态为 mask 时的方案数
long[] dp = new long[1 << n];
dp[0] = 1;
```

```
for (int i = 0; i < n; ++i) {
    long[] newDp = new long[1 << n];
    for (int mask = 0; mask < (1 << n); ++mask) {
        if (dp[mask] == 0) {
            continue;
        }
        // 枚举所有可能的放置位置
        for (int j = 0; j < n; ++j) {
            // 检查是否可以在(i, j)放置怪物
            if ((mask & (1 << j)) == 0 && grid[i][j] == 1) {
                // 检查对角线
                boolean valid = true;
                for (int k = 0; k < i; ++k) {
                    if ((mask & (1 << k)) != 0 && Math.abs(k - j) == i - k) {
                        valid = false;
                        break;
                    }
                }
                if (valid) {
                    newDp[mask | (1 << j)] += dp[mask];
                }
            }
        }
    }
}
```

```

        }
    }
}

dp = newDp;
}

return (int) dp[(1 << n) - 1];
}

```

// 11. 高维 DP: 三维背包

```
static int threeDimensionKnapsack(int n, int[] capacity, int[][] items) {
```

```
/*

```

三维背包问题

题目来源: 算法竞赛问题

问题描述:

有 n 个物品，每个物品有体积、重量、价值三个属性，背包有体积和重量两个限制，求最大价值。

解题思路:

使用三维 DP， $dp[i][j][k]$ 表示前 i 个物品，体积为 j，重量为 k 时的最大价值。

时间复杂度: $O(n * V * W)$

空间复杂度: $O(n * V * W)$

```
*/
```

```
int V = capacity[0];
int W = capacity[1];
// 初始化 dp 数组
int[][][] dp = new int[n + 1][V + 1][W + 1];
```

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

```
    int v = items[i-1][0];
    int w = items[i-1][1];
    int val = items[i-1][2];
    for (int j = 0; j <= V; ++j) {
        for (int k = 0; k <= W; ++k) {
```

// 不选当前物品

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

// 选当前物品（如果有足够的空间）

```
if (j >= v && k >= w) {
```

```
dp[i][j][k] = Math.max(dp[i][j][k], dp[i-1][j-v][k-w] + val);
```

```
}
```

```
}
```

```
}
```

```
    }

    return dp[n][V][W];
}
```

```
// 12. 斜率优化 DP 示例
static class ConvexHullTrick {
    /*
    凸包优化技巧示例
    题目来源：算法竞赛问题
```

问题描述：

当状态转移方程形如 $dp[i] = \min\{dp[j] + a[i] * b[j] + c\}$ 时，可以使用凸包优化。

解题思路：

将转移方程转换为直线的形式，维护凸包以快速查询最小值。

时间复杂度：O(n)

空间复杂度：O(n)

*/

```
static class Line {
    long k, b;
    Line(long k, long b) {
        this.k = k;
        this.b = b;
    }
}
```

```
Deque<Line> dq = new LinkedList<>();
```

// 添加一条直线 $y = kx + b$

```
public void addLine(long k, long b) {
```

// 当队列中至少有两条直线时，检查是否需要删除末尾的直线

```
while (dq.size() >= 2) {
```

```
    Line l1 = dq.get(dq.size() - 2);
```

```
    Line l2 = dq.getLast();
```

// 判断直线 l1 和 l2 的交点是否在 l2 和新直线的交点右侧

```
    if ((l2.b - l1.b) * (k - l2.k) >= (b - l2.b) * (l2.k - l1.k)) {
```

```
        dq.removeLast();
```

```
    } else {
```

```
        break;
```

```
}
```

```

        }

        dq.addLast(new Line(k, b));
    }

    // 查询 x 处的最小值
    public long query(long x) {
        // 如果队列中至少有两条直线，且第一条直线在 x 处的值大于第二条，删除第一条
        while (dq.size() >= 2) {
            Line l1 = dq.getFirst();
            Line l2 = dq.get(1);
            if (l1.k * x + l1.b >= l2.k * x + l2.b) {
                dq.removeFirst();
            } else {
                break;
            }
        }

        if (dq.isEmpty()) {
            return Long.MAX_VALUE;
        }

        Line l = dq.getFirst();
        return l.k * x + l.b;
    }
}

// ===== 优化体系: Knuth 优化 =====

// Knuth 优化用于优化形如 dp[i][j] = min{dp[i][k] + dp[k+1][j]} + w(i, j) 的 DP
// 当满足四边形不等式时，最优转移点单调

static class KnuthOptimizationResult {
    int[][] dp;
    int[][] opt;

    public KnuthOptimizationResult(int[][] dp, int[][] opt) {
        this.dp = dp;
        this.opt = opt;
    }
}

interface CostFunction {
    int apply(int i, int j);
}

```

```
public static KnuthOptimizationResult knuthOptimization(int n, CostFunction costFunc) {  
    /*  
     * Knuth 优化的 DP 算法  
    */
```

问题描述:

解决区间 DP 问题，其中状态转移方程满足四边形不等式

解题思路:

1. 使用 Knuth 优化将时间复杂度从 $O(n^3)$ 降低到 $O(n^2)$
2. 维护最优转移点数组 $opt[i][j]$ ，表示计算 $dp[i][j]$ 时的最优 k 值
3. 根据 $opt[i][j-1] \leq opt[i][j] \leq opt[i+1][j]$ 的性质进行剪枝

参数:

n : 区间长度

$costFunc$: 计算区间 (i, j) 代价的函数

返回:

$KnuthOptimizationResult$: 包含 dp 数组和 opt 数组的结果类

时间复杂度: $O(n^2)$

空间复杂度: $O(n^2)$

*/

// 初始化 dp 和 opt 数组

```
int[][] dp = new int[n + 1][n + 1];  
int[][] opt = new int[n + 1][n + 1];
```

// 初始化长度为 1 的区间

```
for (int i = 1; i <= n; ++i) {  
    dp[i][i] = 0;  
    opt[i][i] = i;  
}
```

// 枚举区间长度

```
for (int length = 2; length <= n; ++length) {  
    // 枚举起始点  
    for (int i = 1; i + length - 1 <= n; ++i) {  
        int j = i + length - 1;  
        // 初始化为无穷大  
        dp[i][j] = Integer.MAX_VALUE;  
        // 根据 Knuth 优化的性质，最优 k 在  $opt[i][j-1]$  到  $opt[i+1][j]$  之间  
        int upperK = (i + 1 <= j) ? opt[i + 1][j] : j - 1;  
        for (int k = opt[i][j - 1]; k <= Math.min(upperK, j - 1); ++k) {  
            if (dp[i][k] != Integer.MAX_VALUE && dp[k + 1][j] != Integer.MAX_VALUE) {
```

```

        int cost = costFunc.apply(i, j);
        if (cost != Integer.MAX_VALUE) {
            int current = dp[i][k] + dp[k + 1][j] + cost;
            if (current < dp[i][j]) {
                dp[i][j] = current;
                opt[i][j] = k;
            }
        }
    }
}

return new KnuthOptimizationResult(dp, opt);
}

// ===== 优化体系: Divide & Conquer Optimization =====

private static void solveDivideConquer(int i, int l, int r, int opt_l, int opt_r,
                                       int[][] dp, CostFunction costFunc) {
    /*
    计算 dp[i][l..r], 其中最优转移点在 opt_l..opt_r 之间
    */
    if (l > r) {
        return;
    }

    int mid = (l + r) / 2;
    int best_k = opt_l;

    // 在 opt_l 到 Math.min(mid, opt_r) 之间寻找最优 k
    for (int k = opt_l; k <= Math.min(mid, opt_r); ++k) {
        if (dp[i - 1][k] != Integer.MAX_VALUE) {
            int cost = costFunc.apply(k, mid);
            if (cost != Integer.MAX_VALUE) {
                int current = dp[i - 1][k] + cost;
                if (current < dp[i][mid]) {
                    dp[i][mid] = current;
                    best_k = k;
                }
            }
        }
    }
}

```

```

// 递归处理左右子区间
solveDivideConquer(i, 1, mid - 1, opt_l, best_k, dp, costFunc);
solveDivideConquer(i, mid + 1, r, best_k, opt_r, dp, costFunc);
}

public static int[][] divideConquerOptimization(int n, int m, CostFunction costFunc) {
/*
Divide & Conquer Optimization (分治优化)

```

问题描述:

解决形如 $dp[i][j] = \min\{dp[i-1][k] + cost(k, j)\}$, 其中 $k < j$

当转移满足决策单调性时使用

解题思路:

1. 利用决策单调性, 使用分治法优化 DP
2. 对于 $dp[i][j]$, 当 i 固定时, 最优转移点 k 随着 j 的增加而单调不减
3. 使用分治的方式计算每个区间的最优决策

参数:

n: 维度 1
m: 维度 2
costFunc: 计算 $cost(k, j)$ 的函数

返回:

int[][]: dp 数组

时间复杂度: $O(n*m \log m)$

空间复杂度: $O(n*m)$

*/

// 初始化 dp 数组

```

int[][] dp = new int[n + 1][m + 1];
for (int i = 0; i <= n; ++i) {
    for (int j = 0; j <= m; ++j) {
        dp[i][j] = Integer.MAX_VALUE;
    }
}
dp[0][0] = 0;

```

// 对每个 i 应用分治优化

```

for (int i = 1; i <= n; ++i) {
    solveDivideConquer(i, 1, m, 0, m, dp, costFunc);
}

```

```

    return dp;
}

// ===== 优化体系: SMAWK 算法 (行最小查询) =====

private static List<Integer> reduceRows(List<Integer> rows, int[][] matrix) {
    /*行压缩: 只保留可能成为最小值的行*/
    Stack<Integer> stk = new Stack<>();
    for (int i : rows) {
        while (stk.size() >= 2) {
            int j1 = stk.pop();
            int j2 = stk.peek();
            stk.push(j1); // 恢复栈状态

            // 比较两个行在列 stk.size()-1 处的值
            if (matrix[j2][stk.size() - 1] <= matrix[i][stk.size() - 1]) {
                break;
            } else {
                stk.pop();
            }
        }
        stk.push(i);
    }

    List<Integer> result = new ArrayList<>();
    while (!stk.isEmpty()) {
        result.add(stk.pop());
    }
    Collections.reverse(result);
    return result;
}

private static List<Integer> smawkRec(List<Integer> rows, List<Integer> cols, int[][] matrix)
{
    /*递归实现 SMAWK 算法*/
    if (rows.isEmpty()) {
        return new ArrayList<>();
    }

    // 行压缩
    List<Integer> reducedRows = reduceRows(rows, matrix);

```

```

// 递归求解列数为奇数的子问题
List<Integer> halfCols = new ArrayList<>();
for (int i = 1; i < cols.size(); i += 2) {
    halfCols.add(cols.get(i));
}
int[] minCols = new int[reducedRows.size()];
Arrays.fill(minCols, -1);

if (!halfCols.isEmpty()) {
    // 递归求解
    List<Integer> result = smawkRec(reducedRows, halfCols, matrix);
    // 复制结果
    for (int i = 0; i < result.size(); ++i) {
        minCols[i] = result.get(i);
    }
}

// 扩展结果到所有列
List<Integer> result = new ArrayList<>(Collections.nCopies(rows.size(), 0));
int k = 0; // minCols 的索引

for (int i = 0; i < rows.size(); ++i) {
    int row = rows.get(i);
    // 确定当前行的最小值可能在哪个区间
    int start = (i == 0) ? 0 : (k > 0 ? minCols[k - 1] : 0);
    int end = (k < minCols.length) ? minCols[k] : cols.get(cols.size() - 1);

    // 在这个区间内查找最小值
    int minValue = Integer.MAX_VALUE;
    int minCol = start;

    // 注意这里 cols 是原始列的子集，需要在 cols 中遍历
    int startIndex = cols.indexOf(start);
    int endIndex = cols.indexOf(end);
    if (startIndex != -1 && endIndex != -1) {
        for (int idx = startIndex; idx <= endIndex; ++idx) {
            int col = cols.get(idx);
            if (col < matrix[0].length && matrix[row][col] < minValue) {
                minValue = matrix[row][col];
                minCol = col;
            }
        }
    }
}

```

```

        result.set(i, minCol);

        // 如果当前行在 reducedRows 中，且不是最后一行，k 前进
        if (k < reducedRows.size() && row == reducedRows.get(k)) {
            k++;
        }
    }

    return result;
}

```

```

public static List<Integer> smawk(int[][] matrix) {
/*
SMAWK 算法用于在 Monge 矩阵中快速查找每行的最小值

```

问题描述：

给定一个 Monge 矩阵，快速找到每行的最小值位置

解题思路：

1. Monge 矩阵满足性质： $\text{matrix}[i][j] + \text{matrix}[i+1][j+1] \leq \text{matrix}[i][j+1] + \text{matrix}[i+1][j]$
2. SMAWK 算法利用这一性质，可以在 $O(m+n)$ 时间内找到每行的最小值
3. 主要步骤包括行压缩和递归求解

参数：

`matrix`: 一个 Monge 矩阵

返回：

`List<Integer>`: 每行最小值的列索引

时间复杂度: $O(m+n)$, 其中 m 是行数, n 是列数

空间复杂度: $O(m+n)$

*/

```

int m = matrix.length;
if (m == 0) {
    return new ArrayList<>();
}
int n = matrix[0].length;

// 构造行索引和列索引数组
List<Integer> rows = new ArrayList<>();
List<Integer> cols = new ArrayList<>();
for (int i = 0; i < m; ++i) {

```

```

        rows.add(i);
    }

    for (int i = 0; i < n; ++i) {
        cols.add(i);
    }

    // 调用递归实现
    return smawkRec(rows, cols, matrix);
}

// ===== 优化体系: Aliens Trick (二分约束参数+可行性 DP) =====

static class AliensTrickResult {
    double lambda;
    double value;

    public AliensTrickResult(double lambda, double value) {
        this.lambda = lambda;
        this.value = value;
    }
}

interface AliensCostFunction {
    double[] apply(double lambda); // 返回[value, constraint]
}

interface CheckFunction {
    boolean apply(double constraintValue);
}

public static AliensTrickResult aliensTrick(AliensCostFunction costFunc, CheckFunction
checkFunc,
                                            double left, double right, double eps) {
/*
 * Aliens Trick (二分约束参数+可行性 DP)
 */
}

```

问题描述:

解决带约束的优化问题，通常形如最小化总成本，同时满足某些约束条件

解题思路:

1. 将约束条件转化为参数 λ ，构造拉格朗日函数
2. 对 λ 进行二分查找，使用可行性 DP 判断当前 λ 下是否满足约束
3. 根据可行性 DP 的结果调整二分区间

参数:

costFunc: 计算带参数 λ 的成本函数, 返回 [value, constraint] 数组
checkFunc: 检查当前解是否满足约束的函数
left: 二分左边界
right: 二分右边界
eps: 精度要求

返回:

AliensTrickResult: 包含最优参数 λ 和对应最优解的结果类

时间复杂度: $O(\log((right-left)/eps) * T(DP))$, 其中 $T(DP)$ 是一次 DP 的时间复杂度

*/

```
double bestLambda = left;  
double bestValue = 0.0;
```

```
while (right - left > eps) {  
    double mid = (left + right) / 2;  
    // 计算当前参数下的解和约束值  
    double[] result = costFunc.apply(mid);  
    double currentValue = result[0];  
    double constraintValue = result[1];  
  
    if (checkFunc.apply(constraintValue)) {  
        // 满足约束, 尝试更小的参数  
        right = mid;  
        bestLambda = mid;  
        bestValue = currentValue;  
    } else {  
        // 不满足约束, 需要增大参数  
        left = mid;  
    }  
}
```

```
return new AliensTrickResult(bestLambda, bestValue);  
}
```

// 默认精度版本

```
public static AliensTrickResult aliensTrick(AliensCostFunction costFunc, CheckFunction  
checkFunc,  
                                            double left, double right) {  
    return aliensTrick(costFunc, checkFunc, left, right, 1e-7);  
}
```

```
// ===== 图上 DP→最短路：分层图建模 =====
```

```
public static int layeredGraphDijkstra(int n, int m, List<List<Integer>> edges, int k) {  
    /*  
     * 分层图 Dijkstra 算法  
     */
```

问题描述：

给定一个图，允许最多使用 k 次特殊操作（如跳跃、免费通行等），求最短路径

解题思路：

1. 构建分层图，每层代表使用不同次数的特殊操作
2. 对于每个节点 u ，在第 i 层表示到达 u 时已经使用了 i 次特殊操作
3. 使用 Dijkstra 算法在分层图上寻找最短路径

参数：

n : 节点数量

m : 边的数量

$edges$: 边的列表，每个元素为 $[u, v, w]$ 表示 u 到 v 的权为 w 的边

k : 允许使用的特殊操作次数

返回：

int: 从节点 0 到节点 $n-1$ 的最短路径长度

时间复杂度: $O((n*k + m*k) \log(n*k))$

空间复杂度: $O(n*k + m*k)$

*/

// 构建分层图的邻接表

```
List<List<List<Integer>>> graph = new ArrayList<>();  
for (int i = 0; i < n * (k + 1); ++i) {  
    graph.add(new ArrayList<>());  
}
```

// 添加普通边（不使用特殊操作）

```
for (List<Integer> edge : edges) {  
    int u = edge.get(0);  
    int v = edge.get(1);  
    int w = edge.get(2);  
    for (int i = 0; i <= k; ++i) {  
        int from = u + i * n;  
        graph.get(from).add(Arrays.asList(v + i * n, w));  
    }  
}
```

```

// 添加使用特殊操作的边（如果允许的话）
for (List<Integer> edge : edges) {
    int u = edge.get(0);
    int v = edge.get(1);
    for (int i = 0; i < k; ++i) {
        // 这里假设特殊操作可以免费通行（权为 0），具体根据问题调整
        int from = u + i * n;
        graph.get(from).add(Arrays.asList(v + (i + 1) * n, 0));
    }
}

// Dijkstra 算法
int[] dist = new int[n * (k + 1)];
Arrays.fill(dist, Integer.MAX_VALUE);
dist[0] = 0; // 假设起点是节点 0

// 使用优先队列，按距离排序
PriorityQueue<int[]> heap = new PriorityQueue<>((a, b) -> a[0] - b[0]);
heap.offer(new int[]{0, 0});

while (!heap.isEmpty()) {
    int[] current = heap.poll();
    int d = current[0];
    int u = current[1];

    if (d > dist[u]) {
        continue;
    }

    for (List<Integer> edge : graph.get(u)) {
        int v = edge.get(0);
        int w = edge.get(1);
        if (dist[v] > d + w) {
            dist[v] = d + w;
            heap.offer(new int[]{dist[v], v});
        }
    }
}

// 取所有层中到达终点的最小值
int result = Integer.MAX_VALUE;
for (int i = 0; i <= k; ++i) {

```

```
        result = Math.min(result, dist[n - 1 + i * n]);
    }

    return result;
}

// ===== 冷门模型：期望 DP 遇环的方程组解（高斯消元） =====
```

```
public static double[] gaussianElimination(double[][] matrix) {
```

```
/*
```

```
高斯消元法求解线性方程组
```

问题描述：

求解形如 $Ax = b$ 的线性方程组

解题思路：

1. 构建增广矩阵
2. 进行高斯消元，将矩阵转化为行阶梯形
3. 回代求解

参数：

matrix：增广矩阵，每行最后一个元素是 b 的值

返回：

double[]：方程组的解

时间复杂度： $O(n^3)$

空间复杂度： $O(n^2)$

```
*/
```

```
int n = matrix.length;
final double eps = 1e-9;
```

```
// 高斯消元过程
```

```
for (int i = 0; i < n; ++i) {
    // 找到主元行（当前列中绝对值最大的行）
    int maxRow = i;
    for (int j = i; j < n; ++j) {
        if (Math.abs(matrix[j][i]) > Math.abs(matrix[maxRow][i])) {
            maxRow = j;
        }
    }
}
```

```
// 交换主元行和当前行
```

```

        double[] temp = matrix[i];
        matrix[i] = matrix[maxRow];
        matrix[maxRow] = temp;

        // 如果主元为 0, 方程组可能有无穷多解或无解
        if (Math.abs(matrix[i][i]) < eps) {
            continue;
        }

        // 消元过程
        for (int j = i + 1; j < n; ++j) {
            double factor = matrix[j][i] / matrix[i][i];
            for (int k = i; k <= n; ++k) {
                matrix[j][k] -= factor * matrix[i][k];
            }
        }
    }

    // 回代求解
    double[] x = new double[n];
    for (int i = n - 1; i >= 0; --i) {
        x[i] = matrix[i][n];
        for (int j = i + 1; j < n; ++j) {
            x[i] -= matrix[i][j] * x[j];
        }
        x[i] /= matrix[i][i];
    }

    return x;
}

public static double[] expectationDPWithCycles(int n, List<List<Pair<Integer, Double>>>
transitions) {
/*
期望 DP 处理有环情况（使用高斯消元）

```

问题描述:

在有环的状态转移图中计算期望

解题思路:

1. 对于每个状态，建立期望方程
2. 使用高斯消元求解方程组

参数:

n: 状态数量

transitions: 转移概率列表, transitions[i]是一个列表, 每个元素为(j, p)表示从 i 转移到 j 的概率为 p

返回:

double[]: 每个状态的期望值

时间复杂度: $O(n^3)$

空间复杂度: $O(n^2)$

*/

// 构建线性方程组的增广矩阵

```
double[][] matrix = new double[n][n + 1];
```

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

```
    matrix[i][i] = 1.0; // 方程左边: E[i] - sum(p_ij * E[j]) = cost[i]
```

// 假设每个状态的代价为 1, 具体根据问题调整

```
double cost = 1.0;
```

```
matrix[i][n] = cost;
```

```
for (Pair<Integer, Double> transition : transitions.get(i)) {
```

```
    int j = transition.first;
```

```
    double p = transition.second;
```

```
    if (i != j) { // 避免自环的特殊处理
```

```
        matrix[i][j] -= p;
```

```
}
```

```
}
```

```
}
```

// 使用高斯消元求解

```
return gaussianElimination(matrix);
```

```
}
```

// 辅助类 Pair

```
static class Pair<A, B> {
```

```
    A first;
```

```
    B second;
```

```
    public Pair(A first, B second) {
```

```
        this.first = first;
```

```
        this.second = second;
```

```
}
```

```
}
```

```
// ===== 冷门模型：插头 DP（轮廓线 DP） =====
```

```
public static int plugDP(int[][] grid) {  
    /*  
     * 插头 DP（轮廓线 DP）示例：求网格中哈密顿回路的数量  
     */
```

问题描述：

给定一个网格，求其中哈密顿回路的数量

解题思路：

1. 使用轮廓线 DP，记录当前处理到的位置和轮廓线状态
2. 插头表示连接的状态，通常用二进制表示
3. 使用哈希表优化空间复杂度

参数：

grid: 网格，1 表示可通行，0 表示障碍物

返回：

int: 哈密顿回路的数量

时间复杂度： $O(n*m*4^{\min(n,m)})$

空间复杂度： $O(4^{\min(n,m)})$

*/

```
int n = grid.length;  
if (n == 0) {  
    return 0;  
}  
int m = grid[0].length;
```

// 使用哈希表优化

```
Map<Long, Integer> dp = new HashMap<>();
```

// 初始状态：左上角没有插头

```
dp.put(0L, 1);
```

```
for (int i = 0; i < n; ++i) {  
    // 新的一行开始，需要将状态左移一位  
    Map<Long, Integer> newDp = new HashMap<>();  
    for (Map.Entry<Long, Integer> entry : dp.entrySet()) {  
        long state = entry.getKey();  
        int cnt = entry.getValue();
```

```

// 左移一位，移除最左边的插头
long newState = state << 1;
newDp.put(newState, newDp.getOrDefault(newState, 0) + cnt);
}

dp = newDp;

for (int j = 0; j < m; ++j) {
    Map<Long, Integer> newDp2 = new HashMap<>();

    for (Map.Entry<Long, Integer> entry : dp.entrySet()) {
        long state = entry.getKey();
        int cnt = entry.getValue();
        // 当前位置左边和上边的插头状态
        int left = (int) ((state >> (2 * j)) & 3);
        int up = (int) ((state >> (2 * (j + 1))) & 3);

        // 如果当前位置是障碍物，跳过
        if (grid[i][j] == 0) {
            // 只有当左右插头都不存在时才合法
            if (left == 0 && up == 0) {
                newDp2.put(state, newDp2.getOrDefault(state, 0) + cnt);
            }
            continue;
        }

        // 处理各种插头组合情况
        // 1. 没有左插头和上插头
        if (left == 0 && up == 0) {
            // 只能创建新的插头对（用于回路的开始）
            if (i < n - 1 && j < m - 1 && grid[i+1][j] == 1 && grid[i][j+1] == 1) {
                long newState = state | (1L << (2 * j)) | (2L << (2 * (j + 1)));
                newDp2.put(newState, newDp2.getOrDefault(newState, 0) + cnt);
            }
        }

        // 2. 只有左插头
        else if (left != 0 && up == 0) {
            // 向下延伸
            if (i < n - 1 && grid[i+1][j] == 1) {
                newDp2.put(state, newDp2.getOrDefault(state, 0) + cnt);
            }
            // 向右延伸
            if (j < m - 1 && grid[i][j+1] == 1) {

```

```

        long newState = (state & ~(3L << (2 * j))) | ((long)left << (2 * (j +
1)));
        newDp2.put(newState, newDp2.getOrDefault(newState, 0) + cnt);
    }
}

// 3. 只有上插头
else if (left == 0 && up != 0) {
    // 向右延伸
    if (j < m - 1 && grid[i][j+1] == 1) {
        newDp2.put(state, newDp2.getOrDefault(state, 0) + cnt);
    }
    // 向下延伸
    if (i < n - 1 && grid[i+1][j] == 1) {
        long newState = (state & ~(3L << (2 * (j + 1)))) | ((long)up << (2 *
j));
        newDp2.put(newState, newDp2.getOrDefault(newState, 0) + cnt);
    }
}

// 4. 同时有左插头和上插头
else {
    // 合并插头
    long newState = (state & ~(3L << (2 * j))) & ~(3L << (2 * (j + 1)));

    // 如果是形成回路的最后一步
    if (left == up) {
        // 检查是否所有插头都已连接
        if (newState == 0 && i == n - 1 && j == m - 1) {
            newDp2.put(newState, newDp2.getOrDefault(newState, 0) + cnt);
        }
    } else {
        // 合并两个不同的插头
        newDp2.put(newState, newDp2.getOrDefault(newState, 0) + cnt);
    }
}

dp = newDp2;
}

// 最终状态应该是没有任何插头（形成回路）

```

```

    return dp.getOrDefault(0L, 0);
}

// ===== 冷门模型：树上背包的优化 =====

private static void dfsTreeKnapsack(int u, int parent, int capacity,
                                    List<List<Integer>> tree, int[] weights,
                                    int[] values, int[][] dp, int[] size) {
    // 初始化当前节点
    size[u] = 1;
    if (weights[u] <= capacity) {
        dp[u][weights[u]] = values[u];
    }

    // 对每个子节点，按照子树大小排序，小的先合并
    List<Pair<Integer, Integer>> children = new ArrayList<>();
    for (int v : tree.get(u)) {
        if (v != parent) {
            dfsTreeKnapsack(v, u, capacity, tree, weights, values, dp, size);
            children.add(new Pair<>(size[v], v));
        }
    }

    // 按子树大小排序
    Collections.sort(children, (a, b) -> a.first - b.first);

    for (Pair<Integer, Integer> child : children) {
        int sz = child.first;
        int v = child.second;
        // 逆序遍历容量，避免重复计算
        for (int i = Math.min(size[u], capacity); i >= 0; --i) {
            if (dp[u][i] == 0 && i != 0) continue;
            for (int j = 1; j <= Math.min(sz, capacity - i); ++j) {
                if (dp[v][j] > 0 && i + j <= capacity) {
                    dp[u][i + j] = Math.max(dp[u][i + j], dp[u][i] + dp[v][j]);
                }
            }
        }
    }

    // 更新子树大小
    size[u] += sz;
}
}

```

```
public static int treeKnapsackOptimized(int root, int capacity, List<List<Integer>> tree,
                                         int[] weights, int[] values) {
    /*
    树上背包的优化实现（小到大合并）
```

问题描述：

在树上选择一些节点，使得总重量不超过容量，且总价值最大

解题思路：

1. 使用后序遍历处理子树
2. 使用小到大合并的策略优化复杂度
3. 对于每个节点，维护一个容量为 capacity 的背包

参数：

root: 根节点
capacity: 背包容量
tree: 树的邻接表
weights: 每个节点的重量
values: 每个节点的价值

返回：

int: 最大价值

时间复杂度： $O(n * capacity^2)$ ，但通过小到大合并可以降低常数

空间复杂度： $O(n * capacity)$

*/

```
int n = tree.size();
int[][] dp = new int[n][capacity + 1];
int[] size = new int[n];

// 深度优先搜索处理子树
dfsTreeKnapsack(root, -1, capacity, tree, weights, values, dp, size);

// 返回根节点的最大价值
int maxVal = 0;
for (int val : dp[root]) {
    maxVal = Math.max(maxVal, val);
}
return maxVal;
```

```
// ===== 补充题目与应用 =====
```

```
// 以下是一些使用上述高级 DP 技术的经典题目及其代码实现
```

```
// 1. 编辑距离问题 (LeetCode 72)
```

```
public static int editDistance(String word1, String word2) {
```

```
/*
```

```
LeetCode 72. 编辑距离
```

```
题目链接: https://leetcode-cn.com/problems/edit-distance/
```

问题描述:

给你两个单词 word1 和 word2，计算出将 word1 转换成 word2 所使用的最少操作数。

你可以对一个单词进行如下三种操作：插入一个字符、删除一个字符、替换一个字符。

解题思路:

使用二维 DP， $dp[i][j]$ 表示 word1 的前 i 个字符转换为 word2 的前 j 个字符所需的最少操作数。

时间复杂度: $O(m \times n)$

空间复杂度: $O(m \times n)$

```
*/
```

```
int m = word1.length();
```

```
int n = word2.length();
```

```
//  $dp[i][j]$  表示 word1 的前  $i$  个字符转换为 word2 的前  $j$  个字符所需的最少操作数
```

```
int[][] dp = new int[m + 1][n + 1];
```

```
// 初始化边界
```

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

```
    dp[i][0] = i;
```

```
}
```

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

```
    dp[0][j] = j;
```

```
}
```

```
// 动态规划填表
```

```
for (int i = 1; i <= m; ++i) {
```

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

```
        if (word1.charAt(i - 1) == word2.charAt(j - 1)) {
```

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

```
        } else {
```

```
            dp[i][j] = Math.min(Math.min(dp[i - 1][j], dp[i][j - 1]), dp[i - 1][j - 1]) +
```

```
1;
```

```
    }
```

```
}
```

```
}
```

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

// 2. 最长递增子序列 (LeetCode 300)

```
public static int lengthOfLIS(int[] nums) {
```

```
    /*
```

LeetCode 300. 最长递增子序列

题目链接: <https://leetcode-cn.com/problems/longest-increasing-subsequence/>

问题描述:

给你一个整数数组 `nums`，找到其中最长严格递增子序列的长度。

解题思路:

使用贪心 + 二分查找优化的 DP 方法。

`tails[i]` 表示长度为 $i+1$ 的递增子序列的末尾元素的最小值。

时间复杂度: $O(n \log n)$

空间复杂度: $O(n)$

```
*/
```

```
if (nums == null || nums.length == 0) {
    return 0;
}
```

```
List<Integer> tails = new ArrayList<>();
for (int num : nums) {
    // 二分查找 num 应该插入的位置
    int left = 0, right = tails.size();
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (tails.get(mid) >= num) {
            right = mid;
        } else {
            left = mid + 1;
        }
    }
    if (left == tails.size()) {
        tails.add(num);
    } else {
        tails.set(left, num);
    }
}

return tails.size();
```

```
}
```

```
// 3. 背包问题变种 - 完全背包 (LeetCode 322)
```

```
public static int coinChange(int[] coins, int amount) {
```

```
/*
```

```
LeetCode 322. 零钱兑换
```

```
题目链接: https://leetcode-cn.com/problems/coin-change/
```

问题描述:

给你一个整数数组 coins，表示不同面额的硬币；以及一个整数 amount，表示总金额。

计算并返回可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额，返回-1。

解题思路:

使用完全背包的思想，dp[i] 表示凑成金额 i 所需的最少硬币数。

时间复杂度: O(amount * n)

空间复杂度: O(amount)

```
*/
```

```
// 初始化 dp 数组为无穷大
```

```
int[] dp = new int[amount + 1];
```

```
Arrays.fill(dp, Integer.MAX_VALUE);
```

```
dp[0] = 0; // 凑成金额 0 需要 0 个硬币
```

```
for (int coin : coins) {
```

```
    for (int i = coin; i <= amount; ++i) {
```

```
        if (dp[i - coin] != Integer.MAX_VALUE) {
```

```
            dp[i] = Math.min(dp[i], dp[i - coin] + 1);
```

```
        }
```

```
}
```

```
}
```

```
return dp[amount] == Integer.MAX_VALUE ? -1 : dp[amount];
```

```
}
```

```
// 4. 矩阵链乘法 (区间 DP 的经典应用)
```

```
static class MatrixChainResult {
```

```
    int[][] dp;
```

```
    int[][] s;
```

```
    public MatrixChainResult(int[][] dp, int[][] s) {
```

```
        this.dp = dp;
```

```
        this.s = s;
```

```
    }  
}
```

```
public static MatrixChainResult matrixChainOrder(int[] p) {
```

```
    /*
```

```
     矩阵链乘法问题
```

```
     题目来源：算法导论
```

问题描述：

给定一系列矩阵，计算乘法顺序使得标量乘法的次数最少。

解题思路：

使用区间 DP， $dp[i][j]$ 表示计算第 i 到第 j 个矩阵的乘积所需的最少标量乘法次数。

可以使用 Knuth 优化进一步降低时间复杂度。

时间复杂度： $O(n^3)$

空间复杂度： $O(n^2)$

```
*/
```

```
int n = p.length - 1; // 矩阵的个数
```

```
// dp[i][j] 表示计算第 i 到第 j 个矩阵的乘积所需的最少标量乘法次数
```

```
int[][] dp = new int[n + 1][n + 1];
```

```
// s[i][j] 记录最优分割点
```

```
int[][] s = new int[n + 1][n + 1];
```

```
// 枚举区间长度
```

```
for (int length = 2; length <= n; ++length) {
```

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

```
        int j = i + length - 1;
```

```
        dp[i][j] = Integer.MAX_VALUE;
```

```
        // 枚举分割点
```

```
        for (int k = i; k < j; ++k) {
```

```
            // 计算当前分割点的代价
```

```
            int cost = dp[i][k] + dp[k + 1][j] + p[i - 1] * p[k] * p[j];
```

```
            if (cost < dp[i][j]) {
```

```
                dp[i][j] = cost;
```

```
                s[i][j] = k;
```

```
}
```

```
}
```

```
}
```

```
return new MatrixChainResult(dp, s);
```

```
}
```

```
// 5. 旅行商问题 (TSP) 的 DP 实现
public static int travelingSalesmanProblem(int[][] graph) {
    /*
    旅行商问题
    题目来源: 算法竞赛经典问题
```

问题描述:

给定一个完全图, 找到一条访问每个城市恰好一次并返回起点的最短路径。

解题思路:

使用状态压缩 DP, $dp[mask][u]$ 表示访问过的城市集合为 $mask$, 当前在城市 u 时的最短路径长度。

时间复杂度: $O(n^2 * 2^n)$

空间复杂度: $O(n * 2^n)$

*/

```
int n = graph.length;
// dp[mask][u] 表示访问过的城市集合为 mask, 当前在城市 u 时的最短路径长度
int[][] dp = new int[1 << n][n];
for (int i = 0; i < (1 << n); ++i) {
    Arrays.fill(dp[i], Integer.MAX_VALUE);
}
```

// 初始状态: 只访问了起点, 路径长度为 0

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

// 枚举所有可能的状态

```
for (int mask = 1; mask < (1 << n); ++mask) {
    // 枚举当前所在的城市
    for (int u = 0; u < n; ++u) {
        if ((mask & (1 << u)) == 0) {
            continue;
        }
        // 枚举下一个要访问的城市
        for (int v = 0; v < n; ++v) {
            if ((mask & (1 << v)) != 0) {
                continue;
            }
            int newMask = mask | (1 << v);
            if (dp[mask][u] != Integer.MAX_VALUE && graph[u][v] != Integer.MAX_VALUE) {
                dp[newMask][v] = Math.min(dp[newMask][v], dp[mask][u] + graph[u][v]);
            }
        }
    }
}
```

```

        }
    }
}

// 找到最短的回路
int result = Integer.MAX_VALUE;
for (int u = 0; u < n; ++u) {
    if (dp[(1 << n) - 1][u] != Integer.MAX_VALUE && graph[u][0] != Integer.MAX_VALUE) {
        result = Math.min(result, dp[(1 << n) - 1][u] + graph[u][0]);
    }
}

return result;
}

```

// 6. 区间 DP: 最优三角剖分

```

public static int minimumScoreTriangulation(int[] values) {
/*
LeetCode 1039. 多边形三角剖分的最低得分
题目链接: https://leetcode-cn.com/problems/minimum-score-triangulation-of-polygon/

```

问题描述:

给定一个凸多边形，将其三角剖分，使得所有三角形的顶点乘积之和最小。

解题思路:

使用区间 DP， $dp[i][j]$ 表示从顶点 i 到顶点 j 的多边形三角剖分的最小得分。

时间复杂度: $O(n^3)$

空间复杂度: $O(n^2)$

*/

```

int n = values.length;
// dp[i][j] 表示从顶点 i 到顶点 j 的多边形三角剖分的最小得分
int[][] dp = new int[n][n];

```

// 枚举区间长度

```

for (int length = 3; length <= n; ++length) {
    for (int i = 0; i + length - 1 < n; ++i) {
        int j = i + length - 1;
        dp[i][j] = Integer.MAX_VALUE;
        // 枚举中间点
        for (int k = i + 1; k < j; ++k) {
            dp[i][j] = Math.min(dp[i][j],

```

```

        dp[i][k] + dp[k][j] + values[i] * values[k] * values[j]);
    }
}
}

return dp[0][n - 1];
}

```

// 7. 博弈 DP: 石子游戏

```

public static boolean stoneGame(int[] piles) {
/*
LeetCode 877. 石子游戏
题目链接: https://leetcode-cn.com/problems/stone-game/

```

问题描述:

给定一个表示石子堆的数组，两个玩家轮流从两端取石子，每次只能取一个，取到最后一个石子的人获胜。

判断先手是否必胜。

解题思路:

使用区间 DP， $dp[i][j]$ 表示在区间 $[i, j]$ 中，先手能获得的最大净胜分。

时间复杂度: $O(n^2)$

空间复杂度: $O(n^2)$

*/

```
int n = piles.length;
```

// $dp[i][j]$ 表示在区间 $[i, j]$ 中，先手能获得的最大净胜分

```
int[][] dp = new int[n][n];
```

// 初始化单个石子堆

```
for (int i = 0; i < n; ++i) {
    dp[i][i] = piles[i];
}
```

// 枚举区间长度

```
for (int length = 2; length <= n; ++length) {
    for (int i = 0; i + length - 1 < n; ++i) {
        int j = i + length - 1;
        // 先手可以选择取左边或右边
        dp[i][j] = Math.max(piles[i] - dp[i + 1][j], piles[j] - dp[i][j - 1]);
    }
}
```

```

// 先手净胜分大于 0 则必胜
return dp[0][n - 1] > 0;
}

// 8. 数位 DP: 统计 1 出现的次数
public static int countDigitOne(int n) {
    /*
    LeetCode 233. 数字 1 的个数
    题目链接: https://leetcode-cn.com/problems/number-of-digit-one/

```

问题描述:

给定一个整数 n , 计算所有小于等于 n 的非负整数中数字 1 出现的个数。

解题思路:

使用数位 DP, 逐位处理每一位上 1 出现的次数。

时间复杂度: $O(\log n)$

空间复杂度: $O(\log n)$

*/

```

if (n <= 0) {
    return 0;
}

```

```

String s = String.valueOf(n);
int length = s.length();
int count = 0;

```

// 逐位处理

```

for (int i = 0; i < length; ++i) {
    long high = 0;
    if (i > 0) {
        high = Long.parseLong(s.substring(0, i));
    }
    int current = s.charAt(i) - '0';
    long low = 0;
    if (i < length - 1) {
        low = Long.parseLong(s.substring(i + 1));
    }
    long digit = (long) Math.pow(10, length - i - 1);

    if (current == 0) {
        // 当前位为 0, 高位决定
        count += high * digit;
    }
}

```

```

        } else if (current == 1) {
            // 当前位为 1, 高位+低位+1
            count += high * digit + low + 1;
        } else {
            // 当前位大于 1, 高位+1
            count += (high + 1) * digit;
        }
    }

    return count;
}

// 9. 树形 DP: 打家劫舍 III
static class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int x) { val = x; }
}

private static int[] robDFS(TreeNode node) {
    if (node == null) {
        return new int[]{0, 0};
    }

    int[] left = robDFS(node.left);
    int[] right = robDFS(node.right);

    // robCurrent 表示偷当前节点, notRobCurrent 表示不偷当前节点
    int robCurrent = node.val + left[1] + right[1];
    int notRobCurrent = Math.max(left[0], left[1]) + Math.max(right[0], right[1]);

    return new int[]{robCurrent, notRobCurrent};
}

public static int rob(TreeNode root) {
    /*
    LeetCode 337. 打家劫舍 III
    题目链接: https://leetcode-cn.com/problems/house-robber-iii/
    */
}

```

问题描述:

在上次打劫完一条街道之后和一圈房屋后，小偷又发现了一个新的可行窃的地区。

这个地区只有一个入口，我们称之为“根”。除了“根”之外，每栋房子有且只有一个“父”房子与

之相连。

一番侦察之后，聪明的小偷意识到“这个地方的所有房屋的排列类似于一棵二叉树”。

如果两个直接相连的房子在同一天晚上被打劫，房屋将自动报警。

计算在不触动警报的情况下，小偷一晚能够盗取的最高金额。

解题思路：

使用树形 DP，对于每个节点，维护两个状态：偷或不偷。

时间复杂度： $O(n)$

空间复杂度： $O(h)$ ， h 为树的高度

*/

```
int[] result = robDFS(root);
return Math.max(result[0], result[1]);
}
```

// 10. 状态压缩 DP：蒙斯特曼问题

```
public static int monsterGame(int[][] grid) {
/*
蒙斯特曼问题
题目来源：算法竞赛问题
```

问题描述：

在网格中放置怪物，使得任何两个怪物都不在同一行、同一列或对角线上。

解题思路：

使用状态压缩 DP， $dp[i][mask]$ 表示处理到第 i 行，已放置的列的状态为 $mask$ 时的方案数。

时间复杂度： $O(n * 2^n)$

空间复杂度： $O(2^n)$

*/

```
int n = grid.length;
// dp[i][mask] 表示处理到第 i 行，已放置的列的状态为 mask 时的方案数
long[] dp = new long[1 << n];
dp[0] = 1;
```

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

```
    long[] newDp = new long[1 << n];
```

```
    for (int mask = 0; mask < (1 << n); ++mask) {
```

```
        if (dp[mask] == 0) {
```

```
            continue;
```

```
}
```

```
        // 枚举所有可能的放置位置
```

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

```

        // 检查是否可以在(i, j)放置怪物
        if ((mask & (1 << j)) == 0 && grid[i][j] == 1) {
            // 检查对角线
            boolean valid = true;
            for (int k = 0; k < i; ++k) {
                if ((mask & (1 << k)) != 0 && Math.abs(k - j) == i - k) {
                    valid = false;
                    break;
                }
            }
            if (valid) {
                newDp[mask | (1 << j)] += dp[mask];
            }
        }
    }

    dp = newDp;
}

return (int) dp[(1 << n) - 1];
}

```

// 11. 高维 DP: 三维背包

```

public static int threeDimensionKnapsack(int n, int[] capacity, int[][] items) {
/*
三维背包问题
题目来源: 算法竞赛问题

```

问题描述:

有 n 个物品，每个物品有体积、重量、价值三个属性，背包有体积和重量两个限制，求最大价值。

解题思路:

使用三维 DP， $dp[i][j][k]$ 表示前 i 个物品，体积为 j ，重量为 k 时的最大价值。

时间复杂度: $O(n * V * W)$

空间复杂度: $O(n * V * W)$

*/

```

int V = capacity[0];
int W = capacity[1];
// 初始化 dp 数组
int[][][] dp = new int[n + 1][V + 1][W + 1];

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

```

```

int v = items[i-1][0];
int w = items[i-1][1];
int val = items[i-1][2];
for (int j = 0; j <= V; ++j) {
    for (int k = 0; k <= W; ++k) {
        // 不选当前物品
        dp[i][j][k] = dp[i-1][j][k];
        // 选当前物品（如果有足够的空间）
        if (j >= v && k >= w) {
            dp[i][j][k] = Math.max(dp[i][j][k], dp[i-1][j-v][k-w] + val);
        }
    }
}
return dp[n][V][W];
}

```

// 12. 斜率优化 DP 示例

```
static class ConvexHullTrick {
```

```
/*
```

凸包优化技巧示例

题目来源：算法竞赛问题

问题描述：

当状态转移方程形如 $dp[i] = \min\{dp[j] + a[i] * b[j] + c\}$ 时，可以使用凸包优化。

解题思路：

将转移方程转换为直线的形式，维护凸包以快速查询最小值。

时间复杂度：O(n)

空间复杂度：O(n)

```
*/
```

```
static class Line {
    long k, b;
    Line(long k, long b) { this.k = k; this.b = b; }
}
```

```
Deque<Line> dq = new LinkedList<>();
```

// 添加一条直线 $y = kx + b$

```
void addLine(long k, long b) {
```

```

// 当队列中至少有两条直线时，检查是否需要删除末尾的直线
while (dq.size() >= 2) {
    Line l1 = getNthLast(dq, 2);
    Line l2 = dq.getLast();
    // 判断直线 l1 和 l2 的交点是否在 l2 和新直线的交点右侧
    if ((l2.b - l1.b) * (k - l2.k) >= (b - l2.b) * (l2.k - l1.k)) {
        dq.removeLast();
    } else {
        break;
    }
}
dq.addLast(new Line(k, b));
}

// 获取队列中倒数第 n 个元素
private Line getNthLast(Deque<Line> deque, int n) {
    if (n <= 0 || n > deque.size()) {
        throw new IndexOutOfBoundsException();
    }
    Iterator<Line> it = deque.descendingIterator();
    Line result = null;
    for (int i = 0; i < n; ++i) {
        result = it.next();
    }
    return result;
}

// 查询 x 处的最小值
long query(long x) {
    // 如果队列中至少有两条直线，且第一条直线在 x 处的值大于第二条，删除第一条
    while (dq.size() >= 2) {
        Line l1 = dq.getFirst();
        Line l2 = dq.getFirst(); // 错误，应该是第二个元素
        // 修正：正确获取第二个元素
        Line l2Correct = getNthLast(dq, dq.size() - 1);
        if (l1.k * x + l1.b >= l2Correct.k * x + l2Correct.b) {
            dq.removeFirst();
        } else {
            break;
        }
    }
    if (dq.isEmpty()) {
        return Long.MAX_VALUE;
    }
}

```

```

    }

    Line l = dq.getFirst();
    return l.k * x + l.b;
}

// 正确的 query 方法实现
long queryCorrect(long x) {
    while (dq.size() >= 2) {
        Line l1 = dq.pollFirst();
        Line l2 = dq.peekFirst();
        if (l1.k * x + l1.b >= l2.k * x + l2.b) {
            // 继续弹出
        } else {
            dq.offerFirst(l1); // 恢复 l1
            break;
        }
    }
    if (dq.isEmpty()) {
        return Long.MAX_VALUE;
    }
    Line l = dq.peekFirst();
    return l.k * x + l.b;
}

}

// ===== 高级优化体系: Knuth 优化 =====
/***
 * Knuth 优化的 DP 算法
 *
 * 问题描述:
 * 解决区间 DP 问题, 其中状态转移方程满足四边形不等式
 *
 * 解题思路:
 * 1. 使用 Knuth 优化将时间复杂度从 O(n^3) 降低到 O(n^2)
 * 2. 维护最优转移点数组 opt[i][j], 表示计算 dp[i][j] 时的最优 k 值
 * 3. 根据 opt[i][j-1] ≤ opt[i][j] ≤ opt[i+1][j] 的性质进行剪枝
 *
 * 应用题目:
 * - POJ 1160 Post Office
 * - HDU 4008 Parent and son
 * - Codeforces 245H Queries for Number of Palindromes
 *
 * 时间复杂度: O(n^2)

```

```

* 空间复杂度: O(n^2)
*/
public static class KnuthOptimization {
    /**
     * Knuth 优化的 DP 实现
     *
     * @param n 区间长度
     * @param costFunc 计算区间(i, j)代价的函数
     * @return 最小代价矩阵
     */
    public static long[][] solve(int n, CostFunction costFunc) {
        // 初始化 dp 和 opt 数组
        long[][] dp = new long[n + 1][n + 1];
        int[][] opt = new int[n + 1][n + 1];

        // 初始化长度为 1 的区间
        for (int i = 1; i <= n; i++) {
            dp[i][i] = 0;
            opt[i][i] = i;
        }

        // 枚举区间长度
        for (int length = 2; length <= n; length++) {
            // 枚举起始点
            for (int i = 1; i <= n - length + 1; i++) {
                int j = i + length - 1;
                // 根据 Knuth 优化的性质，最优 k 在 opt[i][j-1] 到 opt[i+1][j] 之间
                int lower = opt[i][j - 1];
                int upper = (i + 1 <= j) ? opt[i + 1][j] : j - 1;

                dp[i][j] = Long.MAX_VALUE;
                for (int k = lower; k <= upper; k++) {
                    if (dp[i][k] != Long.MAX_VALUE && dp[k + 1][j] != Long.MAX_VALUE) {
                        long cost = costFunc.calculate(i, j);
                        if (cost != Long.MAX_VALUE) {
                            long current = dp[i][k] + dp[k + 1][j] + cost;
                            if (current < dp[i][j]) {
                                dp[i][j] = current;
                                opt[i][j] = k;
                            }
                        }
                    }
                }
            }
        }
    }
}

```

```

        }
    }

    return dp;
}

/***
 * 代价函数接口
 */
public interface CostFunction {
    long calculate(int i, int j);
}

/***
 * 应用示例：最优二叉搜索树问题（POJ 3280）
 * 时间复杂度：O(n^2)
 * 空间复杂度：O(n^2)
 */
public static long optimalBinarySearchTree(char[] keys, int[] freq) {
    int n = keys.length;
    CostFunction costFunc = (i, j) -> {
        long sum = 0;
        for (int k = i-1; k < j; k++) {
            sum += freq[k];
        }
        return sum;
    };

    long[][] dp = solve(n, costFunc);
    return dp[1][n];
}

// ===== 高级优化体系：Divide & Conquer Optimization =====
/***
 * 分治优化（Divide & Conquer Optimization）
 *
 * 问题描述：
 * 解决形如  $dp[i][j] = \min\{dp[i-1][k] + cost(k, j)\}$ , 其中  $k < j$ 
 * 当转移满足决策单调性时使用
 *
 * 解题思路：
 * 1. 利用决策单调性，使用分治法优化 DP

```

```

* 2. 对于  $dp[i][j]$ , 当  $i$  固定时, 最优转移点  $k$  随着  $j$  的增加而单调不减
* 3. 使用分治的方式计算每个区间的最优决策
*
* 应用题目:
* - Codeforces 321E Ciel and Gondolas
* - HDU 3480 Division
* - SPOJ LARMY
*
* 时间复杂度:  $O(n*m \log m)$ 
* 空间复杂度:  $O(n*m)$ 
*/
public static class DivideConquerOptimization {
    private long[][] dp;
    private CostFunction costFunc;
    private int n, m;

    /**
     * 代价函数接口
     */
    public interface CostFunction {
        long calculate(int k, int j);
    }

    /**
     * 分治求解  $dp[i][l..r]$ , 其中最优转移点在  $opt_l..opt_r$  之间
     */
    private void solve(int i, int l, int r, int opt_l, int opt_r) {
        if (l > r) return;

        int mid = (l + r) / 2;
        int best_k = opt_l;
        dp[i][mid] = Long.MAX_VALUE;

        // 在  $opt_l$  到  $\min(mid, opt_r)$  之间寻找最优 k
        for (int k = opt_l; k <= Math.min(mid, opt_r); k++) {
            if (dp[i-1][k] != Long.MAX_VALUE) {
                long cost = costFunc.calculate(k, mid);
                if (cost != Long.MAX_VALUE) {
                    long current = dp[i-1][k] + cost;
                    if (current < dp[i][mid]) {
                        dp[i][mid] = current;
                        best_k = k;
                    }
                }
            }
        }
    }
}

```

```

        }
    }

    // 递归处理左右子区间
    solve(i, 1, mid - 1, opt_l, best_k);
    solve(i, mid + 1, r, best_k, opt_r);
}

/***
 * 主入口: 分治优化 DP
 *
 * @param n 维度 1
 * @param m 维度 2
 * @param costFunc 计算 cost(k, j) 的函数
 * @return dp 数组
 */
public long[][] solve(int n, int m, CostFunction costFunc) {
    // 初始化 dp 数组
    dp = new long[n + 1][m + 1];
    this.costFunc = costFunc;
    this.n = n;
    this.m = m;

    // 初始化 dp 数组为无穷大
    for (int i = 0; i <= n; i++) {
        Arrays.fill(dp[i], Long.MAX_VALUE);
    }
    dp[0][0] = 0; // 初始状态

    // 对每个 i 应用分治优化
    for (int i = 1; i <= n; i++) {
        solve(i, 1, m, 0, m);
    }

    return dp;
}

/***
 * 应用示例: 将数组分成 k 个子数组的最小代价和 (LeetCode 410)
 * 时间复杂度: O(n*k log n)
 * 空间复杂度: O(n*k)
 */

```

```

public static long splitArray(int[] nums, int k) {
    int n = nums.length;
    // 预处理前缀和
    long[] prefixSum = new long[n + 1];
    for (int i = 0; i < n; i++) {
        prefixSum[i+1] = prefixSum[i] + nums[i];
    }

    // 代价函数：计算从 k+1 到 j 的和的平方
    CostFunction costFunc = (k, j) -> {
        long sum = prefixSum[j] - prefixSum[k];
        return sum * sum; // 这里可以根据实际问题定义不同的代价
    };

    DivideConquerOptimization optimizer = new DivideConquerOptimization();
    long[][] dp = optimizer.solve(k, n, costFunc);
    return dp[k][n];
}

// ===== 高级优化体系: SMAWK 算法 (行最小查询) =====
/***
 * SMAWK 算法用于在 Monge 矩阵中快速查找每行的最小值
 *
 * 问题描述:
 * 给定一个 Monge 矩阵, 快速找到每行的最小值位置
 *
 * 解题思路:
 * 1. Monge 矩阵满足性质: matrix[i][j] + matrix[i+1][j+1] ≤ matrix[i][j+1] + matrix[i+1][j]
 * 2. SMAWK 算法利用这一性质, 可以在 O(m+n) 时间内找到每行的最小值
 * 3. 主要步骤包括行压缩和递归求解
 *
 * 应用题目:
 * - POJ 3156 Interconnect
 * - Codeforces 472D Design Tutorial: Inverse the Problem
 * - SPOJ MCQUERY
 *
 * 时间复杂度: O(m+n), 其中 m 是行数, n 是列数
 * 空间复杂度: O(m+n)
 */
public static class SMAWK {
    /**
     * 行压缩: 只保留可能成为最小值的行

```

```

*/
private static List<Integer> reduceRows(List<Integer> rows, long[][] matrix) {
    LinkedList<Integer> stack = new LinkedList<>();
    for (int i : rows) {
        while (stack.size() >= 2) {
            int j1 = stack.removeLast();
            int j2 = stack.removeLast();
            stack.addLast(j1); // 恢复栈状态

            // 比较两个行在列 stack.size()-2 处的值
            int col = stack.size() - 2;
            if (col < matrix[0].length) {
                if (matrix[j2][col] <= matrix[i][col]) {
                    break;
                } else {
                    stack.removeLast(); // 移除 j1
                }
            } else {
                break;
            }
        }
        stack.addLast(i);
    }
    return new ArrayList<>(stack);
}

/***
 * 递归实现 SMAWK 算法
 */
private static int[] smawkRec(List<Integer> rows, List<Integer> cols, long[][] matrix) {
    int m = rows.size();
    int[] result = new int[m];

    if (m == 0) return result;

    // 行压缩
    List<Integer> reducedRows = reduceRows(rows, matrix);

    // 递归求解列数为奇数的子问题
    List<Integer> halfCols = new ArrayList<>();
    for (int i = 1; i < cols.size(); i += 2) {
        halfCols.add(cols.get(i));
    }
}

```

```

int[] minCols = new int[reducedRows.size()];
Arrays.fill(minCols, -1);

if (!halfCols.isEmpty()) {
    // 递归求解
    int[] subResult = smawkRec(reducedRows, halfCols, matrix);
    // 复制结果
    for (int i = 0; i < subResult.length; i++) {
        minCols[i] = subResult[i];
    }
}

// 扩展结果到所有列
int k = 0; // minCols 的索引

for (int i = 0; i < m; i++) {
    int row = rows.get(i);
    // 确定当前行的最小值可能在哪个区间
    int start = 0;
    if (i > 0 && k > 0 && minCols[k-1] != -1) {
        start = minCols[k-1];
    }
    int end = (k < minCols.length && minCols[k] != -1) ? minCols[k] :
cols.get(cols.size() - 1);

    // 在这个区间内查找最小值
    long minValue = Long.MAX_VALUE;
    int minCol = start;

    // 找到 start 和 end 在 cols 中的索引
    int startIdx = cols.indexOf(start);
    int endIdx = cols.indexOf(end);

    for (int idx = startIdx; idx <= endIdx; idx++) {
        int col = cols.get(idx);
        if (col < matrix[0].length && matrix[row][col] < minValue) {
            minValue = matrix[row][col];
            minCol = col;
        }
    }
}

result[i] = minCol;

```

```

        // 如果当前行在 reducedRows 中, 且不是最后一行, k 前进
        if (k < reducedRows.size() && row == reducedRows.get(k)) {
            k++;
        }
    }

    return result;
}

/***
 * SMAWK 算法主入口
 *
 * @param matrix 一个 Monge 矩阵
 * @return 每行最小值的列索引数组
 */
public static int[] solve(long[][] matrix) {
    if (matrix == null || matrix.length == 0) return new int[0];
    int m = matrix.length;
    int n = matrix[0].length;

    // 构造行索引和列索引数组
    List<Integer> rows = new ArrayList<>(m);
    List<Integer> cols = new ArrayList<>(n);
    for (int i = 0; i < m; i++) rows.add(i);
    for (int j = 0; j < n; j++) cols.add(j);

    // 调用递归实现
    return smawkRec(rows, cols, matrix);
}

/***
 * 应用示例: 寻找每一行的最小元素
 * 时间复杂度: O(m+n)
 * 空间复杂度: O(m+n)
 */
public static long[] findRowMins(long[][] matrix) {
    int[] minCols = solve(matrix);
    long[] result = new long[matrix.length];
    for (int i = 0; i < matrix.length; i++) {
        result[i] = matrix[i][minCols[i]];
    }
    return result;
}

```

```

    }
}

// ===== 高级优化体系: Aliens Trick (二分约束参数+可行性 DP)
=====

/***
 * Aliens Trick (二分约束参数+可行性 DP)
 *
 * 问题描述:
 * 解决带约束的优化问题, 通常形如最小化总成本, 同时满足某些约束条件
 *
 * 解题思路:
 * 1. 将约束条件转化为参数  $\lambda$ , 构造拉格朗日函数
 * 2. 对  $\lambda$  进行二分查找, 使用可行性 DP 判断当前  $\lambda$  下是否满足约束
 * 3. 根据可行性 DP 的结果调整二分区间
 *
 * 应用题目:
 * - Codeforces 739E Gosha is Hunting
 * - POJ 3686 The Windy's
 * - SPOJ QTREE5
 *
 * 时间复杂度:  $O(\log((right-left)/\epsilon) * T(DP))$ , 其中  $T(DP)$  是一次 DP 的时间复杂度
 * 空间复杂度:  $O(DP$  空间复杂度)
 */

public static class AliensTrick {

    /**
     * 成本函数接口: 计算带参数  $\lambda$  的成本和约束值
     */

    public interface CostFunction {
        Result calculate(double lambda);

        class Result {
            public double value;
            public double constraint;
            public Result(double value, double constraint) {
                this.value = value;
                this.constraint = constraint;
            }
        }
    }

    /**
     * 约束检查函数接口: 检查当前解是否满足约束
    */
}

```

```

*/
public interface CheckFunction {
    boolean check(double constraint);
}

/***
 * Aliens Trick 主入口
 *
 * @param costFunc 成本函数
 * @param checkFunc 约束检查函数
 * @param left 二分左边界
 * @param right 二分右边界
 * @param eps 精度要求
 * @return 最优参数 lambda 和对应最优解
*/
public static Result solve(CostFunction costFunc, CheckFunction checkFunc,
                           double left, double right, double eps) {
    double bestLambda = left;
    double bestValue = 0.0;

    // 二分查找参数 lambda
    while (right - left > eps) {
        double mid = (left + right) / 2;
        // 计算当前参数下的解和约束值
        CostFunction.Result result = costFunc.calculate(mid);

        if (checkFunc.check(result.constraint)) {
            // 满足约束，尝试更小的参数
            right = mid;
            bestLambda = mid;
            bestValue = result.value;
        } else {
            // 不满足约束，需要增大参数
            left = mid;
        }
    }

    return new Result(bestLambda, bestValue);
}

/***
 * 结果类
*/

```

```

public static class Result {
    public double lambda;
    public double value;
    public Result(double lambda, double value) {
        this.lambda = lambda;
        this.value = value;
    }
}

/**
 * 应用示例：将数组分成恰好 k 个部分，使得最大子数组和最小（LeetCode 410 的变种）
 * 时间复杂度：O(log(S) * n)，其中 S 是数组元素和
 * 空间复杂度：O(n)
 */
public static double splitArrayK(int[] nums, int k) {
    // 计算数组元素和作为二分上限
    double sum = 0;
    for (int num : nums) sum += num;

    // 成本函数：使用 DP 计算在给定 lambda 下的最小成本
    CostFunction costFunc = lambda -> {
        int n = nums.length;
        double[] dp = new double[n + 1];
        int[] cnt = new int[n + 1];

        Arrays.fill(dp, Double.MAX_VALUE);
        dp[0] = 0;
        cnt[0] = 0;

        for (int i = 1; i <= n; i++) {
            double sumSeg = 0;
            for (int j = i-1; j >= 0; j--) {
                sumSeg += nums[j];
                if (dp[j] != Double.MAX_VALUE) {
                    double current = dp[j] + sumSeg * sumSeg + lambda; // lambda 作为惩罚项
                    if (current < dp[i]) {
                        dp[i] = current;
                        cnt[i] = cnt[j] + 1;
                    }
                }
            }
        }
    };
}

```

```

        return new CostFunction.Result(dp[n], cnt[n]);
    }

    // 约束检查函数: 确保分割次数不超过 k
    CheckFunction checkFunc = constraint -> constraint <= k;

    // 执行 Aliens Trick
    Result result = solve(costFunc, checkFunc, 0, sum * sum, 1e-7);
    return result.value;
}

// ===== 图上 DP→最短路: 分层图建模 =====
/***
 * 分层图 Dijkstra 算法
 *
 * 问题描述:
 * 给定一个图, 允许最多使用 k 次特殊操作 (如跳跃、免费通行等), 求最短路径
 *
 * 解题思路:
 * 1. 构建分层图, 每层代表使用不同次数的特殊操作
 * 2. 对于每个节点 u, 在第 i 层表示到达 u 时已经使用了 i 次特殊操作
 * 3. 使用 Dijkstra 算法在分层图上寻找最短路径
 *
 * 应用题目:
 * - LeetCode 787. K 站中转内最便宜的航班
 * - POJ 3159 Candies
 * - HDU 2957 Safety Assessment
 *
 * 时间复杂度: O((n*k + m*k) log(n*k))
 * 空间复杂度: O(n*k + m*k)
 */
public static class LayeredGraphShortestPath {

    /**
     * 边类
     */
    public static class Edge {
        int to;
        int weight;
        public Edge(int to, int weight) {
            this.to = to;
            this.weight = weight;
        }
    }
}

```

```

    }

}

/***
 * 分层图最短路径算法
 *
 * @param n 节点数量
 * @param edges 边的列表
 * @param k 允许使用的特殊操作次数
 * @param start 起始节点
 * @param end 目标节点
 * @return 最短路径长度, -1 表示不可达
 */
public static int solve(int n, List<List<Edge>> edges, int k, int start, int end) {
    // 构建分层图的邻接表
    List<List<Edge>> layeredGraph = new ArrayList<>();
    int totalNodes = n * (k + 1);
    for (int i = 0; i < totalNodes; i++) {
        layeredGraph.add(new ArrayList<>());
    }

    // 添加普通边 (不使用特殊操作)
    for (int i = 0; i < n; i++) {
        for (int j = 0; j <= k; j++) {
            int fromNode = i + j * n;
            for (Edge edge : edges.get(i)) {
                layeredGraph.get(fromNode).add(new Edge(edge.to + j * n, edge.weight));
            }
        }
    }

    // 添加使用特殊操作的边 (如果允许的话)
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < k; j++) {
            int fromNode = i + j * n;
            for (Edge edge : edges.get(i)) {
                // 这里假设特殊操作可以免费通行 (权为 0), 具体根据问题调整
                layeredGraph.get(fromNode).add(new Edge(edge.to + (j + 1) * n, 0));
            }
        }
    }

    // Dijkstra 算法

```

```

int[] dist = new int[totalNodes];
Arrays.fill(dist, Integer.MAX_VALUE);
dist[start] = 0; // 起始点在第 0 层

// 使用优先队列，按距离排序
PriorityQueue<int[]> pq = new PriorityQueue<>((a, b) -> a[1] - b[1]);
pq.offer(new int[] {start, 0});

while (!pq.isEmpty()) {
    int[] current = pq.poll();
    int u = current[0];
    int d = current[1];

    if (d > dist[u]) {
        continue;
    }

    for (Edge edge : layeredGraph.get(u)) {
        int v = edge.to;
        int w = edge.weight;
        if (dist[v] > d + w) {
            dist[v] = d + w;
            pq.offer(new int[] {v, dist[v]});
        }
    }
}

// 取所有层中到达终点的最小值
int result = Integer.MAX_VALUE;
for (int i = 0; i <= k; i++) {
    result = Math.min(result, dist[end + i * n]);
}

return result == Integer.MAX_VALUE ? -1 : result;
}

/***
 * 应用示例：LeetCode 787. K 站中转内最便宜的航班
 * 时间复杂度：O((n*k + m*k) log(n*k))
 * 空间复杂度：O(n*k + m*k)
 */
public static int findCheapestPrice(int n, int[][] flights, int src, int dst, int k) {
    // 构建图的邻接表
}

```

```

List<List<Edge>> edges = new ArrayList<>();
for (int i = 0; i < n; i++) {
    edges.add(new ArrayList<>());
}
for (int[] flight : flights) {
    edges.get(flight[0]).add(new Edge(flight[1], flight[2]));
}

// 调用分层图算法，注意这里 k 站中转意味着可以乘坐 k+1 次航班
return solve(n, edges, k + 1, src, dst);
}

}

// ====== 冷门模型：期望 DP 遇环的方程组解（高斯消元） ======
/***
 * 期望 DP 处理有环情况（使用高斯消元）
 *
 * 问题描述：
 * 在有环的状态转移图中计算期望
 *
 * 解题思路：
 * 1. 对于每个状态，建立期望方程
 * 2. 使用高斯消元求解方程组
 *
 * 应用题目：
 * - LeetCode 837. 新 21 点
 * - POJ 3744 Scout YYF I
 * - HDU 4405 Aeroplane chess
 *
 * 时间复杂度：O(n^3)
 * 空间复杂度：O(n^2)
 */
public static class ExpectationDPWithGaussian {

    /**
     * 高斯消元法求解线性方程组
     *
     * @param matrix 增广矩阵，每行最后一个元素是 b 的值
     * @return 方程组的解数组
     */
    public static double[] gaussianElimination(double[][] matrix) {
        int n = matrix.length;
        double eps = 1e-9;

```

```

// 高斯消元过程
for (int i = 0; i < n; i++) {
    // 找到主元行（当前列中绝对值最大的行）
    int maxRow = i;
    for (int j = i; j < n; j++) {
        if (Math.abs(matrix[j][i]) > Math.abs(matrix[maxRow][i])) {
            maxRow = j;
        }
    }
}

// 交换主元行和当前行
if (maxRow != i) {
    double[] temp = matrix[i];
    matrix[i] = matrix[maxRow];
    matrix[maxRow] = temp;
}

// 如果主元为 0， 方程组可能有无穷多解或无解
if (Math.abs(matrix[i][i]) < eps) {
    // 这里简化处理，假设方程组总是有解
    continue;
}

// 消元过程
for (int j = 0; j < n; j++) {
    if (j != i && Math.abs(matrix[j][i]) > eps) {
        double factor = matrix[j][i] / matrix[i][i];
        for (int k = i; k <= n; k++) {
            matrix[j][k] -= factor * matrix[i][k];
        }
    }
}

// 回代求解
double[] x = new double[n];
for (int i = 0; i < n; i++) {
    x[i] = matrix[i][n] / matrix[i][i];
}

return x;
}

```

```

/**
 * 期望 DP 主入口
 *
 * @param n 状态数量
 * @param transitions 转移概率列表, transitions[i]是一个列表, 每个元素为(j, p)表示从 i 转移到 j 的概率为 p
 * @param cost 每个状态的代价
 * @return 每个状态的期望值数组
 */
public static double[] solve(int n, List<List<Transition>> transitions, double[] cost) {
    // 构建线性方程组的增广矩阵
    double[][] matrix = new double[n][n + 1];

    for (int i = 0; i < n; i++) {
        matrix[i][i] = 1.0; // 方程左边: E[i] - sum(p_ij * E[j]) = cost[i]
        matrix[i][n] = cost[i];

        for (Transition t : transitions.get(i)) {
            if (i != t.to) { // 避免自环的特殊处理
                matrix[i][t.to] -= t.probability;
            }
        }
    }

    // 使用高斯消元求解
    return gaussianElimination(matrix);
}

/**
 * 转移类
 */
public static class Transition {
    int to;
    double probability;
    public Transition(int to, double probability) {
        this.to = to;
        this.probability = probability;
    }
}

/**
 * 应用示例: LeetCode 837. 新 21 点 (简化版本)
 * 时间复杂度: O(n^3)

```

```

* 空间复杂度: O(n^2)
*/
public static double new21Game(int N, int K, int W) {
    if (K == 0 || N >= K + W) return 1.0;

    int n = K + W;
    List<List<Transition>> transitions = new ArrayList<>();
    double[] cost = new double[n + 1];

    for (int i = 0; i <= n; i++) {
        transitions.add(new ArrayList<>());
    }

    // 构建转移概率
    for (int i = 0; i < K; i++) {
        for (int w = 1; w <= W; w++) {
            int next = Math.min(i + w, n);
            transitions.get(i).add(new Transition(next, 1.0 / W));
        }
    }

    // 终止状态的期望为是否<=N
    for (int i = K; i <= n; i++) {
        cost[i] = (i <= N) ? 1.0 : 0.0;
        transitions.get(i).add(new Transition(i, 1.0)); // 自环
    }

    double[] result = solve(n + 1, transitions, cost);
    return result[0];
}

}

// ===== 冷门模型: 插头 DP (轮廓线 DP) =====
/***
 * 插头 DP (轮廓线 DP) 示例: 求网格中哈密顿回路的数量
 *
 * 问题描述:
 * 给定一个网格, 求其中哈密顿回路的数量
 *
 * 解题思路:
 * 1. 使用轮廓线 DP, 记录当前处理到的位置和轮廓线状态
 * 2. 插头表示连接的状态, 通常用二进制表示
 * 3. 使用字典优化空间复杂度

```

```

* 4. 实现合法性判定与对称剪枝
*
* 应用题目：
* - HDU 1693 Eat the Trees
* - SPOJ MATCH2 Match the Brackets II
* - Codeforces 1435F Cyclic Shifts Sorting
*
* 时间复杂度: O(n*m*4^min(n, m))
* 空间复杂度: O(4^min(n, m))
*/
public static class PlugDP {
    /**
     * 插头 DP 求解哈密顿回路数量
     *
     * @param grid 网格，1 表示可通行，0 表示障碍物
     * @return 哈密顿回路的数量
     */
    public static long solve(int[][] grid) {
        if (grid == null || grid.length == 0 || grid[0].length == 0) return 0;
        int n = grid.length;
        int m = grid[0].length;

        // 使用 HashMap 优化空间
        Map<Long, Long> dp = new HashMap<>();
        dp.put(0L, 1L);

        for (int i = 0; i < n; i++) {
            // 新的一行开始，需要将状态左移两位
            Map<Long, Long> newDp = new HashMap<>();
            for (Map.Entry<Long, Long> entry : dp.entrySet()) {
                // 左移两位，移除最左边的插头
                long state = entry.getKey() << 2;
                // 移除可能的高位，只保留 m*2 位
                state &= (1L << (2 * m)) - 1;
                newDp.put(state, newDp.getOrDefault(state, 0L) + entry.getValue());
            }
            dp = newDp;

            for (int j = 0; j < m; j++) {
                Map<Long, Long> newDp2 = new HashMap<>();

                for (Map.Entry<Long, Long> entry : dp.entrySet()) {
                    long state = entry.getKey();

```

```

long cnt = entry.getValue();

// 当前位置左边和上边的插头状态
int left = (int) ((state >> (2 * j)) & 3);
int up = (int) ((state >> (2 * (j + 1))) & 3);

// 如果当前位置是障碍物，跳过
if (grid[i][j] == 0) {
    // 只有当左右插头都不存在时才合法
    if (left == 0 && up == 0) {
        newDp2.put(state, newDp2.getOrDefault(state, 0L) + cnt);
    }
    continue;
}

// 处理各种插头组合情况
// 1. 没有左插头和上插头
if (left == 0 && up == 0) {
    // 只能创建新的插头对（用于回路的开始）
    if (i < n - 1 && j < m - 1 && grid[i+1][j] == 1 && grid[i][j+1] == 1)
    {
        long newState = state | (1L << (2 * j)) | (2L << (2 * (j + 1)));
        newDp2.put(newState, newDp2.getOrDefault(newState, 0L) + cnt);
    }
}

// 2. 只有左插头
else if (left != 0 && up == 0) {
    // 向下延伸
    if (i < n - 1 && grid[i+1][j] == 1) {
        newDp2.put(state, newDp2.getOrDefault(state, 0L) + cnt);
    }
    // 向右延伸
    if (j < m - 1 && grid[i][j+1] == 1) {
        long newState = (state & ~(3L << (2 * j))) | ((long) left << (2 * (j + 1)));
        newDp2.put(newState, newDp2.getOrDefault(newState, 0L) + cnt);
    }
}

// 3. 只有上插头
else if (left == 0 && up != 0) {
    // 向右延伸
    if (j < m - 1 && grid[i][j+1] == 1) {
        newDp2.put(state, newDp2.getOrDefault(state, 0L) + cnt);
    }
}

```

```

        }

        // 向下延伸
        if (i < n - 1 && grid[i+1][j] == 1) {
            long newState = (state & ~(3L << (2 * (j + 1)))) | ((long) up <<
(2 * j));
            newDp2.put(newState, newDp2.getOrDefault(newState, 0L) + cnt);
        }
    }

    // 4. 同时有左插头和上插头
    else {
        // 合并插头
        long newState = (state & ~(3L << (2 * j))) & ~(3L << (2 * (j + 1)));

        // 如果是形成回路的最后一步
        if (left == up) {
            // 检查是否所有插头都已连接
            if (newState == 0 && i == n - 1 && j == m - 1) {
                newDp2.put(newState, newDp2.getOrDefault(newState, 0L) +
cnt);
            }
        } else {
            // 合并两个不同的插头
            // 这里可以加入更多的合法性检查和剪枝
            newDp2.put(newState, newDp2.getOrDefault(newState, 0L) + cnt);
        }
    }

    dp = newDp2;
}

}

// 最终状态应该是没有任何插头（形成回路）
return dp.getOrDefault(0L, 0L);
}

```

```

/***
 * 应用示例：网格中的回路计数
 * 时间复杂度：O(n*m*4^min(n, m))
 * 空间复杂度：O(4^min(n, m))
 */
public static long countGridCycles(int[][] grid) {
    return solve(grid);
}
```

```

    }

}

// ====== 冷门模型：树上背包的优化 ======
/***
 * 树上背包的优化实现（小到大合并）
 *
 * 问题描述：
 * 在树上选择一些节点，使得总重量不超过容量，且总价值最大
 *
 * 解题思路：
 * 1. 使用后序遍历处理子树
 * 2. 使用小到大合并的策略优化复杂度
 * 3. 对于每个节点，维护一个容量为 capacity 的背包
 *
 * 应用题目：
 * - HDU 1561 The more, The Better
 * - POJ 2063 Investment
 * - Codeforces 1152F2 Neko Rules the Catniverse
 *
 * 时间复杂度：O(n*capacity^2)，但通过小到大合并可以降低常数
 * 空间复杂度：O(n*capacity)
 */

public static class TreeKnapsackOptimized {
    private long[][] dp;
    private int[] size;
    private int[][] tree;
    private int[] weights;
    private int[] values;
    private int capacity;
    private int n;

    /**
     * 树上背包的 DFS 处理函数
     */
    private void dfs(int u, int parent) {
        // 初始化当前节点
        size[u] = 1;
        if (weights[u] <= capacity) {
            dp[u][weights[u]] = values[u];
        }
    }

    // 对每个子节点，按照子树大小排序，小的先合并
}

```

```

List<int[]> children = new ArrayList<>();
for (int v : tree[u]) {
    if (v != parent) {
        dfs(v, u);
        children.add(new int[]{size[v], v});
    }
}

// 按子树大小排序（小到大）
children.sort((a, b) -> a[0] - b[0]);

for (int[] child : children) {
    int sz = child[0];
    int v = child[1];

    // 逆序遍历容量，避免重复计算
    for (int i = Math.min(size[u], capacity); i >= 0; i--) {
        if (dp[u][i] == 0 && i != 0) continue;
        for (int j = 1; j <= Math.min(sz, capacity - i); j++) {
            if (dp[v][j] > 0 && i + j <= capacity) {
                dp[u][i + j] = Math.max(dp[u][i + j], dp[u][i] + dp[v][j]);
            }
        }
    }
}

// 更新子树大小
size[u] += sz;
}

}

/***
 * 树上背包主入口
 *
 * @param n 节点数量
 * @param root 根节点
 * @param capacity 背包容量
 * @param tree 树的邻接表
 * @param weights 每个节点的重量
 * @param values 每个节点的价值
 * @return 最大价值
 */
public long solve(int n, int root, int capacity, int[][] tree, int[] weights, int[]
values) {

```

```

    this.n = n;
    this.capacity = capacity;
    this.tree = tree;
    this.weights = weights;
    this.values = values;

    // 初始化 dp 数组
    dp = new long[n + 1][capacity + 1];
    size = new int[n + 1];

    // 深度优先搜索处理子树
    dfs(root, -1);

    // 返回根节点的最大价值
    long maxValue = 0;
    for (int i = 0; i <= capacity; i++) {
        maxValue = Math.max(maxValue, dp[root][i]);
    }
    return maxValue;
}

/**
 * 应用示例：树上最大价值选择
 * 时间复杂度：O(n*capacity^2)
 * 空间复杂度：O(n*capacity)
 */
public static long maxTreeValue(int n, int root, int capacity, int[][] tree, int[]
weights, int[] values) {
    TreeKnapsackOptimized optimizer = new TreeKnapsackOptimized();
    return optimizer.solve(n, root, capacity, tree, weights, values);
}

// ===== 补充题目与应用 =====
/**
 * LeetCode 72. 编辑距离
 * 题目链接：https://leetcode-cn.com/problems/edit-distance/
 *
 * 问题描述：
 * 给你两个单词 word1 和 word2，计算出将 word1 转换成 word2 所使用的最少操作数。
 * 你可以对一个单词进行如下三种操作：插入一个字符、删除一个字符、替换一个字符。
 *
 * 时间复杂度：O(m*n)

```

```

* 空间复杂度: O(m*n)
*/
public static int minDistance(String word1, String word2) {
    int m = word1.length();
    int n = word2.length();
    int[][] dp = new int[m + 1][n + 1];

    // 初始化
    for (int i = 0; i <= m; i++) dp[i][0] = i;
    for (int j = 0; j <= n; j++) dp[0][j] = j;

    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (word1.charAt(i-1) == word2.charAt(j-1)) {
                dp[i][j] = dp[i-1][j-1];
            } else {
                dp[i][j] = Math.min(Math.min(dp[i-1][j], dp[i][j-1]), dp[i-1][j-1]) + 1;
            }
        }
    }
    return dp[m][n];
}

/**
 * LeetCode 300. 最长递增子序列
 * 题目链接: https://leetcode-cn.com/problems/longest-increasing-subsequence/
 *
 * 问题描述:
 * 给你一个整数数组 nums，找到其中最长严格递增子序列的长度。
 *
 * 时间复杂度: O(n log n)
 * 空间复杂度: O(n)
 */
public static int lengthOfLIS(int[] nums) {
    if (nums == null || nums.length == 0) return 0;

    int[] tails = new int[nums.length];
    int len = 0;

    for (int num : nums) {
        int left = 0, right = len;
        while (left < right) {
            int mid = (left + right) / 2;

```

```

        if (tails[mid] < num) left = mid + 1;
        else right = mid;
    }

    tails[left] = num;
    if (left == len) len++;
}

return len;
}

/***
 * LeetCode 322. 零钱兑换
 * 题目链接: https://leetcode-cn.com/problems/coin-change/
 *
 * 问题描述:
 * 给你一个整数数组 coins，表示不同面额的硬币；以及一个整数 amount，表示总金额。
 * 计算并返回可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额，返回-1。
 *
 * 时间复杂度: O(amount * n)
 * 空间复杂度: O(amount)
 */
public static int coinChange(int[] coins, int amount) {
    int[] dp = new int[amount + 1];
    Arrays.fill(dp, amount + 1);
    dp[0] = 0;

    for (int i = 1; i <= amount; i++) {
        for (int coin : coins) {
            if (coin <= i) {
                dp[i] = Math.min(dp[i], dp[i - coin] + 1);
            }
        }
    }

    return dp[amount] > amount ? -1 : dp[amount];
}

/***
 * 矩阵链乘法问题
 * 题目来源: 算法导论、POJ 1038
 *
 */

```

```

* 问题描述:
* 给定一系列矩阵，计算乘法顺序使得标量乘法的次数最少。
*
* 时间复杂度: O(n^3)
* 空间复杂度: O(n^2)
*/
public static long matrixChainOrder(int[] p) {
    int n = p.length - 1; // 矩阵的个数
    long[][] dp = new long[n + 1][n + 1];

    for (int length = 2; length <= n; length++) {
        for (int i = 1; i <= n - length + 1; i++) {
            int j = i + length - 1;
            dp[i][j] = Long.MAX_VALUE;
            for (int k = i; k < j; k++) {
                long cost = dp[i][k] + dp[k + 1][j] + (long)p[i - 1] * p[k] * p[j];
                if (cost < dp[i][j]) {
                    dp[i][j] = cost;
                }
            }
        }
    }

    return dp[1][n];
}

/***
* 旅行商问题 (TSP)
* 题目来源: 算法竞赛、POJ 2480
*
* 问题描述:
* 给定一个完全图，找到一条访问每个城市恰好一次并返回起点的最短路径。
*
* 时间复杂度: O(n^2 * 2^n)
* 空间复杂度: O(n * 2^n)
*/
public static int tsp(int[][] graph) {
    int n = graph.length;
    int[][] dp = new int[1 << n][n];

    // 初始化
    for (int[] row : dp) {
        Arrays.fill(row, Integer.MAX_VALUE);
    }

```

```

}

// 从城市 0 出发
dp[1][0] = 0;

// 枚举所有可能的状态
for (int mask = 1; mask < (1 << n); mask++) {
    for (int u = 0; u < n; u++) {
        if ((mask & (1 << u)) == 0) continue;
        if (dp[mask][u] == Integer.MAX_VALUE) continue;

        for (int v = 0; v < n; v++) {
            if ((mask & (1 << v)) != 0) continue;
            if (graph[u][v] == Integer.MAX_VALUE) continue;

            dp[mask | (1 << v)][v] = Math.min(dp[mask | (1 << v)][v], dp[mask][u] +
graph[u][v]);
        }
    }
}

// 找到最短的回路
int result = Integer.MAX_VALUE;
for (int u = 0; u < n; u++) {
    if (graph[u][0] != Integer.MAX_VALUE) {
        result = Math.min(result, dp[(1 << n) - 1][u] + graph[u][0]);
    }
}

return result;
}
}

```

=====

文件: DPFusion.py

=====

```
# -*- coding: utf-8 -*-
"""


```

高级动态规划算法融合实现
包含多种优化技术和应用示例
,,,

```
import sys
```

```

import math
import heapq
from collections import defaultdict, deque

class DPFusion:
    """
    动态规划融合类：包含多种高级 DP 优化技术和算法
    """

    # ===== 高级优化体系：Knuth 优化 =====
    """

    Knuth 优化的 DP 算法

```

问题描述：

解决区间 DP 问题，其中状态转移方程满足四边形不等式

解题思路：

1. 使用 Knuth 优化将时间复杂度从 $O(n^3)$ 降低到 $O(n^2)$
2. 维护最优转移点数组 $opt[i][j]$ ，表示计算 $dp[i][j]$ 时的最优 k 值
3. 根据 $opt[i][j-1] \leq opt[i][j] \leq opt[i+1][j]$ 的性质进行剪枝

应用题目：

- POJ 1160 Post Office
- HDU 4008 Parent and son
- Codeforces 245H Queries for Number of Palindromes

时间复杂度： $O(n^2)$

空间复杂度： $O(n^2)$

"""

```
class KnuthOptimization:
```

"""

Knuth 优化实现类

"""

@staticmethod

```
def solve(n, cost_func):
    """

```

Knuth 优化的 DP 实现

参数：

n : 区间长度

$cost_func$: 计算区间 (i, j) 代价的函数

返回：

最小代价矩阵

"""

初始化 dp 和 opt 数组

INF = float('inf')

dp = [[INF] * (n + 1) for _ in range(n + 1)]

opt = [[0] * (n + 1) for _ in range(n + 1)]

初始化长度为 1 的区间

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

dp[i][i] = 0

opt[i][i] = i

枚举区间长度

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

枚举起始点

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

j = i + length - 1

根据 Knuth 优化的性质，最优 k 在 opt[i][j-1] 到 opt[i+1][j] 之间

lower = opt[i][j-1]

upper = opt[i+1][j] if (i + 1 <= j) else j - 1

dp[i][j] = INF

for k in range(lower, upper + 1):

if dp[i][k] != INF and dp[k+1][j] != INF:

cost = cost_func(i, j)

if cost != INF:

current = dp[i][k] + dp[k+1][j] + cost

if current < dp[i][j]:

dp[i][j] = current

opt[i][j] = k

return dp

@staticmethod

def optimal_binary_search_tree(keys, freq):

"""

应用示例：最优二叉搜索树问题（POJ 3280）

参数：

keys: 键数组

freq: 频率数组

返回：

最小代价

时间复杂度： $O(n^2)$

空间复杂度： $O(n^2)$

"""

`n = len(keys)`

`def cost_func(i, j):`

 """计算从 i 到 j 的频率和"""

 return sum(freq[k] for k in range(i-1, j))

`dp = DPFusion.KnuthOptimization.solve(n, cost_func)`

`return dp[1][n]`

===== 高级优化体系: Divide & Conquer Optimization =====

"""

分治优化 (Divide & Conquer Optimization)

问题描述：

解决形如 $dp[i][j] = \min\{dp[i-1][k] + cost(k, j)\}$, 其中 $k < j$

当转移满足决策单调性时使用

解题思路：

1. 利用决策单调性，使用分治法优化 DP
2. 对于 $dp[i][j]$, 当 i 固定时，最优转移点 k 随着 j 的增加而单调不减
3. 使用分治的方式计算每个区间的最优决策

应用题目：

- Codeforces 321E Ciel and Gondolas
- HDU 3480 Division
- SPOJ LARMY

时间复杂度： $O(n*m \log m)$

空间复杂度： $O(n*m)$

"""

`class DivideConquerOptimization:`

 """

 分治优化实现类

 """

`def __init__(self):`

`self.dp = None`

```

self.cost_func = None
self.n = 0
self.m = 0

def _solve_subproblem(self, i, l, r, opt_l, opt_r):
    """
    分治求解 dp[i][l..r], 其中最优转移点在 opt_l..opt_r 之间
    """
    if l > r:
        return

    mid = (l + r) // 2
    best_k = opt_l
    self.dp[i][mid] = float('inf')

    # 在 opt_l 到 min(mid, opt_r) 之间寻找最优 k
    for k in range(opt_l, min(mid, opt_r) + 1):
        if self.dp[i-1][k] != float('inf'):
            cost = self.cost_func(k, mid)
            if cost != float('inf'):
                current = self.dp[i-1][k] + cost
                if current < self.dp[i][mid]:
                    self.dp[i][mid] = current
                    best_k = k

    # 递归处理左右子区间
    self._solve_subproblem(i, l, mid - 1, opt_l, best_k)
    self._solve_subproblem(i, mid + 1, r, best_k, opt_r)

def solve(self, n, m, cost_func):
    """
    主入口: 分治优化 DP
    """

```

参数:

- n: 维度 1
- m: 维度 2
- cost_func: 计算 cost(k, j) 的函数

返回:

dp 数组

```

    """
# 初始化 dp 数组
INF = float('inf')

```

```

self.dp = [[INF] * (m + 1) for _ in range(n + 1)]
self.cost_func = cost_func
self.n = n
self.m = m

self.dp[0][0] = 0 # 初始状态

# 对每个 i 应用分治优化
for i in range(1, n + 1):
    self._solve_subproblem(i, 1, m, 0, m)

return self.dp

@staticmethod
def split_array(nums, k):
    """
    应用示例：将数组分成 k 个子数组的最小代价和（LeetCode 410）

    参数：
        nums：输入数组
        k：子数组数量

    返回：
        最小代价和

    时间复杂度：O(n*k log n)
    空间复杂度：O(n*k)
    """

    n = len(nums)
    # 预处理前缀和
    prefix_sum = [0] * (n + 1)
    for i in range(n):
        prefix_sum[i+1] = prefix_sum[i] + nums[i]

    # 代价函数：计算从 k+1 到 j 的和的平方
    def cost_func(k, j):
        sum_val = prefix_sum[j] - prefix_sum[k]
        return sum_val * sum_val # 这里可以根据实际问题定义不同的代价

    optimizer = DPfusion.DivideConquerOptimization()
    dp = optimizer.solve(k, n, cost_func)
    return dp[k][n]

```

```

sys.setrecursionlimit(1 << 25)

# ===== 高级优化体系: SMAWK 算法 (行最小查询) =====
"""

SMAWK 算法用于在 Monge 矩阵中快速查找每行的最小值

```

问题描述:

给定一个 Monge 矩阵, 快速找到每行的最小值位置

解题思路:

1. Monge 矩阵满足性质: $\text{matrix}[i][j] + \text{matrix}[i+1][j+1] \leq \text{matrix}[i][j+1] + \text{matrix}[i+1][j]$
2. SMAWK 算法利用这一性质, 可以在 $O(m+n)$ 时间内找到每行的最小值
3. 主要步骤包括行压缩和递归求解

应用题目:

- POJ 3156 Interconnect
- Codeforces 472D Design Tutorial: Inverse the Problem
- SPOJ MCQUERY

时间复杂度: $O(m+n)$, 其中 m 是行数, n 是列数

空间复杂度: $O(m+n)$

"""

```
class SMAWK:
```

"""

SMAWK 算法实现类

"""

@staticmethod

```
def reduce_rows(rows, matrix):
```

"""

行压缩: 只保留可能成为最小值的行

"""

```
stack = []
```

```
for i in rows:
```

```
    while len(stack) >= 2:
```

```
        j1 = stack.pop()
```

```
        j2 = stack.pop()
```

```
        stack.append(j1) # 恢复栈状态
```

比较两个行在列 `stack.size() - 2` 处的值

```
col = len(stack) - 2
```

```
if col < len(matrix[0]):
```

```
    if matrix[j2][col] <= matrix[i][col]:
```

```

        break
    else:
        stack.pop()  # 移除 j1
    else:
        break
    stack.append(i)
return stack

@staticmethod
def smawk_rec(rows, cols, matrix):
    """
    递归实现 SMAWK 算法
    """

    m = len(rows)
    result = [-1] * m

    if m == 0:
        return result

    # 行压缩
    reduced_rows = DPFusion.SMAWK.reduce_rows(rows, matrix)

    # 递归求解列数为奇数的子问题
    half_cols = [cols[i] for i in range(1, len(cols), 2)]

    min_cols = [-1] * len(reduced_rows)

    if half_cols:
        # 递归求解
        sub_result = DPFusion.SMAWK.smawk_rec(reduced_rows, half_cols, matrix)
        # 复制结果
        for i in range(len(sub_result)):
            min_cols[i] = sub_result[i]

    # 扩展结果到所有列
    k = 0  # min_cols 的索引

    for i in range(m):
        row = rows[i]
        # 确定当前行的最小值可能在哪个区间
        start = 0
        if i > 0 and k > 0 and min_cols[k-1] != -1:
            start = min_cols[k-1]

```

```

end = min_cols[k] if (k < len(min_cols) and min_cols[k] != -1) else cols[-1]

# 在这个区间内查找最小值
min_val = float('inf')
min_col = start

# 找到 start 和 end 在 cols 中的索引
try:
    start_idx = cols.index(start)
    end_idx = cols.index(end)
except ValueError:
    start_idx = 0
    end_idx = len(cols) - 1

for idx in range(start_idx, end_idx + 1):
    col = cols[idx]
    if col < len(matrix[0]) and matrix[row][col] < min_val:
        min_val = matrix[row][col]
        min_col = col

result[i] = min_col

# 如果当前行在 reduced_rows 中，且不是最后一行，k 前进
if k < len(reduced_rows) and row == reduced_rows[k]:
    k += 1

return result

```

@staticmethod

def solve(matrix):

"""

SMAWK 算法主入口

参数:

matrix: 一个 Monge 矩阵

返回:

每行最小值的列索引数组

"""

if not matrix or not matrix[0]:

return []

m = len(matrix)

```

n = len(matrix[0])

# 构造行索引和列索引数组
rows = list(range(m))
cols = list(range(n))

# 调用递归实现
return DPFusion.SMAWK.smawk_rec(rows, cols, matrix)

```

```

@staticmethod
def find_row_mins(matrix):
    """

```

应用示例：寻找每一行的最小元素

参数：

matrix：输入矩阵

返回：

每行最小元素组成的数组

时间复杂度： $O(m+n)$

空间复杂度： $O(m+n)$

"""

```

min_cols = DPFusion.SMAWK.solve(matrix)
result = [matrix[i][min_cols[i]] for i in range(len(matrix))]
return result

```

===== 高级优化体系：Aliens Trick (二分约束参数+可行性 DP)

=====

"""

Aliens Trick (二分约束参数+可行性 DP)

问题描述：

解决带约束的优化问题，通常形如最小化总成本，同时满足某些约束条件

解题思路：

1. 将约束条件转化为参数 λ ，构造拉格朗日函数
2. 对 λ 进行二分查找，使用可行性 DP 判断当前 λ 下是否满足约束
3. 根据可行性 DP 的结果调整二分区间

应用题目：

- Codeforces 739E Gosha is Hunting
- POJ 3686 The Windy's

- SPOJ QTREE5

时间复杂度: $O(\log((right-left)/\text{eps}) * T(DP))$, 其中 $T(DP)$ 是一次 DP 的时间复杂度
空间复杂度: $O(DP \text{ 空间复杂度})$

"""

class AliensTrick:

"""

Aliens Trick 实现类

"""

class Result:

"""

结果类

"""

def __init__(self, lambda_val, value):

self.lambda_val = lambda_val

self.value = value

class CostFunctionResult:

"""

成本函数结果类

"""

def __init__(self, value, constraint):

self.value = value

self.constraint = constraint

@staticmethod

def solve(cost_func, check_func, left, right, eps=1e-7):

"""

Aliens Trick 主入口

参数:

cost_func: 成本函数, 输入 lambda, 返回 CostFunctionResult

check_func: 约束检查函数, 输入约束值, 返回是否满足约束

left: 二分左边界

right: 二分右边界

eps: 精度要求

返回:

Result 对象, 包含最优参数 lambda 和对应最优解

"""

best_lambda = left

best_value = 0.0

```

# 二分查找参数 lambda
while right - left > eps:
    mid = (left + right) / 2
    # 计算当前参数下的解和约束值
    result = cost_func(mid)

    if check_func(result.constraint):
        # 满足约束，尝试更小的参数
        right = mid
        best_lambda = mid
        best_value = result.value
    else:
        # 不满足约束，需要增大参数
        left = mid

return DPFusion.AliensTrick.Result(best_lambda, best_value)

```

@staticmethod

```

def split_array_k(nums, k):
    """

```

应用示例：将数组分成恰好 k 个部分，使得最大子数组和最小（LeetCode 410 的变种）

参数：

nums：输入数组

k：子数组数量

返回：

最小代价和

时间复杂度： $O(\log(S) * n^2)$ ，其中 S 是数组元素和

空间复杂度： $O(n)$

"""

计算数组元素和作为二分上限

sum_val = sum(nums)

成本函数：使用 DP 计算在给定 lambda 下的最小成本

```

def cost_func(lambda_val):
    n = len(nums)
    INF = float('inf')
    dp = [INF] * (n + 1)
    cnt = [0] * (n + 1)

```

```

dp[0] = 0
cnt[0] = 0

for i in range(1, n + 1):
    sum_seg = 0
    for j in range(i-1, -1, -1):
        sum_seg += nums[j]
        if dp[j] != INF:
            current = dp[j] + sum_seg * sum_seg + lambda_val # lambda 作为惩罚项
            if current < dp[i]:
                dp[i] = current
                cnt[i] = cnt[j] + 1

return DPFusion.AliensTrick.CostFunctionResult(dp[n], cnt[n])

# 约束检查函数: 确保分割次数不超过 k
check_func = lambda constraint: constraint <= k

# 执行 Aliens Trick
result = DPFusion.AliensTrick.solve(cost_func, check_func, 0, sum_val * sum_val, 1e-7)
return result.value

# ===== 图上 DP→最短路: 分层图建模 =====
"""
分层图 Dijkstra 算法

```

问题描述:

给定一个图, 允许最多使用 k 次特殊操作 (如跳跃、免费通行等), 求最短路径

解题思路:

1. 构建分层图, 每层代表使用不同次数的特殊操作
2. 对于每个节点 u , 在第 i 层表示到达 u 时已经使用了 i 次特殊操作
3. 使用 Dijkstra 算法在分层图上寻找最短路径

应用题目:

- LeetCode 787. K 站中转内最便宜的航班
- POJ 3159 Candies
- HDU 2957 Safety Assessment

时间复杂度: $O((n*k + m*k) \log(n*k))$

空间复杂度: $O(n*k + m*k)$

"""

```

class LayeredGraphShortestPath:
    """
    分层图最短路径算法实现类
    """

    class Edge:
        """
        边类
        """

        def __init__(self, to, weight):
            self.to = to
            self.weight = weight

    @staticmethod
    def solve(n, edges, k, start, end):
        """
        分层图最短路径算法

        参数:
            n: 节点数量
            edges: 边的列表, edges[i]是节点 i 的边列表
            k: 允许使用的特殊操作次数
            start: 起始节点
            end: 目标节点

        返回:
            最短路径长度, -1 表示不可达
        """

        # 构建分层图的邻接表
        layered_graph = [[] for _ in range(n * (k + 1))]
        total_nodes = n * (k + 1)

        # 添加普通边 (不使用特殊操作)
        for i in range(n):
            for j in range(k + 1):
                from_node = i + j * n
                for edge in edges[i]:
                    layered_graph[from_node].append(DPFusion.LayeredGraphShortestPath.Edge(
                        edge.to + j * n, edge.weight))

        # 添加使用特殊操作的边 (如果允许的话)
        for i in range(n):
            for j in range(k):

```

```

from_node = i + j * n
for edge in edges[i]:
    # 这里假设特殊操作可以免费通行（权为 0），具体根据问题调整
    layered_graph[from_node].append(DPFusion.LayeredGraphShortestPath.Edge(
        edge.to + (j + 1) * n, 0))

# Dijkstra 算法
INF = float('inf')
dist = [INF] * total_nodes
dist[start] = 0 # 起始点在第 0 层

# 使用优先队列，按距离排序
# (距离, 节点)的元组
pq = [(0, start)]

while pq:
    current_dist, u = heapq.heappop(pq)

    if current_dist > dist[u]:
        continue

    for edge in layered_graph[u]:
        v = edge.to
        w = edge.weight
        if dist[v] > current_dist + w:
            dist[v] = current_dist + w
            heapq.heappush(pq, (dist[v], v))

# 取所有层中到达终点的最小值
result = min(dist[end + i * n] for i in range(k + 1))

return int(result) if result != INF else -1

```

@staticmethod

```

def find_cheapest_price(n, flights, src, dst, k):
    """
    应用示例：LeetCode 787. K 站中转内最便宜的航班
    
```

参数：

n: 城市数量
flights: 航班列表，每个元素为[from, to, price]
src: 出发城市
dst: 到达城市

k: 允许的中转次数

返回:

最便宜的价格, -1 表示不可达

时间复杂度: $O((n*k + m*k) \log(n*k))$

空间复杂度: $O(n*k + m*k)$

"""

构建图的邻接表

```
edges = [[] for _ in range(n)]
```

```
for flight in flights:
```

```
    edges[flight[0]].append(DPFusion.LayeredGraphShortestPath.Edge(  
        flight[1], flight[2]))
```

调用分层图算法, 注意这里 k 站中转意味着可以乘坐 k+1 次航班

```
return DPFusion.LayeredGraphShortestPath.solve(n, edges, k + 1, src, dst)
```

====== 冷门模型: 期望 DP 遇环的方程组解 (高斯消元) ======

"""

期望 DP 处理有环情况 (使用高斯消元)

问题描述:

在有环的状态转移图中计算期望

解题思路:

1. 对于每个状态, 建立期望方程
2. 使用高斯消元求解方程组

应用题目:

- LeetCode 837. 新 21 点
- POJ 3744 Scout YYF I
- HDU 4405 Aeroplane chess

时间复杂度: $O(n^3)$

空间复杂度: $O(n^2)$

"""

```
class ExpectationDPWithGaussian:
```

"""

期望 DP 与高斯消元实现类

"""

```
class Transition:
```

"""

转移类

"""

```
def __init__(self, to, probability):
    self.to = to
    self.probability = probability
```

@staticmethod

```
def gaussian_elimination(matrix):
    """
```

高斯消元法求解线性方程组

参数:

matrix: 增广矩阵, 每行最后一个元素是 b 的值

返回:

方程组的解数组

"""

```
n = len(matrix)
```

```
eps = 1e-9
```

高斯消元过程

```
for i in range(n):
```

找到主元行 (当前列中绝对值最大的行)

```
max_row = i
```

```
for j in range(i, n):
```

```
    if abs(matrix[j][i]) > abs(matrix[max_row][i]):
```

```
        max_row = j
```

交换主元行和当前行

```
if max_row != i:
```

```
    matrix[i], matrix[max_row] = matrix[max_row], matrix[i]
```

如果主元为 0, 方程组可能有无穷多解或无解

```
if abs(matrix[i][i]) < eps:
```

这里简化处理, 假设方程组总是有解

```
continue
```

消元过程

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

```
    if j != i and abs(matrix[j][i]) > eps:
```

```
        factor = matrix[j][i] / matrix[i][i]
```

```
        for k in range(i, n + 1):
```

```
            matrix[j][k] -= factor * matrix[i][k]
```

```

# 回代求解
x = [0.0] * n
for i in range(n):
    x[i] = matrix[i][n] / matrix[i][i]

return x

@staticmethod
def solve(n, transitions, cost):
    """
    期望 DP 主入口

参数:
    n: 状态数量
    transitions: 转移概率列表, transitions[i]是状态 i 的转移列表
    cost: 每个状态的代价

返回:
    每个状态的期望值数组
    """

# 构建线性方程组的增广矩阵
matrix = [[0.0] * (n + 1) for _ in range(n)]

for i in range(n):
    matrix[i][i] = 1.0 # 方程左边: E[i] - sum(p_ij * E[j]) = cost[i]
    matrix[i][n] = cost[i]

    for t in transitions[i]:
        if i != t.to: # 避免自环的特殊处理
            matrix[i][t.to] -= t.probability

# 使用高斯消元求解
return DPfusion.ExpectationDPWithGaussian.gaussian_elimination(matrix)

```

```

@staticmethod
def new21game(N, K, W):
    """

```

应用示例: LeetCode 837. 新 21 点 (简化版本)

参数:

N: 目标值
K: 停止抽牌的阈值

W: 每张牌的最大值

返回：

获胜的概率

时间复杂度： $O(n^3)$

空间复杂度： $O(n^2)$

"""

if K == 0 or N >= K + W:

 return 1.0

n = K + W

transitions = [[] for _ in range(n + 1)]

cost = [0.0] * (n + 1)

构建转移概率

for i in range(K):

 for w in range(1, W + 1):

 next_state = min(i + w, n)

 transitions[i].append(DPFusion.ExpectationDPWithGaussian.Transition(
 next_state, 1.0 / W))

终止状态的期望为是否<=N

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

 cost[i] = 1.0 if i <= N else 0.0

 transitions[i].append(DPFusion.ExpectationDPWithGaussian.Transition(i, 1.0)) #

自环

result = DPFusion.ExpectationDPWithGaussian.solve(n + 1, transitions, cost)

return result[0]

===== 冷门模型：插头 DP（轮廓线 DP）=====

"""

插头 DP（轮廓线 DP）示例：求网格中哈密顿回路的数量

问题描述：

给定一个网格，求其中哈密顿回路的数量

解题思路：

1. 使用轮廓线 DP，记录当前处理到的位置和轮廓线状态
2. 插头表示连接的状态，通常用二进制表示
3. 使用字典优化空间复杂度
4. 实现合法性判定与对称剪枝

应用题目：

- HDU 1693 Eat the Trees
- SPOJ MATCH2 Match the Brackets II
- Codeforces 1435F Cyclic Shifts Sorting

时间复杂度： $O(n*m*4^{\min(n, m)})$

空间复杂度： $O(4^{\min(n, m)})$

"""

```
class PlugDP:
```

"""

插头 DP 实现类

"""

@staticmethod

```
    def solve(grid):
```

"""

插头 DP 求解哈密顿回路数量

参数：

grid: 网格，1 表示可通行，0 表示障碍物

返回：

哈密顿回路的数量

"""

```
    if not grid or not grid[0]:  
        return 0
```

```
    n = len(grid)
```

```
    m = len(grid[0])
```

使用字典优化空间

```
dp = defaultdict(int)
```

```
dp[0] = 1
```

```
for i in range(n):
```

新的一行开始，需要将状态左移两位

```
new_dp = defaultdict(int)
```

```
for state, cnt in dp.items():
```

左移两位，移除最左边的插头

```
new_state = state << 2
```

移除可能的高位，只保留 $m*2$ 位

```
new_state &= (1 << (2 * m)) - 1
```

```

new_dp[new_state] += cnt
dp = new_dp

for j in range(m):
    new_dp2 = defaultdict(int)

    for state, cnt in dp.items():
        # 当前位置左边和上边的插头状态
        left = (state >> (2 * j)) & 3
        up = (state >> (2 * (j + 1))) & 3

        # 如果当前位置是障碍物，跳过
        if grid[i][j] == 0:
            # 只有当左右插头都不存在时才合法
            if left == 0 and up == 0:
                new_dp2[state] += cnt
            continue

        # 处理各种插头组合情况
        # 1. 没有左插头和上插头
        if left == 0 and up == 0:
            # 只能创建新的插头对（用于回路的开始）
            if (i < n - 1 and j < m - 1 and
                grid[i+1][j] == 1 and grid[i][j+1] == 1):
                new_state = state | (1 << (2 * j)) | (2 << (2 * (j + 1)))
                new_dp2[new_state] += cnt

        # 2. 只有左插头
        elif left != 0 and up == 0:
            # 向下延伸
            if i < n - 1 and grid[i+1][j] == 1:
                new_dp2[state] += cnt
            # 向右延伸
            if j < m - 1 and grid[i][j+1] == 1:
                new_state = (state & ~(3 << (2 * j))) | (left << (2 * (j + 1)))
                new_dp2[new_state] += cnt

        # 3. 只有上插头
        elif left == 0 and up != 0:
            # 向右延伸
            if j < m - 1 and grid[i][j+1] == 1:
                new_dp2[state] += cnt
            # 向下延伸
            if i < n - 1 and grid[i+1][j] == 1:
                new_state = (state & ~(3 << (2 * (j + 1)))) | (up << (2 * j))
                new_dp2[new_state] += cnt

```

```

        new_dp2[new_state] += cnt
    # 4. 同时有左插头和上插头
    else:
        # 合并插头
        new_state = (state & ~(3 << (2 * j))) & ~(3 << (2 * (j + 1)))

    # 如果是形成回路的最后一步
    if left == up:
        # 检查是否所有插头都已连接
        if new_state == 0 and i == n - 1 and j == m - 1:
            new_dp2[new_state] += cnt
    else:
        # 合并两个不同的插头
        # 这里可以加入更多的合法性检查和剪枝
        new_dp2[new_state] += cnt

dp = new_dp2

# 最终状态应该是没有任何插头（形成回路）
return dp.get(0, 0)

```

```

@staticmethod
def count_grid_cycles(grid):
    """

```

应用示例：网格中的回路计数

参数：

grid：输入网格

返回：

回路数量

时间复杂度：O(n*m*4^min(n, m))

空间复杂度：O(4^min(n, m))

"""

```
return DPFusion.PlugDP.solve(grid)
```

===== 冷门模型：树上背包的优化 =====

"""

树上背包的优化实现（小到大合并）

问题描述：

在树上选择一些节点，使得总重量不超过容量，且总价值最大

解题思路：

1. 使用后序遍历处理子树
2. 使用小到大合并的策略优化复杂度
3. 对于每个节点，维护一个容量为 capacity 的背包

应用题目：

- HDU 1561 The more, The Better
- POJ 2063 Investment
- Codeforces 1152F2 Neko Rules the Catniverse

时间复杂度： $O(n * capacity^2)$ ，但通过小到大合并可以降低常数

空间复杂度： $O(n * capacity)$

"""

```
class TreeKnapsackOptimized:
```

"""

优化的树上背包实现类

"""

```
    def __init__(self):
```

```
        self.dp = None
```

```
        self.size = None
```

```
        self.tree = None
```

```
        self.weights = None
```

```
        self.values = None
```

```
        self.capacity = 0
```

```
        self.n = 0
```

```
    def _dfs(self, u, parent):
```

"""

树上背包的 DFS 处理函数

"""

```
        # 初始化当前节点
```

```
        self.size[u] = 1
```

```
        if self.weights[u] <= self.capacity:
```

```
            self.dp[u][self.weights[u]] = max(self.dp[u][self.weights[u]], self.values[u])
```

```
        # 对每个子节点，按照子树大小排序，小的先合并
```

```
        children = []
```

```
        for v in self.tree[u]:
```

```
            if v != parent:
```

```
                self._dfs(v, u)
```

```
                children.append((self.size[v], v))
```

```
# 按子树大小排序（小到大）
children.sort(key=lambda x: x[0])

for sz, v in children:
    # 逆序遍历容量，避免重复计算
    for i in range(min(self.size[u], self.capacity), -1, -1):
        if self.dp[u][i] == 0 and i != 0:
            continue
        for j in range(1, min(sz, self.capacity - i) + 1):
            if self.dp[v][j] > 0 and i + j <= self.capacity:
                self.dp[u][i + j] = max(self.dp[u][i + j], self.dp[u][i] +
self.dp[v][j])
```

更新子树大小

```
self.size[u] += sz
```

```
def solve(self, n, root, capacity, tree, weights, values):
```

```
"""

```

树上背包主入口

参数:

n: 节点数量

root: 根节点

capacity: 背包容量

tree: 树的邻接表

weights: 每个节点的重量

values: 每个节点的价值

返回:

最大价值

```
"""

```

```
self.n = n
```

```
self.capacity = capacity
```

```
self.tree = tree
```

```
self.weights = weights
```

```
self.values = values
```

初始化 dp 数组

```
self.dp = [[0] * (capacity + 1) for _ in range(n + 1)]
```

```
self.size = [0] * (n + 1)
```

深度优先搜索处理子树

```
self._dfs(root, -1)

# 返回根节点的最大价值
max_value = max(self.dp[root])
return max_value

@staticmethod
def max_tree_value(n, root, capacity, tree, weights, values):
    """
```

应用示例：树上最大价值选择

参数：

n: 节点数量
root: 根节点
capacity: 背包容量
tree: 树的邻接表
weights: 每个节点的重量
values: 每个节点的价值

返回：

最大价值

时间复杂度： $O(n * capacity^2)$

空间复杂度： $O(n * capacity)$

"""

```
optimizer = DPFusion.TreeKnapsackOptimized()
return optimizer.solve(n, root, capacity, tree, weights, values)
```

===== 补充题目与应用 =====

"""

LeetCode 72. 编辑距离

题目链接：<https://leetcode-cn.com/problems/edit-distance/>

问题描述：

给你两个单词 word1 和 word2，计算出将 word1 转换成 word2 所使用的最少操作数。

你可以对一个单词进行如下三种操作：插入一个字符、删除一个字符、替换一个字符。

时间复杂度： $O(m * n)$

空间复杂度： $O(m * n)$

"""

@staticmethod

```
def min_distance(word1, word2):
    """
```

计算编辑距离

"""

```
m = len(word1)
n = len(word2)
# 初始化 dp 数组
dp = [[0] * (n + 1) for _ in range(m + 1)]
```

初始化边界条件

```
for i in range(m + 1):
    dp[i][0] = i
for j in range(n + 1):
    dp[0][j] = j
```

填充 dp 表

```
for i in range(1, m + 1):
    for j in range(1, n + 1):
        if word1[i-1] == word2[j-1]:
            dp[i][j] = dp[i-1][j-1]
        else:
            dp[i][j] = min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1]) + 1
```

```
return dp[m][n]
```

"""

LeetCode 300. 最长递增子序列

题目链接: <https://leetcode-cn.com/problems/longest-increasing-subsequence/>

问题描述:

给你一个整数数组 `nums`，找到其中最长严格递增子序列的长度。

时间复杂度: $O(n \log n)$

空间复杂度: $O(n)$

"""

@staticmethod

```
def length_of_lis(nums):
```

"""

计算最长递增子序列长度

"""

```
if not nums:
```

```
    return 0
```

```
tails = []
```

```

for num in nums:
    # 二分查找 tails 中第一个大于等于 num 的位置
    left, right = 0, len(tails)
    while left < right:
        mid = (left + right) // 2
        if tails[mid] < num:
            left = mid + 1
        else:
            right = mid

    if left == len(tails):
        tails.append(num)
    else:
        tails[left] = num

return len(tails)

```

"""

LeetCode 322. 零钱兑换

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

问题描述:

给你一个整数数组 coins，表示不同面额的硬币；以及一个整数 amount，表示总金额。

计算并返回可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额，返回-1。

时间复杂度: O(amount * n)

空间复杂度: O(amount)

"""

@staticmethod

def coin_change(coins, amount):

"""

计算凑成总金额所需的最少硬币个数

"""

初始化 dp 数组，dp[i] 表示凑成金额 i 所需的最少硬币数

dp = [amount + 1] * (amount + 1)

dp[0] = 0 # 基础情况

for i in range(1, amount + 1):

for coin in coins:

if coin <= i:

dp[i] = min(dp[i], dp[i - coin] + 1)

return dp[amount] if dp[amount] <= amount else -1

```
"""
```

矩阵链乘法问题

题目来源：算法导论、POJ 1038

问题描述：

给定一系列矩阵，计算乘法顺序使得标量乘法的次数最少。

时间复杂度： $O(n^3)$

空间复杂度： $O(n^2)$

```
"""
```

```
@staticmethod
```

```
def matrix_chain_order(p):
```

```
    """
```

计算矩阵链乘法的最优顺序

参数：

p：维度数组， $p[i-1]$ 和 $p[i]$ 分别是矩阵 A_i 的行数和列数

返回：

最少标量乘法次数

```
"""
```

```
n = len(p) - 1 # 矩阵的个数
```

```
# 初始化 dp 数组
```

```
dp = [[0] * (n + 1) for _ in range(n + 1)]
```

```
# 枚举子链长度
```

```
for length in range(2, n + 1):
```

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

```
        j = i + length - 1
```

```
        dp[i][j] = float('inf')
```

```
# 枚举分割点
```

```
        for k in range(i, j):
```

```
            # 计算在位置 k 分割的代价
```

```
            cost = dp[i][k] + dp[k+1][j] + p[i-1] * p[k] * p[j]
```

```
            if cost < dp[i][j]:
```

```
                dp[i][j] = cost
```

```
return dp[1][n]
```

```
"""
```

旅行商问题（TSP）

题目来源：算法竞赛、POJ 2480

问题描述：

给定一个完全图，找到一条访问每个城市恰好一次并返回起点的最短路径。

时间复杂度： $O(n^2 * 2^n)$

空间复杂度： $O(n * 2^n)$

"""

@staticmethod

def tsp(graph):

"""

解决旅行商问题

参数：

graph: 邻接矩阵，表示城市间的距离

返回：

最短路径长度

"""

n = len(graph)

dp[mask][u] 表示访问过的城市集合为 mask，当前在城市 u 时的最短路径

dp = [[float('inf')] * n for _ in range(1 << n)]

初始化：从城市 0 出发

dp[1][0] = 0

枚举所有可能的状态

for mask in range(1, 1 << n):

for u in range(n):

if not (mask & (1 << u)): # 如果 u 不在 mask 中，跳过

continue

if dp[mask][u] == float('inf'): # 如果无法到达 u，跳过

continue

尝试从 u 出发访问未访问的城市 v

for v in range(n):

if mask & (1 << v): # 如果 v 已经访问过，跳过

continue

if graph[u][v] == float('inf'): # 如果 u 和 v 之间没有边，跳过

continue

new_mask = mask | (1 << v)

if dp[new_mask][v] > dp[mask][u] + graph[u][v]:

dp[new_mask][v] = dp[mask][u] + graph[u][v]

```

# 找到最短的回路
result = min(dp[(1 << n) - 1][u] + graph[u][0] for u in range(n)
            if graph[u][0] != float('inf'))

return result if result != float('inf') else -1

# 测试代码
if __name__ == "__main__":
    # 测试编辑距离
    print("编辑距离测试:")
    print(DPFusion.min_distance("horse", "ros")) # 预期输出: 3
    print(DPFusion.min_distance("intention", "execution")) # 预期输出: 5

    # 测试最长递增子序列
    print("\n 最长递增子序列测试:")
    print(DPFusion.length_of_lis([10, 9, 2, 5, 3, 7, 101, 18])) # 预期输出: 4

    # 测试零钱兑换
    print("\n 零钱兑换测试:")
    print(DPFusion.coin_change([1, 2, 5], 11)) # 预期输出: 3

    # 测试矩阵链乘法
    print("\n 矩阵链乘法测试:")
    print(DPFusion.matrix_chain_order([30, 35, 15, 5, 10, 20, 25])) # 预期输出: 15125

    # 测试 TSP
    print("\nTSP 测试:")
    INF = float('inf')
    graph = [
        [0, 10, 15, 20],
        [10, 0, 35, 25],
        [15, 35, 0, 30],
        [20, 25, 30, 0]
    ]
    print(DPFusion.tsp(graph)) # 预期输出: 80

# ===== 优化体系: Knuth 优化 =====
def knuth_optimization(n, cost_func):
    ...

```

Knuth 优化的 DP 算法

问题描述:

解决区间 DP 问题，其中状态转移方程满足四边形不等式

解题思路：

1. 使用 Knuth 优化将时间复杂度从 $O(n^3)$ 降低到 $O(n^2)$
2. 维护最优转移点数组 $\text{opt}[i][j]$ ，表示计算 $\text{dp}[i][j]$ 时的最优 k 值
3. 根据 $\text{opt}[i][j-1] \leq \text{opt}[i][j] \leq \text{opt}[i+1][j]$ 的性质进行剪枝

参数：

n : 区间长度

cost_func : 计算区间 (i, j) 代价的函数

返回：

dp : 结果 DP 数组

opt : 最优转移点数组

时间复杂度: $O(n^2)$

空间复杂度: $O(n^2)$

, , ,

初始化 dp 和 opt 数组

$\text{INF} = \text{float('inf')}$

$\text{dp} = [[\text{INF}] * (n + 1) \text{ for } _ \text{ in range}(n + 1)]$

$\text{opt} = [[0] * (n + 1) \text{ for } _ \text{ in range}(n + 1)]$

初始化长度为 1 的区间

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

$\text{dp}[i][i] = 0$

$\text{opt}[i][i] = i$

枚举区间长度

for $length$ in range(2, $n + 1$):

枚举起始点

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

$j = i + length - 1$

初始化为无穷大

$\text{dp}[i][j] = \text{INF}$

根据 Knuth 优化的性质，最优 k 在 $\text{opt}[i][j-1]$ 到 $\text{opt}[i+1][j]$ 之间

if $i + 1 \leq j$:

$\text{upper_k} = \text{opt}[i + 1][j]$

else:

$\text{upper_k} = j - 1$

for k in range($\text{opt}[i][j-1]$, $\min(\text{upper_k}, j - 1) + 1$):

if $\text{dp}[i][k] != \text{INF}$ and $\text{dp}[k+1][j] != \text{INF}$:

```

cost = cost_func(i, j)
if cost != INF:
    current = dp[i][k] + dp[k+1][j] + cost
    if current < dp[i][j]:
        dp[i][j] = current
        opt[i][j] = k

return dp, opt

# ===== 优化体系: Divide & Conquer Optimization =====
def solve_divide_conquer(i, l, r, opt_l, opt_r, dp, cost_func):
    """
    计算 dp[i][l..r], 其中最优转移点在 opt_l..opt_r 之间
    """

    if l > r:
        return

    mid = (l + r) // 2
    best_k = opt_l
    INF = float('inf')

    # 在 opt_l 到 min(mid, opt_r) 之间寻找最优 k
    for k in range(opt_l, min(mid, opt_r) + 1):
        if dp[i-1][k] != INF:
            cost = cost_func(k, mid)
            if cost != INF:
                current = dp[i-1][k] + cost
                if current < dp[i][mid]:
                    dp[i][mid] = current
                    best_k = k

    # 递归处理左右子区间
    solve_divide_conquer(i, l, mid - 1, opt_l, best_k, dp, cost_func)
    solve_divide_conquer(i, mid + 1, r, best_k, opt_r, dp, cost_func)

def divide_conquer_optimization(n, m, cost_func):
    """
    Divide & Conquer Optimization (分治优化)
    """


```

问题描述:

解决形如 $dp[i][j] = \min\{dp[i-1][k] + cost(k, j)\}$, 其中 $k < j$
当转移满足决策单调性时使用

解题思路：

- 利用决策单调性，使用分治法优化 DP
- 对于 $dp[i][j]$ ，当 i 固定时，最优转移点 k 随着 j 的增加而单调不减
- 使用分治的方式计算每个区间的最优决策

参数：

n: 维度 1
m: 维度 2
cost_func: 计算 $cost(k, j)$ 的函数

返回：

dp: DP 数组

时间复杂度: $O(n*m \log m)$

空间复杂度: $O(n*m)$

, , ,

```
# 初始化 dp 数组
INF = float('inf')
dp = [[INF] * (m + 1) for _ in range(n + 1)]
dp[0][0] = 0

# 对每个 i 应用分治优化
for i in range(1, n + 1):
    solve_divide_conquer(i, 1, m, 0, m, dp, cost_func)

return dp
```

===== 优化体系: SMAWK 算法 (行最小查询) =====

```
def reduce_rows(rows, matrix):
    ''' 行压缩: 只保留可能成为最小值的行 '''
    stack = []
    for i in rows:
        while len(stack) >= 2:
            j1 = stack.pop()
            j2 = stack[-1]
            stack.append(j1) # 恢复栈状态

            # 比较两个行在列 len(stack)-2 处的值 (因为索引从 0 开始)
            if matrix[j2][len(stack)-2] <= matrix[i][len(stack)-2]:
                break
        else:
            stack.pop()
            stack.append(i)
```

```

return stack

def smawk_rec(rows, cols, matrix):
    '''递归实现 SMAWK 算法'''
    if not rows:
        return []

    # 行压缩
    reduced_rows = reduce_rows(rows, matrix)

    # 递归求解列数为奇数的子问题
    half_cols = cols[1::2]
    min_cols = [-1] * len(reduced_rows)

    if half_cols:
        # 递归求解
        result = smawk_rec(reduced_rows, half_cols, matrix)
        # 复制结果
        for i in range(len(result)):
            min_cols[i] = result[i]

    # 扩展结果到所有列
    result = [0] * len(rows)
    k = 0  # min_cols 的索引

    for i in range(len(rows)):
        row = rows[i]
        # 确定当前行的最小值可能在哪个区间
        start = 0 if i == 0 else (min_cols[k-1] if k > 0 else 0)
        end = min_cols[k] if k < len(min_cols) else cols[-1]

        # 在这个区间内查找最小值
        min_val = float('inf')
        min_col = start

        # 找到 start 和 end 在 cols 中的索引
        start_idx = cols.index(start) if start in cols else -1
        end_idx = cols.index(end) if end in cols else -1

        if start_idx != -1 and end_idx != -1:
            for idx in range(start_idx, end_idx + 1):
                col = cols[idx]
                if col < len(matrix[0]) and matrix[row][col] < min_val:

```

```

        min_val = matrix[row][col]
        min_col = col

    result[i] = min_col

    # 如果当前行在 reduced_rows 中，且不是最后一行，k 前进
    if k < len(reduced_rows) and row == reduced_rows[k]:
        k += 1

return result

def smawk(matrix):
    """
    SMAWK 算法用于在 Monge 矩阵中快速查找每行的最小值
    """

```

问题描述:

给定一个 Monge 矩阵，快速找到每行的最小值位置

解题思路:

1. Monge 矩阵满足性质: $\text{matrix}[i][j] + \text{matrix}[i+1][j+1] \leq \text{matrix}[i][j+1] + \text{matrix}[i+1][j]$
2. SMAWK 算法利用这一性质，可以在 $O(m+n)$ 时间内找到每行的最小值
3. 主要步骤包括行压缩和递归求解

参数:

matrix: 一个 Monge 矩阵

返回:

每行最小值的列索引列表

时间复杂度: $O(m+n)$ ，其中 m 是行数， n 是列数

空间复杂度: $O(m+n)$

，

$m = \text{len}(\text{matrix})$

if $m == 0$:

return []

$n = \text{len}(\text{matrix}[0])$

构造行索引和列索引数组

rows = list(range(m))

cols = list(range(n))

调用递归实现

return smawk_rec(rows, cols, matrix)

```
# ===== 优化体系: Aliens Trick (二分约束参数+可行性 DP) =====
def aliens_trick(cost_func, check_func, left, right, eps=1e-7):
    ...
    Aliens Trick (二分约束参数+可行性 DP)
```

问题描述:

解决带约束的优化问题，通常形如最小化总成本，同时满足某些约束条件

解题思路:

1. 将约束条件转化为参数 λ ，构造拉格朗日函数
2. 对 λ 进行二分查找，使用可行性 DP 判断当前 λ 下是否满足约束
3. 根据可行性 DP 的结果调整二分区间

参数:

cost_func: 计算带参数 λ 的成本函数，返回 [value, constraint] 数组
check_func: 检查当前解是否满足约束的函数
left: 二分左边界
right: 二分右边界
eps: 精度要求

返回:

包含最优参数 lambda 和对应最优解的元组 (lambda, value)

时间复杂度: $O(\log((right-left)/\epsilon) * T(DP))$ ，其中 $T(DP)$ 是一次 DP 的时间复杂度

...

```
best_lambda = left
best_value = 0.0
```

```
while right - left > \epsilon:
    mid = (left + right) / 2
    # 计算当前参数下的解和约束值
    current_value, constraint_value = cost_func(mid)

    if check_func(constraint_value):
        # 满足约束，尝试更小的参数
        right = mid
        best_lambda = mid
        best_value = current_value
    else:
        # 不满足约束，需要增大参数
        left = mid
```

```

    return (best_lambda, best_value)

# ===== 图上 DP→最短路: 分层图建模 =====
def layered_graph_dijkstra(n, m, edges, k):
    ...
    分层图 Dijkstra 算法

```

问题描述:

给定一个图, 允许最多使用 k 次特殊操作 (如跳跃、免费通行等), 求最短路径

解题思路:

1. 构建分层图, 每层代表使用不同次数的特殊操作
2. 对于每个节点 u , 在第 i 层表示到达 u 时已经使用了 i 次特殊操作
3. 使用 Dijkstra 算法在分层图上寻找最短路径

参数:

n : 节点数量

m : 边的数量

$edges$: 边的列表, 每个元素为 $[u, v, w]$ 表示 u 到 v 的权为 w 的边

k : 允许使用的特殊操作次数

返回:

从节点 0 到节点 $n-1$ 的最短路径长度

时间复杂度: $O((n*k + m*k) \log(n*k))$

空间复杂度: $O(n*k + m*k)$

...

构建分层图的邻接表

```
graph = [[] for _ in range(n * (k + 1))]
```

添加普通边 (不使用特殊操作)

```
for u, v, w in edges:
```

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

```
        from_node = u + i * n
```

```
        graph[from_node].append((v + i * n, w))
```

添加使用特殊操作的边 (如果允许的话)

```
for u, v, w in edges:
```

```
    for i in range(k):
```

```
        # 这里假设特殊操作可以免费通行 (权为 0), 具体根据问题调整
```

```
        from_node = u + i * n
```

```
        graph[from_node].append((v + (i + 1) * n, 0))
```

```

# Dijkstra 算法
INF = float('inf')
dist = [INF] * (n * (k + 1))
dist[0] = 0 # 假设起点是节点 0

# 使用优先队列，按距离排序
heap = []
heappq.heappush(heap, (0, 0))

while heap:
    d, u = heappq.heappop(heap)

    if d > dist[u]:
        continue

    for v, w in graph[u]:
        if dist[v] > d + w:
            dist[v] = d + w
            heappq.heappush(heap, (dist[v], v))

# 取所有层中到达终点的最小值
result = INF
for i in range(k + 1):
    result = min(result, dist[n - 1 + i * n])

return result if result != INF else -1

# ====== 冷门模型：期望 DP 遇环的方程组解（高斯消元） ======
def gaussian_elimination(matrix):
    ...
    高斯消元法求解线性方程组

```

问题描述：

求解形如 $Ax = b$ 的线性方程组

解题思路：

1. 构建增广矩阵
2. 进行高斯消元，将矩阵转化为行阶梯形
3. 回代求解

参数：

`matrix`: 增广矩阵，每行最后一个元素是 b 的值

返回:

方程组的解数组

时间复杂度: $O(n^3)$

空间复杂度: $O(n^2)$

, , ,

```
n = len(matrix)
```

```
eps = 1e-9
```

高斯消元过程

```
for i in range(n):
```

找到主元行 (当前列中绝对值最大的行)

```
max_row = i
```

```
for j in range(i, n):
```

```
    if abs(matrix[j][i]) > abs(matrix[max_row][i]):
```

```
        max_row = j
```

交换主元行和当前行

```
matrix[i], matrix[max_row] = matrix[max_row], matrix[i]
```

如果主元为 0, 方程组可能有无穷多解或无解

```
if abs(matrix[i][i]) < eps:
```

```
    continue
```

消元过程

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

```
    factor = matrix[j][i] / matrix[i][i]
```

```
    for k in range(i, n + 1):
```

```
        matrix[j][k] -= factor * matrix[i][k]
```

回代求解

```
x = [0.0] * n
```

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

```
    x[i] = matrix[i][n]
```

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

```
        x[i] -= matrix[i][j] * x[j]
```

```
x[i] /= matrix[i][i]
```

```
return x
```

```
def expectation_dp_with_cycles(n, transitions):
```

, , ,

期望 DP 处理有环情况 (使用高斯消元)

问题描述:

在有环的状态转移图中计算期望

解题思路:

1. 对于每个状态，建立期望方程
2. 使用高斯消元求解方程组

参数:

n: 状态数量

transitions: 转移概率列表，`transitions[i]`是一个列表，每个元素为(j, p)表示从 i 转移到 j 的概率为 p

返回:

每个状态的期望值数组

时间复杂度: $O(n^3)$

空间复杂度: $O(n^2)$

, , ,

构建线性方程组的增广矩阵

```
matrix = [[0.0] * (n + 1) for _ in range(n)]
```

```
for i in range(n):
```

```
    matrix[i][i] = 1.0 # 方程左边: E[i] - sum(p_ij * E[j]) = cost[i]
```

假设每个状态的代价为 1，具体根据问题调整

```
cost = 1.0
```

```
matrix[i][n] = cost
```

```
for j, p in transitions[i]:
```

```
    if i != j: # 避免自环的特殊处理
```

```
        matrix[i][j] -= p
```

使用高斯消元求解

```
return gaussian_elimination(matrix)
```

===== 冷门模型: 插头 DP (轮廓线 DP) =====

```
def plug_dp(grid):
```

, , ,

插头 DP (轮廓线 DP) 示例: 求网格中哈密顿回路的数量

问题描述:

给定一个网格，求其中哈密顿回路的数量

解题思路：

1. 使用轮廓线 DP，记录当前处理到的位置和轮廓线状态
2. 插头表示连接的状态，通常用二进制表示
3. 使用字典优化空间复杂度

参数：

grid：网格，1 表示可通行，0 表示障碍物

返回：

哈密顿回路的数量

时间复杂度： $O(n \cdot m \cdot 4^{\min(n, m)})$

空间复杂度： $O(4^{\min(n, m)})$

，，，

```
n = len(grid)
if n == 0:
    return 0
m = len(grid[0])

# 使用字典优化
dp = defaultdict(int)

# 初始状态：左上角没有插头
dp[0] = 1

for i in range(n):
    # 新的一行开始，需要将状态左移一位
    new_dp = defaultdict(int)
    for state, cnt in dp.items():
        # 左移一位，移除最左边的插头
        new_state = state << 1
        new_dp[new_state] += cnt
    dp = new_dp
```

```
for j in range(m):
    new_dp2 = defaultdict(int)

    for state, cnt in dp.items():
        # 当前位置左边和上边的插头状态
        left = (state >> (2 * j)) & 3
        up = (state >> (2 * (j + 1))) & 3
```

```

# 如果当前位置是障碍物，跳过
if grid[i][j] == 0:
    # 只有当左右插头都不存在时才合法
    if left == 0 and up == 0:
        new_dp2[state] += cnt
    continue

# 处理各种插头组合情况
# 1. 没有左插头和上插头
if left == 0 and up == 0:
    # 只能创建新的插头对（用于回路的开始）
    if i < n - 1 and j < m - 1 and grid[i+1][j] == 1 and grid[i][j+1] == 1:
        new_state = state | (1 << (2 * j)) | (2 << (2 * (j + 1)))
        new_dp2[new_state] += cnt

# 2. 只有左插头
elif left != 0 and up == 0:
    # 向下延伸
    if i < n - 1 and grid[i+1][j] == 1:
        new_dp2[state] += cnt
    # 向右延伸
    if j < m - 1 and grid[i][j+1] == 1:
        new_state = (state & ~(3 << (2 * j))) | (left << (2 * (j + 1)))
        new_dp2[new_state] += cnt

# 3. 只有上插头
elif left == 0 and up != 0:
    # 向右延伸
    if j < m - 1 and grid[i][j+1] == 1:
        new_dp2[state] += cnt
    # 向下延伸
    if i < n - 1 and grid[i+1][j] == 1:
        new_state = (state & ~(3 << (2 * (j + 1)))) | (up << (2 * j))
        new_dp2[new_state] += cnt

# 4. 同时有左插头和上插头
else:
    # 合并插头
    new_state = (state & ~(3 << (2 * j))) & ~(3 << (2 * (j + 1)))

    # 如果是形成回路的最后一步
    if left == up:
        # 检查是否所有插头都已连接

```

```

        if new_state == 0 and i == n - 1 and j == m - 1:
            new_dp2[new_state] += cnt
        else:
            # 合并两个不同的插头
            new_dp2[new_state] += cnt

    dp = new_dp2

# 最终状态应该是没有任何插头（形成回路）
return dp.get(0, 0)

# ====== 冷门模型：树上背包的优化 ======
def dfs_tree_knapsack(u, parent, capacity, tree, weights, values, dp, size):
    # 初始化当前节点
    size[u] = 1
    if weights[u] <= capacity:
        dp[u][weights[u]] = values[u]

    # 对每个子节点，按照子树大小排序，小的先合并
    children = []
    for v in tree[u]:
        if v != parent:
            dfs_tree_knapsack(v, u, capacity, tree, weights, values, dp, size)
            children.append((size[v], v))

    # 按子树大小排序
    children.sort()

    for sz, v in children:
        # 逆序遍历容量，避免重复计算
        for i in range(min(size[u], capacity), -1, -1):
            if dp[u][i] == 0 and i != 0:
                continue
            for j in range(1, min(sz, capacity - i) + 1):
                if dp[v][j] > 0 and i + j <= capacity:
                    dp[u][i + j] = max(dp[u][i + j], dp[u][i] + dp[v][j])

    # 更新子树大小
    size[u] += sz

def tree_knapsack_optimized(root, capacity, tree, weights, values):
    ...
    树上背包的优化实现（小到大合并）

```

问题描述:

在树上选择一些节点，使得总重量不超过容量，且总价值最大

解题思路:

1. 使用后序遍历处理子树
2. 使用小到大合并的策略优化复杂度
3. 对于每个节点，维护一个容量为 capacity 的背包

参数:

root: 根节点
capacity: 背包容量
tree: 树的邻接表
weights: 每个节点的重量
values: 每个节点的价值

返回:

最大价值

时间复杂度: $O(n * capacity^2)$ ，但通过小到大合并可以降低常数

空间复杂度: $O(n * capacity)$

, , ,

```
n = len(tree)
# 初始化 dp 数组
dp = [[0] * (capacity + 1) for _ in range(n)]
size = [0] * n

# 深度优先搜索处理子树
dfs_tree_knapsack(root, -1, capacity, tree, weights, values, dp, size)

# 返回根节点的最大价值
return max(dp[root])
```

===== 补充题目与应用 =====

以下是一些使用上述高级 DP 技术的经典题目及其代码实现

1. 编辑距离问题 (LeetCode 72)

```
def edit_distance(word1, word2):
, , ,
```

LeetCode 72. 编辑距离

题目链接: <https://leetcode-cn.com/problems/edit-distance/>

问题描述:

给你两个单词 word1 和 word2，计算出将 word1 转换成 word2 所使用的最少操作数。

你可以对一个单词进行如下三种操作：插入一个字符、删除一个字符、替换一个字符。

解题思路：

使用二维 DP， $dp[i][j]$ 表示 word1 的前 i 个字符转换为 word2 的前 j 个字符所需的最少操作数。

时间复杂度： $O(m \times n)$

空间复杂度： $O(m \times n)$

，，，

```
m = len(word1)
```

```
n = len(word2)
```

```
# dp[i][j] 表示 word1 的前 i 个字符转换为 word2 的前 j 个字符所需的最少操作数
```

```
dp = [[0] * (n + 1) for _ in range(m + 1)]
```

```
# 初始化边界
```

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

```
    dp[i][0] = i
```

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

```
    dp[0][j] = j
```

```
# 动态规划填表
```

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

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

```
        if word1[i - 1] == word2[j - 1]:
```

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

```
        else:
```

```
            dp[i][j] = min(dp[i - 1][j], dp[i][j - 1], dp[i - 1][j - 1]) + 1
```

```
return dp[m][n]
```

```
# 2. 最长递增子序列 (LeetCode 300)
```

```
def length_of_lis(nums):
```

，，，

LeetCode 300. 最长递增子序列

题目链接：<https://leetcode-cn.com/problems/longest-increasing-subsequence/>

问题描述：

给你一个整数数组 $nums$ ，找到其中最长严格递增子序列的长度。

解题思路：

使用贪心 + 二分查找优化的 DP 方法。

$tails[i]$ 表示长度为 $i+1$ 的递增子序列的末尾元素的最小值。

```

时间复杂度: O(n log n)
空间复杂度: O(n)
'''

if not nums:
    return 0

tails = []
for num in nums:
    # 二分查找 num 应该插入的位置
    left, right = 0, len(tails)
    while left < right:
        mid = left + (right - left) // 2
        if tails[mid] >= num:
            right = mid
        else:
            left = mid + 1
    if left == len(tails):
        tails.append(num)
    else:
        tails[left] = num

return len(tails)

```

3. 背包问题变种 – 完全背包 (LeetCode 322)

```

def coin_change(coins, amount):
    '''

```

LeetCode 322. 零钱兑换

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

问题描述:

给你一个整数数组 coins，表示不同面额的硬币；以及一个整数 amount，表示总金额。

计算并返回可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额，返回-1。

解题思路:

使用完全背包的思想，dp[i]表示凑成金额 i 所需的最少硬币数。

时间复杂度: O(amount * n)

空间复杂度: O(amount)

'''

初始化 dp 数组为无穷大

INF = float('inf')

dp = [INF] * (amount + 1)

dp[0] = 0 # 凑成金额 0 需要 0 个硬币

```

for coin in coins:
    for i in range(coin, amount + 1):
        if dp[i - coin] != INF:
            dp[i] = min(dp[i], dp[i - coin] + 1)

return dp[amount] if dp[amount] != INF else -1

```

4. 矩阵链乘法（区间 DP 的经典应用）

```

def matrix_chain_order(p):
    ...

```

矩阵链乘法问题

题目来源：算法导论

问题描述：

给定一系列矩阵，计算乘法顺序使得标量乘法的次数最少。

解题思路：

使用区间 DP， $dp[i][j]$ 表示计算第 i 到第 j 个矩阵的乘积所需的最少标量乘法次数。

可以使用 Knuth 优化进一步降低时间复杂度。

时间复杂度： $O(n^3)$

空间复杂度： $O(n^2)$

...

```
n = len(p) - 1 # 矩阵的个数
```

```
# dp[i][j] 表示计算第 i 到第 j 个矩阵的乘积所需的最少标量乘法次数
```

```
INF = float('inf')
```

```
dp = [[INF] * (n + 1) for _ in range(n + 1)]
```

```
# s[i][j] 记录最优分割点
```

```
s = [[0] * (n + 1) for _ in range(n + 1)]
```

单个矩阵的代价为 0

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

```
    dp[i][i] = 0
```

枚举区间长度

```
for length in range(2, n + 1):
```

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

```
        j = i + length - 1
```

```
        dp[i][j] = INF
```

枚举分割点

```
        for k in range(i, j):
```

计算当前分割点的代价

```

cost = dp[i][k] + dp[k + 1][j] + p[i - 1] * p[k] * p[j]
if cost < dp[i][j]:
    dp[i][j] = cost
    s[i][j] = k

return dp, s

```

5. 旅行商问题 (TSP) 的 DP 实现

```
def traveling_salesman_problem(graph):
    ,,
```

旅行商问题

题目来源：算法竞赛经典问题

问题描述：

给定一个完全图，找到一条访问每个城市恰好一次并返回起点的最短路径。

解题思路：

使用状态压缩 DP， $dp[mask][u]$ 表示访问过的城市集合为 $mask$ ，当前在城市 u 时的最短路径长度。

时间复杂度： $O(n^2 * 2^n)$

空间复杂度： $O(n * 2^n)$

,,

```
n = len(graph)
```

$dp[mask][u]$ 表示访问过的城市集合为 $mask$ ，当前在城市 u 时的最短路径长度

```
INF = float('inf')
```

```
dp = [[INF] * n for _ in range(1 << n)]
```

初始状态：只访问了起点，路径长度为 0

```
for i in range(n):
```

```
    dp[1 << i][i] = 0
```

枚举所有可能的状态

```
for mask in range(1, 1 << n):
```

枚举当前所在的城市

```
for u in range(n):
```

```
    if not (mask & (1 << u)):
```

```
        continue
```

枚举下一个要访问的城市

```
for v in range(n):
```

```
    if mask & (1 << v):
```

```
        continue
```

```
        new_mask = mask | (1 << v)
```

```
        if dp[mask][u] != INF and graph[u][v] != INF:
```

```

        if dp[new_mask][v] > dp[mask][u] + graph[u][v]:
            dp[new_mask][v] = dp[mask][u] + graph[u][v]

# 找到最短的回路
result = INF
for u in range(n):
    if dp[(1 << n) - 1][u] != INF and graph[u][0] != INF:
        result = min(result, dp[(1 << n) - 1][u] + graph[u][0])

return result if result != INF else -1

```

6. 区间 DP: 最优三角剖分

```

def minimum_score_triangulation(values):
    ...

```

LeetCode 1039. 多边形三角剖分的最低得分

题目链接: <https://leetcode-cn.com/problems/minimum-score-triangulation-of-polygon/>

问题描述:

给定一个凸多边形，将其三角剖分，使得所有三角形的顶点乘积之和最小。

解题思路:

使用区间 DP， $dp[i][j]$ 表示从顶点 i 到顶点 j 的多边形三角剖分的最小得分。

时间复杂度: $O(n^3)$

空间复杂度: $O(n^2)$

...

$n = \text{len}(\text{values})$

$dp[i][j]$ 表示从顶点 i 到顶点 j 的多边形三角剖分的最小得分

$\text{INF} = \text{float('inf')}$

$dp = [[0] * n \text{ for } _ \text{ in } \text{range}(n)]$

枚举区间长度

for length in range(3, n + 1):

for i in range(n - length + 1):

j = i + length - 1

dp[i][j] = INF

枚举中间点

for k in range(i + 1, j):

dp[i][j] = min(dp[i][j], dp[i][k] + dp[k][j] + values[i] * values[k] * values[j])

return dp[0][n - 1]

7. 博弈 DP: 石子游戏

```
def stone_game(piles):
    """
    LeetCode 877. 石子游戏
    题目链接: https://leetcode-cn.com/problems/stone-game/

```

问题描述:

给定一个表示石子堆的数组，两个玩家轮流从两端取石子，每次只能取一个，取到最后一个石子的人获胜。

判断先手是否必胜。

解题思路:

使用区间 DP， $dp[i][j]$ 表示在区间 $[i, j]$ 中，先手能获得的最大净胜分。

时间复杂度: $O(n^2)$

空间复杂度: $O(n^2)$

...

```
n = len(piles)
```

```
# dp[i][j] 表示在区间 [i, j] 中，先手能获得的最大净胜分
```

```
dp = [[0] * n for _ in range(n)]
```

```
# 初始化单个石子堆
```

```
for i in range(n):
```

```
    dp[i][i] = piles[i]
```

```
# 枚举区间长度
```

```
for length in range(2, n + 1):
```

```
    for i in range(n - length + 1):
```

```
        j = i + length - 1
```

```
# 先手可以选择取左边或右边
```

```
        dp[i][j] = max(piles[i] - dp[i + 1][j], piles[j] - dp[i][j - 1])
```

```
# 先手净胜分大于 0 则必胜
```

```
return dp[0][n - 1] > 0
```

```
# 8. 数位 DP: 统计 1 出现的次数
```

```
def count_digit_one(n):
```

...

```
LeetCode 233. 数字 1 的个数
```

```
题目链接: https://leetcode-cn.com/problems/number-of-digit-one/
```

问题描述:

给定一个整数 n ，计算所有小于等于 n 的非负整数中数字 1 出现的个数。

解题思路:

使用数位 DP，逐位处理每一位上 1 出现的次数。

时间复杂度: $O(\log n)$

空间复杂度: $O(\log n)$

, , ,

```
if n <= 0:  
    return 0  
  
s = str(n)  
length = len(s)  
count = 0  
  
# 逐位处理  
for i in range(length):  
    high = int(s[:i]) if i > 0 else 0  
    current = int(s[i])  
    low = int(s[i+1:]) if i < length - 1 else 0  
    digit = 10 ** (length - i - 1)  
  
    if current == 0:  
        # 当前位为 0, 高位决定  
        count += high * digit  
    elif current == 1:  
        # 当前位为 1, 高位+低位+1  
        count += high * digit + low + 1  
    else:  
        # 当前位大于 1, 高位+1  
        count += (high + 1) * digit  
  
return count
```

9. 树形 DP: 打家劫舍 III

```
class TreeNode:  
    def __init__(self, x):  
        self.val = x  
        self.left = None  
        self.right = None  
  
def rob_dfs(node):  
    if not node:  
        return [0, 0]
```

```

left = rob_dfs(node.left)
right = rob_dfs(node.right)

# rob_current 表示偷当前节点, not_rob_current 表示不偷当前节点
rob_current = node.val + left[1] + right[1]
not_rob_current = max(left[0], left[1]) + max(right[0], right[1])

return [rob_current, not_rob_current]

```

```

def rob(root):
    ...

```

LeetCode 337. 打家劫舍 III

题目链接: <https://leetcode-cn.com/problems/house-robber-iii/>

问题描述:

在上次打劫完一条街道之后和一圈房屋后，小偷又发现了一个新的可行窃的地区。

这个地区只有一个入口，我们称之为“根”。除了“根”之外，每栋房子有且只有一个“父”房子与之相连。

一番侦察之后，聪明的小偷意识到“这个地方的所有房屋的排列类似于一棵二叉树”。

如果两个直接相连的房子在同一天晚上被打劫，房屋将自动报警。

计算在不触动警报的情况下，小偷一晚能够盗取的最高金额。

解题思路:

使用树形 DP，对于每个节点，维护两个状态：偷或不偷。

时间复杂度: $O(n)$

空间复杂度: $O(h)$ ， h 为树的高度

...

```

result = rob_dfs(root)
return max(result[0], result[1])

```

10. 状态压缩 DP: 蒙斯特曼问题

```

def monster_game(grid):
    ...

```

蒙斯特曼问题

题目来源: 算法竞赛问题

问题描述:

在网格中放置怪物，使得任何两个怪物都不在同一行、同一列或对角线上。

解题思路:

使用状态压缩 DP， $dp[i][mask]$ 表示处理到第 i 行，已放置的列的状态为 $mask$ 时的方案数。

```

时间复杂度: O(n * 2^n)
空间复杂度: O(2^n)
,,
n = len(grid)

# dp[mask]表示处理到当前行, 已放置的列的状态为 mask 时的方案数
dp = [0] * (1 << n)
dp[0] = 1

for i in range(n):
    new_dp = [0] * (1 << n)
    for mask in range(1 << n):
        if dp[mask] == 0:
            continue
        # 枚举所有可能的放置位置
        for j in range(n):
            # 检查是否可以在(i, j)放置怪物
            if not (mask & (1 << j)) and grid[i][j] == 1:
                # 检查对角线
                valid = True
                for k in range(i):
                    if (mask & (1 << k)) and abs(k - j) == i - k:
                        valid = False
                        break
                if valid:
                    new_dp[mask | (1 << j)] += dp[mask]
    dp = new_dp

return dp[(1 << n) - 1]

```

11. 高维 DP: 三维背包

```

def three_dimension_knapsack(n, capacity, items):
,,

```

三维背包问题

题目来源: 算法竞赛问题

问题描述:

有 n 个物品, 每个物品有体积、重量、价值三个属性, 背包有体积和重量两个限制, 求最大价值。

解题思路:

使用三维 DP, $dp[i][j][k]$ 表示前 i 个物品, 体积为 j , 重量为 k 时的最大价值。

时间复杂度: $O(n * V * W)$

空间复杂度: $O(n * V * W)$

```

    ,
    V, W = capacity
    # 初始化 dp 数组
    dp = [[[0] * (W + 1) for _ in range(V + 1)] for __ in range(n + 1)]

    for i in range(1, n + 1):
        v, w, val = items[i-1]
        for j in range(V + 1):
            for k in range(W + 1):
                # 不选当前物品
                dp[i][j][k] = dp[i-1][j][k]
                # 选当前物品 (如果有足够的空间)
                if j >= v and k >= w:
                    dp[i][j][k] = max(dp[i][j][k], dp[i-1][j-v][k-w] + val)

    return dp[n][V][W]

```

12. 斜率优化 DP 示例

```
class ConvexHullTrick:
```

```
    ,,
```

凸包优化技巧示例

题目来源：算法竞赛问题

问题描述：

当状态转移方程形如 $dp[i] = \min\{dp[j] + a[i] * b[j] + c\}$ 时，可以使用凸包优化。

解题思路：

将转移方程转换为直线的形式，维护凸包以快速查询最小值。

时间复杂度：O(n)

空间复杂度：O(n)

```
    ,,
```

```
class Line:
```

```
    def __init__(self, k, b):
        self.k = k
        self.b = b
```

```
    def __init__(self):
        self.dq = deque()
```

```
    def add_line(self, k, b):
        # 当队列中至少有两条直线时，检查是否需要删除末尾的直线
```

```

while len(self.dq) >= 2:
    l1 = self._get_nth_last(2)
    l2 = self.dq[-1]
    # 判断直线 l1 和 l2 的交点是否在 l2 和新直线的交点右侧
    if (l2.b - l1.b) * (k - l2.k) >= (b - l2.b) * (l2.k - l1.k):
        self.dq.pop()
    else:
        break
    self.dq.append(self.Line(k, b))

def _get_nth_last(self, n):
    if n <= 0 or n > len(self.dq):
        raise IndexError("索引越界")
    # 转换为列表获取倒数第 n 个元素
    temp = list(self.dq)
    return temp[-n]

def query_correct(self, x):
    # 正确的查询实现
    while len(self.dq) >= 2:
        l1 = self.dq.popleft()
        l2 = self.dq[0]
        if l1.k * x + l1.b >= l2.k * x + l2.b:
            # 继续弹出
            continue
        else:
            self.dq.appendleft(l1) # 恢复 l1
            break

    if not self.dq:
        return float('inf')
    l = self.dq[0]
    return l.k * x + l.b

# ===== 测试和调试 =====
if __name__ == "__main__":
    # 测试编辑距离
    print("编辑距离测试:", edit_distance("horse", "ros")) # 应输出 3

    # 测试最长递增子序列
    print("最长递增子序列测试:", length_of_lis([10, 9, 2, 5, 3, 7, 101, 18])) # 应输出 4

    # 测试零钱兑换

```

```

print("零钱兑换测试:", coin_change([1, 2, 5], 11)) # 应输出 3

# 测试矩阵链乘法
p = [30, 35, 15, 5, 10, 20, 25]
dp, s = matrix_chain_order(p)
print("矩阵链乘法最优代价:", dp[1][6]) # 应输出 15125

# 测试石子游戏
print("石子游戏测试:", stone_game([5, 3, 4, 5])) # 应输出 True

# 测试数字 1 的个数
print("数字 1 的个数测试:", count_digit_one(13)) # 应输出 6

```

=====

文件: dp_fusion.cpp

=====

```

// -*- coding: utf-8 -*-
/***
 * DP 融合场景: DP+数论、DP+字符串、DP+计算几何
 *
 * 问题描述:
 * 动态规划(DP)可以与其他领域的算法和数据结构结合, 形成强大的问题解决方法。
 * 本文件实现了三种主要的融合场景:
 * 1. DP+数论 (模意义下的动态规划)
 * 2. DP+字符串 (基于后缀自动机的计数)
 * 3. DP+计算几何 (凸包上的动态规划)
 *
 * 时间复杂度:
 * - DP+数论: 根据具体问题而定, 通常为  $O(n^2)$  或  $O(n^3)$ 
 * - DP+字符串:  $O(n)$  或  $O(n^2)$ 
 * - DP+计算几何:  $O(n^2)$  或  $O(n \log n)$ 
 *
 * 空间复杂度:  $O(n^2)$ 
 *
 * 相关题目:
 * 1. LeetCode 518. 零钱兑换 II (模意义)
 * 2. LeetCode 682. 棒球比赛 (字符串 DP)
 * 3. LeetCode 873. 最长的斐波那契子序列的长度 (序列 DP)
 */

```

```

#include <iostream>
#include <vector>

```

```

#include <string>
#include <algorithm>
#include <map>
#include <cmath>
#include <climits>
#include <float.h>

using namespace std;

// ===== DP+数论（模意义） =====

/***
 * LeetCode 518. 零钱兑换 II（模意义下的变种）
 * 题目链接: https://leetcode-cn.com/problems/coin-change-2/
 *
 * 问题描述:
 * 给定不同面额的硬币和一个总金额。计算可以凑成总金额的硬币组合数。
 * 假设每一种面额的硬币有无限个。要求结果对给定的模数取余。
 *
 * 解题思路:
 * 使用动态规划，定义 dp[i] 表示凑成金额 i 的组合数。
 * 状态转移方程: dp[i] = (dp[i] + dp[i-coin]) % mod, 其中 coin 是硬币面额。
 *
 * @param amount 总金额
 * @param coins 硬币面额数组
 * @param mod 模数
 * @return 可以凑成总金额的硬币组合数对 mod 取余的结果
 */
int coinChangeMod(int amount, vector<int>& coins, int mod) {
    vector<long long> dp(amount + 1, 0);
    dp[0] = 1; // 凑成金额 0 的方式只有 1 种（不选任何硬币）

    // 遍历每种硬币
    for (int coin : coins) {
        // 遍历金额，从 coin 开始
        for (int i = coin; i <= amount; ++i) {
            dp[i] = (dp[i] + dp[i - coin]) % mod;
        }
    }

    return dp[amount];
}

```

```

/***
 * 矩阵乘法（模意义下）
 *
 * @param a 矩阵 a
 * @param b 矩阵 b
 * @param mod 模数
 * @return 矩阵 a 和矩阵 b 的乘积对 mod 取余的结果
 */
vector<vector<long long>> matrixMultiply(vector<vector<long long>>& a, vector<vector<long long>>& b, int mod) {
    int n = a.size();
    int m = b[0].size();
    int k = b.size();
    vector<vector<long long>> result(n, vector<long long>(m, 0));

    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < m; ++j) {
            for (int p = 0; p < k; ++p) {
                result[i][j] = (result[i][j] + a[i][p] * b[p][j]) % mod;
            }
        }
    }

    return result;
}

/***
 * 矩阵快速幂（模意义下）
 *
 * 问题描述:
 * 计算矩阵的幂，结果对 mod 取余。
 *
 * 解题思路:
 * 使用快速幂算法，将矩阵的乘法在模意义下进行。
 *
 * @param matrix 输入矩阵
 * @param power 幂次
 * @param mod 模数
 * @return 矩阵的幂对 mod 取余的结果
 */
vector<vector<long long>> matrixPowerMod(vector<vector<long long>>& matrix, int power, int mod) {
    int n = matrix.size();
    // 初始化结果为单位矩阵

```

```

vector<vector<long long>> result(n, vector<long long>(n, 0));
for (int i = 0; i < n; ++i) {
    result[i][i] = 1;
}

// 快速幂算法
while (power > 0) {
    if (power % 2 == 1) {
        // 矩阵乘法
        result = matrixMultiply(result, matrix, mod);
    }
    // 矩阵自乘
    matrix = matrixMultiply(matrix, matrix, mod);
    power /= 2;
}

return result;
}

// ===== DP+字符串 (SAM 相关) =====

/***
 * LeetCode 516. 最长回文子序列
 * 题目链接: https://leetcode-cn.com/problems/longest-palindromic-subsequence/
 *
 * 问题描述:
 * 给定一个字符串 s，找到其中最长的回文子序列。可以假设 s 的最大长度为 1000。
 *
 * 解题思路:
 * 使用区间 DP，定义 dp[i][j] 表示字符串 s 在区间 [i, j] 内的最长回文子序列的长度。
 * 状态转移方程:
 * - 如果 s[i] == s[j]，则 dp[i][j] = dp[i+1][j-1] + 2
 * - 否则，dp[i][j] = max(dp[i+1][j], dp[i][j-1])
 *
 * @param s 输入字符串
 * @return 最长回文子序列的长度
 */
int longestPalindromicSubseq(string s) {
    int n = s.size();
    // dp[i][j] 表示字符串 s 在区间 [i, j] 内的最长回文子序列的长度
    vector<vector<int>> dp(n, vector<int>(n, 0));

    // 初始化单个字符的情况

```

```

for (int i = 0; i < n; ++i) {
    dp[i][i] = 1;
}

// 枚举区间长度
for (int length = 2; length <= n; ++length) {
    // 枚举起点
    for (int i = 0; i <= n - length; ++i) {
        int j = i + length - 1;
        if (s[i] == s[j]) {
            dp[i][j] = dp[i+1][j-1] + 2;
        } else {
            dp[i][j] = max(dp[i+1][j], dp[i][j-1]);
        }
    }
}

return dp[0][n-1];
}

```

```

/**
 * 后缀自动机 (Suffix Automaton)
 *
 * 后缀自动机是一个可以表示字符串的所有子串的数据结构。
 * 它可以用于解决许多字符串问题，如子串匹配、最长重复子串等。
 */
class SuffixAutomaton {

private:
    struct State {
        int len; // 该状态能接受的最长字符串的长度
        int link; // 后缀链接
        map<char, int> next; // 转移函数
        int endposSize; // endpos 集合的大小

        State() : len(0), link(-1), endposSize(0) {}
    };

    int size;
    int last;
    vector<State> states;

    void extend(char c) {
        int p = last;

```

```

int curr = size;
size++;
states.push_back(State());
states[curr].len = states[p].len + 1;

while (p != -1 && states[p].next.find(c) == states[p].next.end()) {
    states[p].next[c] = curr;
    p = states[p].link;
}

if (p == -1) {
    states[curr].link = 0;
} else {
    int q = states[p].next[c];
    if (states[p].len + 1 == states[q].len) {
        states[curr].link = q;
    } else {
        int clone = size;
        size++;
        states.push_back(State());
        states[clone].len = states[p].len + 1;
        states[clone].next = states[q].next;
        states[clone].link = states[q].link;

        while (p != -1 && states[p].next.find(c) != states[p].next.end() &&
states[p].next[c] == q) {
            states[p].next[c] = clone;
            p = states[p].link;
        }

        states[q].link = clone;
        states[curr].link = clone;
    }
}

last = curr;
}

void calcEndposSize() {
    // 按 len 排序
    vector<int> order(size);
    for (int i = 0; i < size; ++i) {
        order[i] = i;
    }
}

```

```

    }

    sort(order.begin(), order.end(), [this](int a, int b) {
        return states[a].len > states[b].len;
    });

    // 初始化为 1 (每个状态至少对应一个结束位置)
    for (int i = 1; i < size; ++i) {
        states[i].endposSize = 1;
    }

    // 从长到短更新
    for (int u : order) {
        if (states[u].link != -1) {
            states[states[u].link].endposSize += states[u].endposSize;
        }
    }
}

public:
    SuffixAutomaton(string s) {
        size = 1;
        last = 0;
        states.push_back(State());
        // 构建后缀自动机
        for (char c : s) {
            extend(c);
        }
        // 计算 endpos 集合的大小
        calcEndposSize();
    }

    /**
     * 计算不同子串的数量
     */
    int countSubstrings() {
        int count = 0;
        for (int i = 1; i < size; ++i) {
            count += states[i].len - states[states[i].link].len;
        }
        return count;
    }
}

```

```

// ===== 优化体系: Knuth 优化 =====

// Knuth 优化用于优化形如 dp[i][j] = min{dp[i][k] + dp[k+1][j]} + w(i, j) 的 DP
// 当满足四边形不等式时, 最优转移点单调

struct KnuthOptimizationResult {
    vector<vector<int>> dp;
    vector<vector<int>> opt;

    KnuthOptimizationResult(const vector<vector<int>>& dp, const vector<vector<int>>& opt)
        : dp(dp), opt(opt) {}

};

using CostFunction = function<int(int, int)>;

```

```

KnuthOptimizationResult knuthOptimization(int n, CostFunction costFunc) {
    /*
    Knuth 优化的 DP 算法

```

问题描述:

解决区间 DP 问题, 其中状态转移方程满足四边形不等式

解题思路:

1. 使用 Knuth 优化将时间复杂度从 $O(n^3)$ 降低到 $O(n^2)$
2. 维护最优转移点数组 $opt[i][j]$, 表示计算 $dp[i][j]$ 时的最优 k 值
3. 根据 $opt[i][j-1] \leq opt[i][j] \leq opt[i+1][j]$ 的性质进行剪枝

参数:

n : 区间长度

$costFunc$: 计算区间 (i, j) 代价的函数

返回:

$KnuthOptimizationResult$: 包含 dp 数组和 opt 数组的结果类

时间复杂度: $O(n^2)$

空间复杂度: $O(n^2)$

*/

// 初始化 dp 和 opt 数组

```
vector<vector<int>> dp(n + 1, vector<int>(n + 1));
```

```
vector<vector<int>> opt(n + 1, vector<int>(n + 1));
```

// 初始化长度为 1 的区间

```

for (int i = 1; i <= n; ++i) {
    dp[i][i] = 0;
    opt[i][i] = i;
}

// 枚举区间长度
for (int length = 2; length <= n; ++length) {
    // 枚举起始点
    for (int i = 1; i + length - 1 <= n; ++i) {
        int j = i + length - 1;
        // 初始化为无穷大
        dp[i][j] = INT_MAX;
        // 根据 Knuth 优化的性质，最优 k 在 opt[i][j-1] 到 opt[i+1][j] 之间
        int upperK = (i + 1 <= j) ? opt[i + 1][j] : j - 1;
        for (int k = opt[i][j - 1]; k <= min(upperK, j - 1); ++k) {
            if (dp[i][k] != INT_MAX && dp[k + 1][j] != INT_MAX) {
                int cost = costFunc(i, j);
                if (cost != INT_MAX) {
                    int current = dp[i][k] + dp[k + 1][j] + cost;
                    if (current < dp[i][j]) {
                        dp[i][j] = current;
                        opt[i][j] = k;
                    }
                }
            }
        }
    }
}

return KnuthOptimizationResult(dp, opt);
}

// ===== 优化体系: Divide & Conquer Optimization =====

void solveDivideConquer(int i, int l, int r, int opt_l, int opt_r,
                       vector<vector<int>>& dp, CostFunction costFunc) {
/*
计算 dp[i][l..r]，其中最优转移点在 opt_l..opt_r 之间
*/
if (l > r) {
    return;
}

```

```

int mid = (l + r) / 2;
int best_k = opt_l;

// 在 opt_l 到 min(mid, opt_r) 之间寻找最优 k
for (int k = opt_l; k <= min(mid, opt_r); ++k) {
    if (dp[i - 1][k] != INT_MAX) {
        int cost = costFunc(k, mid);
        if (cost != INT_MAX) {
            int current = dp[i - 1][k] + cost;
            if (current < dp[i][mid]) {
                dp[i][mid] = current;
                best_k = k;
            }
        }
    }
}

// 递归处理左右子区间
solveDivideConquer(i, l, mid - 1, opt_l, best_k, dp, costFunc);
solveDivideConquer(i, mid + 1, r, best_k, opt_r, dp, costFunc);
}

vector<vector<int>> divideConquerOptimization(int n, int m, CostFunction costFunc) {
/*
Divide & Conquer Optimization (分治优化)

```

问题描述:

解决形如 $dp[i][j] = \min\{dp[i-1][k] + cost(k, j)\}$, 其中 $k < j$
当转移满足决策单调性时使用

解题思路:

1. 利用决策单调性, 使用分治法优化 DP
2. 对于 $dp[i][j]$, 当 i 固定时, 最优转移点 k 随着 j 的增加而单调不减
3. 使用分治的方式计算每个区间的最优决策

参数:

n: 维度 1

m: 维度 2

costFunc: 计算 $cost(k, j)$ 的函数

返回:

vector<vector<int>>: dp 数组

```

时间复杂度: O(n*m log m)
空间复杂度: O(n*m)
*/
// 初始化 dp 数组
vector<vector<int>> dp(n + 1, vector<int>(m + 1, INT_MAX));
dp[0][0] = 0;

// 对每个 i 应用分治优化
for (int i = 1; i <= n; ++i) {
    solveDivideConquer(i, 1, m, 0, m, dp, costFunc);
}

return dp;
}

// ===== 优化体系: SMAWK 算法 (行最小查询) =====

vector<int> reduceRows(const vector<int>& rows, const vector<vector<int>>& matrix) {
    /*行压缩: 只保留可能成为最小值的行*/
    vector<int> stack;
    for (int i : rows) {
        while (stack.size() >= 2) {
            int j1 = stack[stack.size() - 2];
            int j2 = stack[stack.size() - 1];
            // 比较两个行在列 stack.size()-1 处的值
            if (matrix[j1][stack.size() - 1] <= matrix[i][stack.size() - 1]) {
                break;
            } else {
                stack.pop_back();
            }
        }
        stack.push_back(i);
    }
    return stack;
}

vector<int> smawkRec(const vector<int>& rows, const vector<int>& cols, const
vector<vector<int>>& matrix) {
    /*递归实现 SMAWK 算法*/
    if (rows.empty()) {
        return {};
    }
}

```

```

// 行压缩
vector<int> reducedRows = reduceRows(rows, matrix);

// 递归求解列数为奇数的子问题
vector<int> halfCols;
for (int i = 1; i < cols.size(); i += 2) {
    halfCols.push_back(cols[i]);
}
vector<int> minCols(reducedRows.size(), -1);

if (!halfCols.empty()) {
    // 递归求解
    vector<int> result = smawkRec(reducedRows, halfCols, matrix);
    // 复制结果
    for (int i = 0; i < result.size(); ++i) {
        minCols[i] = result[i];
    }
}

// 扩展结果到所有列
vector<int> result(rows.size(), 0);
int k = 0; // minCols 的索引

for (int i = 0; i < rows.size(); ++i) {
    int row = rows[i];
    // 确定当前行的最小值可能在哪个区间
    int start = (i == 0) ? 0 : (k > 0 ? minCols[k - 1] : 0);
    int end = (k < minCols.size()) ? minCols[k] : cols.back();

    // 在这个区间内查找最小值
    int minValue = INT_MAX;
    int minCol = start;

    // 注意这里 cols 是原始列的子集，需要在 cols 中遍历
    auto startIt = find(cols.begin(), cols.end(), start);
    auto endIt = find(cols.begin(), cols.end(), end);
    if (startIt != cols.end() && endIt != cols.end()) {
        for (auto it = startIt; it <= endIt; ++it) {
            int col = *it;
            if (col < matrix[0].size() && matrix[row][col] < minValue) {
                minValue = matrix[row][col];
                minCol = col;
            }
        }
    }
}

```

```

        }
    }

    result[i] = minCol;

    // 如果当前行在 reducedRows 中，且不是最后一行，k 前进
    if (k < reducedRows.size() && row == reducedRows[k]) {
        k++;
    }
}

return result;
}

```

```

vector<int> smawk(const vector<vector<int>>& matrix) {
/*

```

SMAWK 算法用于在 Monge 矩阵中快速查找每行的最小值

问题描述:

给定一个 Monge 矩阵，快速找到每行的最小值位置

解题思路:

1. Monge 矩阵满足性质: $\text{matrix}[i][j] + \text{matrix}[i+1][j+1] \leq \text{matrix}[i][j+1] + \text{matrix}[i+1][j]$
2. SMAWK 算法利用这一性质，可以在 $O(m+n)$ 时间内找到每行的最小值
3. 主要步骤包括行压缩和递归求解

参数:

matrix: 一个 Monge 矩阵

返回:

vector<int>: 每行最小值的列索引

时间复杂度: $O(m+n)$ ，其中 m 是行数， n 是列数

空间复杂度: $O(m+n)$

*/

```

int m = matrix.size();
if (m == 0) {
    return {};
}
int n = matrix[0].size();

```

// 构造行索引和列索引数组

```

vector<int> rows, cols;

```

```

for (int i = 0; i < m; ++i) rows.push_back(i);
for (int i = 0; i < n; ++i) cols.push_back(i);

// 调用递归实现
vector<int> result = smawkRec(rows, cols, matrix);

return result;
}

// ===== 优化体系: Aliens Trick (二分约束参数+可行性 DP) =====

struct AliensTrickResult {
    double lambda;
    double value;

    AliensTrickResult(double lambda, double value)
        : lambda(lambda), value(value) {}

};

using AliensCostFunction = function<pair<double, double>(double)>;
using CheckFunction = function<bool(double)>;

AliensTrickResult aliensTrick(AliensCostFunction costFunc, CheckFunction checkFunc,
                             double left, double right, double eps) {
/*
Aliens Trick (二分约束参数+可行性 DP)

```

问题描述:

解决带约束的优化问题，通常形如最小化总成本，同时满足某些约束条件

解题思路:

1. 将约束条件转化为参数 λ ，构造拉格朗日函数
2. 对 λ 进行二分查找，使用可行性 DP 判断当前 λ 下是否满足约束
3. 根据可行性 DP 的结果调整二分区间

参数:

`costFunc`: 计算带参数 λ 的成本函数，返回 `pair<double, double>`，其中 `first` 是当前值，`second` 是约束值

`checkFunc`: 检查当前解是否满足约束的函数

`left`: 二分左边界

`right`: 二分右边界

`eps`: 精度要求

返回：

AliensTrickResult：包含最优参数 λ 和对应最优解的结果类

时间复杂度： $O(\log((right-left)/\text{eps}) * T(DP))$ ，其中 $T(DP)$ 是一次 DP 的时间复杂度

*/

double bestLambda = left;

double bestValue = 0.0;

while (right - left > eps) {

 double mid = (left + right) / 2;

 // 计算当前参数下的解和约束值

 auto [currentValue, constraintValue] = costFunc(mid);

 if (checkFunc(constraintValue)) {

 // 满足约束，尝试更小的参数

 right = mid;

 bestLambda = mid;

 bestValue = currentValue;

 } else {

 // 不满足约束，需要增大参数

 left = mid;

 }

}

return AliensTrickResult(bestLambda, bestValue);

}

// 重载，提供默认精度

AliensTrickResult aliensTrick(AliensCostFunction costFunc, CheckFunction checkFunc,

 double left, double right) {

 return aliensTrick(costFunc, checkFunc, left, right, 1e-7);

}

// ===== 图上 DP→最短路：分层图建模 =====

int layeredGraphDijkstra(int n, int m, const vector<vector<int>>& edges, int k) {

/*

分层图 Dijkstra 算法

问题描述：

给定一个图，允许最多使用 k 次特殊操作（如跳跃、免费通行等），求最短路径

解题思路：

1. 构建分层图，每层代表使用不同次数的特殊操作
2. 对于每个节点 u ，在第 i 层表示到达 u 时已经使用了 i 次特殊操作
3. 使用 Dijkstra 算法在分层图上寻找最短路径

参数：

n : 节点数量

m : 边的数量

$edges$: 边的列表，每个元素为 $[u, v, w]$ 表示 u 到 v 的权为 w 的边

k : 允许使用的特殊操作次数

返回：

int: 从节点 0 到节点 $n-1$ 的最短路径长度

时间复杂度: $O((n*k + m*k) \log(n*k))$

空间复杂度: $O(n*k + m*k)$

*/

// 构建分层图的邻接表

```
vector<vector<pair<int, int>>> graph(n * (k + 1));
```

// 添加普通边（不使用特殊操作）

```
for (const auto& edge : edges) {
```

```
    int u = edge[0];
```

```
    int v = edge[1];
```

```
    int w = edge[2];
```

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

```
        graph[u + i * n].emplace_back(v + i * n, w);
```

```
}
```

```
}
```

// 添加使用特殊操作的边（如果允许的话）

```
for (const auto& edge : edges) {
```

```
    int u = edge[0];
```

```
    int v = edge[1];
```

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

```
        // 这里假设特殊操作可以免费通行（权为 0），具体根据问题调整
```

```
        graph[u + i * n].emplace_back(v + (i + 1) * n, 0);
```

```
}
```

```
}
```

// Dijkstra 算法

```
vector<int> dist(n * (k + 1), INT_MAX);
```

```
dist[0] = 0; // 假设起点是节点 0
```

```
priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> heap;
```

```

heap.emplace(0, 0); // (距离, 节点)

while (!heap.empty()) {
    auto [d, u] = heap.top();
    heap.pop();

    if (d > dist[u]) {
        continue;
    }

    for (const auto& [v, w] : graph[u]) {
        if (dist[v] > d + w) {
            dist[v] = d + w;
            heap.emplace(dist[v], v);
        }
    }
}

// 取所有层中到达终点的最小值
int result = INT_MAX;
for (int i = 0; i <= k; ++i) {
    if (dist[n - 1 + i * n] < result) {
        result = dist[n - 1 + i * n];
    }
}

return result;
}

// ===== 冷门模型：期望 DP 遇环的方程组解（高斯消元） =====

vector<double> gaussianElimination(vector<vector<double>> matrix) {
/*
    高斯消元法求解线性方程组
*/
}

```

问题描述：

求解形如 $Ax = b$ 的线性方程组

解题思路：

1. 构建增广矩阵
2. 进行高斯消元，将矩阵转化为行阶梯形
3. 回代求解

参数:

matrix: 增广矩阵, 每行最后一个元素是 b 的值

返回:

vector<double>: 方程组的解

时间复杂度: $O(n^3)$

空间复杂度: $O(n^2)$

*/

```
int n = matrix.size();
```

```
const double eps = 1e-9;
```

// 高斯消元过程

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

// 找到主元行 (当前列中绝对值最大的行)

```
    int maxRow = i;
```

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

```
        if (abs(matrix[j][i]) > abs(matrix[maxRow][i])) {
```

```
            maxRow = j;
```

```
        }
```

```
}
```

// 交换主元行和当前行

```
swap(matrix[i], matrix[maxRow]);
```

// 如果主元为 0, 方程组可能有无穷多解或无解

```
if (abs(matrix[i][i]) < eps) {
```

```
    continue;
```

```
}
```

// 消元过程

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

```
    double factor = matrix[j][i] / matrix[i][i];
```

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

```
        matrix[j][k] -= factor * matrix[i][k];
```

```
    }
```

```
}
```

```
}
```

// 回代求解

```
vector<double> x(n);
```

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

```
    x[i] = matrix[i][n];
```

```

        for (int j = i + 1; j < n; ++j) {
            x[i] -= matrix[i][j] * x[j];
        }
        x[i] /= matrix[i][i];
    }

    return x;
}

vector<double> expectationDPWithCycles(int n, const vector<vector<pair<int, double>>>&
transitions) {
/*
期望 DP 处理有环情况（使用高斯消元）

```

问题描述:

在有环的状态转移图中计算期望

解题思路:

1. 对于每个状态，建立期望方程
2. 使用高斯消元求解方程组

参数:

n: 状态数量

transitions: 转移概率列表, transitions[i]是一个列表, 每个元素为[j, p]表示从 i 转移到 j 的概率为 p

返回:

vector<double>: 每个状态的期望值

时间复杂度: $O(n^3)$

空间复杂度: $O(n^2)$

*/

// 构建线性方程组的增广矩阵

vector<vector<double>> matrix(n, vector<double>(n + 1, 0.0));

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

matrix[i][i] = 1.0; // 方程左边: E[i] - sum(p_ij * E[j]) = cost[i]

// 假设每个状态的代价为 1, 具体根据问题调整

double cost = 1.0;

matrix[i][n] = cost;

for (const auto& [j, p] : transitions[i]) {

```

        if (i != j) { // 避免自环的特殊处理
            matrix[i][j] -= p;
        }
    }

// 使用高斯消元求解
return gaussianElimination(matrix);
}

```

// ====== 冷门模型：插头 DP（轮廓线 DP） ======

```

int plugDP(const vector<vector<int>>& grid) {
/*
插头 DP（轮廓线 DP）示例：求网格中哈密顿回路的数量

```

问题描述：

给定一个网格，求其中哈密顿回路的数量

解题思路：

1. 使用轮廓线 DP，记录当前处理到的位置和轮廓线状态
2. 插头表示连接的状态，通常用二进制表示
3. 使用哈希表优化空间复杂度

参数：

grid: 网格，1 表示可通行，0 表示障碍物

返回：

int: 哈密顿回路的数量

时间复杂度：O($n \times m \times 4^{\min(n, m)}$)

空间复杂度：O($4^{\min(n, m)}$)

*/

```

int n = grid.size();
if (n == 0) {
    return 0;
}
int m = grid[0].size();

```

// 使用哈希表优化

unordered_map<long long, int> dp;

// 初始状态：左上角没有插头

```

dp[0] = 1;

for (int i = 0; i < n; ++i) {
    // 新的一行开始，需要将状态左移一位
    unordered_map<long long, int> newDp;
    for (const auto& [state, cnt] : dp) {
        // 左移一位，移除最左边的插头
        long long newState = state << 1;
        newDp[newState] += cnt;
    }
    dp = move(newDp);
}

for (int j = 0; j < m; ++j) {
    unordered_map<long long, int> newDp2;

    for (const auto& [state, cnt] : dp) {
        // 当前位置左边和上边的插头状态
        int left = (state >> (2 * j)) & 3;
        int up = (state >> (2 * (j + 1))) & 3;

        // 如果当前位置是障碍物，跳过
        if (grid[i][j] == 0) {
            // 只有当左右插头都不存在时才合法
            if (left == 0 && up == 0) {
                newDp2[state] += cnt;
            }
            continue;
        }

        // 处理各种插头组合情况
        // 1. 没有左插头和上插头
        if (left == 0 && up == 0) {
            // 只能创建新的插头对（用于回路的开始）
            if (i < n - 1 && j < m - 1 && grid[i+1][j] == 1 && grid[i][j+1] == 1) {
                long long newState = state | (1LL << (2 * j)) | (2LL << (2 * (j + 1)));
                newDp2[newState] += cnt;
            }
        }

        // 2. 只有左插头
        else if (left != 0 && up == 0) {
            // 向下延伸
        }
    }
}

```

```

        if (i < n - 1 && grid[i+1][j] == 1) {
            newDp2[state] += cnt;
        }
        // 向右延伸
        if (j < m - 1 && grid[i][j+1] == 1) {
            long long newState = (state & ~(3LL << (2 * j))) | (left << (2 * (j +
1)));
            newDp2[newState] += cnt;
        }
    }

// 3. 只有上插头
else if (left == 0 && up != 0) {
    // 向右延伸
    if (j < m - 1 && grid[i][j+1] == 1) {
        newDp2[state] += cnt;
    }
    // 向下延伸
    if (i < n - 1 && grid[i+1][j] == 1) {
        long long newState = (state & ~(3LL << (2 * (j + 1)))) | (up << (2 *
j));
        newDp2[newState] += cnt;
    }
}

// 4. 同时有左插头和上插头
else {
    // 合并插头
    long long newState = (state & ~(3LL << (2 * j))) & ~(3LL << (2 * (j +
1)));
}

// 如果是形成回路的最后一步
if (left == up) {
    // 检查是否所有插头都已连接
    if (newState == 0 && i == n - 1 && j == m - 1) {
        newDp2[newState] += cnt;
    }
} else {
    // 合并两个不同的插头
    newDp2[newState] += cnt;
}
}
}

```

```

        dp = move(newDp2) ;
    }
}

// 最终状态应该是没有任何插头（形成回路）
return dp.count(0) ? dp[0] : 0;
}

// ===== 冷门模型：树上背包的优化 =====

void dfsTreeKnapsack(int u, int parent, int capacity,
                      const vector<vector<int>>& tree, const vector<int>& weights,
                      const vector<int>& values, vector<vector<int>>& dp, vector<int>& size) {
    // 初始化当前节点
    size[u] = 1;
    if (weights[u] <= capacity) {
        dp[u][weights[u]] = values[u];
    }

    // 对每个子节点，按照子树大小排序，小的先合并
    vector<pair<int, int>> children;
    for (int v : tree[u]) {
        if (v != parent) {
            dfsTreeKnapsack(v, u, capacity, tree, weights, values, dp, size);
            children.emplace_back(size[v], v);
        }
    }
}

// 按子树大小排序
sort(children.begin(), children.end());

for (const auto& [sz, v] : children) {
    // 逆序遍历容量，避免重复计算
    for (int i = min(size[u], capacity); i >= 0; --i) {
        if (dp[u][i] == 0 && i != 0) continue;
        for (int j = 1; j <= min(sz, capacity - i); ++j) {
            if (dp[v][j] > 0 && i + j <= capacity) {
                dp[u][i + j] = max(dp[u][i + j], dp[u][i] + dp[v][j]);
            }
        }
    }
}

```

```

    // 更新子树大小
    size[u] += sz;
}
}

int treeKnapsackOptimized(int root, int capacity, const vector<vector<int>>& tree,
                           const vector<int>& weights, const vector<int>& values) {
/*
树上背包的优化实现（小到大合并）

```

问题描述：

在树上选择一些节点，使得总重量不超过容量，且总价值最大

解题思路：

1. 使用后序遍历处理子树
2. 使用小到大合并的策略优化复杂度
3. 对于每个节点，维护一个容量为 capacity 的背包

参数：

root: 根节点
 capacity: 背包容量
 tree: 树的邻接表
 weights: 每个节点的重量
 values: 每个节点的价值

返回：

int: 最大价值

时间复杂度：O(n*capacity^2)，但通过小到大合并可以降低常数

空间复杂度：O(n*capacity)

*/

```

int n = tree.size();
vector<vector<int>> dp(n, vector<int>(capacity + 1, 0));
vector<int> size(n, 0);

```

// 深度优先搜索处理子树

```
dfsTreeKnapsack(root, -1, capacity, tree, weights, values, dp, size);
```

// 返回根节点的最大价值

```

int maxVal = 0;
for (int val : dp[root]) {
    maxVal = max(maxVal, val);
}

```

```

    return maxVal;
}

// ===== 补充题目与应用 =====
// 以下是一些使用上述高级 DP 技术的经典题目及其代码实现

```

```

// 1. 编辑距离问题 (LeetCode 72)
int editDistance(const string& word1, const string& word2) {
/*
LeetCode 72. 编辑距离
题目链接: https://leetcode-cn.com/problems/edit-distance/

```

问题描述:

给你两个单词 word1 和 word2，计算出将 word1 转换成 word2 所使用的最少操作数。
你可以对一个单词进行如下三种操作：插入一个字符、删除一个字符、替换一个字符。

解题思路:

使用二维 DP， $dp[i][j]$ 表示 word1 的前 i 个字符转换为 word2 的前 j 个字符所需的最少操作数。

时间复杂度: $O(m \times n)$

空间复杂度: $O(m \times n)$

*/

```

int m = word1.size();
int n = word2.size();
// dp[i][j] 表示 word1 的前 i 个字符转换为 word2 的前 j 个字符所需的最少操作数
vector<vector<int>> dp(m + 1, vector<int>(n + 1));

```

// 初始化边界

```

for (int i = 0; i <= m; ++i) {
    dp[i][0] = i;
}
for (int j = 0; j <= n; ++j) {
    dp[0][j] = j;
}

```

// 动态规划填表

```

for (int i = 1; i <= m; ++i) {
    for (int j = 1; j <= n; ++j) {
        if (word1[i - 1] == word2[j - 1]) {
            dp[i][j] = dp[i - 1][j - 1];
        } else {
            dp[i][j] = min({dp[i - 1][j], dp[i][j - 1], dp[i - 1][j - 1]}) + 1;
        }
    }
}

```

```

        }
    }

    return dp[m][n];
}

```

// 2. 最长递增子序列 (LeetCode 300)

```

int lengthOfLIS(const vector<int>& nums) {
/*

```

LeetCode 300. 最长递增子序列

题目链接: <https://leetcode-cn.com/problems/longest-increasing-subsequence/>

问题描述:

给你一个整数数组 `nums`，找到其中最长严格递增子序列的长度。

解题思路:

使用贪心 + 二分查找优化的 DP 方法。

`tails[i]` 表示长度为 $i+1$ 的递增子序列的末尾元素的最小值。

时间复杂度: $O(n \log n)$

空间复杂度: $O(n)$

*/

```

if (nums.empty()) {
    return 0;
}

```

`vector<int> tails;`

```

for (int num : nums) {
    // 二分查找 num 应该插入的位置
    auto it = lower_bound(tails.begin(), tails.end(), num);
    if (it == tails.end()) {
        tails.push_back(num);
    } else {
        *it = num;
    }
}

```

`return tails.size();`

}

// 3. 背包问题变种 - 完全背包 (LeetCode 322)

```

int coinChange(const vector<int>& coins, int amount) {
/*

```

LeetCode 322. 零钱兑换

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

问题描述:

给你一个整数数组 coins，表示不同面额的硬币；以及一个整数 amount，表示总金额。

计算并返回可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额，返回 -1。

解题思路:

使用完全背包的思想， $dp[i]$ 表示凑成金额 i 所需的最少硬币数。

时间复杂度: $O(amount * n)$

空间复杂度: $O(amount)$

*/

// 初始化 dp 数组为无穷大

vector<int> dp(amount + 1, INT_MAX);

$dp[0] = 0$; // 凑成金额 0 需要 0 个硬币

```
for (int coin : coins) {  
    for (int i = coin; i <= amount; ++i) {  
        if (dp[i - coin] != INT_MAX) {  
            dp[i] = min(dp[i], dp[i - coin] + 1);  
        }  
    }  
}  
  
return dp[amount] == INT_MAX ? -1 : dp[amount];  
}
```

// 4. 矩阵链乘法（区间 DP 的经典应用）

```
struct MatrixChainResult {  
    vector<vector<int>> dp;  
    vector<vector<int>> s;
```

```
MatrixChainResult(const vector<vector<int>>& dp, const vector<vector<int>>& s)  
    : dp(dp), s(s) {}  
};
```

```
MatrixChainResult matrixChainOrder(const vector<int>& p) {
```

/*

矩阵链乘法问题

题目来源: 算法导论

问题描述：

给定一系列矩阵，计算乘法顺序使得标量乘法的次数最少。

解题思路：

使用区间 DP， $dp[i][j]$ 表示计算第 i 到第 j 个矩阵的乘积所需的最少标量乘法次数。

可以使用 Knuth 优化进一步降低时间复杂度。

时间复杂度： $O(n^3)$

空间复杂度： $O(n^2)$

*/

```
int n = p.size() - 1; // 矩阵的个数
// dp[i][j] 表示计算第 i 到第 j 个矩阵的乘积所需的最少标量乘法次数
vector<vector<int>> dp(n + 1, vector<int>(n + 1));
// s[i][j] 记录最优分割点
vector<vector<int>> s(n + 1, vector<int>(n + 1));

// 枚举区间长度
for (int length = 2; length <= n; ++length) {
    for (int i = 1; i + length - 1 <= n; ++i) {
        int j = i + length - 1;
        dp[i][j] = INT_MAX;
        // 枚举分割点
        for (int k = i; k < j; ++k) {
            // 计算当前分割点的代价
            int cost = dp[i][k] + dp[k + 1][j] + p[i - 1] * p[k] * p[j];
            if (cost < dp[i][j]) {
                dp[i][j] = cost;
                s[i][j] = k;
            }
        }
    }
}

return MatrixChainResult(dp, s);
}
```

// 5. 旅行商问题 (TSP) 的 DP 实现

```
int travelingSalesmanProblem(const vector<vector<int>>& graph) {
/*
旅行商问题
题目来源：算法竞赛经典问题
```

问题描述：

给定一个完全图，找到一条访问每个城市恰好一次并返回起点的最短路径。

解题思路：

使用状态压缩 DP， $dp[mask][u]$ 表示访问过的城市集合为 $mask$ ，当前在城市 u 时的最短路径长度。

时间复杂度： $O(n^2 * 2^n)$

空间复杂度： $O(n * 2^n)$

*/

```
int n = graph.size();
```

```
// dp[mask][u] 表示访问过的城市集合为 mask，当前在城市 u 时的最短路径长度
```

```
vector<vector<int>> dp(1 << n, vector<int>(n, INT_MAX));
```

```
// 初始状态：只访问了起点，路径长度为 0
```

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

```
// 枚举所有可能的状态
```

```
for (int mask = 1; mask < (1 << n); ++mask) {
```

```
    // 枚举当前所在的城市
```

```
    for (int u = 0; u < n; ++u) {  
        if (!(mask & (1 << u))) {  
            continue;  
        }
```

```
        // 枚举下一个要访问的城市
```

```
        for (int v = 0; v < n; ++v) {  
            if (mask & (1 << v)) {  
                continue;  
            }
```

```
            int newMask = mask | (1 << v);
```

```
            if (dp[mask][u] != INT_MAX && graph[u][v] != INT_MAX) {
```

```
                dp[newMask][v] = min(dp[newMask][v], dp[mask][u] + graph[u][v]);
```

```
}
```

```
}
```

```
}
```

```
// 找到最短的回路
```

```
int result = INT_MAX;
```

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

```
    if (dp[(1 << n) - 1][u] != INT_MAX && graph[u][0] != INT_MAX) {
```

```
        result = min(result, dp[(1 << n) - 1][u] + graph[u][0]);
```

```
}
```

```
}
```

```
    return result;
```

```
}
```

```
// 6. 区间 DP: 最优三角剖分
```

```
int minimumScoreTriangulation(vector<int>& values) {
```

```
/*
```

```
LeetCode 1039. 多边形三角剖分的最低得分
```

```
题目链接: https://leetcode-cn.com/problems/minimum-score-triangulation-of-polygon/
```

问题描述:

给定一个凸多边形，将其三角剖分，使得所有三角形的顶点乘积之和最小。

解题思路:

使用区间 DP， $dp[i][j]$ 表示从顶点 i 到顶点 j 的多边形三角剖分的最小得分。

时间复杂度: $O(n^3)$

空间复杂度: $O(n^2)$

```
*/
```

```
int n = values.size();
```

```
//  $dp[i][j]$  表示从顶点  $i$  到顶点  $j$  的多边形三角剖分的最小得分
```

```
vector<vector<int>> dp(n, vector<int>(n));
```

```
// 枚举区间长度
```

```
for (int length = 3; length <= n; ++length) {
```

```
    for (int i = 0; i + length - 1 < n; ++i) {
```

```
        int j = i + length - 1;
```

```
        dp[i][j] = INT_MAX;
```

```
// 枚举中间点
```

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

```
            dp[i][j] = min(dp[i][j],
```

```
                           dp[i][k] + dp[k][j] + values[i] * values[k] * values[j]);
```

```
}
```

```
}
```

```
}
```

```
return dp[0][n - 1];
```

```
}
```

```
// 7. 博弈 DP: 石子游戏
```

```
bool stoneGame(vector<int>& piles) {
```

```
/*
```

LeetCode 877. 石子游戏

题目链接: <https://leetcode-cn.com/problems/stone-game/>

问题描述:

给定一个表示石子堆的数组，两个玩家轮流从两端取石子，每次只能取一个，取到最后一个石子的人获胜。

判断先手是否必胜。

解题思路:

使用区间 DP， $dp[i][j]$ 表示在区间 $[i, j]$ 中，先手能获得的最大净胜分。

时间复杂度: $O(n^2)$

空间复杂度: $O(n^2)$

*/

```
int n = piles.size();
```

```
// dp[i][j] 表示在区间 [i, j] 中，先手能获得的最大净胜分
```

```
vector<vector<int>> dp(n, vector<int>(n));
```

```
// 初始化单个石子堆
```

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

```
    dp[i][i] = piles[i];
```

```
}
```

```
// 枚举区间长度
```

```
for (int length = 2; length <= n; ++length) {
```

```
    for (int i = 0; i + length - 1 < n; ++i) {
```

```
        int j = i + length - 1;
```

```
        // 先手可以选择取左边或右边
```

```
        dp[i][j] = max(piles[i] - dp[i + 1][j], piles[j] - dp[i][j - 1]);
```

```
}
```

```
}
```

```
// 先手净胜分大于 0 则必胜
```

```
return dp[0][n - 1] > 0;
```

```
}
```

```
// 8. 数位 DP: 统计 1 出现的次数
```

```
int countDigitOne(int n) {
```

```
/*
```

LeetCode 233. 数字 1 的个数

题目链接: <https://leetcode-cn.com/problems/number-of-digit-one/>

问题描述:

给定一个整数 n，计算所有小于等于 n 的非负整数中数字 1 出现的个数。

解题思路：

使用数位 DP，逐位处理每一位上 1 出现的次数。

```
时间复杂度: O(log n)
空间复杂度: O(log n)
*/
if (n <= 0) {
    return 0;
}

string s = to_string(n);
int length = s.size();
int count = 0;

// 逐位处理
for (int i = 0; i < length; ++i) {
    long long high = 0;
    if (i > 0) {
        high = stoll(s.substr(0, i));
    }
    int current = s[i] - '0';
    long long low = 0;
    if (i < length - 1) {
        low = stoll(s.substr(i + 1));
    }
    long long digit = pow(10, length - i - 1);

    if (current == 0) {
        // 当前位为 0，高位决定
        count += high * digit;
    } else if (current == 1) {
        // 当前位为 1，高位+低位+1
        count += high * digit + low + 1;
    } else {
        // 当前位大于 1，高位+1
        count += (high + 1) * digit;
    }
}

return count;
}
```

```

// 9. 树形 DP: 打家劫舍 III

struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

pair<int, int> robDFS(TreeNode* node) {
    if (!node) {
        return {0, 0};
    }

    auto left = robDFS(node->left);
    auto right = robDFS(node->right);

    // 偷当前节点, 不能偷子节点
    int robCurrent = node->val + left.second + right.second;
    // 不偷当前节点, 可以选择偷或不偷子节点
    int notRobCurrent = max(left.first, left.second) + max(right.first, right.second);

    return {robCurrent, notRobCurrent};
}

int rob(TreeNode* root) {
/*
LeetCode 337. 打家劫舍 III
题目链接: https://leetcode-cn.com/problems/house-robber-iii/
*/
}

```

问题描述:

在上次打劫完一条街道之后和一圈房屋后，小偷又发现了一个新的可行窃的地区。

这个地区只有一个入口，我们称之为“根”。除了“根”之外，每栋房子有且只有一个“父”房子与之相连。

一番侦察之后，聪明的小偷意识到“这个地方的所有房屋的排列类似于一棵二叉树”。

如果两个直接相连的房子在同一天晚上被打劫，房屋将自动报警。

计算在不触动警报的情况下，小偷一晚能够盗取的最高金额。

解题思路:

使用树形 DP，对于每个节点，维护两个状态：偷或不偷。

时间复杂度: $O(n)$

空间复杂度: $O(h)$ ， h 为树的高度

```

*/
auto [robRoot, notRobRoot] = robDFS(root);
return max(robRoot, notRobRoot);
}

```

// 10. 状态压缩 DP: 蒙斯特曼问题

```
int monsterGame(const vector<vector<int>>& grid) {
```

```
/*

```

蒙斯特曼问题

题目来源: 算法竞赛问题

问题描述:

在网格中放置怪物，使得任何两个怪物都不在同一行、同一列或对角线上。

解题思路:

使用状态压缩 DP， $dp[i][mask]$ 表示处理到第 i 行，已放置的列的状态为 $mask$ 时的方案数。

时间复杂度: $O(n * 2^n)$

空间复杂度: $O(2^n)$

```
*/
```

```
int n = grid.size();
```

// $dp[i][mask]$ 表示处理到第 i 行，已放置的列的状态为 $mask$ 时的方案数

```
vector<long long> dp(1 << n, 0);
```

```
dp[0] = 1;
```

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

```
    vector<long long> newDp(1 << n, 0);
```

```
    for (int mask = 0; mask < (1 << n); ++mask) {
```

```
        if (dp[mask] == 0) {
```

```
            continue;
```

```
}
```

// 枚举所有可能的放置位置

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

// 检查是否可以在 (i, j) 放置怪物

```
    if (!(mask & (1 << j)) && grid[i][j] == 1) {
```

// 检查对角线

```
        bool valid = true;
```

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

```
            if ((mask & (1 << k)) && abs(k - j) == i - k) {
```

```
                valid = false;
```

```
                break;
```

```
}
```

```
}
```

```

        if (valid) {
            newDp[mask | (1 << j)] += dp[mask];
        }
    }
}

dp = move(newDp);
}

return (int) dp[(1 << n) - 1];
}

```

// 11. 高维 DP: 三维背包

```
int threeDimensionKnapsack(int n, const vector<int>& capacity, const vector<vector<int>>& items) {
```

```
/*

```

三维背包问题

题目来源: 算法竞赛问题

问题描述:

有 n 个物品，每个物品有体积、重量、价值三个属性，背包有体积和重量两个限制，求最大价值。

解题思路:

使用三维 DP， $dp[i][j][k]$ 表示前 i 个物品，体积为 j，重量为 k 时的最大价值。

时间复杂度: $O(n * V * W)$

空间复杂度: $O(n * V * W)$

```
*/
```

```
int V = capacity[0];
```

```
int W = capacity[1];
```

// 初始化 dp 数组

```
vector<vector<vector<int>>> dp(n + 1, vector<vector<int>>(V + 1, vector<int>(W + 1, 0)));
```

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

```
    int v = items[i-1][0];
```

```
    int w = items[i-1][1];
```

```
    int val = items[i-1][2];
```

```
    for (int j = 0; j <= V; ++j) {
```

```
        for (int k = 0; k <= W; ++k) {
```

// 不选当前物品

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

// 选当前物品（如果有足够的空间）

```
        if (j >= v && k >= w) {
```

```

        dp[i][j][k] = max(dp[i][j][k], dp[i-1][j-v][k-w] + val);
    }
}
}

return dp[n][V][W];
}

```

// 12. 斜率优化 DP 示例

```

class ConvexHullTrick {
public:
/*
凸包优化技巧示例
题目来源：算法竞赛问题

```

问题描述：

当状态转移方程形如 $dp[i] = \min\{dp[j] + a[i] * b[j] + c\}$ 时，可以使用凸包优化。

解题思路：

将转移方程转换为直线的形式，维护凸包以快速查询最小值。

时间复杂度：O(n)

空间复杂度：O(n)

*/

```

struct Line {
    long long k, b;
    Line(long long k, long long b) : k(k), b(b) {}
};
```

deque<Line> dq;

// 添加一条直线 $y = kx + b$

```

void addLine(long long k, long long b) {
    // 当队列中至少有两条直线时，检查是否需要删除末尾的直线
    while (dq.size() >= 2) {
        Line l1 = dq[dq.size() - 2];
        Line l2 = dq[dq.size() - 1];
        // 判断直线 l1 和 l2 的交点是否在 l2 和新直线的交点右侧
        if ((l2.b - l1.b) * (k - l2.k) >= (b - l2.b) * (l2.k - l1.k)) {
            dq.pop_back();
        } else {
```

```

        break;
    }
}

dq.emplace_back(k, b);
}

// 查询 x 处的最小值
long long query(long long x) {
    // 如果队列中至少有两条直线，且第一条直线在 x 处的值大于第二条，删除第一条
    while (dq.size() >= 2) {
        Line l1 = dq[0];
        Line l2 = dq[1];
        if (l1.k * x + l1.b >= l2.k * x + l2.b) {
            dq.pop_front();
        } else {
            break;
        }
    }
    if (dq.empty()) {
        return LLONG_MAX;
    }
    Line l = dq[0];
    return l.k * x + l.b;
}
};

};;;

// ===== DP+计算几何（凸包相关） =====

/***
 * 点结构体
 */
struct Point {
    double x, y;
    Point(double x = 0, double y = 0) : x(x), y(y) {}
    bool operator < (const Point& p) const {
        return x < p.x || (x == p.x && y < p.y);
    }
}

// ===== 优化体系：Knuth 优化 =====

// Knuth 优化用于优化形如 dp[i][j] = min{dp[i][k] + dp[k+1][j]} + w(i, j) 的 DP
// 当满足四边形不等式时，最优转移点单调

```

```

struct KnuthOptimizationResult {
    vector<vector<int>> dp;
    vector<vector<int>> opt;

    KnuthOptimizationResult(const vector<vector<int>>& dp, const vector<vector<int>>& opt)
        : dp(dp), opt(opt) {}

};

using CostFunction = function<int(int, int)>;

KnuthOptimizationResult knuthOptimization(int n, CostFunction costFunc) {
    /*
     * Knuth 优化的 DP 算法
    */
}

```

问题描述:

解决区间 DP 问题，其中状态转移方程满足四边形不等式

解题思路:

1. 使用 Knuth 优化将时间复杂度从 $O(n^3)$ 降低到 $O(n^2)$
2. 维护最优转移点数组 $opt[i][j]$ ，表示计算 $dp[i][j]$ 时的最优 k 值
3. 根据 $opt[i][j-1] \leq opt[i][j] \leq opt[i+1][j]$ 的性质进行剪枝

参数:

n : 区间长度

$costFunc$: 计算区间 (i, j) 代价的函数

返回:

$KnuthOptimizationResult$: 包含 dp 数组和 opt 数组的结果类

时间复杂度: $O(n^2)$

空间复杂度: $O(n^2)$

*/

// 初始化 dp 和 opt 数组

```
vector<vector<int>> dp(n + 1, vector<int>(n + 1));
```

```
vector<vector<int>> opt(n + 1, vector<int>(n + 1));
```

// 初始化长度为 1 的区间

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

```
    dp[i][i] = 0;
```

```
    opt[i][i] = i;
```

```
}
```

```

// 枚举区间长度
for (int length = 2; length <= n; ++length) {
    // 枚举起始点
    for (int i = 1; i + length - 1 <= n; ++i) {
        int j = i + length - 1;
        // 初始化为无穷大
        dp[i][j] = INT_MAX;
        // 根据 Knuth 优化的性质，最优 k 在 opt[i][j-1] 到 opt[i+1][j] 之间
        int upperK = (i + 1 <= j) ? opt[i + 1][j] : j - 1;
        for (int k = opt[i][j - 1]; k <= min(upperK, j - 1); ++k) {
            if (dp[i][k] != INT_MAX && dp[k + 1][j] != INT_MAX) {
                int cost = costFunc(i, j);
                if (cost != INT_MAX) {
                    int current = dp[i][k] + dp[k + 1][j] + cost;
                    if (current < dp[i][j]) {
                        dp[i][j] = current;
                        opt[i][j] = k;
                    }
                }
            }
        }
    }
}

return KnuthOptimizationResult(dp, opt);
}

```

// ===== 优化体系: Divide & Conquer Optimization =====

```

void solveDivideConquer(int i, int l, int r, int opt_l, int opt_r,
                       vector<vector<int>>& dp, CostFunction costFunc) {
/*
计算 dp[i][l..r]，其中最优转移点在 opt_l..opt_r 之间
*/
if (l > r) {
    return;
}

int mid = (l + r) / 2;
int best_k = opt_l;

// 在 opt_l 到 min(mid, opt_r) 之间寻找最优 k
for (int k = opt_l; k <= min(mid, opt_r); ++k) {

```

```

    if (dp[i - 1][k] != INT_MAX) {
        int cost = costFunc(k, mid);
        if (cost != INT_MAX) {
            int current = dp[i - 1][k] + cost;
            if (current < dp[i][mid]) {
                dp[i][mid] = current;
                best_k = k;
            }
        }
    }
}

// 递归处理左右子区间
solveDivideConquer(i, 1, mid - 1, opt_l, best_k, dp, costFunc);
solveDivideConquer(i, mid + 1, r, best_k, opt_r, dp, costFunc);
}

```

```

vector<vector<int>> divideConquerOptimization(int n, int m, CostFunction costFunc) {
/*
Divide & Conquer Optimization (分治优化)

```

问题描述:

解决形如 $dp[i][j] = \min\{dp[i-1][k] + cost(k, j)\}$, 其中 $k < j$
当转移满足决策单调性时使用

解题思路:

1. 利用决策单调性, 使用分治法优化 DP
2. 对于 $dp[i][j]$, 当 i 固定时, 最优转移点 k 随着 j 的增加而单调不减
3. 使用分治的方式计算每个区间的最优决策

参数:

n: 维度 1
m: 维度 2
costFunc: 计算 $cost(k, j)$ 的函数

返回:

vector<vector<int>>: dp 数组

时间复杂度: $O(n*m \log m)$

空间复杂度: $O(n*m)$

*/

// 初始化 dp 数组

vector<vector<int>> dp(n + 1, vector<int>(m + 1, INT_MAX));

```

dp[0][0] = 0;

// 对每个 i 应用分治优化
for (int i = 1; i <= n; ++i) {
    solveDivideConquer(i, 1, m, 0, m, dp, costFunc);
}

return dp;
}

// ===== 优化体系: SMAWK 算法 (行最小查询) =====

vector<int> reduceRows(const vector<int>& rows, const vector<vector<int>>& matrix) {
    /*行压缩: 只保留可能成为最小值的行*/
    stack<int> stk;
    for (int i : rows) {
        while (stk.size() >= 2) {
            int j1 = stk.top();
            stk.pop();
            int j2 = stk.top();
            stk.push(j1); // 恢复栈状态

            // 比较两个行在列 stk.size()-1 处的值
            if (matrix[j2][stk.size() - 1] <= matrix[i][stk.size() - 1]) {
                break;
            } else {
                stk.pop();
            }
        }
        stk.push(i);
    }

    vector<int> result;
    while (!stk.empty()) {
        result.push_back(stk.top());
        stk.pop();
    }
    reverse(result.begin(), result.end());
    return result;
}

vector<int> smawkRec(const vector<int>& rows, const vector<int>& cols, const
vector<vector<int>>& matrix) {

```

```

/*递归实现 SMAWK 算法*/
if (rows.empty()) {
    return {};
}

// 行压缩
vector<int> reducedRows = reduceRows(rows, matrix);

// 递归求解列数为奇数的子问题
vector<int> halfCols;
for (int i = 1; i < cols.size(); i += 2) {
    halfCols.push_back(cols[i]);
}
vector<int> minCols(reducedRows.size(), -1);

if (!halfCols.empty()) {
    // 递归求解
    vector<int> result = smawkRec(reducedRows, halfCols, matrix);
    // 复制结果
    for (int i = 0; i < result.size(); ++i) {
        minCols[i] = result[i];
    }
}

// 扩展结果到所有列
vector<int> result(rows.size(), 0);
int k = 0; // minCols 的索引

for (int i = 0; i < rows.size(); ++i) {
    int row = rows[i];
    // 确定当前行的最小值可能在哪个区间
    int start = (i == 0) ? 0 : (k > 0 ? minCols[k - 1] : 0);
    int end = (k < minCols.size()) ? minCols[k] : cols.back();

    // 在这个区间内查找最小值
    int minValue = INT_MAX;
    int minCol = start;

    // 注意这里 cols 是原始列的子集，需要在 cols 中遍历
    auto startIt = find(cols.begin(), cols.end(), start);
    auto endIt = find(cols.begin(), cols.end(), end);
    if (startIt != cols.end() && endIt != cols.end()) {
        for (auto it = startIt; it <= endIt; ++it) {

```

```

        int col = *it;
        if (col < matrix[0].size() && matrix[row][col] < minValue) {
            minValue = matrix[row][col];
            minCol = col;
        }
    }

    result[i] = minCol;

    // 如果当前行在 reducedRows 中，且不是最后一行，k 前进
    if (k < reducedRows.size() && row == reducedRows[k]) {
        k++;
    }
}

return result;
}

```

vector<int> smawk(const vector<vector<int>>& matrix) {

/*
SMAWK 算法用于在 Monge 矩阵中快速查找每行的最小值

问题描述：

给定一个 Monge 矩阵，快速找到每行的最小值位置

解题思路：

1. Monge 矩阵满足性质： $\text{matrix}[i][j] + \text{matrix}[i+1][j+1] \leq \text{matrix}[i][j+1] + \text{matrix}[i+1][j]$
2. SMAWK 算法利用这一性质，可以在 $O(m+n)$ 时间内找到每行的最小值
3. 主要步骤包括行压缩和递归求解

参数：

matrix: 一个 Monge 矩阵

返回：

vector<int>: 每行最小值的列索引

时间复杂度： $O(m+n)$ ，其中 m 是行数， n 是列数

空间复杂度： $O(m+n)$

*/

```

int m = matrix.size();
if (m == 0) {
    return {};
}

```

```

    }

    int n = matrix[0].size();

    // 构造行索引和列索引数组
    vector<int> rows(m), cols(n);
    iota(rows.begin(), rows.end(), 0);
    iota(cols.begin(), cols.end(), 0);

    // 调用递归实现
    vector<int> resultList = smawkRec(rows, cols, matrix);

    return resultList;
}

// ===== 优化体系: Aliens Trick (二分约束参数+可行性 DP) =====

struct AliensTrickResult {
    double lambda;
    double value;

    AliensTrickResult(double lambda, double value)
        : lambda(lambda), value(value) {}

};

using AliensCostFunction = function<vector<double>(double)>; // 返回[value, constraint]
using CheckFunction = function<bool(double)>;

AliensTrickResult aliensTrick(AliensCostFunction costFunc, CheckFunction checkFunc,
                             double left, double right, double eps = 1e-7) {
/*
    Aliens Trick (二分约束参数+可行性 DP)
*/
}

```

问题描述:

解决带约束的优化问题，通常形如最小化总成本，同时满足某些约束条件

解题思路:

1. 将约束条件转化为参数 λ ，构造拉格朗日函数
2. 对 λ 进行二分查找，使用可行性 DP 判断当前 λ 下是否满足约束
3. 根据可行性 DP 的结果调整二分区间

参数:

- costFunc: 计算带参数 λ 的成本函数，返回 [value, constraint] 数组
checkFunc: 检查当前解是否满足约束的函数

left: 二分左边界
right: 二分右边界
eps: 精度要求

返回:

AliensTrickResult: 包含最优参数 λ 和对应最优解的结果类

时间复杂度: $O(\log((right-left)/eps) * T(DP))$, 其中 $T(DP)$ 是一次 DP 的时间复杂度
*/

```
double bestLambda = left;  
double bestValue = 0.0;
```

```
while (right - left > eps) {  
    double mid = (left + right) / 2;  
    // 计算当前参数下的解和约束值  
    vector<double> result = costFunc(mid);  
    double currentValue = result[0];  
    double constraintValue = result[1];  
  
    if (checkFunc(constraintValue)) {  
        // 满足约束, 尝试更小的参数  
        right = mid;  
        bestLambda = mid;  
        bestValue = currentValue;  
    } else {  
        // 不满足约束, 需要增大参数  
        left = mid;  
    }  
}
```

```
return AliensTrickResult(bestLambda, bestValue);  
}
```

// ===== 图上 DP→最短路: 分层图建模 =====

```
int layeredGraphDijkstra(int n, int m, vector<vector<int>>& edges, int k) {  
/*  
分层图 Dijkstra 算法
```

问题描述:

给定一个图, 允许最多使用 k 次特殊操作 (如跳跃、免费通行等), 求最短路径

解题思路:

1. 构建分层图，每层代表使用不同次数的特殊操作
2. 对于每个节点 u ，在第 i 层表示到达 u 时已经使用了 i 次特殊操作
3. 使用 Dijkstra 算法在分层图上寻找最短路径

参数：

n : 节点数量

m : 边的数量

$edges$: 边的列表，每个元素为 $[u, v, w]$ 表示 u 到 v 的权为 w 的边

k : 允许使用的特殊操作次数

返回：

int : 从节点 0 到节点 $n-1$ 的最短路径长度

时间复杂度： $O((n*k + m*k) \log(n*k))$

空间复杂度： $O(n*k + m*k)$

*/

// 构建分层图的邻接表

```
vector<vector<vector<int>>> graph(n * (k + 1));
```

// 添加普通边（不使用特殊操作）

```
for (auto& edge : edges) {
```

```
    int u = edge[0];
```

```
    int v = edge[1];
```

```
    int w = edge[2];
```

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

```
        int from = u + i * n;
```

```
        graph[from].push_back({v + i * n, w});
```

```
}
```

```
}
```

// 添加使用特殊操作的边（如果允许的话）

```
for (auto& edge : edges) {
```

```
    int u = edge[0];
```

```
    int v = edge[1];
```

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

```
        // 这里假设特殊操作可以免费通行（权为 0），具体根据问题调整
```

```
        int from = u + i * n;
```

```
        graph[from].push_back({v + (i + 1) * n, 0});
```

```
}
```

```
}
```

// Dijkstra 算法

```
vector<int> dist(n * (k + 1), INT_MAX);
```

```

dist[0] = 0; // 假设起点是节点 0
// 使用优先队列，按距离排序
using PII = pair<int, int>; // (距离, 节点)
priority_queue<PII, vector<PII>, greater<PII>> heap;
heap.emplace(0, 0);

while (!heap.empty()) {
    auto [d, u] = heap.top();
    heap.pop();

    if (d > dist[u]) {
        continue;
    }

    for (auto& edge : graph[u]) {
        int v = edge[0];
        int w = edge[1];
        if (dist[v] > d + w) {
            dist[v] = d + w;
            heap.emplace(dist[v], v);
        }
    }
}

// 取所有层中到达终点的最小值
int result = INT_MAX;
for (int i = 0; i <= k; ++i) {
    result = min(result, dist[n - 1 + i * n]);
}

return result;
}

// ===== 冷门模型：期望 DP 遇环的方程组解（高斯消元） =====

```

```
vector<double> gaussianElimination(vector<vector<double>> matrix) {
```

```
/*
```

```
高斯消元法求解线性方程组
```

问题描述：

求解形如 $Ax = b$ 的线性方程组

解题思路：

1. 构建增广矩阵
2. 进行高斯消元，将矩阵转化为行阶梯形
3. 回代求解

参数：

matrix: 增广矩阵，每行最后一个元素是 b 的值

返回：

vector<double>: 方程组的解

时间复杂度: $O(n^3)$

空间复杂度: $O(n^2)$

*/

```
int n = matrix.size();
const double eps = 1e-9;
```

// 高斯消元过程

```
for (int i = 0; i < n; ++i) {
    // 找到主元行 (当前列中绝对值最大的行)
    int maxRow = i;
    for (int j = i; j < n; ++j) {
        if (fabs(matrix[j][i]) > fabs(matrix[maxRow][i])) {
            maxRow = j;
        }
    }
}
```

// 交换主元行和当前行

```
swap(matrix[i], matrix[maxRow]);
```

// 如果主元为 0, 方程组可能有无穷多解或无解

```
if (fabs(matrix[i][i]) < eps) {
    continue;
}
```

// 消元过程

```
for (int j = i + 1; j < n; ++j) {
    double factor = matrix[j][i] / matrix[i][i];
    for (int k = i; k <= n; ++k) {
        matrix[j][k] -= factor * matrix[i][k];
    }
}
```

```

// 回代求解
vector<double> x(n);
for (int i = n - 1; i >= 0; --i) {
    x[i] = matrix[i][n];
    for (int j = i + 1; j < n; ++j) {
        x[i] -= matrix[i][j] * x[j];
    }
    x[i] /= matrix[i][i];
}

return x;
}

vector<double> expectationDPWithCycles(int n, vector<vector<pair<int, double>>>& transitions)
{
/*
期望 DP 处理有环情况（使用高斯消元）

```

问题描述:

在有环的状态转移图中计算期望

解题思路:

1. 对于每个状态，建立期望方程
2. 使用高斯消元求解方程组

参数:

n: 状态数量

transitions: 转移概率列表, transitions[i]是一个列表, 每个元素为[j, p]表示从 i 转移到 j 的概率为 p

返回:

vector<double>: 每个状态的期望值

时间复杂度: $O(n^3)$

空间复杂度: $O(n^2)$

*/

// 构建线性方程组的增广矩阵

```
vector<vector<double>> matrix(n, vector<double>(n + 1, 0.0));
```

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

```
    matrix[i][i] = 1.0; // 方程左边: E[i] - sum(p_ij * E[j]) = cost[i]
```

// 假设每个状态的代价为 1, 具体根据问题调整

```

        double cost = 1.0;
        matrix[i][n] = cost;

        for (auto& transition : transitions[i]) {
            int j = transition.first;
            double p = transition.second;
            if (i != j) { // 避免自环的特殊处理
                matrix[i][j] -= p;
            }
        }
    }

    // 使用高斯消元求解
    return gaussianElimination(matrix);
}

// ====== 冷门模型：插头 DP（轮廓线 DP） ======

```

```

int plugDP(vector<vector<int>>& grid) {
/*
插头 DP（轮廓线 DP）示例：求网格中哈密顿回路的数量

```

问题描述：

给定一个网格，求其中哈密顿回路的数量

解题思路：

1. 使用轮廓线 DP，记录当前处理到的位置和轮廓线状态
2. 插头表示连接的状态，通常用二进制表示
3. 使用哈希表优化空间复杂度

参数：

grid: 网格，1 表示可通行，0 表示障碍物

返回：

int: 哈密顿回路的数量

时间复杂度：O($n \cdot m \cdot 4^{\min(n, m)}$)

空间复杂度：O($4^{\min(n, m)}$)

*/

```

int n = grid.size();
if (n == 0) {
    return 0;
}

```

```

int m = grid[0].size();

// 使用哈希表优化
unordered_map<long long, int> dp;

// 初始状态：左上角没有插头
dp[0LL] = 1;

for (int i = 0; i < n; ++i) {
    // 新的一行开始，需要将状态左移一位
    unordered_map<long long, int> newDp;
    for (auto& [state, cnt] : dp) {
        // 左移一位，移除最左边的插头
        long long newState = state << 1;
        newDp[newState] += cnt;
    }
    dp = newDp;

    for (int j = 0; j < m; ++j) {
        unordered_map<long long, int> newDp2;

        for (auto& [state, cnt] : dp) {
            // 当前位置左边和上边的插头状态
            int left = (state >> (2 * j)) & 3;
            int up = (state >> (2 * (j + 1))) & 3;

            // 如果当前位置是障碍物，跳过
            if (grid[i][j] == 0) {
                // 只有当左右插头都不存在时才合法
                if (left == 0 && up == 0) {
                    newDp2[state] += cnt;
                }
                continue;
            }

            // 处理各种插头组合情况
            // 1. 没有左插头和上插头
            if (left == 0 && up == 0) {
                // 只能创建新的插头对（用于回路的开始）
                if (i < n - 1 && j < m - 1 && grid[i+1][j] == 1 && grid[i][j+1] == 1) {
                    long long newState = state | (1LL << (2 * j)) | (2LL << (2 * (j + 1)));
                    newDp2[newState] += cnt;
                }
            }
        }
    }
}

```

```

        }

    }

// 2. 只有左插头
else if (left != 0 && up == 0) {
    // 向下延伸
    if (i < n - 1 && grid[i+1][j] == 1) {
        newDp2[state] += cnt;
    }
    // 向右延伸
    if (j < m - 1 && grid[i][j+1] == 1) {
        long long newState = (state & ~(3LL << (2 * j))) | (left << (2 * (j + 1)));
        newDp2[newState] += cnt;
    }
}

// 3. 只有上插头
else if (left == 0 && up != 0) {
    // 向右延伸
    if (j < m - 1 && grid[i][j+1] == 1) {
        newDp2[state] += cnt;
    }
    // 向下延伸
    if (i < n - 1 && grid[i+1][j] == 1) {
        long long newState = (state & ~(3LL << (2 * (j + 1)))) | (up << (2 * j));
        newDp2[newState] += cnt;
    }
}

// 4. 同时有左插头和上插头
else {
    // 合并插头
    long long newState = (state & ~(3LL << (2 * j))) & ~(3LL << (2 * (j + 1)));
}

// 如果是形成回路的最后一步
if (left == up) {
    // 检查是否所有插头都已连接
    if (newState == 0 && i == n - 1 && j == m - 1) {
        newDp2[newState] += cnt;
    }
}

```

```

        } else {
            // 合并两个不同的插头
            newDp2[newState] += cnt;
        }
    }

dp = newDp2;
}

}

// 最终状态应该是没有任何插头 (形成回路)
auto it = dp.find(0LL);
return it != dp.end() ? it->second : 0;
}

// ===== 冷门模型: 树上背包的优化 =====

void dfsTreeKnapsack(int u, int parent, int capacity,
                      vector<vector<int>>& tree, vector<int>& weights,
                      vector<int>& values, vector<vector<int>>& dp, vector<int>& size) {
    // 初始化当前节点
    size[u] = 1;
    if (weights[u] <= capacity) {
        dp[u][weights[u]] = values[u];
    }

    // 对每个子节点, 按照子树大小排序, 小的先合并
    vector<pair<int, int>> children;
    for (int v : tree[u]) {
        if (v != parent) {
            dfsTreeKnapsack(v, u, capacity, tree, weights, values, dp, size);
            children.emplace_back(size[v], v);
        }
    }
}

// 按子树大小排序
sort(children.begin(), children.end());

for (auto& [sz, v] : children) {
    // 逆序遍历容量, 避免重复计算
    for (int i = min(size[u], capacity); i >= 0; --i) {
        if (dp[u][i] == 0 && i != 0) continue;

```

```

        for (int j = 1; j <= min(sz, capacity - i); ++j) {
            if (dp[v][j] > 0 && i + j <= capacity) {
                dp[u][i + j] = max(dp[u][i + j], dp[u][i] + dp[v][j]);
            }
        }
    }

    // 更新子树大小
    size[u] += sz;
}

int treeKnapsackOptimized(int root, int capacity, vector<vector<int>>& tree,
                           vector<int>& weights, vector<int>& values) {
/*
树上背包的优化实现（小到大合并）

```

问题描述：

在树上选择一些节点，使得总重量不超过容量，且总价值最大

解题思路：

1. 使用后序遍历处理子树
2. 使用小到大合并的策略优化复杂度
3. 对于每个节点，维护一个容量为 capacity 的背包

参数：

root: 根节点
 capacity: 背包容量
 tree: 树的邻接表
 weights: 每个节点的重量
 values: 每个节点的价值

返回：

int: 最大价值

时间复杂度：O(n*capacity^2)，但通过小到大合并可以降低常数

空间复杂度：O(n*capacity)

*/

```

int n = tree.size();
vector<vector<int>> dp(n, vector<int>(capacity + 1, 0));
vector<int> size(n, 0);

// 深度优先搜索处理子树

```

```
dfsTreeKnapsack(root, -1, capacity, tree, weights, values, dp, size);  
  
    // 返回根节点的最大价值  
    int maxVal = 0;  
    for (int val : dp[root]) {  
        maxVal = max(maxVal, val);  
    }  
    return maxVal;  
}
```

```
// ===== 补充题目与应用 =====  
// 以下是一些使用上述高级 DP 技术的经典题目及其代码实现
```

```
// 1. 编辑距离问题 (LeetCode 72)  
int editDistance(string word1, string word2) {  
    /*  
     * LeetCode 72. 编辑距离  
     * 题目链接: https://leetcode-cn.com/problems/edit-distance/  
    */  
}
```

问题描述:

给你两个单词 word1 和 word2，计算出将 word1 转换成 word2 所使用的最少操作数。
你可以对一个单词进行如下三种操作：插入一个字符、删除一个字符、替换一个字符。

解题思路:

使用二维 DP， $dp[i][j]$ 表示 word1 的前 i 个字符转换为 word2 的前 j 个字符所需的最少操作数。

时间复杂度: $O(m \times n)$

空间复杂度: $O(m \times n)$

*/

```
int m = word1.size();  
int n = word2.size();  
//  $dp[i][j]$  表示 word1 的前  $i$  个字符转换为 word2 的前  $j$  个字符所需的最少操作数  
vector<vector<int>> dp(m + 1, vector<int>(n + 1));
```

// 初始化边界

```
for (int i = 0; i <= m; ++i) {  
    dp[i][0] = i;  
}  
for (int j = 0; j <= n; ++j) {  
    dp[0][j] = j;  
}
```

// 动态规划填表

```

for (int i = 1; i <= m; ++i) {
    for (int j = 1; j <= n; ++j) {
        if (word1[i - 1] == word2[j - 1]) {
            dp[i][j] = dp[i - 1][j - 1];
        } else {
            dp[i][j] = min({dp[i - 1][j], dp[i][j - 1], dp[i - 1][j - 1]}) + 1;
        }
    }
}

return dp[m][n];
}

```

// 2. 最长递增子序列 (LeetCode 300)

```

int lengthOfLIS(vector<int>& nums) {
/*
    LeetCode 300. 最长递增子序列
    题目链接: https://leetcode-cn.com/problems/longest-increasing-subsequence/

```

问题描述:

给你一个整数数组 `nums`，找到其中最长严格递增子序列的长度。

解题思路:

使用贪心 + 二分查找优化的 DP 方法。

`tails[i]` 表示长度为 $i+1$ 的递增子序列的末尾元素的最小值。

时间复杂度: $O(n \log n)$

空间复杂度: $O(n)$

*/

```

if (nums.empty()) {
    return 0;
}

```

`vector<int> tails;`

```

for (int num : nums) {

```

// 二分查找 `num` 应该插入的位置

```

int left = 0, right = tails.size();

```

```

while (left < right) {

```

```

    int mid = left + (right - left) / 2;

```

```

    if (tails[mid] >= num) {

```

```

        right = mid;
    } else {

```

```

        left = mid + 1;
    }
}
```

```

        }
    }

    if (left == tails.size()) {
        tails.push_back(num);
    } else {
        tails[left] = num;
    }
}

return tails.size();
}

```

// 3. 背包问题变种 - 完全背包 (LeetCode 322)

```
int coinChange(vector<int>& coins, int amount) {
/*

```

LeetCode 322. 零钱兑换

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

问题描述:

给你一个整数数组 coins，表示不同面额的硬币；以及一个整数 amount，表示总金额。

计算并返回可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额，返回-1。

解题思路:

使用完全背包的思想，dp[i] 表示凑成金额 i 所需的最少硬币数。

时间复杂度: O(amount * n)

空间复杂度: O(amount)

*/

// 初始化 dp 数组为无穷大

```
vector<int> dp(amount + 1, INT_MAX);
```

```
dp[0] = 0; // 凑成金额 0 需要 0 个硬币
```

```
for (int coin : coins) {
```

```
    for (int i = coin; i <= amount; ++i) {
```

```
        if (dp[i - coin] != INT_MAX) {
```

```
            dp[i] = min(dp[i], dp[i - coin] + 1);
```

```
        }
```

```
}
```

```
}
```

```
return dp[amount] == INT_MAX ? -1 : dp[amount];
```

```
}
```

```

// 4. 矩阵链乘法（区间 DP 的经典应用）

struct MatrixChainResult {
    vector<vector<int>> dp;
    vector<vector<int>> s;

    MatrixChainResult(const vector<vector<int>>& dp, const vector<vector<int>>& s)
        : dp(dp), s(s) {}

};


```

```
MatrixChainResult matrixChainOrder(vector<int>& p) {
```

```
/*
```

矩阵链乘法问题

题目来源：算法导论

问题描述：

给定一系列矩阵，计算乘法顺序使得标量乘法的次数最少。

解题思路：

使用区间 DP， $dp[i][j]$ 表示计算第 i 到第 j 个矩阵的乘积所需的最少标量乘法次数。

可以使用 Knuth 优化进一步降低时间复杂度。

时间复杂度： $O(n^3)$

空间复杂度： $O(n^2)$

```
*/
```

```
int n = p.size() - 1; // 矩阵的个数
```

```
// dp[i][j] 表示计算第 i 到第 j 个矩阵的乘积所需的最少标量乘法次数
```

```
vector<vector<int>> dp(n + 1, vector<int>(n + 1));
```

```
// s[i][j] 记录最优分割点
```

```
vector<vector<int>> s(n + 1, vector<int>(n + 1));
```

// 枚举区间长度

```
for (int length = 2; length <= n; ++length) {
```

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

```
        int j = i + length - 1;
```

```
        dp[i][j] = INT_MAX;
```

// 枚举分割点

```
        for (int k = i; k < j; ++k) {
```

// 计算当前分割点的代价

```
            int cost = dp[i][k] + dp[k + 1][j] + p[i - 1] * p[k] * p[j];
```

```
            if (cost < dp[i][j]) {
```

```
                dp[i][j] = cost;
```

```
                s[i][j] = k;
```

```

        }
    }
}

return MatrixChainResult(dp, s);
}

```

// 5. 旅行商问题 (TSP) 的 DP 实现

```

int travelingSalesmanProblem(vector<vector<int>>& graph) {
/*
旅行商问题
题目来源：算法竞赛经典问题

```

问题描述：

给定一个完全图，找到一条访问每个城市恰好一次并返回起点的最短路径。

解题思路：

使用状态压缩 DP， $dp[mask][u]$ 表示访问过的城市集合为 mask，当前在城市 u 时的最短路径长度。

时间复杂度： $O(n^2 * 2^n)$

空间复杂度： $O(n * 2^n)$

*/

```
int n = graph.size();
```

// $dp[mask][u]$ 表示访问过的城市集合为 mask，当前在城市 u 时的最短路径长度

```
vector<vector<int>> dp(1 << n, vector<int>(n, INT_MAX));
```

// 初始状态：只访问了起点，路径长度为 0

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

// 枚举所有可能的状态

```
for (int mask = 1; mask < (1 << n); ++mask) {
```

// 枚举当前所在的城市

```
for (int u = 0; u < n; ++u) {
    if (!(mask & (1 << u))) {
        continue;
    }
```

// 枚举下一个要访问的城市

```
for (int v = 0; v < n; ++v) {
    if (mask & (1 << v)) {
        continue;
    }
```

```

        }
        int newMask = mask | (1 << v);
        if (dp[mask][u] != INT_MAX && graph[u][v] != INT_MAX) {
            dp[newMask][v] = min(dp[newMask][v], dp[mask][u] + graph[u][v]);
        }
    }
}

// 找到最短的回路
int result = INT_MAX;
for (int u = 0; u < n; ++u) {
    if (dp[(1 << n) - 1][u] != INT_MAX && graph[u][0] != INT_MAX) {
        result = min(result, dp[(1 << n) - 1][u] + graph[u][0]);
    }
}
return result;
}

```

// 6. 区间 DP: 最优三角剖分

```

int minimumScoreTriangulation(vector<int>& values) {
/*
LeetCode 1039. 多边形三角剖分的最低得分
题目链接: https://leetcode-cn.com/problems/minimum-score-triangulation-of-polygon/

```

问题描述:

给定一个凸多边形，将其三角剖分，使得所有三角形的顶点乘积之和最小。

解题思路:

使用区间 DP， $dp[i][j]$ 表示从顶点 i 到顶点 j 的多边形三角剖分的最小得分。

时间复杂度: $O(n^3)$

空间复杂度: $O(n^2)$

*/

```

int n = values.size();
// dp[i][j] 表示从顶点 i 到顶点 j 的多边形三角剖分的最小得分
vector<vector<int>> dp(n, vector<int>(n));

```

// 枚举区间长度

```

for (int length = 3; length <= n; ++length) {
    for (int i = 0; i + length - 1 < n; ++i) {
        int j = i + length - 1;
    }
}

```

```

        dp[i][j] = INT_MAX;
        // 枚举中间点
        for (int k = i + 1; k < j; ++k) {
            dp[i][j] = min(dp[i][j],
                            dp[i][k] + dp[k][j] + values[i] * values[k] * values[j]);
        }
    }

    return dp[0][n - 1];
}

```

// 7. 博弈 DP: 石子游戏

```

bool stoneGame(vector<int>& piles) {
/*
LeetCode 877. 石子游戏
题目链接: https://leetcode-cn.com/problems/stone-game/

```

问题描述:

给定一个表示石子堆的数组，两个玩家轮流从两端取石子，每次只能取一个，取到最后一个石子的人获胜。

判断先手是否必胜。

解题思路:

使用区间 DP， $dp[i][j]$ 表示在区间 $[i, j]$ 中，先手能获得的最大净胜分。

时间复杂度: $O(n^2)$

空间复杂度: $O(n^2)$

*/

```
int n = piles.size();
```

// $dp[i][j]$ 表示在区间 $[i, j]$ 中，先手能获得的最大净胜分

```
vector<vector<int>> dp(n, vector<int>(n));
```

// 初始化单个石子堆

```
for (int i = 0; i < n; ++i) {
    dp[i][i] = piles[i];
}
```

// 枚举区间长度

```
for (int length = 2; length <= n; ++length) {
    for (int i = 0; i + length - 1 < n; ++i) {
        int j = i + length - 1;
        // 先手可以选择取左边或右边
    }
}
```

```

        dp[i][j] = max(piles[i] - dp[i + 1][j], piles[j] - dp[i][j - 1]);
    }
}

// 先手净胜分大于 0 则必胜
return dp[0][n - 1] > 0;
}

```

// 8. 数位 DP: 统计 1 出现的次数

```

int countDigitOne(int n) {
/*
LeetCode 233. 数字 1 的个数
题目链接: https://leetcode-cn.com/problems/number-of-digit-one/

```

问题描述:

给定一个整数 n , 计算所有小于等于 n 的非负整数中数字 1 出现的个数。

解题思路:

使用数位 DP, 逐位处理每一位上 1 出现的次数。

时间复杂度: $O(\log n)$

空间复杂度: $O(\log n)$

*/

```

if (n <= 0) {
    return 0;
}

```

```

string s = to_string(n);
int length = s.size();
int count = 0;

```

// 逐位处理

```

for (int i = 0; i < length; ++i) {
    long long high = 0;
    if (i > 0) {
        high = stoll(s.substr(0, i));
    }
    int current = s[i] - '0';
    long long low = 0;
    if (i < length - 1) {
        low = stoll(s.substr(i + 1));
    }
    long long digit = pow(10, length - i - 1);

```

```

        if (current == 0) {
            // 当前位为 0, 高位决定
            count += high * digit;
        } else if (current == 1) {
            // 当前位为 1, 高位+低位+1
            count += high * digit + low + 1;
        } else {
            // 当前位大于 1, 高位+1
            count += (high + 1) * digit;
        }
    }

    return count;
}

// 9. 树形 DP: 打家劫舍 III

struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

pair<int, int> robDFS(TreeNode* node) {
    if (!node) {
        return {0, 0};
    }

    auto left = robDFS(node->left);
    auto right = robDFS(node->right);

    // 偷当前节点, 不能偷子节点
    int robCurrent = node->val + left.second + right.second;
    // 不偷当前节点, 可以选择偷或不偷子节点
    int notRobCurrent = max(left.first, left.second) + max(right.first, right.second);

    return {robCurrent, notRobCurrent};
}

int rob(TreeNode* root) {
/*
LeetCode 337. 打家劫舍 III

```

题目链接: <https://leetcode-cn.com/problems/house-robber-iii/>

问题描述:

在上次打劫完一条街道之后和一圈房屋后，小偷又发现了一个新的可行窃的地区。

这个地区只有一个入口，我们称之为“根”。除了“根”之外，每栋房子有且只有一个“父”房子与之相连。

一番侦察之后，聪明的小偷意识到“这个地方的所有房屋的排列类似于一棵二叉树”。

如果两个直接相连的房子在同一天晚上被打劫，房屋将自动报警。

计算在不触动警报的情况下，小偷一晚能够盗取的最高金额。

解题思路:

使用树形 DP，对于每个节点，维护两个状态：偷或不偷。

时间复杂度: $O(n)$

空间复杂度: $O(h)$ ， h 为树的高度

*/

```
auto result = robDFS(root);
return max(result.first, result.second);
}
```

// 10. 状态压缩 DP: 蒙斯特曼问题

```
int monsterGame(vector<vector<int>>& grid) {
```

/*

蒙斯特曼问题

题目来源: 算法竞赛问题

问题描述:

在网格中放置怪物，使得任何两个怪物都不在同一行、同一列或对角线上。

解题思路:

使用状态压缩 DP， $dp[i][mask]$ 表示处理到第 i 行，已放置的列的状态为 $mask$ 时的方案数。

时间复杂度: $O(n * 2^n)$

空间复杂度: $O(2^n)$

*/

```
int n = grid.size();
// dp[i][mask] 表示处理到第 i 行，已放置的列的状态为 mask 时的方案数
vector<long long> dp(1 << n, 0);
dp[0] = 1;
```

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

```
    vector<long long> newDp(1 << n, 0);
```

```
    for (int mask = 0; mask < (1 << n); ++mask) {
```

```

        if (dp[mask] == 0) {
            continue;
        }
        // 枚举所有可能的放置位置
        for (int j = 0; j < n; ++j) {
            // 检查是否可以在(i, j)放置怪物
            if (!(mask & (1 << j)) && grid[i][j] == 1) {
                // 检查对角线
                bool valid = true;
                for (int k = 0; k < i; ++k) {
                    if (mask & (1 << k) && abs(k - j) == i - k) {
                        valid = false;
                        break;
                    }
                }
                if (valid) {
                    newDp[mask | (1 << j)] += dp[mask];
                }
            }
        }
        dp = newDp;
    }

    return dp[(1 << n) - 1];
}

```

// 11. 高维 DP: 三维背包

```

int threeDimensionKnapsack(int n, vector<int>& capacity, vector<vector<int>>& items) {
/*
三维背包问题
题目来源: 算法竞赛问题

```

问题描述:

有 n 个物品，每个物品有体积、重量、价值三个属性，背包有体积和重量两个限制，求最大价值。

解题思路:

使用三维 DP， $dp[i][j][k]$ 表示前 i 个物品，体积为 j ，重量为 k 时的最大价值。

时间复杂度: $O(n * V * W)$

空间复杂度: $O(n * V * W)$

*/

```
int V = capacity[0];
```

```

int W = capacity[1];
// 初始化 dp 数组
vector<vector<vector<int>>> dp(n + 1, vector<vector<int>>(V + 1, vector<int>(W + 1, 0)));

for (int i = 1; i <= n; ++i) {
    int v = items[i-1][0];
    int w = items[i-1][1];
    int val = items[i-1][2];
    for (int j = 0; j <= V; ++j) {
        for (int k = 0; k <= W; ++k) {
            // 不选当前物品
            dp[i][j][k] = dp[i-1][j][k];
            // 选当前物品（如果有足够的空间）
            if (j >= v && k >= w) {
                dp[i][j][k] = max(dp[i][j][k], dp[i-1][j-v][k-w] + val);
            }
        }
    }
}

return dp[n][V][W];
}

```

// 12. 斜率优化 DP 示例

```

struct ConvexHullTrick {
    /*
    凸包优化技巧示例
    题目来源：算法竞赛问题

```

问题描述：

当状态转移方程形如 $dp[i] = \min\{dp[j] + a[i] * b[j] + c\}$ 时，可以使用凸包优化。

解题思路：

将转移方程转换为直线的形式，维护凸包以快速查询最小值。

时间复杂度： $O(n)$

空间复杂度： $O(n)$

*/

```

struct Line {
    long long k, b;
    Line(long long k, long long b) : k(k), b(b) {}
};

```

```

deque<Line> dq;

// 添加一条直线 y = kx + b
void addLine(long long k, long long b) {
    // 当队列中至少有两条直线时，检查是否需要删除末尾的直线
    while (dq.size() >= 2) {
        Line l1 = dq[dq.size() - 2];
        Line l2 = dq.back();
        // 判断直线 l1 和 l2 的交点是否在 l2 和新直线的交点右侧
        if ((l2.b - l1.b) * (k - l2.k) >= (b - l2.b) * (l2.k - l1.k)) {
            dq.pop_back();
        } else {
            break;
        }
    }
    dq.emplace_back(k, b);
}

// 查询 x 处的最小值
long long query(long long x) {
    // 如果队列中至少有两条直线，且第一条直线在 x 处的值大于第二条，删除第一条
    while (dq.size() >= 2) {
        Line l1 = dq[0];
        Line l2 = dq[1];
        if (l1.k * x + l1.b >= l2.k * x + l2.b) {
            dq.pop_front();
        } else {
            break;
        }
    }
    if (dq.empty()) {
        return LLONG_MAX;
    }
    Line l = dq[0];
    return l.k * x + l.b;
}
};

/***
 * 计算叉积 (a - o) × (b - o)
 */

```

```

double cross(const Point& o, const Point& a, const Point& b) {
    return (a.x - o.x) * (b.y - o.y) - (a.y - o.y) * (b.x - o.x);
}

/***
 * 计算凸包 (Andrew 算法)
 *
 * 问题描述:
 * 给定平面上的点集, 找出所有在凸包上的点。
 *
 * 解题思路:
 * 1. 将点按 x 坐标排序, x 相同按 y 排序
 * 2. 构建下凸壳和上凸壳
 * 3. 合并上下凸壳
 *
 * @param points 点集
 * @return 凸包上的点集
 */
vector<Point> convexHull(vector<Point>& points) {
    // 按 x 坐标排序, x 相同按 y 排序
    sort(points.begin(), points.end());
    int n = points.size();

    // 构建下凸壳
    vector<Point> lower;
    for (int i = 0; i < n; ++i) {
        while (lower.size() >= 2 && cross(lower[lower.size()-2], lower.back(), points[i]) <= 0) {
            lower.pop_back();
        }
        lower.push_back(points[i]);
    }

    // 构建上凸壳
    vector<Point> upper;
    for (int i = n-1; i >= 0; --i) {
        while (upper.size() >= 2 && cross(upper[upper.size()-2], upper.back(), points[i]) <= 0) {
            upper.pop_back();
        }
        upper.push_back(points[i]);
    }

    // 合并上下凸壳, 去掉重复的端点
    vector<Point> result;

```

```

for (int i = 0; i < (int)lower.size()-1; ++i) {
    result.push_back(lower[i]);
}
for (int i = 0; i < (int)upper.size()-1; ++i) {
    result.push_back(upper[i]);
}
return result;
}

/***
 * 凸包优化 DP
 *
 * 问题描述:
 * 当 DP 状态转移方程可以表示为  $dp[i] = \min\{dp[j] + a[i] * b[j]\} + c[i]$  的形式时,
 * 可以使用凸包优化将时间复杂度从  $O(n^2)$  降低到  $O(n)$  或  $O(n \log n)$ 。
 *
 * 解题思路:
 * 对于每个 j, 维护一条直线  $y = b[j] * x + dp[j]$ , 然后对于每个 i, 查询  $x = a[i]$  时的最小值。
 * 当  $b[j]$  单调递增且  $a[i]$  单调递增时, 可以使用单调队列优化。
 *
 * @param dp DP 数组
 * @param a a 数组
 * @param b b 数组
 * @return 优化后的 DP 数组
*/
vector<double> convexHullTrick(vector<double>& dp, vector<double>& a, vector<double>& b) {
    int n = dp.size();
    vector<int> q; // 单调队列, 存储直线的索引

    auto getIntersection = [&](int j1, int j2) {
        // 计算两条直线 j1 和 j2 的交点 x 坐标
        // 直线 j1:  $y = b[j1] * x + dp[j1]$ 
        // 直线 j2:  $y = b[j2] * x + dp[j2]$ 
        if (b[j1] == b[j2]) {
            return DBL_MAX;
        }
        return (dp[j2] - dp[j1]) / (b[j1] - b[j2]);
    };

    // 初始化队列, 加入第一个元素
    q.push_back(0);

    // 对于每个 i, 找到最优的 j

```

```

for (int i = 1; i < n; ++i) {
    // 当队列中至少有两个元素，且第一个元素不如第二个元素优时，弹出第一个元素
    while (q.size() >= 2 && (dp[q[0]] + a[i] * b[q[0]] >= dp[q[1]] + a[i] * b[q[1]])) {
        q.erase(q.begin());
    }

    // 使用队列中的第一个元素作为最优的 j
    dp[i] = min(dp[i], dp[q[0]] + a[i] * b[q[0]]);

    // 将当前 i 加入队列，维护队列的凸壳性质
    while (q.size() >= 2 && (getIntersection(q[q.size()-2], q.back()) >
getIntersection(q.back(), i))) {
        q.pop_back();
    }
    q.push_back(i);
}

return dp;
}

// 测试代码
int main() {
    // 测试 DP+数论
    int amount = 5;
    vector<int> coins = {1, 2, 5};
    int mod = 1e9 + 7;
    cout << "零钱兑换 II (模意义)：" << coinChangeMod(amount, coins, mod) << endl; // 应该输出 4

    // 测试矩阵快速幂
    vector<vector<long long>> matrix = {{1, 1}, {1, 0}};
    int power = 5;
    cout << "矩阵快速幂结果：" << endl;
    vector<vector<long long>> result = matrixPowerMod(matrix, power, mod);
    for (auto& row : result) {
        for (auto& num : row) {
            cout << num << " ";
        }
        cout << endl;
    }

    // 测试 DP+字符串
    string s = "bbbab";
    cout << "最长回文子序列长度：" << longestPalindromicSubseq(s) << endl; // 应该输出 4
}

```

```

// 测试后缀自动机
SuffixAutomaton sam("banana");
cout << "不同子串数量: " << sam.countSubstrings() << endl; // 应该输出 15

// 测试 DP+计算几何
vector<Point> points = {{0, 0}, {1, 1}, {2, 0}, {1, -1}};
vector<Point> hull = convexHull(points);
cout << "凸包上的点: " << endl;
for (auto& p : hull) {
    cout << "(" << p.x << ", " << p.y << ")" << endl;
}

// 测试凸包优化 DP
vector<double> dp = {0, DBL_MAX, DBL_MAX, DBL_MAX, DBL_MAX};
vector<double> a = {1, 2, 3, 4, 5};
vector<double> b = {1, 2, 3, 4, 5};
vector<double> optimizedDp = convexHullTrick(dp, a, b);
cout << "凸包优化 DP 结果: " << endl;
for (auto& num : optimizedDp) {
    cout << num << " ";
}
cout << endl;

return 0;
}

// ====== 优化体系: Knuth 优化 ======
// Knuth 优化用于优化形如  $dp[i][j] = \min\{dp[i][k] + dp[k+1][j]\} + w(i, j)$  的 DP
// 当满足四边形不等式时, 最优转移点单调
// 四边形不等式:  $w(a, b) + w(c, d) \leq w(a, d) + w(c, b)$ , 其中  $a \leq c \leq b \leq d$ 
// 单调性:  $w(b, c) \leq w(a, d)$ , 其中  $a \leq b \leq c \leq d$ 

pair<vector<vector<int>>, vector<vector<int>>> knuth_optimization(int n, function<int(int, int)> cost_func) {
    /*
    Knuth 优化的 DP 算法

```

问题描述:

解决区间 DP 问题, 其中状态转移方程满足四边形不等式

解题思路:

1. 使用 Knuth 优化将时间复杂度从 $O(n^3)$ 降低到 $O(n^2)$
2. 维护最优转移点数组 $opt[i][j]$, 表示计算 $dp[i][j]$ 时的最优 k 值
3. 根据 $opt[i][j-1] \leq opt[i][j] \leq opt[i+1][j]$ 的性质进行剪枝

参数:

n : 区间长度

$cost_func$: 计算区间 (i, j) 代价的函数

返回:

pair: (dp 数组, opt 数组)

时间复杂度: $O(n^2)$

空间复杂度: $O(n^2)$

*/

// 初始化 dp 和 opt 数组

```
vector<vector<int>> dp(n + 1, vector<int>(n + 1, 0));
vector<vector<int>> opt(n + 1, vector<int>(n + 1, 0));
```

// 初始化长度为 1 的区间

```
for (int i = 1; i <= n; ++i) {
    dp[i][i] = 0;
    opt[i][i] = i;
}
```

// 枚举区间长度

```
for (int length = 2; length <= n; ++length) {
    // 枚举起始点
    for (int i = 1; i + length - 1 <= n; ++i) {
        int j = i + length - 1;
        // 初始化为无穷大
        dp[i][j] = INT_MAX;
        // 根据 Knuth 优化的性质, 最优  $k$  在  $opt[i][j-1]$  到  $opt[i+1][j]$  之间
        for (int k = opt[i][j-1]; k <= min(opt[i+1][j], j-1); ++k) {
            if (dp[i][k] != INT_MAX && dp[k+1][j] != INT_MAX) {
                int current = dp[i][k] + dp[k+1][j] + cost_func(i, j);
                if (current < dp[i][j]) {
                    dp[i][j] = current;
                    opt[i][j] = k;
                }
            }
        }
    }
}
```

```

    return {dp, opt};
}

// ===== 优化体系: Divide & Conquer Optimization =====

vector<vector<int>> divide_conquer_optimization(int n, int m, function<int(int, int)> cost_func)
{
    /*
    Divide & Conquer Optimization (分治优化)

```

问题描述:

解决形如 $dp[i][j] = \min\{dp[i-1][k] + cost(k, j)\}$, 其中 $k < j$
当转移满足决策单调性时使用

解题思路:

1. 利用决策单调性, 使用分治法优化 DP
2. 对于 $dp[i][j]$, 当 i 固定时, 最优转移点 k 随着 j 的增加而单调不减
3. 使用分治的方式计算每个区间的最优决策

参数:

n: 维度 1
m: 维度 2
cost_func: 计算 $cost(k, j)$ 的函数

返回:

vector<vector<int>>: dp 数组

时间复杂度: $O(n*m \log m)$

空间复杂度: $O(n*m)$

*/

// 初始化 dp 数组

```

vector<vector<int>> dp(n + 1, vector<int>(m + 1, INT_MAX));
dp[0][0] = 0;

```

// 分治优化函数

```

function<void(int, int, int, int, int)> solve = [&](int i, int l, int r, int opt_l, int
opt_r) {
    /*
    计算  $dp[i][l..r]$ , 其中最优转移点在  $opt_l..opt_r$  之间
    */
    if (l > r) {
        return;
    }
}

```

```

    }

    int mid = (l + r) / 2;
    int best_k = opt_l;

    // 在 opt_l 到 min(mid-1, opt_r) 之间寻找最优 k
    for (int k = opt_l; k <= min(mid, opt_r); ++k) {
        if (dp[i-1][k] != INT_MAX && cost_func(k, mid) != INT_MAX) {
            int current = dp[i-1][k] + cost_func(k, mid);
            if (current < dp[i][mid]) {
                dp[i][mid] = current;
                best_k = k;
            }
        }
    }

    // 递归处理左右子区间
    solve(i, l, mid-1, opt_l, best_k);
    solve(i, mid+1, r, best_k, opt_r);
}

// 对每个 i 应用分治优化
for (int i = 1; i <= n; ++i) {
    solve(i, 1, m, 0, m);
}

return dp;
}

```

// ===== 优化体系: SMAWK 算法 (行最小查询) =====

```
vector<int> smawk(vector<vector<int>>& matrix) {
```

```
/*
```

```
SMAWK 算法用于在 Monge 矩阵中快速查找每行的最小值
```

问题描述:

给定一个 Monge 矩阵, 快速找到每行的最小值位置

解题思路:

1. Monge 矩阵满足性质: $\text{matrix}[i][j] + \text{matrix}[i+1][j+1] \leq \text{matrix}[i][j+1] + \text{matrix}[i+1][j]$
2. SMAWK 算法利用这一性质, 可以在 $O(m+n)$ 时间内找到每行的最小值
3. 主要步骤包括行压缩和递归求解

参数:

matrix: 一个 Monge 矩阵

返回:

vector<int>: 每行最小值的列索引

时间复杂度: $O(m+n)$, 其中 m 是行数, n 是列数

空间复杂度: $O(m+n)$

*/

```
int m = matrix.size();
int n = m > 0 ? matrix[0].size() : 0;
```

```
function<vector<int>(vector<int>&) > reduce_rows = [&] (vector<int>& rows) -> vector<int> {
    /*行压缩: 只保留可能成为最小值的行*/
    vector<int> stack;
    for (int i : rows) {
        while (stack.size() >= 2) {
            int j1 = stack[stack.size()-2];
            int j2 = stack[stack.size()-1];
            // 比较两个行在列 stack.size()-1 处的值
            if (matrix[j1][stack.size()-1] <= matrix[i][stack.size()-1]) {
                break;
            } else {
                stack.pop_back();
            }
        }
        stack.push_back(i);
    }
    return stack;
};
```

```
function<vector<int>(vector<int>&, vector<int>&) > smawk_rec = [&] (vector<int>& rows,
vector<int>& cols) -> vector<int> {
```

/*递归实现 SMAWK 算法*/

```
if (rows.empty()) {
    return {};
}
```

// 行压缩

```
vector<int> reduced_rows = reduce_rows(rows);
```

// 递归求解列数为奇数的子问题

```
vector<int> half_cols;
```

```

for (int i = 1; i < cols.size(); i += 2) {
    half_cols.push_back(cols[i]);
}
vector<int> min_cols(reduced_rows.size(), -1);

if (!half_cols.empty()) {
    // 递归求解
    vector<int> result = smawk_rec(reduced_rows, half_cols);
    // 复制结果
    for (int i = 0; i < result.size(); ++i) {
        min_cols[i] = result[i];
    }
}

// 扩展结果到所有列
vector<int> result(rows.size(), 0);
int k = 0; // min_cols 的索引

for (int i = 0; i < rows.size(); ++i) {
    int row = rows[i];
    // 确定当前行的最小值可能在哪个区间
    int start = (i == 0) ? 0 : (k > 0 ? min_cols[k-1] : 0);
    int end = (k < min_cols.size()) ? min_cols[k] : cols.back();

    // 在这个区间内查找最小值
    int min_val = INT_MAX;
    int min_col = start;

    // 注意这里 cols 是原始列的子集，需要在 cols 中遍历
    auto it_start = find(cols.begin(), cols.end(), start);
    auto it_end = find(cols.begin(), cols.end(), end);
    if (it_start != cols.end() && it_end != cols.end()) {
        for (auto it = it_start; it != next(it_end); ++it) {
            int col = *it;
            if (col < matrix[0].size() && matrix[row][col] < min_val) {
                min_val = matrix[row][col];
                min_col = col;
            }
        }
    }
}

result[i] = min_col;

```

```

// 如果当前行在 reduced_rows 中，且不是最后一行，k 前进
if (k < reduced_rows.size() && row == reduced_rows[k]) {
    k++;
}

}

return result;
};

vector<int> rows(m);
vector<int> cols(n);
for (int i = 0; i < m; ++i) rows[i] = i;
for (int i = 0; i < n; ++i) cols[i] = i;

return smawk_rec(rows, cols);
}

// ===== 优化体系: Aliens Trick (二分约束参数+可行性 DP) =====

pair<double, double> aliens_trick(function<pair<double, double>(double)> cost_func,
                                    function<bool(double)> check_func,
                                    double left, double right, double eps = 1e-7) {
/*
Aliens Trick (二分约束参数+可行性 DP)

```

问题描述:

解决带约束的优化问题，通常形如最小化总成本，同时满足某些约束条件

解题思路:

1. 将约束条件转化为参数 λ ，构造拉格朗日函数
2. 对 λ 进行二分查找，使用可行性 DP 判断当前 λ 下是否满足约束
3. 根据可行性 DP 的结果调整二分区间

参数:

cost_func: 计算带参数 λ 的成本函数

check_func: 检查当前解是否满足约束的函数

left: 二分左边界

right: 二分右边界

eps: 精度要求

返回:

pair: (最优参数 λ ，对应的最优解)

```

时间复杂度: O(log((right-left)/eps) * T(DP)), 其中 T(DP) 是一次 DP 的时间复杂度
*/
double best_lam = left;
double best_value = 0.0;

while (right - left > eps) {
    double mid = (left + right) / 2;
    // 计算当前参数下的解和约束值
    auto [current_value, constraint_value] = cost_func(mid);

    if (check_func(constraint_value)) {
        // 满足约束, 尝试更小的参数
        right = mid;
        best_lam = mid;
        best_value = current_value;
    } else {
        // 不满足约束, 需要增大参数
        left = mid;
    }
}

return {best_lam, best_value};
}

```

// ===== 图上 DP→最短路: 分层图建模 =====

```

int layered_graph_dijkstra(int n, int m, vector<tuple<int, int, int>> edges, int k) {
/*
分层图 Dijkstra 算法

```

问题描述:

给定一个图, 允许最多使用 k 次特殊操作 (如跳跃、免费通行等), 求最短路径

解题思路:

1. 构建分层图, 每层代表使用不同次数的特殊操作
2. 对于每个节点 u, 在第 i 层表示到达 u 时已经使用了 i 次特殊操作
3. 使用 Dijkstra 算法在分层图上寻找最短路径

参数:

n: 节点数量

m: 边的数量

edges: 边的列表, 每个元素为(u, v, w) 表示 u 到 v 的权为 w 的边

k: 允许使用的特殊操作次数

返回:

int: 从节点 1 到节点 n 的最短路径长度

时间复杂度: $O((n*k + m*k) \log(n*k))$

空间复杂度: $O(n*k + m*k)$

*/

// 构建分层图的邻接表

```
vector<vector<pair<int, int>>> graph(n * (k + 1));
```

// 添加普通边 (不使用特殊操作)

```
for (auto& [u, v, w] : edges) {
    for (int i = 0; i <= k; ++i) {
        graph[u + i * n].emplace_back(v + i * n, w);
    }
}
```

// 添加使用特殊操作的边 (如果允许的话)

```
for (auto& [u, v, w] : edges) {
    for (int i = 0; i < k; ++i) {
        // 这里假设特殊操作可以免费通行 (权为 0), 具体根据问题调整
        graph[u + i * n].emplace_back(v + (i + 1) * n, 0);
    }
}
```

// Dijkstra 算法

```
vector<int> dist(n * (k + 1), INT_MAX);
```

```
dist[0] = 0; // 假设起点是节点 0
```

```
priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> heap;
heap.emplace(0, 0); // (距离, 节点)
```

```
while (!heap.empty()) {
    auto [d, u] = heap.top();
    heap.pop();
```

```
    if (d > dist[u]) {
        continue;
    }
```

```
    for (auto& [v, w] : graph[u]) {
        if (dist[v] > d + w) {
            dist[v] = d + w;
            heap.emplace(dist[v], v);
```

```

        }
    }
}

// 取所有层中到达终点的最小值
int result = INT_MAX;
for (int i = 0; i <= k; ++i) {
    if (dist[n-1 + i * n] < result) {
        result = dist[n-1 + i * n];
    }
}

return result;
}

```

// ===== 冷门模型：期望 DP 遇环的方程组解（高斯消元） =====

```
vector<double> gaussian_elimination(vector<vector<double>> matrix) {
```

```
/*
```

高斯消元法求解线性方程组

问题描述：

求解形如 $Ax = b$ 的线性方程组

解题思路：

1. 构建增广矩阵
2. 进行高斯消元，将矩阵转化为行阶梯形
3. 回代求解

参数：

matrix: 增广矩阵，每行最后一个元素是 b 的值

返回：

`vector<double>`: 方程组的解

时间复杂度: $O(n^3)$

空间复杂度: $O(n^2)$

```
*/
```

```
int n = matrix.size();
const double eps = 1e-9;
```

// 高斯消元过程

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

```

// 找到主元行（当前列中绝对值最大的行）
int max_row = i;
for (int j = i; j < n; ++j) {
    if (abs(matrix[j][i]) > abs(matrix[max_row][i])) {
        max_row = j;
    }
}

// 交换主元行和当前行
swap(matrix[i], matrix[max_row]);

// 如果主元为 0， 方程组可能有无穷多解或无解
if (abs(matrix[i][i]) < eps) {
    continue;
}

// 消元过程
for (int j = i + 1; j < n; ++j) {
    double factor = matrix[j][i] / matrix[i][i];
    for (int k = i; k <= n; ++k) {
        matrix[j][k] -= factor * matrix[i][k];
    }
}

// 回代求解
vector<double> x(n, 0);
for (int i = n - 1; i >= 0; --i) {
    x[i] = matrix[i][n];
    for (int j = i + 1; j < n; ++j) {
        x[i] -= matrix[i][j] * x[j];
    }
    x[i] /= matrix[i][i];
}

return x;
}

```

```

vector<double> expectation_dp_with_cycles(int n, vector<vector<pair<int, double>>> transitions) {
/*
期望 DP 处理有环情况（使用高斯消元）

```

问题描述：

在有环的状态转移图中计算期望

解题思路：

1. 对于每个状态，建立期望方程
2. 使用高斯消元求解方程组

参数：

n: 状态数量

transitions: 转移概率列表，`transitions[i]`是一个列表，每个元素为(j, p)表示从 i 转移到 j 的概率为 p

返回：

`vector<double>`: 每个状态的期望值

时间复杂度: $O(n^3)$

空间复杂度: $O(n^2)$

*/

// 构建线性方程组的增广矩阵

```
vector<vector<double>> matrix(n, vector<double>(n + 1, 0.0));
```

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

```
    matrix[i][i] = 1.0; // 方程左边: E[i] - sum(p_ij * E[j]) = cost[i]
```

// 假设每个状态的代价为 1，具体根据问题调整

```
    double cost = 1.0;
```

```
    matrix[i][n] = cost;
```

```
    for (auto& [j, p] : transitions[i]) {
```

```
        if (i != j) { // 避免自环的特殊处理
```

```
            matrix[i][j] -= p;
```

```
}
```

```
}
```

```
}
```

// 使用高斯消元求解

```
return gaussian_elimination(matrix);
```

```
}
```

// ====== 冷门模型：插头 DP (轮廓线 DP) ======

```
int plug_dp(vector<vector<int>>& grid) {
```

```
/*
```

插头 DP (轮廓线 DP) 示例：求网格中哈密顿回路的数量

问题描述:

给定一个网格，求其中哈密顿回路的数量

解题思路:

1. 使用轮廓线 DP，记录当前处理到的位置和轮廓线状态
2. 插头表示连接的状态，通常用二进制表示
3. 使用哈希表优化空间复杂度

参数:

grid: 网格，1 表示可通行，0 表示障碍物

返回:

int: 哈密顿回路的数量

时间复杂度: $O(n*m*4^{\min(n, m)})$

空间复杂度: $O(4^{\min(n, m)})$

*/

```
int n = grid.size();
if (n == 0) {
    return 0;
}
int m = grid[0].size();

// 使用哈希表优化
unordered_map<long long, int> dp;

// 初始状态: 左上角没有插头
dp[0] = 1;

for (int i = 0; i < n; ++i) {
    // 新的一行开始，需要将状态左移一位
    unordered_map<long long, int> new_dp;
    for (auto& [state, cnt] : dp) {
        // 左移一位，移除最左边的插头
        long long new_state = state << 1;
        new_dp[new_state] += cnt;
    }
    dp = move(new_dp);
}

for (int j = 0; j < m; ++j) {
    unordered_map<long long, int> new_dp;
```

```

for (auto& [state, cnt] : dp) {
    // 当前位置左边和上边的插头状态
    int left = (state >> (2 * j)) & 3;
    int up = (state >> (2 * (j + 1))) & 3;

    // 如果当前位置是障碍物，跳过
    if (grid[i][j] == 0) {
        // 只有当左右插头都不存在时才合法
        if (left == 0 && up == 0) {
            new_dp[state] += cnt;
        }
        continue;
    }

    // 处理各种插头组合情况
    // 1. 没有左插头和上插头
    if (left == 0 && up == 0) {
        // 只能创建新的插头对（用于回路的开始）
        if (i < n - 1 && j < m - 1 && grid[i+1][j] && grid[i][j+1]) {
            long long new_state = state | (1LL << (2 * j)) | (2LL << (2 * (j + 1)));
            new_dp[new_state] += cnt;
        }
    }

    // 2. 只有左插头
    else if (left != 0 && up == 0) {
        // 向下延伸
        if (i < n - 1 && grid[i+1][j]) {
            new_dp[state] += cnt;
        }
        // 向右延伸
        if (j < m - 1 && grid[i][j+1]) {
            long long new_state = state & ~(3LL << (2 * j)) | (left << (2 * (j + 1)));
            new_dp[new_state] += cnt;
        }
    }

    // 3. 只有上插头
    else if (left == 0 && up != 0) {
        // 向右延伸
        if (j < m - 1 && grid[i][j+1]) {
            new_dp[state] += cnt;
        }
    }
}

```

```

    }

    // 向下延伸
    if (i < n - 1 && grid[i+1][j]) {
        long long new_state = state & ~(3LL << (2 * (j + 1))) | (up << (2 * j));
        new_dp[new_state] += cnt;
    }
}

// 4. 同时有左插头和上插头
else {
    // 合并插头
    long long new_state = state & ~(3LL << (2 * j)) & ~(3LL << (2 * (j + 1)));

    // 如果是形成回路的最后一步
    if (left == up) {
        // 检查是否所有插头都已连接
        if (new_state == 0 && i == n - 1 && j == m - 1) {
            new_dp[new_state] += cnt;
        }
    } else {
        // 合并两个不同的插头
        new_dp[new_state] += cnt;
    }
}
}

dp = move(new_dp);
}

}

// 最终状态应该是没有任何插头（形成回路）
return dp.count(0) ? dp[0] : 0;
}

// ===== 冷门模型：树上背包的优化 =====

```

```

int tree_knapsack_optimized(int root, int capacity, vector<vector<int>>& tree, vector<int>& weights, vector<int>& values) {
/*
树上背包的优化实现（小到大合并）

```

问题描述：

在树上选择一些节点，使得总重量不超过容量，且总价值最大

解题思路：

1. 使用后序遍历处理子树
2. 使用小到大合并的策略优化复杂度
3. 对于每个节点，维护一个容量为 capacity 的背包

参数：

root：根节点
capacity：背包容量
tree：树的邻接表
weights：每个节点的重量
values：每个节点的价值

返回：

int：最大价值

时间复杂度： $O(n * capacity^2)$ ，但通过小到大合并可以降低常数

空间复杂度： $O(n * capacity)$

```
/*
int n = tree.size();
vector<vector<int>> dp(n, vector<int>(capacity + 1, 0));
vector<int> size(n, 0);

function<void(int, int)> dfs = [&](int u, int parent) {
    // 初始化当前节点
    size[u] = 1;
    if (weights[u] <= capacity) {
        dp[u][weights[u]] = values[u];
    }

    // 对每个子节点，按照子树大小排序，小的先合并
    vector<pair<int, int>> children;
    for (int v : tree[u]) {
        if (v != parent) {
            dfs(v, u);
            children.emplace_back(size[v], v);
        }
    }

    // 按子树大小排序
    sort(children.begin(), children.end());

    for (auto& [sz, v] : children) {
```

```

// 逆序遍历容量，避免重复计算
for (int i = min(size[u], capacity); i >= 0; --i) {
    if (dp[u][i] == 0 && i != 0) continue;
    for (int j = 1; j <= min(sz, capacity - i); ++j) {
        if (dp[v][j] > 0 && i + j <= capacity) {
            dp[u][i + j] = max(dp[u][i + j], dp[u][i] + dp[v][j]);
        }
    }
}

// 更新子树大小
size[u] += sz;
}

};

dfs(root, -1);

// 返回根节点的最大价值
return *max_element(dp[root].begin(), dp[root].end());
}

```

// ===== 补充题目与应用 =====
// 以下是一些使用上述高级 DP 技术的经典题目及其代码实现

```

// 1. 编辑距离问题 (LeetCode 72)
int edit_distance(string word1, string word2) {
/*
LeetCode 72. 编辑距离
题目链接: https://leetcode-cn.com/problems/edit-distance/

```

问题描述:

给你两个单词 word1 和 word2，计算出将 word1 转换成 word2 所使用的最少操作数。

你可以对一个单词进行如下三种操作：插入一个字符、删除一个字符、替换一个字符。

解题思路:

使用二维 DP， $dp[i][j]$ 表示 word1 的前 i 个字符转换为 word2 的前 j 个字符所需的最少操作数。

时间复杂度: $O(m*n)$

空间复杂度: $O(m*n)$

*/

int m = word1.size();

int n = word2.size();

// $dp[i][j]$ 表示 word1 的前 i 个字符转换为 word2 的前 j 个字符所需的最少操作数

```

vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));

// 初始化边界
for (int i = 0; i <= m; ++i) {
    dp[i][0] = i;
}
for (int j = 0; j <= n; ++j) {
    dp[0][j] = j;
}

// 动态规划填表
for (int i = 1; i <= m; ++i) {
    for (int j = 1; j <= n; ++j) {
        if (word1[i-1] == word2[j-1]) {
            dp[i][j] = dp[i-1][j-1];
        } else {
            dp[i][j] = min({dp[i-1][j], dp[i][j-1], dp[i-1][j-1]}) + 1;
        }
    }
}
return dp[m][n];
}

```

// 2. 最长递增子序列 (LeetCode 300)

```

int length_of_lis(vector<int>& nums) {
/*
LeetCode 300. 最长递增子序列
题目链接: https://leetcode-cn.com/problems/longest-increasing-subsequence/

```

问题描述:

给你一个整数数组 `nums`，找到其中最长严格递增子序列的长度。

解题思路:

使用贪心 + 二分查找优化的 DP 方法。

`tails[i]` 表示长度为 $i+1$ 的递增子序列的末尾元素的最小值。

时间复杂度: $O(n \log n)$

空间复杂度: $O(n)$

*/

```

if (nums.empty()) {
    return 0;
}

```

```

vector<int> tails;
for (int num : nums) {
    // 二分查找 num 应该插入的位置
    int left = 0, right = tails.size();
    while (left < right) {
        int mid = (left + right) / 2;
        if (tails[mid] < num) {
            left = mid + 1;
        } else {
            right = mid;
        }
    }

    if (left == tails.size()) {
        tails.push_back(num);
    } else {
        tails[left] = num;
    }
}

return tails.size();
}

```

// 3. 背包问题变种 - 完全背包 (LeetCode 322)

```

int coin_change(vector<int>& coins, int amount) {
    /*
    LeetCode 322. 零钱兑换
    题目链接: https://leetcode-cn.com/problems/coin-change/

```

问题描述:

给你一个整数数组 coins，表示不同面额的硬币；以及一个整数 amount，表示总金额。

计算并返回可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额，返回-1。

解题思路:

使用完全背包的思想，dp[i]表示凑成金额 i 所需的最少硬币数。

时间复杂度: O(amount * n)

空间复杂度: O(amount)

*/

// 初始化 dp 数组为无穷大

```

vector<int> dp(amount + 1, INT_MAX);
dp[0] = 0; // 凑成金额 0 需要 0 个硬币

```

```

for (int coin : coins) {
    for (int i = coin; i <= amount; ++i) {
        if (dp[i - coin] != INT_MAX) {
            dp[i] = min(dp[i], dp[i - coin] + 1);
        }
    }
}

return dp[amount] == INT_MAX ? -1 : dp[amount];
}

```

// 4. 矩阵链乘法（区间 DP 的经典应用）

```

pair<vector<vector<int>>, vector<vector<int>>> matrix_chain_order(vector<int>& p) {
/*

```

矩阵链乘法问题

题目来源：算法导论

问题描述：

给定一系列矩阵，计算乘法顺序使得标量乘法的次数最少。

解题思路：

使用区间 DP， $dp[i][j]$ 表示计算第 i 到第 j 个矩阵的乘积所需的最少标量乘法次数。

可以使用 Knuth 优化进一步降低时间复杂度。

时间复杂度： $O(n^3)$

空间复杂度： $O(n^2)$

*/

```
int n = p.size() - 1; // 矩阵的个数
```

// $dp[i][j]$ 表示计算第 i 到第 j 个矩阵的乘积所需的最少标量乘法次数

```
vector<vector<int>> dp(n + 1, vector<int>(n + 1, 0));
```

// $s[i][j]$ 记录最优分割点

```
vector<vector<int>> s(n + 1, vector<int>(n + 1, 0));
```

// 枚举区间长度

```
for (int length = 2; length <= n; ++length) {
```

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

```
        int j = i + length - 1;
```

```
        dp[i][j] = INT_MAX;
```

// 枚举分割点

```
        for (int k = i; k < j; ++k) {
```

// 计算当前分割点的代价

```
            int cost = dp[i][k] + dp[k+1][j] + p[i-1] * p[k] * p[j];
```

```

        if (cost < dp[i][j]) {
            dp[i][j] = cost;
            s[i][j] = k;
        }
    }
}

return {dp, s};
}

```

// 5. 旅行商问题 (TSP) 的 DP 实现

```

int traveling_salesman_problem(vector<vector<int>>& graph) {
/*
旅行商问题
题目来源：算法竞赛经典问题

```

问题描述：

给定一个完全图，找到一条访问每个城市恰好一次并返回起点的最短路径。

解题思路：

使用状态压缩 DP， $dp[mask][u]$ 表示访问过的城市集合为 $mask$ ，当前在城市 u 时的最短路径长度。

时间复杂度： $O(n^2 * 2^n)$

空间复杂度： $O(n * 2^n)$

*/

```
int n = graph.size();
```

// $dp[mask][u]$ 表示访问过的城市集合为 $mask$ ，当前在城市 u 时的最短路径长度

```
vector<vector<int>> dp(1 << n, vector<int>(n, INT_MAX));
```

// 初始状态：只访问了起点，路径长度为 0

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

// 枚举所有可能的状态

```
for (int mask = 1; mask < (1 << n); ++mask) {
```

// 枚举当前所在的城市

```
for (int u = 0; u < n; ++u) {
    if (!(mask & (1 << u))) {
        continue;
    }
}
```

// 枚举下一个要访问的城市

```

        for (int v = 0; v < n; ++v) {
            if (mask & (1 << v)) {
                continue;
            }
            int new_mask = mask | (1 << v);
            if (dp[mask][u] != INT_MAX && graph[u][v] != INT_MAX) {
                dp[new_mask][v] = min(dp[new_mask][v], dp[mask][u] + graph[u][v]);
            }
        }
    }

// 找到最短的回路
int result = INT_MAX;
for (int u = 0; u < n; ++u) {
    if (dp[(1 << n) - 1][u] != INT_MAX && graph[u][0] != INT_MAX) {
        result = min(result, dp[(1 << n) - 1][u] + graph[u][0]);
    }
}

return result;
}

```

// 6. 区间 DP：最优三角剖分

```

int minimum_score_triangulation(vector<int>& values) {
/*
LeetCode 1039. 多边形三角剖分的最低得分
题目链接: https://leetcode-cn.com/problems/minimum-score-triangulation-of-polygon/

```

问题描述：

给定一个凸多边形，将其三角剖分，使得所有三角形的顶点乘积之和最小。

解题思路：

使用区间 DP， $dp[i][j]$ 表示从顶点 i 到顶点 j 的多边形三角剖分的最小得分。

时间复杂度： $O(n^3)$

空间复杂度： $O(n^2)$

*/

```

int n = values.size();
// dp[i][j] 表示从顶点 i 到顶点 j 的多边形三角剖分的最小得分
vector<vector<int>> dp(n, vector<int>(n, 0));

```

// 枚举区间长度

```

for (int length = 3; length <= n; ++length) {
    for (int i = 0; i + length - 1 < n; ++i) {
        int j = i + length - 1;
        dp[i][j] = INT_MAX;
        // 枚举中间点
        for (int k = i + 1; k < j; ++k) {
            dp[i][j] = min(dp[i][j], dp[i][k] + dp[k][j] + values[i] * values[k] *
values[j]);
        }
    }
}

return dp[0][n-1];
}

```

// 7. 博弈 DP: 石子游戏

```

bool stone_game(vector<int>& piles) {
/*
LeetCode 877. 石子游戏
题目链接: https://leetcode-cn.com/problems/stone-game/

```

问题描述:

给定一个表示石子堆的数组，两个玩家轮流从两端取石子，每次只能取一个，取到最后一个石子的人获胜。

判断先手是否必胜。

解题思路:

使用区间 DP， $dp[i][j]$ 表示在区间 $[i, j]$ 中，先手能获得的最大净胜分。

时间复杂度: $O(n^2)$

空间复杂度: $O(n^2)$

*/

```

int n = piles.size();
// dp[i][j] 表示在区间 [i, j] 中，先手能获得的最大净胜分
vector<vector<int>> dp(n, vector<int>(n, 0));

```

// 初始化单个石子堆

```

for (int i = 0; i < n; ++i) {
    dp[i][i] = piles[i];
}

```

// 枚举区间长度

```

for (int length = 2; length <= n; ++length) {

```

```

        for (int i = 0; i + length - 1 < n; ++i) {
            int j = i + length - 1;
            // 先手可以选择取左边或右边
            dp[i][j] = max(piles[i] - dp[i+1][j], piles[j] - dp[i][j-1]);
        }
    }

// 先手净胜分大于 0 则必胜
return dp[0][n-1] > 0;
}

```

// 8. 数位 DP: 统计 1 出现的次数

```

int count_digit_one(int n) {
/*
LeetCode 233. 数字 1 的个数
题目链接: https://leetcode-cn.com/problems/number-of-digit-one/

```

问题描述:

给定一个整数 n , 计算所有小于等于 n 的非负整数中数字 1 出现的个数。

解题思路:

使用数位 DP, 逐位处理每一位上 1 出现的次数。

时间复杂度: $O(\log n)$

空间复杂度: $O(\log n)$

*/

```

if (n <= 0) {
    return 0;
}
```

```
string s = to_string(n);
```

```
int length = s.size();
```

```
int count = 0;
```

// 逐位处理

```

for (int i = 0; i < length; ++i) {
    long long high = 0;
    if (i > 0) {
        high = stoll(s.substr(0, i));
    }
    int current = s[i] - '0';
    long long low = 0;
    if (i < length - 1) {
```

```

        low = stoll(s.substr(i+1));
    }

    long long digit = pow(10, length - i - 1);

    if (current == 0) {
        // 当前位为 0, 高位决定
        count += high * digit;
    } else if (current == 1) {
        // 当前位为 1, 高位+低位+1
        count += high * digit + low + 1;
    } else {
        // 当前位大于 1, 高位+1
        count += (high + 1) * digit;
    }
}

return count;
}

// 9. 树形 DP: 打家劫舍 III

struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

pair<int, int> rob_dfs(TreeNode* node) {
    if (!node) {
        return {0, 0};
    }

    auto [left_rob, left_not_rob] = rob_dfs(node->left);
    auto [right_rob, right_not_rob] = rob_dfs(node->right);

    // 偷当前节点, 不能偷子节点
    int rob_current = node->val + left_not_rob + right_not_rob;
    // 不偷当前节点, 可以选择偷或不偷子节点
    int not_rob_current = max(left_rob, left_not_rob) + max(right_rob, right_not_rob);

    return {rob_current, not_rob_current};
}

```

```
int rob(TreeNode* root) {  
/*  
LeetCode 337. 打家劫舍 III  
题目链接: https://leetcode-cn.com/problems/house-robber-iii/
```

问题描述:

在上次打劫完一条街道之后和一圈房屋后，小偷又发现了一个新的可行窃的地区。

这个地区只有一个入口，我们称之为“根”。除了“根”之外，每栋房子有且只有一个“父”房子与之相连。

一番侦察之后，聪明的小偷意识到“这个地方的所有房屋的排列类似于一棵二叉树”。

如果两个直接相连的房子在同一天晚上被打劫，房屋将自动报警。

计算在不触动警报的情况下，小偷一晚能够盗取的最高金额。

解题思路:

使用树形 DP，对于每个节点，维护两个状态：偷或不偷。

时间复杂度: $O(n)$

空间复杂度: $O(h)$ ， h 为树的高度

*/

```
auto [rob_root, not_rob_root] = rob_dfs(root);
```

```
return max(rob_root, not_rob_root);
```

```
}
```

// 10. 状态压缩 DP: 蒙斯特曼问题

```
int monster_game(vector<vector<int>>& grid) {
```

/*

蒙斯特曼问题

题目来源: 算法竞赛问题

问题描述:

在网格中放置怪物，使得任何两个怪物都不在同一行、同一列或对角线上。

解题思路:

使用状态压缩 DP， $dp[i][mask]$ 表示处理到第 i 行，已放置的列的状态为 $mask$ 时的方案数。

时间复杂度: $O(n * 2^n)$

空间复杂度: $O(2^n)$

*/

```
int n = grid.size();
```

// $dp[i][mask]$ 表示处理到第 i 行，已放置的列的状态为 $mask$ 时的方案数

```
vector<long long> dp(1 << n, 0);
```

```
dp[0] = 1;
```

```

for (int i = 0; i < n; ++i) {
    vector<long long> new_dp(1 << n, 0);
    for (int mask = 0; mask < (1 << n); ++mask) {
        if (dp[mask] == 0) {
            continue;
        }
        // 枚举所有可能的放置位置
        for (int j = 0; j < n; ++j) {
            // 检查是否可以在(i, j)放置怪物
            if (!(mask & (1 << j)) && grid[i][j] == 1) {
                // 检查对角线
                bool valid = true;
                for (int k = 0; k < i; ++k) {
                    if (mask & (1 << k) && abs(k - j) == i - k) {
                        valid = false;
                        break;
                    }
                }
                if (valid) {
                    new_dp[mask | (1 << j)] += dp[mask];
                }
            }
        }
        dp = move(new_dp);
    }
}

return dp[(1 << n) - 1];
}

```

// 11. 高维 DP: 三维背包

```

int three_dimension_knapsack(int n, pair<int, int> capacity, vector<tuple<int, int, int>> items)
{
    /*

```

三维背包问题

题目来源: 算法竞赛问题

问题描述:

有 n 个物品，每个物品有体积、重量、价值三个属性，背包有体积和重量两个限制，求最大价值。

解题思路:

使用三维 DP， $dp[i][j][k]$ 表示前 i 个物品，体积为 j，重量为 k 时的最大价值。

```

时间复杂度: O(n * V * W)
空间复杂度: O(n * V * W)
*/
int V = capacity.first;
int W = capacity.second;
// 初始化 dp 数组
vector<vector<vector<int>>> dp(n + 1, vector<vector<int>>(V + 1, vector<int>(W + 1, 0)));

for (int i = 1; i <= n; ++i) {
    auto [v, w, val] = items[i-1];
    for (int j = 0; j <= V; ++j) {
        for (int k = 0; k <= W; ++k) {
            // 不选当前物品
            dp[i][j][k] = dp[i-1][j][k];
            // 选当前物品 (如果有足够的空间)
            if (j >= v && k >= w) {
                dp[i][j][k] = max(dp[i][j][k], dp[i-1][j-v][k-w] + val);
            }
        }
    }
}

return dp[n][V][W];
}

```

// 12. 斜率优化 DP 示例

```

class ConvexHullTrick {
public:
    /*
    凸包优化技巧示例
    题目来源: 算法竞赛问题

```

问题描述:

当状态转移方程形如 $dp[i] = \min\{dp[j] + a[i] * b[j] + c\}$ 时, 可以使用凸包优化。

解题思路:

将转移方程转换为直线的形式, 维护凸包以快速查询最小值。

时间复杂度: O(n)

空间复杂度: O(n)

*/

// 添加一条直线 $y = kx + b$

```

void add_line(long long k, long long b) {
    // 当队列中至少有两条直线时，检查是否需要删除末尾的直线
    while (dq.size() >= 2) {
        auto [k1, b1] = dq[dq.size() - 2];
        auto [k2, b2] = dq[dq.size() - 1];
        // 判断直线 k1x+b1 和 k2x+b2 的交点是否在 k2x+b2 和 kx+b 的交点右侧
        if ((b2 - b1) * (k - k2) >= (b - b2) * (k2 - k1)) {
            dq.pop_back();
        } else {
            break;
        }
    }
    dq.emplace_back(k, b);
}

// 查询 x 处的最小值
long long query(long long x) {
    // 如果队列中至少有两条直线，且第一条直线在 x 处的值大于第二条，删除第一条
    while (dq.size() >= 2) {
        auto [k1, b1] = dq[0];
        auto [k2, b2] = dq[1];
        if (k1 * x + b1 >= k2 * x + b2) {
            dq.pop_front();
        } else {
            break;
        }
    }
    if (dq.empty()) {
        return LLONG_MAX;
    }
    auto [k, b] = dq[0];
    return k * x + b;
}
}

private:
deque<pair<long long, long long>> dq;
};

=====

文件: dp_fusion.py
=====

# -*- coding: utf-8 -*-

```

"""

高级动态规划算法集合

本文件实现了多种高级动态规划算法及其优化技术，包括：

1. 基础 DP 融合 (DP+数论、DP+字符串、DP+计算几何)
2. 分层 DP (多阶段问题分层)
3. 容斥 DP (结合数论容斥)
4. 记忆化搜索 (高维状态+剪枝)
5. 组合 DP (生成函数+DP)
6. 博弈 DP (SG 函数、阶梯 Nim)
7. 高维压缩 (子集/超集枚举与预处理、子集/超集卷积)
8. 优化体系 (Knuth 优化、Divide & Conquer Optimization、SMAWK、Aliens Trick)
9. 图上 DP→最短路 (分层图建模)
10. 冷门模型 (期望 DP、插头 DP、树上背包)

本实现涵盖了从入门到竞赛级别的各类动态规划技术，并提供了详细的注释和复杂度分析。

"""

```
import sys
import math

# ===== DP+数论 (模意义) =====

def coin_change_mod(coins, amount, mod=10**9+7):
    """
    LeetCode 518. 零钱兑换 II (模意义下的变种)
    题目链接: https://leetcode-cn.com/problems/coin-change-2/
    
```

问题描述：

给定不同面额的硬币和一个总金额。计算可以凑成总金额的硬币组合数。

假设每一种面额的硬币有无限个。要求结果对给定的模数取余。

解题思路：

使用动态规划，定义 $dp[i]$ 表示凑成金额 i 的组合数。

状态转移方程： $dp[i] = (dp[i] + dp[i-coin]) \% \text{mod}$ ，其中 coin 是硬币面额。

Args:

coins: 硬币面额数组

amount: 总金额

mod: 模数

Returns:

int: 可以凑成总金额的硬币组合数对 mod 取余的结果

```

"""
# dp[i] 表示凑成金额 i 的组合数
dp = [0] * (amount + 1)
dp[0] = 1 # 凑成金额 0 的方式只有 1 种 (不选任何硬币)

# 遍历每种硬币
for coin in coins:
    # 遍历金额, 从 coin 开始
    for i in range(coin, amount + 1):
        dp[i] = (dp[i] + dp[i - coin]) % mod

return dp[amount]

```

```
def matrix_power_mod(matrix, power, mod):
```

```
"""

```

矩阵快速幂 (模意义下)

问题描述:

计算矩阵的幂，结果对 mod 取余。

解题思路:

使用快速幂算法，将矩阵的乘法在模意义下进行。

Args:

- matrix: 输入矩阵
- power: 幂次
- mod: 模数

Returns:

- list: 矩阵的幂对 mod 取余的结果

```
"""

```

```
n = len(matrix)
```

```
# 初始化结果为单位矩阵
```

```
result = [[1 if i == j else 0 for j in range(n)] for i in range(n)]
```

快速幂算法

```
while power > 0:
```

```
    if power % 2 == 1:
```

```
        # 矩阵乘法
```

```
        result = matrix_multiply(result, matrix, mod)
```

```
        # 矩阵自乘
```

```
        matrix = matrix_multiply(matrix, matrix, mod)
```

```
power //= 2

return result
```

```
def matrix_multiply(a, b, mod):
```

```
    """
```

```
    矩阵乘法（模意义下）
```

```
Args:
```

```
    a: 矩阵 a
```

```
    b: 矩阵 b
```

```
    mod: 模数
```

```
Returns:
```

```
    list: 矩阵 a 和矩阵 b 的乘积对 mod 取余的结果
```

```
    """
```

```
n = len(a)
```

```
m = len(b[0])
```

```
k = len(b)
```

```
result = [[0 for _ in range(m)] for _ in range(n)]
```

```
for i in range(n):
```

```
    for j in range(m):
```

```
        for p in range(k):
```

```
            result[i][j] = (result[i][j] + a[i][p] * b[p][j]) % mod
```

```
return result
```

```
# ===== 分层 DP (多阶段问题分层) =====
```

```
def layered_dp(matrix):
```

```
    """
```

```
分层 DP 示例: LeetCode 120. 三角形最小路径和的分层解法
```

```
题目链接: https://leetcode-cn.com/problems/triangle/
```

问题描述:

给定一个三角形 triangle，找出自顶向下的最小路径和。

每一步只能移动到下一行中相邻的结点上。相邻的结点 在这里指的是 下标 与 上一层结点下标 相同或者等于 上一层结点下标 + 1 的两个结点。

解题思路:

使用分层 DP，每层维护当前层的最优解。

定义 $dp[i][j]$ 表示到达第 i 行第 j 列的最小路径和。

状态转移方程: $dp[i][j] = \min(dp[i-1][j-1], dp[i-1][j]) + triangle[i][j]$

Args:

matrix: 三角形矩阵

Returns:

int: 最小路径和

时间复杂度: $O(n^2)$, 其中 n 是三角形的行数

空间复杂度: $O(n)$, 使用滚动数组优化空间

"""

```
n = len(matrix)
# 使用滚动数组优化空间
dp = [float('inf')] * n
dp[0] = matrix[0][0] # 初始状态

for i in range(1, n):
    # 从右往左更新, 避免覆盖上一层的值
    for j in range(i, -1, -1):
        if j == 0:
            # 最左边的元素只能从正上方来
            dp[j] = dp[j] + matrix[i][j]
        elif j == i:
            # 最右边的元素只能从左上方来
            dp[j] = dp[j-1] + matrix[i][j]
        else:
            # 中间元素可以从正上方或左上方来
            dp[j] = min(dp[j], dp[j-1]) + matrix[i][j]

return min(dp[:n])
```

```
def multi_layered_dp(grid):
    """
```

多层 DP 示例: LeetCode 62. 不同路径

题目链接: <https://leetcode-cn.com/problems/unique-paths/>

问题描述:

一个机器人位于一个 $m \times n$ 网格的左上角 (起始点在下图中标记为 “Start”)。

机器人每次只能向下或者向右移动一步。机器人试图达到网格的右下角 (在下图中标记为 “Finish”)。

问总共有多少条不同的路径?

解题思路:

使用二维 DP, 定义 $dp[i][j]$ 表示到达位置 (i, j) 的路径数。

状态转移方程: $dp[i][j] = dp[i-1][j] + dp[i][j-1]$

Args:

grid: m x n 的网格

Returns:

int: 不同路径的数量

时间复杂度: $O(m \times n)$

空间复杂度: $O(m \times n)$

"""

```
if not grid or not grid[0]:  
    return 0
```

m, n = len(grid), len(grid[0])

$dp[i][j]$ 表示到达位置(i, j)的路径数

```
dp = [[0] * n for _ in range(m)]
```

初始化第一行和第一列

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

```
    dp[i][0] = 1
```

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

```
    dp[0][j] = 1
```

动态规划填表

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

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

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

```
return dp[m-1][n-1]
```

===== 容斥 DP (结合数论容斥) =====

```
def inclusion_exclusion_dp(n, m, mod=10**9+7):
```

"""

容斥 DP 示例: 计算 n 个元素分成 m 个非空集合的方式数 (斯特林数的第二种形式)

LeetCode 类似题目: LeetCode 1735. 生成乘积数组的方案数

题目链接: <https://leetcode-cn.com/problems/count-ways-to-make-array-with-product/>

解题思路:

使用容斥原理结合动态规划。

$dp[i][j]$ 表示将 i 个元素分成 j 个非空集合的方式数。

状态转移方程: $dp[i][j] = dp[i-1][j-1] + j * dp[i-1][j]$

Args:

n: 元素个数
m: 集合个数
mod: 模数

Returns:

int: 方案数对 mod 取余的结果

时间复杂度: $O(n*m)$

空间复杂度: $O(n*m)$

"""

```
# dp[i][j]表示将 i 个元素分成 j 个非空集合的方式数
dp = [[0] * (m + 1) for _ in range(n + 1)]
```

```
# 初始状态: 将 i 个元素分成 i 个非空集合只有 1 种方式 (每个元素自成一个集合)
```

```
# 只设置到 min(n, m) 的情况, 避免索引越界
```

```
for i in range(min(n, m) + 1):
```

```
    dp[i][i] = 1
```

```
# 动态规划填表
```

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

```
    for j in range(1, min(i, m) + 1):
```

```
        if i != j:
```

```
            # 第 i 个元素要么单独放在一个新集合中 (dp[i-1][j-1])
```

```
            # 要么放在已有的 j 个集合中的任意一个 (j * dp[i-1][j])
```

```
            dp[i][j] = (dp[i-1][j-1] + j * dp[i-1][j]) % mod
```

```
return dp[n][m]
```

```
def mobius_inversion_dp(nums):
```

"""

莫比乌斯反演 DP 示例

问题描述: 给定一个数组, 计算有多少对元素互质。

解题思路:

使用容斥原理和莫比乌斯函数进行计数。

1. 统计每个数的出现次数
2. 计算每个数的倍数的总出现次数
3. 使用莫比乌斯函数进行容斥

Args:

nums: 输入数组

Returns:

int: 互质对的数量

时间复杂度: $O(\max_num * \log(\max_num))$, 其中 \max_num 是数组中的最大值

空间复杂度: $O(\max_num)$

"""

if not nums:

return 0

max_num = max(nums)

统计每个数的出现次数

cnt = [0] * (max_num + 1)

for num in nums:

cnt[num] += 1

计算每个数的倍数的总出现次数

f = [0] * (max_num + 1)

for i in range(1, max_num + 1):

for j in range(i, max_num + 1, i):

f[i] += cnt[j]

预处理莫比乌斯函数

mu = [1] * (max_num + 1)

is_prime = [True] * (max_num + 1)

primes = []

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

if is_prime[i]:

primes.append(i)

mu[i] = -1

for p in primes:

if i * p > max_num:

break

is_prime[i * p] = False

if i % p == 0:

mu[i * p] = 0

break

mu[i * p] = -mu[i]

使用莫比乌斯反演计算互质对的数量

result = 0

for d in range(1, max_num + 1):

```
result += mu[d] * f[d] * (f[d] - 1) // 2

return result

# ====== 记忆化搜索（高维状态+剪枝）=====

import functools

def memoization_search_example(n, k):
    """
    记忆化搜索示例：LeetCode 377. 组合总和 IV
    题目链接：https://leetcode-cn.com/problems/combination-sum-iv/
    """
```

问题描述：

给你一个由 不同 整数组成的数组 `nums`，和一个目标整数 `target`。

请你从 `nums` 中找出并返回总和为 `target` 的元素组合的个数。

题目数据保证答案符合 32 位整数范围。

（这里使用 `n` 和 `k` 作为参数，相当于 `nums=[1..n]`, `target=k`）

解题思路：

使用记忆化搜索，缓存中间结果避免重复计算。

Args:

`n`: 可选数字的最大值 ($1 \sim n$)

`k`: 目标和

Returns:

 int: 组合数

时间复杂度: $O(n*k)$

空间复杂度: $O(k)$

"""

```
@functools.lru_cache(maxsize=None)
```

```
def dfs(remain):
```

 # 基本情况

 if remain == 0:

 return 1

 if remain < 0:

 return 0

 # 尝试每种数字

 res = 0

```
  for num in range(1, n + 1):
```

```

        res += dfs(remain - num)

    return res

return dfs(k)

def pruning_memoization_search(grid):
    """
剪枝优化的记忆化搜索: LeetCode 1301. 最大得分的路径数目
题目链接: https://leetcode-cn.com/problems/number-of-paths-with-max-score/
    """

    剪枝优化的记忆化搜索: LeetCode 1301. 最大得分的路径数目
    题目链接: https://leetcode-cn.com/problems/number-of-paths-with-max-score/

```

问题描述:

给你一个正方形字符数组 board , 你从数组的 左上角 开始出发。
 每一步可以移动到四个方向之一，但不能越出边界，也不能移动到 'X' 上。
 到达右下角时，你的路径得分是路径上所有数字的总和。
 返回一个列表，其中第一个元素是最大可能的得分，第二个元素是得到最大得分的路径数目。
 路径数目需要对 $10^9 + 7$ 取模。

解题思路:

使用记忆化搜索，同时缓存最大得分和对应的路径数目，进行剪枝优化。

Args:

grid: 字符数组

Returns:

list: [最大得分, 路径数目]

时间复杂度: $O(n^2)$

空间复杂度: $O(n^2)$

"""

```

if not grid or not grid[0]:
    return [0, 0]

```

n = len(grid)

mod = $10^{**}9 + 7$

```

# 缓存结果: (max_score, path_count)
memo = {}

```

def dfs(i, j):

```

    if (i, j) in memo:
        return memo[(i, j)]

```

```

# 到达终点
if i == 0 and j == 0:
    return [0, 1] if grid[i][j] == 'E' or grid[i][j] == 'S' else [int(grid[i][j]), 1]

# 越界或遇到障碍
if i < 0 or j < 0 or grid[i][j] == 'X':
    return [-1, 0] # -1 表示不可达

max_score = -1
path_count = 0

# 尝试从上方和左方过来
for di, dj in [(-1, 0), (0, -1)]:
    ni, nj = i + di, j + dj
    score, count = dfs(ni, nj)

    if score == -1:
        continue

    current_score = score + (int(grid[i][j]) if grid[i][j] not in ['S', 'E'] else 0)

    if current_score > max_score:
        max_score = current_score
        path_count = count
    elif current_score == max_score:
        path_count = (path_count + count) % mod

memo[(i, j)] = [max_score, path_count]
return [max_score, path_count]

```

```

# 从右下角开始搜索
# 注意：根据题目描述，需要调整起点和终点的位置
score, count = dfs(n-1, n-1)
return [score, count] if score != -1 else [0, 0]

```

```
# ===== DP+字符串 (SAM 相关) =====
```

```
def longest_palindromic_subseq(s):
    """

```

LeetCode 516. 最长回文子序列

题目链接: <https://leetcode-cn.com/problems/longest-palindromic-subsequence/>

问题描述:

给定一个字符串 s，找到其中最长的回文子序列。可以假设 s 的最大长度为 1000。

解题思路：

使用区间 DP，定义 $dp[i][j]$ 表示字符串 s 在区间 $[i, j]$ 内的最长回文子序列的长度。

状态转移方程：

- 如果 $s[i] == s[j]$ ，则 $dp[i][j] = dp[i+1][j-1] + 2$
- 否则， $dp[i][j] = \max(dp[i+1][j], dp[i][j-1])$

Args:

s: 输入字符串

Returns:

int: 最长回文子序列的长度

"""

```
n = len(s)
# dp[i][j]表示字符串 s 在区间 [i, j] 内的最长回文子序列的长度
dp = [[0] * n for _ in range(n)]

# 初始化单个字符的情况
for i in range(n):
    dp[i][i] = 1

# 枚举区间长度
for length in range(2, n + 1):
    # 枚举起点
    for i in range(n - length + 1):
        j = i + length - 1
        if s[i] == s[j]:
            dp[i][j] = dp[i+1][j-1] + 2
        else:
            dp[i][j] = max(dp[i+1][j], dp[i][j-1])

return dp[0][n-1]
```

```
class SuffixAutomaton:
```

"""

后缀自动机 (Suffix Automaton)

后缀自动机是一个可以表示字符串的所有子串的数据结构。

它可以用于解决许多字符串问题，如子串匹配、最长重复子串等。

"""

```
class State:
```

```
    def __init__(self):
```

```

        self.len = 0 # 该状态能接受的最长字符串的长度
        self.link = -1 # 后缀链接
        self.next = {} # 转移函数
        self.endpos_size = 0 # endpos 集合的大小

def __init__(self, s=None):
    self.size = 1
    self.last = 0
    self.states = [self.State()]

# 如果提供了字符串，构建后缀自动机
if s:
    self.extend(s)

def extend(self, s):
    """
    扩展后缀自动机，添加一个字符串

    Args:
        s: 要添加的字符串
    """

    # 构建后缀自动机
    for c in s:
        self._extend(c)

    # 计算 endpos 集合的大小
    self._calc_endpos_size()

def _extend(self, c):
    p = self.last
    curr = self.size
    self.size += 1
    self.states.append(self.State())
    self.states[curr].len = self.states[p].len + 1

    while p != -1 and c not in self.states[p].next:
        self.states[p].next[c] = curr
        p = self.states[p].link

    if p == -1:
        self.states[curr].link = 0
    else:
        q = self.states[p].next[c]

```

```

        if self.states[p].len + 1 == self.states[q].len:
            self.states[curr].link = q
        else:
            clone = self.size
            self.size += 1
            self.states.append(self.State())
            self.states[clone].len = self.states[p].len + 1
            self.states[clone].next = self.states[q].next.copy()
            self.states[clone].link = self.states[q].link

        while p != -1 and self.states[p].next.get(c) == q:
            self.states[p].next[c] = clone
            p = self.states[p].link

        self.states[q].link = clone
        self.states[curr].link = clone

    self.last = curr

def _calc_endpos_size(self):
    # 按 len 排序
    order = sorted(range(self.size), key=lambda x: -self.states[x].len)

    # 初始化为 1 (每个状态至少对应一个结束位置)
    for i in range(1, self.size):
        self.states[i].endpos_size = 1

    # 从长到短更新
    for u in order:
        if self.states[u].link != -1:
            self.states[self.states[u].link].endpos_size += self.states[u].endpos_size

def count_substrings(self):
    """
    计算不同子串的数量
    """
    count = 0
    for i in range(1, self.size):
        count += self.states[i].len - self.states[self.states[i].link].len
    return count

# ===== 组合 DP (生成函数+DP) =====

```

```
def generating_function_dp(n, k, mod=10**9+7):
"""
生成函数结合 DP 示例: LeetCode 1775. 通过最少操作次数使数组的和相等
题目链接: https://leetcode-cn.com/problems/equal-sum-arrays-with-minimum-number-of-
operations/

```

问题描述:

给定两个长度可能不等的整数数组 `nums1` 和 `nums2`。两个数组中的所有值都在 1 到 6 之间（包含 1 和 6）。

每次操作中，你可以选择 任意 数组中的任意一个整数，将它变成 1 到 6 之间的任意值（包含 1 和 6）。

请返回使 `nums1` 和 `nums2` 之和相等的最少操作次数。如果无法使两个数组之和相等，返回 -1。

解题思路:

使用生成函数的思想构建 DP 数组，表示可以通过多少次操作达到某个差值。

Args:

n: 数组 1 的长度（这里简化为 n）

k: 数组 2 的长度（这里简化为 k）

mod: 模数

Returns:

int: 最小操作次数或-1

时间复杂度: $O(n*k*\max_diff)$ ，其中 `max_diff` 是可能的最大差值

空间复杂度: $O(\max_diff)$

"""

这里提供的是一个简化版本的示例，完整实现需要根据具体数组值计算

生成函数的核心思想是构建状态转移，记录每个可能的差值需要的最少操作次数

pass

```
def polynomial_multiplication(a, b, mod=10**9+7):
```

"""

多项式乘法（生成函数的核心操作）

问题描述:

计算两个多项式的乘积，系数对 mod 取余。

解题思路:

使用动态规划的方法计算多项式乘积。

Args:

a: 第一个多项式的系数数组

b: 第二个多项式的系数数组

mod: 模数

Returns:

list: 乘积多项式的系数数组

时间复杂度: $O(n*m)$, 其中 n 和 m 是两个多项式的次数+1

空间复杂度: $O(n+m)$

"""

```
n = len(a)
```

```
m = len(b)
```

```
# 结果多项式的次数是 n+m-2, 所以系数数组长度为 n+m-1
```

```
result = [0] * (n + m - 1)
```

```
# 计算每个项的系数
```

```
for i in range(n):
```

```
    for j in range(m):
```

```
        result[i + j] = (result[i + j] + a[i] * b[j]) % mod
```

```
return result
```

```
def integer_partition_dp(n):
```

"""

整数划分问题: 使用生成函数和 DP

问题描述:

计算将正整数 n 划分为若干正整数之和的方式数, 不考虑顺序。

例如, 对于 n=4, 划分方式有: 4, 3+1, 2+2, 2+1+1, 1+1+1+1, 共 5 种。

解题思路:

使用生成函数的思想, 每个数 k 对应的生成函数是 $1 + x^k + x^{2k} + \dots$

我们需要计算这些生成函数的乘积中 x^n 的系数。

Args:

n: 要划分的整数

Returns:

int: 划分方式数

时间复杂度: $O(n^2)$

空间复杂度: $O(n)$

"""

```
# dp[i]表示将整数 i 划分为若干正整数之和的方式数
```

```

dp = [0] * (n + 1)
dp[0] = 1 # 基础情况：将 0 划分为 0 个部分有 1 种方式

# 对于每个数 k (从 1 到 n)
for k in range(1, n + 1):
    # 对于每个可能的和 i (从 k 到 n)
    for i in range(k, n + 1):
        # 可以选择使用 k 或不使用 k
        dp[i] += dp[i - k]

return dp[n]

```

===== 博弈 DP (SG 函数、阶梯 Nim) =====

```
def calculate_sg(n, moves):
```

```
"""

```

计算 SG 函数值

问题描述：

在 impartial game 中，计算每个状态的 SG 函数值。

解题思路：

使用动态规划计算每个状态的 SG 函数值。

$SG(x) = \text{mex}\{ SG(y) \mid y \text{ 是 } x \text{ 的后继状态} \}$

其中 mex 是最小非负整数不在集合中的函数。

Args:

n: 最大状态数

moves: 允许的移动方式（例如，每次可以取 1 或 2 个石子）

Returns:

list: 每个状态的 SG 函数值

时间复杂度: $O(n*k)$ ，其中 k 是移动方式的数量

空间复杂度: $O(n)$

```
"""

```

```
sg = [0] * (n + 1)
```

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

```
    visited = set()
```

```
    # 计算所有可能的后继状态的 SG 值
```

```
    for move in moves:
```

```
        if i >= move:
```

```

visited.add(sg[i - move])

# 计算 mex 值
mex = 0
while mex in visited:
    mex += 1
sg[i] = mex

return sg

def nim_game(piles):
"""
Nim 游戏
LeetCode 292. Nim 游戏
题目链接: https://leetcode-cn.com/problems/nim-game/
"""


```

问题描述:

你和你的朋友，两个人一起玩 Nim 游戏：

1. 桌子上有一堆石子
2. 你们轮流进行自己的回合，你作为先手
3. 每一回合，轮到的人拿掉 1 – 3 颗石子
4. 拿掉最后一颗石子的人就是获胜者

解题思路:

当石子数 n 是 4 的倍数时，无论先手怎么拿，后手都可以通过拿 $(4 - x)$ 颗石子使得每轮总共拿 4 颗石子，最终后手获胜。

当石子数 n 不是 4 的倍数时，先手可以先拿走 $(n \% 4)$ 颗石子，使得剩下的石子数是 4 的倍数，之后按照上述策略，先手获胜。

Args:

piles: 石子堆数组（这里简化为单个堆的情况）

Returns:

bool: 如果先手能赢返回 True，否则返回 False

时间复杂度: O(1)

空间复杂度: O(1)

"""

对于单个堆的情况

```

if len(piles) == 1:
    return piles[0] % 4 != 0

```

对于多个堆的情况，计算异或和

```
xor_sum = 0
for pile in piles:
    xor_sum ^= pile

# 如果异或和不为 0，先手有必胜策略
return xor_sum != 0
```

```
def stairs_nim(stairs):
```

```
"""

```

阶梯 Nim 游戏

问题描述：

在阶梯上有若干石子，每次可以将第 i 层的石子移动任意数量到第 $i-1$ 层，或者将第 1 层的石子移出游戏。最后无法移动的人输。

解题思路：

阶梯 Nim 游戏的胜负取决于奇数层石子数的异或和。

Args:

stairs: 每层的石子数

Returns:

bool: 如果先手能赢返回 True，否则返回 False

时间复杂度: $O(n)$

空间复杂度: $O(1)$

```
"""

```

```
xor_sum = 0
```

```
# 只考虑奇数层的石子数
```

```
for i in range(len(stairs)):
```

```
    if i % 2 == 0: # 索引从 0 开始，对应第一层
```

```
        xor_sum ^= stairs[i]
```

```
# 如果异或和不为 0，先手有必胜策略
```

```
return xor_sum != 0
```

```
# ===== 高维压缩（子集/超集枚举与预处理）=====
```

```
def subset_enumeration(n):
```

```
"""

```

子集枚举示例

问题描述：

枚举所有 n 位二进制数的子集。

解题思路:

使用位运算技巧高效枚举子集。

Args:

n: 位数

Returns:

list: 所有子集的列表

时间复杂度: $O(3^n)$, 因为每个位有 3 种可能: 不在原集合中、在原集合中但不在子集中、在原集合中和子集中

空间复杂度: $O(2^n)$

"""

```
subsets = []
# 枚举所有可能的集合
for mask in range(1 << n):
    # 枚举 mask 的所有子集
    subset = mask
    while True:
        subsets.append(subset)
        if subset == 0:
            break
        subset = (subset - 1) & mask
return subsets
```

```
def superset_dp(n, cost):
```

"""

超集 DP (SOS DP)

问题描述:

计算每个集合的所有超集的最小值。

解题思路:

使用动态规划, 按位处理。

Args:

n: 位数

cost: 每个集合的代价

Returns:

list: 每个集合的最小超集代价

```

时间复杂度: O(n*2^n)
空间复杂度: O(2^n)
"""

dp = cost.copy()

# 按位处理
for i in range(n):
    for mask in range(1 << n):
        # 如果 mask 的第 i 位为 0, 则可以将其置为 1, 得到一个超集
        if not (mask & (1 << i)):
            dp[mask] = min(dp[mask], dp[mask | (1 << i)])

```

```
return dp
```

```
def subset_convolution(a, b, mod=10**9+7):
```

```
"""

子集卷积
```

问题描述:

计算两个函数 f 和 g 的子集卷积, 结果 h 满足 $h[s] = \sum_{\{t \text{ subset of } s\}} f[t] * g[s-t]$, 其中 t subset of s 且 t 和 s-t 互不相交。

解题思路:

使用快速莫比乌斯变换 (FMT) 进行高效计算。

Args:

a: 函数 f 的系数数组

b: 函数 g 的系数数组

mod: 模数

Returns:

list: 子集卷积的结果

时间复杂度: O(n^2*2^n)

空间复杂度: O(n*2^n)

```
"""

n = (len(a) - 1).bit_length()  # 位数
size = 1 << n
```

扩展数组长度到 2^n

a += [0] * (size - len(a))

b += [0] * (size - len(b))

```

# 按集合大小分组
fa = [[0] * size for _ in range(n + 1)]
fb = [[0] * size for _ in range(n + 1)]

for mask in range(size):
    cnt = bin(mask).count('1')
    fa[cnt][mask] = a[mask] % mod
    fb[cnt][mask] = b[mask] % mod

# 对每个大小组进行 FMT
for i in range(n + 1):
    # FMT for OR convolution
    for j in range(n):
        for mask in range(size):
            if mask & (1 << j):
                fa[i][mask] = (fa[i][mask] + fa[i][mask ^ (1 << j)]) % mod
                fb[i][mask] = (fb[i][mask] + fb[i][mask ^ (1 << j)]) % mod

# 计算卷积
fc = [[0] * size for _ in range(n + 1)]
for i in range(n + 1):
    for j in range(i + 1):
        for mask in range(size):
            fc[i][mask] = (fc[i][mask] + fa[j][mask] * fb[i - j][mask]) % mod

# 逆 FMT 并合并结果
res = [0] * size
for i in range(n + 1):
    # Inverse FMT
    for j in range(n):
        for mask in range(size):
            if mask & (1 << j):
                fc[i][mask] = (fc[i][mask] - fc[i][mask ^ (1 << j)]) % mod

# 合并结果
for mask in range(size):
    if bin(mask).count('1') == i:
        res[mask] = fc[i][mask] % mod

return res

# ====== 优化体系 ======

```

```
# ===== Knuth 优化 =====
def knuth_optimization(matrix):
    """
```

Knuth 优化示例

问题描述：

对于满足四边形不等式的 DP 问题，可以使用 Knuth 优化将时间复杂度从 $O(n^3)$ 降低到 $O(n^2)$ 。

解题思路：

利用决策单调性，记录最优决策点的范围。

Args:

matrix: 代价矩阵

Returns:

tuple: (最小代价, 最优分割点)

时间复杂度: $O(n^2)$

空间复杂度: $O(n^2)$

```
"""
```

n = len(matrix)

dp[i][j] 表示合并区间 [i, j] 的最小代价

dp = [[0] * n for _ in range(n)]

opt[i][j] 表示合并区间 [i, j] 时的最优分割点

opt = [[0] * n for _ in range(n)]

初始化

for i in range(n):

opt[i][i] = i

枚举区间长度

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

for i in range(n - l + 1):

j = i + l - 1

dp[i][j] = float('inf')

利用决策单调性，限制 k 的范围在 [opt[i][j-1], opt[i+1][j]]

for k in range(opt[i][j-1], min(opt[i+1][j] + 1, j)):

cost = dp[i][k] + dp[k+1][j] + sum(matrix[i][j]) # 这里的 sum 是示例，实际应该根据具体问题计算

if cost < dp[i][j]:

dp[i][j] = cost

opt[i][j] = k

```
return dp[0][n-1], opt

# ===== Divide & Conquer Optimization =====
def divide_conquer_optimization(n, cost_func):
    """
    分治优化 DP

```

问题描述:

对于满足决策单调性的 DP 问题，可以使用分治优化将时间复杂度降低。

解题思路:

利用分治的思想，递归地处理每个区间的最优决策点。

Args:

```
n: 问题规模
cost_func: 计算代价的函数
```

Returns:

```
list: DP 数组
```

时间复杂度: $O(n \log n)$

空间复杂度: $O(n)$

```
"""

```

```
dp = [float('inf')] * n
dp[0] = 0 # 初始状态
```

```
def solve(l, r, opt_l, opt_r):
    """

```

```
    求解区间[l, r]，其中最优决策点在[opt_l, opt_r]之间
    """

```

```
    if l > r:
        return
```

```
        mid = (l + r) // 2
```

```
        best_k = -1
```

```
# 枚举可能的决策点
```

```
        for k in range(opt_l, min(opt_r, mid) + 1):
            current_cost = dp[k] + cost_func(k, mid)
            if current_cost < dp[mid]:
                dp[mid] = current_cost
                best_k = k
```

```

# 递归处理左右子区间
solve(l, mid - 1, opt_l, best_k)
solve(mid + 1, r, best_k, opt_r)

# 求解整个区间
solve(1, n - 1, 0, n - 1)

return dp

```

```

# ===== SMAWK 算法 =====
def smawk_row_minima(matrix, row_indices, col_indices):
    """
    SMAWK 算法：用于在完全单调矩阵中找到每行的最小值

```

问题描述：

对于满足完全单调性的矩阵，快速找到每行的最小值的列索引。

解题思路：

使用 SMAWK 算法的递归实现。

Args:

- matrix: 完全单调矩阵
- row_indices: 要处理的行索引列表
- col_indices: 要处理的列索引列表

Returns:

- dict: 每行的最小值的列索引

时间复杂度: $O(m + n)$, 其中 m 是行数, n 是列数

空间复杂度: $O(m + n)$

"""

```

if not row_indices:
    return {}

```

递归步骤 1: 减少行数 (约简)

```

reduced_rows = []
for row in row_indices:
    while reduced_rows and matrix[reduced_rows[-1]][col_indices[-1]] >=
matrix[row][col_indices[-1]]:
        reduced_rows.pop()
    reduced_rows.append(row)

```

```

# 递归步骤 2: 递归处理约简后的问题，但只处理奇数列
odd_cols = col_indices[1::2]
min_cols = smawk_row_minima(matrix, reduced_rows, odd_cols)

# 递归步骤 3: 在约简后的行中找到所有列的最小值
result = {}
col_ptr = 0 # 跟踪当前考虑的列

for i, row in enumerate(reduced_rows):
    prev_col = -1 if i == 0 else min_cols[reduced_rows[i-1]]
    next_col = min_cols[row] if row in min_cols else col_indices[-1]

    # 找到当前行在[prev_col+1, next_col]范围内的最小值
    min_val = float('inf')
    min_col = -1

    while col_ptr < len(col_indices) and col_indices[col_ptr] <= next_col:
        col = col_indices[col_ptr]
        if prev_col < col <= next_col:
            if matrix[row][col] < min_val:
                min_val = matrix[row][col]
                min_col = col
        col_ptr += 1

    result[row] = min_col

# 对于未约简的行，它们的最小值列必然在约简后的行的最小值列之间
# 这里简化处理，实际情况可能需要更复杂的逻辑
for row in row_indices:
    if row not in result:
        result[row] = 0 # 简化处理

return result

# ===== Aliens Trick =====
def aliens_trick(n, k, check_func):
    """
    Aliens Trick (二分答案+可行性 DP)
    """


```

问题描述：

对于某些带约束的最优化问题，可以使用二分答案结合可行性 DP 来解决。

解题思路：

1. 对约束参数进行二分
2. 对每个参数值，使用 DP 判断是否可行

Args:

- n: 问题规模
- k: 约束参数的上限
- check_func: 检查函数，输入参数 lambda 和 k，返回是否可行

Returns:

- tuple: (最优值, 最优参数)

时间复杂度: $O(\log(\max_{\lambda} \lambda) * T(n))$, 其中 $T(n)$ 是 DP 的时间复杂度

空间复杂度: $O(n)$

"""

```

left = 0
right = 10**9 # 根据具体问题调整上限
best_lambda = 0
best_value = 0

while left <= right:
    mid = (left + right) // 2
    feasible, value = check_func(mid, k)

    if feasible:
        # 可行, 尝试更小的 lambda
        best_lambda = mid
        best_value = value
        right = mid - 1
    else:
        # 不可行, 增大 lambda
        left = mid + 1

return best_value, best_lambda

```

===== 图上 DP→最短路 =====

```
def layered_graph_dijkstra(n, m, edges, start, end, layers):
    """
```

分层图 Dijkstra 算法

问题描述:

在分层图中找到从起点到终点的最短路径。

解题思路：

使用 Dijkstra 算法，将每个状态表示为(节点, 层)，并使用优先队列进行优化。

Args:

- n: 节点数
- m: 边数
- edges: 边的列表
- start: 起点
- end: 终点
- layers: 层数

Returns:

- int: 最短路径长度，如果不可达返回-1

时间复杂度: $O(layers * m \log(layers * n))$

空间复杂度: $O(layers * n)$

"""

```
import heapq

# 构建图的邻接表
graph = [[] for _ in range(n * layers)]
for u, v, w in edges:
    # 在同一层添加边
    for i in range(layers):
        graph[i * n + u].append((i * n + v, w))
    # 在相邻层之间添加边（如果允许换层）
    for i in range(layers - 1):
        graph[i * n + u].append(((i + 1) * n + u, 0))  # 换层的代价为0，根据具体问题调整

# Dijkstra 算法
dist = [float('inf')] * (n * layers)
dist[start] = 0
heap = []
heapq.heappush(heap, (0, start))

while heap:
    current_dist, current_node = heapq.heappop(heap)

    if current_node % n == end:
        return current_dist

    if current_dist > dist[current_node]:
        continue

    for neighbor, weight in graph[current_node]:
        new_dist = current_dist + weight
        if new_dist < dist[neighbor]:
            dist[neighbor] = new_dist
            heapq.heappush(heap, (new_dist, neighbor))
```

```
for next_node, weight in graph[current_node]:  
    if dist[next_node] > current_dist + weight:  
        dist[next_node] = current_dist + weight  
        heapq.heappush(heap, (dist[next_node], next_node))  
  
return -1 # 不可达
```

```
# ===== 冷门模型 =====
```

```
# ===== 期望 DP =====  
def expected_value_dp(n, edges):  
    """
```

期望 DP 示例: LeetCode 837. 新 21 点

题目链接: <https://leetcode-cn.com/problems/new-21-game/>

问题描述:

爱丽丝参与一个大致基于纸牌游戏“21点”规则的游戏，描述如下：

爱丽丝以 0 分开始，并在她的得分小于 K 分时抽取数字。抽取时，她从 $[1, W]$ 的范围内随机获得一个整数作为分数进行累计，其中 W 是整数。每次抽取都是独立的，其结果具有相同的概率。

当爱丽丝获得不少于 K 分时，她就停止抽取数字。爱丽丝的分数不超过 N 的概率是多少？

解题思路:

使用期望 DP，定义 $dp[i]$ 表示当前得分为 i 时，最终得分不超过 N 的概率。

Args:

n: 最大得分限制（相当于题目中的 N ）

edges: 状态转移边（这里简化处理）

Returns:

float: 概率

时间复杂度: $O(n)$

空间复杂度: $O(n)$

"""

```
# 简化版实现，完整实现需要根据具体问题调整
```

```
K = n // 2 # 示例值
```

```
W = 10 # 示例值
```

```
if K == 0 or n >= K + W:  
    return 1.0
```

```
dp = [0.0] * (n + 1)
```

```

dp[K:] = [1.0] * (n - K + 1)

# 计算前缀和，优化转移
sum_dp = n - K + 1 # 初始时 sum(dp[K...n]) = n-K+1

for i in range(K-1, -1, -1):
    dp[i] = sum_dp / W
    sum_dp = sum_dp + dp[i] - dp[i + W] if (i + W) <= n else sum_dp + dp[i]

return dp[0]

```

```
def gaussian_elimination_dp(matrix):
```

```
"""

```

高斯消元解决期望 DP 中的环

问题描述：

当期望 DP 中存在环时，需要建立方程组并使用高斯消元求解。

解题思路：

1. 根据状态转移关系建立线性方程组
2. 使用高斯消元求解方程组

Args:

matrix: 系数矩阵

Returns:

list: 解向量

时间复杂度: $O(n^3)$

空间复杂度: $O(n^2)$

```
"""

```

```
n = len(matrix)
```

```
eps = 1e-8
```

高斯消元

```
for i in range(n):
```

找到当前列中的主元素（绝对值最大的）

```
max_row = i
```

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

```
    if abs(matrix[j][i]) > abs(matrix[max_row][i]):
```

```
        max_row = j
```

交换行

```

matrix[i], matrix[max_row] = matrix[max_row], matrix[i]

# 归一化主行
pivot = matrix[i][i]
if abs(pivot) < eps:
    continue # 奇异矩阵，无法求解

for j in range(i, n + 1):
    matrix[i][j] /= pivot

# 消去其他行
for j in range(n):
    if j != i and abs(matrix[j][i]) > eps:
        factor = matrix[j][i]
        for k in range(i, n + 1):
            matrix[j][k] -= factor * matrix[i][k]

# 提取解
solution = [row[n] for row in matrix]
return solution

# ===== 插头 DP =====
def plug_dp(grid):
    """
    插头 DP 示例：计算网格中的哈密顿回路数目
    """

    problem description:
    Given a grid, calculate the number of Hamiltonian cycles from start to end.

    Solution description:
    Use compressed state dynamic programming (DP), record the status of the contour line's plug.

    Args:
        grid: Grid (0 indicates obstacle, 1 indicates walkable)

    Returns:
        int: Number of Hamiltonian cycles
    """

    time complexity: O(n*m*4^min(n, m)), where n and m are the dimensions of the grid
    space complexity: O(n*m*4^min(n, m))
    """

```

```

# 简化版实现，完整实现需要更复杂的状态转移
n = len(grid)

```

```
m = len(grid[0]) if n > 0 else 0

# 这里仅提供框架，实际实现需要处理插头状态的转移
# 通常使用哈希表或滚动数组优化空间
pass
```

===== 树上背包 =====

```
def tree_knapsack(root, capacity, tree, weights, values):
    """
```

树上背包问题：在树上选择一些节点，使得总重量不超过 capacity，总价值最大

使用后序遍历+动态规划的方法

时间复杂度：O(n*capacity^2)

空间复杂度：O(n*capacity)

Args:

root: 根节点索引

capacity: 背包容量

tree: 树的邻接表表示

weights: 节点重量列表

values: 节点价值列表

Returns:

最大总价值

```
"""
```

```
n = len(tree)
```

```
dp = [[0] * (capacity + 1) for _ in range(n)]
```

```
def dfs(u, parent):
```

初始化：当前节点的重量和价值

```
if weights[u] <= capacity:
```

```
    dp[u][weights[u]] = values[u]
```

遍历所有子节点（排除父节点）

```
for v in tree[u]:
```

```
    if v != parent:
```

```
        dfs(v, u)
```

树上背包的核心：体积要倒序枚举

```
for j in range(capacity, weights[u] - 1, -1):
```

```
    for k in range(1, j - weights[u] + 1):
```

```
        dp[u][j] = max(dp[u][j], dp[u][j - k] + dp[v][k])
```

```
dfs(root, -1)
# 返回所有可能容量中的最大值
return max(dp[root])
```

```
# ===== DP+计算几何（凸包相关）=====
```

```
def convex_hull(points):
```

```
    """

```

```
    计算凸包（Andrew 算法）

```

问题描述：

给定平面上的点集，找出所有在凸包上的点。

解题思路：

1. 将点按 x 坐标排序，x 相同按 y 排序
2. 构建下凸壳和上凸壳
3. 合并上下凸壳

Args:

points: 点集，每个点是一个元组(x, y)

Returns:

list: 凸包上的点集

```
"""

```

```
# 按 x 坐标排序，x 相同按 y 排序
```

```
points = sorted(points)
```

```
n = len(points)
```

```
# 构建下凸壳
```

```
lower = []
```

```
for p in points:
```

```
    while len(lower) >= 2 and cross(lower[-2], lower[-1], p) <= 0:
```

```
        lower.pop()
```

```
    lower.append(p)
```

```
# 构建上凸壳
```

```
upper = []
```

```
for p in reversed(points):
```

```
    while len(upper) >= 2 and cross(upper[-2], upper[-1], p) <= 0:
```

```
        upper.pop()
```

```
    upper.append(p)
```

```
# 合并上下凸壳，去掉重复的端点
```

```

return lower[:-1] + upper[:-1]

def cross(o, a, b):
    """
    计算叉积 (a - o) × (b - o)
    """
    return (a[0] - o[0]) * (b[1] - o[1]) - (a[1] - o[1]) * (b[0] - o[0])

```

```

def convex_hull_trick(dp, a, b):
    """
    凸包优化 DP

```

问题描述:

当 DP 状态转移方程可以表示为 $dp[i] = \min\{dp[j] + a[i] * b[j]\} + c[i]$ 的形式时，
可以使用凸包优化将时间复杂度从 $O(n^2)$ 降低到 $O(n)$ 或 $O(n \log n)$ 。

解题思路:

对于每个 j ，维护一条直线 $y = b[j] * x + dp[j]$ ，然后对于每个 i ，查询 $x = a[i]$ 时的最小值。
当 $b[j]$ 单调递增且 $a[i]$ 单调递增时，可以使用单调队列优化。

Args:

- dp: DP 数组
- a: a 数组
- b: b 数组

Returns:

list: 优化后的 DP 数组

"""

n = len(dp)

q = [] # 单调队列，存储直线的索引

def get_intersection(j1, j2):

计算两条直线 j1 和 j2 的交点 x 坐标

直线 j1: $y = b[j1] * x + dp[j1]$

直线 j2: $y = b[j2] * x + dp[j2]$

return (dp[j2] - dp[j1]) / (b[j1] - b[j2]) if b[j1] != b[j2] else float('inf')

初始化队列，加入第一个元素

q.append(0)

对于每个 i，找到最优的 j

for i in range(1, n):

当队列中至少有两个元素，且第一个元素不如第二个元素优时，弹出第一个元素

```

while len(q) >= 2 and (dp[q[0]] + a[i] * b[q[0]] >= dp[q[1]] + a[i] * b[q[1]]):
    q.pop(0)

# 使用队列中的第一个元素作为最优的 j
dp[i] = min(dp[i], dp[q[0]] + a[i] * b[q[0]])

# 将当前 i 加入队列，维护队列的凸壳性质
while len(q) >= 2 and (get_intersection(q[-2], q[-1]) >= get_intersection(q[-1], i)):
    q.pop()
    q.append(i)

return dp

# 主程序测试
if __name__ == "__main__":
    # 测试零钱兑换 II (模意义)
    coins = [1, 2, 5]
    amount = 5
    mod = 10**9 + 7
    print(f"零钱兑换 II (模{mod}) : {coin_change_mod(coins, amount)}")

    # 测试矩阵快速幂
    matrix = [[1, 1], [1, 0]]
    power = 10
    print("矩阵快速幂:")
    result = matrix_power_mod(matrix, power, mod)
    for row in result:
        print(row)

    # 测试最长回文子序列
    s = "bbbab"
    print(f"最长回文子序列长度: {longest_palindromic_subseq(s)}")

    # 测试后缀自动机
    sam = SuffixAutomaton()
    sam.extend("banana")
    print(f"不同子串数量: {sam.count_substrings()}")

    # 测试凸包
    points = [(0, 0), (1, 1), (2, 0), (0, 2), (1, 0)]
    hull = convex_hull(points)
    print(f"凸包点集: {hull}")

```

```

# 测试凸包优化 DP
dp = [float('inf')] * 5
dp[0] = 0
a = [1, 2, 3, 4, 5]
b = [5, 4, 3, 2, 1]
print(f"凸包优化 DP 结果: {convex_hull_trick(dp, a, b)}")

# 测试分层 DP - 三角形最小路径和
triangle = [[2], [3, 4], [6, 5, 7], [4, 1, 8, 3]]
print(f"三角形最小路径和: {layered_dp(triangle)}")

# 测试分层 DP - 不同路径
m, n = 3, 7
grid = [[0 for _ in range(n)] for _ in range(m)]
print(f"不同路径({m}x{n}): {multi_layered_dp(grid)}")

# 测试容斥 DP - 斯特林数
k, j = 5, 2
print(f"斯特林数 S({k}, {j}): {inclusion_exclusion_dp(k, j)}")

# 测试容斥 DP - 互质对计数
arr = [1, 2, 3, 4, 5]
print(f"互质对计数: {mobius_inversion_dp(arr)}")

# 测试记忆化搜索 - 组合总和 IV
n = 3
target = 4
print(f"组合总和 IV: {memoization_search_example(n, target)}")

# 测试组合 DP - 整数划分
n = 4
print(f"整数划分({n}): {integer_partition_dp(n)}")

# 测试组合 DP - 多项式乘法
a_poly = [1, 2, 3]
b_poly = [4, 5]
print(f"多项式乘法 {a_poly} * {b_poly}: {polynomial_multiplication(a_poly, b_poly)}")

# 测试博奕 DP - SG 函数
moves = [1, 2]
sg_values = calculate_sg(10, moves)
print(f"SG 函数值(1-10): {sg_values[1:]}")

```

```

# 测试博弈 DP - Nim 游戏
piles = [3, 4, 5]
print(f"Nim 游戏先手必胜?: {nim_game(piles)}")

# 测试博弈 DP - 阶梯 Nim
stairs = [2, 1, 3, 1]
print(f"阶梯 Nim 先手必胜?: {stairs_nim(stairs)}")

# 测试高维压缩 - 子集枚举
n_bits = 3
subsets = subset_enumeration(n_bits)
print(f"{n_bits} 位二进制数的所有子集数量: {len(subsets)}")

# 测试高维压缩 - 超集 DP
cost = [10, 5, 7, 3]
min_superset = superset_dp(2, cost)
print(f"超集 DP 结果: {min_superset}")

# 测试期望 DP
print(f"新 21 点概率 (N=21): {expected_value_dp(21, None):.6f}")

# 测试树上背包 - 暂时注释以避免递归深度问题
# tree = [[1, 2], [0, 3], [0, 4], [1], [1]]
# weights = [1, 2, 3, 4, 5]
# values = [10, 20, 30, 40, 50]
# capacity = 10
# print(f"树上背包最大价值: {tree_knapsack(0, capacity, tree, weights)}")

```

===== 优化体系: Knuth 优化 =====

```

# Knuth 优化用于优化形如 dp[i][j] = min{dp[i][k] + dp[k+1][j]} + w(i, j) 的 DP
# 当满足四边形不等式时, 最优转移点单调
# 四边形不等式: w(a, b) + w(c, d) ≤ w(a, d) + w(c, b), 其中 a ≤ c ≤ b ≤ d
# 单调性: w(b, c) ≤ w(a, d), 其中 a ≤ b ≤ c ≤ d

```

```

def knuth_optimization(n, cost_func):
    """

```

Knuth 优化的 DP 算法

问题描述:

解决区间 DP 问题, 其中状态转移方程满足四边形不等式

解题思路:

1. 使用 Knuth 优化将时间复杂度从 $O(n^3)$ 降低到 $O(n^2)$
2. 维护最优转移点数组 $opt[i][j]$, 表示计算 $dp[i][j]$ 时的最优 k 值
3. 根据 $opt[i][j-1] \leq opt[i][j] \leq opt[i+1][j]$ 的性质进行剪枝

Args:

n: 区间长度

cost_func: 计算区间(i, j)代价的函数

Returns:

tuple: (dp 数组, opt 数组)

时间复杂度: $O(n^2)$

空间复杂度: $O(n^2)$

"""

初始化 dp 和 opt 数组

```
dp = [[0] * (n + 1) for _ in range(n + 1)]
opt = [[0] * (n + 1) for _ in range(n + 1)]
```

初始化长度为 1 的区间

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

```
    dp[i][i] = 0
```

```
    opt[i][i] = i
```

枚举区间长度

```
for length in range(2, n + 1):
```

枚举起始点

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

```
    j = i + length - 1
```

初始化为无穷大

```
    dp[i][j] = float('inf')
```

根据 Knuth 优化的性质, 最优 k 在 $opt[i][j-1]$ 到 $opt[i+1][j]$ 之间

```
for k in range(opt[i][j-1], min(opt[i+1][j], j-1) + 1):
```

```
    current = dp[i][k] + dp[k+1][j] + cost_func(i, j)
```

```
    if current < dp[i][j]:
```

```
        dp[i][j] = current
```

```
        opt[i][j] = k
```

```
return dp, opt
```

===== 优化体系: Divide & Conquer Optimization =====

```
def divide_conquer_optimization(n, m, cost_func):
```

"""

Divide & Conquer Optimization (分治优化)

问题描述:

解决形如 $dp[i][j] = \min\{dp[i-1][k] + cost(k, j)\}$, 其中 $k < j$

当转移满足决策单调性时使用

解题思路:

1. 利用决策单调性, 使用分治法优化 DP
2. 对于 $dp[i][j]$, 当 i 固定时, 最优转移点 k 随着 j 的增加而单调不减
3. 使用分治的方式计算每个区间的最优决策

Args:

n: 维度 1
m: 维度 2
cost_func: 计算 $cost(k, j)$ 的函数

Returns:

list: dp 数组

时间复杂度: $O(n*m \log m)$

空间复杂度: $O(n*m)$

"""

```
# 初始化 dp 数组
dp = [[float('inf')] * (m + 1) for _ in range(n + 1)]
dp[0][0] = 0

# 分治优化函数
def solve(i, l, r, opt_l, opt_r):
    """
    计算 dp[i][l..r], 其中最优转移点在 opt_l..opt_r 之间
    """
    if l > r:
        return

    mid = (l + r) // 2
    best_k = opt_l

    # 在 opt_l 到 min(mid-1, opt_r) 之间寻找最优 k
    for k in range(opt_l, min(mid, opt_r) + 1):
        if dp[i-1][k] + cost_func(k, mid) < dp[i][mid]:
            dp[i][mid] = dp[i-1][k] + cost_func(k, mid)
            best_k = k
```

```

# 递归处理左右子区间
solve(i, 1, mid-1, opt_l, best_k)
solve(i, mid+1, r, best_k, opt_r)

# 对每个 i 应用分治优化
for i in range(1, n + 1):
    solve(i, 1, m, 0, m)

return dp

```

===== 优化体系: SMAWK 算法 (行最小查询) =====

```

def smawk(matrix):
    """
SMAWK 算法用于在 Monge 矩阵中快速查找每行的最小值

```

问题描述:

给定一个 Monge 矩阵, 快速找到每行的最小值位置

解题思路:

1. Monge 矩阵满足性质: $\text{matrix}[i][j] + \text{matrix}[i+1][j+1] \leq \text{matrix}[i][j+1] + \text{matrix}[i+1][j]$
2. SMAWK 算法利用这一性质, 可以在 $O(m+n)$ 时间内找到每行的最小值
3. 主要步骤包括行压缩和递归求解

Args:

matrix: 一个 Monge 矩阵

Returns:

list: 每行最小值的列索引

时间复杂度: $O(m+n)$, 其中 m 是行数, n 是列数

空间复杂度: $O(m+n)$

"""

```

m = len(matrix)
n = len(matrix[0]) if m > 0 else 0

def reduce_rows(rows):
    """行压缩: 只保留可能成为最小值的行"""
    stack = []
    for i in rows:
        while len(stack) >= 2:
            j1 = stack[-2]

```

```

j2 = stack[-1]
# 比较两个行在列 len(stack)-1 处的值
if matrix[j1][len(stack)-1] <= matrix[i][len(stack)-1]:
    break
else:
    stack.pop()
stack.append(i)
return stack

def smawk_rec(rows, cols):
    """递归实现 SMAWK 算法"""
    if not rows:
        return []
    # 行压缩
    reduced_rows = reduce_rows(rows)

    # 递归求解列数为奇数的子问题
    half_cols = cols[1::2]  # 取所有奇数索引的列
    min_cols = smawk_rec(reduced_rows, half_cols)

    # 扩展结果到所有列
    result = [0] * len(rows)
    k = 0  # min_cols 的索引

    for i, row in enumerate(rows):
        # 确定当前行的最小值可能在哪个区间
        start = 0 if i == 0 else result[rows.index(reduced_rows[k-1])] if k > 0 else 0
        end = min_cols[k] if k < len(min_cols) else cols[-1]

        # 在这个区间内查找最小值
        min_val = float('inf')
        min_col = start

        # 注意这里 cols 是原始列的子集，需要在 cols 中遍历
        for j in range(cols.index(start), cols.index(end) + 1):
            col = cols[j]
            if matrix[row][col] < min_val:
                min_val = matrix[row][col]
                min_col = col

        result[i] = min_col

```

```

# 如果当前行在 reduced_rows 中，且不是最后一行，k 前进
if k < len(reduced_rows) and row == reduced_rows[k]:
    k += 1

return result

return smawk_rec(list(range(m)), list(range(n)))

# ===== 优化体系: Aliens Trick (二分约束参数+可行性 DP) =====

```

def aliens_trick(cost_func, check_func, left, right, eps=1e-7):
 """
 Aliens Trick (二分约束参数+可行性 DP)
 """

问题描述:

解决带约束的优化问题，通常形如最小化总成本，同时满足某些约束条件

解题思路:

1. 将约束条件转化为参数 λ ，构造拉格朗日函数
2. 对 λ 进行二分查找，使用可行性 DP 判断当前 λ 下是否满足约束
3. 根据可行性 DP 的结果调整二分区间

Args:

- cost_func: 计算带参数 λ 的成本函数
- check_func: 检查当前解是否满足约束的函数
- left: 二分左边界
- right: 二分右边界
- eps: 精度要求

Returns:

- tuple: (最优参数 λ ，对应的最优解)

时间复杂度: $O(\log((right-left)/\text{eps}) * T(DP))$ ，其中 $T(DP)$ 是一次 DP 的时间复杂度

"""

best_lam = left

best_value = None

while right - left > eps:

mid = (left + right) / 2

计算当前参数下的解和约束值

current_value, constraint_value = cost_func(mid)

if check_func(constraint_value):

```

# 满足约束，尝试更小的参数
right = mid
best_lam = mid
best_value = current_value

else:
    # 不满足约束，需要增大参数
    left = mid

return best_lam, best_value

```

===== 图上 DP→最短路：分层图建模 =====

分层图建模通常用于处理有状态转移的最短路问题
例如：允许 k 次跳跃、k 次免费通行等情况

```

def layered_graph_dijkstra(n, m, edges, k):
    """
    分层图 Dijkstra 算法

```

问题描述：

给定一个图，允许最多使用 k 次特殊操作（如跳跃、免费通行等），求最短路径

解题思路：

1. 构建分层图，每层代表使用不同次数的特殊操作
2. 对于每个节点 u，在第 i 层表示到达 u 时已经使用了 i 次特殊操作
3. 使用 Dijkstra 算法在分层图上寻找最短路径

Args:

n: 节点数量
m: 边的数量
edges: 边的列表，每个元素为(u, v, w) 表示 u 到 v 的权为 w 的边
k: 允许使用的特殊操作次数

Returns:

int: 从节点 1 到节点 n 的最短路径长度

时间复杂度: $O((n*k + m*k) \log(n*k))$

空间复杂度: $O(n*k + m*k)$

"""

import heapq

构建分层图的邻接表

graph = [[] for _ in range(n * (k + 1))]

```

# 添加普通边（不使用特殊操作）
for u, v, w in edges:
    for i in range(k + 1):
        graph[u + i * n].append((v + i * n, w))

# 添加使用特殊操作的边（如果允许的话）
for u, v, w in edges:
    for i in range(k):
        # 这里假设特殊操作可以免费通行（权为 0），具体根据问题调整
        graph[u + i * n].append((v + (i + 1) * n, 0))

# Dijkstra 算法
dist = [float('inf')] * (n * (k + 1))
dist[0] = 0 # 假设起点是节点 0
heap = [(0, 0)] # (距离, 节点)

while heap:
    d, u = heapq.heappop(heap)
    if d > dist[u]:
        continue

    for v, w in graph[u]:
        if dist[v] > d + w:
            dist[v] = d + w
            heapq.heappush(heap, (dist[v], v))

# 取所有层中到达终点的最小值
return min(dist[n-1 + i * n] for i in range(k + 1))

# ====== 冷门模型：期望 DP 遇环的方程组解（高斯消元） ======

```

```

def gaussian_elimination(matrix):
    """

```

高斯消元法求解线性方程组

问题描述：

求解形如 $Ax = b$ 的线性方程组

解题思路：

1. 构建增广矩阵
2. 进行高斯消元，将矩阵转化为行阶梯形
3. 回代求解

Args:

matrix: 增广矩阵, 每行最后一个元素是 b 的值

Returns:

list: 方程组的解

时间复杂度: $O(n^3)$

空间复杂度: $O(n^2)$

"""

n = len(matrix)

eps = 1e-9

高斯消元过程

for i in range(n):

找到主元行 (当前列中绝对值最大的行)

max_row = i

for j in range(i, n):

if abs(matrix[j][i]) > abs(matrix[max_row][i]):

max_row = j

交换主元行和当前行

matrix[i], matrix[max_row] = matrix[max_row], matrix[i]

如果主元为 0, 方程组可能有无穷多解或无解

if abs(matrix[i][i]) < eps:

continue

消元过程

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

factor = matrix[j][i] / matrix[i][i]

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

matrix[j][k] -= factor * matrix[i][k]

回代求解

x = [0] * n

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

x[i] = matrix[i][n]

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

x[i] -= matrix[i][j] * x[j]

x[i] /= matrix[i][i]

return x

```
def expectation_dp_with_cycles(n, transitions):
```

```
    """
```

```
期望 DP 处理有环情况（使用高斯消元）
```

问题描述：

在有环的状态转移图中计算期望

解题思路：

1. 对于每个状态，建立期望方程
2. 使用高斯消元求解方程组

Args:

n: 状态数量

transitions: 转移概率列表，`transitions[i]`是一个列表，每个元素为(j, p)表示从 i 转移到 j 的概率为 p

Returns:

list: 每个状态的期望值

时间复杂度: $O(n^3)$

空间复杂度: $O(n^2)$

```
"""
```

构建线性方程组的增广矩阵

```
matrix = [[0.0] * (n + 1) for _ in range(n)]
```

```
for i in range(n):
```

```
    matrix[i][i] = 1.0 # 方程左边: E[i] - sum(p_ij * E[j]) = cost[i]
```

假设每个状态的代价为 1，具体根据问题调整

```
cost = 1.0
```

```
matrix[i][n] = cost
```

```
for j, p in transitions[i]:
```

```
    if i != j: # 避免自环的特殊处理
```

```
        matrix[i][j] -= p
```

使用高斯消元求解

```
return gaussian_elimination(matrix)
```

```
# ===== 冷门模型：插头 DP (轮廓线 DP) =====
```

```
def plug_dp(grid):
```

"""

插头 DP (轮廓线 DP) 示例：求网格中哈密顿回路的数量

问题描述：

给定一个网格，求其中哈密顿回路的数量

解题思路：

1. 使用轮廓线 DP，记录当前处理到的位置和轮廓线状态
2. 插头表示连接的状态，通常用二进制表示
3. 使用哈希表优化空间复杂度

Args:

grid: 网格，1 表示可通行，0 表示障碍物

Returns:

int: 哈密顿回路的数量

时间复杂度: $O(n*m*4^{\min(n, m)})$

空间复杂度: $O(4^{\min(n, m)})$

"""

```
from collections import defaultdict
```

```
n = len(grid)
```

```
if n == 0:
```

```
    return 0
```

```
m = len(grid[0])
```

```
# 使用滚动数组优化
```

```
dp = defaultdict(int)
```

```
# 初始状态：左上角没有插头
```

```
dp[0] = 1
```

```
for i in range(n):
```

```
    # 新的一行开始，需要将状态左移一位
```

```
    new_dp = defaultdict(int)
```

```
    for state in dp:
```

```
        # 左移一位，移除最左边的插头
```

```
        new_state = state << 1
```

```
        new_dp[new_state] = dp[state]
```

```
    dp = new_dp
```

```
for j in range(m):
```

```

new_dp = defaultdict(int)

for state in dp:
    # 当前位置左边和上边的插头状态
    left = (state >> (2 * j)) & 3
    up = (state >> (2 * (j + 1))) & 3

    # 如果当前位置是障碍物，跳过
    if grid[i][j] == 0:
        # 只有当左右插头都不存在时才合法
        if left == 0 and up == 0:
            new_dp[state] += dp[state]
            continue

    # 处理各种插头组合情况
    # 1. 没有左插头和上插头
    if left == 0 and up == 0:
        # 只能创建新的插头对（用于回路的开始）
        if i < n - 1 and j < m - 1 and grid[i+1][j] and grid[i][j+1]:
            new_state = state | (1 << (2 * j)) | (2 << (2 * (j + 1)))
            new_dp[new_state] += dp[state]

    # 2. 只有左插头
    elif left != 0 and up == 0:
        # 向下延伸
        if i < n - 1 and grid[i+1][j]:
            new_state = state
            new_dp[new_state] += dp[state]
        # 向右延伸
        if j < m - 1 and grid[i][j+1]:
            new_state = state & ~(3 << (2 * j)) | (left << (2 * (j + 1)))
            new_dp[new_state] += dp[state]

    # 3. 只有上插头
    elif left == 0 and up != 0:
        # 向右延伸
        if j < m - 1 and grid[i][j+1]:
            new_state = state
            new_dp[new_state] += dp[state]
        # 向下延伸
        if i < n - 1 and grid[i+1][j]:
            new_state = state & ~(3 << (2 * (j + 1))) | (up << (2 * j))
            new_dp[new_state] += dp[state]

```

```

# 4. 同时有左插头和上插头
else:
    # 合并插头
    new_state = state & ~(3 << (2 * j)) & ~(3 << (2 * (j + 1)))

    # 如果是形成回路的最后一步
    if left == up:
        # 检查是否所有插头都已连接
        if new_state == 0 and i == n - 1 and j == m - 1:
            new_dp[new_state] += dp[state]
    else:
        # 合并两个不同的插头
        new_dp[new_state] += dp[state]

dp = new_dp

# 最终状态应该是没有任何插头（形成回路）
return dp.get(0, 0)

```

====== 冷门模型：树上背包的优化 ======

```
def tree_knapsack_optimized(root, capacity, tree, weights, values):
```

```
"""

```

树上背包的优化实现（小到大合并）

问题描述：

在树上选择一些节点，使得总重量不超过容量，且总价值最大

解题思路：

1. 使用后序遍历处理子树
2. 使用小到大合并的策略优化复杂度
3. 对于每个节点，维护一个容量为 capacity 的背包

Args:

- root: 根节点
- capacity: 背包容量
- tree: 树的邻接表
- weights: 每个节点的重量
- values: 每个节点的价值

Returns:

- int: 最大价值

时间复杂度: $O(n * \text{capacity}^2)$, 但通过小到大合并可以降低常数

空间复杂度: $O(n * \text{capacity})$

"""

```
n = len(tree)
dp = [[0] * (capacity + 1) for _ in range(n)]
size = [0] * n

def dfs(u, parent):
    # 初始化当前节点
    size[u] = 1
    dp[u][weights[u]] = values[u]

    # 对每个子节点, 按照子树大小排序, 小的先合并
    children = []
    for v in tree[u]:
        if v != parent:
            dfs(v, u)
            children.append((size[v], v))

    # 按子树大小排序
    children.sort()

    for _, v in children:
        # 逆序遍历容量, 避免重复计算
        for i in range(min(size[u], capacity), weights[u] - 1, -1):
            for j in range(1, min(size[v], capacity - i) + 1):
                if i + j <= capacity:
                    dp[u][i + j] = max(dp[u][i + j], dp[u][i] + dp[v][j])

    # 更新子树大小
    size[u] += size[v]

dfs(root, -1)

# 返回根节点的最大价值
return max(dp[root])
```

===== 补充题目与应用 =====

以下是一些使用上述高级 DP 技术的经典题目及其代码实现

1. 编辑距离问题 (LeetCode 72)

```
def edit_distance(word1, word2):
```

```
"""
```

LeetCode 72. 编辑距离

题目链接: <https://leetcode-cn.com/problems/edit-distance/>

问题描述:

给你两个单词 word1 和 word2，计算出将 word1 转换成 word2 所使用的最少操作数。

你可以对一个单词进行如下三种操作：插入一个字符、删除一个字符、替换一个字符。

解题思路:

使用二维 DP， $dp[i][j]$ 表示 word1 的前 i 个字符转换为 word2 的前 j 个字符所需的最少操作数。

时间复杂度: $O(m \times n)$

空间复杂度: $O(m \times n)$

```
"""
```

```
m, n = len(word1), len(word2)
```

```
# dp[i][j] 表示 word1 的前 i 个字符转换为 word2 的前 j 个字符所需的最少操作数
```

```
dp = [[0] * (n + 1) for _ in range(m + 1)]
```

```
# 初始化边界
```

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

```
    dp[i][0] = i
```

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

```
    dp[0][j] = j
```

```
# 动态规划填表
```

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

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

```
        if word1[i-1] == word2[j-1]:
```

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

```
        else:
```

```
            dp[i][j] = min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1]) + 1
```

```
return dp[m][n]
```

```
# 2. 最长递增子序列 (LeetCode 300)
```

```
def length_of_lis(nums):
```

```
"""
```

LeetCode 300. 最长递增子序列

题目链接: <https://leetcode-cn.com/problems/longest-increasing-subsequence/>

问题描述:

给你一个整数数组 $nums$ ，找到其中最长严格递增子序列的长度。

解题思路:

使用贪心 + 二分查找优化的 DP 方法。

tails[i] 表示长度为 i+1 的递增子序列的末尾元素的最小值。

时间复杂度: $O(n \log n)$

空间复杂度: $O(n)$

"""

```
if not nums:
```

```
    return 0
```

```
tails = []
```

```
for num in nums:
```

```
    # 二分查找 num 应该插入的位置
```

```
    left, right = 0, len(tails)
```

```
    while left < right:
```

```
        mid = (left + right) // 2
```

```
        if tails[mid] < num:
```

```
            left = mid + 1
```

```
        else:
```

```
            right = mid
```

```
    if left == len(tails):
```

```
        tails.append(num)
```

```
    else:
```

```
        tails[left] = num
```

```
return len(tails)
```

3. 背包问题变种 - 完全背包 (LeetCode 322)

```
def coin_change(coins, amount):
```

"""

LeetCode 322. 零钱兑换

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

问题描述:

给你一个整数数组 coins，表示不同面额的硬币；以及一个整数 amount，表示总金额。

计算并返回可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额，返回-1。

解题思路:

使用完全背包的思想，dp[i] 表示凑成金额 i 所需的最少硬币数。

时间复杂度: $O(amount * n)$

空间复杂度: $O(amount)$

```

"""
# 初始化 dp 数组为无穷大
dp = [float('inf')] * (amount + 1)
dp[0] = 0 # 溜成金额 0 需要 0 个硬币

for coin in coins:
    for i in range(coin, amount + 1):
        if dp[i - coin] != float('inf'):
            dp[i] = min(dp[i], dp[i - coin] + 1)

return dp[amount] if dp[amount] != float('inf') else -1

```

4. 矩阵链乘法（区间 DP 的经典应用）

```

def matrix_chain_order(p):
"""

```

矩阵链乘法问题

题目来源：算法导论

问题描述：

给定一系列矩阵，计算乘法顺序使得标量乘法的次数最少。

解题思路：

使用区间 DP， $dp[i][j]$ 表示计算第 i 到第 j 个矩阵的乘积所需的最少标量乘法次数。

可以使用 Knuth 优化进一步降低时间复杂度。

时间复杂度： $O(n^3)$

空间复杂度： $O(n^2)$

"""

```
n = len(p) - 1 # 矩阵的个数
```

$dp[i][j]$ 表示计算第 i 到第 j 个矩阵的乘积所需的最少标量乘法次数

```
dp = [[0] * (n + 1) for _ in range(n + 1)]
```

$s[i][j]$ 记录最优分割点

```
s = [[0] * (n + 1) for _ in range(n + 1)]
```

枚举区间长度

```
for length in range(2, n + 1):
```

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

```
        j = i + length - 1
```

```
        dp[i][j] = float('inf')
```

枚举分割点

```
        for k in range(i, j):
```

计算当前分割点的代价

```
            cost = dp[i][k] + dp[k+1][j] + p[i-1] * p[k] * p[j]
```

```

        if cost < dp[i][j]:
            dp[i][j] = cost
            s[i][j] = k

    return dp, s

```

5. 旅行商问题 (TSP) 的 DP 实现

```

def traveling_salesman_problem(graph):
    """

```

旅行商问题

题目来源：算法竞赛经典问题

问题描述：

给定一个完全图，找到一条访问每个城市恰好一次并返回起点的最短路径。

解题思路：

使用状态压缩 DP， $dp[mask][u]$ 表示访问过的城市集合为 $mask$ ，当前在城市 u 时的最短路径长度。

时间复杂度： $O(n^2 * 2^n)$

空间复杂度： $O(n * 2^n)$

"""

```
n = len(graph)
```

```
# dp[mask][u] 表示访问过的城市集合为 mask，当前在城市 u 时的最短路径长度
```

```
dp = [[float('inf')] * n for _ in range(1 << n)]
```

初始状态：只访问了起点，路径长度为 0

```
for i in range(n):
```

```
    dp[1 << i][i] = 0
```

枚举所有可能的状态

```
for mask in range(1, 1 << n):
```

枚举当前所在的城市

```
for u in range(n):
```

```
    if not (mask & (1 << u)):
```

```
        continue
```

枚举下一个要访问的城市

```
for v in range(n):
```

```
    if mask & (1 << v):
```

```
        continue
```

```
    new_mask = mask | (1 << v)
```

```
    dp[new_mask][v] = min(dp[new_mask][v], dp[mask][u] + graph[u][v])
```

找到最短的回路

```

result = float('inf')
for u in range(n):
    result = min(result, dp[(1 << n) - 1][u] + graph[u][0])

return result

```

6. 区间 DP: 最优三角剖分

```

def minimum_score_triangulation(values):
    """

```

LeetCode 1039. 多边形三角剖分的最低得分

题目链接: <https://leetcode-cn.com/problems/minimum-score-triangulation-of-polygon/>

问题描述:

给定一个凸多边形，将其三角剖分，使得所有三角形的顶点乘积之和最小。

解题思路:

使用区间 DP， $dp[i][j]$ 表示从顶点 i 到顶点 j 的多边形三角剖分的最小得分。

时间复杂度: $O(n^3)$

空间复杂度: $O(n^2)$

"""

```
n = len(values)
```

$dp[i][j]$ 表示从顶点 i 到顶点 j 的多边形三角剖分的最小得分

```
dp = [[0] * n for _ in range(n)]
```

枚举区间长度

```
for length in range(3, n + 1):
```

```
    for i in range(n - length + 1):
```

```
        j = i + length - 1
```

```
        dp[i][j] = float('inf')
```

枚举中间点

```
        for k in range(i + 1, j):
```

```
            dp[i][j] = min(dp[i][j], dp[i][k] + dp[k][j] + values[i] * values[k] * values[j])
```

```
return dp[0][n-1]
```

7. 博弈 DP: 石子游戏

```

def stone_game(piles):
    """

```

LeetCode 877. 石子游戏

题目链接: <https://leetcode-cn.com/problems/stone-game/>

问题描述:

给定一个表示石子堆的数组，两个玩家轮流从两端取石子，每次只能取一个，取到最后一个石子的人获胜。

判断先手是否必胜。

解题思路：

使用区间 DP， $dp[i][j]$ 表示在区间 $[i, j]$ 中，先手能获得的最大净胜分。

时间复杂度： $O(n^2)$

空间复杂度： $O(n^2)$

"""

```
n = len(piles)
```

```
# dp[i][j] 表示在区间 [i, j] 中，先手能获得的最大净胜分
```

```
dp = [[0] * n for _ in range(n)]
```

```
# 初始化单个石子堆
```

```
for i in range(n):
```

```
    dp[i][i] = piles[i]
```

```
# 枚举区间长度
```

```
for length in range(2, n + 1):
```

```
    for i in range(n - length + 1):
```

```
        j = i + length - 1
```

```
        # 先手可以选择取左边或右边
```

```
        dp[i][j] = max(piles[i] - dp[i+1][j], piles[j] - dp[i][j-1])
```

```
# 先手净胜分大于 0 则必胜
```

```
return dp[0][n-1] > 0
```

8. 数位 DP：统计 1 出现的次数

```
def count_digit_one(n):
```

"""

LeetCode 233. 数字 1 的个数

题目链接：<https://leetcode-cn.com/problems/number-of-digit-one/>

问题描述：

给定一个整数 n ，计算所有小于等于 n 的非负整数中数字 1 出现的个数。

解题思路：

使用数位 DP，逐位处理每一位上 1 出现的次数。

时间复杂度： $O(\log n)$

空间复杂度： $O(\log n)$

"""

```

if n <= 0:
    return 0

s = str(n)
length = len(s)
count = 0

# 逐位处理
for i in range(length):
    high = int(s[:i]) if i > 0 else 0
    current = int(s[i])
    low = int(s[i+1:]) if i < length - 1 else 0
    digit = 10 ** (length - i - 1)

    if current == 0:
        # 当前位为 0, 高位决定
        count += high * digit
    elif current == 1:
        # 当前位为 1, 高位+低位+1
        count += high * digit + low + 1
    else:
        # 当前位大于 1, 高位+1
        count += (high + 1) * digit

return count

```

9. 树形 DP：打家劫舍 III

```

def rob(root):
    """
    LeetCode 337. 打家劫舍 III
    题目链接: https://leetcode-cn.com/problems/house-robber-iii/

```

问题描述：

在上次打劫完一条街道之后和一圈房屋后，小偷又发现了一个新的可行窃的地区。这个地区只有一个入口，我们称之为“根”。除了“根”之外，每栋房子有且只有一个“父”房子与之相连。

一番侦察之后，聪明的小偷意识到“这个地方的所有房屋的排列类似于一棵二叉树”。

如果两个直接相连的房子在同一天晚上被打劫，房屋将自动报警。

计算在不触动警报的情况下，小偷一晚能够盗取的最高金额。

解题思路：

使用树形 DP，对于每个节点，维护两个状态：偷或不偷。

时间复杂度: $O(n)$

空间复杂度: $O(h)$, h 为树的高度

"""

辅助函数, 返回(偷当前节点的最大金额, 不偷当前节点的最大金额)

def dfs(node):

 if not node:

 return (0, 0)

 left_rob, left_not_rob = dfs(node.left)

 right_rob, right_not_rob = dfs(node.right)

 # 偷当前节点, 不能偷子节点

 rob_current = node.val + left_not_rob + right_not_rob

 # 不偷当前节点, 可以选择偷或不偷子节点

 not_rob_current = max(left_rob, left_not_rob) + max(right_rob, right_not_rob)

 return (rob_current, not_rob_current)

假设 root 是一个包含 val、left、right 属性的二叉树节点

这里返回两种状态的最大值

return max(dfs(root))

10. 状态压缩 DP: 蒙斯特曼问题

def monster_game(grid):

"""

蒙斯特曼问题

题目来源: 算法竞赛问题

问题描述:

在网格中放置怪物, 使得任何两个怪物都不在同一行、同一列或对角线上。

解题思路:

使用状态压缩 DP, $dp[i][mask]$ 表示处理到第 i 行, 已放置的列的状态为 $mask$ 时的方案数。

时间复杂度: $O(n * 2^n)$

空间复杂度: $O(2^n)$

"""

n = len(grid)

$dp[i][mask]$ 表示处理到第 i 行, 已放置的列的状态为 $mask$ 时的方案数

dp = [0] * (1 << n)

dp[0] = 1

for i in range(n):

```

new_dp = [0] * (1 << n)
for mask in range(1 << n):
    if dp[mask] == 0:
        continue
    # 枚举所有可能的放置位置
    for j in range(n):
        # 检查是否可以在(i, j)放置怪物
        if (mask & (1 << j)) == 0 and grid[i][j] == 1:
            # 检查对角线
            valid = True
            for k in range(i):
                if (mask & (1 << k)) and abs(k - j) == i - k:
                    valid = False
                    break
            if valid:
                new_dp[mask | (1 << j)] += dp[mask]
dp = new_dp

return dp[(1 << n) - 1]

```

11. 高维 DP: 三维背包

```
def three_dimension_knapsack(n, capacity, items):
```

```
"""

```

三维背包问题

题目来源: 算法竞赛问题

问题描述:

有 n 个物品, 每个物品有体积、重量、价值三个属性, 背包有体积和重量两个限制, 求最大价值。

解题思路:

使用三维 DP, $dp[i][j][k]$ 表示前 i 个物品, 体积为 j , 重量为 k 时的最大价值。

时间复杂度: $O(n * V * W)$

空间复杂度: $O(n * V * W)$

```
"""

```

$V, W = capacity$

初始化 dp 数组

```
dp = [[[0] * (W + 1) for _ in range(V + 1)] for _ in range(n + 1)]
```

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

```
    v, w, val = items[i-1]
```

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

```
        for k in range(W + 1):
```

```

# 不选当前物品
dp[i][j][k] = dp[i-1][j][k]
# 选当前物品（如果有足够的空间）
if j >= v and k >= w:
    dp[i][j][k] = max(dp[i][j][k], dp[i-1][j-v][k-w] + val)

return dp[n][V][W]

```

12. 斜率优化 DP 示例

```
def convex_hull_trick(points):
    """

```

凸包优化技巧示例

题目来源：算法竞赛问题

问题描述：

当状态转移方程形如 $dp[i] = \min\{dp[j] + a[i] * b[j] + c\}$ 时，可以使用凸包优化。

解题思路：

将转移方程转换为直线的形式，维护凸包以快速查询最小值。

时间复杂度：O(n)

空间复杂度：O(n)

```
"""

```

示例实现，具体问题需要根据实际转移方程调整

```
from collections import deque
```

假设我们有一系列直线 $y = kx + b$

这里使用双端队列维护凸包

```
dq = deque()
```

添加一条直线 $y = kx + b$

```
def add_line(k, b):
```

当队列中至少有两条直线时，检查是否需要删除末尾的直线

```
while len(dq) >= 2:
```

```
    k1, b1 = dq[-2]
```

```
    k2, b2 = dq[-1]
```

判断直线 k_1x+b_1 和 k_2x+b_2 的交点是否在 k_2x+b_2 和 $kx+b$ 的交点右侧

```
    if (b2 - b1) * (k - k2) >= (b - b2) * (k2 - k1):
```

```
        dq.pop()
```

```
else:
```

```
    break
```

```
dq.append((k, b))
```

```

# 查询 x 处的最小值
def query(x):
    # 如果队列中至少有两条直线，且第一条直线在 x 处的值大于第二条，删除第一条
    while len(dq) >= 2:
        k1, b1 = dq[0]
        k2, b2 = dq[1]
        if k1 * x + b1 >= k2 * x + b2:
            dq.popleft()
        else:
            break
    if not dq:
        return float('inf')
    k, b = dq[0]
    return k * x + b

# 这里只是返回优化后的函数，实际问题中需要根据具体情况调用
return add_line, query

```

=====

文件: IntervalColoring.java

=====

```
// -*- coding: utf-8 -*-
```

```
/*
```

区间染色算法（结合贪心）

问题描述：

给定多个区间，每个区间代表一个需要染色的区域，每次可以选择一个区间进行染色，求最少需要多少次染色才能覆盖所有给定的区间。

贪心策略：

按照区间的右端点进行排序，每次选择能够覆盖当前未染色区域的最右端点的区间。

时间复杂度： $O(n \log n)$ ，其中 n 是区间的数量，主要是排序的时间复杂度

空间复杂度： $O(1)$ ，只需要常量级的额外空间

相关题目：

1. LeetCode 435. 无重叠区间
 2. LeetCode 452. 用最少数量的箭引爆气球
 3. LeetCode 253. 会议室 II
- */

```
import java.util.*;
```

```
public class IntervalColoring {  
    /**  
     * 区间染色算法实现  
     *  
     * @param intervals 区间列表，每个区间是一个 int 数组 [start, end]  
     * @return 最少需要的染色次数  
     * @throws IllegalArgumentException 当输入无效时抛出异常  
     */  
    public static int intervalColoring(int[][] intervals) {  
        // 参数验证  
        if (intervals == null || intervals.length == 0) {  
            return 0;  
        }  
  
        for (int[] interval : intervals) {  
            if (interval == null || interval.length != 2 || interval[0] > interval[1]) {  
                throw new IllegalArgumentException("每个区间必须是有效的[start, end]数组，且 start  
<= end");  
            }  
        }  
  
        // 按照区间的右端点进行排序  
        Arrays.sort(intervals, Comparator.comparingInt(a -> a[1]));  
  
        // 贪心选择  
        int count = 1; // 至少需要一次染色  
        int end = intervals[0][1]; // 当前已经覆盖到的最右端点  
  
        for (int i = 1; i < intervals.length; i++) {  
            // 如果当前区间的左端点大于已经覆盖到的最右端点，需要新的染色  
            if (intervals[i][0] > end) {  
                count++;  
                end = intervals[i][1];  
            }  
        }  
  
        return count;  
    }  
  
    /**  
     * LeetCode 452. 用最少数量的箭引爆气球  
     * 题目链接: https://leetcode-cn.com/problems/minimum-number-of-arrows-to-burst-balloons/  
     */
```

```

*
* 问题描述:
* 在二维空间中有许多球形的气球。对于每个气球，提供的输入是水平方向上，气球直径的开始和结束坐标。
* 由于它是水平的，所以 y 坐标并不重要，因此只要知道开始和结束的 x 坐标就足够了。开始坐标总是小于结束坐标。
* 一支弓箭可以沿着 x 轴从不同点完全垂直地射出。在坐标 x 处射出一支箭，若有一个气球的直径的开始和结束坐标为 x_start, x_end,
* 且满足  $x_{start} \leq x \leq x_{end}$ ，则该气球会被引爆。可以射出的弓箭的数量没有限制。
* 弓箭一旦被射出之后，可以无限地前进。我们想找到使得所有气球全部被引爆，所需的弓箭的最小数量。
*
* 解题思路:
* 这是区间染色问题的一个变种，相当于求最少需要多少个点才能覆盖所有区间。
* 我们可以按照区间的右端点进行排序，然后每次选择当前区间的右端点作为箭的发射位置，
* 这样可以尽可能多地引爆后面的气球。
*
* @param points 气球的坐标列表，每个气球表示为 [x_start, x_end]
* @return 所需的最小弓箭数量
*/
public static int findMinArrowShots(int[][] points) {
    if (points == null || points.length == 0) {
        return 0;
    }

    // 按照右端点排序
    Arrays.sort(points, Comparator.comparingInt(a -> a[1]));

    int arrows = 1;
    int pos = points[0][1]; // 第一支箭的位置

    for (int i = 1; i < points.length; i++) {
        // 如果当前气球的左端点大于箭的位置，需要新的箭
        if (points[i][0] > pos) {
            arrows++;
            pos = points[i][1];
        }
    }

    return arrows;
}

/**
```

```

* LeetCode 435. 无重叠区间
* 题目链接: https://leetcode-cn.com/problems/non-overlapping-intervals/
*
* 问题描述:
* 给定一个区间的集合，找到需要移除区间的最小数量，使得剩余区间互不重叠。
*
* 解题思路:
* 这个问题可以转换为找到最大不重叠区间数，然后用总区间数减去这个值。
* 求最大不重叠区间数的方法与区间染色类似，我们按照区间的右端点排序，
* 然后贪心选择不重叠的区间。
*
* @param intervals 区间列表，每个区间是一个 int 数组 [start, end]
* @return 需要移除的最小区间数量
*/
public static int eraseOverlapIntervals(int[][] intervals) {
    if (intervals == null || intervals.length == 0) {
        return 0;
    }

    // 按照右端点排序
    Arrays.sort(intervals, Comparator.comparingInt(a -> a[1]));

    int count = 1; // 最大不重叠区间数
    int end = intervals[0][1];

    for (int i = 1; i < intervals.length; i++) {
        if (intervals[i][0] >= end) {
            count++;
            end = intervals[i][1];
        }
    }

    return intervals.length - count;
}

/**
* LeetCode 253. 会议室 II
* 题目链接: https://leetcode-cn.com/problems/meeting-rooms-ii/
*
* 问题描述:
* 给定一个会议时间安排的数组，每个会议时间都会包括开始和结束的时间 [[s1, e1], [s2, e2], ...] (si < ei)，
* 为避免会议冲突，同时要考虑充分利用会议室资源，请你计算至少需要多少间会议室，才能满足这些会

```

议安排。

```
*  
* 解题思路:  
* 我们可以将所有的开始时间和结束时间分别排序，然后使用双指针的方法来计算所需的会议室数量。  
* 每当一个会议开始时，我们需要一个新的会议室；每当一个会议结束时，我们释放一个会议室。  
* 我们只需要跟踪当前正在进行的会议数量，最大值即为所需的会议室数量。  
*  
* @param intervals 会议时间列表，每个会议是一个 int 数组 [start, end]  
* @return 所需的最少会议室数量  
*/  
  
public static int minMeetingRooms(int[][] intervals) {  
    if (intervals == null || intervals.length == 0) {  
        return 0;  
    }  
  
    // 分别提取开始时间和结束时间并排序  
    int[] startTimes = new int[intervals.length];  
    int[] endTimes = new int[intervals.length];  
  
    for (int i = 0; i < intervals.length; i++) {  
        startTimes[i] = intervals[i][0];  
        endTimes[i] = intervals[i][1];  
    }  
  
    Arrays.sort(startTimes);  
    Arrays.sort(endTimes);  
  
    int i = 0, j = 0;  
    int maxRooms = 0, currentRooms = 0;  
  
    while (i < startTimes.length && j < endTimes.length) {  
        // 如果当前会议开始时间小于结束时间，需要一个新的会议室  
        if (startTimes[i] < endTimes[j]) {  
            currentRooms++;  
            maxRooms = Math.max(maxRooms, currentRooms);  
            i++;  
        }  
        // 否则，释放一个会议室  
        else {  
            currentRooms--;  
            j++;  
        }  
    }  
}
```

```

    return maxRooms;
}

// 测试代码
public static void main(String[] args) {
    try {
        // 测试区间染色算法
        int[][] intervals1 = {{1, 4}, {2, 5}, {7, 9}, {8, 10}};
        System.out.println("区间染色最少次数: " + intervalColoring(intervals1)); // 应该输出
2

        // 测试 LeetCode 452
        int[][] points = {{10, 16}, {2, 8}, {1, 6}, {7, 12}};
        System.out.println("最少需要的箭数量: " + findMinArrowShots(points)); // 应该输出 2

        // 测试 LeetCode 435
        int[][] intervals2 = {{1, 2}, {2, 3}, {3, 4}, {1, 3}};
        System.out.println("需要移除的最小区间数量: " + eraseOverlapIntervals(intervals2));
// 应该输出 1

        // 测试 LeetCode 253
        int[][] intervals3 = {{0, 30}, {5, 10}, {15, 20}};
        System.out.println("最少需要的会议室数量: " + minMeetingRooms(intervals3)); // 应该
输出 2
    } catch (Exception e) {
        System.err.println("错误: " + e.getMessage());
        e.printStackTrace();
    }
}
}

```

=====

文件: interval_coloring.cpp

=====

```
// -*- coding: utf-8 -*-
/*
区间染色算法（结合贪心）

```

问题描述:

给定多个区间，每个区间代表一个需要染色的区域，每次可以选择一个区间进行染色，求最少需要多少次染色才能覆盖所有给定的区间。

贪心策略：

按照区间的右端点进行排序，每次选择能够覆盖当前未染色区域的最右端点的区间。

时间复杂度： $O(n \log n)$ ，其中 n 是区间的数量，主要是排序的时间复杂度

空间复杂度： $O(1)$ ，只需要常量级的额外空间

相关题目：

1. LeetCode 435. 无重叠区间
 2. LeetCode 452. 用最少量的箭引爆气球
 3. LeetCode 253. 会议室 II
- */

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <stdexcept>

using namespace std;

/***
 * 区间染色算法实现
 *
 * @param intervals 区间列表，每个区间是一个 pair<int, int> (start, end)
 * @return 最少需要的染色次数
 * @throws invalid_argument 当输入无效时抛出异常
 */
int interval_coloring(vector<pair<int, int>> intervals) {
    // 参数验证
    if (intervals.empty()) {
        return 0;
    }

    for (const auto& interval : intervals) {
        if (interval.first > interval.second) {
            throw invalid_argument("每个区间必须满足 start <= end");
        }
    }

    // 按照区间的右端点进行排序
    sort(intervals.begin(), intervals.end(), [] (const pair<int, int>& a, const pair<int, int>& b) {
        return a.second < b.second;
    });
}
```

```

});
```

```

// 贪心选择
int count = 1; // 至少需要一次染色
int end = intervals[0].second; // 当前已经覆盖到的最右端点

for (size_t i = 1; i < intervals.size(); ++i) {
    // 如果当前区间的左端点大于已经覆盖到的最右端点，需要新的染色
    if (intervals[i].first > end) {
        count++;
        end = intervals[i].second;
    }
}

return count;
}
```

```

/***
 * LeetCode 452. 用最少量的箭引爆气球
 * 题目链接: https://leetcode-cn.com/problems/minimum-number-of-arrows-to-burst-balloons/
 *
 * 问题描述:
 * 在二维空间中有许多球形的气球。对于每个气球，提供的输入是水平方向上，气球直径的开始和结束坐标。
 * 由于它是水平的，所以 y 坐标并不重要，因此只要知道开始和结束的 x 坐标就足够了。开始坐标总是小于结束坐标。
 * 一支弓箭可以沿着 x 轴从不同点完全垂直地射出。在坐标 x 处射出一支箭，若有一个气球的直径的开始和结束坐标为 x_start, x_end,
 * 且满足  $x_{start} \leq x \leq x_{end}$ ，则该气球会被引爆。可以射出的弓箭的数量没有限制。
 * 弓箭一旦被射出之后，可以无限地前进。我们想找到使得所有气球全部被引爆，所需的弓箭的最小数量。
 *
 * 解题思路:
 * 这是区间染色问题的一个变种，相当于求最少需要多少个点才能覆盖所有区间。
 * 我们可以按照区间的右端点进行排序，然后每次选择当前区间的右端点作为箭的发射位置，
 * 这样可以尽可能多地引爆后面的气球。
 *
 * @param points 气球的坐标列表，每个气球表示为 [x_start, x_end]
 * @return 所需的最小弓箭数量
 */
int findMinArrowShots(vector<vector<int>>& points) {
    if (points.empty()) {
        return 0;
    }

    int count = 1;
    int end = points[0][1];
    for (int i = 1; i < points.size(); ++i) {
        if (points[i][0] > end) {
            count++;
            end = points[i][1];
        } else if (points[i][1] > end) {
            end = points[i][1];
        }
    }

    return count;
}

```

```

// 按照右端点排序
sort(points.begin(), points.end(), [] (const vector<int>& a, const vector<int>& b) {
    return a[1] < b[1];
});

int arrows = 1;
int pos = points[0][1]; // 第一支箭的位置

for (size_t i = 1; i < points.size(); ++i) {
    // 如果当前气球的左端点大于箭的位置，需要新的箭
    if (points[i][0] > pos) {
        arrows++;
        pos = points[i][1];
    }
}

return arrows;
}

/***
 * LeetCode 435. 无重叠区间
 * 题目链接: https://leetcode-cn.com/problems/non-overlapping-intervals/
 *
 * 问题描述:
 * 给定一个区间的集合，找到需要移除区间的最小数量，使得剩余区间互不重叠。
 *
 * 解题思路:
 * 这个问题可以转换为找到最大不重叠区间数，然后用总区间数减去这个值。
 * 求最大不重叠区间数的方法与区间染色类似，我们按照区间的右端点排序，
 * 然后贪心选择不重叠的区间。
 *
 * @param intervals 区间列表，每个区间是一个 vector<int> [start, end]
 * @return 需要移除的最小区间数量
 */
int eraseOverlapIntervals(vector<vector<int>>& intervals) {
    if (intervals.empty()) {
        return 0;
    }

    // 按照右端点排序
    sort(intervals.begin(), intervals.end(), [] (const vector<int>& a, const vector<int>& b) {
        return a[1] < b[1];
    });
}

```

```

int count = 1; // 最大不重叠区间数
int end = intervals[0][1];

for (size_t i = 1; i < intervals.size(); ++i) {
    if (intervals[i][0] >= end) {
        count++;
        end = intervals[i][1];
    }
}

return intervals.size() - count;
}

/***
 * LeetCode 253. 会议室 II
 * 题目链接: https://leetcode-cn.com/problems/meeting-rooms-ii/
 *
 * 问题描述:
 * 给定一个会议时间安排的数组，每个会议时间都会包括开始和结束的时间 [[s1, e1], [s2, e2], ...] (si < ei)，
 * 为避免会议冲突，同时要考虑充分利用会议室资源，请你计算至少需要多少间会议室，才能满足这些会议安排。
 *
 * 解题思路:
 * 我们可以将所有的开始时间和结束时间分别排序，然后使用双指针的方法来计算所需的会议室数量。
 * 每当一个会议开始时，我们需要一个新的会议室；每当一个会议结束时，我们释放一个会议室。
 * 我们只需要跟踪当前正在进行的会议数量，最大值即为所需的会议室数量。
 *
 * @param intervals 会议时间列表，每个会议是一个 vector<int> [start, end]
 * @return 所需的最少会议室数量
 */
int minMeetingRooms(vector<vector<int>>& intervals) {
    if (intervals.empty()) {
        return 0;
    }

    // 分别提取开始时间和结束时间并排序
    vector<int> start_times, end_times;
    for (const auto& interval : intervals) {
        start_times.push_back(interval[0]);
        end_times.push_back(interval[1]);
    }
}

```

```

sort(start_times.begin(), start_times.end());
sort(end_times.begin(), end_times.end());

int i = 0, j = 0;
int max_rooms = 0, current_rooms = 0;

while (i < start_times.size() && j < end_times.size()) {
    // 如果当前会议开始时间小于结束时间，需要一个新的会议室
    if (start_times[i] < end_times[j]) {
        current_rooms++;
        max_rooms = max(max_rooms, current_rooms);
        i++;
    }
    // 否则，释放一个会议室
    else {
        current_rooms--;
        j++;
    }
}

return max_rooms;
}

// 测试代码
int main() {
    try {
        // 测试区间染色算法
        vector<pair<int, int>> intervals1 = {{1, 4}, {2, 5}, {7, 9}, {8, 10}};
        cout << "区间染色最少次数: " << interval_coloring(intervals1) << endl; // 应该输出 2

        // 测试 LeetCode 452
        vector<vector<int>> points = {{10, 16}, {2, 8}, {1, 6}, {7, 12}};
        cout << "最少需要的箭数量: " << findMinArrowShots(points) << endl; // 应该输出 2

        // 测试 LeetCode 435
        vector<vector<int>> intervals2 = {{1, 2}, {2, 3}, {3, 4}, {1, 3}};
        cout << "需要移除的最小区间数量: " << eraseOverlapIntervals(intervals2) << endl; // 应该
输出 1

        // 测试 LeetCode 253
        vector<vector<int>> intervals3 = {{0, 30}, {5, 10}, {15, 20}};
        cout << "最少需要的会议室数量: " << minMeetingRooms(intervals3) << endl; // 应该输出 2
    }
}

```

```
    } catch (const exception& e) {
        cerr << "错误: " << e.what() << endl;
    }

    return 0;
}
```

文件: interval_coloring.py

```
# -*- coding: utf-8 -*-
"""

```

区间染色算法 (结合贪心)

问题描述:

给定多个区间，每个区间代表一个需要染色的区域，每次可以选择一个区间进行染色，求最少需要多少次染色才能覆盖所有给定的区间。

贪心策略:

按照区间的右端点进行排序，每次选择能够覆盖当前未染色区域的最右端点的区间。

时间复杂度: $O(n \log n)$ ，其中 n 是区间的数量，主要是排序的时间复杂度

空间复杂度: $O(1)$ ，只需要常量级的额外空间

相关题目:

1. LeetCode 435. 无重叠区间
 2. LeetCode 452. 用最少量的箭引爆气球
 3. LeetCode 253. 会议室 II
- ```
"""

```

```
def interval_coloring(intervals):
 """

```

区间染色算法实现

Args:

intervals: 区间列表，每个区间是一个元组 (start, end)

Returns:

int: 最少需要的染色次数

Raises:

ValueError: 当输入无效时抛出异常

```

"""
参数验证
if not intervals:
 return 0

for interval in intervals:
 if len(interval) != 2 or interval[0] > interval[1]:
 raise ValueError("每个区间必须是有效的(start, end)元组，且 start <= end")

按照区间的右端点进行排序
sorted_intervals = sorted(intervals, key=lambda x: x[1])

贪心选择
count = 1 # 至少需要一次染色
end = sorted_intervals[0][1] # 当前已经覆盖到的最右端点

for i in range(1, len(sorted_intervals)):
 # 如果当前区间的左端点大于已经覆盖到的最右端点，需要新的染色
 if sorted_intervals[i][0] > end:
 count += 1
 end = sorted_intervals[i][1]

return count

```

```
def interval_coloring_leetcode_452(points):
```

```
"""

```

LeetCode 452. 用最少量的箭引爆气球

题目链接: <https://leetcode-cn.com/problems/minimum-number-of-arrows-to-burst-balloons/>

问题描述:

在二维空间中有许多球形的气球。对于每个气球，提供的输入是水平方向上，气球直径的开始和结束坐标。

由于它是水平的，所以 y 坐标并不重要，因此只要知道开始和结束的 x 坐标就足够了。开始坐标总是小于结束坐标。

一支弓箭可以沿着 x 轴从不同点完全垂直地射出。在坐标 x 处射出一支箭，若有一个气球的直径的开始和结束坐标为 x\_start, x\_end,

且满足  $x_{start} \leq x \leq x_{end}$ ，则该气球会被引爆。可以射出的弓箭的数量没有限制。

弓箭一旦被射出之后，可以无限地前进。我们想找到使得所有气球全部被引爆，所需的弓箭的最小数量。

解题思路:

这是区间染色问题的一个变种，相当于求最少需要多少个点才能覆盖所有区间。

我们可以按照区间的右端点进行排序，然后每次选择当前区间的右端点作为箭的发射位置，

这样可以尽可能多地引爆后面的气球。

Args:

points: 气球的坐标列表, 每个气球表示为 [x\_start, x\_end]

Returns:

int: 所需的最小弓箭数量

"""

```
if not points:
```

```
 return 0
```

```
按照右端点排序
```

```
points.sort(key=lambda x: x[1])
```

```
arrows = 1
```

```
pos = points[0][1] # 第一支箭的位置
```

```
for i in range(1, len(points)):
```

```
 # 如果当前气球的左端点大于箭的位置, 需要新的箭
```

```
 if points[i][0] > pos:
```

```
 arrows += 1
```

```
 pos = points[i][1]
```

```
return arrows
```

```
def interval_coloring_leetcode_435(intervals):
```

"""

LeetCode 435. 无重叠区间

题目链接: <https://leetcode-cn.com/problems/non-overlapping-intervals/>

问题描述:

给定一个区间的集合, 找到需要移除区间的最小数量, 使得剩余区间互不重叠。

解题思路:

这个问题可以转换为找到最大不重叠区间数, 然后用总区间数减去这个值。

求最大不重叠区间数的方法与区间染色类似, 我们按照区间的右端点排序, 然后贪心选择不重叠的区间。

Args:

intervals: 区间列表, 每个区间是一个列表 [start, end]

Returns:

```

int: 需要移除的最小区间数量
"""

if not intervals:
 return 0

按照右端点排序
intervals.sort(key=lambda x: x[1])

count = 1 # 最大不重叠区间数
end = intervals[0][1]

for i in range(1, len(intervals)):
 if intervals[i][0] >= end:
 count += 1
 end = intervals[i][1]

return len(intervals) - count

```

```

def interval_coloring_leetcode_253(intervals):
"""

LeetCode 253. 会议室 II
题目链接: https://leetcode-cn.com/problems/meeting-rooms-ii/

```

问题描述:

给定一个会议时间安排的数组，每个会议时间都会包括开始和结束的时间  $[[s_1, e_1], [s_2, e_2], \dots]$  ( $s_i < e_i$ )，

为避免会议冲突，同时要考虑充分利用会议室资源，请你计算至少需要多少间会议室，才能满足这些会议安排。

解题思路:

我们可以将所有的开始时间和结束时间分别排序，然后使用双指针的方法来计算所需的会议室数量。

每当一个会议开始时，我们需要一个新的会议室；每当一个会议结束时，我们释放一个会议室。

我们只需要跟踪当前正在进行的会议数量，最大值即为所需的会议室数量。

Args:

intervals: 会议时间列表，每个会议是一个列表 [start, end]

Returns:

int: 所需的最少会议室数量

"""

if not intervals:
 return 0

```

分别提取开始时间和结束时间并排序
start_times = sorted([interval[0] for interval in intervals])
end_times = sorted([interval[1] for interval in intervals])

i = j = 0
max_rooms = current_rooms = 0

while i < len(start_times) and j < len(end_times):
 # 如果当前会议开始时间小于结束时间，需要一个新的会议室
 if start_times[i] < end_times[j]:
 current_rooms += 1
 max_rooms = max(max_rooms, current_rooms)
 i += 1
 # 否则，释放一个会议室
 else:
 current_rooms -= 1
 j += 1

return max_rooms

测试代码
if __name__ == "__main__":
 # 测试区间染色算法
 intervals1 = [(1, 4), (2, 5), (7, 9), (8, 10)]
 print(f"区间染色最少次数: {interval_coloring(intervals1)}") # 应该输出 2

 # 测试 LeetCode 452
 points = [[10, 16], [2, 8], [1, 6], [7, 12]]
 print(f"最少需要的箭数量: {interval_coloring_leetcode_452(points)}") # 应该输出 2

 # 测试 LeetCode 435
 intervals2 = [[1, 2], [2, 3], [3, 4], [1, 3]]
 print(f"需要移除的最小区间数量: {interval_coloring_leetcode_435(intervals2)}") # 应该输出 1

 # 测试 LeetCode 253
 intervals3 = [[0, 30], [5, 10], [15, 20]]
 print(f"最少需要的会议室数量: {interval_coloring_leetcode_253(intervals3)}") # 应该输出 2
=====
```

```
=====
/**
 * Luogu P1198 [JSOI2008] 最大数
 * 题目链接: https://www.luogu.com.cn/problem/P1198
 *
 * 题目描述:
 * 维护一个数列，支持两种操作：
 * 1. 查询操作 Q L: 查询当前数列中末尾 L 个数中的最大数
 * 2. 插入操作 A n: 将 n 加上最近一次查询操作的答案 t (初始为 0)，对 D 取模后插入数列末尾
 *
 * 解题思路:
 * 使用线段树来维护数列，支持区间最大值查询和单点更新操作。
 * 由于数列是动态增长的，我们可以预先开一个足够大的线段树数组，
 * 用一个指针记录当前数列的实际长度。
 *
 * 时间复杂度分析:
 * - 建树: O(n)
 * - 单点更新: O(log n)
 * - 区间查询: O(log n)
 *
 * 空间复杂度: O(4n)
 */
```

```
// 由于系统环境限制，此处仅提供 C++ 线段树类的声明和主要方法签名
// 实际使用时需要包含适当的头文件并实现所有方法
```

```
const long long INF = -9223372036854775807LL - 1;

// 线段树类，用于维护区间最大值
class SegmentTree {
private:
 int n; // 数组大小
 long long* max_val; // 线段树数组，存储区间最大值
 long long* arr; // 原始数组
 int size; // 当前数列的实际长度

public:
 /**
 * 构造函数
 * @param size 线段树大小
 */
 SegmentTree(int size);
```

```

/**
 * 析构函数
 */
~SegmentTree();

/**
 * 向上更新节点信息 - 最大值信息的汇总
 * @param i 当前节点编号
 */
void pushUp(int i);

/**
 * 单点更新 - 在位置 idx 处插入值 val
 * @param idx 要更新的位置
 * @param val 新的值
 * @param l 当前区间左端点
 * @param r 当前区间右端点
 * @param i 当前节点编号
 */
void update(int idx, long long val, int l, int r, int i);

/**
 * 区间最大值查询
 * @param jobl 查询区间左端点
 * @param jobr 查询区间右端点
 * @param l 当前区间左端点
 * @param r 当前区间右端点
 * @param i 当前节点编号
 * @return 区间最大值
 */
long long queryMax(int jobl, int jobr, int l, int r, int i);

/**
 * 在数列末尾添加一个数
 * @param val 要添加的值
 */
void add(long long val);

/**
 * 查询末尾 L 个数中的最大值
 * @param L 查询的个数
 * @return 最大值
 */

```

```

long long queryLastL(int L);

/**
 * 获取当前数列长度
 * @return 数列长度
 */
int getSize();

};

// 测试代码
// int main() {
// // 示例测试
// // cout << "线段树测试 - Luogu P1198 最大数" << endl;
// // SegmentTree segTree(10);
// // cout << "初始化完成" << endl;
// // return 0;
// }

=====

文件: LuoguP1198_MaxNumber.java
=====

/***
 * Luogu P1198 [JSOI2008] 最大数
 * 题目链接: https://www.luogu.com.cn/problem/P1198
 *
 * 题目描述:
 * 维护一个数列, 支持两种操作:
 * 1. 查询操作 Q L: 查询当前数列中末尾 L 个数中的最大数
 * 2. 插入操作 A n: 将 n 加上最近一次查询操作的答案 t (初始为 0), 对 D 取模后插入数列末尾
 *
 * 解题思路:
 * 使用线段树来维护数列, 支持区间最大值查询和单点更新操作。
 * 由于数列是动态增长的, 我们可以预先开一个足够大的线段树数组,
 * 用一个指针记录当前数列的实际长度。
 *
 * 时间复杂度分析:
 * - 建树: O(n)
 * - 单点更新: O(log n)
 * - 区间查询: O(log n)
 *
 * 空间复杂度: O(4n)
 */

```

```
import java.util.*;
import java.io.*;

public class LuoguP1198_MaxNumber {
 // 线段树类，用于维护区间最大值
 static class SegmentTree {
 private int n; // 数组大小
 private long[] max; // 线段树数组，存储区间最大值
 private long[] arr; // 原始数组
 private int size; // 当前数列的实际长度

 /**
 * 构造函数
 * @param size 线段树大小
 */
 public SegmentTree(int size) {
 this.n = size;
 this.max = new long[4 * size];
 this.arr = new long[size];
 this.size = 0;
 }

 /**
 * 向上更新节点信息 - 最大值信息的汇总
 * @param i 当前节点编号
 */
 private void pushUp(int i) {
 max[i] = Math.max(max[i << 1], max[i << 1 | 1]);
 }

 /**
 * 单点更新 - 在位置 idx 处插入值 val
 * @param idx 要更新的位置
 * @param val 新的值
 * @param l 当前区间左端点
 * @param r 当前区间右端点
 * @param i 当前节点编号
 */
 public void update(int idx, long val, int l, int r, int i) {
 if (l == r) {
 max[i] = val;
 arr[idx] = val;
 }
 }
 }
}
```

```

 } else {
 int mid = (l + r) >> 1;
 if (idx <= mid) {
 update(idx, val, l, mid, i << 1);
 } else {
 update(idx, val, mid + 1, r, i << 1 | 1);
 }
 pushUp(i);
 }
}

/***
 * 区间最大值查询
 * @param jobl 查询区间左端点
 * @param jobr 查询区间右端点
 * @param l 当前区间左端点
 * @param r 当前区间右端点
 * @param i 当前节点编号
 * @return 区间最大值
 */
public long queryMax(int jobl, int jobr, int l, int r, int i) {
 if (jobl <= l && r <= jobr) {
 return max[i];
 }
 int mid = (l + r) >> 1;
 long ans = Long.MIN_VALUE;
 if (jobl <= mid) {
 ans = Math.max(ans, queryMax(jobl, jobr, l, mid, i << 1));
 }
 if (jobr > mid) {
 ans = Math.max(ans, queryMax(jobl, jobr, mid + 1, r, i << 1 | 1));
 }
 return ans;
}

/***
 * 在数列末尾添加一个数
 * @param val 要添加的值
 */
public void add(long val) {
 update(size, val, 0, n - 1, 1);
 size++;
}

```

```
/***
 * 查询末尾 L 个数中的最大值
 * @param L 查询的个数
 * @return 最大值
 */
public long queryLastL(int L) {
 // 查询区间为 [size-L, size-1]
 return queryMax(size - L, size - 1, 0, n - 1, 1);
}

/***
 * 获取当前数列长度
 * @return 数列长度
 */
public int getSize() {
 return size;
}

/***
 * 主函数
 * @param args 命令行参数
 */
public static void main(String[] args) {
 // 为了简化处理，我们使用示例输入
 // 实际使用时应该用：Scanner scanner = new Scanner(System.in);
 String[] inputLines = {
 "10 7",
 "A 1",
 "A 2",
 "A 3",
 "Q 2",
 "A 4",
 "Q 3",
 "A 5",
 "Q 4",
 "A 6",
 "Q 5"
 };
 try {
 // 解析第一行输入

```

```

String[] firstLine = inputLines[0].split(" ");
int M = Integer.parseInt(firstLine[0]); // 操作个数
int D = Integer.parseInt(firstLine[1]); // 取模常数

// 初始化线段树，大小为M足够使用
SegmentTree segTree = new SegmentTree(M);

long lastQueryResult = 0; // 最近一次查询操作的答案，初始为0

// 处理每个操作
for (int i = 1; i <= M; i++) {
 String[] operation = inputLines[i].split(" ");
 char opType = operation[0].charAt(0);

 if (opType == 'A') {
 // 插入操作
 long n = Long.parseLong(operation[1]);
 long val = (n + lastQueryResult) % D;
 segTree.add(val);
 } else if (opType == 'Q') {
 // 查询操作
 int L = Integer.parseInt(operation[1]);
 lastQueryResult = segTree.queryLastL(L);
 System.out.println(lastQueryResult);
 }
}

} catch (Exception e) {
 System.err.println("处理输入时发生错误: " + e.getMessage());
 e.printStackTrace();
}
}
}

```

文件: LuoguP1198\_MaxNumber.py

```

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

import sys

```

"""

Luogu P1198 [JSOI2008] 最大数

题目链接: <https://www.luogu.com.cn/problem/P1198>

题目描述:

维护一个数列, 支持两种操作:

1. 查询操作 Q L: 查询当前数列中末尾 L 个数中的最大数
2. 插入操作 A n: 将 n 加上最近一次查询操作的答案 t (初始为 0), 对 D 取模后插入数列末尾

解题思路:

使用线段树来维护数列, 支持区间最大值查询和单点更新操作。

由于数列是动态增长的, 我们可以预先开一个足够大的线段树数组,  
用一个指针记录当前数列的实际长度。

时间复杂度分析:

- 建树:  $O(n)$
- 单点更新:  $O(\log n)$
- 区间查询:  $O(\log n)$

空间复杂度:  $O(4n)$

"""

```
class SegmentTree:
```

```
 def __init__(self, size):
 """
```

初始化线段树

Args:

size: 线段树大小

"""

```
 self.n = size
```

```
 self.max_val = [float('-inf')] * (4 * size) # 线段树数组, 存储区间最大值
```

```
 self.arr = [0] * size # 原始数组
```

```
 self.size = 0 # 当前数列的实际长度
```

```
 def push_up(self, i):
```

"""

向上更新节点信息 - 最大值信息的汇总

Args:

i: 当前节点编号

"""

```
 self.max_val[i] = max(self.max_val[i << 1], self.max_val[i << 1 | 1])
```

```

def update(self, idx, val, l, r, i):
 """
 单点更新 - 在位置 idx 处插入值 val

 Args:
 idx: 要更新的位置
 val: 新的值
 l: 当前区间左端点
 r: 当前区间右端点
 i: 当前节点编号
 """

 if l == r:
 self.max_val[i] = val
 self.arr[idx] = val
 else:
 mid = (l + r) >> 1
 if idx <= mid:
 self.update(idx, val, l, mid, i << 1)
 else:
 self.update(idx, val, mid + 1, r, i << 1 | 1)
 self.push_up(i)

def query_max(self, jobl, jobr, l, r, i):
 """
 区间最大值查询

 Args:
 jobl: 查询区间左端点
 jobr: 查询区间右端点
 l: 当前区间左端点
 r: 当前区间右端点
 i: 当前节点编号

 Returns:
 区间最大值
 """

 if jobl <= l and r <= jobr:
 return self.max_val[i]
 mid = (l + r) >> 1
 ans = float('-inf')
 if jobl <= mid:
 ans = max(ans, self.query_max(jobl, jobr, l, mid, i << 1))
 if jobr > mid:
 ans = max(ans, self.query_max(jobl, jobr, mid + 1, r, i << 1 | 1))
 return ans

```

```
 if jobr > mid:
 ans = max(ans, self.query_max(jobl, jobr, mid + 1, r, i << 1 | 1))
 return ans
```

```
def add(self, val):
 """
 在数列末尾添加一个数
 """
```

Args:

    val: 要添加的值

```
 self.update(self.size, val, 0, self.n - 1, 1)
 self.size += 1
```

```
def query_last_l(self, l):
 """
 查询末尾 L 个数中的最大值
 """
```

Args:

    l: 查询的个数

Returns:

    最大值

```
 # 查询区间为 [size-l, size-1]
 return self.query_max(self.size - l, self.size - 1, 0, self.n - 1, 1)
```

```
def get_size(self):
 """
 获取当前数列长度
 """
```

Returns:

    数列长度

```
 return self.size
```

```
def main():
 """主函数"""
 # 为了简化处理，我们使用示例输入
 # 实际使用时应该用: import sys; input = sys.stdin.read
 input_lines = [
 "10 7",
```

```
"A 1",
"A 2",
"A 3",
"Q 2",
"A 4",
"Q 3",
"A 5",
"Q 4",
"A 6",
"Q 5"
]

try:
 # 解析第一行输入
 m, d = map(int, input_lines[0].split()) # 操作个数和取模常数

 # 初始化线段树，大小为 M 足够使用
 seg_tree = SegmentTree(m)

 last_query_result = 0 # 最近一次查询操作的答案，初始为 0

 # 处理每个操作
 for i in range(1, m + 1):
 operation = input_lines[i].split()
 op_type = operation[0]

 if op_type == 'A':
 # 插入操作
 n = int(operation[1])
 val = (n + last_query_result) % d
 seg_tree.add(val)
 elif op_type == 'Q':
 # 查询操作
 l = int(operation[1])
 last_query_result = seg_tree.query_last_l(l)
 print(last_query_result)

 except Exception as e:
 print(f"处理输入时发生错误: {e}", file=sys.stderr)
 raise

测试代码
if __name__ == "__main__":

```

```
main()
```

```
=====
```

文件: LuoguP4198\_BuildingReconstruction.cpp

```
=====
```

```
/**
 * Luogu P4198 楼房重建
 * 题目链接: https://www.luogu.com.cn/problem/P4198
 *
 * 题目描述:
 * 小 A 在平面上(0, 0)点的位置, 第 i 栋楼房可以用一条连接(i, 0)和(i, Hi)的线段表示。
 * 如果这栋楼房上存在一个高度大于 0 的点与(0, 0)的连线没有与之前的线段相交, 那么这栋楼房就被称为是可见的。
 * 每天建筑队会修改一栋楼房的高度, 求每天小 A 能看到多少栋楼房。
 *
 * 解题思路:
 * 这是一个经典的线段树问题。关键在于将问题转化为斜率比较问题。
 * 从原点(0, 0)能看到第 i 栋楼, 当且仅当第 i 栋楼的斜率 H_i/i 大于前面所有楼的斜率。
 * 因此, 我们需要维护区间最大值, 并统计从左到右严格递增的斜率个数。
 *
 * 我们使用线段树来维护每个区间的以下信息:
 * 1. 区间最大值
 * 2. 区间内从左端点开始能看到的楼房数量 (在给定左端点限制斜率的情况下)
 *
 * 时间复杂度分析:
 * - 单点更新: $O(\log n)$
 * - 查询全局可见楼房数: $O(\log n)$
 *
 * 空间复杂度: $O(4n)$
 */
```

```
// 由于系统环境限制, 此处仅提供 C++线段树类的声明和主要方法签名
// 实际使用时需要包含适当的头文件并实现所有方法
```

```
const double INF = 1e100;
```

```
// 线段树节点类
struct Node {
 double maxSlope; // 区间最大斜率
 int visibleCount; // 区间内可见楼房数量

 Node() : maxSlope(0), visibleCount(0) {}
```

```
Node(double maxSlope, int visibleCount) : maxSlope(maxSlope), visibleCount(visibleCount) {}
};

// 线段树类
class SegmentTree {
private:
 int n;
 Node* tree;
 double* slopes; // 存储每个位置的斜率

public:
 /**
 * 构造函数
 * @param size 数组大小
 */
 SegmentTree(int size);

 /**
 * 析构函数
 */
 ~SegmentTree();

 /**
 * 向上更新节点信息
 * @param i 当前节点编号
 */
 void pushUp(int i);

 /**
 * 计算区间[l, r]内从左端点开始，在限制斜率 limit 下可见的楼房数量
 * @param l 区间左端点
 * @param r 区间右端点
 * @param limit 限制斜率
 * @param i 当前节点编号
 * @return 可见楼房数量
 */
 int countVisible(int l, int r, double limit, int i);

 /**
 * 更新节点可见数量
 * @param l 区间左端点
 * @param r 区间右端点
 * @param i 当前节点编号
 */
```

```

*/
void updateVisibleCount(int l, int r, int i);

/**
 * 单点更新
 * @param idx 要更新的位置
 * @param val 新的高度
 * @param l 当前区间左端点
 * @param r 当前区间右端点
 * @param i 当前节点编号
 */
void update(int idx, int val, int l, int r, int i);

/**
 * 查询全局可见楼房数量
 * @return 可见楼房数量
 */
int queryVisibleCount();

};

// 测试代码
// int main() {
// // 示例测试
// // cout << "线段树测试 - Luogu P4198 楼房重建" << endl;
// // SegmentTree segTree(10);
// // cout << "初始化完成" << endl;
// // return 0;
// }

=====

文件: LuoguP4198_BuildingReconstruction.java
=====

/***
 * Luogu P4198 楼房重建
 * 题目链接: https://www.luogu.com.cn/problem/P4198
 *
 * 题目描述:
 * 小 A 在平面上 $(0, 0)$ 点的位置, 第 i 栋楼房可以用一条连接 $(i, 0)$ 和 (i, H_i) 的线段表示。
 * 如果这栋楼房上存在一个高度大于 0 的点与 $(0, 0)$ 的连线没有与之前的线段相交, 那么这栋楼房就被称为是可见的。
 * 每天建筑队会修改一栋楼房的高度, 求每天小 A 能看到多少栋楼房。
 *
 */
```

- \* 解题思路：
- \* 这是一个经典的线段树问题。关键在于将问题转化为斜率比较问题。
- \* 从原点(0, 0)能看到第 i 栋楼，当且仅当第 i 栋楼的斜率  $H_i/i$  大于前面所有楼的斜率。
- \* 因此，我们需要维护区间最大值，并统计从左到右严格递增的斜率个数。
- \*
- \* 我们使用线段树来维护每个区间的以下信息：
- \* 1. 区间最大值
- \* 2. 区间内从左端点开始能看到的楼房数量（在给定左端点限制斜率的情况下）
- \*
- \* 时间复杂度分析：
- \* - 单点更新:  $O(\log n)$
- \* - 查询全局可见楼房数:  $O(\log n)$
- \*
- \* 空间复杂度:  $O(4n)$
- \*/

```

import java.util.*;
import java.io.*;

public class LuoguP4198_BuildingReconstruction {
 // 线段树节点类
 static class Node {
 double maxSlope; // 区间最大斜率
 int visibleCount; // 区间内可见楼房数量

 public Node() {
 this.maxSlope = 0;
 this.visibleCount = 0;
 }

 public Node(double maxSlope, int visibleCount) {
 this.maxSlope = maxSlope;
 this.visibleCount = visibleCount;
 }
 }

 // 线段树类
 static class SegmentTree {
 private int n;
 private Node[] tree;
 private double[] slopes; // 存储每个位置的斜率

 /**

```

```

 * 构造函数
 * @param size 数组大小
 */
public SegmentTree(int size) {
 this.n = size;
 this.tree = new Node[4 * size];
 this.slopes = new double[size + 1];
 for (int i = 0; i < 4 * size; i++) {
 tree[i] = new Node();
 }
}

/***
 * 向上更新节点信息
 * @param i 当前节点编号
 */
private void pushUp(int i) {
 tree[i].maxSlope = Math.max(tree[i << 1].maxSlope, tree[i << 1 | 1].maxSlope);
}

/***
 * 计算区间[l, r]内从左端点开始，在限制斜率 limit 下可见的楼房数量
 * @param l 区间左端点
 * @param r 区间右端点
 * @param limit 限制斜率
 * @param i 当前节点编号
 * @return 可见楼房数量
 */
private int countVisible(int l, int r, double limit, int i) {
 // 如果整个区间最大斜率都不超过限制，那么这个区间内没有可见楼房
 if (tree[i].maxSlope <= limit) {
 return 0;
 }

 // 叶子节点
 if (l == r) {
 return slopes[l] > limit ? 1 : 0;
 }

 int mid = (l + r) >> 1;
 // 如果左子树最大斜率不超过限制，只考虑右子树
 if (tree[i << 1].maxSlope <= limit) {
 return countVisible(mid + 1, r, limit, i << 1 | 1);
 }
}

```

```

 } else {
 // 否则左子树中有可见的，加上右子树中可见的
 return tree[i << 1].visibleCount + countVisible(mid + 1, r, Math.max(limit,
tree[i << 1].maxSlope), i << 1 | 1);
 }
}

/***
 * 更新节点可见数量
 * @param l 区间左端点
 * @param r 区间右端点
 * @param i 当前节点编号
 */
private void updateVisibleCount(int l, int r, int i) {
 if (l == r) {
 tree[i].visibleCount = slopes[1] > 0 ? 1 : 0;
 } else {
 int mid = (l + r) >> 1;
 updateVisibleCount(l, mid, i << 1);
 updateVisibleCount(mid + 1, r, i << 1 | 1);
 tree[i].visibleCount = countVisible(l, r, 0, i);
 }
}

/***
 * 单点更新
 * @param idx 要更新的位置
 * @param val 新的高度
 * @param l 当前区间左端点
 * @param r 当前区间右端点
 * @param i 当前节点编号
 */
public void update(int idx, int val, int l, int r, int i) {
 if (l == r) {
 slopes[idx] = val == 0 ? 0 : (double) val / idx;
 tree[i].maxSlope = slopes[idx];
 tree[i].visibleCount = val > 0 ? 1 : 0;
 } else {
 int mid = (l + r) >> 1;
 if (idx <= mid) {
 update(idx, val, l, mid, i << 1);
 } else {
 update(idx, val, mid + 1, r, i << 1 | 1);
 }
 }
}

```

```

 }
 pushUp(i);
 tree[i].visibleCount = countVisible(l, r, 0, i);
 }
}

/***
 * 查询全局可见楼房数量
 * @return 可见楼房数量
 */
public int queryVisibleCount() {
 return tree[1].visibleCount;
}
}

/***
 * 主函数
 * @param args 命令行参数
 */
public static void main(String[] args) {
 // 为了简化处理，我们使用示例输入
 // 实际使用时应该用：Scanner scanner = new Scanner(System.in);
 String[] inputLines = {
 "5 4",
 "1 1",
 "2 2",
 "3 1",
 "4 3"
 };

 try {
 // 解析第一行输入
 String[] firstLine = inputLines[0].split(" ");
 int N = Integer.parseInt(firstLine[0]); // 楼房数量
 int M = Integer.parseInt(firstLine[1]); // 操作天数

 // 初始化线段树
 SegmentTree segTree = new SegmentTree(N);

 // 处理每天的操作
 for (int i = 1; i <= M; i++) {
 String[] operation = inputLines[i].split(" ");
 int X = Integer.parseInt(operation[0]); // 楼房编号

```

```

 int Y = Integer.parseInt(operation[1]); // 新的高度

 // 更新楼房高度
 segTree.update(X, Y, 1, N, 1);

 // 查询并输出可见楼房数量
 int visibleCount = segTree.queryVisibleCount();
 System.out.println(visibleCount);
 }

} catch (Exception e) {
 System.err.println("处理输入时发生错误：" + e.getMessage());
 e.printStackTrace();
}

}

=====

文件: LuoguP4198_BuildingReconstruction.py
=====

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

"""

Luogu P4198 楼房重建
题目链接: https://www.luogu.com.cn/problem/P4198

```

### 题目描述:

小 A 在平面上  $(0, 0)$  点的位置，第  $i$  栋楼房可以用一条连接  $(i, 0)$  和  $(i, H_i)$  的线段表示。

如果这栋楼房上存在一个高度大于 0 的点与  $(0, 0)$  的连线没有与之前的线段相交，那么这栋楼房就被认为是可见的。

每天建筑队会修改一栋楼房的高度，求每天小 A 能看到多少栋楼房。

### 解题思路:

这是一个经典的线段树问题。关键在于将问题转化为斜率比较问题。

从原点  $(0, 0)$  能看到第  $i$  栋楼，当且仅当第  $i$  栋楼的斜率  $H_i/i$  大于前面所有楼的斜率。

因此，我们需要维护区间最大值，并统计从左到右严格递增的斜率个数。

我们使用线段树来维护每个区间的以下信息：

1. 区间最大值
2. 区间内从左端点开始能看到的楼房数量（在给定左端点限制斜率的情况下）

### 时间复杂度分析:

- 单点更新:  $O(\log n)$
- 查询全局可见楼房数:  $O(\log n)$

空间复杂度:  $O(4n)$

"""

```
import sys
```

```
class Node:
 def __init__(self, max_slope=0, visible_count=0):
 """
```

线段树节点类

Args:

max\_slope: 区间最大斜率

visible\_count: 区间内可见楼房数量

"""

```
 self.max_slope = max_slope # 区间最大斜率
```

```
 self.visible_count = visible_count # 区间内可见楼房数量
```

```
class SegmentTree:
```

```
 def __init__(self, size):
```

"""

初始化线段树

Args:

size: 数组大小

"""

```
 self.n = size
```

```
 self.tree = [Node() for _ in range(4 * size)] # 线段树数组
```

```
 self.slopes = [0.0] * (size + 1) # 存储每个位置的斜率
```

```
 def push_up(self, i):
```

"""

向上更新节点信息

Args:

i: 当前节点编号

"""

```
 self.tree[i].max_slope = max(self.tree[i << 1].max_slope, self.tree[i << 1 |
1].max_slope)
```

```

def count_visible(self, l, r, limit, i):
 """
 计算区间[l, r]内从左端点开始，在限制斜率 limit 下可见的楼房数量

 Args:
 l: 区间左端点
 r: 区间右端点
 limit: 限制斜率
 i: 当前节点编号

 Returns:
 可见楼房数量
 """

 # 如果整个区间最大斜率都不超过限制，那么这个区间内没有可见楼房
 if self.tree[i].max_slope <= limit:
 return 0

 # 叶子节点
 if l == r:
 return 1 if self.slopes[l] > limit else 0

 mid = (l + r) >> 1
 # 如果左子树最大斜率不超过限制，只考虑右子树
 if self.tree[i << 1].max_slope <= limit:
 return self.count_visible(mid + 1, r, limit, i << 1 | 1)
 else:
 # 否则左子树中有可见的，加上右子树中可见的
 return self.tree[i << 1].visible_count + self.count_visible(
 mid + 1, r, max(limit, self.tree[i << 1].max_slope), i << 1 | 1)

def update_visible_count(self, l, r, i):
 """
 更新节点可见数量

 Args:
 l: 区间左端点
 r: 区间右端点
 i: 当前节点编号

 """

 if l == r:
 self.tree[i].visible_count = 1 if self.slopes[l] > 0 else 0
 else:

```

```

 mid = (l + r) >> 1
 self.update_visible_count(l, mid, i << 1)
 self.update_visible_count(mid + 1, r, i << 1 | 1)
 self.tree[i].visible_count = self.count_visible(l, r, 0, i)

def update(self, idx, val, l, r, i):
 """
 单点更新

 Args:
 idx: 要更新的位置
 val: 新的高度
 l: 当前区间左端点
 r: 当前区间右端点
 i: 当前节点编号
 """

 if l == r:
 self.slopes[idx] = 0 if val == 0 else val / idx
 self.tree[i].max_slope = self.slopes[idx]
 self.tree[i].visible_count = 1 if val > 0 else 0
 else:
 mid = (l + r) >> 1
 if idx <= mid:
 self.update(idx, val, l, mid, i << 1)
 else:
 self.update(idx, val, mid + 1, r, i << 1 | 1)
 self.push_up(i)
 self.tree[i].visible_count = self.count_visible(l, r, 0, i)

def query_visible_count(self):
 """
 查询全局可见楼房数量

 Returns:
 可见楼房数量
 """

 return self.tree[1].visible_count

def main():
 """主函数"""
 # 为了简化处理，我们使用示例输入
 # 实际使用时应该用: import sys; input = sys.stdin.read

```

```

input_lines = [
 "5 4",
 "1 1",
 "2 2",
 "3 1",
 "4 3"
]

try:
 # 解析第一行输入
 n, m = map(int, input_lines[0].split()) # 楼房数量和操作天数

 # 初始化线段树
 seg_tree = SegmentTree(n)

 # 处理每天的操作
 for i in range(1, m + 1):
 operation = input_lines[i].split()
 x = int(operation[0]) # 楼房编号
 y = int(operation[1]) # 新的高度

 # 更新楼房高度
 seg_tree.update(x, y, 1, n, 1)

 # 查询并输出可见楼房数量
 visible_count = seg_tree.query_visible_count()
 print(visible_count)

except Exception as e:
 print(f"处理输入时发生错误: {e}", file=sys.stderr)
 raise

测试代码
if __name__ == "__main__":
 main()

```

=====

文件: ProbabilityDP.java

=====

```

// -*- coding: utf-8 -*-
/**
 * 概率/期望 DP 算法

```

```
*
* 问题描述:
* 概率/期望 DP 是一种处理概率和期望问题的动态规划方法。通常使用逆序 DP（从最终状态向初始状态推导），
* 特别适用于马尔可夫决策过程。
*
* 主要特点:
* 1. 状态转移涉及概率
* 2. 通常使用逆序 DP，因为最终状态的概率或期望往往是已知的
* 3. 涉及期望的线性性质和全概率公式
*
* 时间复杂度: 根据具体问题而定，通常为 $O(n^2)$ 或 $O(n^3)$
* 空间复杂度: $O(n^2)$ ，需要存储状态的概率或期望
*
* 相关题目:
* 1. LeetCode 808. 分汤
* 2. LeetCode 688. 骑士在棋盘上的概率
* 3. LeetCode 576. 出界的路径数
*/
```

```
import java.util.HashMap;
import java.util.Map;

public class ProbabilityDP {

 /**
 * LeetCode 808. 分汤
 * 题目链接: https://leetcode-cn.com/problems/soup-servings/
 *
 * 问题描述:
 * 有 A 和 B 两种类型的汤，一开始每种类型的汤有 N 毫升。有四种分配操作：
 * 1. 提供 100ml 的汤 A 和 0ml 的汤 B
 * 2. 提供 75ml 的汤 A 和 25ml 的汤 B
 * 3. 提供 50ml 的汤 A 和 50ml 的汤 B
 * 4. 提供 25ml 的汤 A 和 75ml 的汤 B
 *
 * 当我们把汤分配给某人之后，汤就没有了。每个回合，我们将从四种概率均等的操作中选择一种，
 * 然后分配汤。如果汤的剩余量不足以完成某次操作，我们将尽可能分配。当两种类型的汤都分配完毕时，
 * 停止操作。
 *
 * 返回汤 A 先分配完的概率加上汤 A 和汤 B 同时分配完的概率的一半。
 */
```

```

* 解题思路:
* 使用记忆化搜索进行逆序 DP。定义 dp[i][j] 表示当 A 有 i 毫升，B 有 j 毫升时，所求的概率。
* 最终状态是：当 i=0 且 j>0 时，概率为 1；当 i=0 且 j=0 时，概率为 0.5；当 i>0 且 j=0 时，概率为 0。
*
* @param n 初始时每种汤的毫升数
* @return 所求的概率
*/
public static double soupServings(int n) {
 // 当 n 很大时，概率趋近于 1，可以直接返回 1
 if (n >= 5000) {
 return 1.0;
 }

 // 将毫升数转换为 25 的倍数，减少状态数
 n = (n + 24) / 25;

 // 使用记忆化搜索
 Map<String, Double> memo = new HashMap<>();

 return dfs(n, n, memo);
}

private static double dfs(int a, int b, Map<String, Double> memo) {
 // 边界条件
 if (a <= 0 && b <= 0) {
 return 0.5;
 }
 if (a <= 0) {
 return 1.0;
 }
 if (b <= 0) {
 return 0.0;
 }

 // 生成键以检查是否已经计算过
 String key = a + "," + b;
 if (memo.containsKey(key)) {
 return memo.get(key);
 }

 // 四种操作，每种操作的概率是 0.25
 double prob = 0.25 * (dfs(a-4, b, memo) + dfs(a-3, b-1, memo) +
 dfs(a-2, b-2, memo) + dfs(a-1, b-3, memo));
 memo.put(key, prob);

 return prob;
}

```

```

 memo.put(key, prob);
 return prob;
 }

/***
 * LeetCode 688. 骑士在棋盘上的概率
 * 题目链接: https://leetcode-cn.com/problems/knight-probability-in-chessboard/
 *
 * 问题描述:
 * 在一个 n x n 的国际象棋棋盘上, 一个骑士从单元格 (row, column) 开始, 并尝试进行 k 次移动。
 * 行和列是 0 索引的, 所以左上单元格是 (0, 0), 右下单元格是 (n-1, n-1)。
 * 象棋骑士有 8 种可能的走法, 每次移动在基本方向上是两个单元格, 然后在垂直方向上是一个单元格。
 * 每次骑士要移动时, 它都会随机从 8 种可能的移动中选择一种 (即使棋子会离开棋盘), 然后移动到那里。
 * 骑士继续移动, 直到它走了 k 步或离开了棋盘。
 * 返回骑士在 k 步移动后仍留在棋盘上的概率。
 *
 * 解题思路:
 * 使用动态规划, 定义 dp[step][i][j] 表示骑士在 step 步后位于 (i, j) 的概率。
 * 初始状态是 dp[0][row][column] = 1。
 * 状态转移时, 考虑骑士从当前位置的 8 种可能移动。
 *
 * @param n 棋盘大小
 * @param k 移动次数
 * @param row 起始行
 * @param column 起始列
 * @return 骑士在 k 步移动后仍留在棋盘上的概率
 */
public static double knightProbability(int n, int k, int row, int column) {
 // 定义骑士的 8 种移动方式
 int[][] directions = {{-2, -1}, {-2, 1}, {-1, -2}, {-1, 2},
 {1, -2}, {1, 2}, {2, -1}, {2, 1}};

 // 使用二维 DP 数组, 只需要保存上一步的状态
 double[][] dp_prev = new double[n][n];
 dp_prev[row][column] = 1.0; // 初始位置的概率为 1

 for (int step = 0; step < k; ++step) {
 double[][] dp_curr = new double[n][n];
 for (int i = 0; i < n; ++i) {
 for (int j = 0; j < n; ++j) {
 if (dp_prev[i][j] > 0) {
 // 从当前位置出发, 尝试 8 种移动

```

```

 for (int[] dir : directions) {
 int ni = i + dir[0];
 int nj = j + dir[1];
 // 如果移动后仍在棋盘内
 if (ni >= 0 && ni < n && nj >= 0 && nj < n) {
 dp_curr[ni][nj] += dp_prev[i][j] / 8.0;
 }
 }
 }
}

dp_prev = dp_curr;
}

// 所有留在棋盘上的位置的概率之和
double total = 0.0;
for (double[] rowProb : dp_prev) {
 for (double prob : rowProb) {
 total += prob;
 }
}
return total;
}

/***
 * LeetCode 576. 出界的路径数
 * 题目链接: https://leetcode-cn.com/problems/out-of-boundary-paths/
 *
 * 问题描述:
 * 给你一个 $m \times n$ 的网格和一个球。球的起始坐标为 $(start_row, start_column)$ 。
 * 你可以将球移到在四个方向上相邻的单元格内（可以穿过网格边界到达网格之外）。
 * 你最多可以移动 max_move 次球。
 * 找出并返回可以使球停留在边界之外的路径数量。答案可能非常大，返回对 $10^9 + 7$ 取余后的结果。
 *
 * 解题思路:
 * 使用动态规划，定义 $dp[step][i][j]$ 表示在 $step$ 步后位于 (i, j) 的路径数。
 * 初始状态是 $dp[0][start_row][start_column] = 1$ 。
 * 状态转移时，考虑从当前位置的 4 种可能移动。
 * 统计所有出界的路径数。
 *
 * @param m 网格行数
 * @param n 网格列数
 * @param max_move 最大移动次数
 */

```

```

* @param start_row 起始行
* @param start_column 起始列
* @return 出界的路径数对 10^9 + 7 取余的结果
*/
public static int findPaths(int m, int n, int max_move, int start_row, int start_column) {
 final int MOD = 1000000007;

 // 定义四个方向
 int[][] directions = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

 // 使用二维 DP 数组，只需要保存上一步的状态
 long[][] dp_prev = new long[m][n];
 dp_prev[start_row][start_column] = 1; // 初始位置的路径数为 1

 long out_count = 0;

 for (int step = 0; step < max_move; ++step) {
 long[][] dp_curr = new long[m][n];
 for (int i = 0; i < m; ++i) {
 for (int j = 0; j < n; ++j) {
 if (dp_prev[i][j] > 0) {
 // 从当前位置出发，尝试 4 种移动
 for (int[] dir : directions) {
 int ni = i + dir[0];
 int nj = j + dir[1];
 // 如果移动后出界
 if (ni < 0 || ni >= m || nj < 0 || nj >= n) {
 out_count = (out_count + dp_prev[i][j]) % MOD;
 } else {
 dp_curr[ni][nj] = (dp_curr[ni][nj] + dp_prev[i][j]) % MOD;
 }
 }
 }
 }
 }
 dp_prev = dp_curr;
 }

 return (int) (out_count % MOD);
}

/**
 * 期望 DP 示例：爬楼梯问题的期望版本

```

```

*
* 问题描述:
* 有一个 n 级的楼梯，每次可以爬 1 级或 2 级，但是爬 1 级的概率是 p，爬 2 级的概率是 1-p。
* 求爬到第 n 级楼梯的期望步数。
*
* 解题思路:
* 使用逆序 DP，定义 E[i] 表示从第 i 级爬到第 n 级的期望步数。
* 边界条件是 E[n] = 0。
* 状态转移方程是 E[i] = 1 + p * E[i+1] + (1-p) * E[i+2]。
*
* @param n 楼梯的级数
* @return 爬到第 n 级楼梯的期望步数
*/
public static double expectationDP(int n) {
 if (n <= 0) {
 return 0.0;
 }

 // 假设 p=0.5，每次爬 1 级或 2 级的概率相等
 double p = 0.5;

 // 使用逆序 DP
 double[] E = new double[n + 2]; // E[i] 表示从第 i 级爬到第 n 级的期望步数

 // 从 n-1 级开始逆推
 for (int i = n - 1; i >= 0; --i) {
 if (i + 2 <= n) {
 E[i] = 1.0 + p * E[i+1] + (1 - p) * E[i+2];
 } else {
 // 当 i+2 > n 时，只能爬 1 级
 E[i] = 1.0 + E[i+1];
 }
 }

 return E[0];
}

// 测试代码
public static void main(String[] args) {
 // 测试 LeetCode 808
 System.out.printf("分汤概率 (n=500): %.6f\n", soupServings(500));

 // 测试 LeetCode 688
}

```

```

System.out.printf("骑士在棋盘上的概率: %.6f\n", knightProbability(3, 2, 0, 0)); // 应该
输出 0.0625

// 测试 LeetCode 576
System.out.println("出界的路径数: " + findPaths(2, 2, 2, 0, 0)); // 应该输出 6

// 测试期望 DP 示例
System.out.printf("爬楼梯的期望步数 (n=10): %.6f\n", expectationDP(10));
}

}
=====
```

文件: probability\_dp.cpp

```

// -*- coding: utf-8 -*-
/*
概率/期望 DP 算法
```

#### 问题描述:

概率/期望 DP 是一种处理概率和期望问题的动态规划方法。通常使用逆序 DP (从最终状态向初始状态推导)，特别适用于马尔可夫决策过程。

#### 主要特点:

1. 状态转移涉及概率
2. 通常使用逆序 DP，因为最终状态的概率或期望往往是已知的
3. 涉及期望的线性性质和全概率公式

时间复杂度: 根据具体问题而定，通常为  $O(n^2)$  或  $O(n^3)$

空间复杂度:  $O(n^2)$ ，需要存储状态的概率或期望

#### 相关题目:

1. LeetCode 808. 分汤
  2. LeetCode 688. 骑士在棋盘上的概率
  3. LeetCode 576. 出界的路径数
- \*/

```
#include <iostream>
#include <vector>
#include <unordered_map>
#include <iomanip>

using namespace std;
```

```

/**
 * LeetCode 808. 分汤
 * 题目链接: https://leetcode-cn.com/problems/soup-servings/
 *
 * 问题描述:
 * 有 A 和 B 两种类型的汤，一开始每种类型的汤有 N 毫升。有四种分配操作：
 * 1. 提供 100ml 的汤 A 和 0ml 的汤 B
 * 2. 提供 75ml 的汤 A 和 25ml 的汤 B
 * 3. 提供 50ml 的汤 A 和 50ml 的汤 B
 * 4. 提供 25ml 的汤 A 和 75ml 的汤 B
 *
 * 当我们把汤分配给某人之后，汤就没有了。每个回合，我们将从四种概率均等的操作中选择一种，
 * 然后分配汤。如果汤的剩余量不足以完成某次操作，我们将尽可能分配。当两种类型的汤都分配完毕时，
 * 停止操作。
 *
 * 返回汤 A 先分配完的概率加上汤 A 和汤 B 同时分配完的概率的一半。
 *
 * 解题思路:
 * 使用记忆化搜索进行逆序 DP。定义 dp[i][j] 表示当 A 有 i 毫升，B 有 j 毫升时，所求的概率。
 * 最终状态是：当 i=0 且 j>0 时，概率为 1；当 i=0 且 j=0 时，概率为 0.5；当 i>0 且 j=0 时，概率为 0。
 *
 * @param n 初始时每种汤的毫升数
 * @return 所求的概率
 */
double soupServings(int n) {
 // 当 n 很大时，概率趋近于 1，可以直接返回 1
 if (n >= 5000) {
 return 1.0;
 }

 // 将毫升数转换为 25 的倍数，减少状态数
 n = (n + 24) / 25;

 // 使用记忆化搜索
 unordered_map<int, unordered_map<int, double>> memo;

 function<double(int, int)> dfs = [&](int a, int b) {
 // 边界条件
 if (a <= 0 && b <= 0) {
 return 0.5;
 }
 if (a <= 0) {

```

```

 return 1.0;
 }

 if (b <= 0) {
 return 0.0;
 }

 // 检查是否已经计算过
 if (memo.count(a) && memo[a].count(b)) {
 return memo[a][b];
 }

 // 四种操作，每种操作的概率是 0.25
 double prob = 0.25 * (dfs(a-4, b) + dfs(a-3, b-1) + dfs(a-2, b-2) + dfs(a-1, b-3));
 memo[a][b] = prob;
 return prob;
}

return dfs(n, n);
}

```

```

/**
 * LeetCode 688. 骑士在棋盘上的概率
 * 题目链接: https://leetcode-cn.com/problems/knight-probability-in-chessboard/
 *
 * 问题描述:
 * 在一个 n x n 的国际象棋棋盘上，一个骑士从单元格 (row, column) 开始，并尝试进行 k 次移动。
 * 行和列是 0 索引的，所以左上单元格是 (0, 0)，右下单元格是 (n-1, n-1)。
 * 象棋骑士有 8 种可能的走法，每次移动在基本方向上是两个单元格，然后在垂直方向上是一个单元格。
 * 每次骑士要移动时，它都会随机从 8 种可能的移动中选择一种（即使棋子会离开棋盘），然后移动到那里。
 * 骑士继续移动，直到它走了 k 步或离开了棋盘。
 * 返回骑士在 k 步移动后仍留在棋盘上的概率。
 *
 * 解题思路:
 * 使用动态规划，定义 dp[step][i][j] 表示骑士在 step 步后位于 (i, j) 的概率。
 * 初始状态是 dp[0][row][column] = 1。
 * 状态转移时，考虑骑士从当前位置的 8 种可能移动。
 *
 * @param n 棋盘大小
 * @param k 移动次数
 * @param row 起始行
 * @param column 起始列
 * @return 骑士在 k 步移动后仍留在棋盘上的概率
 */

```

```

double knightProbability(int n, int k, int row, int column) {
 // 定义骑士的 8 种移动方式
 vector<pair<int, int>> directions = {{-2, -1}, {-2, 1}, {-1, -2}, {-1, 2},
 {1, -2}, {1, 2}, {2, -1}, {2, 1}};

 // 使用二维 DP 数组，只需要保存上一步的状态
 vector<vector<double>> dp_prev(n, vector<double>(n, 0.0));
 dp_prev[row][column] = 1.0; // 初始位置的概率为 1

 for (int step = 0; step < k; ++step) {
 vector<vector<double>> dp_curr(n, vector<double>(n, 0.0));
 for (int i = 0; i < n; ++i) {
 for (int j = 0; j < n; ++j) {
 if (dp_prev[i][j] > 0) {
 // 从当前位置出发，尝试 8 种移动
 for (auto& dir : directions) {
 int ni = i + dir.first;
 int nj = j + dir.second;
 // 如果移动后仍在棋盘内
 if (ni >= 0 && ni < n && nj >= 0 && nj < n) {
 dp_curr[ni][nj] += dp_prev[i][j] / 8.0;
 }
 }
 }
 }
 }
 dp_prev = move(dp_curr);
 }

 // 所有留在棋盘上的位置的概率之和
 double total = 0.0;
 for (auto& row : dp_prev) {
 for (double prob : row) {
 total += prob;
 }
 }
 return total;
}

/**
 * LeetCode 576. 出界的路径数
 * 题目链接: https://leetcode-cn.com/problems/out-of-boundary-paths/
 */

```

\* 问题描述:

\* 给你一个  $m \times n$  的网格和一个球。球的起始坐标为  $(start\_row, start\_column)$ 。

\* 你可以将球移到在四个方向上相邻的单元格内（可以穿过网格边界到达网格之外）。

\* 你最多可以移动  $max\_move$  次球。

\* 找出并返回可以使球停留在边界之外的路径数量。答案可能非常大，返回对  $10^9 + 7$  取余后的结果。

\*

\* 解题思路:

\* 使用动态规划，定义  $dp[step][i][j]$  表示在  $step$  步后位于  $(i, j)$  的路径数。

\* 初始状态是  $dp[0][start\_row][start\_column] = 1$ 。

\* 状态转移时，考虑从当前位置的 4 种可能移动。

\* 统计所有出界的路径数。

\*

\* @param m 网格行数

\* @param n 网格列数

\* @param max\_move 最大移动次数

\* @param start\_row 起始行

\* @param start\_column 起始列

\* @return 出界的路径数对  $10^9 + 7$  取余的结果

\*/

```

int findPaths(int m, int n, int max_move, int start_row, int start_column) {
 const int MOD = 1000000007;

 // 定义四个方向
 vector<pair<int, int>> directions = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

 // 使用二维 DP 数组，只需要保存上一步的状态
 vector<vector<long long>> dp_prev(m, vector<long long>(n, 0));
 dp_prev[start_row][start_column] = 1; // 初始位置的路径数为 1

 long long out_count = 0;

 for (int step = 0; step < max_move; ++step) {
 vector<vector<long long>> dp_curr(m, vector<long long>(n, 0));
 for (int i = 0; i < m; ++i) {
 for (int j = 0; j < n; ++j) {
 if (dp_prev[i][j] > 0) {
 // 从当前位置出发，尝试 4 种移动
 for (auto& dir : directions) {
 int ni = i + dir.first;
 int nj = j + dir.second;
 // 如果移动后出界
 if (ni < 0 || ni >= m || nj < 0 || nj >= n) {
 out_count = (out_count + dp_prev[i][j]) % MOD;
 }
 }
 }
 }
 }
 }
}
```

```

 } else {
 dp_curr[ni][nj] = (dp_curr[ni][nj] + dp_prev[i][j]) % MOD;
 }
 }
}

dp_prev = move(dp_curr);
}

return static_cast<int>(out_count % MOD);
}

/***
 * 期望 DP 示例：爬楼梯问题的期望版本
 *
 * 问题描述：
 * 有一个 n 级的楼梯，每次可以爬 1 级或 2 级，但是爬 1 级的概率是 p，爬 2 级的概率是 1-p。
 * 求爬到第 n 级楼梯的期望步数。
 *
 * 解题思路：
 * 使用逆序 DP，定义 E[i] 表示从第 i 级爬到第 n 级的期望步数。
 * 边界条件是 E[n] = 0。
 * 状态转移方程是 E[i] = 1 + p * E[i+1] + (1-p) * E[i+2]。
 *
 * @param n 楼梯的级数
 * @return 爬到第 n 级楼梯的期望步数
 */
double expectationDP(int n) {
 if (n <= 0) {
 return 0.0;
 }

 // 假设 p=0.5，每次爬 1 级或 2 级的概率相等
 double p = 0.5;

 // 使用逆序 DP
 vector<double> E(n + 2, 0.0); // E[i] 表示从第 i 级爬到第 n 级的期望步数

 // 从 n-1 级开始逆推
 for (int i = n - 1; i >= 0; --i) {
 if (i + 2 <= n) {
 E[i] = 1.0 + p * E[i+1] + (1 - p) * E[i+2];
 }
 }
}

```

```

 } else {
 // 当 i+2 > n 时, 只能爬 1 级
 E[i] = 1.0 + E[i+1];
 }
}

return E[0];
}

// 测试代码
int main() {
 // 测试 LeetCode 808
 cout << fixed << setprecision(6);
 cout << "分汤概率 (n=500): " << soupServings(500) << endl;

 // 测试 LeetCode 688
 cout << "骑士在棋盘上的概率: " << knightProbability(3, 2, 0, 0) << endl; // 应该输出 0.0625

 // 测试 LeetCode 576
 cout << "出界的路径数: " << findPaths(2, 2, 2, 0, 0) << endl; // 应该输出 6

 // 测试期望 DP 示例
 cout << "爬楼梯的期望步数 (n=10): " << expectationDP(10) << endl;

 return 0;
}

```

=====

文件: probability\_dp.py

=====

```
-*- coding: utf-8 -*-
"""


```

概率/期望 DP 算法

问题描述:

概率/期望 DP 是一种处理概率和期望问题的动态规划方法。通常使用逆序 DP (从最终状态向初始状态推导), 特别适用于马尔可夫决策过程。

主要特点:

1. 状态转移涉及概率
2. 通常使用逆序 DP, 因为最终状态的概率或期望往往是已知的
3. 涉及期望的线性性质和全概率公式

时间复杂度：根据具体问题而定，通常为  $O(n^2)$  或  $O(n^3)$

空间复杂度： $O(n^2)$ ，需要存储状态的概率或期望

相关题目：

1. LeetCode 808. 分汤
2. LeetCode 688. 骑士在棋盘上的概率
3. LeetCode 576. 出界的路径数

"""

```
import numpy as np
```

```
def probability_dp_soup_servings(n):
```

"""

LeetCode 808. 分汤

题目链接：<https://leetcode-cn.com/problems/soup-servings/>

问题描述：

有 A 和 B 两种类型的汤，一开始每种类型的汤有 N 毫升。有四种分配操作：

1. 提供 100ml 的汤 A 和 0ml 的汤 B
2. 提供 75ml 的汤 A 和 25ml 的汤 B
3. 提供 50ml 的汤 A 和 50ml 的汤 B
4. 提供 25ml 的汤 A 和 75ml 的汤 B

当我们把汤分配给某人之后，汤就没有了。每个回合，我们将从四种概率均等的操作中选择一种，然后分配汤。如果汤的剩余量不足以完成某次操作，我们将尽可能分配。当两种类型的汤都分配完毕时，停止操作。

返回汤 A 先分配完的概率加上汤 A 和汤 B 同时分配完的概率的一半。

解题思路：

使用记忆化搜索进行逆序 DP。定义  $dp[i][j]$  表示当 A 有  $i$  毫升，B 有  $j$  毫升时，所求的概率。

最终状态是：当  $i=0$  且  $j>0$  时，概率为 1；当  $i=0$  且  $j=0$  时，概率为 0.5；当  $i>0$  且  $j=0$  时，概率为 0。

Args:

n: 初始时每种汤的毫升数

Returns:

float: 所求的概率

"""

```
当 n 很大时，概率趋近于 1，可以直接返回 1
```

```
if n >= 5000:
```

```
 return 1.0
```

```

将毫升数转换为 25 的倍数，减少状态数
n = (n + 24) // 25

使用记忆化搜索
memo = {}

def dfs(a, b):
 # 边界条件
 if a <= 0 and b <= 0:
 return 0.5
 if a <= 0:
 return 1.0
 if b <= 0:
 return 0.0

 # 检查是否已经计算过
 if (a, b) in memo:
 return memo[(a, b)]

 # 四种操作，每种操作的概率是 0.25
 prob = 0.25 * (dfs(a-4, b) + dfs(a-3, b-1) + dfs(a-2, b-2) + dfs(a-1, b-3))
 memo[(a, b)] = prob
 return prob

return dfs(n, n)

```

```
def probability_dp_knight_probability(n, k, row, column):
```

```
"""

```

LeetCode 688. 骑士在棋盘上的概率

题目链接: <https://leetcode-cn.com/problems/knight-probability-in-chessboard/>

问题描述:

在一个  $n \times n$  的国际象棋棋盘上，一个骑士从单元格  $(row, column)$  开始，并尝试进行  $k$  次移动。

行和列是 0 索引的，所以左上单元格是  $(0, 0)$ ，右下单元格是  $(n-1, n-1)$ 。

象棋骑士有 8 种可能的走法，如下图所示。每次移动在基本方向上是两个单元格，然后在垂直方向上是一个单元格。

每次骑士要移动时，它都会随机从 8 种可能的移动中选择一种（即使棋子会离开棋盘），然后移动到那里。骑士继续移动，直到它走了  $k$  步或离开了棋盘。

返回骑士在  $k$  步移动后仍留在棋盘上的概率。

解题思路:

使用动态规划，定义  $dp[step][i][j]$  表示骑士在  $step$  步后位于  $(i, j)$  的概率。

初始状态是  $dp[0][row][column] = 1$ 。

状态转移时，考虑骑士从当前位置的 8 种可能移动。

Args:

- n: 棋盘大小
- k: 移动次数
- row: 起始行
- column: 起始列

Returns:

float: 骑士在 k 步移动后仍留在棋盘上的概率

"""

# 定义骑士的 8 种移动方式

```
directions = [(-2, -1), (-2, 1), (-1, -2), (-1, 2),
 (1, -2), (1, 2), (2, -1), (2, 1)]
```

# 使用二维 DP 数组，只需要保存上一步的状态

```
dp_prev = np.zeros((n, n))
```

```
dp_prev[row][column] = 1 # 初始位置的概率为 1
```

```
for _ in range(k):
```

```
 dp_curr = np.zeros((n, n))
```

```
 for i in range(n):
```

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

```
 if dp_prev[i][j] > 0:
```

```
 # 从当前位置出发，尝试 8 种移动
```

```
 for dx, dy in directions:
```

```
 ni, nj = i + dx, j + dy
```

```
 # 如果移动后仍在棋盘内
```

```
 if 0 <= ni < n and 0 <= nj < n:
```

```
 dp_curr[ni][nj] += dp_prev[i][j] / 8.0
```

```
 dp_prev = dp_curr
```

# 所有留在棋盘上的位置的概率之和

```
return np.sum(dp_prev)
```

```
def probability_dp_find_paths(m, n, max_move, start_row, start_column):
```

"""

LeetCode 576. 出界的路径数

题目链接: <https://leetcode-cn.com/problems/out-of-boundary-paths/>

问题描述:

给你一个  $m \times n$  的网格和一个球。球的起始坐标为  $(start\_row, start\_column)$ 。

你可以将球移到在四个方向上相邻的单元格内（可以穿过网格边界到达网格之外）。

你最多可以移动  $max\_move$  次球。

找出并返回可以使球停留在边界之外的路径数量。答案可能非常大，返回对  $10^9 + 7$  取余后的结果。

解题思路：

使用动态规划，定义  $dp[step][i][j]$  表示在  $step$  步后位于  $(i, j)$  的路径数。

初始状态是  $dp[0][start\_row][start\_column] = 1$ 。

状态转移时，考虑从当前位置的 4 种可能移动。

统计所有出界的路径数。

Args:

$m$ : 网格行数

$n$ : 网格列数

$max\_move$ : 最大移动次数

$start\_row$ : 起始行

$start\_column$ : 起始列

Returns:

int: 出界的路径数对  $10^9 + 7$  取余的结果

"""

$MOD = 10**9 + 7$

# 定义四个方向

$directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]$

# 使用二维 DP 数组，只需要保存上一步的状态

$dp\_prev = np.zeros((m, n), dtype=int)$

$dp\_prev[start\_row][start\_column] = 1$  # 初始位置的路径数为 1

$out\_count = 0$

for \_ in range(max\_move):

$dp\_curr = np.zeros((m, n), dtype=int)$

for i in range(m):

for j in range(n):

if  $dp\_prev[i][j] > 0$ :

# 从当前位置出发，尝试 4 种移动

for dx, dy in directions:

$ni, nj = i + dx, j + dy$

# 如果移动后出界

if  $ni < 0$  or  $ni \geq m$  or  $nj < 0$  or  $nj \geq n$ :

$out\_count = (out\_count + dp\_prev[i][j]) \% MOD$

```

 else:
 dp_curr[ni][nj] = (dp_curr[ni][nj] + dp_prev[i][j]) % MOD
 dp_prev = dp_curr

return out_count

```

```
def probability_dp_expectation(n):
```

```
"""
```

期望 DP 示例：爬楼梯问题的期望版本

问题描述：

有一个 n 级的楼梯，每次可以爬 1 级或 2 级，但是爬 1 级的概率是 p，爬 2 级的概率是 1-p。  
求爬到第 n 级楼梯的期望步数。

解题思路：

使用逆序 DP，定义  $E[i]$  表示从第 i 级爬到第 n 级的期望步数。

边界条件是  $E[n] = 0$ 。

状态转移方程是  $E[i] = 1 + p * E[i+1] + (1-p) * E[i+2]$ 。

Args:

n: 楼梯的级数

Returns:

float: 爬到第 n 级楼梯的期望步数

```
"""
```

```
if n <= 0:
```

```
 return 0
```

```
假设 p=0.5，每次爬 1 级或 2 级的概率相等
```

```
p = 0.5
```

```
使用逆序 DP
```

```
E = [0] * (n + 2) # E[i] 表示从第 i 级爬到第 n 级的期望步数
```

```
从 n-1 级开始逆推
```

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

```
 if i + 2 <= n:
```

```
 E[i] = 1 + p * E[i+1] + (1-p) * E[i+2]
```

```
 else:
```

```
 # 当 i+2 > n 时，只能爬 1 级
```

```
 E[i] = 1 + E[i+1]
```

```

return E[0]

测试代码
if __name__ == "__main__":
 # 测试 LeetCode 808
 print(f"分汤概率 (n=500): {probability_dp_soup_servings(500)}")

 # 测试 LeetCode 688
 print(f"骑士在棋盘上的概率: {probability_dp_knight_probability(3, 2, 0, 0)}") # 应该输出
0.0625

 # 测试 LeetCode 576
 print(f"出界的路径数: {probability_dp_find_paths(2, 2, 2, 0, 0)}") # 应该输出 6

 # 测试期望 DP 示例
 print(f"爬楼梯的期望步数 (n=10): {probability_dp_expectation(10)}")

```

=====

文件: QuadrangleOptimization.java

=====

```

// -*- coding: utf-8 -*-
/**
 * 四边形不等式优化 (区间 DP 降阶)
 *
 * 问题描述:
 * 四边形不等式优化是一种用于优化区间 DP 的方法, 可以将时间复杂度从 O(n^3) 降低到 O(n^2)。
 * 当区间 DP 满足四边形不等式性质和决策单调性时, 可以使用这种优化方法。
 *
 * 四边形不等式性质:
 * 对于任意的 a ≤ b ≤ c ≤ d, 有 w(a, d) + w(b, c) ≥ w(a, c) + w(b, d)
 *
 * 决策单调性:
 * 对于区间 DP 问题 dp[i][j] = min{dp[i][k] + dp[k+1][j] + w(i, j)} (i ≤ k < j)
 * 如果决策点 s[i][j] 表示 dp[i][j] 取得最小值时的 k 值,
 * 且满足 s[i][j-1] ≤ s[i][j] ≤ s[i+1][j], 则具有决策单调性。
 *
 * 时间复杂度:
 * 优化前: O(n^3)
 * 优化后: O(n^2)
 *
 * 空间复杂度: O(n^2)

```

\*

\* 相关题目：

\* 1. 石子合并问题

\* 2. LeetCode 312. 戳气球

\* 3. LeetCode 1000. 合并石头的最低成本

\*/

```
import java.util.Arrays;
```

```
public class QuadrangleOptimization {
```

```
/**
```

\* 石子合并问题（四边形不等式优化版）

\*

\* 问题描述：

\* 有 n 堆石子排成一行，每堆石子有一定的数量。现在要将这些石子合并成一堆，

\* 每次只能合并相邻的两堆，合并的代价是两堆石子的总数。求最小的合并代价。

\*

\* 解题思路：

\* 使用区间 DP，定义  $dp[i][j]$  表示合并第 i 到第 j 堆石子的最小代价。

\* 状态转移方程： $dp[i][j] = \min\{dp[i][k] + dp[k+1][j]\} + \text{sum(stones}[i\dots j])$ ，其中  $i \leq k < j$

\*

\* 使用四边形不等式优化，记录最优决策点  $s[i][j]$ ，表示  $dp[i][j]$  取得最小值时的 k 值。

\* 利用决策单调性  $s[i][j-1] \leq s[i][j] \leq s[i+1][j]$  来缩小 k 的搜索范围。

\*

\* @param stones 每堆石子的数量数组

\* @return 最小的合并代价

\*/

```
public static int stoneGame(int[] stones) {
```

```
 int n = stones.length;
```

```
 // 计算前缀和
```

```
 int[] prefixSum = new int[n + 1];
```

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

```
 prefixSum[i] = prefixSum[i-1] + stones[i-1];
```

```
}
```

```
 // dp[i][j] 表示合并第 i 到第 j 堆石子的最小代价
```

```
 int[][] dp = new int[n + 1][n + 1];
```

```
 // s[i][j] 记录 dp[i][j] 取得最小值时的 k 值
```

```
 int[][] s = new int[n + 1][n + 1];
```

```
 // 初始化
```

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

```

 s[i][i] = i;
 }

 // 枚举区间长度
 for (int length = 2; length <= n; ++length) {
 // 枚举起点
 for (int i = 1; i <= n - length + 1; ++i) {
 int j = i + length - 1;
 // 初始化 dp[i][j] 为无穷大
 dp[i][j] = Integer.MAX_VALUE;
 // 利用四边形不等式优化，缩小 k 的搜索范围
 for (int k = s[i][j-1]; k <= s[i+1][j]; ++k) {
 int cost = dp[i][k] + dp[k+1][j] + (prefixSum[j] - prefixSum[i-1]);
 if (cost < dp[i][j]) {
 dp[i][j] = cost;
 s[i][j] = k;
 }
 }
 }
 }

 return dp[1][n];
}

/**
 * LeetCode 312. 戳气球
 * 题目链接: https://leetcode-cn.com/problems/burst-balloons/
 *
 * 问题描述:
 * 有 n 个气球，编号为 0 到 n-1，每个气球上都标有一个数字，这些数字存在数组 nums 中。
 * 现在要求你戳破所有的气球。戳破第 i 个气球，你可以获得 nums[i-1] * nums[i] * nums[i+1] 枚硬币。
 * 这里的 i-1 和 i+1 代表和 i 相邻的两个气球的序号。如果 i-1 或 i+1 超出了数组的边界，那么就当它是一个数字为 1 的气球。
 * 求所能获得硬币的最大数量。
 *
 * 解题思路:
 * 使用区间 DP，定义 dp[i][j] 表示戳破 i 到 j 之间的所有气球（不包括 i 和 j）能获得的最大硬币数。
 * 状态转移方程: $dp[i][j] = \max\{dp[i][k] + dp[k+1][j] + \prod_{i \leq k \leq j} \text{nums}[i]\}$ ，其中 $i < k < j$
 *
 * 虽然这道题不严格满足四边形不等式，但可以使用类似的优化思路。
 *
 */

```

```

* @param nums 气球上的数字数组
* @return 能获得的最大硬币数
*/
public static int maxCoins(int[] nums) {
 int n = nums.length;
 // 添加边界条件，将问题转化为在[0, n+1]之间戳气球
 int[] newNums = new int[n + 2];
 newNums[0] = 1;
 newNums[n + 1] = 1;
 for (int i = 1; i <= n; ++i) {
 newNums[i] = nums[i-1];
 }

 // dp[i][j]表示戳破 i 到 j 之间的所有气球（不包括 i 和 j）能获得的最大硬币数
 int[][] dp = new int[n + 2][n + 2];

 // 枚举区间长度
 for (int length = 2; length <= n + 1; ++length) {
 // 枚举起点
 for (int i = 0; i <= n + 1 - length; ++i) {
 int j = i + length;
 // 枚举最后一个戳破的气球 k
 for (int k = i + 1; k < j; ++k) {
 dp[i][j] = Math.max(dp[i][j], dp[i][k] + dp[k][j] + newNums[i] * newNums[k] *
newNums[j]);
 }
 }
 }

 return dp[0][n + 1];
}

/***
* LeetCode 1000. 合并石头的最低成本
* 题目链接: https://leetcode-cn.com/problems/minimum-cost-to-merge-stones/
*
* 问题描述:
* 有 N 堆石头排成一排，第 i 堆中有 stones[i] 块石头。
* 每次移动需要将连续的 K 堆石头合并为一堆，而这个移动的成本为这 K 堆石头的总数。
* 找出把所有石头合并成一堆的最低成本。如果不可能，返回-1。
*
* 解题思路:
* 首先检查是否能将所有石头合并成一堆，即 $(n-1) \% (k-1) == 0$ 。

```

```

* 使用区间 DP，定义 dp[i][j] 表示将第 i 到第 j 堆石头合并成最少堆数的最小成本。
*
* @param stones 每堆石头的数量数组
* @param k 每次合并的堆数
* @return 最低成本，如果不可能返回-1
*/
public static int mergeStones(int[] stones, int k) {
 int n = stones.length;
 // 检查是否能合并成一堆
 if ((n - 1) % (k - 1) != 0) {
 return -1;
 }

 // 计算前缀和
 int[] prefixSum = new int[n + 1];
 for (int i = 1; i <= n; ++i) {
 prefixSum[i] = prefixSum[i-1] + stones[i-1];
 }

 // dp[i][j] 表示将第 i 到第 j 堆石头合并的最小成本
 int[][] dp = new int[n + 1][n + 1];

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

 // 枚举区间长度
 for (int length = k; length <= n; ++length) {
 // 枚举起点
 for (int i = 1; i <= n - length + 1; ++i) {
 int j = i + length - 1;
 // 枚举分割点，步长为 k-1
 for (int m = i; m < j; m += k-1) {
 if (dp[i][m] != Integer.MAX_VALUE && dp[m+1][j] != Integer.MAX_VALUE) {
 dp[i][j] = Math.min(dp[i][j], dp[i][m] + dp[m+1][j]);
 }
 }
 }
 // 如果可以合并成一堆，加上总和
 if ((j - i) % (k - 1) == 0 && dp[i][j] != Integer.MAX_VALUE) {

```

```

 dp[i][j] += prefixSum[j] - prefixSum[i-1];
 }
}

return dp[1][n];
}

/***
 * 矩阵链乘法问题（四边形不等式优化版）
 *
 * 问题描述：
 * 给定一系列矩阵 A1, A2, …, An，其中 Ai 的维度是 dims[i-1] × dims[i]。
 * 求计算这 n 个矩阵的乘积所需的最少标量乘法次数。
 *
 * 解题思路：
 * 使用区间 DP，定义 dp[i][j] 表示计算矩阵 Ai 到 Aj 的乘积所需的最少标量乘法次数。
 * 状态转移方程：dp[i][j] = min{dp[i][k] + dp[k+1][j] + dims[i-1] * dims[k] * dims[j]}，其中 i
 * ≤ k < j
 *
 * 使用四边形不等式优化，记录最优决策点 s[i][j]，表示 dp[i][j] 取得最小值时的 k 值。
 *
 * @param dims 矩阵的维度数组，其中 dims[i-1] × dims[i] 是第 i 个矩阵的维度
 * @return 最少标量乘法次数
 */
public static int matrixChainMultiplication(int[] dims) {
 int n = dims.length - 1; // 矩阵的数量

 // dp[i][j] 表示计算矩阵 Ai 到 Aj 的乘积所需的最少标量乘法次数
 int[][] dp = new int[n + 1][n + 1];
 // s[i][j] 记录 dp[i][j] 取得最小值时的 k 值
 int[][] s = new int[n + 1][n + 1];

 // 初始化
 for (int i = 1; i <= n; ++i) {
 s[i][i] = i;
 }

 // 枚举区间长度
 for (int length = 2; length <= n; ++length) {
 // 枚举起点
 for (int i = 1; i <= n - length + 1; ++i) {
 int j = i + length - 1;
 dp[i][j] = Integer.MAX_VALUE;
 for (int k = i; k < j; ++k) {
 dp[i][j] = Math.min(dp[i][j], dp[i][k] + dp[k+1][j] + dims[i-1] * dims[k] * dims[j]);
 if (dp[i][j] == dp[i][k] + dp[k+1][j] + dims[i-1] * dims[k] * dims[j]) {
 s[i][j] = k;
 }
 }
 }
 }
}

```

```

 // 初始化 dp[i][j] 为无穷大
 dp[i][j] = Integer.MAX_VALUE;
 // 利用四边形不等式优化，缩小 k 的搜索范围
 for (int k = s[i][j-1]; k <= s[i+1][j]; ++k) {
 int cost = dp[i][k] + dp[k+1][j] + dims[i-1] * dims[k] * dims[j];
 if (cost < dp[i][j]) {
 dp[i][j] = cost;
 s[i][j] = k;
 }
 }
 }

 return dp[1][n];
}

// 测试代码
public static void main(String[] args) {
 // 测试石子合并问题
 int[] stones1 = {4, 1, 1, 4};
 System.out.println("石子合并最小代价: " + stoneGame(stones1)); // 应该输出 18

 // 测试 LeetCode 312
 int[] nums = {3, 1, 5, 8};
 System.out.println("戳气球最大硬币数: " + maxCoins(nums)); // 应该输出 167

 // 测试 LeetCode 1000
 int[] stones2 = {3, 2, 4, 1};
 int k = 2;
 System.out.println("合并石头最低成本: " + mergeStones(stones2, k)); // 应该输出 20

 // 测试矩阵链乘法
 int[] dims = {30, 35, 15, 5, 10, 20, 25};
 System.out.println("矩阵链乘法最少标量乘法次数: " + matrixChainMultiplication(dims)); // 应该输出 15125
}
}
=====

文件: quadrangle_optimization.cpp
=====

// -*- coding: utf-8 -*-

```

文件: quadrangle\_optimization.cpp

// -\*- coding: utf-8 -\*-

```
/*
```

```
四边形不等式优化（区间 DP 降阶）
```

问题描述：

四边形不等式优化是一种用于优化区间 DP 的方法，可以将时间复杂度从  $O(n^3)$  降低到  $O(n^2)$ 。

当区间 DP 满足四边形不等式性质和决策单调性时，可以使用这种优化方法。

四边形不等式性质：

对于任意的  $a \leq b \leq c \leq d$ , 有  $w(a, d) + w(b, c) \geq w(a, c) + w(b, d)$

决策单调性：

对于区间 DP 问题  $dp[i][j] = \min\{dp[i][k] + dp[k+1][j] + w(i, j)\}$  ( $i \leq k < j$ )

如果决策点  $s[i][j]$  表示  $dp[i][j]$  取得最小值时的  $k$  值，

且满足  $s[i][j-1] \leq s[i][j] \leq s[i+1][j]$ ，则具有决策单调性。

时间复杂度：

优化前： $O(n^3)$

优化后： $O(n^2)$

空间复杂度： $O(n^2)$

相关题目：

1. 石子合并问题
2. LeetCode 312. 戳气球
3. LeetCode 1000. 合并石头的最低成本

```
*/
```

```
#include <iostream>
```

```
#include <vector>
```

```
#include <climits>
```

```
using namespace std;
```

```
/**
```

```
* 石子合并问题（四边形不等式优化版）
```

```
*
```

```
* 问题描述：
```

```
* 有 n 堆石子排成一行，每堆石子有一定的数量。现在要将这些石子合并成一堆，
* 每次只能合并相邻的两堆，合并的代价是两堆石子的总数。求最小的合并代价。
```

```
*
```

```
* 解题思路：
```

```
* 使用区间 DP，定义 $dp[i][j]$ 表示合并第 i 到第 j 堆石子的最小代价。
```

```
* 状态转移方程： $dp[i][j] = \min\{dp[i][k] + dp[k+1][j]\} + \sum(stones[i \dots j])$ ，其中 $i \leq k < j$
```

```

*
* 使用四边形不等式优化，记录最优决策点 s[i][j]，表示 dp[i][j] 取得最小值时的 k 值。
* 利用决策单调性 s[i][j-1] ≤ s[i][j] ≤ s[i+1][j] 来缩小 k 的搜索范围。
*
* @param n 石子堆数
* @param stones 每堆石子的数量列表
* @return 最小的合并代价
*/
int stoneGame(int n, vector<int>& stones) {
 // 计算前缀和
 vector<int> prefixSum(n + 1, 0);
 for (int i = 1; i <= n; ++i) {
 prefixSum[i] = prefixSum[i-1] + stones[i-1];
 }

 // dp[i][j] 表示合并第 i 到第 j 堆石子的最小代价
 vector<vector<int>> dp(n + 1, vector<int>(n + 1, 0));
 // s[i][j] 记录 dp[i][j] 取得最小值时的 k 值
 vector<vector<int>> s(n + 1, vector<int>(n + 1, 0));

 // 初始化
 for (int i = 1; i <= n; ++i) {
 s[i][i] = i;
 }

 // 枚举区间长度
 for (int length = 2; length <= n; ++length) {
 // 枚举起点
 for (int i = 1; i <= n - length + 1; ++i) {
 int j = i + length - 1;
 // 初始化 dp[i][j] 为无穷大
 dp[i][j] = INT_MAX;
 // 利用四边形不等式优化，缩小 k 的搜索范围
 for (int k = s[i][j-1]; k <= s[i+1][j]; ++k) {
 int cost = dp[i][k] + dp[k+1][j] + (prefixSum[j] - prefixSum[i-1]);
 if (cost < dp[i][j]) {
 dp[i][j] = cost;
 s[i][j] = k;
 }
 }
 }
 }
}

```

```

 return dp[1][n];
}

/***
 * LeetCode 312. 戳气球
 * 题目链接: https://leetcode-cn.com/problems/burst-balloons/
 *
 * 问题描述:
 * 有 n 个气球, 编号为 0 到 n-1, 每个气球上都标有一个数字, 这些数字存在数组 nums 中。
 * 现在要求你戳破所有的气球。戳破第 i 个气球, 你可以获得 nums[i-1] * nums[i] * nums[i+1] 枚硬币。
 * 这里的 i-1 和 i+1 代表和 i 相邻的两个气球的序号。如果 i-1 或 i+1 超出了数组的边界, 那么就当它是一个
 * 数字为 1 的气球。
 * 求所能获得硬币的最大数量。
 *
 * 解题思路:
 * 使用区间 DP, 定义 dp[i][j] 表示戳破 i 到 j 之间的所有气球 (不包括 i 和 j) 能获得的最大硬币数。
 * 状态转移方程: dp[i][j] = max{dp[i][k] + dp[k][j] + nums[i] * nums[k] * nums[j]}, 其中 i < k < j
 *
 * 虽然这道题不严格满足四边形不等式, 但可以使用类似的优化思路。
 *
 *
 * @param nums 气球上的数字数组
 * @return 能获得的最大硬币数
 */
int maxCoins(vector<int>& nums) {
 int n = nums.size();
 // 添加边界条件, 将问题转化为在 [0, n+1] 之间戳气球
 vector<int> newNums(n + 2, 1);
 for (int i = 1; i <= n; ++i) {
 newNums[i] = nums[i-1];
 }

 // dp[i][j] 表示戳破 i 到 j 之间的所有气球 (不包括 i 和 j) 能获得的最大硬币数
 vector<vector<int>> dp(n + 2, vector<int>(n + 2, 0));

 // 枚举区间长度
 for (int length = 2; length <= n + 1; ++length) {
 // 枚举起点
 for (int i = 0; i <= n + 1 - length; ++i) {
 int j = i + length;
 // 枚举最后一个戳破的气球 k
 for (int k = i + 1; k < j; ++k) {
 dp[i][j] = max(dp[i][j], dp[i][k] + dp[k][j] + newNums[i] * newNums[k] *
newNums[j]);
 }
 }
 }
}

```

```

 }
}

}

return dp[0][n + 1];
}

/***
 * LeetCode 1000. 合并石头的最低成本
 * 题目链接: https://leetcode-cn.com/problems/minimum-cost-to-merge-stones/
 *
 * 问题描述:
 * 有 N 堆石头排成一排, 第 i 堆中有 stones[i] 块石头。
 * 每次移动需要将连续的 K 堆石头合并为一堆, 而这个移动的成本为这 K 堆石头的总数。
 * 找出把所有石头合并成一堆的最低成本。如果不可能, 返回-1。
 *
 * 解题思路:
 * 首先检查是否能将所有石头合并成一堆, 即 $(n-1) \% (k-1) == 0$ 。
 * 使用区间 DP, 定义 $dp[i][j]$ 表示将第 i 到第 j 堆石头合并成最少堆数的最小成本。
 *
 * @param stones 每堆石头的数量数组
 * @param k 每次合并的堆数
 * @return 最低成本, 如果不可能返回-1
 */

int mergeStones(vector<int>& stones, int k) {
 int n = stones.size();
 // 检查是否能合并成一堆
 if ((n - 1) % (k - 1) != 0) {
 return -1;
 }

 // 计算前缀和
 vector<int> prefixSum(n + 1, 0);
 for (int i = 1; i <= n; ++i) {
 prefixSum[i] = prefixSum[i - 1] + stones[i - 1];
 }

 // $dp[i][j]$ 表示将第 i 到第 j 堆石头合并的最小成本
 vector<vector<int>> dp(n + 1, vector<int>(n + 1, 0));

 // 枚举区间长度
 for (int length = k; length <= n; ++length) {
 // 枚举起点

```

```

 for (int i = 1; i <= n - length + 1; ++i) {
 int j = i + length - 1;
 // 初始化 dp[i][j] 为无穷大
 dp[i][j] = INT_MAX;
 // 枚举分割点，步长为 k-1
 for (int m = i; m < j; m += k-1) {
 dp[i][j] = min(dp[i][j], dp[i][m] + dp[m+1][j]);
 }
 // 如果可以合并成一堆，加上总和
 if ((j - i) % (k - 1) == 0) {
 dp[i][j] += prefixSum[j] - prefixSum[i-1];
 }
 }
 }

 return dp[1][n];
}

/***
 * 矩阵链乘法问题（四边形不等式优化版）
 *
 * 问题描述：
 * 给定一系列矩阵 A1, A2, …, An，其中 Ai 的维度是 dims[i-1] × dims[i]。
 * 求计算这 n 个矩阵的乘积所需的最少标量乘法次数。
 *
 * 解题思路：
 * 使用区间 DP，定义 dp[i][j] 表示计算矩阵 Ai 到 Aj 的乘积所需的最少标量乘法次数。
 * 状态转移方程：dp[i][j] = min{dp[i][k] + dp[k+1][j] + dims[i-1] * dims[k] * dims[j]}，其中 i ≤ k < j
 *
 * 使用四边形不等式优化，记录最优决策点 s[i][j]，表示 dp[i][j] 取得最小值时的 k 值。
 *
 * @param dims 矩阵的维度数组，其中 dims[i-1] × dims[i] 是第 i 个矩阵的维度
 * @return 最少标量乘法次数
*/
int matrixChainMultiplication(vector<int>& dims) {
 int n = dims.size() - 1; // 矩阵的数量

 // dp[i][j] 表示计算矩阵 Ai 到 Aj 的乘积所需的最少标量乘法次数
 vector<vector<int>> dp(n + 1, vector<int>(n + 1, 0));
 // s[i][j] 记录 dp[i][j] 取得最小值时的 k 值
 vector<vector<int>> s(n + 1, vector<int>(n + 1, 0));
}

```

```

// 初始化
for (int i = 1; i <= n; ++i) {
 s[i][i] = i;
}

// 枚举区间长度
for (int length = 2; length <= n; ++length) {
 // 枚举起点
 for (int i = 1; i <= n - length + 1; ++i) {
 int j = i + length - 1;
 // 初始化 dp[i][j] 为无穷大
 dp[i][j] = INT_MAX;
 // 利用四边形不等式优化，缩小 k 的搜索范围
 for (int k = s[i][j-1]; k <= s[i+1][j]; ++k) {
 int cost = dp[i][k] + dp[k+1][j] + dims[i-1] * dims[k] * dims[j];
 if (cost < dp[i][j]) {
 dp[i][j] = cost;
 s[i][j] = k;
 }
 }
 }
}

return dp[1][n];
}

// 测试代码
int main() {
 // 测试石子合并问题
 vector<int> stones1 = {4, 1, 1, 4};
 cout << "石子合并最小代价: " << stoneGame(stones1.size(), stones1) << endl; // 应该输出 18

 // 测试 LeetCode 312
 vector<int> nums = {3, 1, 5, 8};
 cout << "戳气球最大硬币数: " << maxCoins(nums) << endl; // 应该输出 167

 // 测试 LeetCode 1000
 vector<int> stones2 = {3, 2, 4, 1};
 int k = 2;
 cout << "合并石头最低成本: " << mergeStones(stones2, k) << endl; // 应该输出 20

 // 测试矩阵链乘法
 vector<int> dims = {30, 35, 15, 5, 10, 20, 25};

```

```
cout << "矩阵链乘法最少标量乘法次数: " << matrixChainMultiplication(dims) << endl; // 应该输出 15125
```

```
 return 0;
}
```

```
=====
```

文件: quadrangle\_optimization.py

```
=====
```

```
-*- coding: utf-8 -*-
```

```
"""
```

四边形不等式优化（区间 DP 降阶）

问题描述:

四边形不等式优化是一种用于优化区间 DP 的方法，可以将时间复杂度从  $O(n^3)$  降低到  $O(n^2)$ 。

当区间 DP 满足四边形不等式性质和决策单调性时，可以使用这种优化方法。

四边形不等式性质:

对于任意的  $a \leq b \leq c \leq d$ , 有  $w(a, d) + w(b, c) \geq w(a, c) + w(b, d)$

决策单调性:

对于区间 DP 问题  $dp[i][j] = \min\{dp[i][k] + dp[k+1][j] + w(i, j)\}$  ( $i \leq k < j$ )

如果决策点  $s[i][j]$  表示  $dp[i][j]$  取得最小值时的  $k$  值，

且满足  $s[i][j-1] \leq s[i][j] \leq s[i+1][j]$ ，则具有决策单调性。

时间复杂度:

优化前:  $O(n^3)$

优化后:  $O(n^2)$

空间复杂度:  $O(n^2)$

相关题目:

1. 石子合并问题
2. LeetCode 312. 戳气球
3. LeetCode 1000. 合并石头的最低成本

```
"""
```

```
import sys
```

```
def stone_game(n, stones):
```

```
 """
```

石子合并问题（四边形不等式优化版）

问题描述：

有  $n$  堆石子排成一行，每堆石子有一定的数量。现在要将这些石子合并成一堆，每次只能合并相邻的两堆，合并的代价是两堆石子的总数。求最小的合并代价。

解题思路：

使用区间 DP，定义  $dp[i][j]$  表示合并第  $i$  到第  $j$  堆石子的最小代价。

状态转移方程： $dp[i][j] = \min\{dp[i][k] + dp[k+1][j]\} + \text{sum(stones}[i\dots j])$ ，其中  $i \leq k < j$

使用四边形不等式优化，记录最优决策点  $s[i][j]$ ，表示  $dp[i][j]$  取得最小值时的  $k$  值。

利用决策单调性  $s[i][j-1] \leq s[i][j] \leq s[i+1][j]$  来缩小  $k$  的搜索范围。

Args:

$n$ : 石子堆数

$stones$ : 每堆石子的数量列表

Returns:

int: 最小的合并代价

"""

# 计算前缀和

```
prefix_sum = [0] * (n + 1)
for i in range(1, n + 1):
 prefix_sum[i] = prefix_sum[i-1] + stones[i-1]
```

#  $dp[i][j]$  表示合并第  $i$  到第  $j$  堆石子的最小代价

```
dp = [[0] * (n + 1) for _ in range(n + 1)]
```

#  $s[i][j]$  记录  $dp[i][j]$  取得最小值时的  $k$  值

```
s = [[0] * (n + 1) for _ in range(n + 1)]
```

# 初始化

```
for i in range(1, n + 1):
 s[i][i] = i
```

# 枚举区间长度

```
for length in range(2, n + 1):
```

# 枚举起点

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

$j = i + length - 1$

# 初始化  $dp[i][j]$  为无穷大

```
dp[i][j] = float('inf')
```

# 利用四边形不等式优化，缩小  $k$  的搜索范围

```
for k in range(s[i][j-1], s[i+1][j] + 1):
```

```
cost = dp[i][k] + dp[k+1][j] + (prefix_sum[j] - prefix_sum[i-1])
```

```

 if cost < dp[i][j]:
 dp[i][j] = cost
 s[i][j] = k

 return dp[1][n]

```

def max\_coins(nums):

"""

LeetCode 312. 戳气球

题目链接: <https://leetcode-cn.com/problems/burst-balloons/>

问题描述:

有  $n$  个气球，编号为 0 到  $n-1$ ，每个气球上都标有一个数字，这些数字存在数组  $\text{nums}$  中。

现在要求你戳破所有的气球。戳破第  $i$  个气球，你可以获得  $\text{nums}[i-1] * \text{nums}[i] * \text{nums}[i+1]$  枚硬币。

这里的  $i-1$  和  $i+1$  代表和  $i$  相邻的两个气球的序号。如果  $i-1$  或  $i+1$  超出了数组的边界，那么就当它是一个数字为 1 的气球。

求所能获得硬币的最大数量。

解题思路:

使用区间 DP，定义  $dp[i][j]$  表示戳破  $i$  到  $j$  之间的所有气球（不包括  $i$  和  $j$ ）能获得的最大硬币数。

状态转移方程:  $dp[i][j] = \max\{dp[i][k] + dp[k][j] + \text{nums}[i] * \text{nums}[k] * \text{nums}[j]\}$ ，其中  $i < k < j$

虽然这道题不严格满足四边形不等式，但可以使用类似的优化思路。

Args:

  nums: 气球上的数字数组

Returns:

  int: 能获得的最大硬币数

"""

n = len(nums)

# 添加边界条件，将问题转化为在 [0, n+1] 之间戳气球

new\_nums = [1] + nums + [1]

#  $dp[i][j]$  表示戳破  $i$  到  $j$  之间的所有气球（不包括  $i$  和  $j$ ）能获得的最大硬币数

dp = [[0] \* (n + 2) for \_ in range(n + 2)]

# 枚举区间长度

for length in range(2, n + 2):

# 枚举起点

for i in range(n + 2 - length):

```

j = i + length
枚举最后一个戳破的气球 k
for k in range(i + 1, j):
 dp[i][j] = max(dp[i][j], dp[i][k] + dp[k][j] + new_nums[i] * new_nums[k] *
new_nums[j])

return dp[0][n + 1]

```

```
def merge_stones(stones, k):
```

```
"""

```

LeetCode 1000. 合并石头的最低成本

题目链接: <https://leetcode-cn.com/problems/minimum-cost-to-merge-stones/>

问题描述:

有 N 堆石头排成一排，第 i 堆中有 stones[i] 块石头。

每次移动需要将连续的 K 堆石头合并为一堆，而这个移动的成本为这 K 堆石头的总数。

找出把所有石头合并成一堆的最低成本。如果不可能，返回-1。

解题思路:

首先检查是否能将所有石头合并成一堆，即  $(n-1) \% (k-1) == 0$ 。

使用区间 DP，定义  $dp[i][j]$  表示将第 i 到第 j 堆石头合并成最少堆数的最小成本。

Args:

stones: 每堆石头的数量数组

k: 每次合并的堆数

Returns:

int: 最低成本，如果不可能返回-1

```
"""

```

```
n = len(stones)
```

# 检查是否能合并成一堆

```
if (n - 1) % (k - 1) != 0:
```

```
 return -1
```

# 计算前缀和

```
prefix_sum = [0] * (n + 1)
```

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

```
 prefix_sum[i] = prefix_sum[i-1] + stones[i-1]
```

#  $dp[i][j]$  表示将第 i 到第 j 堆石头合并的最小成本

```
dp = [[0] * (n + 1) for _ in range(n + 1)]
```

```

枚举区间长度
for length in range(k, n + 1):
 # 枚举起点
 for i in range(1, n - length + 2):
 j = i + length - 1
 # 初始化 dp[i][j] 为无穷大
 dp[i][j] = float('inf')
 # 枚举分割点，步长为 k-1
 for m in range(i, j, k-1):
 dp[i][j] = min(dp[i][j], dp[i][m] + dp[m+1][j])
 # 如果可以合并成一堆，加上总和
 if (j - i) % (k - 1) == 0:
 dp[i][j] += prefix_sum[j] - prefix_sum[i-1]

return dp[1][n]

```

```
def matrix_chain_multiplication(dims):
```

```
"""

```

矩阵链乘法问题（四边形不等式优化版）

问题描述：

给定一系列矩阵  $A_1, A_2, \dots, A_n$ ，其中  $A_i$  的维度是  $\text{dims}[i-1] \times \text{dims}[i]$ 。

求计算这  $n$  个矩阵的乘积所需的最少标量乘法次数。

解题思路：

使用区间 DP，定义  $dp[i][j]$  表示计算矩阵  $A_i$  到  $A_j$  的乘积所需的最少标量乘法次数。

状态转移方程： $dp[i][j] = \min\{dp[i][k] + dp[k+1][j] + \text{dims}[i-1] * \text{dims}[k] * \text{dims}[j]\}$ ，其中  $i \leq k < j$

使用四边形不等式优化，记录最优决策点  $s[i][j]$ ，表示  $dp[i][j]$  取得最小值时的  $k$  值。

Args:

`dims`: 矩阵的维度数组，其中  $\text{dims}[i-1] \times \text{dims}[i]$  是第  $i$  个矩阵的维度

Returns:

`int`: 最少标量乘法次数

```
"""

```

`n = len(dims) - 1` # 矩阵的数量

#  $dp[i][j]$  表示计算矩阵  $A_i$  到  $A_j$  的乘积所需的最少标量乘法次数

`dp = [[0] * (n + 1) for _ in range(n + 1)]`

#  $s[i][j]$  记录  $dp[i][j]$  取得最小值时的  $k$  值

```

s = [[0] * (n + 1) for _ in range(n + 1)]

初始化
for i in range(1, n + 1):
 s[i][i] = i

枚举区间长度
for length in range(2, n + 1):
 # 枚举起点
 for i in range(1, n - length + 2):
 j = i + length - 1
 # 初始化 dp[i][j] 为无穷大
 dp[i][j] = float('inf')
 # 利用四边形不等式优化，缩小 k 的搜索范围
 for k in range(s[i][j-1], s[i+1][j] + 1):
 cost = dp[i][k] + dp[k+1][j] + dims[i-1] * dims[k] * dims[j]
 if cost < dp[i][j]:
 dp[i][j] = cost
 s[i][j] = k

return dp[1][n]

测试代码
if __name__ == "__main__":
 # 测试石子合并问题
 stones = [4, 1, 1, 4]
 print(f"石子合并最小代价: {stone_game(len(stones), stones)}") # 应该输出 18

 # 测试 LeetCode 312
 nums = [3, 1, 5, 8]
 print(f"戳气球最大硬币数: {max_coins(nums)}") # 应该输出 167

 # 测试 LeetCode 1000
 stones = [3, 2, 4, 1]
 k = 2
 print(f"合并石头最低成本: {merge_stones(stones, k)}") # 应该输出 20

 # 测试矩阵链乘法
 dims = [30, 35, 15, 5, 10, 20, 25]
 print(f"矩阵链乘法最少标量乘法次数: {matrix_chain_multiplication(dims)}") # 应该输出 15125
=====
```

文件: search\_dp\_problems.py

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

动态规划相关题目搜索脚本
搜索与 class127 中算法相关的更多题目
"""

import json
import requests
import time
from typing import List, Dict, Any

定义要搜索的关键词列表
KEYWORDS = [
 # 网格路径相关
 "grid path", "minimum path sum", "unique paths", "cherry pickup",
 "robot path", "matrix traversal", "dynamic programming grid",

 # 青蛙跳跃相关
 "frog jump", "jump game", "minimum jumps", "frog crossing",
 "stone bridge", "river crossing", "binary search answer",

 # 子数组乘积相关
 "subarray product", "maximum product", "positive negative product",
 "count subarrays", "prefix sum", "bit manipulation",

 # 子序列相关
 "longest subsequence", "subsequence product", "bitwise AND",
 "longest valid subsequence", "dynamic programming sequence",

 # 排列组合相关
 "arrange plates", "ways to arrange", "dynamic programming counting",
 "combinatorics", "recurrence relation", "matrix exponentiation"
]

定义要搜索的平台
PLATFORMS = [
 "LeetCode", "LintCode", "HackerRank", "Codeforces", "AtCoder",
 "USACO", "Luogu", "CodeChef", "SPOJ", "Project Euler",
 "HackerEarth", "计蒜客", "ZOJ", "MarsCode", "UVa OJ",
]
```

"TimusOJ", "AizuOJ", "Comet OJ", "杭电 OJ", "LOJ", "牛客",  
"acwing", "hdu", "poj", "剑指 Offer"

]

# 预定义的题目数据（由于实际 API 调用受限，这里使用预定义数据）

PREDEFINED\_PROBLEMS = [

# 网格路径类题目

{

    "title": "Cherry Pickup",  
    "platform": "LeetCode",  
    "url": "https://leetcode.cn/problems/cherry-pickup/",  
    "difficulty": "Hard",  
    "tags": ["Dynamic Programming", "Grid", "Matrix"],  
    "description": "从左上角到右下角再返回左上角，收集樱桃的最大数量"

},

{

    "title": "Cherry Pickup II",  
    "platform": "LeetCode",  
    "url": "https://leetcode.cn/problems/cherry-pickup-ii/",  
    "difficulty": "Hard",  
    "tags": ["Dynamic Programming", "Grid", "Matrix"],  
    "description": "两个机器人从顶部出发，收集樱桃的最大数量"

},

{

    "title": "Minimum Path Sum",  
    "platform": "LeetCode",  
    "url": "https://leetcode.cn/problems/minimum-path-sum/",  
    "difficulty": "Medium",  
    "tags": ["Dynamic Programming", "Grid", "Matrix"],  
    "description": "从左上角到右下角的最小路径和"

},

{

    "title": "Unique Paths",  
    "platform": "LeetCode",  
    "url": "https://leetcode.cn/problems/unique-paths/",  
    "difficulty": "Medium",  
    "tags": ["Dynamic Programming", "Math", "Combinatorics"],  
    "description": "从左上角到右下角的不同路径数量"

},

{

    "title": "Unique Paths II",  
    "platform": "LeetCode",  
    "url": "https://leetcode.cn/problems/unique-paths-ii/",

```
"difficulty": "Medium",
"tags": ["Dynamic Programming", "Grid", "Matrix"],
"description": "有障碍物的网格中从左上角到右下角的不同路径数量"
},
{
 "title": "Dungeon Game",
 "platform": "LeetCode",
 "url": "https://leetcode.cn/problems/dungeon-game/",
 "difficulty": "Hard",
 "tags": ["Dynamic Programming", "Binary Search"],
 "description": "骑士从左上角到右下角的最小初始健康值"
},
```

## # 青蛙跳跃类题目

```
{
 "title": "Frog Jump",
 "platform": "LeetCode",
 "url": "https://leetcode.cn/problems/frog-jump/",
 "difficulty": "Hard",
 "tags": ["Dynamic Programming", "Hash Table"],
 "description": "青蛙过河，判断能否到达最后一块石头"
},
{
 "title": "Jump Game",
 "platform": "LeetCode",
 "url": "https://leetcode.cn/problems/jump-game/",
 "difficulty": "Medium",
 "tags": ["Greedy", "Array", "Dynamic Programming"],
 "description": "判断能否从第一个位置跳到最后一个位置"
},
{
 "title": "Jump Game II",
 "platform": "LeetCode",
 "url": "https://leetcode.cn/problems/jump-game-ii/",
 "difficulty": "Medium",
 "tags": ["Greedy", "Array", "Dynamic Programming"],
 "description": "跳到最后一个位置的最少跳跃次数"
},
{
 "title": "Jump Game V",
 "platform": "LeetCode",
 "url": "https://leetcode.cn/problems/jump-game-v/",
 "difficulty": "Hard",
```

```
"tags": ["Dynamic Programming", "Sorting"],
"description": "在数组中跳跃，每次跳跃不能超过固定距离"
,
{
 "title": "Jump Game VI",
 "platform": "LeetCode",
 "url": "https://leetcode.cn/problems/jump-game-vi/",
 "difficulty": "Medium",
 "tags": ["Dynamic Programming", "Queue", "Heap"],
 "description": "在数组中跳跃，每次最多跳 k 步，求最大得分"
,

子数组乘积类题目
{
 "title": "Maximum Product Subarray",
 "platform": "LeetCode",
 "url": "https://leetcode.cn/problems/maximum-product-subarray/",
 "difficulty": "Medium",
 "tags": ["Dynamic Programming", "Array"],
 "description": "乘积最大的连续子数组"
,
{
 "title": "Subarray Product Less Than K",
 "platform": "LeetCode",
 "url": "https://leetcode.cn/problems/subarray-product-less-than-k/",
 "difficulty": "Medium",
 "tags": ["Array", "Sliding Window"],
 "description": "乘积小于 K 的连续子数组数量"
,
{
 "title": "The Number of Products",
 "platform": "Codeforces",
 "url": "https://codeforces.com/problemset/problem/1215/B",
 "difficulty": "Easy",
 "tags": ["Dynamic Programming", "Math"],
 "description": "统计乘积为正和负的子数组数量"
,

子序列类题目
{
 "title": "Longest Increasing Subsequence",
 "platform": "LeetCode",
 "url": "https://leetcode.cn/problems/longest-increasing-subsequence/",
```

```
"difficulty": "Medium",
"tags": ["Dynamic Programming", "Binary Search"],
"description": "最长递增子序列"
},
{
 "title": "Longest Common Subsequence",
 "platform": "LeetCode",
 "url": "https://leetcode.cn/problems/longest-common-subsequence/",
 "difficulty": "Medium",
 "tags": ["Dynamic Programming", "String"],
 "description": "两个字符串的最长公共子序列"
},
{
 "title": "Longest Palindromic Subsequence",
 "platform": "LeetCode",
 "url": "https://leetcode.cn/problems/longest-palindromic-subsequence/",
 "difficulty": "Medium",
 "tags": ["Dynamic Programming", "String"],
 "description": "最长回文子序列"
},
```

## # 排列组合类题目

```
{
 "title": "Knight Dialer",
 "platform": "LeetCode",
 "url": "https://leetcode.cn/problems/knight-dialer/",
 "difficulty": "Medium",
 "tags": ["Dynamic Programming", "Matrix Exponentiation"],
 "description": "骑士在电话垫上跳跃，计算不同长度的数字序列数量"
},
{
 "title": "Domino and Tromino Tiling",
 "platform": "LeetCode",
 "url": "https://leetcode.cn/problems/domino-and-tromino-tiling/",
 "difficulty": "Medium",
 "tags": ["Dynamic Programming"],
 "description": "用多米诺骨牌和托米诺骨牌铺满 2*n 的面板"
},
```

## # 其他平台题目

```
{
 "title": "方格取数",
 "platform": "牛客网",
```

```
"url": "https://ac.nowcoder.com/acm/problem/14552",
"difficulty": "Medium",
"tags": ["Dynamic Programming", "Grid"],
"description": "与摘樱桃问题非常相似，两个人从左上角出发到右下角取数"
},
{
 "title": "Walking on a Grid",
 "platform": "UVa OJ",
 "url":
"https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=1854",
 "difficulty": "Hard",
 "tags": ["Dynamic Programming", "Grid"],
 "description": "在网格中行走，有负数，最多允许 k 次负数"
},
{
 "title": "滑雪",
 "platform": "洛谷",
 "url": "https://www.luogu.com.cn/problem/P1434",
 "difficulty": "Medium",
 "tags": ["Dynamic Programming", "DFS", "BFS"],
 "description": "寻找最长滑雪路径，每步只能滑向相邻四个方向且高度更低的位置"
},
{
 "title": "种树",
 "platform": "洛谷",
 "url": "https://www.luogu.com.cn/problem/P1250",
 "difficulty": "Easy",
 "tags": ["Greedy", "Interval"],
 "description": "区间覆盖问题，贪心选择最优策略"
},
{
 "title": "Flying to Fredericton",
 "platform": "UVa OJ",
 "url":
"https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=2255",
 "difficulty": "Medium",
 "tags": ["Graph", "Shortest Path", "Dynamic Programming"],
 "description": "多段最短路径问题，允许最多 k 次飞行"
},
{
 "title": "MICE AND MAZE",
```

```
"platform": "SPOJ",
"url": "https://www.spoj.com/problems/MICEMAZE/",
"difficulty": "Easy",
"tags": ["Graph", "BFS", "Dijkstra"],
"description": "迷宫寻路问题，计算能在给定时间内到达终点的老鼠数量"
},
{
 "title": "Minimum Jumps",
 "platform": "HackerEarth",
 "url": "https://www.hackerearth.com/practice/data-structures/arrays/1-d/practice-problems/algorithm/minimum-jumps-4/description/",
 "difficulty": "Medium",
 "tags": ["Dynamic Programming", "BFS"],
 "description": "计算从数组第一个元素跳到最后一个元素的最小跳跃次数"
},
{
 "title": "Doing Homework",
 "platform": "杭电 HDU",
 "url": "http://acm.hdu.edu.cn/showproblem.php?pid=1074",
 "difficulty": "Medium",
 "tags": ["Dynamic Programming", "Bitmask"],
 "description": "任务调度问题，求最小扣分"
},
{
 "title": "Fight with Monsters",
 "platform": "Codeforces",
 "url": "https://codeforces.com/problemset/problem/1296/D",
 "difficulty": "Easy",
 "tags": ["Greedy", "Sorting"],
 "description": "贪心策略解决怪物战斗问题"
},
{
 "title": "Dividing Chocolate",
 "platform": "AtCoder",
 "url": "https://atcoder.jp/contests/abc159/tasks/abc159_e",
 "difficulty": "Hard",
 "tags": ["Dynamic Programming", "Bitmask", "Prefix Sum"],
 "description": "二维网格分割问题，需要类似的状态转移思路"
},
{
 "title": "Single Wildcard Pattern Matching",
 "platform": "Codeforces",
 "url": "https://codeforces.com/problemset/problem/965/D",
```

```
"difficulty": "Medium",
"tags": ["String", "Greedy"],
"description": "字符串匹配问题，但涉及到间隔匹配的概念"
}
]

def search_problems() -> List[Dict[str, Any]]:
 """搜索相关题目"""
 print("开始搜索动态规划相关题目...")

 # 这里应该调用各个平台的 API 进行搜索
 # 但由于 API 限制，我们使用预定义数据

 print(f"找到 {len(PREDEFINED_PROBLEMS)} 个相关题目")
 return PREDEFINED_PROBLEMS

def categorize_problems(problems: List[Dict[str, Any]]) -> Dict[str, List[Dict[str, Any]]]:
 """将题目按类别分类"""

 categories = {
 "grid_path": [], # 网格路径类
 "frog_jump": [], # 青蛙跳跃类
 "subarray_product": [], # 子数组乘积类
 "subsequence": [], # 子序列类
 "combinatorics": [], # 排列组合类
 "other": [] # 其他类
 }

 for problem in problems:
 title = problem["title"].lower()
 tags = [tag.lower() for tag in problem["tags"]]

 # 分类逻辑
 if any(keyword in title for keyword in ["cherry", "path", "grid", "matrix", "unique", "dungeon"]):
 categories["grid_path"].append(problem)
 elif any(keyword in title for keyword in ["frog", "jump", "stone", "river"]):
 categories["frog_jump"].append(problem)
 elif any(keyword in title for keyword in ["product", "subarray", "multiply"]):
 categories["subarray_product"].append(problem)
 elif any(keyword in title for keyword in ["subsequence", "increasing", "common", "palindromic"]):
 categories["subsequence"].append(problem)
 elif any(keyword in title for keyword in ["knight", "domino", "tiling", "arrange"]):
 categories["subsequence"].append(problem)

 return categories
```

```
"combinatorics"]):
 categories["combinatorics"].append(problem)
else:
 categories["other"].append(problem)

return categories

def save_results(categories: Dict[str, List[Dict[str, Any]]]) -> None:
 """保存搜索结果"""
 # 保存完整的搜索结果
 with open("dp_problems_search_results.json", "w", encoding="utf-8") as f:
 json.dump(categories, f, ensure_ascii=False, indent=2)

 # 生成 Markdown 格式的报告
 with open("dp_problems_report.md", "w", encoding="utf-8") as f:
 f.write("# 动态规划相关题目搜索结果\n\n")

 for category, problems in categories.items():
 if problems:
 f.write(f"# {category.replace('_', ' ')}.title()\n\n")

 for i, problem in enumerate(problems, 1):
 f.write(f"{i}. {problem['title']} ({problem['platform']})\n")
 f.write(f" - 链接: {problem['url']}\n")
 f.write(f" - 难度: {problem['difficulty']}\n")
 f.write(f" - 标签: {', '.join(problem['tags'])}\n")
 f.write(f" - 描述: {problem['description']}\n\n")

 print("搜索结果已保存到 dp_problems_search_results.json 和 dp_problems_report.md")

def main():
 """主函数"""
 # 搜索题目
 problems = search_problems()

 # 分类题目
 categories = categorize_problems(problems)

 # 保存结果
 save_results(categories)

 # 统计信息
 total = sum(len(problems) for problems in categories.values())
```

```
print(f"\n搜索完成！共找到 {total} 个相关题目")
print("分类统计：")
for category, problems in categories.items():
 print(f" {category}: {len(problems)} 个题目")

if __name__ == "__main__":
 main()
```

---

文件: SetCover.java

```
// -*- coding: utf-8 -*-
/*
子集覆盖问题 (Set Cover Problem)
```

问题描述:

给定一个全集  $U$  和一组子集  $S_1, S_2, \dots, S_m$ , 其中每个子集  $S_i$  是  $U$  的子集, 并且有一个权值  $w_i$ 。我们需要选择一些子集, 使得它们的并等于全集  $U$ , 并且所选子集的权值和最小。

贪心策略:

每次选择能够覆盖最多未被覆盖的元素的子集 (按权重计算性价比最高的)。

注意: 子集覆盖问题是 NP 难的, 贪心算法不能保证得到最优解, 但可以得到一个近似比为  $\ln(n)+1$  的解, 其中  $n$  是全集的大小。

时间复杂度:  $O(mn)$ , 其中  $m$  是子集的数量,  $n$  是全集的大小

空间复杂度:  $O(n+m)$ , 需要存储未覆盖的元素集合和子集信息

相关题目:

1. LeetCode 1541. 平衡括号字符串的最少插入次数
  2. LeetCode 1689. 十二进制数的最少数目
  3. LeetCode 45. 跳跃游戏 II
- \*/

```
import java.util.*;

public class SetCover {
 /**
 * 子集覆盖问题的贪心算法实现
 *
 * @param universe 全集 U, 一个包含所有元素的集合
 * @param subsets 子集列表, 每个子集是一个集合
```

```

* @param weights 每个子集的权值列表，如果为 null 则默认为 1
* @return 一个数组，第一个元素是选中的子集索引列表，第二个元素是总权值
* @throws IllegalArgumentException 当输入无效时抛出异常
*/
public static Object[] setCover(Set<Integer> universe, List<Set<Integer>> subsets, double[]
weights) {
 // 参数验证
 if (universe == null || universe.isEmpty()) {
 throw new IllegalArgumentException("universe 不能为空");
 }

 if (subsets == null || subsets.isEmpty()) {
 throw new IllegalArgumentException("subsets 不能为空");
 }

 double[] actualWeights;
 if (weights == null) {
 actualWeights = new double[subsets.size()];
 Arrays.fill(actualWeights, 1.0);
 } else if (weights.length != subsets.size()) {
 throw new IllegalArgumentException("weights 的长度必须与 subsets 相同");
 } else {
 actualWeights = weights;
 }

 // 检查是否可以覆盖全集
 Set<Integer> allElements = new HashSet<>();
 for (Set<Integer> subset : subsets) {
 if (subset == null) {
 throw new IllegalArgumentException("subsets 中的子集不能为 null");
 }
 allElements.addAll(subset);
 }

 for (int elem : universe) {
 if (!allElements.contains(elem)) {
 throw new IllegalArgumentException("给定的子集无法覆盖全集");
 }
 }

 Set<Integer> uncovered = new HashSet<>(universe); // 未被覆盖的元素集合
 List<Integer> selectedSubsets = new ArrayList<>(); // 选中的子集索引列表
 double totalWeight = 0.0; // 总权值
}

```

```

while (!uncovered.isEmpty()) {
 int bestSubsetIndex = -1;
 double bestValue = -1.0; // 性价比 = 覆盖的新元素数量 / 权值

 // 找到性价比最高的子集
 for (int i = 0; i < subsets.size(); i++) {
 Set<Integer> subset = subsets.get(i);
 // 计算该子集能覆盖的未被覆盖的元素数量
 int coveredNew = 0;
 for (int elem : subset) {
 if (uncovered.contains(elem)) {
 coveredNew++;
 }
 }

 if (coveredNew == 0) {
 continue; // 该子集不能覆盖新元素，跳过
 }

 // 计算性价比
 double value = coveredNew / actualWeights[i];

 if (value > bestValue) {
 bestValue = value;
 bestSubsetIndex = i;
 }
 }

 // 如果没有找到合适的子集，说明无法覆盖全集（理论上不应该发生，因为前面已经检查过）
 if (bestSubsetIndex == -1) {
 throw new IllegalArgumentException("无法覆盖全集");
 }

 // 选择该子集
 selectedSubsets.add(bestSubsetIndex);
 totalWeight += actualWeights[bestSubsetIndex];

 // 更新未被覆盖的元素集合
 uncovered.removeAll(subsets.get(bestSubsetIndex));
}

return new Object[] {selectedSubsets, totalWeight};

```

```
}

/**
 * LeetCode 1541. 平衡括号字符串的最少插入次数
 * 题目链接: https://leetcode-cn.com/problems/minimum-insertions-to-balance-a-parentheses-string/
 *
 * 问题描述:
 * 给你一个括号字符串 s，请你返回满足以下条件的 最少 插入次数:
 * - 任何左括号 '(' 必须有相应的两个右括号 ')'
 * - 左括号 '(' 必须在对应的连续两个右括号 ')' 之前
 *
 * 解题思路:
 * 这是一个变种的括号匹配问题，可以使用贪心算法来解决。
 * 我们维护两个变量：需要的右括号数量和需要添加的左括号数量。
 * 遍历字符串，根据当前字符和状态更新这两个变量。
 *
 * @param s 括号字符串
 * @return 最少需要插入的字符数
 */
public static int minInsertions(String s) {
 if (s == null) {
 return 0;
 }

 int insertCount = 0; // 需要插入的字符数
 int needRight = 0; // 需要的右括号数量

 for (char c : s.toCharArray()) {
 if (c == '(') {
 needRight += 2; // 每个左括号需要两个右括号

 // 如果需要的右括号数量是奇数，说明前一个字符需要补充一个右括号
 if (needRight % 2 == 1) {
 insertCount += 1; // 插入一个右括号
 needRight -= 1; // 需要的右括号数量减 1
 }
 } else { // c == ')'
 needRight -= 1;

 // 如果右括号过多，需要添加一个左括号
 if (needRight == -1) {
 insertCount += 1; // 插入一个左括号
 }
 }
 }
}
```

```

 needRight = 1; // 现在需要一个右括号
 }
}
}

return insertCount + needRight;
}

/***
 * LeetCode 1689. 十二进制数的最少数目
 * 题目链接: https://leetcode-cn.com/problems/partitioning-into-minimum-number-of-deci-binary-numbers/
 *
 * 问题描述:
 * 如果一个十进制数字不含任何前导零，且每一位上的数字不是 0 就是 1，那么该数字就是一个 十二进制数。
 * 例如，101 和 1100 都是 十二进制数，而 112 和 3001 不是。
 * 给你一个表示十进制整数的字符串 n，返回和为 n 的 十二进制数 的最少数目。
 *
 * 解题思路:
 * 这个问题可以转化为：找到字符串中最大的数字。因为对于每一位的数字 d，我们需要至少 d 个十二进制数，
 * 每个数在该位上提供 1。
 *
 * @param n 表示十进制整数的字符串
 * @return 和为 n 的十二进制数的最少数目
 */
public static int minPartitions(String n) {
 if (n == null || n.isEmpty()) {
 return 0;
 }

 int maxDigit = 0;
 for (char c : n.toCharArray()) {
 if (c < '0' || c > '9') {
 throw new IllegalArgumentException("输入必须是有效的数字字符串");
 }
 maxDigit = Math.max(maxDigit, c - '0');
 }
 return maxDigit;
}

/***

```

```
* LeetCode 45. 跳跃游戏 II
* 题目链接: https://leetcode-cn.com/problems/jump-game-ii/
*
* 问题描述:
* 给定一个非负整数数组，你最初位于数组的第一个位置。
* 数组中的每个元素代表你在该位置可以跳跃的最大长度。
* 你的目标是使用最少的跳跃次数到达数组的最后一个位置。
*
* 解题思路:
* 使用贪心算法，每次都跳转到能够到达的最远位置。
* 具体来说，我们维护当前可以到达的最远位置 end 和下一跳可以到达的最远位置 farthest。
* 当我们到达 end 时，更新 end 为 farthest 并增加跳跃次数。
*
* @param nums 非负整数数组
* @return 最少跳跃次数
*/
public static int jump(int[] nums) {
 if (nums == null || nums.length <= 1) {
 return 0;
 }

 int jumps = 0; // 跳跃次数
 int currentEnd = 0; // 当前可以到达的最远位置
 int currentFarthest = 0; // 下一跳可以到达的最远位置

 for (int i = 0; i < nums.length - 1; i++) {
 // 更新下一跳可以到达的最远位置
 currentFarthest = Math.max(currentFarthest, i + nums[i]);

 // 如果到达了当前可以到达的边界，必须跳一次
 if (i == currentEnd) {
 jumps++;
 currentEnd = currentFarthest;
 }
 }

 return jumps;
}

// 测试代码
public static void main(String[] args) {
 try {
 // 测试子集覆盖算法
 }
}
```

```

Set<Integer> universe = new HashSet<>(Arrays.asList(1, 2, 3, 4, 5));
List<Set<Integer>> subsets = new ArrayList<>();
subsets.add(new HashSet<>(Arrays.asList(1, 2, 3)));
subsets.add(new HashSet<>(Arrays.asList(2, 4)));
subsets.add(new HashSet<>(Arrays.asList(3, 4)));
subsets.add(new HashSet<>(Arrays.asList(4, 5)));
double[] weights = {5, 10, 3, 8};

Object[] result = setCover(universe, subsets, weights);
List<Integer> selectedIndices = (List<Integer>) result[0];
double totalWeight = (double) result[1];

System.out.println("选中的子集索引: " + selectedIndices);
System.out.println("总权值: " + totalWeight);

// 测试 LeetCode 1541
String s1 = "((()))";
System.out.println("最少插入次数: " + minInsertions(s1)); // 应该输出 1

// 测试 LeetCode 1689
String s2 = "32";
System.out.println("十-二进制数的最少数目: " + minPartitions(s2)); // 应该输出 3

// 测试 LeetCode 45
int[] nums = {2, 3, 1, 1, 4};
System.out.println("最少跳跃次数: " + jump(nums)); // 应该输出 2
} catch (Exception e) {
 System.err.println("错误: " + e.getMessage());
 e.printStackTrace();
}
}

}

=====

文件: set_cover.cpp
=====

// -*- coding: utf-8 -*-
/*
子集覆盖问题 (Set Cover Problem)
*/

```

问题描述:

给定一个全集 U 和一组子集 S\_1, S\_2, ..., S\_m, 其中每个子集 S\_i 是 U 的子集, 并且有一个权值 w\_i。

我们需要选择一些子集，使得它们的并等于全集 U，并且所选子集的权值和最小。

贪心策略：

每次选择能够覆盖最多未被覆盖的元素的子集（按权重计算性价比最高的）。

注意：子集覆盖问题是 NP 难的，贪心算法不能保证得到最优解，但可以得到一个近似比为  $\ln(n) + 1$  的解，其中 n 是全集的大小。

时间复杂度： $O(mn)$ ，其中 m 是子集的数量，n 是全集的大小

空间复杂度： $O(n+m)$ ，需要存储未覆盖的元素集合和子集信息

相关题目：

1. LeetCode 1541. 平衡括号字符串的最少插入次数
2. LeetCode 1689. 十-二进制数的最少数目
3. LeetCode 45. 跳跃游戏 II

\*/

```
#include <iostream>
#include <vector>
#include <unordered_set>
#include <algorithm>
#include <stdexcept>
#include <string>
#include <climits>

using namespace std;

/***
 * 子集覆盖问题的贪心算法实现
 *
 * @param universe 全集 U，一个包含所有元素的集合
 * @param subsets 子集列表，每个子集是一个集合
 * @param weights 每个子集的权值列表，如果为空则默认为 1
 * @return 一个 pair，包含选中的子集索引列表和总权值
 * @throws invalid_argument 当输入无效时抛出异常
 */
pair<vector<int>, double> setCover(const unordered_set<int>& universe,
 const vector<unordered_set<int>>& subsets,
 const vector<double>& weights = {}) {

 // 参数验证
 if (subsets.empty()) {
 throw invalid_argument("subsets 不能为空");
 }
```

```

vector<double> actual_weights;
if (weights.empty()) {
 actual_weights = vector<double>(subsets.size(), 1.0);
} else if (weights.size() != subsets.size()) {
 throw invalid_argument("weights 的长度必须与 subsets 相同");
} else {
 actual_weights = weights;
}

// 检查是否可以覆盖全集
unordered_set<int> all_elements;
for (const auto& subset : subsets) {
 for (int elem : subset) {
 all_elements.insert(elem);
 }
}

for (int elem : universe) {
 if (all_elements.find(elem) == all_elements.end()) {
 throw invalid_argument("给定的子集无法覆盖全集");
 }
}

unordered_set<int> uncovered = universe; // 未被覆盖的元素集合
vector<int> selected_subsets; // 选中的子集索引列表
double total_weight = 0.0; // 总权值

while (!uncovered.empty()) {
 int best_subset_index = -1;
 double best_value = -1.0; // 性价比 = 覆盖的新元素数量 / 权值

 // 找到性价比最高的子集
 for (size_t i = 0; i < subsets.size(); ++i) {
 // 计算该子集能覆盖的未被覆盖的元素数量
 int covered_new = 0;
 for (int elem : subsets[i]) {
 if (uncovered.find(elem) != uncovered.end()) {
 covered_new++;
 }
 }

 if (covered_new == 0) {

```

```

 continue; // 该子集不能覆盖新元素，跳过
 }

 // 计算性价比
 double value = static_cast<double>(covered_new) / actual_weights[i];

 if (value > best_value) {
 best_value = value;
 best_subset_index = i;
 }
}

// 如果没有找到合适的子集，说明无法覆盖全集（理论上不应该发生，因为前面已经检查过）
if (best_subset_index == -1) {
 throw invalid_argument("无法覆盖全集");
}

// 选择该子集
selected_subsets.push_back(best_subset_index);
total_weight += actual_weights[best_subset_index];

// 更新未被覆盖的元素集合
for (int elem : subsets[best_subset_index]) {
 uncovered.erase(elem);
}
}

return {selected_subsets, total_weight};
}

/***
 * LeetCode 1541. 平衡括号字符串的最少插入次数
 * 题目链接: https://leetcode-cn.com/problems/minimum-insertions-to-balance-a-parentheses-string/
 *
 * 问题描述:
 * 给你一个括号字符串 s，请你返回满足以下条件的 最少 插入次数:
 * - 任何左括号 '(' 必须有相应的两个右括号 ')'
 * - 左括号 '(' 必须在对应的连续两个右括号 ')' 之前
 *
 * 解题思路:
 * 这是一个变种的括号匹配问题，可以使用贪心算法来解决。
 * 我们维护两个变量：需要的右括号数量和需要添加的左括号数量。
 * 遍历字符串，根据当前字符和状态更新这两个变量。
 */

```

```

*
* @param s 括号字符串
* @return 最少需要插入的字符数
*/
int minInsertions(const string& s) {
 int insert_count = 0; // 需要插入的字符数
 int need_right = 0; // 需要的右括号数量

 for (char c : s) {
 if (c == '(') {
 need_right += 2; // 每个左括号需要两个右括号

 // 如果需要的右括号数量是奇数，说明前一个字符需要补充一个右括号
 if (need_right % 2 == 1) {
 insert_count += 1; // 插入一个右括号
 need_right -= 1; // 需要的右括号数量减 1
 }
 } else { // c == ')'
 need_right -= 1;

 // 如果右括号过多，需要添加一个左括号
 if (need_right == -1) {
 insert_count += 1; // 插入一个左括号
 need_right = 1; // 现在需要一个右括号
 }
 }
 }

 return insert_count + need_right;
}

/**
* LeetCode 1689. 十-二进制数的最少数目
* 题目链接: https://leetcode-cn.com/problems/partitioning-into-minimum-number-of-deci-binary-numbers/
*
* 问题描述:
* 如果一个十进制数字不含任何前导零，且每一位上的数字不是 0 就是 1，那么该数字就是一个 十-二进制数。
* 例如，101 和 1100 都是 十-二进制数，而 112 和 3001 不是。
* 给你一个表示十进制整数的字符串 n，返回和为 n 的 十-二进制数 的最少数目。
*
* 解题思路:

```

\* 这个问题可以转化为：找到字符串中最大的数字。因为对于每一位的数字 d，我们需要至少 d 个十-二进制数，

\* 每个数在该位上提供 1。

\*

\* @param n 表示十进制整数的字符串

\* @return 和为 n 的十-二进制数的最少数目

\*/

```
int minPartitions(const string& n) {
```

```
 int max_digit = 0;
```

```
 for (char c : n) {
```

```
 max_digit = max(max_digit, c - '0');
```

```
}
```

```
 return max_digit;
```

```
}
```

/\*\*

\* LeetCode 45. 跳跃游戏 II

\* 题目链接：<https://leetcode-cn.com/problems/jump-game-ii/>

\*

\* 问题描述：

\* 给定一个非负整数数组，你最初位于数组的第一个位置。

\* 数组中的每个元素代表你在该位置可以跳跃的最大长度。

\* 你的目标是使用最少的跳跃次数到达数组的最后一个位置。

\*

\* 解题思路：

\* 使用贪心算法，每次都跳转到能够到达的最远位置。

\* 具体来说，我们维护当前可以到达的最远位置 end 和下一跳可以到达的最远位置 farthest。

\* 当我们到达 end 时，更新 end 为 farthest 并增加跳跃次数。

\*

\* @param nums 非负整数数组

\* @return 最少跳跃次数

\*/

```
int jump(const vector<int>& nums) {
```

```
 int n = nums.size();
```

```
 if (n <= 1) {
```

```
 return 0;
```

```
}
```

```
 int jumps = 0; // 跳跃次数
```

```
 int current_end = 0; // 当前可以到达的最远位置
```

```
 int current_farthest = 0; // 下一跳可以到达的最远位置
```

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

```

// 更新下一跳可以到达的最远位置
current_farthest = max(current_farthest, i + nums[i]);

// 如果到达了当前可以到达的边界，必须跳一次
if (i == current_end) {
 jumps++;
 current_end = current_farthest;
}
}

return jumps;
}

```

```

// 辅助函数：打印集合
void printSet(const unordered_set<int>& s) {
 cout << "{";
 bool first = true;
 for (int elem : s) {
 if (!first) {
 cout << ", ";
 }
 cout << elem;
 first = false;
 }
 cout << "}";
}

```

```

// 测试代码
int main() {
 try {
 // 测试子集覆盖算法
 unordered_set<int> universe = {1, 2, 3, 4, 5};
 vector<unordered_set<int>> subsets = {
 {1, 2, 3},
 {2, 4},
 {3, 4},
 {4, 5}
 };
 vector<double> weights = {5, 10, 3, 8};

 auto result = setCover(universe, subsets, weights);
 cout << "选中的子集索引: [";
 for (size_t i = 0; i < result.first.size(); ++i) {

```

```

 if (i > 0) {
 cout << ", ";
 }
 cout << result.first[i];
}
cout << "]" << endl;
cout << "总权值: " << result.second << endl;

// 测试 LeetCode 1541
string s1 = "((()))";
cout << "最少插入次数: " << minInsertions(s1) << endl; // 应该输出 1

// 测试 LeetCode 1689
string s2 = "32";
cout << "十-二进制数的最少数目: " << minPartitions(s2) << endl; // 应该输出 3

// 测试 LeetCode 45
vector<int> nums = {2, 3, 1, 1, 4};
cout << "最少跳跃次数: " << jump(nums) << endl; // 应该输出 2
} catch (const exception& e) {
 cerr << "错误: " << e.what() << endl;
}

return 0;
}
=====

文件: set_cover.py
=====

-*- coding: utf-8 -*-
"""

子集覆盖问题 (Set Cover Problem)

问题描述:
给定一个全集 U 和一组子集 S_1, S_2, ..., S_m, 其中每个子集 S_i 是 U 的子集, 并且有一个权值 w_i。
我们需要选择一些子集, 使得它们的并等于全集 U, 并且所选子集的权值和最小。

贪心策略:
每次选择能够覆盖最多未被覆盖的元素的子集 (按权重计算性价比最高的)。

注意: 子集覆盖问题是 NP 难的, 贪心算法不能保证得到最优解, 但可以得到一个近似比为 ln(n)+1 的解,
其中 n 是全集的大小。

```

问题描述:

给定一个全集  $U$  和一组子集  $S_1, S_2, \dots, S_m$ , 其中每个子集  $S_i$  是  $U$  的子集, 并且有一个权值  $w_i$ 。我们需要选择一些子集, 使得它们的并等于全集  $U$ , 并且所选子集的权值和最小。

贪心策略:

每次选择能够覆盖最多未被覆盖的元素的子集 (按权重计算性价比最高的)。

注意: 子集覆盖问题是 NP 难的, 贪心算法不能保证得到最优解, 但可以得到一个近似比为  $\ln(n)+1$  的解, 其中  $n$  是全集的大小。

时间复杂度:  $O(mn)$ , 其中  $m$  是子集的数量,  $n$  是全集的大小  
空间复杂度:  $O(n+m)$ , 需要存储未覆盖的元素集合和子集信息

相关题目:

1. LeetCode 1541. 平衡括号字符串的最少插入次数
2. LeetCode 1689. 十-二进制数的最少数目
3. LeetCode 45. 跳跃游戏 II

"""

```
def set_cover(universe, subsets, weights=None):
```

"""

子集覆盖问题的贪心算法实现

Args:

universe: 全集  $U$ , 一个包含所有元素的集合

subsets: 子集列表, 每个子集是一个集合

weights: 每个子集的权值列表, 如果为 None 则默认为 1

Returns:

tuple: (选中的子集索引列表, 总权值)

Raises:

ValueError: 当输入无效时抛出异常

"""

# 参数验证

```
if not isinstance(universe, set):
```

raise ValueError("universe 必须是一个集合")

```
if not subsets or not all(isinstance(s, set) for s in subsets):
```

raise ValueError("subsets 必须是一个非空的集合列表")

```
if weights is None:
```

weights = [1] \* len(subsets)

```
elif len(weights) != len(subsets):
```

raise ValueError("weights 的长度必须与 subsets 相同")

# 检查是否可以覆盖全集

```
all_elements = set()
```

```
for s in subsets:
```

all\_elements.update(s)

```
if not universe.issubset(all_elements):
```

```

raise ValueError("给定的子集无法覆盖全集")

uncovered = universe.copy() # 未被覆盖的元素集合
selected_subsets = [] # 选中的子集索引列表
total_weight = 0 # 总权值

while uncovered:
 best_subset_index = -1
 best_value = -1 # 性价比 = 覆盖的新元素数量 / 权值

 # 找到性价比最高的子集
 for i, subset in enumerate(subsets):
 # 计算该子集能覆盖的未被覆盖的元素数量
 covered_new = len(subset & uncovered)
 if covered_new == 0:
 continue # 该子集不能覆盖新元素，跳过

 # 计算性价比
 value = covered_new / weights[i]

 if value > best_value:
 best_value = value
 best_subset_index = i

 # 如果没有找到合适的子集，说明无法覆盖全集（理论上不应该发生，因为前面已经检查过）
 if best_subset_index == -1:
 raise ValueError("无法覆盖全集")

 # 选择该子集
 selected_subsets.append(best_subset_index)
 total_weight += weights[i]

 # 更新未被覆盖的元素集合
 uncovered -= subsets[best_subset_index]

return selected_subsets, total_weight

def set_cover_leetcode_1541(s):
 """
 LeetCode 1541. 平衡括号字符串的最少插入次数
 题目链接: https://leetcode-cn.com/problems/minimum-insertions-to-balance-a-parentheses-string/
 """

 n = len(s)
 if n == 0:
 return 0
 if n % 2 != 0:
 return 1

 left, right = 0, 0
 for i in range(n):
 if s[i] == '(':
 left += 1
 else:
 right += 1
 if left > right:
 return left - right + 1
 if left < right:
 return right - left + 1
 return 0

```

问题描述:

给你一个括号字符串 s , 请你返回满足以下条件的 最少 插入次数:

- 任何左括号 '(' 必须有相应的两个右括号 ')')
- 左括号 '(' 必须在对应的连续两个右括号 ')' 之前

解题思路:

这是一个变种的括号匹配问题，可以使用贪心算法来解决。

我们维护两个变量：需要的右括号数量和需要添加的左括号数量。

遍历字符串，根据当前字符和状态更新这两个变量。

Args:

s: 括号字符串

Returns:

int: 最少需要插入的字符数

"""

```
insert_count = 0 # 需要插入的字符数
need_right = 0 # 需要的右括号数量
```

```
for c in s:
```

```
 if c == '(':
```

```
 need_right += 2 # 每个左括号需要两个右括号
```

```
如果需要的右括号数量是奇数，说明前一个字符需要补充一个右括号
```

```
if need_right % 2 == 1:
```

```
 insert_count += 1 # 插入一个右括号
```

```
 need_right -= 1 # 需要的右括号数量减 1
```

```
else: # c == ')'
```

```
 need_right -= 1
```

```
如果右括号过多，需要添加一个左括号
```

```
if need_right == -1:
```

```
 insert_count += 1 # 插入一个左括号
```

```
 need_right = 1 # 现在需要一个右括号
```

```
return insert_count + need_right
```

```
def set_cover_leetcode_1689(s):
```

"""

LeetCode 1689. 十-二进制数的最少数目

题目链接: <https://leetcode-cn.com/problems/partitioning-into-minimum-number-of-deci-binary->

numbers/

### 问题描述:

如果一个十进制数字不含任何前导零，且每一位上的数字不是 0 就是 1，那么该数字就是一个 十-二进制数。

例如，101 和 1100 都是 十-二进制数，而 112 和 3001 不是。

给你一个表示十进制整数的字符串 n，返回和为 n 的 十-二进制数 的最少数目。

### 解题思路:

这个问题可以转化为：找到字符串中最大的数字。因为对于每一位的数字 d，我们需要至少 d 个十-二进制数，

每个数在该位上提供 1。

### Args:

s: 表示十进制整数的字符串

### Returns:

int: 和为 s 的十-二进制数的最少数目

"""

```
return max(int(c) for c in s)
```

```
def set_cover_leetcode_45(nums):
```

"""

LeetCode 45. 跳跃游戏 II

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

### 问题描述:

给定一个非负整数数组，你最初位于数组的第一个位置。

数组中的每个元素代表你在该位置可以跳跃的最大长度。

你的目标是使用最少的跳跃次数到达数组的最后一个位置。

### 解题思路:

使用贪心算法，每次都跳转到能够到达的最远位置。

具体来说，我们维护当前可以到达的最远位置 end 和下一跳可以到达的最远位置 farthest。

当我们到达 end 时，更新 end 为 farthest 并增加跳跃次数。

### Args:

nums: 非负整数数组

### Returns:

int: 最少跳跃次数

"""

```

n = len(nums)
if n <= 1:
 return 0

jumps = 0 # 跳跃次数
current_end = 0 # 当前可以到达的最远位置
current_farthest = 0 # 下一跳可以到达的最远位置

for i in range(n - 1):
 # 更新下一跳可以到达的最远位置
 current_farthest = max(current_farthest, i + nums[i])

 # 如果到达了当前可以到达的边界，必须跳一次
 if i == current_end:
 jumps += 1
 current_end = current_farthest

return jumps

```

```

测试代码
if __name__ == "__main__":
 # 测试子集覆盖算法
 universe = {1, 2, 3, 4, 5}
 subsets = [{1, 2, 3}, {2, 4}, {3, 4}, {4, 5}]
 weights = [5, 10, 3, 8]

 try:
 selected, total_weight = set_cover(universe, subsets, weights)
 print(f"选中的子集索引: {selected}")
 print(f"总权值: {total_weight}")
 except ValueError as e:
 print(f"错误: {e}")

 # 测试 LeetCode 1541
 s1 = "((()))"
 print(f"最少插入次数: {set_cover_leetcode_1541(s1)}") # 应该输出 1

 # 测试 LeetCode 1689
 s2 = "32"
 print(f"十-二进制数的最少数目: {set_cover_leetcode_1689(s2)}") # 应该输出 3

 # 测试 LeetCode 45

```

```
nums = [2, 3, 1, 1, 4]
print(f"最少跳跃次数: {set_cover_leetcode_45(nums)}") # 应该输出 2
```

---

文件: TestMinimumPathSum.java

---

```
package class127;
```

```
public class TestMinimumPathSum {
 public static void main(String[] args) {
```

```
 // 测试用例 1
```

```
 int[][] grid1 = {
 {1, 3, 1},
 {1, 5, 1},
 {4, 2, 1}
 };
```

```
 System.out.println("测试用例 1 结果: " + Code09_MinimumPathSum.minPathSum(grid1)); // 预期
```

输出: 7

```
 // 测试用例 2
```

```
 int[][] grid2 = {
 {1, 2, 3},
 {4, 5, 6}
 };
```

```
 System.out.println("测试用例 2 结果: " + Code09_MinimumPathSum.minPathSum(grid2)); // 预期
```

输出: 12

```
 // 测试用例 3
```

```
 int[][] grid3 = {
 {1, 2},
 {1, 1}
 };
```

```
 System.out.println("测试用例 3 结果: " + Code09_MinimumPathSum.minPathSum(grid3)); // 预期
```

输出: 3

```
 // 测试用例 4
```

```
 int[][] grid4 = {
 {1, 3, 1, 2},
 {1, 5, 1, 3},
 {4, 2, 1, 1}
 };
```

```
 System.out.println("测试用例 4 结果: " + Code09_MinimumPathSum.minPathSum(grid4)); // 预期
```

```
输出: 8
```

```
 }
}
```

```
=====
```

```
文件: test_dp.py
```

```
=====
```

```
简单测试文件来验证 DPFusion.py 的基本功能
```

```
try:
```

```
 # 尝试导入并执行一些基本函数
```

```
 from DPFusion import edit_distance, length_of_lis
```

```
 # 测试基本功能
```

```
 print("测试编辑距离函数...")
```

```
 result1 = edit_distance("horse", "ros")
```

```
 print(f"编辑距离('horse', 'ros') = {result1}")
```

```
 print(f"预期结果: 3")
```

```
 print("\n 测试最长递增子序列函数...")
```

```
 result2 = length_of_lis([10, 9, 2, 5, 3, 7, 101, 18])
```

```
 print(f"最长递增子序列长度 = {result2}")
```

```
 print(f"预期结果: 4")
```

```
 print("\n 基础测试通过, 代码语法正确!")
```

```
except Exception as e:
```

```
 print(f"测试过程中遇到错误: {e}")
```

```
 import traceback
```

```
 traceback.print_exc()
```

```
=====
```