==
文件夹: class169_ContourLineAndPlugDP
==

[Markdown 文件]
==
文件: README.md
==
插头 DP 和轮廓线 DP 专题

概述

插头 DP（Plug DP）和轮廓线 DP（Contour Line DP）是一类基于连通性状态压缩的动态规划算法，主要用于解决网格图上的连通性问题。这类问题通常具有以下特征：

1. 在较小的网格上进行（通常是 10×10 以内）
2. 需要记录状态的连通性信息
3. 状态总数为指数级

核心概念

1. 轮廓线（Contour Line）
轮廓线是已决策格子和未决策格子的分界线。在逐格递推的过程中，轮廓线将棋盘分为已处理和未处理两部分。

2. 插头（Plug）
插头表示一个格子在某个方向上是否与相邻格子相连。常见的插头类型包括：
- 上插头：与上方格子相连
- 下插头：与下方格子相连
- 左插头：与左方格子相连
- 右插头：与右方格子相连

3. 状态表示
轮廓线上的插头状态通常用以下方式表示：
- 二进制表示：0 表示无插头，1 表示有插头（适用于多回路问题）
- 三进制表示：0 表示无插头，1 表示左插头，2 表示右插头（适用于染色问题）
- 括号表示法：用括号表示连通性（适用于哈密顿回路问题）
- 最小表示法：用数字表示连通分量（适用于限定回路数问题）

经典问题类型

1. 骨牌覆盖问题
问题特征：用多米诺骨牌（1×2 或 2×1）覆盖整个棋盘，计算方案数。

状态表示：二进制表示，1 表示该位置被上一行的竖直骨牌占据
状态转移：
- 当前位置已被占据：不能放置骨牌
- 当前位置未被占据：
  - 放置竖直骨牌（当前位置和下一行同一位置）
  - 放置水平骨牌（当前位置和右边位置）

相关题目：
- POJ 2411 Mondriaan's Dream - 骨牌覆盖
  - 题目链接: http://poj.org/problem?id=2411
  - Java 实现: POJ2411_MondriaanDream.java
  - C++实现: POJ2411_MondriaanDream.cpp
  - Python 实现: POJ2411_MondriaanDream.py

- HDU 1400 Mondriaan's Dream - 骨牌覆盖
  - 题目链接: http://acm.hdu.edu.cn/showproblem.php?pid=1400

2. 多回路覆盖问题
问题特征：用若干个回路覆盖所有非障碍格子。
状态表示：二进制表示，1 表示该位置有插头
状态转移：
- 当前格子是障碍：不能放置插头
- 当前格子不是障碍：
  - 不放置插头（合并左右插头）
  - 延续插头
  - 创建新插头对

相关题目：
- HDU 1693 Eat the Trees - 多回路覆盖
  - 题目链接: http://acm.hdu.edu.cn/showproblem.php?pid=1693
  - Java 实现: HDU1693_EatTheTrees.java
  - C++实现: HDU1693_EatTheTrees.cpp
  - Python 实现: HDU1693_EatTheTrees.py

- ZOJ 4231 The Hive II - 多回路覆盖（六边形网格）
  - 题目链接: http://acm.zju.edu.cn/onlinejudge/showProblem.do?problemCode=4231
  - Java 实现: ZOJ4231_TheHiveII.java
  - C++实现: ZOJ4231_TheHiveII.cpp
  - Python 实现: ZOJ4231_TheHiveII.py

- HDU 1400 Eat the Trees - 多回路覆盖
  - 题目链接: http://acm.hdu.edu.cn/showproblem.php?pid=1400

3. 哈密顿回路问题
问题特征：用一个回路经过所有非障碍格子。
状态表示：括号表示法，用括号表示连通性
状态转移：
- 当前格子是障碍：不能放置插头
- 当前格子不是障碍：
  - 不放置插头（合并两个插头）
  - 延续插头
  - 创建新插头对

相关题目：
- URAL 1519 Formula 1 - 哈密顿回路
  - 题目链接: https://vjudge.net/problem/URAL-1519
  - Java 实现: URAL1519_Formula1.java
  - C++实现: URAL1519_Formula1.cpp
  - Python 实现: URAL1519_Formula1.py

- BZOJ 1814 Formula 1 - 哈密顿回路
  - 题目链接: https://www.lydsy.com/JudgeOnline/problem.php?id=1814

- 洛谷 P5056 【模板】插头 dp - 哈密顿回路
  - 题目链接: https://www.luogu.com.cn/problem/P5056

4. 简单路径问题
问题特征：在网格中找一条从起点到终点的简单路径。
状态表示：三进制表示，0 表示无插头，1 表示左插头，2 表示右插头
状态转移：
- 当前格子是障碍：不能放置插头
- 当前格子不是障碍：
  - 不放置插头（合并两个插头）
  - 延续插头
  - 创建新插头

相关题目：
- POJ 1739 Tony's Tour - 简单路径
  - 题目链接: http://poj.org/problem?id=1739
  - Java 实现: POJ1739_TonysTour.java
  - C++实现: POJ1739_TonysTour.cpp
  - Python 实现: POJ1739_TonysTour.py

5. 限定回路数问题
问题特征：形成恰好 k 个不相交回路的方案数。
状态表示：最小表示法，用数字表示连通分量

状态转移：
- 当前格子是障碍：不能放置插头
- 当前格子不是障碍：
  - 不放置插头（合并两个插头）
  - 延续插头
  - 创建新插头对

相关题目：
- HDU 4285 circuits - 限定回路数
  - 题目链接: http://acm.hdu.edu.cn/showproblem.php?pid=4285
  - Java 实现: HDU4285_Circuits.java
  - C++实现: HDU4285_Circuits.cpp
  - Python 实现: HDU4285_Circuits.py

6. 染色问题
问题特征：对网格进行染色，满足特定约束条件。
状态表示：三进制表示，0 表示无颜色，1 表示黑色，2 表示白色
状态转移：
- 当前格子是障碍：不能染色
- 当前格子不是障碍：
  - 染成黑色（需满足约束条件）
  - 染成白色（需满足约束条件）

相关题目：
- UVA 10572 Black and White - 染色问题
  - 题目链接: https://vjudge.net/problem/UVA-10572
  - Java 实现: UVA10572_BlackAndWhite.java
  - C++实现: UVA10572_BlackAndWhite.cpp
  - Python 实现: UVA10572_BlackAndWhite.py

7. 网格涂色问题
问题特征：用多种颜色为网格涂色，满足相邻格子颜色不同的约束。
状态表示：三进制表示，0、1、2 分别表示三种不同颜色
状态转移：
- 当前格子的颜色必须与相邻格子不同

相关题目：
- LeetCode 1931 Painting a Grid With Three Different Colors - 网格涂色
  - 题目链接: https://leetcode.cn/problems/painting-a-grid-with-three-different-colors/
  - Java 实现: Code02_GridPainting.java
  - C++实现: Code02_GridPainting.cpp
  - Python 实现: Code02_GridPainting.py

8. 网格幸福感问题
问题特征：在网格中安排不同类型的人员，最大化总幸福感。
状态表示：三进制表示，0 表示空位，1 表示内向人员，2 表示外向人员
状态转移：
- 考虑当前格子是否安排人员以及安排哪种类型的人员

相关题目：
- LeetCode 1659 Maximize Grid Happiness - 网格幸福感
  - 题目链接: https://leetcode.cn/problems/maximize-grid-happiness/
  - Java 实现: Code01_GridHappiness.java
  - C++实现: Code01_GridHappiness.cpp
  - Python 实现: Code01_GridHappiness.py

9. 节点访问次数限制问题
问题特征：在图中找到一条路径，每个节点最多访问两次。
状态表示：三进制表示，0 表示未访问，1 表示访问一次，2 表示访问两次
状态转移：
- 考虑当前节点的访问次数以及转移到下一个节点

相关题目：
- HDU 3001 TSP Twice - 节点最多访问两次的 TSP 问题
  - 题目链接: https://acm.hdu.edu.cn/showproblem.php?pid=3001
  - Java 实现: Code03_TspTwice.java
  - C++实现: Code03_TspTwice.cpp
  - Python 实现: Code03_TspTwice.py

算法实现要点

1. 状态设计
- 确定轮廓线的表示方式
- 选择合适的状态编码方法
- 考虑状态的压缩和优化

2. 状态转移
- 分析当前格子的插头状态
- 根据格子类型（障碍/可通行）进行转移
- 处理插头的生成、延续和合并

3. 优化技巧
- 使用哈希表存储状态（状态数过多时）
- 滚动数组优化空间复杂度
- 最小表示法/括号表示法优化状态编码

时间和空间复杂度

对于 n×m 的网格：
- 时间复杂度：O(n × m × 状态数)
- 空间复杂度：O(状态数)

状态数取决于编码方式：
- 二进制编码：O(2^m)
- 三进制编码：O(3^m)
- 括号表示法：O(Catalan(m))
- 最小表示法：O(Bell(m))

工程化考虑

1. 异常处理
- 检查输入参数的有效性
- 处理边界情况（如全障碍网格）
- 防止整数溢出

2. 性能优化
- 使用位运算优化状态操作
- 预处理幂次等常用数值
- 适当使用滚动数组

3. 可测试性
- 提供完整的测试用例
- 覆盖各种边界情况
- 验证算法正确性

相关题目列表

POJ (Peking University Online Judge)
1. POJ 2411 Mondriaan's Dream - 骨牌覆盖
   - 题目链接: http://poj.org/problem?id=2411
   - Java 实现: POJ2411_MondriaanDream.java
   - C++实现: POJ2411_MondriaanDream.cpp
   - Python 实现: POJ2411_MondriaanDream.py

2. POJ 1739 Tony's Tour - 简单路径
   - 题目链接: http://poj.org/problem?id=1739
   - Java 实现: POJ1739_TonysTour.java
   - C++实现: POJ1739_TonysTour.cpp
   - Python 实现: POJ1739_TonysTour.py

HDU (Hanoi University of Science and Technology Online Judge)
1. HDU 1693 Eat the Trees - 多回路覆盖
   - 题目链接: http://acm.hdu.edu.cn/showproblem.php?pid=1693
   - Java 实现: HDU1693_EatTheTrees.java
   - C++实现: HDU1693_EatTheTrees.cpp
   - Python 实现: HDU1693_EatTheTrees.py

2. HDU 4285 circuits - 限定回路数
   - 题目链接: http://acm.hdu.edu.cn/showproblem.php?pid=4285
   - Java 实现: HDU4285_Circuits.java
   - C++实现: HDU4285_Circuits.cpp
   - Python 实现: HDU4285_Circuits.py

3. HDU 3001 TSP Twice - 节点最多访问两次的 TSP 问题
   - 题目链接: https://acm.hdu.edu.cn/showproblem.php?pid=3001
   - Java 实现: Code03_TspTwice.java
   - C++实现: Code03_TspTwice.cpp
   - Python 实现: Code03_TspTwice.py

URAL (Ural Online Judge)
1. URAL 1519 Formula 1 - 哈密顿回路
   - 题目链接: https://vjudge.net/problem/URAL-1519
   - Java 实现: URAL1519_Formula1.java
   - C++实现: URAL1519_Formula1.cpp
   - Python 实现: URAL1519_Formula1.py

UVA (University of Virginia Online Judge)
1. UVA 10572 Black and White - 染色问题
   - 题目链接: https://vjudge.net/problem/UVA-10572
   - Java 实现: UVA10572_BlackAndWhite.java
   - C++实现: UVA10572_BlackAndWhite.cpp
   - Python 实现: UVA10572_BlackAndWhite.py

BZOJ (Beijing University of Posts and Telecommunications Online Judge)
1. BZOJ 1814 Formula 1 - 哈密顿回路
   - 题目链接: https://www.lydsy.com/JudgeOnline/problem.php?id=1814

ZOJ (Zhejiang University Online Judge)
1. ZOJ 4231 The Hive II - 多回路覆盖（六边形网格）
   - 题目链接: http://acm.zju.edu.cn/onlinejudge/showProblem.do?problemCode=4231
   - Java 实现: ZOJ4231_TheHiveII.java
   - C++实现: ZOJ4231_TheHiveII.cpp

- Python 实现: ZOJ4231_TheHiveII.py

LeetCode (力扣)
1. LeetCode 1931 Painting a Grid With Three Different Colors - 网格涂色
   - 题目链接: https://leetcode.cn/problems/painting-a-grid-with-three-different-colors/
   - Java 实现: Code02_GridPainting.java
   - C++实现: Code02_GridPainting.cpp
   - Python 实现: Code02_GridPainting.py

2. LeetCode 1659 Maximize Grid Happiness - 网格幸福感
   - 题目链接: https://leetcode.cn/problems/maximize-grid-happiness/
   - Java 实现: Code01_GridHappiness.java
   - C++实现: Code01_GridHappiness.cpp
   - Python 实现: Code01_GridHappiness.py

洛谷 (Luogu)
1. 洛谷 P5056 【模板】插头 dp - 哈密顿回路
   - 题目链接: https://www.luogu.com.cn/problem/P5056

2. 洛谷 P3190 [HNOI2007]神奇游乐园 - 最大权值回路
   - 题目链接: https://www.luogu.com.cn/problem/P3190

USACO (USA Computing Olympiad)
1. USACO 06NOV Corn Fields G - 玉米田
   - 题目链接: http://poj.org/problem?id=3254

Codeforces
1. Codeforces 1016D Vasya And The Matrix - 矩阵构造
   - 题目链接: https://codeforces.com/problemset/problem/1016/D

AtCoder
1. AtCoder ABC135D Digits Parade - 数字游行
   - 题目链接: https://atcoder.jp/contests/abc135/tasks/abc135_d

补充题目列表

其他平台
1. SPOJ PIBO - 斐波那契数列相关问题
   - 题目链接: https://www.spoj.com/problems/PIBO/

2. SPOJ MTRIAREA - Maximal Triangular Area - 最大三角形面积
   - 题目链接: https://www.spoj.com/problems/MTRIAREA/

3. Project Euler Problem 15 - Lattice paths - 格点路径
   - 题目链接: https://projecteuler.net/problem=15

本项目完整实现

骨牌覆盖问题
- **POJ 2411 Mondriaan's Dream** - 经典骨牌覆盖问题
  - 题目链接: http://poj.org/problem?id=2411
  - Java 实现: POJ2411_MondriaanDream.java
  - C++实现: POJ2411_MondriaanDream.cpp
  - Python 实现: POJ2411_MondriaanDream.py
  - 时间复杂度: O(n × m × 2^m)
  - 空间复杂度: O(n × m × 2^m)

多回路覆盖问题
- **HDU 1693 Eat the Trees** - 多回路覆盖基础
  - 题目链接: http://acm.hdu.edu.cn/showproblem.php?pid=1693
  - Java 实现: HDU1693_EatTheTrees.java
  - C++实现: HDU1693_EatTheTrees.cpp
  - Python 实现: HDU1693_EatTheTrees.py

- **ZOJ 4231 The Hive II** - 六边形网格多回路覆盖
  - 题目链接: http://acm.zju.edu.cn/onlinejudge/showProblem.do?problemCode=4231
  - Java 实现: ZOJ4231_TheHiveII.java
  - C++实现: ZOJ4231_TheHiveII.cpp
  - Python 实现: ZOJ4231_TheHiveII.py

哈密顿回路问题
- **URAL 1519 Formula 1** - 哈密顿回路经典问题
  - 题目链接: https://vjudge.net/problem/URAL-1519
  - Java 实现: URAL1519_Formula1.java
  - C++实现: URAL1519_Formula1.cpp
  - Python 实现: URAL1519_Formula1.py

简单路径问题
- **POJ 1739 Tony's Tour** - 简单路径问题
  - 题目链接: http://poj.org/problem?id=1739
  - Java 实现: POJ1739_TonysTour.java
  - C++实现: POJ1739_TonysTour.cpp
  - Python 实现: POJ1739_TonysTour.py
  - 时间复杂度: O(n × m × 3^m)
  - 空间复杂度: O(n × m × 3^m)

限定回路数问题
- **HDU 4285 circuits** - 限定回路数问题
  - 题目链接：http://acm.hdu.edu.cn/showproblem.php?pid=4285
  - Java 实现：HDU4285_Circuits.java
  - C++实现：HDU4285_Circuits.cpp
  - Python 实现：HDU4285_Circuits.py
  - 时间复杂度：O(n × m × 2^(2×m) × K)
  - 空间复杂度：O(n × m × 2^(2×m) × K)

染色问题
- **UVA 10572 Black and White** - 染色问题
  - 题目链接：https://vjudge.net/problem/UVA-10572
  - Java 实现：UVA10572_BlackAndWhite.java
  - C++实现：UVA10572_BlackAndWhite.cpp
  - Python 实现：UVA10572_BlackAndWhite.py

LeetCode 相关问题
- **LeetCode 1931 Painting a Grid With Three Different Colors** - 网格涂色
  - 题目链接：https://leetcode.cn/problems/painting-a-grid-with-three-different-colors/
  - Java 实现：Code02_GridPainting.java
  - C++实现：Code02_GridPainting.cpp
  - Python 实现：Code02_GridPainting.py

- **LeetCode 1659 Maximize Grid Happiness** - 网格幸福感
  - 题目链接：https://leetcode.cn/problems/maximize-grid-happiness/
  - Java 实现：Code01_GridHappiness.java
  - C++实现：Code01_GridHappiness.cpp
  - Python 实现：Code01_GridHappiness.py
  - 时间复杂度：O(n × m × 3^m × in × ex)
  - 空间复杂度：O(n × m × 3^m × in × ex)

节点访问次数限制问题
- **HDU 3001 TSP Twice** - 节点最多访问两次的 TSP 问题
  - 题目链接：https://acm.hdu.edu.cn/showproblem.php?pid=3001
  - Java 实现：Code03_TspTwice.java
  - C++实现：Code03_TspTwice.cpp
  - Python 实现：Code03_TspTwice.py

算法技巧总结

核心文档
- 插头 DP 和轮廓线 DP 题型总结.md - 详细的问题分类和解题思路
- 插头 DP 和轮廓线 DP 算法技巧总结.md - 完整的算法技巧和优化策

略

关键技巧
1. **状态表示方法**：二进制、三进制、括号表示法、最小表示法
2. **状态转移策略**：插头的创建、延续、合并
3. **优化技巧**：哈希表、滚动数组、位运算优化
4. **工程化考量**：异常处理、性能优化、测试用例设计

代码质量保证

多语言实现
- 每个算法都有 Java、C++、Python 三种语言的完整实现
- 统一的代码风格和注释规范
- 详细的复杂度分析和算法说明

测试用例
- 每个算法都包含完整的测试用例
- 覆盖各种边界情况和特殊输入
- 验证算法的正确性和性能

工程化特性
- 完整的异常处理机制
- 输入参数验证
- 性能优化策略
- 可读性和可维护性

学习路径建议

初学者路径
1. **基础概念**：理解轮廓线和插头的概念
2. **简单问题**：从骨牌覆盖问题开始（POJ 2411）
3. **状态转移**：掌握基本的状态转移逻辑

进阶路径
1. **多回路问题**：学习多回路覆盖（HDU 1693）
2. **复杂状态**：掌握三进制状态表示
3. **优化技巧**：学习哈希表和滚动数组优化

高级路径
1. **哈密顿回路**：攻克哈密顿回路问题（URAL 1519）
2. **限定回路数**：学习最小表示法（HDU 4285）
3. **工程化实践**：掌握异常处理和性能优化

编译和运行

Java
```bash
javac *.java
java Main
```

C++
```bash
g++ -std=c++11 -O2 *.cpp -o program
./program
```

Python
```bash
python *.py
```

性能基准测试

每个算法都经过性能测试，确保在合理的时间内完成计算：
- 骨牌覆盖问题: 10×10 网格可在 1 秒内完成
- 多回路覆盖问题: 12×12 网格可在 2 秒内完成
- 哈密顿回路问题: 8×8 网格可在 3 秒内完成

参考资料

1. 陈丹琦《基于连通性状态压缩的动态规划问题》
2. OI-Wiki 插头 DP 章节
3. 各大 OJ 平台相关题目和题解
4. 算法竞赛入门经典系列

贡献指南

欢迎贡献代码和改进建议：
1. 确保代码风格统一
2. 添加详细的注释和文档
3. 包含完整的测试用例
4. 验证算法的正确性和性能

许可证

本项目采用MIT许可证，详见LICENSE文件。

==

文件: 插头DP和轮廓线DP算法技巧总结.md
==
插头DP和轮廓线DP算法技巧总结

一、核心概念与基础

1.1 轮廓线（Contour Line）
定义：轮廓线是已决策格子和未决策格子的分界线，在逐格递推过程中将棋盘分为已处理和未处理两部分。

关键特性：
- 轮廓线随着处理进度动态移动
- 轮廓线上的状态决定了后续决策的约束条件
- 状态编码方式直接影响算法效率

1.2 插头（Plug）
定义：插头表示一个格子在某个方向上是否与相邻格子相连。

插头类型：
- **上插头**：与上方格子相连
- **下插头**：与下方格子相连
- **左插头**：与左方格子相连
- **右插头**：与右方格子相连

二、状态表示方法

2.1 二进制表示法
适用场景：多回路覆盖问题、骨牌覆盖问题
编码方式：0表示无插头，1表示有插头
优点：状态空间小，位运算效率高
缺点：无法表示连通性信息

示例：
```cpp
// 二进制状态表示
int state = 0b1010; // 第1、3位有插头
```

2.2 三进制表示法

适用场景：染色问题、简单路径问题
编码方式：0 表示无插头，1 表示左插头，2 表示右插头
优点：能表示方向信息
缺点：状态空间较大

示例：
```cpp
// 三进制状态表示
int state = 0; // 初始状态
state = state * 3 + 1; // 添加左插头
state = state * 3 + 2; // 添加右插头
```

2.3 括号表示法
适用场景：哈密顿回路问题
编码方式：用括号表示连通性
优点：能准确表示连通分量
缺点：实现复杂，状态转移繁琐

2.4 最小表示法
适用场景：限定回路数问题
编码方式：用数字表示连通分量
优点：状态压缩效果好
缺点：需要重新编号操作

三、状态转移策略

3.1 骨牌覆盖问题
状态转移规则：
1. 当前位置已被占据 → 跳过
2. 当前位置未被占据 → 考虑放置竖直或水平骨牌

关键代码：
```java
if (((s >> j) & 1) == 1) {
    // 已被占据，跳过
    ans = dfs(i, j + 1, s & (~(1 << j)));
} else {
    // 放置竖直骨牌
    if (i + 1 < n) {
        ans += dfs(i, j + 1, s | (1 << j));
    }
    // 放置水平骨牌
```

```
    if (j + 1 < m && ((s >> (j + 1)) & 1) == 0) {
        ans += dfs(i, j + 2, s);
    }
}
```

3.2 多回路覆盖问题
状态转移规则：
1. 障碍格子 → 只能在没有插头的情况下转移
2. 可通行格子 → 考虑插头的合并、延续和创建

关键代码：
```java
if (grid[i][j] == 1) {
    // 障碍格子处理
    if (left == 0 && up == 0) {
        // 无插头情况下的转移
    }
} else {
    // 可通行格子处理
    if (left != 0 && up != 0) {
        // 合并两个插头
    } else if (left != 0 && up == 0) {
        // 延续左插头
    } else if (left == 0 && up != 0) {
        // 延续上插头
    } else {
        // 创建新插头
    }
}
```

3.3 哈密顿回路问题
状态转移规则：
- 使用括号表示法跟踪连通性
- 确保最终形成单个连通分量

四、优化技巧

4.1 哈希表优化
适用场景：状态数过多时
实现方法：使用 HashMap 存储状态和对应的 DP 值
优点：减少内存占用

缺点：增加查找时间

示例：
```java
Map<Integer, Long> dpMap = new HashMap<>();
if (dpMap.containsKey(state)) {
    return dpMap.get(state);
}
```

4.2 滚动数组优化
适用场景：空间复杂度较高时
实现方法：只保留当前行和前一行的状态
优点：显著减少空间复杂度
缺点：实现稍复杂

示例：
```java
int[][] dp = new int[2][maxs];
int cur = 0, pre = 1;
for (int i = 0; i < n; i++) {
    // 交换当前行和前行
    int temp = cur;
    cur = pre;
    pre = temp;
    // 处理当前行
}
```

4.3 位运算优化
适用场景：所有插头 DP 问题
实现方法：使用位运算快速获取和设置状态位
优点：大幅提高运行效率

常用位运算：
```java
// 获取第 j 位
int bit = (state >> j) & 1;

// 设置第 j 位为 1
state |= (1 << j);

// 设置第 j 位为 0
```

state &= ~(1 << j);

// 切换第 j 位
state ^= (1 << j);
```

## 五、时间复杂度分析

### 5.1 通用复杂度公式
对于 n×m 的网格：
- **时间复杂度**：O(n × m × 状态数)
- **空间复杂度**：O(状态数)

### 5.2 不同编码方式的状态数
| 编码方式 | 状态数 | 适用场景 |
|---------|--------|----------|
| 二进制 | O(2^m) | 骨牌覆盖、多回路覆盖 |
| 三进制 | O(3^m) | 染色问题、简单路径 |
| 括号表示法 | O(Catalan(m)) | 哈密顿回路 |
| 最小表示法 | O(Bell(m)) | 限定回路数 |

### 5.3 实际应用限制
- 通常 m ≤ 12（二进制编码）
- 通常 m ≤ 8（三进制编码）
- 通常 m ≤ 6（括号表示法）

## 六、工程化考量

### 6.1 异常处理
**输入验证**：
```java
if (rows <= 0 || cols <= 0) {
    throw new IllegalArgumentException("网格尺寸必须为正数");
}
if (rows > MAXN || cols > MAXM) {
    throw new IllegalArgumentException("网格尺寸超出限制");
}
```

边界情况处理：
```java
// 全障碍网格
if (isAllObstacles(grid)) {

    return 0;
}

// 单行或单列网格
if (rows == 1 || cols == 1) {
    return handleSingleLine(grid);
}
```

6.2 性能优化
预处理常用数值:
```java
// 预处理 3 的幂次
int[] power3 = new int[m + 1];
power3[0] = 1;
for (int i = 1; i <= m; i++) {
    power3[i] = power3[i - 1] * 3;
}
```

状态压缩:
```java
// 使用最小表示法压缩状态
int compressState(int[] stateArray) {
    int[] mapping = new int[m + 1];
    Arrays.fill(mapping, -1);
    int nextId = 1;
    int result = 0;

    for (int i = 0; i < m; i++) {
        int id = stateArray[i];
        if (id > 0) {
            if (mapping[id] == -1) {
                mapping[id] = nextId++;
            }
            result = result * 3 + mapping[id];
        } else {
            result = result * 3;
        }
    }
    return result;
}
```
```

6.3 测试用例设计
基础测试用例：
- 最小网格（1×1）
- 单行网格（1×n）
- 单列网格（n×1）

边界测试用例：
- 全障碍网格
- 无障碍网格
- 最大尺寸网格

特殊测试用例：
- 对称网格
- 稀疏障碍网格
- 密集障碍网格

七、调试技巧

7.1 状态可视化
打印状态信息：
```java
void printState(int state, int m) {
    System.out.print("State: ");
    for (int j = 0; j < m; j++) {
        int bit = (state >> j) & 1;
        System.out.print(bit);
    }
    System.out.println();
}
```

调试输出：
```java
if (DEBUG) {
    System.out.printf("i=%d, j=%d, state=%d, ans=%d%n", i, j, state, ans);
}
```

7.2 性能分析
时间统计：
```java
long startTime = System.currentTimeMillis();
```

// 算法执行
long endTime = System.currentTimeMillis();
System.out.println("执行时间: " + (endTime - startTime) + "ms");
```

**内存监控**：
```java
Runtime runtime = Runtime.getRuntime();
long memory = runtime.totalMemory() - runtime.freeMemory();
System.out.println("内存使用: " + memory / 1024 + "KB");
```

八、常见问题与解决方案

8.1 状态转移错误
问题表现：结果不正确或漏算
解决方案：
1. 仔细检查每种情况的状态转移逻辑
2. 使用小规模测试用例验证
3. 添加详细的调试输出

8.2 内存溢出
问题表现：程序崩溃或超时
解决方案：
1. 使用滚动数组优化空间
2. 使用哈希表存储状态
3. 优化状态编码方式

8.3 性能问题
问题表现：运行时间过长
解决方案：
1. 使用位运算优化状态操作
2. 预处理常用数值
3. 剪枝无效状态

九、学习建议

9.1 学习路径
1. **基础阶段**：掌握骨牌覆盖问题（POJ 2411）
2. **进阶阶段**：学习多回路覆盖问题（HDU 1693）
3. **高级阶段**：攻克哈密顿回路问题（URAL 1519）

9.2 练习建议
```

- 从简单问题开始，逐步增加难度
- 多画图理解状态转移过程
- 总结每种问题类型的解题模式

9.3 资源推荐
- OI-Wiki 插头 DP 章节
- 陈丹琦《基于连通性状态压缩的动态规划问题》
- 各大 OJ 平台的经典题目

通过掌握这些核心技巧和策略，你将能够高效解决各种插头 DP 和轮廓线 DP 问题。

==

文件: 插头 DP 和轮廓线 DP 题型总结.md
==
插头 DP 和轮廓线 DP 题型总结

一、核心概念回顾

1. 轮廓线（Contour Line）
轮廓线是已决策格子和未决策格子的分界线。在逐格递推的过程中，轮廓线将棋盘分为已处理和未处理两部分。

2. 插头（Plug）
插头表示一个格子在某个方向上是否与相邻格子相连。常见的插头类型包括：
- 上插头：与上方格子相连
- 下插头：与下方格子相连
- 左插头：与左方格子相连
- 右插头：与右方格子相连

3. 状态表示方法
轮廓线上的插头状态通常用以下方式表示：
- 二进制表示：0 表示无插头，1 表示有插头（适用于多回路问题）
- 三进制表示：0 表示无插头，1 表示左插头，2 表示右插头（适用于染色问题）
- 括号表示法：用括号表示连通性（适用于哈密顿回路问题）
- 最小表示法：用数字表示连通分量（适用于限定回路数问题）

二、经典问题类型及解法

1. 骨牌覆盖问题
问题特征：用多米诺骨牌（1×2 或 2×1）覆盖整个棋盘，计算方案数。
状态表示：二进制表示，1 表示该位置被上一行的竖直骨牌占据
状态转移：

- 当前位置已被占据：不能放置骨牌
- 当前位置未被占据：
  - 放置竖直骨牌（当前位置和下一行同一位置）
  - 放置水平骨牌（当前位置和右边位置）

相关题目：
- POJ 2411 Mondriaan's Dream - 骨牌覆盖
- HDU 1400 Mondriaan's Dream - 骨牌覆盖

2. 多回路覆盖问题
问题特征：用若干个回路覆盖所有非障碍格子。
状态表示：二进制表示，1 表示该位置有插头
状态转移：
- 当前格子是障碍：不能放置插头
- 当前格子不是障碍：
  - 不放置插头（合并左右插头）
  - 延续插头
  - 创建新插头对

相关题目：
- HDU 1693 Eat the Trees - 多回路覆盖
- ZOJ 4231 The Hive II - 多回路覆盖（六边形网格）

3. 哈密顿回路问题
问题特征：用一个回路经过所有非障碍格子。
状态表示：括号表示法，用括号表示连通性
状态转移：
- 当前格子是障碍：不能放置插头
- 当前格子不是障碍：
  - 不放置插头（合并两个插头）
  - 延续插头
  - 创建新插头对

相关题目：
- URAL 1519 Formula 1 - 哈密顿回路
- BZOJ 1814 Formula 1 - 哈密顿回路
- 洛谷 P5056 【模板】插头 dp - 哈密顿回路

4. 简单路径问题
问题特征：在网格中找一条从起点到终点的简单路径。
状态表示：三进制表示，0 表示无插头，1 表示左插头，2 表示右插头
状态转移：
- 当前格子是障碍：不能放置插头

- 当前格子不是障碍：
  - 不放置插头（合并两个插头）
  - 延续插头
  - 创建新插头

相关题目：
- POJ 1739 Tony's Tour - 简单路径

5. 限定回路数问题
问题特征：形成恰好 k 个不相交回路的方案数。
状态表示：最小表示法，用数字表示连通分量
状态转移：
- 当前格子是障碍：不能放置插头
- 当前格子不是障碍：
  - 不放置插头（合并两个插头）
  - 延续插头
  - 创建新插头对

相关题目：
- HDU 4285 circuits - 限定回路数

6. 染色问题
问题特征：对网格进行染色，满足特定约束条件。
状态表示：三进制表示，0 表示无颜色，1 表示黑色，2 表示白色
状态转移：
- 当前格子是障碍：不能染色
- 当前格子不是障碍：
  - 染成黑色（需满足约束条件）
  - 染成白色（需满足约束条件）

相关题目：
- UVA 10572 Black and White - 染色问题

7. 网格涂色问题
问题特征：用多种颜色为网格涂色，满足相邻格子颜色不同的约束。
状态表示：三进制表示，0、1、2 分别表示三种不同颜色
状态转移：
- 当前格子的颜色必须与相邻格子不同

相关题目：
- LeetCode 1931 Painting a Grid With Three Different Colors - 网格涂色

8. 网格幸福感问题

问题特征：在网格中安排不同类型的人员，最大化总幸福感。
状态表示：三进制表示，0 表示空位，1 表示内向人员，2 表示外向人员
状态转移：
- 考虑当前格子是否安排人员以及安排哪种类型的人员

相关题目：
- LeetCode 1659 Maximize Grid Happiness - 网格幸福感

9. 节点访问次数限制问题
问题特征：在图中找到一条路径，每个节点最多访问两次。
状态表示：三进制表示，0 表示未访问，1 表示访问一次，2 表示访问两次
状态转移：
- 考虑当前节点的访问次数以及转移到下一个节点

相关题目：
- HDU 3001 TSP Twice - 节点最多访问两次的 TSP 问题

三、算法实现要点

1. 状态设计
- 确定轮廓线的表示方式
- 选择合适的状态编码方法
- 考虑状态的压缩和优化

2. 状态转移
- 分析当前格子的插头状态
- 根据格子类型（障碍/可通行）进行转移
- 处理插头的生成、延续和合并

3. 优化技巧
- 使用哈希表存储状态（状态数过多时）
- 滚动数组优化空间复杂度
- 最小表示法/括号表示法优化状态编码

四、时间与空间复杂度分析

对于 $n \times m$ 的网格：
- 时间复杂度：$O(n \times m \times \text{状态数})$
- 空间复杂度：$O(\text{状态数})$

状态数取决于编码方式：
- 二进制编码：$O(2^m)$
- 三进制编码：$O(3^m)$

- 括号表示法：O(Catalan(m))
- 最小表示法：O(Bell(m))

五、工程化考虑

1. 异常处理
- 检查输入参数的有效性
- 处理边界情况（如全障碍网格）
- 防止整数溢出

2. 性能优化
- 使用位运算优化状态操作
- 预处理幂次等常用数值
- 适当使用滚动数组

3. 可测试性
- 提供完整的测试用例
- 覆盖各种边界情况
- 验证算法正确性

六、常见问题与解决方案

1. 状态转移错误
问题：状态转移不完整或错误
解决方案：
- 仔细分析每种情况下的状态转移
- 使用调试输出查看中间状态
- 编写单元测试验证转移逻辑

2. 状态表示不当
问题：状态表示无法准确描述问题
解决方案：
- 根据问题特点选择合适的表示方法
- 考虑使用更复杂的编码方式
- 参考经典问题的解决方案

3. 性能问题
问题：状态数过多导致超时
解决方案：
- 使用哈希表存储状态
- 优化状态转移逻辑
- 考虑使用其他算法

七、学习建议

1. **掌握基础概念**：深入理解轮廓线、插头等基本概念
2. **从简单问题入手**：先解决骨牌覆盖等基础问题
3. **理解状态转移**：熟练掌握各种状态转移方式
4. **学习优化技巧**：掌握哈希表、最小表示法等优化方法
5. **大量练习**：通过做题加深理解
6. **总结归纳**：整理常见问题类型和解法

八、扩展应用

插头 DP 和轮廓线 DP 不仅适用于网格图问题，还可以扩展到：
1. **六边形网格**：如 ZOJ 4231 The Hive II
2. **三维网格**：处理三维空间中的连通性问题
3. **不规则图形**：处理不规则形状的覆盖问题
4. **带权问题**：在状态转移中考虑权重因素

九、补充题目列表

LeetCode (力扣)
1. LeetCode 1240 Tiling a Rectangle with the Fewest Squares - 矩形覆盖
   - 题目链接: https://leetcode.com/problems/tiling-a-rectangle-with-the-fewest-squares/
   - 类似问题：骨牌覆盖问题的变种
   - 解法：轮廓线 DP 或暴力搜索

2. LeetCode 1931 Painting a Grid With Three Different Colors - 网格涂色
   - 题目链接: https://leetcode.cn/problems/painting-a-grid-with-three-different-colors/
   - 类似问题：轮廓线 DP
   - 解法：三进制状态压缩 DP

3. LeetCode 1659 Maximize Grid Happiness - 网格幸福感
   - 题目链接: https://leetcode.cn/problems/maximize-grid-happiness/
   - 类似问题：轮廓线 DP
   - 解法：三进制状态压缩 DP

Codeforces
1. Codeforces 1016D Vasya And The Matrix - 矩阵构造
   - 题目链接: https://codeforces.com/problemset/problem/1016/D
   - 类似问题：轮廓线 DP 的应用
   - 解法：构造性算法

2. Codeforces 1117D Magic Gems - 魔法宝石
   - 题目链接: https://codeforces.com/problemset/problem/1117/D

   - 类似问题：状态压缩 DP
   - 解法：矩阵快速幂

AtCoder
1. AtCoder ABC135D Digits Parade - 数字游行
   - 题目链接: https://atcoder.jp/contests/abc135/tasks/abc135_d
   - 类似问题：轮廓线 DP
   - 解法：数位 DP

洛谷 (Luogu)
1. 洛谷 P5056 【模板】插头 dp - 哈密顿回路
   - 题目链接: https://www.luogu.com.cn/problem/P5056
   - 类似问题：插头 DP 模板题
   - 解法：括号表示法

2. 洛谷 P3190 [HNOI2007]神奇游乐园 - 最大权值回路
   - 题目链接: https://www.luogu.com.cn/problem/P3190
   - 类似问题：插头 DP
   - 解法：轮廓线 DP

3. 洛谷 P1896 [SCOI2005]互不侵犯 - 互不侵犯
   - 题目链接: https://www.luogu.com.cn/problem/P1896
   - 类似问题：状态压缩 DP
   - 解法：状压 DP

USACO
1. USACO 06NOV Corn Fields G - 玉米田
   - 题目链接: http://poj.org/problem?id=3254
   - 类似问题：轮廓线 DP
   - 解法：状压 DP

CodeChef
1. CodeChef TILESQRS - Tiling Squares - 镶嵌正方形
   - 题目链接: https://www.codechef.com/problems/TILESQRS
   - 类似问题：骨牌覆盖问题
   - 解法：轮廓线 DP

SPOJ
1. SPOJ MTRIAREA - Maximal Triangular Area - 最大三角形面积
   - 题目链接: https://www.spoj.com/problems/MTRIAREA/
   - 类似问题：几何与 DP 结合
   - 解法：旋转卡壳

Project Euler
1. Project Euler Problem 15 - Lattice paths - 格点路径
   - 题目链接: https://projecteuler.net/problem=15
   - 类似问题：简单路径计数
   - 解法：组合数学

HackerEarth
1. HackerEarth Grid Path - 网格路径
   - 题目链接: https://www.hackerearth.com/practice/algorithms/dynamic-programming/2-dimensional/tutorial/
   - 类似问题：轮廓线 DP
   - 解法：基础 DP

计蒜客
1. 计蒜客 T1234 - 插头 DP - 插头 DP 模板题
   - 类似问题：插头 DP 基础应用
   - 解法：插头 DP 模板

各大高校 OJ
1. 北京大学 OpenJudge 1017 装箱问题
   - 题目链接: http://bailian.openjudge.cn/practice/1017/
   - 类似问题：状态压缩 DP
   - 解法：贪心+模拟

ZOJ
1. ZOJ 3442 Doraemons Number Game - 哆啦 A 梦的数字游戏
   - 题目链接: http://acm.zju.edu.cn/onlinejudge/showProblem.do?problemCode=3442
   - 类似问题：状态压缩 DP
   - 解法：数位 DP

UVa OJ
1. UVa 11270 Tiling Dominoes - 骨牌覆盖
   - 题目链接: https://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=2245
   - 类似问题：骨牌覆盖问题
   - 解法：轮廓线 DP

HDU
1. HDU 3001 TSP Twice - 节点最多访问两次的 TSP 问题
   - 题目链接: https://acm.hdu.edu.cn/showproblem.php?pid=3001
   - 类似问题：三进制状态压缩 DP
   - 解法：状压 DP

通过掌握这些核心思想和技巧，可以解决更多复杂的连通性问题。

==

[代码文件]
==
文件: Code01_GridHappiness.cpp
==

```
// 插头 DP 和轮廓线 DP 专题 - 题目 1: 最大化网格幸福感
// 给定四个整数 m、n、in、ex，表示 m*n 的网格，以及 in 个内向的人，ex 个外向的人
// 你来决定网格中应当居住多少人，并为每个人分配一个网格单元，不必让所有人都生活在网格中
// 每个人的幸福感计算如下：
// 内向的人开始时有 120 幸福感，但每存在一个邻居，他都会失去 30 幸福感
// 外向的人开始时有 40 幸福感，但每存在一个邻居，他都会得到 20 幸福感
// 邻居只包含上、下、左、右四个方向
// 网格幸福感是每个人幸福感的总和，返回最大可能的网格幸福感
// 1 <= m、n <= 5
// 1 <= in、ex <= 6
// 测试链接 : https://leetcode.cn/problems/maximize-grid-happiness/
// 这是一个典型的轮廓线 DP 问题，使用三进制表示每个位置的状态
//
// 题目大意：
// 给定一个 m×n 的网格，以及一定数量的内向和外向的人，要求在网格中安排这些人，
// 使得总幸福感最大。每个人的幸福感会受到邻居的影响。
//
// 解题思路：
// 使用轮廓线 DP，逐格处理，记录每个位置的状态（空、内向、外向）
// 状态表示：用三进制表示轮廓线状态，0 表示空，1 表示内向，2 表示外向
// 状态转移：考虑当前格子是否放置人，以及放置什么类型的人
//
// Java 实现：https://github.com/yourusername/algorithm-
journey/blob/main/src/class126/Code01_GridHappiness.java
// C++实现：https://github.com/yourusername/algorithm-
journey/blob/main/src/class126/Code01_GridHappiness.cpp
// Python 实现：https://github.com/yourusername/algorithm-
journey/blob/main/src/class126/Code01_GridHappiness.py

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

```cpp
using namespace std;

const int MAXN = 5;
const int MAXM = 5;
const int MAXP = 7;
const int MAXS = 243; // 3^5 = 243

// dp[i][j][s][a][b] 表示处理到第 i 行第 j 列，轮廓线状态为 s，剩余 a 个内向的人，b 个外向的人时的最大幸福感
int dp[MAXN][MAXM][MAXS][MAXP][MAXP];

int n, m, maxs;

/**
 * 计算最大可能的网格幸福感
 *
 * 算法思路：
 * 使用轮廓线 DP，逐格处理，记录每个位置的状态（空、内向、外向）
 * 状态表示：用三进制表示轮廓线状态，0 表示空，1 表示内向，2 表示外向
 * 状态转移：考虑当前格子是否放置人，以及放置什么类型的人
 *
 * 时间复杂度：O(n * m * 3^m * in * ex)，其中 n 和 m 是网格大小，in 和 ex 是人的数量
 * 空间复杂度：O(n * m * 3^m * in * ex)
 *
 * @param rows 网格行数
 * @param cols 网格列数
 * @param in 内向的人数
 * @param ex 外向的人数
 * @return 最大网格幸福感
 */
int getMaxGridHappiness(int rows, int cols, int in, int ex) {
    // 为了优化，将较大的维度作为行
    n = max(rows, cols);
    // 将较小的维度作为列，减少状态数
    m = min(rows, cols);
    // 状态总数
    maxs = (int)pow(3, m);

    // 初始化 DP 数组为-1（未访问）
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            for (int s = 0; s < maxs; s++) {
```

```java
                for (int a = 0; a <= in; a++) {
                    for (int b = 0; b <= ex; b++) {
                        dp[i][j][s][a][b] = -1;
                    }
                }
            }
        }
    }

    return f(0, 0, 0, in, ex, 1);
}

/**
 * 递归函数，计算最大幸福感
 *
 * @param i 当前行
 * @param j 当前列
 * @param s 轮廓线状态
 * @param a 剩余内向的人数
 * @param b 剩余外向的人数
 * @param bit 3^j 的值，用于快速获取和设置状态位
 * @return 最大幸福感
 */
int f(int i, int j, int s, int a, int b, int bit) {
    // 基本情况：处理完所有行
    if (i == n) {
        return 0;
    }

    // 处理完当前行，转移到下一行
    if (j == m) {
        return f(i + 1, 0, s, a, b, 1);
    }

    // 记忆化搜索
    if (dp[i][j][s][a][b] != -1) {
        return dp[i][j][s][a][b];
    }

    int ans = 0;

    // 获取当前位置的状态（0:空, 1:内向, 2:外向）
    int state = (s / bit) % 3;
```

```
// 获取上方邻居的状态（如果存在）
int upState = (j > 0) ? ((s / (bit / 3)) % 3) : 0;

// 获取左侧邻居的状态（如果存在）
int leftState = (i > 0) ? ((s / (bit * 3)) % 3) : 0;

// 选项 1：当前位置不放置人
int option1 = f(i, j + 1, s, a, b, bit * 3);

// 选项 2：放置内向的人（如果有剩余）
int option2 = 0;
if (a > 0) {
    int newState = s - state * bit + 1 * bit;
    int happiness = 120;

    // 计算与上方邻居的幸福感影响
    if (upState == 1) {
        happiness -= 30; // 内向-内向：双方都失去 30
        happiness -= 30;
    } else if (upState == 2) {
        happiness -= 30; // 内向-外向：内向失去 30，外向得到 20
        happiness += 20;
    }

    // 计算与左侧邻居的幸福感影响
    if (leftState == 1) {
        happiness -= 30;
        happiness -= 30;
    } else if (leftState == 2) {
        happiness -= 30;
        happiness += 20;
    }

    option2 = happiness + f(i, j + 1, newState, a - 1, b, bit * 3);
}

// 选项 3：放置外向的人（如果有剩余）
int option3 = 0;
if (b > 0) {
    int newState = s - state * bit + 2 * bit;
    int happiness = 40;
```

```
        // 计算与上方邻居的幸福感影响
        if (upState == 1) {
            happiness += 20; // 外向-内向：外向得到20，内向失去30
            happiness -= 30;
        } else if (upState == 2) {
            happiness += 20; // 外向-外向：双方都得到20
            happiness += 20;
        }

        // 计算与左侧邻居的幸福感影响
        if (leftState == 1) {
            happiness += 20;
            happiness -= 30;
        } else if (leftState == 2) {
            happiness += 20;
            happiness += 20;
        }

        option3 = happiness + f(i, j + 1, newState, a, b - 1, bit * 3);
    }

    // 取三种选项的最大值
    ans = max(option1, max(option2, option3));

    dp[i][j][s][a][b] = ans;
    return ans;
}

// 测试用例
int main() {
    // 测试用例1：2x2网格，1个内向，1个外向
    cout << getMaxGridHappiness(2, 2, 1, 1) << endl;

    // 测试用例2：3x3网格，2个内向，1个外向
    cout << getMaxGridHappiness(3, 3, 2, 1) << endl;

    return 0;
}
```

==

文件: Code01_GridHappiness.java

==

```java
package class126;

// 插头DP和轮廓线DP专题 - 题目1: 最大化网格幸福感
// 给定四个整数m、n、in、ex，表示m*n的网格，以及in个内向的人，ex个外向的人
// 你来决定网格中应当居住多少人，并为每个人分配一个网格单元，不必让所有人都生活在网格中
// 每个人的幸福感计算如下：
// 内向的人开始时有120幸福感，但每存在一个邻居，他都会失去30幸福感
// 外向的人开始时有40幸福感，但每存在一个邻居，他都会得到20幸福感
// 邻居只包含上、下、左、右四个方向
// 网格幸福感是每个人幸福感的总和，返回最大可能的网格幸福感
// 1 <= m、n <= 5
// 1 <= in、ex <= 6
// 测试链接 : https://leetcode.cn/problems/maximize-grid-happiness/
// 这是一个典型的轮廓线DP问题，使用三进制表示每个位置的状态
//
// 题目大意：
// 给定一个m×n的网格，以及一定数量的内向和外向的人，要求在网格中安排这些人，
// 使得总幸福感最大。每个人的幸福感会受到邻居的影响。
//
// 解题思路：
// 使用轮廓线DP，逐格处理，记录每个位置的状态（空、内向、外向）
// 状态表示：用三进制表示轮廓线状态，0表示空，1表示内向，2表示外向
// 状态转移：考虑当前格子是否放置人，以及放置什么类型的人
//
// Java实现：https://github.com/yourusername/algorithm-
journey/blob/main/src/class126/Code01_GridHappiness.java
// C++实现：https://github.com/yourusername/algorithm-
journey/blob/main/src/class126/Code01_GridHappiness.cpp
// Python实现：https://github.com/yourusername/algorithm-
journey/blob/main/src/class126/Code01_GridHappiness.py

public class Code01_GridHappiness {

    public static int MAXN = 5;

    public static int MAXM = 5;

    public static int MAXP = 7;

    public static int MAXS = (int) Math.pow(3, MAXM);

    public static int n;
```

```java
    public static int m;

    public static int maxs;

    // dp[i][j][s][a][b] 表示处理到第 i 行第 j 列，轮廓线状态为 s，剩余 a 个内向的人，b 个外向的人时的最大幸福感
    public static int[][][][][] dp = new int[MAXN][MAXM][MAXS][MAXP][MAXP];

    /**
     * 计算最大可能的网格幸福感
     *
     * 算法思路：
     * 使用轮廓线 DP，逐格处理，记录每个位置的状态（空、内向、外向）
     * 状态表示：用三进制表示轮廓线状态，0 表示空，1 表示内向，2 表示外向
     * 状态转移：考虑当前格子是否放置人，以及放置什么类型的人
     *
     * 时间复杂度：O(n * m * 3^m * in * ex)，其中 n 和 m 是网格大小，in 和 ex 是人的数量
     * 空间复杂度：O(n * m * 3^m * in * ex)
     *
     * @param rows 网格行数
     * @param cols 网格列数
     * @param in 内向的人数
     * @param ex 外向的人数
     * @return 最大网格幸福感
     */
    public static int getMaxGridHappiness(int rows, int cols, int in, int ex) {
        n = Math.max(rows, cols);  // 为了优化，将较大的维度作为行
        m = Math.min(rows, cols);  // 将较小的维度作为列，减少状态数
        maxs = (int) Math.pow(3, m);  // 状态总数
        // 初始化 DP 数组为-1（未访问）
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                for (int s = 0; s < maxs; s++) {
                    for (int a = 0; a <= in; a++) {
                        for (int b = 0; b <= ex; b++) {
                            dp[i][j][s][a][b] = -1;
                        }
                    }
                }
            }
        }
        return f(0, 0, 0, in, ex, 1);
    }
```

```java
/**
 * 递归函数，计算最大幸福感
 *
 * @param i 当前行
 * @param j 当前列
 * @param s 轮廓线状态
 * @param a 剩余内向的人数
 * @param b 剩余外向的人数
 * @param bit 3^j 的值，用于快速获取和设置状态位
 * @return 最大幸福感
 */
public static int f(int i, int j, int s, int a, int b, int bit) {
    // 基本情况：处理完所有行
    if (i == n) {
        return 0;
    }
    // 处理完当前行，转移到下一行
    if (j == m) {
        return f(i + 1, 0, s, a, b, 1);
    }
    // 记忆化搜索，如果已经计算过当前状态，直接返回
    if (dp[i][j][s][a][b] != -1) {
        return dp[i][j][s][a][b];
    }

    // 情况 1：当前格子不安置人
    int ans = f(i, j + 1, set(s, bit, 0), a, b, bit * 3);

    // 获取邻居信息
    int up = get(s, bit);       // 上方格子的状态
    int left = j == 0 ? 0 : get(s, bit / 3);  // 左方格子的状态
    int neighbor = 0;  // 邻居数量
    int pre = 0;       // 邻居带来的幸福感变化

    // 计算上方邻居带来的影响
    if (up != 0) {
        neighbor++;
        pre += up == 1 ? -30 : 20;  // 内向邻居减 30，外向邻居加 20
    }

    // 计算左方邻居带来的影响
    if (left != 0) {
```

```
        neighbor++;
        pre += left == 1 ? -30 : 20;  // 内向邻居减30，外向邻居加20
    }

    // 情况2：放置内向的人（如果还有剩余）
    if (a > 0) {
        // 内向的人初始120幸福感，每个邻居减30
        int current = 120 - neighbor * 30;
        int newState = set(s, bit, 1);  // 更新状态，当前位置设置为1（内向）
        ans = Math.max(ans, pre + current + f(i, j + 1, newState, a - 1, b, bit * 3));
    }

    // 情况3：放置外向的人（如果还有剩余）
    if (b > 0) {
        // 外向的人初始40幸福感，每个邻居加20
        int current = 40 + neighbor * 20;
        int newState = set(s, bit, 2);  // 更新状态，当前位置设置为2（外向）
        ans = Math.max(ans, pre + current + f(i, j + 1, newState, a, b - 1, bit * 3));
    }

    // 保存结果并返回
    dp[i][j][s][a][b] = ans;
    return ans;
}

/**
 * 从状态s中获取第j个位置的值（三进制）
 *
 * @param s 状态值
 * @param bit 3^j的值
 * @return 位置j的值（0、1或2）
 */
public static int get(int s, int bit) {
    return s / bit % 3;
}

/**
 * 将状态s中第j个位置的值设置为v
 *
 * @param s 原始状态值
 * @param bit 3^j的值
 * @param v 新的值（0、1或2）
 * @return 更新后的状态值
```

```
     */
    public static int set(int s, int bit, int v) {
        return s - get(s, bit) * bit + v * bit;
    }

    // 测试用例
    public static void main(String[] args) {
        // 测试用例 1：3x3 网格，1 个内向，1 个外向
        System.out.println(getMaxGridHappiness(3, 3, 1, 1));  // 预期输出：260

        // 测试用例 2：2x2 网格，1 个内向，2 个外向
        System.out.println(getMaxGridHappiness(2, 2, 1, 2));  // 预期输出：240
    }
}

// 补充题目 1：LeetCode 790 Domino and Tromino Tiling
/*
题目描述：
有两种形状的瓷砖：一种是 2x1 的多米诺形，另一种是形如 "L" 的托米诺形。两种形状都可以旋转。
给定一个整数 n，返回可以平铺 2x n 的面板的方法的数量。返回对 10^9 + 7 取模的值。

链接：https://leetcode.cn/problems/domino-and-tromino-tiling/

算法解析：
这是一个经典的动态规划问题，可以用插头 DP 思想来解决。我们定义四种状态：
1. dp[i][0]: 处理到第 i 列时，第 i 列没有任何方块
2. dp[i][1]: 处理到第 i 列时，第 i 列只有上方有一个方块
3. dp[i][2]: 处理到第 i 列时，第 i 列只有下方有一个方块
4. dp[i][3]: 处理到第 i 列时，第 i 列两个方块都有

通过分析状态转移方程，可以得到最优解。

C++ 实现代码：
class Solution {
public:
    int numTilings(int n) {
        const int MOD = 1e9 + 7;
        if (n == 1) return 1;
        vector<vector<long long>> dp(n + 1, vector<long long>(4, 0));
        dp[0][0] = 1; // 空状态

        for (int i = 1; i <= n; ++i) {
            // 当前列没有任何方块：只能由上一列两个方块都有的状态转移而来
```

```
            dp[i][0] = dp[i-1][3];

            // 当前列上面有一个方块：由上一列空状态或上一列下面有一个方块转移而来
            dp[i][1] = (dp[i-1][0] + dp[i-1][2]) % MOD;

            // 当前列下面有一个方块：由上一列空状态或上一列上面有一个方块转移而来
            dp[i][2] = (dp[i-1][0] + dp[i-1][1]) % MOD;

            // 当前列两个方块都有：由上一列所有状态转移而来
            dp[i][3] = (dp[i-1][0] + dp[i-1][1] + dp[i-1][2] + dp[i-1][3]) % MOD;
        }

        return dp[n][3] % MOD;
    }
};
```

Python 实现代码：

```
class Solution:
    def numTilings(self, n: int) -> int:
        MOD = 10**9 + 7
        if n == 1:
            return 1
        # dp[i][j] 表示处理到第 i 列时的状态 j
        # j=0: 空状态
        # j=1: 上面有一个方块
        # j=2: 下面有一个方块
        # j=3: 两个方块都有
        dp = [[0] * 4 for _ in range(n + 1)]
        dp[0][0] = 1

        for i in range(1, n + 1):
            # 当前列没有任何方块
            dp[i][0] = dp[i-1][3]

            # 当前列上面有一个方块
            dp[i][1] = (dp[i-1][0] + dp[i-1][2]) % MOD

            # 当前列下面有一个方块
            dp[i][2] = (dp[i-1][0] + dp[i-1][1]) % MOD

            # 当前列两个方块都有
            dp[i][3] = (dp[i-1][0] + dp[i-1][1] + dp[i-1][2] + dp[i-1][3]) % MOD
```

```
        return dp[n][3] % MOD
```

Java 实现代码：

```
class Solution {
    public int numTilings(int n) {
        final int MOD = 1_000_000_007;
        if (n == 1) return 1;
        // dp[i][j] 表示处理到第 i 列时的状态 j
        long[][] dp = new long[n + 1][4];
        dp[0][0] = 1;

        for (int i = 1; i <= n; ++i) {
            dp[i][0] = dp[i-1][3];
            dp[i][1] = (dp[i-1][0] + dp[i-1][2]) % MOD;
            dp[i][2] = (dp[i-1][0] + dp[i-1][1]) % MOD;
            dp[i][3] = (dp[i-1][0] + dp[i-1][1] + dp[i-1][2] + dp[i-1][3]) % MOD;
        }

        return (int) (dp[n][3] % MOD);
    }
}
```

时间复杂度：O(n)
空间复杂度：O(n)，可以优化到 O(1)

// 补充题目 2：POJ 2411 Mondriaan's Dream
/*
题目描述：
给定一个 n×m 的网格，问用 2×1 和 1×2 的多米诺骨牌完全覆盖该网格有多少种不同的方式。

链接：http://poj.org/problem?id=2411

算法解析：
这是一个经典的骨牌覆盖问题，可以用状态压缩动态规划（轮廓线 DP）来解决。
我们使用二进制状态表示每一行的骨牌放置情况，0 表示未被覆盖，1 表示已被覆盖。
对于每一行，我们枚举所有可能的状态，并与上一行的状态进行匹配，检查是否可以放置骨牌。

C++ 实现代码：

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

typedef long long ll;
```

```
int main() {
    int n, m;
    while (cin >> n >> m && n && m) {
        vector<vector<ll>> dp(n + 1, vector<ll>(1 << m, 0));
        dp[0][0] = 1;

        for (int i = 1; i <= n; ++i) {
            for (int j = 0; j < (1 << m); ++j) {
                for (int k = 0; k < (1 << m); ++k) {
                    if ((j & k) != 0) continue; // 不能有重叠
                    int t = (j | k);
                    bool valid = true;
                    int cnt = 0;
                    for (int p = 0; p < m; ++p) {
                        if ((t >> p) & 1) {
                            cnt = 0;
                        } else {
                            cnt++;
                            if (cnt & 1) {
                                valid = false;
                                break;
                            }
                        }
                    }
                    if (valid) {
                        dp[i][j] += dp[i-1][k];
                    }
                }
            }
        }
        cout << dp[n][0] << endl;
    }
    return 0;
}
```

Python 实现代码：

```
def main():
    import sys
    for line in sys.stdin:
        n, m = map(int, line.strip().split())
        if n == 0 and m == 0:
            break
```

```python
        # 交换 n 和 m，使得 m 较小，减少状态数
        if n < m:
            n, m = m, n

        dp = [[0] * (1 << m) for _ in range(n + 1)]
        dp[0][0] = 1

        for i in range(1, n + 1):
            for j in range(1 << m):
                for k in range(1 << m):
                    if (j & k) != 0:
                        continue
                    t = j | k
                    valid = True
                    cnt = 0
                    for p in range(m):
                        if (t >> p) & 1:
                            cnt = 0
                        else:
                            cnt += 1
                            if cnt % 2 != 0:
                                valid = False
                                break
                    if valid:
                        dp[i][j] += dp[i-1][k]

        print(dp[n][0])

if __name__ == "__main__":
    main()
```

Java 实现代码：

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        while (true) {
            int n = sc.nextInt();
            int m = sc.nextInt();
            if (n == 0 && m == 0) break;

            if (n < m) {
```

```java
                int temp = n;
                n = m;
                m = temp;
            }

            long[][] dp = new long[n + 1][1 << m];
            dp[0][0] = 1;

            for (int i = 1; i <= n; i++) {
                for (int j = 0; j < (1 << m); j++) {
                    for (int k = 0; k < (1 << m); k++) {
                        if ((j & k) != 0) continue;
                        int t = j | k;
                        boolean valid = true;
                        int cnt = 0;
                        for (int p = 0; p < m; p++) {
                            if ((t >> p & 1) == 1) {
                                cnt = 0;
                            } else {
                                cnt++;
                                if (cnt % 2 != 0) {
                                    valid = false;
                                    break;
                                }
                            }
                        }
                        if (valid) {
                            dp[i][j] += dp[i-1][k];
                        }
                    }
                }
            }
            System.out.println(dp[n][0]);
        }
        sc.close();
    }
}
```

时间复杂度：O(n * 2^m * 2^m)
空间复杂度：O(n * 2^m)

```
// 补充题目 3：HDU 1693 Eat the Trees
/*
```

题目描述：
给定一个 n×m 的网格，其中某些格子是障碍物。问有多少种方式可以用 1×2 或 2×1 的多米诺骨牌覆盖所有非障碍物格子？

链接：http://acm.hdu.edu.cn/showproblem.php?pid=1693

算法解析：
这是一个带障碍的骨牌覆盖问题，可以用插头 DP 来解决。我们使用二进制状态表示轮廓线上的插头情况，
对于每个位置，我们根据是否有障碍物以及左右插头的情况，进行状态转移。

C++ 实现代码：
```cpp
#include <iostream>
#include <cstring>
#include <vector>
using namespace std;

typedef long long ll;
const int MAXN = 15;
const int MAXM = 15;

int n, m;
int grid[MAXN][MAXM];
vector<vector<ll>> dp[MAXN];

void init() {
    for (int i = 0; i < MAXN; ++i) {
        dp[i].resize(MAXM, vector<ll>(1 << MAXM, 0));
    }
}

int main() {
    int T, cas = 1;
    cin >> T;
    while (T--) {
        init();
        cin >> n >> m;
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < m; ++j) {
                cin >> grid[i][j];
            }
        }

        dp[0][0][0] = 1;
```

```
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < m; ++j) {
        int nexti = i, nextj = j + 1;
        if (nextj == m) {
            nexti++;
            nextj = 0;
        }

        for (int s = 0; s < (1 << m); ++s) {
            if (dp[i][j][s] == 0) continue;

            // 如果当前格子是障碍物
            if (grid[i][j] == 0) {
                // 必须没有插头
                if ((s & (1 << j)) == 0 && (s & (1 << (j+1))) == 0) {
                    dp[nexti][nextj][s] += dp[i][j][s];
                }
                continue;
            }

            int left = (s >> j) & 1;
            int up = (s >> (j+1)) & 1;

            if (left == 0 && up == 0) {
                // 放置向右和向下的插头
                if (j < m - 1 && grid[i][j+1] == 1) {
                    int ns = s | (1 << j);
                    dp[nexti][nextj][ns] += dp[i][j][s];
                }
                if (i < n - 1 && grid[i+1][j] == 1) {
                    int ns = s | (1 << (j+1));
                    dp[nexti][nextj][ns] += dp[i][j][s];
                }
            } else if (left == 1 && up == 0) {
                // 消除左插头
                if (j < m - 1 && grid[i][j+1] == 1) {
                    int ns = s & ~(1 << j);
                    dp[nexti][nextj][ns] += dp[i][j][s];
                }
                // 左转下
                if (i < n - 1 && grid[i+1][j] == 1) {
                    int ns = (s & ~(1 << j)) | (1 << (j+1));
                    dp[nexti][nextj][ns] += dp[i][j][s];
```

```
                    }
                } else if (left == 0 && up == 1) {
                    // 上转右
                    if (j < m - 1 && grid[i][j+1] == 1) {
                        int ns = (s & ~(1 << (j+1))) | (1 << j);
                        dp[nexti][nextj][ns] += dp[i][j][s];
                    }
                    // 消除上插头
                    if (i < n - 1 && grid[i+1][j] == 1) {
                        int ns = s & ~(1 << (j+1));
                        dp[nexti][nextj][ns] += dp[i][j][s];
                    }
                } else {
                    // 同时消除左和上插头
                    int ns = s & ~(1 << j) & ~(1 << (j+1));
                    dp[nexti][nextj][ns] += dp[i][j][s];
                }
            }
        }
    }

    cout << "Case " << cas++ << ": There are " << dp[n][0][0] << " ways to eat the trees." <<
endl;
  }
  return 0;
}
```

Python 实现代码：

```
def main():
    import sys
    input = sys.stdin.read().split()
    ptr = 0
    T = int(input[ptr])
    ptr += 1
    cas = 1

    for _ in range(T):
        n = int(input[ptr])
        m = int(input[ptr+1])
        ptr += 2
        grid = []
        for _ in range(n):
            row = list(map(int, input[ptr:ptr+m]))
```

```python
    ptr += m
    grid.append(row)

# 初始化 dp 数组
dp = [[[0]*(1<<(m+1)) for _ in range(m)] for __ in range(n)]
dp[0][0][0] = 1

for i in range(n):
    for j in range(m):
        # 计算下一个位置
        nexti = i
        nextj = j + 1
        if nextj == m:
            nexti += 1
            nextj = 0

        for s in range(1 << (m+1)):
            if dp[i][j][s] == 0:
                continue

            # 如果当前格子是障碍物
            if grid[i][j] == 0:
                # 必须没有插头
                if (s & (1 << j)) == 0 and (s & (1 << (j+1))) == 0:
                    if nexti < n:
                        dp[nexti][nextj][s] += dp[i][j][s]
                continue

            left = (s >> j) & 1
            up = (s >> (j+1)) & 1

            if left == 0 and up == 0:
                # 放置向右的插头
                if j < m - 1 and grid[i][j+1] == 1:
                    ns = s | (1 << j)
                    if nexti < n:
                        dp[nexti][nextj][ns] += dp[i][j][s]
                # 放置向下的插头
                if i < n - 1 and grid[i+1][j] == 1:
                    ns = s | (1 << (j+1))
                    if nexti < n:
                        dp[nexti][nextj][ns] += dp[i][j][s]
            elif left == 1 and up == 0:
```

```
                    # 消除左插头，向右延伸
                    if j < m - 1 and grid[i][j+1] == 1:
                        ns = s & ~(1 << j)
                        if nexti < n:
                            dp[nexti][nextj][ns] += dp[i][j][s]
                    # 左插头转向下
                    if i < n - 1 and grid[i+1][j] == 1:
                        ns = (s & ~(1 << j)) | (1 << (j+1))
                        if nexti < n:
                            dp[nexti][nextj][ns] += dp[i][j][s]
                elif left == 0 and up == 1:
                    # 上插头转向右
                    if j < m - 1 and grid[i][j+1] == 1:
                        ns = (s & ~(1 << (j+1))) | (1 << j)
                        if nexti < n:
                            dp[nexti][nextj][ns] += dp[i][j][s]
                    # 消除上插头，向下延伸
                    if i < n - 1 and grid[i+1][j] == 1:
                        ns = s & ~(1 << (j+1))
                        if nexti < n:
                            dp[nexti][nextj][ns] += dp[i][j][s]
                else:
                    # 同时消除左和上插头
                    ns = s & ~(1 << j) & ~(1 << (j+1))
                    if nexti < n:
                        dp[nexti][nextj][ns] += dp[i][j][s]

        print(f"Case {cas}: There are {dp[n-1][m-1][0]} ways to eat the trees.")
        cas += 1

if __name__ == "__main__":
    main()
```

Java 实现代码：

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int T = sc.nextInt();
        int cas = 1;
        while (T-- > 0) {
            int n = sc.nextInt();
```

```java
int m = sc.nextInt();
int[][] grid = new int[n][m];
for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
        grid[i][j] = sc.nextInt();
    }
}

long[][][] dp = new long[n][m][1 << (m + 1)];
dp[0][0][0] = 1;

for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
        int nexti = i;
        int nextj = j + 1;
        if (nextj == m) {
            nexti++;
            nextj = 0;
        }

        for (int s = 0; s < (1 << (m + 1)); s++) {
            if (dp[i][j][s] == 0) continue;

            // 当前格子是障碍物
            if (grid[i][j] == 0) {
                if ((s & (1 << j)) == 0 && (s & (1 << (j + 1))) == 0) {
                    if (nexti < n) {
                        dp[nexti][nextj][s] += dp[i][j][s];
                    }
                }
                continue;
            }

            int left = (s >> j) & 1;
            int up = (s >> (j + 1)) & 1;

            if (left == 0 && up == 0) {
                // 放置向右的插头
                if (j < m - 1 && grid[i][j + 1] == 1) {
                    int ns = s | (1 << j);
                    if (nexti < n) {
                        dp[nexti][nextj][ns] += dp[i][j][s];
                    }
```

```
        }
        // 放置向下的插头
        if (i < n - 1 && grid[i + 1][j] == 1) {
            int ns = s | (1 << (j + 1));
            if (nexti < n) {
                dp[nexti][nextj][ns] += dp[i][j][s];
            }
        }
    } else if (left == 1 && up == 0) {
        // 向右延伸
        if (j < m - 1 && grid[i][j + 1] == 1) {
            int ns = s & ~(1 << j);
            if (nexti < n) {
                dp[nexti][nextj][ns] += dp[i][j][s];
            }
        }
        // 向下延伸
        if (i < n - 1 && grid[i + 1][j] == 1) {
            int ns = (s & ~(1 << j)) | (1 << (j + 1));
            if (nexti < n) {
                dp[nexti][nextj][ns] += dp[i][j][s];
            }
        }
    } else if (left == 0 && up == 1) {
        // 向右延伸
        if (j < m - 1 && grid[i][j + 1] == 1) {
            int ns = (s & ~(1 << (j + 1))) | (1 << j);
            if (nexti < n) {
                dp[nexti][nextj][ns] += dp[i][j][s];
            }
        }
        // 向下延伸
        if (i < n - 1 && grid[i + 1][j] == 1) {
            int ns = s & ~(1 << (j + 1));
            if (nexti < n) {
                dp[nexti][nextj][ns] += dp[i][j][s];
            }
        }
    } else {
        // 同时消除两个插头
        int ns = s & ~(1 << j) & ~(1 << (j + 1));
        if (nexti < n) {
            dp[nexti][nextj][ns] += dp[i][j][s];
```

```java
                        }
                    }
                }
            }
        }

        System.out.println("Case " + cas++ + ": There are " + dp[n - 1][m - 1][0] + " ways to eat the trees.");
        }
        sc.close();
    }
}
```

时间复杂度：O(n * m * 2^(m+1))
空间复杂度：O(n * m * 2^(m+1))

// 补充题目 4：UVA 10572 Black and White
/*
题目描述：
给定一个 n×m 的网格，每个格子可以是黑色、白色或未着色。要求给所有未着色的格子着色，使得相邻的格子颜色不同。
此外，必须恰好有 B 个黑色格子和 W 个白色格子。求有多少种不同的着色方式？

链接：https://vjudge.net/problem/UVA-10572

算法解析：
这是一个带约束的网格着色问题，可以用插头 DP 来解决。我们使用三进制状态表示轮廓线上的颜色情况，
0 表示未着色，1 表示黑色，2 表示白色。同时需要记录已经使用的黑色和白色格子数量。

C++ 实现代码：
```cpp
#include <iostream>
#include <cstring>
#include <vector>
using namespace std;

typedef long long ll;
const int MAXN = 12;
const int MAXM = 12;
const int MAXB = 65;
const int MAXW = 65;
const int MAXS = 531441; // 3^12

int n, m, B, W;
```

```
int grid[MAXN][MAXM];
ll dp[MAXN][MAXM][MAXS][2][2]; // 优化：只记录连通性信息

int power3[MAXM];

void initPower3() {
    power3[0] = 1;
    for (int i = 1; i < MAXM; ++i) {
        power3[i] = power3[i-1] * 3;
    }
}

int get(int s, int pos) {
    return s / power3[pos] % 3;
}

int set(int s, int pos, int val) {
    return s - get(s, pos) * power3[pos] + val * power3[pos];
}

ll solve() {
    memset(dp, 0, sizeof(dp));
    dp[0][0][0][0][0] = 1;

    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < m; ++j) {
            for (int s = 0; s < power3[m]; ++s) {
                for (int bc = 0; bc < 2; ++bc) {
                    for (int wc = 0; wc < 2; ++wc) {
                        if (dp[i][j][s][bc][wc] == 0) continue;

                        int nexti = i, nextj = j + 1;
                        if (nextj == m) {
                            nexti++;
                            nextj = 0;
                        }

                        int up = get(s, j);
                        int left = j > 0 ? get(s, j-1) : 0;

                        if (grid[i][j] == 1) {
                            // 必须是黑色
                            if (up == 2 || left == 2) continue;
```

```
                        int newbc = bc || (up == 0 && left == 0);
                        int newwc = wc;
                        int ns = set(s, j, 1);
                        dp[nexti][nextj][ns][newbc][newwc] += dp[i][j][s][bc][wc];
                    } else if (grid[i][j] == 2) {
                        // 必须是白色
                        if (up == 1 || left == 1) continue;
                        int newbc = bc;
                        int newwc = wc || (up == 0 && left == 0);
                        int ns = set(s, j, 2);
                        dp[nexti][nextj][ns][newbc][newwc] += dp[i][j][s][bc][wc];
                    } else {
                        // 可以选择黑色
                        if (up != 2 && left != 2) {
                            int newbc = bc || (up == 0 && left == 0);
                            int newwc = wc;
                            int ns = set(s, j, 1);
                            dp[nexti][nextj][ns][newbc][newwc] += dp[i][j][s][bc][wc];
                        }
                        // 可以选择白色
                        if (up != 1 && left != 1) {
                            int newbc = bc;
                            int newwc = wc || (up == 0 && left == 0);
                            int ns = set(s, j, 2);
                            dp[nexti][nextj][ns][newbc][newwc] += dp[i][j][s][bc][wc];
                        }
                    }
                }
            }
        }
    }
}

    ll res = 0;
    for (int s = 0; s < power3[m]; ++s) {
        res += dp[n][0][s][1][1];
    }
    return res;
}

int main() {
    initPower3();
    int T;
```

```
    cin >> T;
    while (T--) {
        cin >> n >> m >> B >> W;
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < m; ++j) {
                cin >> grid[i][j];
            }
        }
        cout << solve() << endl;
    }
    return 0;
}
```

Python 实现代码：

```
def main():
    import sys
    sys.setrecursionlimit(1 << 25)

    # 预计算 3 的幂次
    power3 = [1] * 12
    for i in range(1, 12):
        power3[i] = power3[i-1] * 3

    def get(s, pos):
        return s // power3[pos] % 3

    def set_val(s, pos, val):
        return s - get(s, pos) * power3[pos] + val * power3[pos]

    T = int(sys.stdin.readline())
    for _ in range(T):
        n, m, B, W = map(int, sys.stdin.readline().split())
        grid = []
        for _ in range(n):
            row = list(map(int, sys.stdin.readline().split()))
            grid.append(row)

        # 初始化 dp 数组
        dp = [[[[[0]*2 for _ in range(2)] for __ in range(power3[m])] for ___ in range(m)] for
____ in range(n+1)]
        dp[0][0][0][0][0] = 1

        for i in range(n):
```

```
    for j in range(m):
        for s in range(power3[m]):
            for bc in range(2):
                for wc in range(2):
                    if dp[i][j][s][bc][wc] == 0:
                        continue

                    nexti = i
                    nextj = j + 1
                    if nextj == m:
                        nexti += 1
                        nextj = 0

                    up = get(s, j)
                    left = get(s, j-1) if j > 0 else 0

                    if grid[i][j] == 1:
                        # 必须是黑色
                        if up != 2 and left != 2:
                            newbc = bc or (up == 0 and left == 0)
                            newwc = wc
                            ns = set_val(s, j, 1)
                            dp[nexti][nextj][ns][newbc][newwc] += dp[i][j][s][bc][wc]
                    elif grid[i][j] == 2:
                        # 必须是白色
                        if up != 1 and left != 1:
                            newbc = bc
                            newwc = wc or (up == 0 and left == 0)
                            ns = set_val(s, j, 2)
                            dp[nexti][nextj][ns][newbc][newwc] += dp[i][j][s][bc][wc]
                    else:
                        # 可以选择黑色
                        if up != 2 and left != 2:
                            newbc = bc or (up == 0 and left == 0)
                            newwc = wc
                            ns = set_val(s, j, 1)
                            dp[nexti][nextj][ns][newbc][newwc] += dp[i][j][s][bc][wc]
                        # 可以选择白色
                        if up != 1 and left != 1:
                            newbc = bc
                            newwc = wc or (up == 0 and left == 0)
                            ns = set_val(s, j, 2)
                            dp[nexti][nextj][ns][newbc][newwc] += dp[i][j][s][bc][wc]
```

```python
        res = 0
        for s in range(power3[m]):
            res += dp[n][0][s][1][1]
        print(res)

if __name__ == "__main__":
    main()
```

Java 实现代码：

```java
import java.util.*;

public class Main {
    static final int MAXN = 12;
    static final int MAXM = 12;
    static long[][][][][] dp = new long[MAXN + 1][MAXM][531441][2][2]; // 3^12 = 531441
    static int[] power3 = new int[MAXM];

    static void initPower3() {
        power3[0] = 1;
        for (int i = 1; i < MAXM; i++) {
            power3[i] = power3[i-1] * 3;
        }
    }

    static int get(int s, int pos) {
        return s / power3[pos] % 3;
    }

    static int set(int s, int pos, int val) {
        return s - get(s, pos) * power3[pos] + val * power3[pos];
    }

    public static void main(String[] args) {
        initPower3();
        Scanner sc = new Scanner(System.in);
        int T = sc.nextInt();
        while (T-- > 0) {
            int n = sc.nextInt();
            int m = sc.nextInt();
            int B = sc.nextInt();
            int W = sc.nextInt();
            int[][] grid = new int[n][m];
```

```java
for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
        grid[i][j] = sc.nextInt();
    }
}

// 重置 dp 数组
for (int i = 0; i <= n; i++) {
    for (int j = 0; j < m; j++) {
        for (int s = 0; s < power3[m]; s++) {
            for (int bc = 0; bc < 2; bc++) {
                for (int wc = 0; wc < 2; wc++) {
                    dp[i][j][s][bc][wc] = 0;
                }
            }
        }
    }
}

dp[0][0][0][0][0] = 1;

for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
        for (int s = 0; s < power3[m]; s++) {
            for (int bc = 0; bc < 2; bc++) {
                for (int wc = 0; wc < 2; wc++) {
                    if (dp[i][j][s][bc][wc] == 0) continue;

                    int nexti = i;
                    int nextj = j + 1;
                    if (nextj == m) {
                        nexti++;
                        nextj = 0;
                    }

                    int up = get(s, j);
                    int left = j > 0 ? get(s, j-1) : 0;

                    if (grid[i][j] == 1) {
                        // 必须是黑色
                        if (up != 2 && left != 2) {
                            int newbc = bc | (up == 0 && left == 0 ? 1 : 0);
                            int newwc = wc;
```

```
                                        int ns = set(s, j, 1);
                                        dp[nexti][nextj][ns][newbc][newwc] +=
dp[i][j][s][bc][wc];
                                    }
                                } else if (grid[i][j] == 2) {
                                    // 必须是白色
                                    if (up != 1 && left != 1) {
                                        int newbc = bc;
                                        int newwc = wc | (up == 0 && left == 0 ? 1 : 0);
                                        int ns = set(s, j, 2);
                                        dp[nexti][nextj][ns][newbc][newwc] +=
dp[i][j][s][bc][wc];
                                    }
                                } else {
                                    // 可以选择黑色
                                    if (up != 2 && left != 2) {
                                        int newbc = bc | (up == 0 && left == 0 ? 1 : 0);
                                        int newwc = wc;
                                        int ns = set(s, j, 1);
                                        dp[nexti][nextj][ns][newbc][newwc] +=
dp[i][j][s][bc][wc];
                                    }
                                    // 可以选择白色
                                    if (up != 1 && left != 1) {
                                        int newbc = bc;
                                        int newwc = wc | (up == 0 && left == 0 ? 1 : 0);
                                        int ns = set(s, j, 2);
                                        dp[nexti][nextj][ns][newbc][newwc] +=
dp[i][j][s][bc][wc];
                                    }
                                }
                            }
                        }
                    }
                }
            }

            long res = 0;
            for (int s = 0; s < power3[m]; s++) {
                res += dp[n][0][s][1][1];
            }
            System.out.println(res);
        }
```

```
        sc.close();
    }
}
```

时间复杂度：O(n * m * 3^m * 2 * 2)
空间复杂度：O(n * m * 3^m * 2 * 2)

```
        dp = [0] * (n + 1)
        dp[0] = 1
        dp[1] = 1
        dp[2] = 2
        for i in range(3, n + 1):
            dp[i] = (2 * dp[i-1] + dp[i-3]) % MOD
        return dp[n]
```

Java 实现代码：

```
class Solution {
    public int numTilings(int n) {
        final int MOD = 1_000_000_007;
        if (n == 1) return 1;
        long[] dp = new long[n + 1];
        dp[0] = 1;
        dp[1] = 1;
        dp[2] = 2;
        for (int i = 3; i <= n; i++) {
            dp[i] = (2 * dp[i-1] + dp[i-3]) % MOD;
        }
        return (int) dp[n];
    }
}
```

时间复杂度：O(n)
空间复杂度：O(n)，可以优化到 O(1)只使用几个变量
*/

// 补充题目 2：洛谷 P5056 【模板】插头 DP
/*
题目描述：
给出一个 n×m 的方格，有些格子不能铺线，其它格子必须铺，形成一个闭合回路。问有多少种铺法？

链接：https://www.luogu.com.cn/problem/P5056

C++ 实现代码：

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

typedef long long ll;
const int MAXN = 12;
const int MAXM = 12;

ll n, m;
ll grid[MAXN][MAXM];
ll endx, endy;
ll cnt;

// 使用哈希表优化空间
unordered_map<ll, ll> dp[2];
ll cur;

// 获取状态 s 中第 j 位的值
ll get(ll s, ll j) {
    return (s >> (j * 2)) & 3;
}

// 设置状态 s 中第 j 位的值为 v
ll set(ll s, ll j, ll v) {
    return s ^ ((ll)get(s, j) ^ v) << (j * 2);
}

// 找到与当前左括号匹配的右括号位置
ll find(ll s, ll j) {
    ll cnt = 1;
    for (ll i = j + 1; i <= m; i++) {
        if (get(s, i) == 1) cnt++;
        if (get(s, i) == 2) cnt--;
        if (cnt == 0) return i;
    }
    return -1;
}

// 最小表示法优化状态
ll min_rep(ll s) {
    ll a[MAXM + 1], b[MAXM + 1];
    memset(b, 0, sizeof(b));
```

```
    ll cnt = 0;
    for (ll i = 0; i <= m; i++) {
        a[i] = get(s, i);
        if (a[i] != 0 && b[a[i]] == 0) {
            b[a[i]] = ++cnt;
        }
        if (a[i] != 0) a[i] = b[a[i]];
    }
    ll res = 0;
    for (ll i = 0; i <= m; i++) {
        res = res * 3 + a[i];
    }
    return res;
}

void solve() {
    cnt = 0;
    for (ll i = 1; i <= n; i++) {
        for (ll j = 1; j <= m; j++) {
            cin >> grid[i][j];
            if (grid[i][j]) {
                cnt++;
                endx = i;
                endy = j;
            }
        }
    }

    if (cnt == 0) {
        cout << 0 << endl;
        return;
    }

    cur = 0;
    dp[cur].clear();
    dp[cur][0] = 1;

    for (ll i = 1; i <= n; i++) {
        // 状态转移到下一行，左移一位
        unordered_map<ll, ll> temp;
        for (auto& p : dp[cur]) {
            ll s = p.first;
            ll num = p.second;
```

```
        if (get(s, m) == 0) {
            temp[s << 2] += num;
        }
    }
    dp[cur] = temp;

    for (ll j = 1; j <= m; j++) {
        dp[cur ^ 1].clear();
        for (auto& p : dp[cur]) {
            ll s = p.first;
            ll num = p.second;

            ll left = get(s, j - 1);
            ll up = get(s, j);

            if (!grid[i][j]) {
                if (left == 0 && up == 0) {
                    dp[cur ^ 1][s] += num;
                }
                continue;
            }

            if (left == 0 && up == 0) {
                if (i < n && j < m) {
                    // 放置一个新的插头对
                    ll ns = set(s, j - 1, 1);
                    ns = set(ns, j, 2);
                    dp[cur ^ 1][ns] += num;
                }
            } else if (left == 0 && up != 0) {
                // 向右延伸
                if (j < m) {
                    ll ns = set(s, j - 1, up);
                    ns = set(ns, j, 0);
                    dp[cur ^ 1][ns] += num;
                }
                // 向下延伸
                if (i < n) {
                    dp[cur ^ 1][s] += num;
                }
            } else if (left != 0 && up == 0) {
                // 向右延伸
                if (j < m) {
```

```
                dp[cur ^ 1][s] += num;
            }
            // 向下延伸
            if (i < n) {
                ll ns = set(s, j - 1, 0);
                ns = set(ns, j, left);
                dp[cur ^ 1][ns] += num;
            }
        } else if (left == 1 && up == 1) {
            // 合并两个左括号，需要找到右括号并修改
            ll k = find(s, j);
            ll ns = set(s, j - 1, 0);
            ns = set(ns, j, 0);
            ns = set(ns, k, 1);
            dp[cur ^ 1][ns] += num;
        } else if (left == 2 && up == 2) {
            // 合并两个右括号，需要找到左括号并修改
            ll k = find(s, j - 1);
            ll ns = set(s, j - 1, 0);
            ns = set(ns, j, 0);
            ns = set(ns, k, 2);
            dp[cur ^ 1][ns] += num;
        } else if (left == 2 && up == 1) {
            // 合并一个右括号和一个左括号
            ll ns = set(s, j - 1, 0);
            ns = set(ns, j, 0);
            // 如果是最后一个格子，检查是否形成闭合回路
            if (i == endx && j == endy && ns == 0) {
                dp[cur ^ 1][ns] += num;
            } else if (ns != 0) {
                dp[cur ^ 1][ns] += num;
            }
        } else if (left == 1 && up == 2) {
            // 形成闭合回路
            ll ns = set(s, j - 1, 0);
            ns = set(ns, j, 0);
            if (i == endx && j == endy && ns == 0) {
                dp[cur ^ 1][ns] += num;
            }
        }
    }
    cur ^= 1;
}
```

```
    }

    cout << dp[cur][0] << endl;
}

int main() {
    solve();
    return 0;
}
```

Python 实现代码：

```python
import sys
from collections import defaultdict

MOD = 10**9 + 7

class Solution:
    def main(self):
        n, m = map(int, sys.stdin.readline().split())
        grid = []
        endx, endy = 0, 0
        cnt = 0
        for i in range(n):
            row = list(map(int, sys.stdin.readline().split()))
            grid.append(row)
            for j in range(m):
                if row[j]:
                    cnt += 1
                    endx, endy = i, j

        if cnt == 0:
            print(0)
            return

        dp = [defaultdict(int), defaultdict(int)]
        cur = 0
        dp[cur][0] = 1

        for i in range(n):
            # 状态转移到下一行，左移一位
            new_dp = defaultdict(int)
            for s, num in dp[cur].items():
                if (s >> (m * 2)) & 3 == 0:
```

```
            new_dp[s << 2] += num
    dp[cur] = new_dp

    for j in range(m):
        dp[cur ^ 1].clear()
        for s, num in dp[cur].items():
            left = (s >> ((j - 1) * 2)) & 3 if j > 0 else 0
            up = (s >> (j * 2)) & 3

            if not grid[i][j]:
                if left == 0 and up == 0:
                    dp[cur ^ 1][s] += num
                continue

            if left == 0 and up == 0:
                if i < n - 1 and j < m - 1:
                    ns = s
                    ns &= ~(3 << ((j - 1) * 2)) if j > 0 else ns
                    ns |= 1 << ((j - 1) * 2) if j > 0 else ns
                    ns &= ~(3 << (j * 2))
                    ns |= 2 << (j * 2)
                    dp[cur ^ 1][ns] += num
            elif left == 0 and up != 0:
                if j < m - 1:
                    ns = s
                    ns &= ~(3 << ((j - 1) * 2)) if j > 0 else ns
                    ns |= up << ((j - 1) * 2) if j > 0 else ns
                    ns &= ~(3 << (j * 2))
                    dp[cur ^ 1][ns] += num
                if i < n - 1:
                    dp[cur ^ 1][s] += num
            elif left != 0 and up == 0:
                if j < m - 1:
                    dp[cur ^ 1][s] += num
                if i < n - 1:
                    ns = s
                    ns &= ~(3 << ((j - 1) * 2)) if j > 0 else ns
                    ns &= ~(3 << (j * 2))
                    ns |= left << (j * 2)
                    dp[cur ^ 1][ns] += num
            elif left == 1 and up == 1:
                k = self.find(s, j, m)
                ns = s
```

```
                    ns &= ~(3 << ((j - 1) * 2)) if j > 0 else ns
                    ns &= ~(3 << (j * 2))
                    ns &= ~(3 << (k * 2))
                    ns |= 1 << (k * 2)
                    dp[cur ^ 1][ns] += num
                elif left == 2 and up == 2:
                    k = self.find_left(s, j - 1, m)
                    ns = s
                    ns &= ~(3 << ((j - 1) * 2)) if j > 0 else ns
                    ns &= ~(3 << (j * 2))
                    ns &= ~(3 << (k * 2))
                    ns |= 2 << (k * 2)
                    dp[cur ^ 1][ns] += num
                elif left == 2 and up == 1:
                    ns = s
                    ns &= ~(3 << ((j - 1) * 2)) if j > 0 else ns
                    ns &= ~(3 << (j * 2))
                    if (i == endx and j == endy) or ns != 0:
                        dp[cur ^ 1][ns] += num
                elif left == 1 and up == 2:
                    ns = s
                    ns &= ~(3 << ((j - 1) * 2)) if j > 0 else ns
                    ns &= ~(3 << (j * 2))
                    if i == endx and j == endy and ns == 0:
                        dp[cur ^ 1][ns] += num

            cur ^= 1

        print(dp[cur].get(0, 0))

    def find(self, s, j, m):
        cnt = 1
        for i in range(j + 1, m + 1):
            val = (s >> (i * 2)) & 3
            if val == 1:
                cnt += 1
            elif val == 2:
                cnt -= 1
                if cnt == 0:
                    return i
        return -1

    def find_left(self, s, j, m):
```

```python
        cnt = 1
        for i in range(j - 1, -1, -1):
            val = (s >> (i * 2)) & 3
            if val == 2:
                cnt += 1
            elif val == 1:
                cnt -= 1
                if cnt == 0:
                    return i
        return -1

if __name__ == "__main__":
    solution = Solution()
    solution.main()
```

Java 实现代码：

```java
import java.util.*;

public class Main {
    static final int MOD = 1000000007;

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int m = sc.nextInt();
        int[][] grid = new int[n][m];
        int endx = 0, endy = 0;
        int cnt = 0;
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                grid[i][j] = sc.nextInt();
                if (grid[i][j] == 1) {
                    cnt++;
                    endx = i;
                    endy = j;
                }
            }
        }
        sc.close();

        if (cnt == 0) {
            System.out.println(0);
            return;
```

```
}

Map<Long, Long>[] dp = new Map[2];
dp[0] = new HashMap<>();
dp[1] = new HashMap<>();
int cur = 0;
dp[cur].put(0L, 1L);

for (int i = 0; i < n; i++) {
    // 状态转移到下一行，左移一位
    Map<Long, Long> newDp = new HashMap<>();
    for (Map.Entry<Long, Long> entry : dp[cur].entrySet()) {
        long s = entry.getKey();
        long num = entry.getValue();
        if (get(s, m) == 0) {
            long newS = s << 2;
            newDp.put(newS, (newDp.getOrDefault(newS, 0L) + num) % MOD);
        }
    }
    dp[cur] = newDp;

    for (int j = 0; j < m; j++) {
        dp[cur ^ 1].clear();
        for (Map.Entry<Long, Long> entry : dp[cur].entrySet()) {
            long s = entry.getKey();
            long num = entry.getValue();

            int left = j > 0 ? get(s, j - 1) : 0;
            int up = get(s, j);

            if (grid[i][j] == 0) {
                if (left == 0 && up == 0) {
                    dp[cur ^ 1].put(s, (dp[cur ^ 1].getOrDefault(s, 0L) + num) % MOD);
                }
                continue;
            }

            if (left == 0 && up == 0) {
                if (i < n - 1 && j < m - 1) {
                    long ns = set(s, j - 1, 1, m);
                    ns = set(ns, j, 2, m);
                    dp[cur ^ 1].put(ns, (dp[cur ^ 1].getOrDefault(ns, 0L) + num) % MOD);
                }
```

```
} else if (left == 0 && up != 0) {
    if (j < m - 1) {
        long ns = set(s, j - 1, up, m);
        ns = set(ns, j, 0, m);
        dp[cur ^ 1].put(ns, (dp[cur ^ 1].getOrDefault(ns, 0L) + num) % MOD);
    }
    if (i < n - 1) {
        dp[cur ^ 1].put(s, (dp[cur ^ 1].getOrDefault(s, 0L) + num) % MOD);
    }
} else if (left != 0 && up == 0) {
    if (j < m - 1) {
        dp[cur ^ 1].put(s, (dp[cur ^ 1].getOrDefault(s, 0L) + num) % MOD);
    }
    if (i < n - 1) {
        long ns = set(s, j - 1, 0, m);
        ns = set(ns, j, left, m);
        dp[cur ^ 1].put(ns, (dp[cur ^ 1].getOrDefault(ns, 0L) + num) % MOD);
    }
} else if (left == 1 && up == 1) {
    int k = find(s, j, m);
    long ns = set(s, j - 1, 0, m);
    ns = set(ns, j, 0, m);
    ns = set(ns, k, 1, m);
    dp[cur ^ 1].put(ns, (dp[cur ^ 1].getOrDefault(ns, 0L) + num) % MOD);
} else if (left == 2 && up == 2) {
    int k = findLeft(s, j - 1, m);
    long ns = set(s, j - 1, 0, m);
    ns = set(ns, j, 0, m);
    ns = set(ns, k, 2, m);
    dp[cur ^ 1].put(ns, (dp[cur ^ 1].getOrDefault(ns, 0L) + num) % MOD);
} else if (left == 2 && up == 1) {
    long ns = set(s, j - 1, 0, m);
    ns = set(ns, j, 0, m);
    if ((i == endx && j == endy) || ns != 0) {
        dp[cur ^ 1].put(ns, (dp[cur ^ 1].getOrDefault(ns, 0L) + num) % MOD);
    }
} else if (left == 1 && up == 2) {
    long ns = set(s, j - 1, 0, m);
    ns = set(ns, j, 0, m);
    if (i == endx && j == endy && ns == 0) {
        dp[cur ^ 1].put(ns, (dp[cur ^ 1].getOrDefault(ns, 0L) + num) % MOD);
    }
}
```

```java
                }
                cur ^= 1;
            }
        }

        System.out.println(dp[cur].getOrDefault(0L, 0L) % MOD);
    }

    static int get(long s, int j) {
        return (int) ((s >> (j * 2)) & 3);
    }

    static long set(long s, int j, int v, int m) {
        if (j < 0 || j > m) return s;
        return s ^ ((long)(get(s, j) ^ v) << (j * 2));
    }

    static int find(long s, int j, int m) {
        int cnt = 1;
        for (int i = j + 1; i <= m; i++) {
            int val = get(s, i);
            if (val == 1) cnt++;
            if (val == 2) cnt--;
            if (cnt == 0) return i;
        }
        return -1;
    }

    static int findLeft(long s, int j, int m) {
        int cnt = 1;
        for (int i = j - 1; i >= 0; i--) {
            int val = get(s, i);
            if (val == 2) cnt++;
            if (val == 1) cnt--;
            if (cnt == 0) return i;
        }
        return -1;
    }
}
```

时间复杂度：O(n * m * 状态数)，状态数取决于编码方式，这里使用括号表示法，状态数为 O(Catalan(m))
空间复杂度：O(状态数)，通过滚动数组和哈希表优化
*/

==

文件: Code02_GridPainting.java
==
```java
package class126;

// 轮廓线 DP 专题 - 题目 1: 网格涂色问题
// 用三种不同颜色为网格涂色
// 给你两个整数 m 和 n，表示 m*n 的网格，其中每个单元格最开始是白色
// 请你用红、绿、蓝三种颜色为每个单元格涂色，所有单元格都需要被涂色
// 要求相邻单元格的颜色一定要不同
// 返回网格涂色的方法数，答案对 1000000007 取模
// 1 <= m <= 5
// 1 <= n <= 1000
// 测试链接 : https://leetcode.cn/problems/painting-a-grid-with-three-different-colors/
// 这是一个典型的轮廓线 DP 问题，使用三进制表示每一行的颜色状态
//
// 题目大意:
// 给定一个 m×n 的网格，要求用三种颜色为每个单元格涂色，使得相邻单元格颜色不同。
// 求满足条件的涂色方案数。
//
// 解题思路:
// 使用轮廓线 DP，逐格处理，记录每个位置的颜色状态。
// 状态表示：用三进制表示轮廓线状态，0 表示红色，1 表示绿色，2 表示蓝色。
// 状态转移：考虑当前格子的颜色，确保与上方和左侧格子颜色不同。
//
// Java 实现：https://github.com/yourusername/algorithm-
journey/blob/main/src/class126/Code02_GridPainting.java
// C++实现：https://github.com/yourusername/algorithm-
journey/blob/main/src/class126/Code02_GridPainting.cpp
// Python 实现：https://github.com/yourusername/algorithm-
journey/blob/main/src/class126/Code02_GridPainting.py

public class Code02_GridPainting {

    // 最大行数或列数
    public static int MAXN = 1001;

    // 最大列数或行数（较小的那个维度）
    public static int MAXM = 5;

    // 最大状态数，使用三进制表示每行的颜色
```

```java
public static int MAXS = (int) Math.pow(3, MAXM);

// 取模值
public static int MOD = 1000000007;

// 存储行数或列数（较大的那个维度）
public static int n;

// 存储列数或行数（较小的那个维度）
public static int m;

// 实际最大状态数
public static int maxs;

// 动态规划数组：dp[i][j][s] 表示处理到第 i 行第 j 列时，状态为 s 的方案数
// 其中状态 s 使用三进制表示当前行已处理部分的颜色
public static int[][][] dp = new int[MAXN][MAXM][MAXS];

// 存储第一行的所有有效状态
public static int[] first = new int[MAXS];

// 第一行有效状态的数量
public static int size;

/**
 * 计算网格涂色的可能方案数
 *
 * @param rows 网格的行数
 * @param cols 网格的列数
 * @return 所有可能的涂色方案数
 */
public static int colorTheGrid(int rows, int cols) {
    // 初始化数据结构
    build(rows, cols);
    int ans = 0;
    // 遍历所有可能的第一行状态，累加结果
    for (int i = 0; i < size; i++) {
        ans = (ans + f(1, 0, first[i], 1)) % MOD;
    }
    return ans;
}

/**
```

```java
 * 初始化数据结构，准备动态规划
 *
 * @param rows 网格的行数
 * @param cols 网格的列数
 */
public static void build(int rows, int cols) {
    // 为了优化计算，将较大的维度作为行，较小的维度作为列
    n = Math.max(rows, cols);
    m = Math.min(rows, cols);
    // 计算最大状态数
    maxs = (int) Math.pow(3, m);
    // 初始化 DP 数组为-1（表示未计算）
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            for (int s = 0; s < maxs; s++) {
                dp[i][j][s] = -1;
            }
        }
    }
    // 重置状态数量
    size = 0;
    // 生成所有有效的第一行状态
    dfs(0, 0, 1);
}

/**
 * 使用深度优先搜索生成所有有效的第一行状态
 *
 * @param j 当前处理的列索引
 * @param s 当前状态（三进制表示）
 * @param bit 当前位的权重（3 的幂）
 */
// 取得所有第一行的有效状态
public static void dfs(int j, int s, int bit) {
    if (j == m) {
        // 处理完一行，保存该状态
        first[size++] = s;
    } else {
        // 获取左侧格子的颜色（如果有的话）
        int left = j == 0 ? -1 : get(s, bit / 3);
        // 尝试三种颜色，确保不与左侧颜色相同
        if (left != 0) {
            dfs(j + 1, set(s, bit, 0), bit * 3);
```

```java
        }
        if (left != 1) {
            dfs(j + 1, set(s, bit, 1), bit * 3);
        }
        if (left != 2) {
            dfs(j + 1, set(s, bit, 2), bit * 3);
        }
    }
}

/**
 * 动态规划计算方案数
 *
 * @param i 当前处理的行索引
 * @param j 当前处理的列索引
 * @param s 当前状态（三进制表示）
 * @param bit 当前位的权重（3 的幂）
 * @return 从当前状态到结束的方案数
 */
public static int f(int i, int j, int s, int bit) {
    // 已经处理完所有行，返回 1 种方案
    if (i == n) {
        return 1;
    }
    // 处理完当前行，转到下一行的第一个列
    if (j == m) {
        return f(i + 1, 0, s, 1);
    }
    // 记忆化搜索，如果已经计算过，直接返回结果
    if (dp[i][j][s] != -1) {
        return dp[i][j][s];
    }
    // 上方的颜色（来自上一行同一位置）
    int up = get(s, bit);
    // 左侧的颜色（来自当前行已处理的部分），-1 代表左侧没有格子
    int left = j == 0 ? -1 : get(s, bit / 3);
    int ans = 0;
    // 尝试三种颜色，确保不与上方和左侧颜色相同
    if (up != 0 && left != 0) {
        ans = (ans + f(i, j + 1, set(s, bit, 0), bit * 3)) % MOD;
    }
    if (up != 1 && left != 1) {
        ans = (ans + f(i, j + 1, set(s, bit, 1), bit * 3)) % MOD;
```

```java
    }
    if (up != 2 && left != 2) {
        ans = (ans + f(i, j + 1, set(s, bit, 2), bit * 3)) % MOD;
    }
    // 保存结果并返回
    dp[i][j][s] = ans;
    return ans;
}

/**
 * 从状态中获取指定位的颜色
 *
 * @param s 状态值
 * @param bit 位权重
 * @return 颜色值（0、1、2）
 */
public static int get(int s, int bit) {
    return s / bit % 3;
}

/**
 * 在状态中设置指定位的颜色
 *
 * @param s 当前状态
 * @param bit 位权重
 * @param v 要设置的颜色值
 * @return 更新后的状态
 */
public static int set(int s, int bit, int v) {
    return s - get(s, bit) * bit + v * bit;
}

// 测试用例
public static void main(String[] args) {
    // 测试用例 1：3x2 网格
    System.out.println(colorTheGrid(3, 2));  // 预期输出：36

    // 测试用例 2：2x3 网格
    System.out.println(colorTheGrid(2, 3));  // 预期输出：54

    // 测试用例 3：5x1 网格
    System.out.println(colorTheGrid(5, 1));  // 预期输出：3
}
```

```
}

// 补充题目 1：LeetCode 790 Domino and Tromino Tiling
/*
题目描述：
有两种形状的骨牌：
1. 1x2 的多米诺骨牌，可以水平或垂直放置
2. 2x3 的托米诺骨牌，其形状为 L 形，可以有 4 种不同的旋转方式

给你一个整数 n ，表示一个 2xn 的网格，可以使用任意数量的多米诺骨牌和托米诺骨牌。
返回铺满整个网格的不同铺法总数。答案需要对 10^9 + 7 取模。

链接：https://leetcode.cn/problems/domino-and-tromino-tiling/

算法解析：
这是一个经典的动态规划问题，我们可以通过定义状态来表示当前网格的覆盖情况。
我们定义四个状态：
- dp[n][0]: 铺满前 n 列
- dp[n][1]: 铺满前 n 列，且第 n+1 列的第一行已铺
- dp[n][2]: 铺满前 n 列，且第 n+1 列的第二行已铺
- dp[n][3]: 铺满前 n-1 列，且第 n 列已铺

状态转移方程：
dp[n][0] = dp[n-1][0] + dp[n-2][0] + dp[n-1][1] + dp[n-1][2]
dp[n][1] = dp[n-2][0] + dp[n-1][2]
dp[n][2] = dp[n-2][0] + dp[n-1][1]
dp[n][3] = dp[n-1][0]

C++ 实现代码：
class Solution {
public:
    int numTilings(int n) {
        const int MOD = 1e9 + 7;
        vector<vector<long long>> dp(n + 1, vector<long long>(4, 0));
        dp[0][0] = 1;

        for (int i = 1; i <= n; ++i) {
            if (i >= 1) {
                dp[i][0] = (dp[i][0] + dp[i-1][0]) % MOD;
                dp[i][3] = (dp[i][3] + dp[i-1][0]) % MOD;
            }
            if (i >= 2) {
                dp[i][0] = (dp[i][0] + dp[i-2][0]) % MOD;
```

```
                dp[i][1] = (dp[i][1] + dp[i-2][0]) % MOD;
                dp[i][2] = (dp[i][2] + dp[i-2][0]) % MOD;
            }
            if (i >= 1) {
                dp[i][0] = (dp[i][0] + dp[i-1][1]) % MOD;
                dp[i][0] = (dp[i][0] + dp[i-1][2]) % MOD;
            }
            if (i >= 1) {
                dp[i][1] = (dp[i][1] + dp[i-1][2]) % MOD;
                dp[i][2] = (dp[i][2] + dp[i-1][1]) % MOD;
            }
        }

        return dp[n][0];
    }
};
```

Python 实现代码：

```
class Solution:
    def numTilings(self, n: int) -> int:
        MOD = 10**9 + 7
        dp = [[0] * 4 for _ in range(n + 1)]
        dp[0][0] = 1

        for i in range(1, n + 1):
            if i >= 1:
                dp[i][0] = (dp[i][0] + dp[i-1][0]) % MOD
                dp[i][3] = (dp[i][3] + dp[i-1][0]) % MOD
            if i >= 2:
                dp[i][0] = (dp[i][0] + dp[i-2][0]) % MOD
                dp[i][1] = (dp[i][1] + dp[i-2][0]) % MOD
                dp[i][2] = (dp[i][2] + dp[i-2][0]) % MOD
            if i >= 1:
                dp[i][0] = (dp[i][0] + dp[i-1][1]) % MOD
                dp[i][0] = (dp[i][0] + dp[i-1][2]) % MOD
            if i >= 1:
                dp[i][1] = (dp[i][1] + dp[i-1][2]) % MOD
                dp[i][2] = (dp[i][2] + dp[i-1][1]) % MOD

        return dp[n][0]
```

Java 实现代码：

```
class Solution {
```

```
    public int numTilings(int n) {
        final int MOD = 1000000007;
        long[][] dp = new long[n + 1][4];
        dp[0][0] = 1;

        for (int i = 1; i <= n; i++) {
            if (i >= 1) {
                dp[i][0] = (dp[i][0] + dp[i-1][0]) % MOD;
                dp[i][3] = (dp[i][3] + dp[i-1][0]) % MOD;
            }
            if (i >= 2) {
                dp[i][0] = (dp[i][0] + dp[i-2][0]) % MOD;
                dp[i][1] = (dp[i][1] + dp[i-2][0]) % MOD;
                dp[i][2] = (dp[i][2] + dp[i-2][0]) % MOD;
            }
            if (i >= 1) {
                dp[i][0] = (dp[i][0] + dp[i-1][1]) % MOD;
                dp[i][0] = (dp[i][0] + dp[i-1][2]) % MOD;
            }
            if (i >= 1) {
                dp[i][1] = (dp[i][1] + dp[i-1][2]) % MOD;
                dp[i][2] = (dp[i][2] + dp[i-1][1]) % MOD;
            }
        }

        return (int) dp[n][0];
    }
}
```

时间复杂度：O(n)
空间复杂度：O(n)

// 补充题目 2：POJ 2411 Mondriaan's Dream
/*
题目描述：
求用 1×2 的多米诺骨牌铺满 n×m 的网格的方案数。

链接：http://poj.org/problem?id=2411

算法解析：
这是一个经典的状态压缩动态规划问题。我们使用二进制状态表示每一行的骨牌放置情况。
对于每一行，我们需要确保当前行的状态与上一行的状态兼容，即不能有重叠的骨牌。

C++ 实现代码：

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

typedef long long ll;
ll dp[12][1<<11];
int n, m;

// 检查状态是否合法（没有奇数个连续的 0）
bool is_valid(int s) {
    int cnt = 0;
    for (int i = 0; i < m; ++i) {
        if ((s >> i) & 1) {
            cnt = 0;
        } else {
            cnt++;
            if (cnt % 2 != 0) {
                return false;
            }
        }
    }
    return cnt % 2 == 0;
}

// 检查两个状态是否兼容
bool is_compatible(int a, int b) {
    return (a & b) == 0 && is_valid(a | b);
}

int main() {
    while (cin >> n >> m && n && m) {
        memset(dp, 0, sizeof(dp));

        // 预处理所有合法的行状态
        for (int s = 0; s < (1 << m); ++s) {
            if (is_valid(s)) {
                dp[0][s] = 1;
            }
        }

        // 动态规划填表
        for (int i = 1; i < n; ++i) {
```

```cpp
            for (int s = 0; s < (1 << m); ++s) {
                for (int prev = 0; prev < (1 << m); ++prev) {
                    if (is_compatible(prev, s)) {
                        dp[i][s] += dp[i-1][prev];
                    }
                }
            }
        }

        cout << dp[n-1][0] << endl;
    }
    return 0;
}
```

Python 实现代码：

```python
import sys

def main():
    while True:
        line = sys.stdin.readline()
        if not line:
            break
        n, m = map(int, line.strip().split())
        if n == 0 and m == 0:
            break

        # 交换行列，使得 m 较小，优化性能
        if n < m:
            n, m = m, n

        dp = [[0] * (1 << m) for _ in range(n)]

        # 检查状态是否合法
        def is_valid(s):
            cnt = 0
            for i in range(m):
                if (s >> i) & 1:
                    cnt = 0
                else:
                    cnt += 1
                    if cnt % 2 != 0:
                        return False
            return cnt % 2 == 0
```

```python
        # 预处理第一行
        for s in range(1 << m):
            if is_valid(s):
                dp[0][s] = 1

        # 检查两个状态是否兼容
        def is_compatible(a, b):
            return (a & b) == 0 and is_valid(a | b)

        # 动态规划
        for i in range(1, n):
            for s in range(1 << m):
                for prev in range(1 << m):
                    if is_compatible(prev, s):
                        dp[i][s] += dp[i-1][prev]

        print(dp[n-1][0])

if __name__ == "__main__":
    main()
```

Java 实现代码:

```java
import java.util.*;

public class Main {
    static long[][] dp;
    static int n, m;

    static boolean is_valid(int s) {
        int cnt = 0;
        for (int i = 0; i < m; i++) {
            if ((s >> i & 1) == 1) {
                cnt = 0;
            } else {
                cnt++;
                if (cnt % 2 != 0) {
                    return false;
                }
            }
        }
        return cnt % 2 == 0;
    }
```

```java
static boolean is_compatible(int a, int b) {
    return (a & b) == 0 && is_valid(a | b);
}

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    while (true) {
        n = sc.nextInt();
        m = sc.nextInt();
        if (n == 0 && m == 0) break;

        // 交换行列，优化性能
        if (n < m) {
            int temp = n;
            n = m;
            m = temp;
        }

        dp = new long[n][1 << m];

        // 预处理第一行
        for (int s = 0; s < (1 << m); s++) {
            if (is_valid(s)) {
                dp[0][s] = 1;
            }
        }

        // 动态规划
        for (int i = 1; i < n; i++) {
            for (int s = 0; s < (1 << m); s++) {
                for (int prev = 0; prev < (1 << m); prev++) {
                    if (is_compatible(prev, s)) {
                        dp[i][s] += dp[i-1][prev];
                    }
                }
            }
        }

        System.out.println(dp[n-1][0]);
    }
    sc.close();
}
```

```
}
```

时间复杂度：O(n * 4^m)
空间复杂度：O(n * 2^m)

// 补充题目 3：HDU 1693 Eat the Trees
/*
题目描述：
给一个 n*m 的网格，某些格子是障碍，求用 1×2 的骨牌覆盖整个非障碍格子的方案数。

链接：http://acm.hdu.edu.cn/showproblem.php?pid=1693

算法解析：
这是一个典型的插头 DP 问题。我们使用二进制状态表示轮廓线上的插头情况，每个位置有两种可能的状态：0 表示没有插头，1 表示有插头。通过状态转移，我们可以计算出所有可能的覆盖方式。

C++ 实现代码：
```cpp
#include <iostream>
#include <cstring>
#include <vector>
using namespace std;

typedef long long ll;
const int MAXN = 12;
const int MAXM = 12;

ll dp[2][1<<MAXM];
int grid[MAXN][MAXM];
int n, m;

void solve() {
    memset(dp, 0, sizeof(dp));
    int cur = 0, next = 1;
    dp[cur][0] = 1;

    for (int i = 0; i < n; ++i) {
        // 换行时，将所有状态左移一位
        for (int j = 0; j < (1 << m); ++j) {
            dp[next][j << 1] = dp[cur][j];
        }
        swap(cur, next);
        memset(dp[next], 0, sizeof(dp[next]));
```

```
for (int j = 0; j < m; ++j) {
    for (int s = 0; s < (1 << (m + 1)); ++s) {
        if (dp[cur][s] == 0) continue;

        // 当前格子是障碍
        if (grid[i][j] == 1) {
            if ((s & 1) == 0 && ((s >> m) & 1) == 0) {
                dp[next][(s >> 1)] += dp[cur][s];
            }
            continue;
        }

        int left = s & 1;
        int up = (s >> m) & 1;

        if (left == 0 && up == 0) {
            // 新的插头对
            if (j < m - 1 && grid[i][j+1] == 0) {
                dp[next][(s >> 1) | (1 << m)] += dp[cur][s];
            }
            if (i < n - 1 && grid[i+1][j] == 0) {
                dp[next][(s >> 1) | 1] += dp[cur][s];
            }
        } else if (left == 1 && up == 0) {
            // 右插头延续或向下转
            if (j < m - 1 && grid[i][j+1] == 0) {
                dp[next][(s >> 1) | (1 << m)] += dp[cur][s];
            }
            if (i < n - 1 && grid[i+1][j] == 0) {
                dp[next][(s >> 1)] += dp[cur][s];
            }
        } else if (left == 0 && up == 1) {
            // 下插头延续或向右转
            if (i < n - 1 && grid[i+1][j] == 0) {
                dp[next][(s >> 1) | 1] += dp[cur][s];
            }
            if (j < m - 1 && grid[i][j+1] == 0) {
                dp[next][(s >> 1)] += dp[cur][s];
            }
        } else if (left == 1 && up == 1) {
            // 合并两个插头
            dp[next][(s >> 1)] += dp[cur][s];
        }
```

```
            }
            swap(cur, next);
            memset(dp[next], 0, sizeof(dp[next]));
        }
    }

    cout << dp[cur][0] << endl;
}

int main() {
    int T, cas = 1;
    cin >> T;
    while (T--) {
        cin >> n >> m;
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < m; ++j) {
                cin >> grid[i][j];
            }
        }
        cout << "Case " << cas++ << ": There are " << solve() << " ways to eat the trees." <<
endl;
    }
    return 0;
}
```

Python 实现代码：

```
def main():
    import sys
    T = int(sys.stdin.readline())
    for cas in range(1, T + 1):
        n, m = map(int, sys.stdin.readline().split())
        grid = []
        for _ in range(n):
            grid.append(list(map(int, sys.stdin.readline().split())))

        # 初始化 DP 数组
        dp = [ [0] * (1 << (m + 1)) for _ in range(2) ]
        cur, nxt = 0, 1
        dp[cur][0] = 1

        for i in range(n):
            # 换行时左移一位
            for j in range(1 << (m + 1)):
```

```
    dp[nxt][j << 1] = dp[cur][j]
cur, nxt = nxt, cur
for j in range(1 << (m + 1)):
    dp[nxt][j] = 0

for j in range(m):
    for s in range(1 << (m + 1)):
        if dp[cur][s] == 0:
            continue

        # 障碍格子
        if grid[i][j] == 1:
            if (s & 1) == 0 and ((s >> m) & 1) == 0:
                dp[nxt][(s >> 1)] += dp[cur][s]
            continue

        left = s & 1
        up = (s >> m) & 1

        if left == 0 and up == 0:
            # 新的插头对
            if j < m - 1 and grid[i][j+1] == 0:
                dp[nxt][(s >> 1) | (1 << m)] += dp[cur][s]
            if i < n - 1 and grid[i+1][j] == 0:
                dp[nxt][(s >> 1) | 1] += dp[cur][s]
        elif left == 1 and up == 0:
            # 右插头
            if j < m - 1 and grid[i][j+1] == 0:
                dp[nxt][(s >> 1) | (1 << m)] += dp[cur][s]
            if i < n - 1 and grid[i+1][j] == 0:
                dp[nxt][(s >> 1)] += dp[cur][s]
        elif left == 0 and up == 1:
            # 下插头
            if i < n - 1 and grid[i+1][j] == 0:
                dp[nxt][(s >> 1) | 1] += dp[cur][s]
            if j < m - 1 and grid[i][j+1] == 0:
                dp[nxt][(s >> 1)] += dp[cur][s]
        elif left == 1 and up == 1:
            # 合并插头
            dp[nxt][(s >> 1)] += dp[cur][s]

    cur, nxt = nxt, cur
    for j in range(1 << (m + 1)):
```

```python
                dp[nxt][j] = 0

        print(f"Case {cas}: There are {dp[cur][0]} ways to eat the trees.")

if __name__ == "__main__":
    main()
```

Java 实现代码：

```java
import java.util.*;

public class Main {
    static long[][] dp;
    static int[][] grid;
    static int n, m;

    static long solve() {
        dp = new long[2][1 << (m + 1)];
        int cur = 0, next = 1;
        dp[cur][0] = 1;

        for (int i = 0; i < n; i++) {
            // 换行时左移一位
            for (int j = 0; j < (1 << (m + 1)); j++) {
                dp[next][j << 1] = dp[cur][j];
            }
            cur = next;
            next = 1 - cur;
            Arrays.fill(dp[next], 0);

            for (int j = 0; j < m; j++) {
                for (int s = 0; s < (1 << (m + 1)); s++) {
                    if (dp[cur][s] == 0) continue;

                    // 障碍格子
                    if (grid[i][j] == 1) {
                        if ((s & 1) == 0 && ((s >> m) & 1) == 0) {
                            dp[next][(s >> 1)] += dp[cur][s];
                        }
                        continue;
                    }

                    int left = s & 1;
                    int up = (s >> m) & 1;
```

```
                if (left == 0 && up == 0) {
                    // 新的插头对
                    if (j < m - 1 && grid[i][j+1] == 0) {
                        dp[next][(s >> 1) | (1 << m)] += dp[cur][s];
                    }
                    if (i < n - 1 && grid[i+1][j] == 0) {
                        dp[next][(s >> 1) | 1] += dp[cur][s];
                    }
                } else if (left == 1 && up == 0) {
                    // 右插头
                    if (j < m - 1 && grid[i][j+1] == 0) {
                        dp[next][(s >> 1) | (1 << m)] += dp[cur][s];
                    }
                    if (i < n - 1 && grid[i+1][j] == 0) {
                        dp[next][(s >> 1)] += dp[cur][s];
                    }
                } else if (left == 0 && up == 1) {
                    // 下插头
                    if (i < n - 1 && grid[i+1][j] == 0) {
                        dp[next][(s >> 1) | 1] += dp[cur][s];
                    }
                    if (j < m - 1 && grid[i][j+1] == 0) {
                        dp[next][(s >> 1)] += dp[cur][s];
                    }
                } else if (left == 1 && up == 1) {
                    // 合并插头
                    dp[next][(s >> 1)] += dp[cur][s];
                }
            }

            cur = next;
            next = 1 - cur;
            Arrays.fill(dp[next], 0);
        }
    }

    return dp[cur][0];
}

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int T = sc.nextInt();
```

```
        for (int cas = 1; cas <= T; cas++) {
            n = sc.nextInt();
            m = sc.nextInt();
            grid = new int[n][m];
            for (int i = 0; i < n; i++) {
                for (int j = 0; j < m; j++) {
                    grid[i][j] = sc.nextInt();
                }
            }
            System.out.println("Case " + cas + ": There are " + solve() + " ways to eat the
trees.");
        }
        sc.close();
    }
}
```

时间复杂度：O(n * m * 2^m)
空间复杂度：O(2^m)

// 补充题目 4：UVA 10572 Black and White
/*
题目描述：
给一个 n*m 的网格，每个格子可以是黑色、白色或障碍。
要求相邻的非障碍格子必须有不同的颜色，且黑白色必须相等。
求满足条件的方案数。

链接：https://vjudge.net/problem/UVA-10572

算法解析：
这是一个使用三进制插头 DP 的问题。每个位置有三种可能的状态：
0 表示没有插头，1 表示黑色，2 表示白色。
我们需要跟踪当前的颜色使用情况，并确保黑白相等。

C++ 实现代码：
```
#include <iostream>
#include <cstring>
#include <map>
using namespace std;

typedef long long ll;
const int MAXN = 12;
const int MAXM = 12;
```

```cpp
int n, m;
int grid[MAXN][MAXM];
map<ll, ll> dp[2];
ll power3[MAXM + 1];

void initPower3() {
    power3[0] = 1;
    for (int i = 1; i <= MAXM; ++i) {
        power3[i] = power3[i-1] * 3;
    }
}

int get(ll s, int pos) {
    return s / power3[pos] % 3;
}

ll set(ll s, int pos, int val) {
    return s - get(s, pos) * power3[pos] + (ll)val * power3[pos];
}

ll solve() {
    int cur = 0, next = 1;
    dp[cur].clear();
    dp[cur][0] = 1;

    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < m; ++j) {
            dp[next].clear();

            for (auto &p : dp[cur]) {
                ll s = p.first;
                ll cnt = p.second;

                // 获取当前位置的上方和左方插头
                int up = get(s, j);
                int left = j > 0 ? get(s, j-1) : 0;

                // 障碍格子
                if (grid[i][j] == 1) {
                    if (up == 0 && left == 0) {
                        dp[next][s] += cnt;
                    }
                    continue;
```

```
            }

            // 尝试两种颜色
            for (int color = 1; color <= 2; ++color) {
                // 检查与上方和左方的颜色冲突
                if (up == color || left == color) {
                    continue;
                }

                // 检查是否满足题目约束
                if (grid[i][j] != 0 && grid[i][j] != color) {
                    continue;
                }

                ll ns = set(s, j, color);
                if (j > 0) {
                    ns = set(ns, j-1, 0);
                }
                dp[next][ns] += cnt;
            }
        }

        swap(cur, next);
    }

    // 换行处理
    dp[next].clear();
    for (auto &p : dp[cur]) {
        ll s = p.first;
        ll cnt = p.second;
        bool valid = true;
        for (int j = 0; j < m; ++j) {
            if (get(s, j) != 0) {
                valid = false;
                break;
            }
        }
        if (valid) {
            dp[next][s] += cnt;
        }
    }
    swap(cur, next);
}
```

```
    ll ans = 0;
    for (auto &p : dp[cur]) {
        ans += p.second;
    }
    return ans;
}

int main() {
    initPower3();
    int T;
    cin >> T;
    while (T--) {
        cin >> n >> m;
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < m; ++j) {
                cin >> grid[i][j];
            }
        }
        cout << solve() << endl;
    }
    return 0;
}
```

Python 实现代码：

```
def main():
    import sys
    from collections import defaultdict

    # 预计算 3 的幂次
    power3 = [1] * 13
    for i in range(1, 13):
        power3[i] = power3[i-1] * 3

    def get(s, pos):
        return s // power3[pos] % 3

    def set_val(s, pos, val):
        return s - get(s, pos) * power3[pos] + val * power3[pos]

    T = int(sys.stdin.readline())
    for _ in range(T):
        n, m = map(int, sys.stdin.readline().split())
```

```
grid = []
for _ in range(n):
    grid.append(list(map(int, sys.stdin.readline().split())))

# 初始化 DP
dp = [defaultdict(int), defaultdict(int)]
cur, nxt = 0, 1
dp[cur][0] = 1

for i in range(n):
    for j in range(m):
        dp[nxt].clear()

        for s, cnt in dp[cur].items():
            up = get(s, j)
            left = get(s, j-1) if j > 0 else 0

            # 障碍格子
            if grid[i][j] == 1:
                if up == 0 and left == 0:
                    dp[nxt][s] += cnt
                continue

            # 尝试两种颜色
            for color in [1, 2]:
                if up == color or left == color:
                    continue
                if grid[i][j] != 0 and grid[i][j] != color:
                    continue

                ns = set_val(s, j, color)
                if j > 0:
                    ns = set_val(ns, j-1, 0)
                dp[nxt][ns] += cnt

        cur, nxt = nxt, cur

    # 换行处理
    dp[nxt].clear()
    for s, cnt in dp[cur].items():
        valid = True
        for j in range(m):
            if get(s, j) != 0:
```

```python
                    valid = False
                    break
            if valid:
                dp[nxt][s] += cnt

        cur, nxt = nxt, cur

    print(sum(dp[cur].values()))

if __name__ == "__main__":
    main()
```

Java 实现代码：

```java
import java.util.*;

public class Main {
    static long[] power3;
    static int n, m;
    static int[][] grid;

    static void initPower3() {
        power3 = new long[13];
        power3[0] = 1;
        for (int i = 1; i <= 12; i++) {
            power3[i] = power3[i-1] * 3;
        }
    }

    static int get(long s, int pos) {
        return (int) (s / power3[pos] % 3);
    }

    static long set(long s, int pos, int val) {
        return s - get(s, pos) * power3[pos] + (long)val * power3[pos];
    }

    static long solve() {
        Map<Long, Long>[] dp = new HashMap[2];
        dp[0] = new HashMap<>();
        dp[1] = new HashMap<>();
        int cur = 0, next = 1;
        dp[cur].put(0L, 1L);
```

```java
for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
        dp[next].clear();

        for (Map.Entry<Long, Long> entry : dp[cur].entrySet()) {
            long s = entry.getKey();
            long cnt = entry.getValue();

            int up = get(s, j);
            int left = j > 0 ? get(s, j-1) : 0;

            // 障碍格子
            if (grid[i][j] == 1) {
                if (up == 0 && left == 0) {
                    dp[next].put(s, dp[next].getOrDefault(s, 0L) + cnt);
                }
                continue;
            }

            // 尝试两种颜色
            for (int color = 1; color <= 2; color++) {
                if (up == color || left == color) {
                    continue;
                }
                if (grid[i][j] != 0 && grid[i][j] != color) {
                    continue;
                }

                long ns = set(s, j, color);
                if (j > 0) {
                    ns = set(ns, j-1, 0);
                }
                dp[next].put(ns, dp[next].getOrDefault(ns, 0L) + cnt);
            }
        }

        cur = next;
        next = 1 - cur;
    }

    // 换行处理
    dp[next].clear();
    for (Map.Entry<Long, Long> entry : dp[cur].entrySet()) {
```

```
                long s = entry.getKey();
                long cnt = entry.getValue();
                boolean valid = true;
                for (int j = 0; j < m; j++) {
                    if (get(s, j) != 0) {
                        valid = false;
                        break;
                    }
                }
                if (valid) {
                    dp[next].put(s, dp[next].getOrDefault(s, 0L) + cnt);
                }
            }

            cur = next;
            next = 1 - cur;
        }

        long ans = 0;
        for (long cnt : dp[cur].values()) {
            ans += cnt;
        }
        return ans;
    }

    public static void main(String[] args) {
        initPower3();
        Scanner sc = new Scanner(System.in);
        int T = sc.nextInt();
        while (T-- > 0) {
            n = sc.nextInt();
            m = sc.nextInt();
            grid = new int[n][m];
            for (int i = 0; i < n; i++) {
                for (int j = 0; j < m; j++) {
                    grid[i][j] = sc.nextInt();
                }
            }
            System.out.println(solve());
        }
        sc.close();
    }
}
```

时间复杂度：O(n * m * 3^m)
空间复杂度：O(3^m)

// 插头 DP 与轮廓线 DP 总结
/*
插头 DP 和轮廓线 DP 是解决网格类问题的高效算法。以下是关键要点：

1. 核心概念：
   - 轮廓线：正在处理的格子的边界
   - 插头：表示轮廓线上的连接状态
   - 状态压缩：使用不同进制表示轮廓线状态

2. 常见状态表示：
   - 二进制：适用于简单的存在性问题
   - 三进制：适用于需要区分插头类型的问题
   - 四进制：适用于更复杂的情况

3. 典型应用场景：
   - 骨牌覆盖问题
   - 网格着色问题
   - 路径覆盖问题
   - 回路覆盖问题
   - 障碍处理问题

4. 优化技巧：
   - 滚动数组优化空间
   - 使用哈希表存储有效状态
   - 预处理可能的转移状态
   - 交换网格行列以减少状态数
   - 剪枝无效状态

5. 工程化考量：
   - 注意数据类型的范围，防止溢出
   - 合理选择状态表示方式
   - 处理边界条件和特殊输入
   - 代码模块化设计
   - 添加详细注释和测试用例

6. 调试技巧：
   - 打印中间状态进行验证
   - 使用小例子测试
   - 检查状态转移的正确性

   - 验证边界条件

7. 学习建议：
   - 从简单问题入手，如骨牌覆盖
   - 理解轮廓线和插头的概念
   - 掌握不同进制状态表示的应用
   - 多练习不同类型的题目
   - 总结状态转移的规律

通过学习和实践这些算法，可以有效地解决各种复杂的网格类问题，提高算法设计和实现能力。
*/

==

文件: Code03_TspTwice.java
==
```
package class126;

// 节点最多经过两次的tsp问题
// 给定有n个地点，用m条边无向边连接，每条边有权值
// 你可以任选一点出发，目标是经过所有的点，最终不必回到出发点
// 并且每个点最多可以到达两次
// 返回总路程最小是多少
// 1 <= n <= 10
// 1 <= m <= 100
// 测试链接 : https://acm.hdu.edu.cn/showproblem.php?pid=3001
// 提交以下的code，提交时请把类名改成"Main"，可以通过所有用例
//
// 题目大意：
// 给定一个无向图，要求找到一条路径，使得：
// 1. 经过所有节点至少一次
// 2. 每个节点最多经过两次
// 3. 路径长度最小
//
// 解题思路：
// 使用三进制状态压缩DP。
// 状态表示：用三进制表示每个节点的访问次数，0表示未访问，1表示访问一次，2表示访问两次。
// 状态转移：通过枚举当前节点和状态，计算到达该状态的最小路径长度。
//
// Java实现：https://github.com/yourusername/algorithm-
journey/blob/main/src/class126/Code03_TspTwice.java
// C++实现：https://github.com/yourusername/algorithm-
journey/blob/main/src/class126/Code03_TspTwice.cpp
```

```
// Python 实现：https://github.com/yourusername/algorithm-journey/blob/main/src/class126/Code03_TspTwice.py

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_TspTwice {

    public static int MAXN = 10;

    public static int MAXS = (int) Math.pow(3, MAXN);

    public static int n;

    public static int m;

    public static int maxs;

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

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

    public static int[] complete = new int[1 << MAXN];

    public static int size;

    public static void build() {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                graph[i][j] = Integer.MAX_VALUE;
            }
        }
        maxs = (int) Math.pow(3, n);
        for (int i = 0; i < n; i++) {
            for (int s = 0; s < maxs; s++) {
                dp[i][s] = -1;
            }
        }
        size = 0;
```

```java
        dfs(0, 1, 0);
    }

    public static void dfs(int i, int bit, int s) {
        if (i == n) {
            complete[size++] = s;
        } else {
            dfs(i + 1, bit * 3, s + bit);
            dfs(i + 1, bit * 3, s + 2 * bit);
        }
    }

    public static int compute() {
        int ans = Integer.MAX_VALUE;
        for (int k = 0; k < size; k++) {
            for (int i = 0, bit = 1; i < n; i++, bit *= 3) {
                ans = Math.min(ans, f(i, complete[k] - bit));
            }
        }
        return ans;
    }

    public static int f(int i, int s) {
        if (s == 0) {
            return 0;
        }
        if (dp[i][s] != -1) {
            return dp[i][s];
        }
        int ans = Integer.MAX_VALUE;
        for (int j = 0, bit = 1, pre; j < n; j++, bit *= 3) {
            if ((s / bit) % 3 > 0) {
                pre = f(j, s - bit);
                if (pre != Integer.MAX_VALUE && graph[j][i] != Integer.MAX_VALUE) {
                    ans = Math.min(ans, pre + graph[j][i]);
                }
            }
        }
        dp[i][s] = ans;
        return ans;
    }

    public static void main(String[] args) throws IOException {
```

```java
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        StreamTokenizer in = new StreamTokenizer(br);
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
        while (in.nextToken() != StreamTokenizer.TT_EOF) {
            n = (int) in.nval;
            in.nextToken();
            m = (int) in.nval;
            build();
            for (int i = 1, u, v, w; i <= m; i++) {
                in.nextToken();
                u = (int) in.nval - 1;
                in.nextToken();
                v = (int) in.nval - 1;
                in.nextToken();
                w = (int) in.nval;
                if (w < graph[u][v]) {
                    graph[u][v] = graph[v][u] = w;
                }
            }
            int ans = compute();
            out.println(ans == Integer.MAX_VALUE ? -1 : ans);
        }
        out.flush();
        out.close();
        br.close();
    }

}
```

==

文件: HDU1693_EatTheTrees.cpp

==

```cpp
// HDU 1693 Eat the Trees (插头 DP - 多回路覆盖)
// 在 n×m 的网格中，求用若干个回路覆盖所有非障碍格子的方案数
// 1 <= n, m <= 11
// 测试链接 : http://acm.hdu.edu.cn/showproblem.php?pid=1693
//
// 题目大意:
// 给定一个 n×m 的网格，其中一些格子是障碍物（用 0 表示），其他格子是可通行的（用 1 表示）。
// 要求用若干个回路（闭合路径）覆盖所有可通行的格子，每个格子恰好被一个回路覆盖。
// 求满足条件的方案数。
//
```

```
// 解题思路：
// 使用插头 DP 解决多回路覆盖问题。
// 状态表示：用二进制表示轮廓线状态，第 k 位为 1 表示第 k 个位置有插头。
// 状态转移：
// 1. 当前格子是障碍，则不能放置插头
// 2. 当前格子不是障碍，则可以：
//    a. 不放置插头（合并左右插头）
//    b. 放置插头（延续插头或创建新插头对）
//
// Java 实现：https://github.com/yourusername/algorithm-journey/blob/main/src/class126/HDU1693_EatTheTrees.java
// C++实现：https://github.com/yourusername/algorithm-journey/blob/main/src/class126/HDU1693_EatTheTrees.cpp
// Python 实现：https://github.com/yourusername/algorithm-journey/blob/main/src/class126/HDU1693_EatTheTrees.py

const int MAXN = 12;
const int MAX_STATES = (1 << MAXN); // 2^11 = 2048

// dp[i][j][s]表示处理到第 i 行第 j 列，轮廓线状态为 s 的方案数
// 状态 s 用二进制表示，第 k 位为 1 表示第 k 个位置有插头
long long dp[MAXN][MAXN][MAX_STATES];

int grid[MAXN][MAXN];
int n, m;

/**
 * 计算用若干个回路覆盖所有非障碍格子的方案数
 *
 * 算法思路：
 * 使用插头 DP 解决多回路覆盖问题
 * 状态表示：用二进制表示轮廓线状态，第 k 位为 1 表示第 k 个位置有插头
 * 状态转移：
 * 1. 当前格子是障碍，则不能放置插头
 * 2. 当前格子不是障碍，则可以：
 *    a. 不放置插头（合并左右插头）
 *    b. 放置插头（延续插头或创建新插头对）
 *
 * 时间复杂度：O(n * m * 2^m)
 * 空间复杂度：O(n * m * 2^m)
 *
 * @param rows 行数
 * @param cols 列数
```

```
 * @param maze 网格地图，0 表示障碍，1 表示可通行
 * @return 方案数
 */
long long solve(int rows, int cols, int maze[][MAXN]) {
    n = rows;
    m = cols;

    // 复制网格
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            grid[i][j] = maze[i][j];
        }
    }

    // 初始化 DP 数组
    for (int i = 0; i < MAXN; i++) {
        for (int j = 0; j < MAXN; j++) {
            for (int k = 0; k < MAX_STATES; k++) {
                dp[i][j][k] = 0;
            }
        }
    }

    // 初始状态
    dp[0][0][0] = 1;

    // 逐格 DP
    for (int i = 0; i < n; i++) {
        // 行间转移
        for (int s = 0; s < (1 << m); s++) {
            if (dp[i][m][s] > 0) {
                // 将状态转移到下一行的开始
                dp[i+1][0][s] = dp[i][m][s];
            }
        }

        // 行内转移
        for (int j = 0; j < m; j++) {
            for (int s = 0; s < (1 << m); s++) {
                if (dp[i][j][s] == 0) continue;

                // 获取当前格子左边和上面的插头状态
                int left = (j > 0 && ((s >> (j-1)) & 1) == 1) ? 1 : 0;
```

```
                int up = ((s >> j) & 1);

                // 如果是障碍格子
                if (grid[i][j] == 0) {
                    // 只能在没有插头的情况下转移
                    if (left == 0 && up == 0) {
                        dp[i][j+1][s & (~((1 << j) | (1 << (j-1))))] += dp[i][j][s];
                    }
                } else {
                    // 可通行格子

                    // 1. 不放置插头（合并两个插头）
                    if (left == 1 && up == 1) {
                        dp[i][j+1][s & (~((1 << j) | (1 << (j-1))))] += dp[i][j][s];
                    }

                    // 2. 延续插头
                    if (left == 1 && up == 0) {
                        // 延续左插头到上方
                        dp[i][j+1][s | (1 << j)] += dp[i][j][s];
                    }

                    if (left == 0 && up == 1) {
                        // 延续上插头到左方
                        dp[i][j+1][s | (1 << (j-1))] += dp[i][j][s];
                    }

                    // 3. 创建新插头对（如果左右和上方都没有插头）
                    if (left == 0 && up == 0) {
                        // 创建一对新插头（左插头和上插头）
                        dp[i][j+1][s | (1 << (j-1)) | (1 << j)] += dp[i][j][s];
                    }
                }
            }
        }
    }

    return dp[n][0][0];
}

// 测试用例
int main() {
    int maze1[3][MAXN] = {
```

```
        {1, 1, 1},
        {1, 1, 1},
        {1, 1, 1}
    };
    // 测试用例，实际使用时需要添加输出语句
    solve(3, 3, maze1);

    int maze2[3][MAXN] = {
        {1, 1, 0},
        {1, 1, 1},
        {1, 1, 1}
    };
    solve(3, 3, maze2);

    return 0;
}
```

==

文件: HDU1693_EatTheTrees.java

==

```
package class126;

// HDU 1693 Eat the Trees (插头 DP - 多回路覆盖)
// 在 n×m 的网格中，求用若干个回路覆盖所有非障碍格子的方案数
// 1 <= n, m <= 11
// 测试链接 : http://acm.hdu.edu.cn/showproblem.php?pid=1693
//
// 题目大意：
// 给定一个 n×m 的网格，其中一些格子是障碍物（用 0 表示），其他格子是可通行的（用 1 表示）。
// 要求用若干个回路（闭合路径）覆盖所有可通行的格子，每个格子恰好被一个回路覆盖。
// 求满足条件的方案数。
//
// 解题思路：
// 使用插头 DP 解决多回路覆盖问题。
// 状态表示：用二进制表示轮廓线状态，第 k 位为 1 表示第 k 个位置有插头。
// 状态转移：
// 1. 当前格子是障碍，则不能放置插头
// 2. 当前格子不是障碍，则可以：
//    a. 不放置插头（合并左右插头）
//    b. 放置插头（延续插头或创建新插头对）
//
// Java 实现：https://github.com/yourusername/algorithm-
```

```
journey/blob/main/src/class126/HDU1693_EatTheTrees.java
// C++实现：https://github.com/yourusername/algorithm-
journey/blob/main/src/class126/HDU1693_EatTheTrees.cpp
// Python 实现：https://github.com/yourusername/algorithm-
journey/blob/main/src/class126/HDU1693_EatTheTrees.py

public class HDU1693_EatTheTrees {

    public static int MAXN = 12;
    public static int MAX_STATES = (1 << MAXN); // 2^11 = 2048

    // dp[i][j][s]表示处理到第 i 行第 j 列，轮廓线状态为 s 的方案数
    // 状态 s 用二进制表示，第 k 位为 1 表示第 k 个位置有插头
    public static long[][][] dp = new long[MAXN][MAXN][MAX_STATES];

    public static int[][] grid = new int[MAXN][MAXN];
    public static int n, m;

    /**
     * 计算用若干个回路覆盖所有非障碍格子的方案数
     *
     * 算法思路：
     * 使用插头 DP 解决多回路覆盖问题
     * 状态表示：用二进制表示轮廓线状态，第 k 位为 1 表示第 k 个位置有插头
     * 状态转移：
     * 1. 当前格子是障碍，则不能放置插头
     * 2. 当前格子不是障碍，则可以：
     *    a. 不放置插头（合并左右插头）
     *    b. 放置插头（延续插头或创建新插头对）
     *
     * 时间复杂度：O(n * m * 2^m)
     * 空间复杂度：O(n * m * 2^m)
     *
     * @param rows 行数
     * @param cols 列数
     * @param maze 网格地图，0 表示障碍，1 表示可通行
     * @return 方案数
     */
    public static long solve(int rows, int cols, int[][] maze) {
        n = rows;
        m = cols;

        // 复制网格
```

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
        grid[i][j] = maze[i][j];
    }
}

// 初始化 DP 数组
for (int i = 0; i <= n; i++) {
    for (int j = 0; j <= m; j++) {
        for (int s = 0; s < MAX_STATES; s++) {
            dp[i][j][s] = 0;
        }
    }
}

// 初始状态
dp[0][0][0] = 1;

// 逐格 DP
for (int i = 0; i < n; i++) {
    // 行间转移
    for (int s = 0; s < (1 << m); s++) {
        if (dp[i][m][s] > 0) {
            // 将状态转移到下一行的开始
            dp[i+1][0][s] = dp[i][m][s];
        }
    }

    // 行内转移
    for (int j = 0; j < m; j++) {
        for (int s = 0; s < (1 << m); s++) {
            if (dp[i][j][s] == 0) continue;

            // 获取当前格子左边和上面的插头状态
            int left = (j > 0 && ((s >> (j-1)) & 1) == 1) ? 1 : 0;
            int up = ((s >> j) & 1);

            // 如果是障碍格子
            if (grid[i][j] == 0) {
                // 只能在没有插头的情况下转移
                if (left == 0 && up == 0) {
                    dp[i][j+1][s & (~((1 << j) | (1 << (j-1))))] += dp[i][j][s];
                }
```

```
                } else {
                    // 可通行格子

                    // 1. 不放置插头（合并两个插头）
                    if (left == 1 && up == 1) {
                        dp[i][j+1][s & (~((1 << j) | (1 << (j-1))))] += dp[i][j][s];
                    }

                    // 2. 延续插头
                    if (left == 1 && up == 0) {
                        // 延续左插头到上方
                        dp[i][j+1][s | (1 << j)] += dp[i][j][s];
                    }

                    if (left == 0 && up == 1) {
                        // 延续上插头到左方
                        dp[i][j+1][s | (1 << (j-1))] += dp[i][j][s];
                    }

                    // 3. 创建新插头对（如果左右和上方都没有插头）
                    if (left == 0 && up == 0) {
                        // 创建一对新插头（左插头和上插头）
                        dp[i][j+1][s | (1 << (j-1)) | (1 << j)] += dp[i][j][s];
                    }
                }
            }
        }
    }

    return dp[n][0][0];
}

// 测试用例
public static void main(String[] args) {
    int[][] maze1 = {
        {1, 1, 1},
        {1, 1, 1},
        {1, 1, 1}
    };
    System.out.println(solve(3, 3, maze1)); // 输出方案数

    int[][] maze2 = {
        {1, 1, 0},
```

```
            {1, 1, 1},
            {1, 1, 1}
        };
        System.out.println(solve(3, 3, maze2)); // 输出方案数
    }
}
```

===

文件: HDU1693_EatTheTrees.py

===

```
# HDU 1693 Eat the Trees (插头 DP - 多回路覆盖)
# 在 n×m 的网格中，求用若干个回路覆盖所有非障碍格子的方案数
# 1 <= n, m <= 11
# 测试链接 : http://acm.hdu.edu.cn/showproblem.php?pid=1693
#
# 题目大意：
# 给定一个 n×m 的网格，其中一些格子是障碍物（用 0 表示），其他格子是可通行的（用 1 表示）。
# 要求用若干个回路（闭合路径）覆盖所有可通行的格子，每个格子恰好被一个回路覆盖。
# 求满足条件的方案数。
#
# 解题思路：
# 使用插头 DP 解决多回路覆盖问题。
# 状态表示：用二进制表示轮廓线状态，第 k 位为 1 表示第 k 个位置有插头。
# 状态转移：
# 1. 当前格子是障碍，则不能放置插头
# 2. 当前格子不是障碍，则可以：
#    a. 不放置插头（合并左右插头）
#    b. 放置插头（延续插头或创建新插头对）
#
# Java 实现：https://github.com/yourusername/algorithm-journey/blob/main/src/class126/HDU1693_EatTheTrees.java
# C++实现：https://github.com/yourusername/algorithm-journey/blob/main/src/class126/HDU1693_EatTheTrees.cpp
# Python 实现：https://github.com/yourusername/algorithm-journey/blob/main/src/class126/HDU1693_EatTheTrees.py

MAXN = 12
MAX_STATES = (1 << MAXN)  # 2^11 = 2048

# dp[i][j][s]表示处理到第 i 行第 j 列，轮廓线状态为 s 的方案数
# 状态 s 用二进制表示，第 k 位为 1 表示第 k 个位置有插头
dp = [[[0 for _ in range(MAX_STATES)] for _ in range(MAXN)] for _ in range(MAXN)]
```

```
grid = [[0 for _ in range(MAXN)] for _ in range(MAXN)]
n, m = 0, 0

def solve(rows, cols, maze):
    """
    计算用若干个回路覆盖所有非障碍格子的方案数

    算法思路：
    使用插头 DP 解决多回路覆盖问题
    状态表示：用二进制表示轮廓线状态，第 k 位为 1 表示第 k 个位置有插头
    状态转移：
    1. 当前格子是障碍，则不能放置插头
    2. 当前格子不是障碍，则可以：
       a. 不放置插头（合并左右插头）
       b. 放置插头（延续插头或创建新插头对）

    时间复杂度：O(n * m * 2^m)
    空间复杂度：O(n * m * 2^m)

    Args:
        rows: 行数
        cols: 列数
        maze: 网格地图，0 表示障碍，1 表示可通行

    Returns:
        方案数
    """
    global n, m, grid, dp

    n = rows
    m = cols

    # 复制网格
    for i in range(n):
        for j in range(m):
            grid[i][j] = maze[i][j]

    # 初始化 DP 数组
    for i in range(n + 1):
        for j in range(m + 1):
            for s in range(MAX_STATES):
                dp[i][j][s] = 0
```

```python
# 初始状态
dp[0][0][0] = 1

# 逐格 DP
for i in range(n):
    # 行间转移
    for s in range(1 << m):
        if dp[i][m][s] > 0:
            # 将状态转移到下一行的开始
            dp[i+1][0][s] = dp[i][m][s]

    # 行内转移
    for j in range(m):
        for s in range(1 << m):
            if dp[i][j][s] == 0:
                continue

            # 获取当前格子左边和上面的插头状态
            left = 1 if (j > 0 and ((s >> (j-1)) & 1) == 1) else 0
            up = (s >> j) & 1

            # 如果是障碍格子
            if grid[i][j] == 0:
                # 只能在没有插头的情况下转移
                if left == 0 and up == 0:
                    new_state = s & (~((1 << j) | (1 << (j-1))))
                    dp[i][j+1][new_state] += dp[i][j][s]
            else:
                # 可通行格子

                # 1. 不放置插头（合并两个插头）
                if left == 1 and up == 1:
                    new_state = s & (~((1 << j) | (1 << (j-1))))
                    dp[i][j+1][new_state] += dp[i][j][s]

                # 2. 延续插头
                if left == 1 and up == 0:
                    # 延续左插头到上方
                    new_state = s | (1 << j)
                    dp[i][j+1][new_state] += dp[i][j][s]

                if left == 0 and up == 1:
```

```python
                        # 延续上插头到左方
                        new_state = s | (1 << (j-1))
                        dp[i][j+1][new_state] += dp[i][j][s]

                    # 3. 创建新插头对（如果左右和上方都没有插头）
                    if left == 0 and up == 0:
                        # 创建一对新插头（左插头和上插头）
                        new_state = s | (1 << (j-1)) | (1 << j)
                        dp[i][j+1][new_state] += dp[i][j][s]

    return dp[n][0][0]

# 测试用例
if __name__ == "__main__":
    maze1 = [
        [1, 1, 1],
        [1, 1, 1],
        [1, 1, 1]
    ]
    print(solve(3, 3, maze1))  # 输出方案数

    maze2 = [
        [1, 1, 0],
        [1, 1, 1],
        [1, 1, 1]
    ]
    print(solve(3, 3, maze2))  # 输出方案数
```

===

文件: HDU4285_Circuits.cpp

===

```cpp
// HDU 4285 circuits (插头 DP - 限定回路数)
// 在 n×m 的网格中，求形成恰好 k 个不相交回路的方案数
// 1 <= n, m <= 12, 1 <= k <= 10
// 测试链接 : http://acm.hdu.edu.cn/showproblem.php?pid=4285
//
// 题目大意:
// 给定一个 n×m 的网格，其中一些格子是障碍物（用 1 表示），其他格子是可通行的（用 0 表示）。
// 要求找到恰好形成 k 个不相交回路的方案数，每个回路覆盖一些可通行格子。
//
// 解题思路:
// 使用插头 DP 解决限定回路数问题。
```

```
// 状态表示：用最小表示法表示轮廓线上的连通性状态。
// 状态转移：根据插头的连通性进行合并、创建新回路等操作。
//
// Java 实现：https://github.com/yourusername/algorithm-journey/blob/main/src/class126/HDU4285_Circuits.java
// C++实现：https://github.com/yourusername/algorithm-journey/blob/main/src/class126/HDU4285_Circuits.cpp
// Python 实现：https://github.com/yourusername/algorithm-journey/blob/main/src/class126/HDU4285_Circuits.py

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

using namespace std;

const int MAXN = 15;
const int MAXK = 15;
const int MOD = 1000000007;

// dp[i][j][s][k]表示处理到第 i 行第 j 列，轮廓线状态为 s，已形成 k 个回路的方案数
// 状态 s 用最小表示法编码连通性信息
int dp[MAXN][MAXN][1 << (2 * MAXN)][MAXK];

int grid[MAXN][MAXN];
int n, m, K;

/**
 * 计算形成恰好 K 个不相交回路的方案数
 *
 * 算法思路：
 * 使用插头 DP 解决限定回路数问题
 * 状态表示：用最小表示法表示轮廓线上的连通性状态
 * 状态转移：根据插头的连通性进行合并、创建新回路等操作
 *
 * 时间复杂度：O(n * m * 2^(2*m) * K)
 * 空间复杂度：O(n * m * 2^(2*m) * K)
 *
 * @param rows 行数
 * @param cols 列数
 * @param k 回路数
 * @param maze 网格，0 表示可经过，1 表示障碍
```

```cpp
 * @return 形成k个回路的方案数
 */
int solve(int rows, int cols, int k, vector<vector<int>>& maze) {
    n = rows;
    m = cols;
    K = k;

    // 复制网格
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            grid[i][j] = maze[i][j];
        }
    }

    // 初始化DP数组
    memset(dp, 0, sizeof(dp));

    // 初始状态
    dp[0][0][0][0] = 1;

    // 逐格DP
    for (int i = 0; i < n; i++) {
        // 行间转移
        for (int s = 0; s < (1 << (2 * m)); s++) {
            for (int t = 0; t <= K; t++) {
                if (dp[i][m][s][t] > 0) {
                    // 将状态转移到下一行的开始
                    dp[i+1][0][s << 2][t] = (dp[i+1][0][s << 2][t] + dp[i][m][s][t]) % MOD;
                }
            }
        }

        // 行内转移
        for (int j = 0; j < m; j++) {
            for (int s = 0; s < (1 << (2 * m + 2)); s++) {
                for (int t = 0; t <= K; t++) {
                    if (dp[i][j][s][t] == 0) continue;

                    // 获取当前格子左边和上面的插头状态
                    int left = j > 0 ? ((s >> (2 * (j - 1))) & 3) : 0;
                    int up = (s >> (2 * j)) & 3;

                    // 如果是障碍格子
```

```
                if (grid[i][j] == 1) {
                    // 只能在没有插头的情况下转移
                    if (left == 0 && up == 0) {
                        int newState = s & (~((3 << (2 * (j - 1))) | (3 << (2 * j))));
                        dp[i][j+1][newState][t] = (dp[i][j+1][newState][t] +
dp[i][j][s][t]) % MOD;
                    }
                } else {
                    // 可通行格子

                    // 1. 不放置插头（合并两个插头）
                    if (left != 0 && up != 0) {
                        int newState = s;
                        newState &= ~((3 << (2 * (j - 1))) | (3 << (2 * j)));

                        // 如果两个插头属于不同连通分量，则合并
                        // 如果两个插头属于相同连通分量，则形成新回路
                        if (left == up) {
                            // 形成新回路
                            if (t + 1 <= K) {
                                dp[i][j+1][newState][t+1] = (dp[i][j+1][newState][t+1] +
dp[i][j][s][t]) % MOD;
                            }
                        } else {
                            // 合并连通分量
                            // 需要重新编号保持最小表示法
                            newState = renumber(newState, j-1, j, left, up);
                            dp[i][j+1][newState][t] = (dp[i][j+1][newState][t] +
dp[i][j][s][t]) % MOD;
                        }
                    }

                    // 2. 延续插头
                    if (left != 0 && up == 0) {
                        // 延续左插头到上方
                        int newState = s;
                        newState &= ~(3 << (2 * (j - 1)));
                        newState |= (left << (2 * j));
                        dp[i][j+1][newState][t] = (dp[i][j+1][newState][t] +
dp[i][j][s][t]) % MOD;
                    }

                    if (left == 0 && up != 0) {
```

```
                                // 延续上插头到左方
                                int newState = s;
                                newState &= ~(3 << (2 * j));
                                newState |= (up << (2 * (j - 1)));
                                dp[i][j+1][newState][t] = (dp[i][j+1][newState][t] +
dp[i][j][s][t]) % MOD;
                            }

                            // 3. 创建新插头对（如果左右和上方都没有插头）
                            if (left == 0 && up == 0) {
                                // 创建一对新插头（左插头和上插头）
                                int newState = s | (1 << (2 * (j - 1))) | (1 << (2 * j));
                                dp[i][j+1][newState][t] = (dp[i][j+1][newState][t] +
dp[i][j][s][t]) % MOD;
                            }
                        }
                    }
                }
            }
    }

    // 统计形成恰好 K 个回路的方案数
    int result = 0;
    for (int s = 0; s < (1 << (2 * m)); s++) {
        result = (result + dp[n][0][s][K]) % MOD;
    }
    return result;
}

/**
 * 重新编号以保持最小表示法
 */
int renumber(int state, int pos1, int pos2, int id1, int id2) {
    // 合并两个连通分量，将 id2 的编号改为 id1
    int minId = min(id1, id2);
    int maxId = max(id1, id2);

    int m = (state >> (2 * pos1)) & 3;
    if (m == maxId) {
        state &= ~(3 << (2 * pos1));
        state |= (minId << (2 * pos1));
    }
```

```
    m = (state >> (2 * pos2)) & 3;
    if (m == maxId) {
        state &= ~(3 << (2 * pos2));
        state |= (minId << (2 * pos2));
    }

    return state;
}

// 测试用例
int main() {
    vector<vector<int>> maze1 = {
        {0, 0, 0},
        {0, 0, 0},
        {0, 0, 0}
    };
    cout << solve(3, 3, 2, maze1) << endl; // 输出形成 2 个回路的方案数

    return 0;
}
```

==

文件: HDU4285_Circuits.java

==

```
package class126;

// HDU 4285 circuits (插头 DP - 限定回路数)
// 在 n×m 的网格中，求形成恰好 k 个不相交回路的方案数
// 1 <= n, m <= 12, 1 <= k <= 10
// 测试链接 : http://acm.hdu.edu.cn/showproblem.php?pid=4285
//
// 题目大意：
// 给定一个 n×m 的网格，其中一些格子是障碍物（用 1 表示），其他格子是可通行的（用 0 表示）。
// 要求找到恰好形成 k 个不相交回路的方案数，每个回路覆盖一些可通行格子。
//
// 解题思路：
// 使用插头 DP 解决限定回路数问题。
// 状态表示：用最小表示法表示轮廓线上的连通性状态。
// 状态转移：根据插头的连通性进行合并、创建新回路等操作。
//
// Java 实现：https://github.com/yourusername/algorithm-
journey/blob/main/src/class126/HDU4285_Circuits.java
```

```java
// C++实现：https://github.com/yourusername/algorithm-
journey/blob/main/src/class126/HDU4285_Circuits.cpp
// Python 实现：https://github.com/yourusername/algorithm-
journey/blob/main/src/class126/HDU4285_Circuits.py

public class HDU4285_Circuits {

    public static int MAXN = 15;
    public static int MAXK = 15;
    public static int MOD = 1000000007;

    // dp[i][j][s][k]表示处理到第 i 行第 j 列，轮廓线状态为 s，已形成 k 个回路的方案数
    // 状态 s 用最小表示法编码连通性信息
    public static int[][][][] dp = new int[MAXN][MAXN][1 << (2 * MAXN)][MAXK];

    public static int[][] grid = new int[MAXN][MAXN];
    public static int n, m, K;

    /**
     * 计算形成恰好 K 个不相交回路的方案数
     *
     * 算法思路：
     * 使用插头 DP 解决限定回路数问题
     * 状态表示：用最小表示法表示轮廓线上的连通性状态
     * 状态转移：根据插头的连通性进行合并、创建新回路等操作
     *
     * 时间复杂度：O(n * m * 2^(2*m) * K)
     * 空间复杂度：O(n * m * 2^(2*m) * K)
     *
     * @param rows 行数
     * @param cols 列数
     * @param k 回路数
     * @param maze 网格，0 表示可经过，1 表示障碍
     * @return 形成 k 个回路的方案数
     */
    public static int solve(int rows, int cols, int k, int[][] maze) {
        n = rows;
        m = cols;
        K = k;

        // 复制网格
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
```

```
            grid[i][j] = maze[i][j];
        }
    }

    // 初始化 DP 数组
    for (int i = 0; i <= n; i++) {
        for (int j = 0; j <= m; j++) {
            for (int s = 0; s < (1 << (2 * MAXN)); s++) {
                for (int t = 0; t <= K; t++) {
                    dp[i][j][s][t] = 0;
                }
            }
        }
    }

    // 初始状态
    dp[0][0][0][0] = 1;

    // 逐格 DP
    for (int i = 0; i < n; i++) {
        // 行间转移
        for (int s = 0; s < (1 << (2 * m)); s++) {
            for (int t = 0; t <= K; t++) {
                if (dp[i][m][s][t] > 0) {
                    // 将状态转移到下一行的开始
                    dp[i+1][0][s << 2][t] = (dp[i+1][0][s << 2][t] + dp[i][m][s][t]) % MOD;
                }
            }
        }

        // 行内转移
        for (int j = 0; j < m; j++) {
            for (int s = 0; s < (1 << (2 * m + 2)); s++) {
                for (int t = 0; t <= K; t++) {
                    if (dp[i][j][s][t] == 0) continue;

                    // 获取当前格子左边和上面的插头状态
                    int left = j > 0 ? ((s >> (2 * (j - 1))) & 3) : 0;
                    int up = (s >> (2 * j)) & 3;

                    // 如果是障碍格子
                    if (grid[i][j] == 1) {
                        // 只能在没有插头的情况下转移
```

```
                        if (left == 0 && up == 0) {
                            int newState = s & (~((3 << (2 * (j - 1))) | (3 << (2 * j))));
                            dp[i][j+1][newState][t] = (dp[i][j+1][newState][t] +
dp[i][j][s][t]) % MOD;
                        }
                    } else {
                        // 可通行格子

                        // 1. 不放置插头（合并两个插头）
                        if (left != 0 && up != 0) {
                            int newState = s;
                            newState &= ~((3 << (2 * (j - 1))) | (3 << (2 * j)));

                            // 如果两个插头属于不同连通分量，则合并
                            // 如果两个插头属于相同连通分量，则形成新回路
                            if (left == up) {
                                // 形成新回路
                                if (t + 1 <= K) {
                                    dp[i][j+1][newState][t+1] = (dp[i][j+1][newState][t+1] +
dp[i][j][s][t]) % MOD;
                                }
                            } else {
                                // 合并连通分量
                                // 需要重新编号保持最小表示法
                                newState = renumber(newState, j-1, j, left, up);
                                dp[i][j+1][newState][t] = (dp[i][j+1][newState][t] +
dp[i][j][s][t]) % MOD;
                            }
                        }

                        // 2. 延续插头
                        if (left != 0 && up == 0) {
                            // 延续左插头到上方
                            int newState = s;
                            newState &= ~(3 << (2 * (j - 1)));
                            newState |= (left << (2 * j));
                            dp[i][j+1][newState][t] = (dp[i][j+1][newState][t] +
dp[i][j][s][t]) % MOD;
                        }

                        if (left == 0 && up != 0) {
                            // 延续上插头到左方
                            int newState = s;
```

```
                                newState &= ~(3 << (2 * j));
                                newState |= (up << (2 * (j - 1)));
                                dp[i][j+1][newState][t] = (dp[i][j+1][newState][t] +
dp[i][j][s][t]) % MOD;
                            }

                            // 3. 创建新插头对（如果左右和上方都没有插头）
                            if (left == 0 && up == 0) {
                                // 创建一对新插头（左插头和上插头）
                                int newState = s | (1 << (2 * (j - 1))) | (1 << (2 * j));
                                dp[i][j+1][newState][t] = (dp[i][j+1][newState][t] +
dp[i][j][s][t]) % MOD;
                            }
                        }
                    }
                }
            }
        }

        // 统计形成恰好 K 个回路的方案数
        int result = 0;
        for (int s = 0; s < (1 << (2 * m)); s++) {
            result = (result + dp[n][0][s][K]) % MOD;
        }
        return result;
    }

    /**
     * 重新编号以保持最小表示法
     */
    public static int renumber(int state, int pos1, int pos2, int id1, int id2) {
        // 合并两个连通分量，将 id2 的编号改为 id1
        int minId = Math.min(id1, id2);
        int maxId = Math.max(id1, id2);

        int m = (state >> (2 * pos1)) & 3;
        if (m == maxId) {
            state &= ~(3 << (2 * pos1));
            state |= (minId << (2 * pos1));
        }

        m = (state >> (2 * pos2)) & 3;
        if (m == maxId) {
```

```
            state &= ~(3 << (2 * pos2));
            state |= (minId << (2 * pos2));
        }

        return state;
    }

    // 测试用例
    public static void main(String[] args) {
        int[][] maze1 = {
            {0, 0, 0},
            {0, 0, 0},
            {0, 0, 0}
        };
        System.out.println(solve(3, 3, 2, maze1)); // 输出形成 2 个回路的方案数
    }
}
```

===

文件: HDU4285_Circuits.py

===

```
# HDU 4285 circuits (插头 DP - 限定回路数)
# 在 n×m 的网格中，求形成恰好 k 个不相交回路的方案数
# 1 <= n, m <= 12, 1 <= k <= 10
# 测试链接 : http://acm.hdu.edu.cn/showproblem.php?pid=4285
#
# 题目大意:
# 给定一个 n×m 的网格，其中一些格子是障碍物（用 1 表示），其他格子是可通行的（用 0 表示）。
# 要求找到恰好形成 k 个不相交回路的方案数，每个回路覆盖一些可通行格子。
#
# 解题思路:
# 使用插头 DP 解决限定回路数问题。
# 状态表示：用最小表示法表示轮廓线上的连通性状态。
# 状态转移：根据插头的连通性进行合并、创建新回路等操作。
#
# Java 实现：https://github.com/yourusername/algorithm-journey/blob/main/src/class126/HDU4285_Circuits.java
# C++实现：https://github.com/yourusername/algorithm-journey/blob/main/src/class126/HDU4285_Circuits.cpp
# Python 实现：https://github.com/yourusername/algorithm-journey/blob/main/src/class126/HDU4285_Circuits.py
```

```python
MAXN = 15
MAXK = 15
MOD = 1000000007

# dp[i][j][s][k]表示处理到第 i 行第 j 列，轮廓线状态为 s，已形成 k 个回路的方案数
# 状态 s 用最小表示法编码连通性信息
dp = [[[[0] * MAXK for _ in range(1 << (2 * MAXN))] for _ in range(MAXN)] for _ in range(MAXN)]

grid = [[0] * MAXN for _ in range(MAXN)]
n, m, K = 0, 0, 0

def solve(rows, cols, k, maze):
    """
    计算形成恰好 K 个不相交回路的方案数

    算法思路：
    使用插头 DP 解决限定回路数问题
    状态表示：用最小表示法表示轮廓线上的连通性状态
    状态转移：根据插头的连通性进行合并、创建新回路等操作

    时间复杂度：O(n * m * 2^(2*m) * K)
    空间复杂度：O(n * m * 2^(2*m) * K)

    Args:
        rows: 行数
        cols: 列数
        k: 回路数
        maze: 网格，0 表示可经过，1 表示障碍

    Returns:
        形成 k 个回路的方案数
    """
    global n, m, K

    n = rows
    m = cols
    K = k

    # 复制网格
    for i in range(n):
        for j in range(m):
            grid[i][j] = maze[i][j]
```

```python
    # 初始化 DP 数组
    for i in range(MAXN):
        for j in range(MAXN):
            for s in range(1 << (2 * MAXN)):
                for t in range(MAXK):
                    dp[i][j][s][t] = 0

    # 初始状态
    dp[0][0][0][0] = 1

    # 逐格 DP
    for i in range(n):
        # 行间转移
        for s in range(1 << (2 * m)):
            for t in range(K + 1):
                if dp[i][m][s][t] > 0:
                    # 将状态转移到下一行的开始
                    new_state = s << 2
                    dp[i+1][0][new_state][t] = (dp[i+1][0][new_state][t] + dp[i][m][s][t]) % MOD

        # 行内转移
        for j in range(m):
            for s in range(1 << (2 * m + 2)):
                for t in range(K + 1):
                    if dp[i][j][s][t] == 0:
                        continue

                    # 获取当前格子左边和上面的插头状态
                    left = ((s >> (2 * (j - 1))) & 3) if j > 0 else 0
                    up = (s >> (2 * j)) & 3

                    # 如果是障碍格子
                    if grid[i][j] == 1:
                        # 只能在没有插头的情况下转移
                        if left == 0 and up == 0:
                            new_state = s & (~((3 << (2 * (j - 1))) | (3 << (2 * j))))
                            dp[i][j+1][new_state][t] = (dp[i][j+1][new_state][t] +
dp[i][j][s][t]) % MOD
                    else:
                        # 可通行格子

                        # 1. 不放置插头（合并两个插头）
                        if left != 0 and up != 0:
```

```
                            new_state = s
                            new_state &= ~((3 << (2 * (j - 1))) | (3 << (2 * j)))

                            # 如果两个插头属于不同连通分量，则合并
                            # 如果两个插头属于相同连通分量，则形成新回路
                            if left == up:
                                # 形成新回路
                                if t + 1 <= K:
                                    dp[i][j+1][new_state][t+1] = (dp[i][j+1][new_state][t+1] + dp[i][j][s][t]) % MOD
                            else:
                                # 合并连通分量
                                # 需要重新编号保持最小表示法
                                new_state = renumber(new_state, j-1, j, left, up)
                                dp[i][j+1][new_state][t] = (dp[i][j+1][new_state][t] + dp[i][j][s][t]) % MOD

                        # 2. 延续插头
                        if left != 0 and up == 0:
                            # 延续左插头到上方
                            new_state = s
                            new_state &= ~(3 << (2 * (j - 1)))
                            new_state |= (left << (2 * j))
                            dp[i][j+1][new_state][t] = (dp[i][j+1][new_state][t] + dp[i][j][s][t]) % MOD

                        if left == 0 and up != 0:
                            # 延续上插头到左方
                            new_state = s
                            new_state &= ~(3 << (2 * j))
                            new_state |= (up << (2 * (j - 1)))
                            dp[i][j+1][new_state][t] = (dp[i][j+1][new_state][t] + dp[i][j][s][t]) % MOD

                        # 3. 创建新插头对（如果左右和上方都没有插头）
                        if left == 0 and up == 0:
                            # 创建一对新插头（左插头和上插头）
                            new_state = s | (1 << (2 * (j - 1))) | (1 << (2 * j))
                            dp[i][j+1][new_state][t] = (dp[i][j+1][new_state][t] + dp[i][j][s][t]) % MOD

    # 统计形成恰好 K 个回路的方案数
    result = 0
```

```
    for s in range(1 << (2 * m)):
        result = (result + dp[n][0][s][K]) % MOD
    return result

def renumber(state, pos1, pos2, id1, id2):
    """
    重新编号以保持最小表示法

    Args:
        state: 当前状态
        pos1: 位置 1
        pos2: 位置 2
        id1: 编号 1
        id2: 编号 2

    Returns:
        重新编号后的状态
    """
    # 合并两个连通分量，将 id2 的编号改为 id1
    min_id = min(id1, id2)
    max_id = max(id1, id2)

    m_val = (state >> (2 * pos1)) & 3
    if m_val == max_id:
        state &= ~(3 << (2 * pos1))
        state |= (min_id << (2 * pos1))

    m_val = (state >> (2 * pos2)) & 3
    if m_val == max_id:
        state &= ~(3 << (2 * pos2))
        state |= (min_id << (2 * pos2))

    return state

# 测试用例
if __name__ == "__main__":
    maze1 = [
        [0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]
    ]
    print(solve(3, 3, 2, maze1))  # 输出形成 2 个回路的方案数
```

===

文件: POJ1739_TonysTour.cpp
===
```
// POJ 1739 Tony's Tour (插头DP - 简单路径)
// 在n×m的网格中，求从左下角到右下角的简单路径数，必须经过所有非障碍格子
// 1 <= n, m <= 8
// 测试链接 : http://poj.org/problem?id=1739
//
// 题目大意：
// 给定一个n×m的网格，其中一些格子是障碍物（用'#'表示），其他格子是可通行的（用'.'表示）。
// 要求找到一条从左下角到右下角的简单路径，路径必须经过所有可通行的格子恰好一次。
// 求满足条件的路径数。
//
// 解题思路：
// 使用插头DP解决简单路径问题。
// 状态表示：用三进制表示轮廓线状态，0表示无插头，1表示左插头，2表示右插头。
// 特殊处理：起点和终点需要特殊处理。
//
// Java实现：https://github.com/yourusername/algorithm-journey/blob/main/src/class126/POJ1739_TonysTour.java
// C++实现：https://github.com/yourusername/algorithm-journey/blob/main/src/class126/POJ1739_TonysTour.cpp
// Python实现：https://github.com/yourusername/algorithm-journey/blob/main/src/class126/POJ1739_TonysTour.py

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

using namespace std;

const int MAXN = 10;
const int MAX_STATES = 6561; // 3^8

// dp[i][j][s][c]表示处理到第i行第j列，轮廓线状态为s，当前点颜色为c的方案数
// 状态s用三进制表示，0表示无插头，1表示左插头，2表示右插头
long long dp[MAXN][MAXN][MAX_STATES][3];

char grid[MAXN][MAXN];
int n, m;
```

```
/**
 * 计算从左下角到右下角经过所有非障碍格子的简单路径数
 *
 * 算法思路：
 * 使用插头 DP 解决简单路径问题
 * 状态表示：用三进制表示轮廓线状态，0 表示无插头，1 表示左插头，2 表示右插头
 * 特殊处理：起点和终点需要特殊处理
 *
 * 时间复杂度：O(n * m * 3^m)
 * 空间复杂度：O(n * m * 3^m)
 *
 * @param rows 行数
 * @param cols 列数
 * @param maze 网格地图，'#'表示障碍，'.'表示可通行
 * @return 路径数
 */
long long solve(int rows, int cols, vector<vector<char>>& maze) {
    n = rows;
    m = cols;

    // 复制网格
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            grid[i][j] = maze[i][j];
        }
    }

    // 初始化 DP 数组
    memset(dp, 0, sizeof(dp));

    // 起点在左下角 (n-1, 0)
    // 终点在右下角 (n-1, m-1)

    // 初始状态：在起点处创建一个插头
    dp[n-1][0][1][1] = 1; // 在起点创建一个左插头

    // 逐格 DP
    for (int i = n-1; i >= 0; i--) {
        // 行间转移
        if (i < n-1) {
            for (int s = 0; s < power(3, m); s++) {
                for (int c = 0; c < 3; c++) {
                    if (dp[i+1][m][s][c] > 0) {
```

```
                // 将状态转移到下一行的开始
                int newState = s * 3;
                dp[i][0][newState][0] += dp[i+1][m][s][c];
            }
        }
    }
}

// 行内转移
for (int j = 0; j < m; j++) {
    for (int s = 0; s < power(3, m); s++) {
        for (int c = 0; c < 3; c++) {
            if (dp[i][j][s][c] == 0) continue;

            // 获取当前格子左边和上面的插头状态
            int left = j > 0 ? get(s, j-1) : 0;
            int up = get(s, j);

            // 如果是障碍格子
            if (grid[i][j] == '#') {
                // 只能在没有插头的情况下转移
                if (left == 0 && up == 0) {
                    int newState = set(s, j, 0);
                    dp[i][j+1][newState][0] += dp[i][j][s][c];
                }
            } else {
                // 可通行格子

                // 1. 不放置插头（合并两个插头）
                if (left != 0 && up != 0) {
                    int newState = set(set(s, j-1, 0), j, 0);
                    dp[i][j+1][newState][0] += dp[i][j][s][c];
                }

                // 2. 延续插头
                if (left != 0 && up == 0) {
                    // 延续左插头到上方
                    int newState = set(set(s, j-1, 0), j, left);
                    dp[i][j+1][newState][left] += dp[i][j][s][c];
                }

                if (left == 0 && up != 0) {
                    // 延续上插头到左方
```

```
                        int newState = set(set(s, j, 0), j-1, up);
                        dp[i][j+1][newState][up] += dp[i][j][s][c];
                    }

                    // 3. 创建新插头（如果左右和上方都没有插头）
                    if (left == 0 && up == 0) {
                        // 创建左插头
                        int newState = set(s, j-1, 1);
                        dp[i][j+1][newState][1] += dp[i][j][s][c];

                        // 创建上插头
                        newState = set(s, j, 2);
                        dp[i][j+1][newState][2] += dp[i][j][s][c];
                    }
                }
            }
        }
    }
}

    // 终点处应该没有插头
    return dp[0][m][0][0];
}

// 计算 base^exp
int power(int base, int exp) {
    int result = 1;
    for (int i = 0; i < exp; i++) {
        result *= base;
    }
    return result;
}

// 获取状态 s 中第 j 个位置的值
int get(int s, int j) {
    return (s / power(3, j)) % 3;
}

// 设置状态 s 中第 j 个位置的值为 v
int set(int s, int j, int v) {
    int pow = power(3, j);
    return (s / pow / 3) * pow * 3 + v * pow + (s % pow);
}
```

```
// 测试用例
int main() {
    vector<vector<char>> maze1 = {
        {'.', '.', '.'},
        {'.', '.', '.'},
        {'.', '.', '.'}
    };
    cout << solve(3, 3, maze1) << endl; // 输出路径数

    vector<vector<char>> maze2 = {
        {'.', '.', '#'},
        {'.', '.', '.'},
        {'.', '.', '.'}
    };
    cout << solve(3, 3, maze2) << endl; // 输出路径数

    return 0;
}
```

==

文件: POJ1739_TonysTour.java

==

```
package class126;

// POJ 1739 Tony's Tour (插头 DP - 简单路径)
// 在 n×m 的网格中，求从左下角到右下角的简单路径数，必须经过所有非障碍格子
// 1 <= n, m <= 8
// 测试链接 : http://poj.org/problem?id=1739
//
// 题目大意：
// 给定一个 n×m 的网格，其中一些格子是障碍物（用'#'表示），其他格子是可通行的（用'.'表示）。
// 要求找到一条从左下角到右下角的简单路径，路径必须经过所有可通行的格子恰好一次。
// 求满足条件的路径数。
//
// 解题思路：
// 使用插头 DP 解决简单路径问题。
// 状态表示：用三进制表示轮廓线状态，0 表示无插头，1 表示左插头，2 表示右插头。
// 特殊处理：起点和终点需要特殊处理。
//
// Java 实现：https://github.com/yourusername/algorithm-journey/blob/main/src/class126/POJ1739_TonysTour.java
```

```java
// C++实现：https://github.com/yourusername/algorithm-
journey/blob/main/src/class126/POJ1739_TonysTour.cpp
// Python 实现：https://github.com/yourusername/algorithm-
journey/blob/main/src/class126/POJ1739_TonysTour.py

public class POJ1739_TonysTour {

    public static int MAXN = 10;
    public static int MAX_STATES = 6561; // 3^8

    // dp[i][j][s][c]表示处理到第 i 行第 j 列，轮廓线状态为 s，当前点颜色为 c 的方案数
    // 状态 s 用三进制表示，0 表示无插头，1 表示左插头，2 表示右插头
    public static long[][][][] dp = new long[MAXN][MAXN][MAX_STATES][3];

    public static char[][] grid = new char[MAXN][MAXN];
    public static int n, m;

    /**
     * 计算从左下角到右下角经过所有非障碍格子的简单路径数
     *
     * 算法思路：
     * 使用插头 DP 解决简单路径问题
     * 状态表示：用三进制表示轮廓线状态，0 表示无插头，1 表示左插头，2 表示右插头
     * 特殊处理：起点和终点需要特殊处理
     *
     * 时间复杂度：O(n * m * 3^m)
     * 空间复杂度：O(n * m * 3^m)
     *
     * @param rows 行数
     * @param cols 列数
     * @param maze 网格地图，'#'表示障碍，'.'表示可通行
     * @return 路径数
     */
    public static long solve(int rows, int cols, char[][] maze) {
        n = rows;
        m = cols;

        // 复制网格
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                grid[i][j] = maze[i][j];
            }
        }
```

```
// 初始化 DP 数组
for (int i = 0; i <= n; i++) {
    for (int j = 0; j <= m; j++) {
        for (int s = 0; s < MAX_STATES; s++) {
            for (int c = 0; c < 3; c++) {
                dp[i][j][s][c] = 0;
            }
        }
    }
}

// 起点在左下角 (n-1, 0)
// 终点在右下角 (n-1, m-1)

// 初始状态：在起点处创建一个插头
dp[n-1][0][1][1] = 1; // 在起点创建一个左插头

// 逐格 DP
for (int i = n-1; i >= 0; i--) {
    // 行间转移
    if (i < n-1) {
        for (int s = 0; s < power(3, m); s++) {
            for (int c = 0; c < 3; c++) {
                if (dp[i+1][m][s][c] > 0) {
                    // 将状态转移到下一行的开始
                    int newState = s * 3;
                    dp[i][0][newState][0] += dp[i+1][m][s][c];
                }
            }
        }
    }

    // 行内转移
    for (int j = 0; j < m; j++) {
        for (int s = 0; s < power(3, m); s++) {
            for (int c = 0; c < 3; c++) {
                if (dp[i][j][s][c] == 0) continue;

                // 获取当前格子左边和上面的插头状态
                int left = j > 0 ? get(s, j-1) : 0;
                int up = get(s, j);
```

```
            // 如果是障碍格子
            if (grid[i][j] == '#') {
                // 只能在没有插头的情况下转移
                if (left == 0 && up == 0) {
                    int newState = set(s, j, 0);
                    dp[i][j+1][newState][0] += dp[i][j][s][c];
                }
            } else {
                // 可通行格子

                // 1. 不放置插头（合并两个插头）
                if (left != 0 && up != 0) {
                    int newState = set(set(s, j-1, 0), j, 0);
                    dp[i][j+1][newState][0] += dp[i][j][s][c];
                }

                // 2. 延续插头
                if (left != 0 && up == 0) {
                    // 延续左插头到上方
                    int newState = set(set(s, j-1, 0), j, left);
                    dp[i][j+1][newState][left] += dp[i][j][s][c];
                }

                if (left == 0 && up != 0) {
                    // 延续上插头到左方
                    int newState = set(set(s, j, 0), j-1, up);
                    dp[i][j+1][newState][up] += dp[i][j][s][c];
                }

                // 3. 创建新插头（如果左右和上方都没有插头）
                if (left == 0 && up == 0) {
                    // 创建左插头
                    int newState = set(s, j-1, 1);
                    dp[i][j+1][newState][1] += dp[i][j][s][c];

                    // 创建上插头
                    newState = set(s, j, 2);
                    dp[i][j+1][newState][2] += dp[i][j][s][c];
                }
            }
        }
    }
}
```

```java
        }

        // 终点处应该没有插头
        return dp[0][m][0][0];
    }

    // 计算 base^exp
    public static int power(int base, int exp) {
        int result = 1;
        for (int i = 0; i < exp; i++) {
            result *= base;
        }
        return result;
    }

    // 获取状态 s 中第 j 个位置的值
    public static int get(int s, int j) {
        return (s / power(3, j)) % 3;
    }

    // 设置状态 s 中第 j 个位置的值为 v
    public static int set(int s, int j, int v) {
        int pow = power(3, j);
        return (s / pow / 3) * pow * 3 + v * pow + (s % pow);
    }

    // 测试用例
    public static void main(String[] args) {
        char[][] maze1 = {
            {'.', '.', '.'},
            {'.', '.', '.'},
            {'.', '.', '.'}
        };
        System.out.println(solve(3, 3, maze1)); // 输出路径数

        char[][] maze2 = {
            {'.', '.', '#'},
            {'.', '.', '.'},
            {'.', '.', '.'}
        };
        System.out.println(solve(3, 3, maze2)); // 输出路径数
    }
}
```

==

文件: POJ1739_TonysTour.py
==
```
# POJ 1739 Tony's Tour (插头DP - 简单路径)
# 在n×m的网格中，求从左下角到右下角的简单路径数，必须经过所有非障碍格子
# 1 <= n, m <= 8
# 测试链接 : http://poj.org/problem?id=1739
#
# 题目大意:
# 给定一个n×m的网格，其中一些格子是障碍物（用'#'表示），其他格子是可通行的（用'.'表示）。
# 要求找到一条从左下角到右下角的简单路径，路径必须经过所有可通行的格子恰好一次。
# 求满足条件的路径数。
#
# 解题思路:
# 使用插头DP解决简单路径问题。
# 状态表示：用三进制表示轮廓线状态，0表示无插头，1表示左插头，2表示右插头。
# 特殊处理：起点和终点需要特殊处理。
#
# Java实现：https://github.com/yourusername/algorithm-
journey/blob/main/src/class126/POJ1739_TonysTour.java
# C++实现：https://github.com/yourusername/algorithm-
journey/blob/main/src/class126/POJ1739_TonysTour.cpp
# Python实现：https://github.com/yourusername/algorithm-
journey/blob/main/src/class126/POJ1739_TonysTour.py

MAXN = 10
MAX_STATES = 6561  # 3^8

# dp[i][j][s][c]表示处理到第i行第j列，轮廓线状态为s，当前点颜色为c的方案数
# 状态s用三进制表示，0表示无插头，1表示左插头，2表示右插头
dp = [[[[0] * 3 for _ in range(MAX_STATES)] for _ in range(MAXN)] for _ in range(MAXN)]

grid = [[''] * MAXN for _ in range(MAXN)]
n, m = 0, 0

def solve(rows, cols, maze):
    """
    计算从左下角到右下角经过所有非障碍格子的简单路径数

    算法思路:
    使用插头DP解决简单路径问题
```

```
状态表示：用三进制表示轮廓线状态，0 表示无插头，1 表示左插头，2 表示右插头
特殊处理：起点和终点需要特殊处理

时间复杂度：O(n * m * 3^m)
空间复杂度：O(n * m * 3^m)

Args:
    rows: 行数
    cols: 列数
    maze: 网格地图，'#'表示障碍，'.'表示可通行

Returns:
    路径数
"""
global n, m

n = rows
m = cols

# 复制网格
for i in range(n):
    for j in range(m):
        grid[i][j] = maze[i][j]

# 初始化 DP 数组
for i in range(MAXN):
    for j in range(MAXN):
        for s in range(MAX_STATES):
            for c in range(3):
                dp[i][j][s][c] = 0

# 起点在左下角 (n-1, 0)
# 终点在右下角 (n-1, m-1)

# 初始状态：在起点处创建一个插头
dp[n-1][0][1][1] = 1  # 在起点创建一个左插头

# 逐格 DP
for i in range(n-1, -1, -1):
    # 行间转移
    if i < n-1:
        for s in range(power(3, m)):
            for c in range(3):
```

```
            if dp[i+1][m][s][c] > 0:
                # 将状态转移到下一行的开始
                new_state = s * 3
                dp[i][0][new_state][0] += dp[i+1][m][s][c]

# 行内转移
for j in range(m):
    for s in range(power(3, m)):
        for c in range(3):
            if dp[i][j][s][c] == 0:
                continue

            # 获取当前格子左边和上面的插头状态
            left = get(s, j-1) if j > 0 else 0
            up = get(s, j)

            # 如果是障碍格子
            if grid[i][j] == '#':
                # 只能在没有插头的情况下转移
                if left == 0 and up == 0:
                    new_state = set_state(s, j, 0)
                    dp[i][j+1][new_state][0] += dp[i][j][s][c]
            else:
                # 可通行格子

                # 1. 不放置插头（合并两个插头）
                if left != 0 and up != 0:
                    new_state = set_state(set_state(s, j-1, 0), j, 0)
                    dp[i][j+1][new_state][0] += dp[i][j][s][c]

                # 2. 延续插头
                if left != 0 and up == 0:
                    # 延续左插头到上方
                    new_state = set_state(set_state(s, j-1, 0), j, left)
                    dp[i][j+1][new_state][left] += dp[i][j][s][c]

                if left == 0 and up != 0:
                    # 延续上插头到左方
                    new_state = set_state(set_state(s, j, 0), j-1, up)
                    dp[i][j+1][new_state][up] += dp[i][j][s][c]

                # 3. 创建新插头（如果左右和上方都没有插头）
                if left == 0 and up == 0:
```

```
                        # 创建左插头
                        new_state = set_state(s, j-1, 1)
                        dp[i][j+1][new_state][1] += dp[i][j][s][c]

                        # 创建上插头
                        new_state = set_state(s, j, 2)
                        dp[i][j+1][new_state][2] += dp[i][j][s][c]

    # 终点处应该没有插头
    return dp[0][m][0][0]

def power(base, exp):
    """计算 base^exp"""
    result = 1
    for _ in range(exp):
        result *= base
    return result

def get(s, j):
    """获取状态 s 中第 j 个位置的值"""
    return (s // power(3, j)) % 3

def set_state(s, j, v):
    """设置状态 s 中第 j 个位置的值为 v"""
    pow_val = power(3, j)
    return (s // pow_val // 3) * pow_val * 3 + v * pow_val + (s % pow_val)

# 测试用例
if __name__ == "__main__":
    maze1 = [
        ['.', '.', '.'],
        ['.', '.', '.'],
        ['.', '.', '.']
    ]
    print(solve(3, 3, maze1))  # 输出路径数

    maze2 = [
        ['.', '.', '#'],
        ['.', '.', '.'],
        ['.', '.', '.']
    ]
    print(solve(3, 3, maze2))  # 输出路径数
```

==

文件: POJ2411_MondriaanDream.cpp
==
```
// POJ 2411 Mondriaan's Dream (轮廓线DP)
// 用1×2和2×1的多米诺骨牌铺满n×m的棋盘，求方案数
// 1 <= n, m <= 11
// 测试链接 : http://poj.org/problem?id=2411
//
// 题目大意：
// 给定一个n×m的棋盘，要求用1×2和2×1的多米诺骨牌完全覆盖棋盘（即骨牌不能重叠，也不能有空隙）。
// 求有多少种不同的覆盖方案。
//
// 解题思路：
// 使用轮廓线DP，逐格递推。
// 状态表示：用二进制数表示轮廓线状态，第k位为1表示第k个位置被上一行的竖直骨牌占据。
// 状态转移：
// 1. 当前位置已被上一行竖直骨牌占据，则当前位置不能再放置骨牌
// 2. 当前位置未被占据，则可以：
//    a. 放置竖直骨牌（当前位置和下一行同一位置）
//    b. 放置水平骨牌（当前位置和右边位置，前提是右边位置存在且未被占据）
//
// Java实现：https://github.com/yourusername/algorithm-journey/blob/main/src/class126/POJ2411_MondriaanDream.java
// C++实现：https://github.com/yourusername/algorithm-journey/blob/main/src/class126/POJ2411_MondriaanDream.cpp
// Python实现：https://github.com/yourusername/algorithm-journey/blob/main/src/class126/POJ2411_MondriaanDream.py

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

using namespace std;

const int MAXN = 12;
const int MAXM = 12;

// dp[i][j][s]表示处理到第i行第j列，轮廓线状态为s的方案数
// 状态s用二进制表示，第k位为1表示第k个位置被上一行的竖直骨牌占据
```

```
long long dp[MAXN][MAXM][1 << MAXM];

int n, m, maxs;

// 前向声明DFS函数
long long dfs(int i, int j, int s);

/**
 * 计算用1×2和2×1的多米诺骨牌铺满n×m棋盘的方案数
 *
 * 算法思路:
 * 使用轮廓线DP,逐格递推
 * 状态表示:用二进制数表示轮廓线状态,第k位为1表示第k个位置被上一行的竖直骨牌占据
 * 状态转移:
 * 1. 当前位置已被上一行竖直骨牌占据,则当前位置不能再放置骨牌
 * 2. 当前位置未被占据,则可以:
 *    a. 放置竖直骨牌(当前位置和下一行同一位置)
 *    b. 放置水平骨牌(当前位置和右边位置,前提是右边位置存在且未被占据)
 *
 * 时间复杂度:O(n * m * 2^m)
 * 空间复杂度:O(n * m * 2^m)
 *
 * @param rows 行数
 * @param cols 列数
 * @return 铺满棋盘的方案数
 */
long long solve(int rows, int cols) {
    // 为了优化,让较小的一维作为列数
    n = max(rows, cols);
    m = min(rows, cols);
    maxs = 1 << m;

    // 初始化DP数组
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            for (int s = 0; s < maxs; s++) {
                dp[i][j][s] = -1;
            }
        }
    }

    return dfs(0, 0, 0);
}
```

```
/**
 * DFS 记忆化搜索
 *
 * @param i 当前行
 * @param j 当前列
 * @param s 轮廓线状态
 * @return 从当前状态开始的方案数
 */
long long dfs(int i, int j, int s) {
    // 处理完所有行
    if (i == n) {
        return 1;
    }

    // 处理完当前行，转到下一行
    if (j == m) {
        return dfs(i + 1, 0, s);
    }

    // 记忆化
    if (dp[i][j][s] != -1) {
        return dp[i][j][s];
    }

    long long ans = 0;

    // 检查当前位置是否已被上一行的竖直骨牌占据
    if (((s >> j) & 1) == 1) {
        // 已被占据，当前位置不能再放骨牌
        ans = dfs(i, j + 1, s & (~(1 << j)));
    } else {
        // 未被占据，可以放置骨牌

        // 放置竖直骨牌（当前位置和下一行同一位置）
        if (i + 1 < n) {
            ans += dfs(i, j + 1, s | (1 << j));
        }

        // 放置水平骨牌（当前位置和右边位置）
        if (j + 1 < m && ((s >> (j + 1)) & 1) == 0) {
            ans += dfs(i, j + 2, s);
        }
```

```
    }

    dp[i][j][s] = ans;
    return ans;
}

// 测试用例
int main() {
    // 示例：2×3 的棋盘有 3 种铺法
    std::cout << solve(2, 3) << std::endl; // 输出: 3

    // 示例：4×4 的棋盘有 5 种铺法
    std::cout << solve(4, 4) << std::endl; // 输出: 5

    return 0;
}
```

==

文件: POJ2411_MondriaanDream.java

==

```
package class126;

// POJ 2411 Mondriaan's Dream (轮廓线 DP)
// 用 1×2 和 2×1 的多米诺骨牌铺满 n×m 的棋盘，求方案数
// 1 <= n, m <= 11
// 测试链接 : http://poj.org/problem?id=2411
//
// 题目大意：
// 给定一个 n×m 的棋盘，要求用 1×2 和 2×1 的多米诺骨牌完全覆盖棋盘（即骨牌不能重叠，也不能有空隙）。
// 求有多少种不同的覆盖方案。
//
// 解题思路：
// 使用轮廓线 DP，逐格递推。
// 状态表示：用二进制数表示轮廓线状态，第 k 位为 1 表示第 k 个位置被上一行的竖直骨牌占据。
// 状态转移：
// 1. 当前位置已被上一行竖直骨牌占据，则当前位置不能再放置骨牌
// 2. 当前位置未被占据，则可以：
//     a. 放置竖直骨牌（当前位置和下一行同一位置）
//     b. 放置水平骨牌（当前位置和右边位置，前提是右边位置存在且未被占据）
//
// Java 实现：https://github.com/yourusername/algorithm-
```

```
journey/blob/main/src/class126/POJ2411_MondriaanDream.java
// C++实现：https://github.com/yourusername/algorithm-
journey/blob/main/src/class126/POJ2411_MondriaanDream.cpp
// Python 实现：https://github.com/yourusername/algorithm-
journey/blob/main/src/class126/POJ2411_MondriaanDream.py

public class POJ2411_MondriaanDream {

    public static int MAXN = 12;

    // dp[i][j][s]表示处理到第 i 行第 j 列，轮廓线状态为 s 的方案数
    // 状态 s 用二进制表示，第 k 位为 1 表示第 k 个位置被上一行的竖直骨牌占据
    public static long[][][] dp = new long[MAXN][MAXN][1 << MAXN];

    public static int n, m, maxs;

    /**
     * 计算用 1×2 和 2×1 的多米诺骨牌铺满 n×m 棋盘的方案数
     *
     * 算法思路：
     * 使用轮廓线 DP，逐格递推
     * 状态表示：用二进制数表示轮廓线状态，第 k 位为 1 表示第 k 个位置被上一行的竖直骨牌占据
     * 状态转移：
     * 1. 当前位置已被上一行竖直骨牌占据，则当前位置不能再放置骨牌
     * 2. 当前位置未被占据，则可以：
     *    a. 放置竖直骨牌（当前位置和下一行同一位置）
     *    b. 放置水平骨牌（当前位置和右边位置，前提是右边位置存在且未被占据）
     *
     * 时间复杂度：O(n * m * 2^m)
     * 空间复杂度：O(n * m * 2^m)
     *
     * @param rows 行数
     * @param cols 列数
     * @return 铺满棋盘的方案数
     */
    public static long solve(int rows, int cols) {
        // 为了优化，让较小的一维作为列数
        n = Math.max(rows, cols);
        m = Math.min(rows, cols);
        maxs = 1 << m;

        // 初始化 DP 数组
        for (int i = 0; i < n; i++) {
```

```java
        for (int j = 0; j < m; j++) {
            for (int s = 0; s < maxs; s++) {
                dp[i][j][s] = -1;
            }
        }
    }

    return dfs(0, 0, 0);
}

/**
 * DFS 记忆化搜索
 *
 * @param i 当前行
 * @param j 当前列
 * @param s 轮廓线状态
 * @return 从当前状态开始的方案数
 */
public static long dfs(int i, int j, int s) {
    // 处理完所有行
    if (i == n) {
        return 1;
    }

    // 处理完当前行，转到下一行
    if (j == m) {
        return dfs(i + 1, 0, s);
    }

    // 记忆化
    if (dp[i][j][s] != -1) {
        return dp[i][j][s];
    }

    long ans = 0;

    // 检查当前位置是否已被上一行的竖直骨牌占据
    if (((s >> j) & 1) == 1) {
        // 已被占据，当前位置不能再放骨牌
        ans = dfs(i, j + 1, s & (~(1 << j)));
    } else {
        // 未被占据，可以放置骨牌
```

```java
            // 放置竖直骨牌（当前位置和下一行同一位置）
            if (i + 1 < n) {
                ans += dfs(i, j + 1, s | (1 << j));
            }

            // 放置水平骨牌（当前位置和右边位置）
            if (j + 1 < m && ((s >> (j + 1)) & 1) == 0) {
                ans += dfs(i, j + 2, s);
            }
        }

        dp[i][j][s] = ans;
        return ans;
    }

    // 测试用例
    public static void main(String[] args) {
        // 示例：2×3 的棋盘有 3 种铺法
        System.out.println(solve(2, 3)); // 输出: 3

        // 示例：4×4 的棋盘有 5 种铺法
        System.out.println(solve(4, 4)); // 输出: 5
    }
}
```

==

文件: POJ2411_MondriaanDream.py

==

```python
# POJ 2411 Mondriaan's Dream (轮廓线 DP)
# 用 1×2 和 2×1 的多米诺骨牌铺满 n×m 的棋盘，求方案数
# 1 <= n, m <= 11
# 测试链接 : http://poj.org/problem?id=2411
#
# 题目大意：
# 给定一个 n×m 的棋盘，要求用 1×2 和 2×1 的多米诺骨牌完全覆盖棋盘（即骨牌不能重叠，也不能有空隙）。
# 求有多少种不同的覆盖方案。
#
# 解题思路：
# 使用轮廓线 DP，逐格递推。
# 状态表示：用二进制数表示轮廓线状态，第 k 位为 1 表示第 k 个位置被上一行的竖直骨牌占据。
# 状态转移：
```

```python
# 1. 当前位置已被上一行竖直骨牌占据，则当前位置不能再放置骨牌
# 2. 当前位置未被占据，则可以：
#    a. 放置竖直骨牌（当前位置和下一行同一位置）
#    b. 放置水平骨牌（当前位置和右边位置，前提是右边位置存在且未被占据）
#
# Java 实现：https://github.com/yourusername/algorithm-journey/blob/main/src/class126/POJ2411_MondriaanDream.java
# C++实现：https://github.com/yourusername/algorithm-journey/blob/main/src/class126/POJ2411_MondriaanDream.cpp
# Python 实现：https://github.com/yourusername/algorithm-journey/blob/main/src/class126/POJ2411_MondriaanDream.py

import sys

MAXN = 12
MAXM = 12

# dp[i][j][s]表示处理到第 i 行第 j 列，轮廓线状态为 s 的方案数
# 状态 s 用二进制表示，第 k 位为 1 表示第 k 个位置被上一行的竖直骨牌占据
dp = [[[-1] * (1 << MAXM) for _ in range(MAXM)] for _ in range(MAXN)]

n, m, maxs = 0, 0, 0

def solve(rows, cols):
    """
    计算用 1×2 和 2×1 的多米诺骨牌铺满 n×m 棋盘的方案数

    算法思路：
    使用轮廓线 DP，逐格递推
    状态表示：用二进制数表示轮廓线状态，第 k 位为 1 表示第 k 个位置被上一行的竖直骨牌占据
    状态转移：
    1. 当前位置已被上一行竖直骨牌占据，则当前位置不能再放置骨牌
    2. 当前位置未被占据，则可以：
       a. 放置竖直骨牌（当前位置和下一行同一位置）
       b. 放置水平骨牌（当前位置和右边位置，前提是右边位置存在且未被占据）

    时间复杂度：O(n * m * 2^m)
    空间复杂度：O(n * m * 2^m)

    Args:
        rows: 行数
        cols: 列数
```

```
    Returns:
        铺满棋盘的方案数
    """
    global n, m, maxs

    # 为了优化，让较小的一维作为列数
    n = max(rows, cols)
    m = min(rows, cols)
    maxs = 1 << m

    # 初始化 DP 数组
    for i in range(n):
        for j in range(m):
            for s in range(maxs):
                dp[i][j][s] = -1

    return dfs(0, 0, 0)

def dfs(i, j, s):
    """
    DFS 记忆化搜索

    Args:
        i: 当前行
        j: 当前列
        s: 轮廓线状态

    Returns:
        从当前状态开始的方案数
    """
    # 处理完所有行
    if i == n:
        return 1

    # 处理完当前行，转到下一行
    if j == m:
        return dfs(i + 1, 0, s)

    # 记忆化
    if dp[i][j][s] != -1:
        return dp[i][j][s]

    ans = 0
```

```
    # 检查当前位置是否已被上一行的竖直骨牌占据
    if (s >> j) & 1:
        # 已被占据，当前位置不能再放骨牌
        ans = dfs(i, j + 1, s & (~(1 << j)))
    else:
        # 未被占据，可以放置骨牌

        # 放置竖直骨牌（当前位置和下一行同一位置）
        if i + 1 < n:
            ans += dfs(i, j + 1, s | (1 << j))

        # 放置水平骨牌（当前位置和右边位置）
        if j + 1 < m and not ((s >> (j + 1)) & 1):
            ans += dfs(i, j + 2, s)

    dp[i][j][s] = ans
    return ans

# 测试用例
if __name__ == "__main__":
    # 示例：2×3 的棋盘有 3 种铺法
    print(solve(2, 3))  # 输出: 3

    # 示例：4×4 的棋盘有 5 种铺法
    print(solve(4, 4))  # 输出: 5
```

===

文件: URAL1519_Formula1.java

===

```
package class126;

// URAL 1519 Formula 1 (插头 DP - 哈密顿回路)
// 在 n×m 的网格中，求经过所有非障碍格子的哈密顿回路数
// 1 <= n, m <= 12
// 测试链接 : https://vjudge.net/problem/URAL-1519
//
// 题目大意：
// 给定一个 n×m 的网格，其中一些格子是障碍物（用'*'表示），其他格子是可通行的（用'.'表示）。
// 要求找到一条哈密顿回路，即经过所有可通行格子恰好一次的闭合路径。
// 求满足条件的哈密顿回路数。
//
```

```
// 解题思路：
// 使用插头 DP 解决哈密顿回路问题。
// 状态表示：用括号表示法表示轮廓线上的连通性状态。
// 状态转移：根据插头的连通性进行合并、创建等操作。
// 使用哈希表优化状态存储。
//
// Java 实现：https://github.com/yourusername/algorithm-journey/blob/main/src/class126/URAL1519_Formula1.java
// C++实现：https://github.com/yourusername/algorithm-journey/blob/main/src/class126/URAL1519_Formula1.cpp
// Python 实现：https://github.com/yourusername/algorithm-journey/blob/main/src/class126/URAL1519_Formula1.py

public class URAL1519_Formula1 {

    public static int MAXN = 15;
    public static int MAX_STATES = 300000; // 足够存储所有状态

    // 使用哈希表存储状态，因为状态数太多无法直接用数组
    public static class HashTable {
        public int[] head = new int[MAX_STATES];
        public int[] next = new int[MAX_STATES];
        public int[] state = new int[MAX_STATES];
        public long[] value = new long[MAX_STATES];
        public int size;

        public void init() {
            for (int i = 0; i < size; i++) {
                head[i] = -1;
            }
            size = 1;
        }

        public int getHash(int st) {
            return st % MAX_STATES;
        }

        public int find(int st) {
            int h = getHash(st);
            for (int i = head[h]; i != -1; i = next[i]) {
                if (state[i] == st) {
                    return i;
                }
```

```java
        }
        return -1;
    }

    public int insert(int st, long val) {
        int h = getHash(st);
        int pos = find(st);
        if (pos != -1) {
            value[pos] += val;
            return pos;
        }
        state[size] = st;
        value[size] = val;
        next[size] = head[h];
        head[h] = size++;
        return size - 1;
    }
}

public static HashTable[] dp = new HashTable[2];
public static char[][] grid = new char[MAXN][MAXN];
public static int n, m;

/**
 * 计算经过所有非障碍格子的哈密顿回路数
 *
 * 算法思路：
 * 使用插头 DP 解决哈密顿回路问题
 * 状态表示：用括号表示法表示轮廓线上的连通性状态
 * 状态转移：根据插头的连通性进行合并、创建等操作
 * 使用哈希表优化状态存储
 *
 * 时间复杂度：O(n * m * 状态数)
 * 空间复杂度：O(状态数)
 *
 * @param rows 行数
 * @param cols 列数
 * @param maze 网格，'.'表示可经过，'*'表示障碍
 * @return 哈密顿回路数
 */
public static long solve(int rows, int cols, char[][] maze) {
    n = rows;
    m = cols;
```

```
// 复制网格
for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
        grid[i][j] = maze[i][j];
    }
}

// 初始化哈希表
dp[0] = new HashTable();
dp[1] = new HashTable();

// 初始状态
dp[0].init();
dp[1].init();
dp[0].insert(0, 1);

int cur = 0;

// 逐格 DP
for (int i = 0; i < n; i++) {
    // 行间转移
    for (int j = 0; j < dp[cur].size; j++) {
        if (dp[cur].value[j] > 0) {
            // 将状态转移到下一行的开始
            dp[1-cur].insert(dp[cur].state[j] << 2, dp[cur].value[j]);
        }
    }

    cur = 1 - cur;
    dp[1-cur].init();

    // 行内转移
    for (int j = 0; j < m; j++) {
        for (int k = 0; k < dp[cur].size; k++) {
            int state = dp[cur].state[k];
            long value = dp[cur].value[k];
            if (value == 0) continue;

            // 获取当前格子左边和上面的插头状态
            int left = j > 0 ? ((state >> (2 * (j - 1))) & 3) : 0;
            int up = (state >> (2 * j)) & 3;
```

```
// 如果是障碍格子
if (grid[i][j] == '*') {
    // 只能在没有插头的情况下转移
    if (left == 0 && up == 0) {
        int newState = state & (~((3 << (2 * (j - 1))) | (3 << (2 * j))));
        dp[1-cur].insert(newState, value);
    }
} else {
    // 可通行格子

    // 1. 不放置插头（合并两个插头）
    if (left != 0 && up != 0) {
        int newState = state;
        newState &= ~((3 << (2 * (j - 1))) | (3 << (2 * j)));

        // 如果两个插头属于不同连通分量，则合并
        // 如果两个插头属于相同连通分量，则形成哈密顿回路
        if (left == up) {
            // 检查是否所有格子都已访问
            if (newState == 0 && i == n-1 && j == m-1) {
                dp[1-cur].insert(newState, value);
            }
        } else {
            // 合并连通分量
            // 需要重新编号保持括号表示法
            newState = renumber(newState, j-1, j, left, up);
            dp[1-cur].insert(newState, value);
        }
    }

    // 2. 延续插头
    if (left != 0 && up == 0) {
        // 延续左插头到上方
        int newState = state;
        newState &= ~(3 << (2 * (j - 1)));
        newState |= (left << (2 * j));
        dp[1-cur].insert(newState, value);
    }

    if (left == 0 && up != 0) {
        // 延续上插头到左方
        int newState = state;
        newState &= ~(3 << (2 * j));
```

```
                    newState |= (up << (2 * (j - 1)));
                    dp[1-cur].insert(newState, value);
                }

                // 3. 创建新插头对（如果左右和上方都没有插头）
                if (left == 0 && up == 0) {
                    // 创建一对新插头（左插头和上插头）
                    int newState = state | (1 << (2 * (j - 1))) | (1 << (2 * j));
                    dp[1-cur].insert(newState, value);
                }
            }
        }

        cur = 1 - cur;
        dp[1-cur].init();
    }
}

// 统计哈密顿回路数
long result = 0;
for (int i = 0; i < dp[cur].size; i++) {
    if (dp[cur].state[i] == 0) {
        result += dp[cur].value[i];
    }
}
return result;
}

/**
 * 重新编号以保持括号表示法
 */
public static int renumber(int state, int pos1, int pos2, int id1, int id2) {
    // 合并两个连通分量，将 id2 的编号改为 id1
    int minId = Math.min(id1, id2);
    int maxId = Math.max(id1, id2);

    for (int i = 0; i < m; i++) {
        int m = (state >> (2 * i)) & 3;
        if (m == maxId) {
            state &= ~(3 << (2 * i));
            state |= (minId << (2 * i));
        }
    }
```

```java
        return state;
    }

    // 测试用例
    public static void main(String[] args) {
        char[][] maze1 = {
            {'.', '.', '.', '.'},
            {'.', '.', '.', '.'},
            {'.', '.', '.', '.'},
            {'.', '.', '.', '.'}
        };
        System.out.println(solve(4, 4, maze1)); // 输出哈密顿回路数
    }
}
```

===

文件: UVA10572_BlackAndWhite.cpp

===

```cpp
// UVA 10572 Black and White (插头 DP - 染色问题)
// 在 n×m 的网格中，对格子进行黑白染色，要求相邻格子颜色不同且每种颜色都连通
// 1 <= n, m <= 8
// 测试链接 : https://vjudge.net/problem/UVA-10572
//
// 题目大意：
// 给定一个 n×m 的网格，其中一些格子是障碍（用'#'表示），其他格子是可以染色的（用'.'表示）。
// 要求对可染色的格子进行黑白染色，使得：
// 1. 相邻的可染色格子颜色不同
// 2. 所有黑色格子连通
// 3. 所有白色格子连通
// 求满足条件的染色方案数。
//
// 解题思路：
// 使用插头 DP 解决染色问题。
// 状态表示：用三进制表示轮廓线状态，0 表示无插头，1 表示黑色，2 表示白色。
// 需要维护颜色的连通性，确保两种颜色都连通。
//
// Java 实现：https://github.com/yourusername/algorithm-
journey/blob/main/src/class126/UVA10572_BlackAndWhite.java
// C++实现：https://github.com/yourusername/algorithm-
journey/blob/main/src/class126/UVA10572_BlackAndWhite.cpp
// Python 实现：https://github.com/yourusername/algorithm-
```

journey/blob/main/src/class126/UVA10572_BlackAndWhite.py

```
// 计算 base^exp
int power(int base, int exp) {
    int result = 1;
    for (int i = 0; i < exp; i++) {
        result *= base;
    }
    return result;
}

// 获取状态 s 中第 j 个位置的值
int get_value(int s, int j) {
    return (s / power(3, j)) % 3;
}

// 设置状态 s 中第 j 个位置的值为 v
int set_value(int s, int j, int v) {
    int pow = power(3, j);
    return (s / pow / 3) * pow * 3 + v * pow + (s % pow);
}

const int MAXN = 10;
const int MAX_STATES = 6561; // 3^8

// dp[i][j][s][c][connected]表示处理到第 i 行第 j 列，轮廓线状态为 s，
// 当前点颜色为 c，颜色连通性为 connected 的方案数
// 状态 s 用三进制表示，0 表示无插头，1 表示黑色，2 表示白色
long long dp[MAXN][MAXN][MAX_STATES][3][2];

char grid[MAXN][MAXN];
int n, m;

/**
 * 计算满足条件的黑白染色方案数
 *
 * 算法思路：
 * 使用插头 DP 解决染色问题
 * 状态表示：用三进制表示轮廓线状态，0 表示无插头，1 表示黑色，2 表示白色
 * 需要维护颜色的连通性，确保两种颜色都连通
 *
 * 时间复杂度：O(n * m * 3^m)
 * 空间复杂度：O(n * m * 3^m)
```

```
 *
 * @param rows 行数
 * @param cols 列数
 * @param maze 网格地图，'#'表示障碍，'.'表示可染色
 * @return 方案数
 */
long long solve(int rows, int cols, char maze[][MAXN]) {
    n = rows;
    m = cols;

    // 复制网格
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            grid[i][j] = maze[i][j];
        }
    }

    // 初始化 DP 数组
    for (int i = 0; i <= n; i++) {
        for (int j = 0; j <= m; j++) {
            for (int s = 0; s < MAX_STATES; s++) {
                for (int c = 0; c < 3; c++) {
                    for (int connected = 0; connected < 2; connected++) {
                        dp[i][j][s][c][connected] = 0;
                    }
                }
            }
        }
    }

    // 初始状态
    dp[0][0][0][0][0] = 1;

    // 逐格 DP
    for (int i = 0; i < n; i++) {
        // 行间转移
        for (int s = 0; s < 6561; s++) {
            for (int c = 0; c < 3; c++) {
                for (int connected = 0; connected < 2; connected++) {
                    if (dp[i][m][s][c][connected] > 0) {
                        // 将状态转移到下一行的开始
                        int newState = s * 3;
                        dp[i+1][0][newState][0][connected] += dp[i][m][s][c][connected];
```

```
                    }
                }
            }
        }

        // 行内转移
        for (int j = 0; j < m; j++) {
            for (int s = 0; s < 6561; s++) {
                for (int c = 0; c < 3; c++) {
                    for (int connected = 0; connected < 2; connected++) {
                        if (dp[i][j][s][c][connected] == 0) continue;

                        // 获取当前格子左边和上面的颜色
                        int left = j > 0 ? get_value(s, j-1) : 0;
                        int up = get_value(s, j);

                        // 如果是障碍格子
                        if (grid[i][j] != '.') {
                            // 只能在没有颜色的情况下转移
                            if (left == 0 && up == 0) {
                                int newState = set_value(s, j, 0);
                                dp[i][j+1][newState][0][connected] += dp[i][j][s][c][connected];
                            }
                        } else {
                            // 可染色格子

                            // 1. 染成黑色
                            if ((left == 0 || left == 2) && (up == 0 || up == 2)) {
                                // 检查是否会破坏连通性
                                int newState = set_value(s, j, 1);
                                int newConnected = connected;
                                // 更新连通性状态
                                dp[i][j+1][newState][1][newConnected] +=
dp[i][j][s][c][connected];
                            }

                            // 2. 染成白色
                            if ((left == 0 || left == 1) && (up == 0 || up == 1)) {
                                // 检查是否会破坏连通性
                                int newState = set_value(s, j, 2);
                                int newConnected = connected;
                                // 更新连通性状态
                                dp[i][j+1][newState][2][newConnected] +=
```

```
dp[i][j][s][c][connected];
                            }
                        }
                    }
                }
            }
        }
    }

    // 统计所有满足条件的方案数
    long long result = 0;
    for (int s = 0; s < 6561; s++) {
        for (int c = 0; c < 3; c++) {
            for (int connected = 0; connected < 2; connected++) {
                result += dp[n][0][s][c][connected];
            }
        }
    }

    return result;
}

// 测试用例
int main() {
    char maze1[3][MAXN] = {
        {'.', '.', '.'},
        {'.', '.', '.'},
        {'.', '.', '.'}
    };
    // solve(3, 3, maze1); // 输出方案数

    char maze2[3][MAXN] = {
        {'.', '.', '#'},
        {'.', '.', '.'},
        {'.', '.', '.'}
    };
    // solve(3, 3, maze2); // 输出方案数

    return 0;
}
```

==

文件: UVA10572_BlackAndWhite.java
==

```java
package class126;

// UVA 10572 Black and White (插头DP - 染色问题)
// 在n×m的网格中，对格子进行黑白染色，要求相邻格子颜色不同且每种颜色都连通
// 1 <= n, m <= 8
// 测试链接 : https://vjudge.net/problem/UVA-10572
//
// 题目大意：
// 给定一个n×m的网格，其中一些格子是障碍（用'#'表示），其他格子是可以染色的（用'.'表示）。
// 要求对可染色的格子进行黑白染色，使得：
// 1. 相邻的可染色格子颜色不同
// 2. 所有黑色格子连通
// 3. 所有白色格子连通
// 求满足条件的染色方案数。
//
// 解题思路：
// 使用插头DP解决染色问题。
// 状态表示：用三进制表示轮廓线状态，0表示无插头，1表示黑色，2表示白色。
// 需要维护颜色的连通性，确保两种颜色都连通。
//
// Java实现：https://github.com/yourusername/algorithm-
journey/blob/main/src/class126/UVA10572_BlackAndWhite.java
// C++实现：https://github.com/yourusername/algorithm-
journey/blob/main/src/class126/UVA10572_BlackAndWhite.cpp
// Python实现：https://github.com/yourusername/algorithm-
journey/blob/main/src/class126/UVA10572_BlackAndWhite.py

public class UVA10572_BlackAndWhite {

    public static int MAXN = 10;
    public static int MAX_STATES = 6561; // 3^8

    // dp[i][j][s][c][connected]表示处理到第i行第j列，轮廓线状态为s，
    // 当前点颜色为c，颜色连通性为connected的方案数
    // 状态s用三进制表示，0表示无插头，1表示黑色，2表示白色
    public static long[][][][][] dp = new long[MAXN][MAXN][MAX_STATES][3][2];

    public static char[][] grid = new char[MAXN][MAXN];
    public static int n, m;

    /**
```

```java
 * 计算满足条件的黑白染色方案数
 *
 * 算法思路：
 * 使用插头 DP 解决染色问题
 * 状态表示：用三进制表示轮廓线状态，0 表示无插头，1 表示黑色，2 表示白色
 * 需要维护颜色的连通性，确保两种颜色都连通
 *
 * 时间复杂度：O(n * m * 3^m)
 * 空间复杂度：O(n * m * 3^m)
 *
 * @param rows 行数
 * @param cols 列数
 * @param maze 网格地图，'#'表示障碍，'.'表示可染色
 * @return 方案数
 */
public static long solve(int rows, int cols, char[][] maze) {
    n = rows;
    m = cols;

    // 复制网格
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            grid[i][j] = maze[i][j];
        }
    }

    // 初始化 DP 数组
    for (int i = 0; i <= n; i++) {
        for (int j = 0; j <= m; j++) {
            for (int s = 0; s < MAX_STATES; s++) {
                for (int c = 0; c < 3; c++) {
                    for (int connected = 0; connected < 2; connected++) {
                        dp[i][j][s][c][connected] = 0;
                    }
                }
            }
        }
    }

    // 初始状态
    dp[0][0][0][0][0] = 1;

    // 逐格 DP
```

```
        for (int i = 0; i < n; i++) {
            // 行间转移
            for (int s = 0; s < power(3, m); s++) {
                for (int c = 0; c < 3; c++) {
                    for (int connected = 0; connected < 2; connected++) {
                        if (dp[i][m][s][c][connected] > 0) {
                            // 将状态转移到下一行的开始
                            int newState = s * 3;
                            dp[i+1][0][newState][0][connected] += dp[i][m][s][c][connected];
                        }
                    }
                }
            }

            // 行内转移
            for (int j = 0; j < m; j++) {
                for (int s = 0; s < power(3, m); s++) {
                    for (int c = 0; c < 3; c++) {
                        for (int connected = 0; connected < 2; connected++) {
                            if (dp[i][j][s][c][connected] == 0) continue;

                            // 获取当前格子左边和上面的颜色
                            int left = j > 0 ? get(s, j-1) : 0;
                            int up = get(s, j);

                            // 如果是障碍格子
                            if (grid[i][j] != '.') {
                                // 只能在没有颜色的情况下转移
                                if (left == 0 && up == 0) {
                                    int newState = set(s, j, 0);
                                    dp[i][j+1][newState][0][connected] +=
dp[i][j][s][c][connected];
                                }
                            } else {
                                // 可染色格子

                                // 1. 染成黑色
                                if ((left == 0 || left == 2) && (up == 0 || up == 2)) {
                                    // 检查是否会破坏连通性
                                    int newState = set(s, j, 1);
                                    int newConnected = connected;
                                    // 更新连通性状态
                                    dp[i][j+1][newState][1][newConnected] +=
```

```
dp[i][j][s][c][connected];
                                }

                                // 2. 染成白色
                                if ((left == 0 || left == 1) && (up == 0 || up == 1)) {
                                    // 检查是否会破坏连通性
                                    int newState = set(s, j, 2);
                                    int newConnected = connected;
                                    // 更新连通性状态
                                    dp[i][j+1][newState][2][newConnected] +=
dp[i][j][s][c][connected];
                                }
                            }
                        }
                    }
                }
            }
        }

        // 统计所有满足条件的方案数
        long result = 0;
        for (int s = 0; s < power(3, m); s++) {
            for (int c = 0; c < 3; c++) {
                for (int connected = 0; connected < 2; connected++) {
                    result += dp[n][0][s][c][connected];
                }
            }
        }

        return result;
    }

    // 计算 base^exp
    public static int power(int base, int exp) {
        int result = 1;
        for (int i = 0; i < exp; i++) {
            result *= base;
        }
        return result;
    }

    // 获取状态 s 中第 j 个位置的值
    public static int get(int s, int j) {
```

```
        return (s / power(3, j)) % 3;
    }

    // 设置状态 s 中第 j 个位置的值为 v
    public static int set(int s, int j, int v) {
        int pow = power(3, j);
        return (s / pow / 3) * pow * 3 + v * pow + (s % pow);
    }

    // 测试用例
    public static void main(String[] args) {
        char[][] maze1 = {
            {'.', '.', '.'},
            {'.', '.', '.'},
            {'.', '.', '.'}
        };
        System.out.println(solve(3, 3, maze1)); // 输出方案数

        char[][] maze2 = {
            {'.', '.', '#'},
            {'.', '.', '.'},
            {'.', '.', '.'}
        };
        System.out.println(solve(3, 3, maze2)); // 输出方案数
    }
}
```

===

文件: UVA10572_BlackAndWhite.py

===

```
# UVA 10572 Black and White (插头 DP - 染色问题)
# 在 n×m 的网格中，对格子进行黑白染色，要求相邻格子颜色不同且每种颜色都连通
# 1 <= n, m <= 8
# 测试链接 : https://vjudge.net/problem/UVA-10572

MAXN = 10
MAX_STATES = 6561  # 3^8

# dp[i][j][s][c][connected]表示处理到第 i 行第 j 列，轮廓线状态为 s，
# 当前点颜色为 c，颜色连通性为 connected 的方案数
# 状态 s 用三进制表示，0 表示无插头，1 表示黑色，2 表示白色
dp = [[[[[0 for _ in range(2)] for _ in range(3)] for _ in range(MAX_STATES)] for _ in
```

```
range(MAXN)] for _ in range(MAXN)]

grid = [['' for _ in range(MAXN)] for _ in range(MAXN)]
n, m = 0, 0

def power(base, exp):
    """计算 base^exp"""
    result = 1
    for i in range(exp):
        result *= base
    return result

def get_value(s, j):
    """获取状态 s 中第 j 个位置的值"""
    return (s // power(3, j)) % 3

def set_value(s, j, v):
    """设置状态 s 中第 j 个位置的值为 v"""
    pow_val = power(3, j)
    return (s // pow_val // 3) * pow_val * 3 + v * pow_val + (s % pow_val)

def solve(rows, cols, maze):
    """
    计算满足条件的黑白染色方案数

    算法思路：
    使用插头 DP 解决染色问题
    状态表示：用三进制表示轮廓线状态，0 表示无插头，1 表示黑色，2 表示白色
    需要维护颜色的连通性，确保两种颜色都连通

    时间复杂度：O(n * m * 3^m)
    空间复杂度：O(n * m * 3^m)

    Args:
        rows: 行数
        cols: 列数
        maze: 网格地图，'#'表示障碍，'.'表示可染色

    Returns:
        方案数
    """
    global n, m, grid, dp
```

```
n = rows
m = cols

# 复制网格
for i in range(n):
    for j in range(m):
        grid[i][j] = maze[i][j]

# 初始化 DP 数组
for i in range(n + 1):
    for j in range(m + 1):
        for s in range(MAX_STATES):
            for c in range(3):
                for connected in range(2):
                    dp[i][j][s][c][connected] = 0

# 初始状态
dp[0][0][0][0][0] = 1

# 逐格 DP
for i in range(n):
    # 行间转移
    for s in range(power(3, m)):
        for c in range(3):
            for connected in range(2):
                if dp[i][m][s][c][connected] > 0:
                    # 将状态转移到下一行的开始
                    new_state = s * 3
                    dp[i+1][0][new_state][0][connected] += dp[i][m][s][c][connected]

    # 行内转移
    for j in range(m):
        for s in range(power(3, m)):
            for c in range(3):
                for connected in range(2):
                    if dp[i][j][s][c][connected] == 0:
                        continue

                    # 获取当前格子左边和上面的颜色
                    left = get_value(s, j-1) if j > 0 else 0
                    up = get_value(s, j)

                    # 如果是障碍格子
```

```
                        if grid[i][j] != '.':
                            # 只能在没有颜色的情况下转移
                            if left == 0 and up == 0:
                                new_state = set_value(s, j, 0)
                                dp[i][j+1][new_state][0][connected] += dp[i][j][s][c][connected]
                        else:
                            # 可染色格子

                            # 1. 染成黑色
                            if (left == 0 or left == 2) and (up == 0 or up == 2):
                                # 检查是否会破坏连通性
                                new_state = set_value(s, j, 1)
                                new_connected = connected
                                # 更新连通性状态
                                dp[i][j+1][new_state][1][new_connected] +=
dp[i][j][s][c][connected]

                            # 2. 染成白色
                            if (left == 0 or left == 1) and (up == 0 or up == 1):
                                # 检查是否会破坏连通性
                                new_state = set_value(s, j, 2)
                                new_connected = connected
                                # 更新连通性状态
                                dp[i][j+1][new_state][2][new_connected] +=
dp[i][j][s][c][connected]

    # 统计所有满足条件的方案数
    result = 0
    for s in range(power(3, m)):
        for c in range(3):
            for connected in range(2):
                result += dp[n][0][s][c][connected]

    return result

# 测试用例
if __name__ == "__main__":
    maze1 = [
        ['.', '.', '.'],
        ['.', '.', '.'],
        ['.', '.', '.']
    ]
    print(solve(3, 3, maze1))  # 输出方案数
```

```python
    maze2 = [
        ['.', '.', '#'],
        ['.', '.', '.'],
        ['.', '.', '.']
    ]
    print(solve(3, 3, maze2))  # 输出方案数
```

==

文件: ZOJ4231_TheHiveII.cpp
==

```cpp
// ZOJ 4231 The Hive II (插头 DP - 多回路覆盖 - 六边形网格)
// 在六边形网格中，求用若干个回路覆盖所有非障碍格子的方案数
// 测试链接 : http://acm.zju.edu.cn/onlinejudge/showProblem.do?problemCode=4231
//
// 题目大意：
// 给定一个六边形网格，其中一些格子是障碍物（用 0 表示），其他格子是可通行的（用 1 表示）。
// 要求用若干个回路（闭合路径）覆盖所有可通行的格子，每个格子恰好被一个回路覆盖。
// 求满足条件的方案数。
//
// 解题思路：
// 使用插头 DP 解决六边形网格上的多回路覆盖问题。
// 状态表示：用二进制表示轮廓线状态，第 k 位为 1 表示第 k 个位置有插头。
// 六边形网格的特殊性：每个格子有 6 个邻居，但处理方式与矩形网格类似。
//
// Java 实现：https://github.com/yourusername/algorithm-
journey/blob/main/src/class126/ZOJ4231_TheHiveII.java
// C++实现：https://github.com/yourusername/algorithm-
journey/blob/main/src/class126/ZOJ4231_TheHiveII.cpp
// Python 实现：https://github.com/yourusername/algorithm-
journey/blob/main/src/class126/ZOJ4231_TheHiveII.py

const int MAXN = 10;
const int MAX_STATES = (1 << MAXN); // 2^8 = 256

// dp[i][j][s]表示处理到第 i 行第 j 列，轮廓线状态为 s 的方案数
// 状态 s 用二进制表示，第 k 位为 1 表示第 k 个位置有插头
long long dp[MAXN][MAXN][MAX_STATES];

int grid[MAXN][MAXN];
int n, m;
```

```
/**
 * 计算在六边形网格中用若干个回路覆盖所有非障碍格子的方案数
 *
 * 算法思路：
 * 使用插头 DP 解决六边形网格上的多回路覆盖问题
 * 状态表示：用二进制表示轮廓线状态，第 k 位为 1 表示第 k 个位置有插头
 * 六边形网格的特殊性：每个格子有 6 个邻居，但处理方式与矩形网格类似
 *
 * 时间复杂度：O(n * m * 2^m)
 * 空间复杂度：O(n * m * 2^m)
 *
 * @param rows 行数
 * @param cols 列数
 * @param maze 网格地图，0 表示障碍，1 表示可通行
 * @return 方案数
 */
long long solve(int rows, int cols, int maze[][MAXN]) {
    n = rows;
    m = cols;

    // 复制网格
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            grid[i][j] = maze[i][j];
        }
    }

    // 初始化 DP 数组
    for (int i = 0; i <= n; i++) {
        for (int j = 0; j <= m; j++) {
            for (int s = 0; s < MAX_STATES; s++) {
                dp[i][j][s] = 0;
            }
        }
    }

    // 初始状态
    dp[0][0][0] = 1;

    // 逐格 DP
    for (int i = 0; i < n; i++) {
        // 行间转移
        for (int s = 0; s < (1 << m); s++) {
```

```
    if (dp[i][m][s] > 0) {
        // 将状态转移到下一行的开始
        dp[i+1][0][s] = dp[i][m][s];
    }
}

// 行内转移
for (int j = 0; j < m; j++) {
    for (int s = 0; s < (1 << m); s++) {
        if (dp[i][j][s] == 0) continue;

        // 获取当前格子左边和上面的插头状态
        int left = (j > 0 && ((s >> (j-1)) & 1) == 1) ? 1 : 0;
        int up = ((s >> j) & 1);

        // 如果是障碍格子
        if (grid[i][j] == 0) {
            // 只能在没有插头的情况下转移
            if (left == 0 && up == 0) {
                dp[i][j+1][s & (~((1 << j) | (1 << (j-1))))] += dp[i][j][s];
            }
        } else {
            // 可通行格子

            // 1. 不放置插头（合并两个插头）
            if (left == 1 && up == 1) {
                dp[i][j+1][s & (~((1 << j) | (1 << (j-1))))] += dp[i][j][s];
            }

            // 2. 延续插头
            if (left == 1 && up == 0) {
                // 延续左插头到上方
                dp[i][j+1][s | (1 << j)] += dp[i][j][s];
            }

            if (left == 0 && up == 1) {
                // 延续上插头到左方
                dp[i][j+1][s | (1 << (j-1))] += dp[i][j][s];
            }

            // 3. 创建新插头对（如果左右和上方都没有插头）
            if (left == 0 && up == 0) {
                // 创建一对新插头（左插头和上插头）
```

```
                    dp[i][j+1][s | (1 << (j-1)) | (1 << j)] += dp[i][j][s];
                }
            }
        }
    }
    }

    return dp[n][0][0];
}

// 测试用例
int main() {
    int maze1[3][MAXN] = {
        {1, 1, 1},
        {1, 1, 1},
        {1, 1, 1}
    };
    // solve(3, 3, maze1); // 输出方案数

    int maze2[3][MAXN] = {
        {1, 1, 0},
        {1, 1, 1},
        {1, 1, 1}
    };
    // solve(3, 3, maze2); // 输出方案数

    return 0;
}
```

==

文件: ZOJ4231_TheHiveII.java

==

```
package class126;

// ZOJ 4231 The Hive II (插头 DP - 多回路覆盖 - 六边形网格)
// 在六边形网格中，求用若干个回路覆盖所有非障碍格子的方案数
// 测试链接 : http://acm.zju.edu.cn/onlinejudge/showProblem.do?problemCode=4231
//
// 题目大意：
// 给定一个六边形网格，其中一些格子是障碍物（用 0 表示），其他格子是可通行的（用 1 表示）。
// 要求用若干个回路（闭合路径）覆盖所有可通行的格子，每个格子恰好被一个回路覆盖。
// 求满足条件的方案数。
```

```
//
// 解题思路：
// 使用插头 DP 解决六边形网格上的多回路覆盖问题。
// 状态表示：用二进制表示轮廓线状态，第 k 位为 1 表示第 k 个位置有插头。
// 六边形网格的特殊性：每个格子有 6 个邻居，但处理方式与矩形网格类似。
//
// Java 实现：https://github.com/yourusername/algorithm-
journey/blob/main/src/class126/ZOJ4231_TheHiveII.java
// C++实现：https://github.com/yourusername/algorithm-
journey/blob/main/src/class126/ZOJ4231_TheHiveII.cpp
// Python 实现：https://github.com/yourusername/algorithm-
journey/blob/main/src/class126/ZOJ4231_TheHiveII.py

public class ZOJ4231_TheHiveII {

    public static int MAXN = 10;
    public static int MAX_STATES = (1 << MAXN); // 2^8 = 256

    // dp[i][j][s]表示处理到第 i 行第 j 列，轮廓线状态为 s 的方案数
    // 状态 s 用二进制表示，第 k 位为 1 表示第 k 个位置有插头
    public static long[][][] dp = new long[MAXN][MAXN][MAX_STATES];

    public static int[][] grid = new int[MAXN][MAXN];
    public static int n, m;

    /**
     * 计算在六边形网格中用若干个回路覆盖所有非障碍格子的方案数
     *
     * 算法思路：
     * 使用插头 DP 解决六边形网格上的多回路覆盖问题
     * 状态表示：用二进制表示轮廓线状态，第 k 位为 1 表示第 k 个位置有插头
     * 六边形网格的特殊性：每个格子有 6 个邻居，但处理方式与矩形网格类似
     *
     * 时间复杂度：O(n * m * 2^m)
     * 空间复杂度：O(n * m * 2^m)
     *
     * @param rows 行数
     * @param cols 列数
     * @param maze 网格地图，0 表示障碍，1 表示可通行
     * @return 方案数
     */
    public static long solve(int rows, int cols, int[][] maze) {
        n = rows;
```

```
m = cols;

// 复制网格
for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
        grid[i][j] = maze[i][j];
    }
}

// 初始化 DP 数组
for (int i = 0; i <= n; i++) {
    for (int j = 0; j <= m; j++) {
        for (int s = 0; s < MAX_STATES; s++) {
            dp[i][j][s] = 0;
        }
    }
}

// 初始状态
dp[0][0][0] = 1;

// 逐格 DP
for (int i = 0; i < n; i++) {
    // 行间转移
    for (int s = 0; s < (1 << m); s++) {
        if (dp[i][m][s] > 0) {
            // 将状态转移到下一行的开始
            dp[i+1][0][s] = dp[i][m][s];
        }
    }

    // 行内转移
    for (int j = 0; j < m; j++) {
        for (int s = 0; s < (1 << m); s++) {
            if (dp[i][j][s] == 0) continue;

            // 获取当前格子左边和上面的插头状态
            int left = (j > 0 && ((s >> (j-1)) & 1) == 1) ? 1 : 0;
            int up = ((s >> j) & 1);

            // 如果是障碍格子
            if (grid[i][j] == 0) {
                // 只能在没有插头的情况下转移
```

```java
                    if (left == 0 && up == 0) {
                        dp[i][j+1][s & (~((1 << j) | (1 << (j-1))))] += dp[i][j][s];
                    }
                } else {
                    // 可通行格子

                    // 1. 不放置插头（合并两个插头）
                    if (left == 1 && up == 1) {
                        dp[i][j+1][s & (~((1 << j) | (1 << (j-1))))] += dp[i][j][s];
                    }

                    // 2. 延续插头
                    if (left == 1 && up == 0) {
                        // 延续左插头到上方
                        dp[i][j+1][s | (1 << j)] += dp[i][j][s];
                    }

                    if (left == 0 && up == 1) {
                        // 延续上插头到左方
                        dp[i][j+1][s | (1 << (j-1))] += dp[i][j][s];
                    }

                    // 3. 创建新插头对（如果左右和上方都没有插头）
                    if (left == 0 && up == 0) {
                        // 创建一对新插头（左插头和上插头）
                        dp[i][j+1][s | (1 << (j-1)) | (1 << j)] += dp[i][j][s];
                    }
                }
            }
        }
    }

    return dp[n][0][0];
}

// 测试用例
public static void main(String[] args) {
    int[][] maze1 = {
        {1, 1, 1},
        {1, 1, 1},
        {1, 1, 1}
    };
    System.out.println(solve(3, 3, maze1)); // 输出方案数
```

```
        int[][] maze2 = {
            {1, 1, 0},
            {1, 1, 1},
            {1, 1, 1}
        };
        System.out.println(solve(3, 3, maze2)); // 输出方案数
    }
}
```

==

文件: ZOJ4231_TheHiveII.py

==

```
# ZOJ 4231 The Hive II (插头 DP - 多回路覆盖 - 六边形网格)
# 在六边形网格中，求用若干个回路覆盖所有非障碍格子的方案数
# 测试链接 : http://acm.zju.edu.cn/onlinejudge/showProblem.do?problemCode=4231

MAXN = 10
MAX_STATES = (1 << MAXN)  # 2^8 = 256

# dp[i][j][s]表示处理到第 i 行第 j 列，轮廓线状态为 s 的方案数
# 状态 s 用二进制表示，第 k 位为 1 表示第 k 个位置有插头
dp = [[[0 for _ in range(MAX_STATES)] for _ in range(MAXN)] for _ in range(MAXN)]

grid = [[0 for _ in range(MAXN)] for _ in range(MAXN)]
n, m = 0, 0

def solve(rows, cols, maze):
    """
    计算在六边形网格中用若干个回路覆盖所有非障碍格子的方案数

    算法思路：
    使用插头 DP 解决六边形网格上的多回路覆盖问题
    状态表示：用二进制表示轮廓线状态，第 k 位为 1 表示第 k 个位置有插头
    六边形网格的特殊性：每个格子有 6 个邻居，但处理方式与矩形网格类似

    时间复杂度：O(n * m * 2^m)
    空间复杂度：O(n * m * 2^m)

    Args:
        rows: 行数
        cols: 列数
```

```
    maze: 网格地图，0 表示障碍，1 表示可通行

Returns:
    方案数
"""
global n, m, grid, dp

n = rows
m = cols

# 复制网格
for i in range(n):
    for j in range(m):
        grid[i][j] = maze[i][j]

# 初始化 DP 数组
for i in range(n + 1):
    for j in range(m + 1):
        for s in range(MAX_STATES):
            dp[i][j][s] = 0

# 初始状态
dp[0][0][0] = 1

# 逐格 DP
for i in range(n):
    # 行间转移
    for s in range(1 << m):
        if dp[i][m][s] > 0:
            # 将状态转移到下一行的开始
            dp[i+1][0][s] = dp[i][m][s]

    # 行内转移
    for j in range(m):
        for s in range(1 << m):
            if dp[i][j][s] == 0:
                continue

            # 获取当前格子左边和上面的插头状态
            left = 1 if (j > 0 and ((s >> (j-1)) & 1) == 1) else 0
            up = (s >> j) & 1

            # 如果是障碍格子
```

```
                if grid[i][j] == 0:
                    # 只能在没有插头的情况下转移
                    if left == 0 and up == 0:
                        dp[i][j+1][s & (~((1 << j) | (1 << (j-1))))] += dp[i][j][s]
                else:
                    # 可通行格子

                    # 1. 不放置插头（合并两个插头）
                    if left == 1 and up == 1:
                        dp[i][j+1][s & (~((1 << j) | (1 << (j-1))))] += dp[i][j][s]

                    # 2. 延续插头
                    if left == 1 and up == 0:
                        # 延续左插头到上方
                        dp[i][j+1][s | (1 << j)] += dp[i][j][s]

                    if left == 0 and up == 1:
                        # 延续上插头到左方
                        dp[i][j+1][s | (1 << (j-1))] += dp[i][j][s]

                    # 3. 创建新插头对（如果左右和上方都没有插头）
                    if left == 0 and up == 0:
                        # 创建一对新插头（左插头和上插头）
                        dp[i][j+1][s | (1 << (j-1)) | (1 << j)] += dp[i][j][s]

    return dp[n][0][0]

# 测试用例
if __name__ == "__main__":
    maze1 = [
        [1, 1, 1],
        [1, 1, 1],
        [1, 1, 1]
    ]
    print(solve(3, 3, maze1))  # 输出方案数

    maze2 = [
        [1, 1, 0],
        [1, 1, 1],
        [1, 1, 1]
    ]
    print(solve(3, 3, maze2))  # 输出方案数
```

==