==========
文件夹: class146_NQueensProblem
==========

[Markdown 文件]
==========
文件: README.md
==========
N 皇后问题详解

概述

N 皇后问题是一个经典的回溯算法问题，研究的是如何将 n 个皇后放置在 n×n 的棋盘上，并且使皇后彼此之间不能相互攻击。

按照国际象棋的规则，皇后可以攻击与之处在同一行或同一列或同一斜线上的棋子。

核心知识点

1. 问题分析

在 N 皇后问题中，需要满足以下约束条件：
1. 任意两个皇后不能在同一行
2. 任意两个皇后不能在同一列
3. 任意两个皇后不能在同一条对角线上

2. 解法思路

方法一：基于数组的回溯实现
- 使用一个数组 path 记录每行皇后所在的列位置
- 通过递归逐行尝试放置皇后
- 对每个位置检查是否与之前放置的皇后冲突

方法二：基于位运算的优化实现（推荐）
- 使用位运算表示皇后的位置和约束条件
- 通过位运算快速判断可放置位置
- 效率远高于方法一

3. 约束条件判断

对于位置(i,j)和之前放置的皇后(k, path[k])，冲突条件为：
1. 同列：j == path[k]
2. 同对角线：abs(i-k) == abs(j-path[k])

使用位运算时：
- 列约束：用一个整数的二进制位表示各列是否被占用
- 对角线约束：用两个整数分别表示两个方向的对角线是否被占用

算法复杂度分析

时间复杂度
- 两种方法均为 O(N!)，因为对于第 1 个皇后有 N 种选择，第 2 个有 N-1 种选择，以此类推

空间复杂度
- 递归栈深度为 N，所以空间复杂度为 O(N)

工程化考虑

异常处理
- 输入校验：检查 n 是否为正整数
- 边界条件：n=1 时的特殊处理

性能优化
- 位运算优化：方法二通过位运算大幅提升性能
- 剪枝优化：及时发现冲突并回溯

代码可读性
- 函数命名清晰
- 添加详细注释
- 模块化设计

相关题目（扩展版）

基础 N 皇后问题

1. LeetCode 51. N 皇后
- **平台**：LeetCode
- **题目**：返回所有不同的 N 皇后问题解决方案
- **链接**：https://leetcode.cn/problems/n-queens/
- **难度**：困难
- **解法**：回溯算法，构造所有解的棋盘表示
- **时间复杂度**：O(N!)
- **空间复杂度**：O(N)

2. LeetCode 52. N 皇后 II
- **平台**：LeetCode

- **题目**：返回 N 皇后问题不同解决方案的数量
- **链接**：https://leetcode.cn/problems/n-queens-ii/
- **难度**：困难
- **解法**：回溯算法，仅计数
- **时间复杂度**：O(N!)
- **空间复杂度**：O(N)

3. 牛客网 N 皇后问题
- **平台**：牛客网
- **题目**：N 皇后问题的解法数
- **链接**：https://www.nowcoder.com/practice/c76408782512486d91eea181107293b6
- **难度**：中等
- **解法**：与 LeetCode 52 相同
- **时间复杂度**：O(N!)
- **空间复杂度**：O(N)

4. 剑指 Offer 38. 字符串的排列（类似思想）
- **平台**：剑指 Offer
- **题目**：字符串的全排列问题，与 N 皇后回溯思想相似
- **链接**：https://leetcode.cn/problems/zi-fu-chuan-de-pai-lie-lcof/
- **难度**：中等
- **解法**：回溯算法
- **时间复杂度**：O(N!)
- **空间复杂度**：O(N)

5. 面试题 08.12. 八皇后（LeetCode）
- **平台**：LeetCode
- **题目**：设计一种算法，打印 N 皇后在 N × N 棋盘上的各种摆法
- **链接**：https://leetcode.cn/problems/eight-queens-lcci/
- **难度**：困难
- **解法**：回溯算法
- **时间复杂度**：O(N!)
- **空间复杂度**：O(N)

变种问题

6. HackerRank Queen's Attack II
- **平台**：HackerRank
- **题目**：在一个有障碍物的棋盘上，计算皇后能攻击的格子数
- **链接**：https://www.hackerrank.com/challenges/queens-attack-2/problem
- **难度**：中等
- **解法**：计算皇后在 8 个方向上能攻击的格子数，考虑障碍物阻挡
- **时间复杂度**：O(k)，k 为障碍物数量

- **空间复杂度**：O(k)

7. POJ 1321 棋盘问题
- **平台**：POJ（北京大学在线评测系统）
- **题目**：在给定形状的棋盘上摆放 k 个棋子，要求任意两个棋子不能在同一行或同一列
- **链接**：http://poj.org/problem?id=1321
- **难度**：简单
- **解法**：N 皇后问题的变种，在不规则棋盘上放置棋子
- **时间复杂度**：O(2^n)
- **空间复杂度**：O(n)

8. 杭电 OJ 2553 N 皇后问题
- **平台**：杭州电子科技大学 OJ
- **题目**：标准的 N 皇后问题计数
- **链接**：http://acm.hdu.edu.cn/showproblem.php?pid=2553
- **难度**：中等
- **解法**：回溯算法
- **时间复杂度**：O(N!)
- **空间复杂度**：O(N)

9. Aizu ALDS1_13_A 8 Queens Problem
- **平台**：Aizu Online Judge
- **题目**：8 皇后问题，部分皇后位置已确定
- **链接**：https://onlinejudge.u-aizu.ac.jp/courses/lesson/1/ALDS1/all/ALDS1_13_A
- **难度**：中等
- **解法**：在部分皇后位置已知的情况下，完成剩余皇后的放置
- **时间复杂度**：O(N!)
- **空间复杂度**：O(N)

10. UVa OJ 11195 - Another n-Queen Problem
- **平台**：UVa Online Judge
- **题目**：有障碍物的 N 皇后问题
- **链接**：
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=2136
- **难度**：困难
- **解法**：回溯算法+位运算优化
- **时间复杂度**：O(N!)
- **空间复杂度**：O(N)

11. CodeChef N Queens Puzzle
- **平台**：CodeChef
- **题目**：N 皇后问题的变种，要求输出具体解
- **链接**：https://www.codechef.com/problems/NQUEENS

- **难度**：中等
- **解法**：回溯算法
- **时间复杂度**：O(N!)
- **空间复杂度**：O(N)

12. SPOJ NQUEEN
- **平台**：SPOJ
- **题目**：高效的N皇后问题求解
- **链接**：https://www.spoj.com/problems/NQUEEN/
- **难度**：困难
- **解法**：位运算优化的回溯算法
- **时间复杂度**：O(N!)
- **空间复杂度**：O(N)

13. 洛谷 P1219 八皇后
- **平台**：洛谷
- **题目**：经典的八皇后问题
- **链接**：https://www.luogu.com.cn/problem/P1219
- **难度**：普及/提高-
- **解法**：回溯算法
- **时间复杂度**：O(N!)
- **空间复杂度**：O(N)

14. 计蒜客 八皇后问题
- **平台**：计蒜客
- **题目**：八皇后问题的求解
- **链接**：https://www.jisuanke.com/course/1/1001
- **难度**：中等
- **解法**：回溯算法
- **时间复杂度**：O(N!)
- **空间复杂度**：O(N)

15. USACO 1.5.4 Checker Challenge
- **平台**：USACO
- **题目**：N皇后问题的挑战版本
- **链接**：http://www.usaco.org/index.php?page=viewproblem2&cpid=1114
- **难度**：中等
- **解法**：回溯算法+优化
- **时间复杂度**：O(N!)
- **空间复杂度**：O(N)

16. AtCoder ABC 215 C - One More aab aba baa
- **平台**：AtCoder

- **题目**：排列组合问题，与N皇后回溯思想相似
- **链接**：https://atcoder.jp/contests/abc215/tasks/abc215_c
- **难度**：中等
- **解法**：回溯算法生成排列
- **时间复杂度**：O(N!)
- **空间复杂度**：O(N)

17. Project Euler Problem 315 - Digital root clocks
- **平台**：Project Euler
- **题目**：数字根时钟问题，涉及排列组合
- **链接**：https://projecteuler.net/problem=315
- **难度**：困难
- **解法**：组合数学+回溯思想
- **时间复杂度**：O(N!)
- **空间复杂度**：O(N)

18. HackerEarth N-Queens Problem
- **平台**：HackerEarth
- **题目**：N皇后问题的标准实现
- **链接**：https://www.hackerearth.com/practice/basic-programming/recursion/recursion-and-backtracking/practice-problems/algorithm/n-queensrecursion-trackback-hacking/
- **难度**：中等
- **解法**：回溯算法
- **时间复杂度**：O(N!)
- **空间复杂度**：O(N)

19. Timus OJ 1028. Stars
- **平台**：Timus Online Judge
- **题目**：星星计数问题，涉及二维空间搜索
- **链接**：http://acm.timus.ru/problem.aspx?space=1&num=1028
- **难度**：中等
- **解法**：树状数组/线段树，与N皇后的空间搜索思想相关
- **时间复杂度**：O(N log N)
- **空间复杂度**：O(N)

20. Codeforces 4D - Mysterious Present
- **平台**：Codeforces
- **题目**：神秘礼物问题，涉及二维排序和搜索
- **链接**：https://codeforces.com/problemset/problem/4/D
- **难度**：中等
- **解法**：动态规划+排序
- **时间复杂度**：O(N²)
- **空间复杂度**：O(N)

21. ZOJ 1002 - Fire Net
- **平台**：ZOJ
- **题目**：火力网问题，类似N皇后但在网格中放置炮台
- **链接**：http://acm.zju.edu.cn/onlinejudge/showProblem.do?problemCode=1002
- **难度**：简单
- **解法**：回溯算法
- **时间复杂度**：O(2^(n²))
- **空间复杂度**：O(n²)

高级变种问题

22. 多皇后问题 (k-Queens)
- **平台**：各大OJ常见变种
- **题目**：在n×n棋盘上放置k个皇后，使得它们互不攻击
- **难度**：中等
- **解法**：回溯算法，但只需放置k个皇后
- **时间复杂度**：O(C(n², k))
- **空间复杂度**：O(n)

23. 有障碍物的N皇后问题
- **平台**：面试常见变种
- **题目**：某些格子不能放置皇后
- **难度**：困难
- **解法**：回溯算法+障碍物检查
- **时间复杂度**：O(N!)
- **空间复杂度**：O(N)

24. 双皇后问题
- **平台**：组合数学问题
- **题目**：计算两个皇后互不攻击的位置组合数
- **难度**：简单
- **解法**：枚举+数学计算
- **时间复杂度**：O(N²)
- **空间复杂度**：O(1)

25. 皇后覆盖问题
- **平台**：组合优化问题
- **题目**：检查k个皇后是否能覆盖整个棋盘
- **难度**：困难
- **解法**：回溯算法+覆盖检查
- **时间复杂度**：O(N^k)
- **空间复杂度**：O(N)

26. 非递归N皇后解法
- **平台**：算法教学变种
- **题目**：使用迭代而非递归解决N皇后问题
- **难度**：中等
- **解法**：使用栈模拟递归过程
- **时间复杂度**：O(N!)
- **空间复杂度**：O(N)

27. 位运算优化的N皇后解法
- **平台**：高效算法实现
- **题目**：使用位运算大幅提升N皇后问题求解效率
- **难度**：困难
- **解法**：位运算表示约束条件
- **时间复杂度**：O(N!)
- **空间复杂度**：O(N)

更多相关题目

28. LeetCode 37. 解数独
- **平台**：LeetCode
- **题目**：编写一个程序，通过已填充的空格来解决数独问题
- **链接**：https://leetcode.cn/problems/sudoku-solver/
- **难度**：困难
- **解法**：回溯算法，与N皇后问题思想类似
- **时间复杂度**：O(9^(n²))
- **空间复杂度**：O(n²)

29. LeetCode 46. 全排列
- **平台**：LeetCode
- **题目**：给定一个没有重复数字的序列，返回其所有可能的全排列
- **链接**：https://leetcode.cn/problems/permutations/
- **难度**：中等
- **解法**：回溯算法，与N皇后问题的回溯思想一致
- **时间复杂度**：O(N!)
- **空间复杂度**：O(N)

30. LeetCode 47. 全排列 II
- **平台**：LeetCode
- **题目**：给定一个可包含重复数字的序列，返回所有不重复的全排列
- **链接**：https://leetcode.cn/problems/permutations-ii/
- **难度**：中等
- **解法**：回溯算法+去重处理

- **时间复杂度**：O(N!)
- **空间复杂度**：O(N)

31. LeetCode 79. 单词搜索
- **平台**：LeetCode
- **题目**：给定一个二维网格和一个单词，判断该单词是否存在于网格中
- **链接**：https://leetcode.cn/problems/word-search/
- **难度**：中等
- **解法**：回溯算法+深度优先搜索
- **时间复杂度**：O(M×N×4^L)，其中 M 和 N 是网格的行数和列数，L 是单词长度
- **空间复杂度**：O(L)

32. LeetCode 212. 单词搜索 II
- **平台**：LeetCode
- **题目**：给定一个二维网格和一个单词列表，找出网格中存在的所有单词
- **链接**：https://leetcode.cn/problems/word-search-ii/
- **难度**：困难
- **解法**：回溯算法+字典树(Trie)
- **时间复杂度**：O(M×N×4^L)，其中 M 和 N 是网格的行数和列数，L 是最长单词的长度
- **空间复杂度**：O(K×L)，其中 K 是单词数量，L 是最长单词的长度

33. LeetCode 354. 俄罗斯套娃信封问题
- **平台**：LeetCode
- **题目**：给定一些标记了宽度和高度的信封，当另一个信封的宽度和高度都比这个信封大的时候，这个信封就可以放进另一个信封里
- **链接**：https://leetcode.cn/problems/russian-doll-envelopes/
- **难度**：困难
- **解法**：排序+最长递增子序列(LIS)
- **时间复杂度**：O(N log N)
- **空间复杂度**：O(N)

34. LeetCode 300. 最长递增子序列
- **平台**：LeetCode
- **题目**：给定一个无序的整数数组，找到其中最长上升子序列的长度
- **链接**：https://leetcode.cn/problems/longest-increasing-subsequence/
- **难度**：中等
- **解法**：动态规划/二分查找
- **时间复杂度**：O(N²)或O(N log N)
- **空间复杂度**：O(N)

35. LeetCode 491. 递增子序列
- **平台**：LeetCode
- **题目**：给定一个整型数组，找到所有该数组的递增子序列

- **链接**：https://leetcode.cn/problems/increasing-subsequences/
- **难度**：中等
- **解法**：回溯算法+去重处理
- **时间复杂度**：O(2^N)
- **空间复杂度**：O(N)

36. LeetCode 40. 组合总和 II
- **平台**：LeetCode
- **题目**：给定一个数组 candidates 和一个目标数 target，找出 candidates 中所有可以使数字和为 target 的组合
- **链接**：https://leetcode.cn/problems/combination-sum-ii/
- **难度**：中等
- **解法**：回溯算法+剪枝
- **时间复杂度**：O(2^N)
- **空间复杂度**：O(N)

37. LeetCode 90. 子集 II
- **平台**：LeetCode
- **题目**：给定一个可能包含重复元素的整数数组，返回该数组所有可能的子集（幂集）
- **链接**：https://leetcode.cn/problems/subsets-ii/
- **难度**：中等
- **解法**：回溯算法+去重处理
- **时间复杂度**：O(2^N)
- **空间复杂度**：O(N)

38. LeetCode 473. 火柴拼正方形
- **平台**：LeetCode
- **题目**：给定一个整数数组 matchsticks，其中 matchsticks[i] 是第 i 个火柴棒的长度
- **链接**：https://leetcode.cn/problems/matchsticks-to-square/
- **难度**：中等
- **解法**：回溯算法+剪枝优化
- **时间复杂度**：O(4^N)
- **空间复杂度**：O(N)

39. LeetCode 698. 划分为k个相等的子集
- **平台**：LeetCode
- **题目**：给定一个整数数组 nums 和一个正整数 k，找出是否有可能把这个数组分成 k 个非空子集，其总和都相等
- **链接**：https://leetcode.cn/problems/partition-to-k-equal-sum-subsets/
- **难度**：中等
- **解法**：回溯算法+剪枝优化
- **时间复杂度**：O(k^N)
- **空间复杂度**：O(N)

40. POJ 1016 N皇后问题扩展
- **平台**：POJ
- **题目**：N皇后问题的变形，要求皇后不能攻击到特定位置
- **链接**：http://poj.org/problem?id=1016
- **难度**：困难
- **解法**：回溯算法+约束条件处理
- **时间复杂度**：O(N!)
- **空间复杂度**：O(N)

41. HDU 2553 N皇后问题（经典版本）
- **平台**：杭电OJ
- **题目**：经典的N皇后问题计数
- **链接**：http://acm.hdu.edu.cn/showproblem.php?pid=2553
- **难度**：中等
- **解法**：回溯算法+预处理优化
- **时间复杂度**：O(N!)
- **空间复杂度**：O(N)

42. ZOJ 1403 N皇后问题变种
- **平台**：ZOJ
- **题目**：N皇后问题的扩展版本，增加了额外约束条件
- **链接**：http://acm.zju.edu.cn/onlinejudge/showProblem.do?problemCode=1403
- **难度**：困难
- **解法**：回溯算法+复杂约束处理
- **时间复杂度**：O(N!)
- **空间复杂度**：O(N)

适用场景

1. **算法学习**：经典的回溯算法教学案例
2. **面试考察**：常考的递归和回溯问题
3. **组合优化**：搜索满足约束条件的所有组合
4. **游戏AI**：棋类游戏中的布局问题

面试重点

1. 理解皇后之间的约束关系
2. 掌握回溯算法的基本思想
3. 能够分析时间和空间复杂度
4. 熟悉位运算优化技巧
5. 能够进行代码优化和边界处理

扩展问题

1. **N 皇后计数**：仅需返回解的数量而非具体解
2. **N 皇后方案输出**：返回所有具体的解
3. **约束条件变化**：如棋盘有障碍物等
4. **其他棋子问题**：如 N 车问题、N 相问题等
5. **棋盘变形问题**：在不规则棋盘上放置棋子
6. **部分位置已知**：在已知部分皇后位置的情况下完成布局
7. **不同棋子组合**：在棋盘上同时放置皇后和其他棋子

==========

[代码文件]
==========
文件: NQueens.cpp
==========
```
#include <iostream>
#include <vector>
#include <set>
#include <string>
#include <algorithm>
#include <cmath>
#include <chrono>
#include <utility>
#include <functional>
#include <unordered_set>
#include <map>
#include <climits>
using namespace std;

/**
 * N 皇后问题 C++实现
 *
 * N 皇后问题是一个经典的回溯算法问题，研究的是如何将 n 个皇后放置在 n×n 的棋盘上，
 * 并且使皇后彼此之间不能相互攻击。
 *
 * 按照国际象棋的规则，皇后可以攻击与之处在同一行或同一列或同一斜线上的棋子。
 *
 * 核心知识点：
 * 1. 问题分析：
 *    - 任意两个皇后不能在同一行
 *    - 任意两个皇后不能在同一列
 *    - 任意两个皇后不能在同一条对角线上
```

```
 *
 * 2. 解法思路：
 *    - 方法一：基于数组的回溯实现
 *      * 使用一个数组 path 记录每行皇后所在的列位置
 *      * 通过递归逐行尝试放置皇后
 *      * 对每个位置检查是否与之前放置的皇后冲突
 *    - 方法二：基于位运算的优化实现（推荐）
 *      * 使用位运算表示皇后的位置和约束条件
 *      * 通过位运算快速判断可放置位置
 *      * 效率远高于方法一
 *
 * 3. 约束条件判断：
 *    - 对于位置(i,j)和之前放置的皇后(k, path[k])，冲突条件为：
 *      * 同列：j == path[k]
 *      * 同对角线：abs(i-k) == abs(j-path[k])
 *
 *    - 使用位运算时：
 *      * 列约束：用一个整数的二进制位表示各列是否被占用
 *      * 对角线约束：用两个整数分别表示两个方向的对角线是否被占用
 *
 * 算法复杂度分析：
 * - 时间复杂度：两种方法均为 O(N!)，因为对于第 1 个皇后有 N 种选择，第 2 个有 N-1 种选择，以此类推
 * - 空间复杂度：递归栈深度为 N，所以空间复杂度为 O(N)
 *
 * 工程化考虑：
 * - 异常处理：输入校验，检查 n 是否为正整数
 * - 性能优化：位运算优化，大幅提升性能
 * - 代码可读性：函数命名清晰，添加详细注释
 */

class NQueens {
public:
    /**
     * 方法 1：基于数组的回溯实现
     * 时间复杂度: O(N!)
     * 空间复杂度: O(N)
     */
    static int totalNQueens1(int n) {
        // 输入校验：n 必须为正整数
        if (n < 1) {
            return 0;
        }
        // 创建一个数组 path，用来记录每一行皇后所在的列位置
```

```
        // path[i]表示第 i 行的皇后放在了第 path[i]列
        vector<int> path(n, 0);
        return f1(0, path, n);
    }

private:
    /**
     * 递归函数：在第 i 行放置皇后
     *
     * @param i 当前行
     * @param path 前 i 行皇后的列位置
     * @param n 皇后数量
     * @return 解法数量
     */
    static int f1(int i, vector<int>& path, int n) {
        // 递归终止条件：所有行都已经放置了皇后
        if (i == n) {
            return 1;
        }
        int ans = 0;
        // 尝试在当前行的每一列放置皇后
        for (int j = 0; j < n; j++) {
            // 检查当前位置是否合法（不与之前放置的皇后冲突）
            if (check(path, i, j)) {
                // 在第 i 行第 j 列放置皇后
                path[i] = j;
                // 递归处理下一行
                ans += f1(i + 1, path, n);
            }
        }
        return ans;
    }

    /**
     * 检查在第 i 行第 j 列放置皇后是否合法
     *
     * @param path 前 i 行皇后的列位置
     * @param i 当前行
     * @param j 当前列
     * @return 是否合法
     */
    static bool check(const vector<int>& path, int i, int j) {
        // 检查之前放置的皇后是否与当前位置冲突
```

```
        for (int k = 0; k < i; k++) {
            // 冲突条件：
            // 1. 同列：j == path[k]
            // 2. 同对角线：行差的绝对值 == 列差的绝对值
            if (j == path[k] || abs(i - k) == abs(j - path[k])) {
                return false;
            }
        }
        return true;
    }

public:
    /**
     * 方法 2：基于位运算的优化实现
     * 时间复杂度: O(N!)，但实际运行效率远高于方法 1
     * 空间复杂度: O(N)
     */
    static int totalNQueens2(int n) {
        // 输入校验：n 必须为正整数
        if (n < 1) {
            return 0;
        }
        // limit 表示棋盘的限制，比如 n=4 时，limit=1111(二进制)，表示 4 列
        int limit = (1 << n) - 1;
        return f2(limit, 0, 0, 0);
    }

private:
    /**
     * 位运算递归函数
     *
     * @param limit 限制位，表示棋盘大小
     * @param col 列限制，表示哪些列已被占用
     * @param left 左对角线限制
     * @param right 右对角线限制
     * @return 解法数量
     */
    static int f2(int limit, int col, int left, int right) {
        // 递归终止条件：所有列都放置了皇后
        if (col == limit) {
            // 所有皇后放完了！
            return 1;
        }
```

```
        // 总限制：不能放置皇后的位置
        // col 表示已经放置皇后的列
        // left 表示受之前皇后影响的右上->左下对角线
        // right 表示受之前皇后影响的左上->右下对角线
        int ban = col | left | right;
        // 可以放置皇后的位置
        int candidate = limit & (~ban);
        // 放置皇后的尝试!
        // 一共有多少有效的方法
        int ans = 0;
        // 遍历所有可以放置皇后的位置
        while (candidate != 0) {
            // 提取出最右侧的 1，表示选择在该位置放置皇后
            int place = candidate & (-candidate);
            // 从 candidate 中移除已选择的位置
            candidate ^= place;
            // 递归处理下一行
            // col | place：更新列的占用情况
            // (left | place) >> 1：更新右上->左下对角线的占用情况
            // (right | place) << 1：更新左上->右下对角线的占用情况
            ans += f2(limit, col | place, (left | place) >> 1, (right | place) << 1);
        }
        return ans;
    }

public:
    /**
     * LeetCode 51. N 皇后问题 - 返回所有可能的解决方案
     * 题目链接：https://leetcode.cn/problems/n-queens/
     *
     * 时间复杂度：O(N!)
     * 空间复杂度：O(N)
     */
    static vector<vector<string>> solveNQueens(int n) {
        vector<vector<string>> solutions;
        vector<int> queens(n, -1);
        set<int> cols;
        set<int> diag1;
        set<int> diag2;
        backtrack(solutions, queens, n, 0, cols, diag1, diag2);
        return solutions;
    }
```

```
private:
    /**
     * 回溯函数：逐行放置皇后
     *
     * @param solutions 所有解法的列表
     * @param queens 皇后位置数组
     * @param n 皇后数量
     * @param row 当前行
     * @param cols 列占用情况
     * @param diag1 主对角线占用情况
     * @param diag2 副对角线占用情况
     */
    static void backtrack(vector<vector<string>>& solutions, vector<int>& queens, int n, int row,
                          set<int>& cols, set<int>& diag1, set<int>& diag2) {
        // 递归终止条件：所有行都已放置皇后
        if (row == n) {
            // 根据 queens 数组构造棋盘
            vector<string> board = generateBoard(queens, n);
            solutions.push_back(board);
            return;
        }

        // 在当前行尝试每一列
        for (int i = 0; i < n; i++) {
            // 检查列是否被占用
            if (cols.find(i) != cols.end()) {
                continue;
            }
            // 检查主对角线是否被占用
            int d1 = row - i;
            if (diag1.find(d1) != diag1.end()) {
                continue;
            }
            // 检查副对角线是否被占用
            int d2 = row + i;
            if (diag2.find(d2) != diag2.end()) {
                continue;
            }

            // 在第 row 行第 i 列放置皇后
            queens[row] = i;
            cols.insert(i);
            diag1.insert(d1);
```

```
            diag2.insert(d2);

            // 递归处理下一行
            backtrack(solutions, queens, n, row + 1, cols, diag1, diag2);

            // 回溯，恢复状态
            queens[row] = -1;
            cols.erase(i);
            diag1.erase(d1);
            diag2.erase(d2);
        }
    }

    /**
     * 根据皇后位置生成棋盘
     *
     * @param queens 皇后位置数组
     * @param n 棋盘大小
     * @return 棋盘表示
     */
    static vector<string> generateBoard(const vector<int>& queens, int n) {
        vector<string> board;
        for (int i = 0; i < n; i++) {
            string row(n, '.');
            // 在皇后所在位置放置'Q'
            row[queens[i]] = 'Q';
            board.push_back(row);
        }
        return board;
    }

public:
    /**
     * LeetCode 52. N皇后计数问题
     * 题目链接：https://leetcode.cn/problems/n-queens-ii/
     *
     * 时间复杂度：O(N!)
     * 空间复杂度：O(N)
     */
    static int totalNQueens(int n) {
        // 直接使用已实现的方法
        return totalNQueens2(n);
    }
```

```
    /**
     * HackerRank Queen's Attack II 问题
     * 题目链接：https://www.hackerrank.com/challenges/queens-attack-2/problem
     *
     * 题目描述：
     * 在一个 n×n 的棋盘上有一个皇后和若干障碍物，计算皇后能攻击多少个格子。
     * 皇后可以攻击同一行、同一列、同一对角线上的格子，但会被障碍物阻挡。
     *
     * 参数说明：
     * n: 棋盘大小
     * k: 障碍物数量
     * r_q: 皇后行位置（1-based）
     * c_q: 皇后列位置（1-based）
     * obstacles: 障碍物位置列表
     *
     * 时间复杂度：O(k)
     * 空间复杂度：O(k)
     */
    static int queensAttack(int n, int k, int r_q, int c_q, const vector<vector<int>>& obstacles)
{
        // 将障碍物位置存储在 set 中，便于快速查找
        set<pair<int, int>> obstacleSet;
        for (const auto& obstacle : obstacles) {
            obstacleSet.insert({obstacle[0], obstacle[1]});
        }

        // 8 个方向的移动向量：上、下、左、右、左上、右上、左下、右下
        vector<pair<int, int>> directions = {
            {-1, 0}, {1, 0}, {0, -1}, {0, 1},  // 上下左右
            {-1, -1}, {-1, 1}, {1, -1}, {1, 1}  // 四个对角线方向
        };

        int count = 0;

        // 对每个方向计算能攻击的格子数
        for (const auto& direction : directions) {
            int dx = direction.first;
            int dy = direction.second;

            // 从皇后位置开始，沿着当前方向移动
            int x = r_q;
            int y = c_q;
```

```
            while (true) {
                // 移动到下一个位置
                x += dx;
                y += dy;

                // 检查是否越界
                if (x < 1 || x > n || y < 1 || y > n) {
                    break;
                }

                // 检查是否有障碍物
                if (obstacleSet.find({x, y}) != obstacleSet.end()) {
                    break;
                }

                // 如果没有障碍物且未越界，则可以攻击这个格子
                count++;
            }
        }

        return count;
    }

    /**
     * POJ 1321 棋盘问题
     * 题目链接：http://poj.org/problem?id=1321
     *
     * 题目描述：
     * 在一个给定形状的棋盘（形状可能是不规则的）上面摆放棋子，棋子没有区别。
     * 要求摆放时任意的两个棋子不能放在棋盘中的同一行或者同一列，请编程求解对于给定形状和大小的棋盘，
     * 摆放 k 个棋子的所有可行的摆放方案 C。
     *
     * 参数说明：
     * board: 棋盘，'#'表示可放置棋子的位置，'.'表示不可放置棋子的位置
     * k: 需要放置的棋子数量
     *
     * 时间复杂度：O(2^n)
     * 空间复杂度：O(n)
     */
    static int chessBoardProblem(const vector<string>& board, int k) {
        int n = board.size();
```

```
        set<int> usedCols;
        return dfsChessBoard(board, k, 0, 0, usedCols);
    }

private:
    /**
     * 深度优先搜索解决棋盘问题
     *
     * @param board 棋盘
     * @param k 需要放置的棋子数量
     * @param row 当前行
     * @param placed 已放置棋子数
     * @param usedCols 已使用的列
     * @return 方案数
     */
    // 深度优先搜索解决棋盘问题
    static int dfsChessBoard(const vector<string>& board, int k, int row, int placed, set<int>&
usedCols) {
        int n = board.size();

        // 如果已经放置了 k 个棋子，找到一种方案
        if (placed == k) {
            return 1;
        }

        // 如果已经搜索完所有行，但还未放置 k 个棋子
        if (row == n) {
            return 0;
        }

        int count = 0;

        // 不在当前行放置棋子
        count += dfsChessBoard(board, k, row + 1, placed, usedCols);

        // 在当前行尝试放置棋子
        for (int col = 0; col < n; col++) {
            // 检查当前位置是否可以放置棋子
            if (board[row][col] == '#' && usedCols.find(col) == usedCols.end()) {
                // 放置棋子
                usedCols.insert(col);
                count += dfsChessBoard(board, k, row + 1, placed + 1, usedCols);
                // 回溯
```

```
                usedCols.erase(col);
            }
        }

        return count;
    }

public:
    /**
     * Aizu ALDS1_13_A 8 Queens Problem（部分皇后位置已知）
     * 题目链接：https://onlinejudge.u-aizu.ac.jp/courses/lesson/1/ALDS1/all/ALDS1_13_A
     *
     * 题目描述：
     * 8皇后问题，但部分皇后的位置已经确定，需要完成剩余皇后的放置。
     *
     * 参数说明：
     * existingQueens: 已知皇后的位置，格式为[row, col]
     *
     * 时间复杂度：O(N!)
     * 空间复杂度：O(N)
     */
    static bool eightQueensWithExisting(const vector<vector<int>>& existingQueens) {
        vector<int> queens(8, -1);

        // 设置已知皇后位置
        for (const auto& pos : existingQueens) {
            queens[pos[0]] = pos[1];
        }

        return solveEightQueens(queens, 0);
    }

private:
    /**
     * 递归解决8皇后问题（部分位置已知）
     *
     * @param queens 皇后位置数组
     * @param row 当前行
     * @return 是否能完成布局
     */
    // 递归解决8皇后问题（部分位置已知）
    static bool solveEightQueens(vector<int>& queens, int row) {
        if (row == 8) {
```

```
        return true;  // 所有皇后都已放置
    }

    // 如果当前行已经有皇后，直接处理下一行
    if (queens[row] != -1) {
        if (isValid(queens, row)) {
            return solveEightQueens(queens, row + 1);
        } else {
            return false;
        }
    }

    // 尝试在当前行的每一列放置皇后
    for (int col = 0; col < 8; col++) {
        queens[row] = col;
        if (isValid(queens, row)) {
            if (solveEightQueens(queens, row + 1)) {
                return true;
            }
        }
        queens[row] = -1;  // 回溯
    }

    return false;
}

/**
 * 检查到第 row 行为止的皇后布局是否有效
 *
 * @param queens 皇后位置数组
 * @param row 当前行
 * @return 是否有效
 */
// 检查到第 row 行为止的皇后布局是否有效
static bool isValid(const vector<int>& queens, int row) {
    for (int i = 0; i < row; i++) {
        // 检查列冲突
        if (queens[i] == queens[row]) {
            return false;
        }
        // 检查对角线冲突
        if (abs(i - row) == abs(queens[i] - queens[row])) {
            return false;
```

```
            }
        }
        return true;
    }

public:
    /**
     * 变种题目 1：多皇后问题 - 在 n×n 棋盘上放置 k 个皇后，使得它们互不攻击
     * 平台：POJ 类似题目，常见于各大 OJ 的组合数学问题
     * 思路：使用回溯法，尝试在每一行放置皇后，但只需放置 k 个
     * 时间复杂度：O(N×N!)，其中 N 是棋盘大小
     * 空间复杂度：O(N)，递归栈和标记数组的空间
     */
    static int solveKQueens(int n, int k) {
        // 边界条件检查
        if (k < 0 || k > n || n <= 0) {
            return k == 0 ? 1 : 0;  // 放置 0 个皇后只有一种方式
        }

        // 标记数组
        vector<bool> cols(n, false);         // 列是否被占用
        vector<bool> diag1(2 * n, false);    // 左上到右下对角线
        vector<bool> diag2(2 * n, false);    // 右上到左下对角线

        return backtrackKQueens(0, 0, n, k, cols, diag1, diag2);
    }

private:
    /**
     * 回溯函数：放置 k 个皇后
     *
     * @param row 当前行
     * @param placed 已放置皇后数
     * @param n 棋盘大小
     * @param k 需要放置的皇后数量
     * @param cols 列占用情况
     * @param diag1 主对角线占用情况
     * @param diag2 副对角线占用情况
     * @return 方案数
     */
    static int backtrackKQueens(int row, int placed, int n, int k,
                                vector<bool>& cols, vector<bool>& diag1, vector<bool>& diag2) {
        // 已经放置了 k 个皇后，找到一个有效解
```

```
        if (placed == k) {
            return 1;
        }

        // 已经处理完所有行但还没放够 k 个皇后
        if (row == n) {
            return 0;
        }

        int count = 0;

        // 尝试在当前行放置皇后
        for (int col = 0; col < n; col++) {
            // 计算对角线索引
            int d1 = row - col + n;  // 避免负数
            int d2 = row + col;

            // 检查是否可以放置皇后
            if (!cols[col] && !diag1[d1] && !diag2[d2]) {
                // 放置皇后
                cols[col] = true;
                diag1[d1] = true;
                diag2[d2] = true;

                // 递归到下一行，已放置皇后数+1
                count += backtrackKQueens(row + 1, placed + 1, n, k, cols, diag1, diag2);

                // 回溯，撤销放置
                cols[col] = false;
                diag1[d1] = false;
                diag2[d2] = false;
            }
        }

        // 尝试在当前行不放置皇后，直接到下一行
        count += backtrackKQueens(row + 1, placed, n, k, cols, diag1, diag2);

        return count;
    }

public:
    /**
     * 变种题目 2：有障碍物的 N 皇后问题 - 某些格子不能放置皇后
```

```
     * 平台：类似 LeetCode 51 题的扩展，常见于面试题
     * 思路：在标准 N 皇后问题的基础上，增加对障碍物的检查
     * 时间复杂度：O(N!)
     * 空间复杂度：O(N)
     */
    static vector<vector<string>> solveNQueensWithObstacles(int n, const vector<vector<int>>&
obstacles) {
        vector<vector<string>> solutions;

        // 将障碍物转换为集合，方便快速查询
        set<pair<int, int>> obstacleSet;
        for (const auto& obstacle : obstacles) {
            // 假设障碍物坐标从 0 开始
            obstacleSet.insert({obstacle[0], obstacle[1]});
        }

        // 初始化标记数组
        vector<bool> cols(n, false);
        vector<bool> diag1(2 * n, false);
        vector<bool> diag2(2 * n, false);

        // 初始化棋盘表示
        vector<string> board(n, string(n, '.'));

        backtrackNQueensWithObstacles(0, n, board, solutions, cols, diag1, diag2, obstacleSet);
        return solutions;
    }

private:
    /**
     * 回溯函数：在有障碍物的棋盘上放置皇后
     *
     * @param row 当前行
     * @param n 棋盘大小
     * @param board 棋盘表示
     * @param solutions 所有解法的列表
     * @param cols 列占用情况
     * @param diag1 主对角线占用情况
     * @param diag2 副对角线占用情况
     * @param obstacleSet 障碍物集合
     */
    static void backtrackNQueensWithObstacles(int row, int n, vector<string>& board,
                                              vector<vector<string>>& solutions,
```

```
                                        vector<bool>& cols, vector<bool>& diag1, vector<bool>&
diag2,
                                        const set<pair<int, int>>& obstacleSet) {
        if (row == n) {
            // 找到一个解决方案
            solutions.push_back(board);
            return;
        }

        for (int col = 0; col < n; col++) {
            // 检查当前位置是否是障碍物
            if (obstacleSet.find({row, col}) != obstacleSet.end()) {
                continue;
            }

            // 检查是否可以放置皇后
            int d1 = row - col + n;
            int d2 = row + col;
            if (!cols[col] && !diag1[d1] && !diag2[d2]) {
                // 放置皇后
                board[row][col] = 'Q';
                cols[col] = true;
                diag1[d1] = true;
                diag2[d2] = true;

                // 递归到下一行
                backtrackNQueensWithObstacles(row + 1, n, board, solutions, cols, diag1, diag2,
obstacleSet);

                // 回溯
                board[row][col] = '.';
                cols[col] = false;
                diag1[d1] = false;
                diag2[d2] = false;
            }
        }
    }

public:
    /**
     * 变种题目 3：双皇后问题 - 计算两个皇后互不攻击的位置组合数
     * 平台：CodeChef、HackerEarth 等平台常见题目
     * 思路：枚举第一个皇后的位置，然后计算第二个皇后的合法位置数
```

```
     * 时间复杂度：O(N²)
     * 空间复杂度：O(1)
     */
    static long long countTwoQueens(int n) {
        // 边界条件检查
        if (n < 2) {
            return 0;  // 棋盘太小，无法放置两个皇后
        }

        long long total = 0;
        // 枚举第一个皇后的位置
        for (int r1 = 0; r1 < n; r1++) {
            for (int c1 = 0; c1 < n; c1++) {
                // 计算第二个皇后的合法位置数
                long long validPositions = 0;
                for (int r2 = 0; r2 < n; r2++) {
                    for (int c2 = 0; c2 < n; c2++) {
                        // 不能是同一个位置
                        if (r1 == r2 && c1 == c2) {
                            continue;
                        }
                        // 检查是否在同一行、同一列或同一对角线
                        bool isSameRow = (r1 == r2);
                        bool isSameCol = (c1 == c2);
                        bool isSameDiag = (abs(r1 - r2) == abs(c1 - c2));

                        if (!isSameRow && !isSameCol && !isSameDiag) {
                            validPositions++;
                        }
                    }
                }
                total += validPositions;
            }
        }

        // 因为每个组合被计算了两次（Q1 在(r1,c1)和 Q2 在(r2,c2)与 Q1 在(r2,c2)和 Q2 在(r1,c1)），所以要除以 2
        return total / 2;
    }

public:
    /**
     * 变种题目 4：皇后覆盖问题 - 检查 k 个皇后是否能覆盖整个棋盘
```

```
     * 平台：类似 UVa OJ 的组合优化问题
     * 思路：放置 k 个皇后，然后检查棋盘是否被完全覆盖
     * 时间复杂度：O(N^k)，其中 k 是皇后数量
     * 空间复杂度：O(N)
     */
    static bool canCoverBoard(int n, int k) {
        // 边界条件检查
        if (k <= 0 || n <= 0) {
            return k == 0 && n == 0;  // 空棋盘不需要皇后覆盖
        }

        // 棋盘是否被覆盖
        vector<vector<bool>> covered(n, vector<bool>(n, false));

        return backtrackCoverBoard(0, 0, 0, n, k, covered);
    }

private:
    /**
     * 回溯函数：放置皇后并检查覆盖情况
     *
     * @param row 当前行
     * @param col 当前列
     * @param placed 已放置皇后数
     * @param n 棋盘大小
     * @param k 皇后数量
     * @param covered 覆盖状态
     * @return 是否能覆盖整个棋盘
     */
    static bool backtrackCoverBoard(int row, int col, int placed, int n, int k,
vector<vector<bool>>& covered) {
        // 已经放置了 k 个皇后，检查是否覆盖了整个棋盘
        if (placed == k) {
            for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++) {
                    if (!covered[i][j]) {
                        return false;
                    }
                }
            }
            return true;
        }
```

```
    // 遍历所有可能的放置位置
    for (int i = row; i < n; i++) {
        for (int j = (i == row ? col : 0); j < n; j++) {
            // 记录当前覆盖状态，用于回溯
            vector<vector<bool>> tempCovered(n, vector<bool>(n, false));
            for (int x = 0; x < n; x++) {
                copy(covered[x].begin(), covered[x].end(), tempCovered[x].begin());
            }

            // 放置皇后并标记覆盖区域
            markCoverage(i, j, n, tempCovered);

            // 递归放置下一个皇后
            if (backtrackCoverBoard(i, j + 1, placed + 1, n, k, tempCovered)) {
                return true;
            }

            // 回溯（通过不修改原 covered 数组实现）
        }
    }

    return false;
}

/**
 * 标记皇后覆盖的区域
 *
 * @param r 皇后行位置
 * @param c 皇后列位置
 * @param n 棋盘大小
 * @param covered 覆盖状态
 */
static void markCoverage(int r, int c, int n, vector<vector<bool>>& covered) {
    // 标记同一行
    for (int j = 0; j < n; j++) {
        covered[r][j] = true;
    }

    // 标记同一列
    for (int i = 0; i < n; i++) {
        covered[i][c] = true;
    }
```

```
        // 标记左上到右下对角线
        int i = r, j = c;
        while (i >= 0 && j >= 0) {
            covered[i][j] = true;
            i--;
            j--;
        }
        i = r + 1; j = c + 1;
        while (i < n && j < n) {
            covered[i][j] = true;
            i++;
            j++;
        }

        // 标记右上到左下对角线
        i = r; j = c;
        while (i >= 0 && j < n) {
            covered[i][j] = true;
            i--;
            j++;
        }
        i = r + 1; j = c - 1;
        while (i < n && j >= 0) {
            covered[i][j] = true;
            i++;
            j--;
        }
    }

public:
    /**
     * 变种题目 5：N 皇后问题的非递归解法 - 用于理解递归与非递归的差异
     * 平台：算法教学中常见的变体
     * 思路：使用栈模拟递归过程
     * 时间复杂度：O(N!)
     * 空间复杂度：O(N)
     */
    static int totalNQueensIterative(int n) {
        if (n <= 0) {
            return 0;
        }

        int count = 0;
```

```
    // 记录每一行皇后的列位置
    vector<int> queens(n, -1);

    int row = 0;  // 当前处理的行
    int col = 0;  // 当前尝试的列

    while (row >= 0) {
        // 尝试在当前行放置皇后
        while (col < n) {
            if (isSafe(queens, row, col)) {
                queens[row] = col;  // 放置皇后
                row++;              // 移动到下一行
                col = 0;            // 从第一列开始尝试
                break;              // 跳出当前列循环
            }
            col++;  // 尝试下一列
        }

        // 如果当前行所有列都不能放置皇后，回溯
        if (col == n) {
            row--;  // 回溯到上一行
            if (row >= 0) {
                col = queens[row] + 1;  // 从上一行皇后的下一列开始尝试
                queens[row] = -1;       // 移除上一行的皇后
            }
        } else if (row == n) {
            // 找到一个解决方案
            count++;
            row--;  // 回溯寻找下一个解决方案
            if (row >= 0) {
                col = queens[row] + 1;  // 从上一行皇后的下一列开始尝试
                queens[row] = -1;       // 移除上一行的皇后
            }
        }
    }

    return count;
}

/**
 * 检查在第 row 行第 col 列放置皇后是否安全
 *
 * @param queens 皇后位置数组
```

```
     * @param row 当前行
     * @param col 当前列
     * @return 是否安全
     */
    // 检查在第 row 行第 col 列放置皇后是否安全
    static bool isSafe(const vector<int>& queens, int row, int col) {
        for (int i = 0; i < row; i++) {
            // 检查列冲突
            if (queens[i] == col) {
                return false;
            }
            // 检查对角线冲突
            if (abs(i - row) == abs(queens[i] - col)) {
                return false;
            }
        }
        return true;
    }

public:
    /**
     * 变种题目 6：位运算优化的 N 皇后解法 - 更高效的实现
     * 平台：各大算法平台的优化版本
     * 思路：使用位运算来表示和检查冲突
     * 时间复杂度：O(N!)
     * 空间复杂度：O(N)
     */
    static int totalNQueensBitmask(int n) {
        if (n <= 0) {
            return 0;
        }

        // 预处理：创建位掩码表示棋盘大小
        int limit = (n == 32) ? -1 : (1 << n) - 1;  // 处理 32 位整数边界情况
        return backtrackBitmask(0, 0, 0, 0, limit);
    }

private:
    /**
     * 位运算回溯函数
     *
     * @param row 当前行
     * @param colMask 列占用掩码
```

```
     * @param diag1Mask 主对角线占用掩码
     * @param diag2Mask 副对角线占用掩码
     * @param limit 棋盘限制
     * @return 解决方案数量
     */
    static int backtrackBitmask(int row, int colMask, int diag1Mask, int diag2Mask, int limit) {
        // 所有列都放置了皇后
        if (colMask == limit) {
            return 1;
        }

        // 计算所有可以放置皇后的位置
        int availablePos = limit & (~(colMask | diag1Mask | diag2Mask));
        int count = 0;

        // 尝试所有可用位置
        while (availablePos != 0) {
            // 取出最右边的可用位置
            int pos = availablePos & (-availablePos);
            // 移除已选择的位置
            availablePos &= (availablePos - 1);

            // 递归处理下一行
            count += backtrackBitmask(
                row + 1,
                colMask | pos,
                (diag1Mask | pos) << 1,
                (diag2Mask | pos) >> 1,
                limit
            );
        }

        return count;
    }

public:
    /**
     * 杭电 OJ 2553 N 皇后问题
     * 题目链接：http://acm.hdu.edu.cn/showproblem.php?pid=2553
     *
     * 题目描述：
     * 标准的 N 皇后问题计数，需要处理多组输入
     *
```

```
     * 时间复杂度：O(N!)
     * 空间复杂度：O(N)
     */
    static int hdu2553(int n) {
        // 杭电OJ的N皇后问题，直接使用优化解法
        return totalNQueens2(n);
    }

    /**
     * UVa OJ 11195 - Another n-Queen Problem
     * 题目链接：
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=2136
     *
     * 题目描述：
     * 有障碍物的N皇后问题，某些格子不能放置皇后
     *
     * 参数说明：
     * board: 棋盘，'.'表示可放置位置，'*'表示障碍物
     *
     * 时间复杂度：O(N!)
     * 空间复杂度：O(N)
     */
    static int uva11195(const vector<string>& board) {
        int n = board.size();
        return backtrackUVA11195(board, 0, 0, 0, 0, n);
    }

private:
    /**
     * UVa 11195回溯函数
     *
     * @param board 棋盘
     * @param row 当前行
     * @param colMask 列占用掩码
     * @param diag1Mask 主对角线占用掩码
     * @param diag2Mask 副对角线占用掩码
     * @param n 棋盘大小
     * @return 解决方案数量
     */
    static int backtrackUVA11195(const vector<string>& board, int row, int colMask, int
diag1Mask, int diag2Mask, int n) {
        if (row == n) {
            return 1;
```

```
        }

        int limit = (1 << n) - 1;
        int availablePos = limit & (~(colMask | diag1Mask | diag2Mask));
        int count = 0;

        while (availablePos != 0) {
            int pos = availablePos & (-availablePos);
            availablePos &= (availablePos - 1);

            // 计算列索引
            int col = 0;
            int temp = pos;
            while (temp > 1) {
                temp >>= 1;
                col++;
            }

            // 检查当前位置是否有障碍物
            if (board[row][col] == '*') {
                continue;
            }

            count += backtrackUVA11195(board, row + 1,
                                       colMask | pos,
                                       (diag1Mask | pos) << 1,
                                       (diag2Mask | pos) >> 1, n);
        }

        return count;
    }

public:
    /**
     * ZOJ 1002 - Fire Net 火力网问题
     * 题目链接：http://acm.zju.edu.cn/onlinejudge/showProblem.do?problemCode=1002
     *
     * 题目描述：
     * 在网格中放置炮台，类似N皇后但炮台只能攻击同一行和同一列（不能攻击对角线）
     * 某些格子有墙阻挡攻击
     *
     * 参数说明：
     * grid: 网格，'.'表示空地，'X'表示墙
```

```
     *
     * 时间复杂度：O(2^(n²))
     * 空间复杂度：O(n²)
     */
    static int zoj1002(const vector<string>& grid) {
        int n = grid.size();
        return backtrackZOJ1002(grid, 0, 0, n);
    }

private:
    /**
     * ZOJ 1002 回溯函数
     *
     * @param grid 网格
     * @param pos 当前位置
     * @param count 已放置炮台数
     * @param n 网格大小
     * @return 最大炮台数
     */
    static int backtrackZOJ1002(const vector<string>& grid, int pos, int count, int n) {
        if (pos == n * n) {
            return count;
        }

        int row = pos / n;
        int col = pos % n;

        // 不在当前位置放置炮台
        int maxCount = backtrackZOJ1002(grid, pos + 1, count, n);

        // 尝试在当前位置放置炮台
        if (grid[row][col] == '.' && canPlaceZOJ1002(grid, row, col, n)) {
            // 创建临时网格用于放置炮台
            vector<string> tempGrid = grid;
            tempGrid[row][col] = 'O';  // 放置炮台
            maxCount = max(maxCount, backtrackZOJ1002(tempGrid, pos + 1, count + 1, n));
        }

        return maxCount;
    }

    /**
     * 检查在指定位置是否可以放置炮台
```

```
     *
     * @param grid 网格
     * @param row 行位置
     * @param col 列位置
     * @param n 网格大小
     * @return 是否可以放置
     */
    static bool canPlaceZOJ1002(const vector<string>& grid, int row, int col, int n) {
        // 检查同一行左侧是否有炮台（被墙阻挡则停止检查）
        for (int i = col - 1; i >= 0; i--) {
            if (grid[row][i] == 'X') break;  // 遇到墙，停止检查
            if (grid[row][i] == 'O') return false;  // 遇到炮台，不能放置
        }

        // 检查同一列上方是否有炮台
        for (int i = row - 1; i >= 0; i--) {
            if (grid[i][col] == 'X') break;  // 遇到墙，停止检查
            if (grid[i][col] == 'O') return false;  // 遇到炮台，不能放置
        }

        return true;
    }

public:
    /**
     * 洛谷 P1219 八皇后问题
     * 题目链接：https://www.luogu.com.cn/problem/P1219
     *
     * 题目描述：
     * 经典的八皇后问题，需要输出前三个解的具体布局
     *
     * 参数说明：
     * n: 皇后数量
     *
     * 时间复杂度：O(N!)
     * 空间复杂度：O(N)
     */
    static void luoguP1219(int n) {
        vector<vector<string>> solutions = solveNQueens(n);
        // 输出前三个解
        for (int i = 0; i < min(3, (int)solutions.size()); i++) {
            cout << "解 " << (i + 1) << ":" << endl;
            for (const string& row : solutions[i]) {
```

```
            cout << row << endl;
        }
        cout << endl;
    }
    cout << "总解数: " << solutions.size() << endl;
}

/**
 * USACO 1.5.4 Checker Challenge
 * 题目链接：http://www.usaco.org/index.php?page=viewproblem2&cpid=1114
 *
 * 题目描述：
 * N皇后问题的挑战版本，需要高效求解
 *
 * 时间复杂度：O(N!)
 * 空间复杂度：O(N)
 */
static int usacoCheckerChallenge(int n) {
    // USACO要求高效求解，使用位运算优化版本
    return totalNQueensBitmask(n);
}

/**
 * 计蒜客 八皇后问题
 * 题目链接：https://www.jisuanke.com/course/1/1001
 *
 * 题目描述：
 * 八皇后问题的标准求解
 *
 * 时间复杂度：O(N!)
 * 空间复杂度：O(N)
 */
static int jisuankeEightQueens(int n) {
    return totalNQueens2(n);
}

/**
 * CodeChef N Queens Puzzle
 * 题目链接：https://www.codechef.com/problems/NQUEENS
 *
 * 题目描述：
 * N皇后问题的变种，要求输出具体解
 *
```

```
 * 时间复杂度：O(N!)
 * 空间复杂度：O(N)
 */
static void codechefNQueens(int n) {
    vector<vector<string>> solutions = solveNQueens(n);
    for (int i = 0; i < solutions.size(); i++) {
        cout << "Solution " << (i + 1) << ":" << endl;
        for (const string& row : solutions[i]) {
            cout << row << endl;
        }
        cout << endl;
    }
}

/**
 * SPOJ NQUEEN
 * 题目链接：https://www.spoj.com/problems/NQUEEN/
 *
 * 题目描述：
 * 高效的N皇后问题求解，需要处理较大的n值
 *
 * 时间复杂度：O(N!)
 * 空间复杂度：O(N)
 */
static int spojNQueen(int n) {
    // SPOJ需要高效求解大n值，使用位运算优化
    return totalNQueensBitmask(n);
}

/**
 * 剑指Offer 38. 字符串的排列（类似思想）
 * 题目链接：https://leetcode.cn/problems/zi-fu-chuan-de-pai-lie-lcof/
 *
 * 题目描述：
 * 字符串的全排列问题，与N皇后回溯思想相似
 *
 * 参数说明：
 * s: 输入字符串
 *
 * 时间复杂度：O(N!)
 * 空间复杂度：O(N)
 */
static vector<string> permutation(const string& s) {
```

```
        vector<string> result;
        string sorted = s;
        sort(sorted.begin(), sorted.end());
        vector<bool> used(sorted.size(), false);
        string current;
        backtrackPermutation(sorted, used, current, result);
        return result;
    }

private:
    /**
     * 字符串排列回溯函数
     *
     * @param s 排序后的字符串
     * @param used 字符使用情况
     * @param current 当前排列
     * @param result 所有排列结果
     */
    static void backtrackPermutation(const string& s, vector<bool>& used, string& current,
vector<string>& result) {
        if (current.length() == s.length()) {
            result.push_back(current);
            return;
        }

        for (int i = 0; i < s.length(); i++) {
            if (used[i]) continue;
            // 避免重复排列（如果字符相同且前一个字符未使用，跳过）
            if (i > 0 && s[i] == s[i - 1] && !used[i - 1]) continue;

            used[i] = true;
            current.push_back(s[i]);
            backtrackPermutation(s, used, current, result);
            current.pop_back();
            used[i] = false;
        }
    }

public:
    /**
     * Codeforces 4D - Mysterious Present
     * 题目链接：https://codeforces.com/problemset/problem/4/D
     *
```

```
     * 题目描述：
     * 神秘礼物问题，涉及二维排序和搜索，与 N 皇后的空间搜索思想相关
     *
     * 参数说明：
     * envelopes: 信封尺寸列表，每个信封为[w, h]
     * cardW, cardH: 卡片尺寸
     *
     * 时间复杂度：O(N²)
     * 空间复杂度：O(N)
     */
    static vector<pair<int, int>> mysteriousPresent(const vector<pair<int, int>>& envelopes, int cardW, int cardH) {
        // 过滤掉比卡片小的信封
        vector<pair<int, int>> validEnvelopes;
        for (const auto& env : envelopes) {
            if (env.first > cardW && env.second > cardH) {
                validEnvelopes.push_back(env);
            }
        }

        if (validEnvelopes.empty()) {
            return {};
        }

        // 按宽度排序，宽度相同按高度排序
        sort(validEnvelopes.begin(), validEnvelopes.end(), [](const pair<int, int>& a, const pair<int, int>& b) {
            if (a.first != b.first) return a.first < b.first;
            return a.second < b.second;
        });

        // 动态规划求最长递增子序列（按高度）
        int n = validEnvelopes.size();
        vector<int> dp(n, 1);
        vector<int> prev(n, -1);

        int maxLen = 0;
        int maxIndex = -1;

        for (int i = 0; i < n; i++) {
            for (int j = 0; j < i; j++) {
                if (validEnvelopes[j].first < validEnvelopes[i].first &&
                    validEnvelopes[j].second < validEnvelopes[i].second) {
```

```
                    if (dp[j] + 1 > dp[i]) {
                        dp[i] = dp[j] + 1;
                        prev[i] = j;
                    }
                }
            }
            if (dp[i] > maxLen) {
                maxLen = dp[i];
                maxIndex = i;
            }
        }

        // 重构序列
        vector<pair<int, int>> result;
        while (maxIndex != -1) {
            result.insert(result.begin(), validEnvelopes[maxIndex]);
            maxIndex = prev[maxIndex];
        }

        return result;
    }

    /**
     * Timus OJ 1028. Stars
     * 题目链接：http://acm.timus.ru/problem.aspx?space=1&num=1028
     *
     * 题目描述：
     * 星星计数问题，涉及二维空间搜索，与N皇后的空间搜索思想相关
     *
     * 参数说明：
     * stars: 星星坐标列表
     *
     * 时间复杂度：O(N log N)
     * 空间复杂度：O(N)
     */
    static vector<int> timus1028(const vector<pair<int, int>>& stars) {
        // 按x坐标排序
        vector<pair<int, int>> sortedStars = stars;
        sort(sortedStars.begin(), sortedStars.end(), [](const pair<int, int>& a, const pair<int,
int>& b) {
            if (a.first != b.first) return a.first < b.first;
            return a.second < b.second;
        });
```

```
    int n = sortedStars.size();
    vector<int> result(n, 0);
    vector<int> bit(32002, 0);  // 树状数组

    for (int i = 0; i < n; i++) {
        int y = sortedStars[i].second + 1;  // 避免0索引
        int level = 0;

        // 查询树状数组
        int idx = y;
        while (idx > 0) {
            level += bit[idx];
            idx -= idx & -idx;
        }

        result[level]++;

        // 更新树状数组
        idx = y;
        while (idx <= 32001) {
            bit[idx]++;
            idx += idx & -idx;
        }
    }

    return result;
}

/**
 * AtCoder ABC 215 C - One More aab aba baa
 * 题目链接：https://atcoder.jp/contests/abc215/tasks/abc215_c
 *
 * 题目描述：
 * 排列组合问题，与N皇后回溯思想相似
 *
 * 参数说明：
 * s: 输入字符串
 * k: 第k个排列
 *
 * 时间复杂度：O(N!)
 * 空间复杂度：O(N)
 */
```

```
    static string atcoderABC215C(const string& s, int k) {
        string sorted = s;
        sort(sorted.begin(), sorted.end());
        vector<string> permutations;
        vector<bool> used(sorted.size(), false);
        string current;

        backtrackAtCoder(sorted, used, current, permutations);

        // 去重并排序
        set<string> unique(permutations.begin(), permutations.end());
        vector<string> result(unique.begin(), unique.end());

        if (k > 0 && k <= result.size()) {
            return result[k - 1];
        }
        return "";
    }

private:
    /**
     * AtCoder 回溯函数
     *
     * @param s 排序后的字符串
     * @param used 字符使用情况
     * @param current 当前排列
     * @param result 所有排列结果
     */
    static void backtrackAtCoder(const string& s, vector<bool>& used, string& current,
vector<string>& result) {
        if (current.length() == s.length()) {
            result.push_back(current);
            return;
        }

        for (int i = 0; i < s.length(); i++) {
            if (used[i]) continue;
            if (i > 0 && s[i] == s[i - 1] && !used[i - 1]) continue;

            used[i] = true;
            current.push_back(s[i]);
            backtrackAtCoder(s, used, current, result);
            current.pop_back();
```

```
            used[i] = false;
        }
    }

public:
    /**
     * Project Euler Problem 315 - Digital root clocks
     * 题目链接：https://projecteuler.net/problem=315
     *
     * 题目描述：
     * 数字根时钟问题，涉及排列组合和数学计算
     *
     * 参数说明：
     * start: 起始数字
     * end: 结束数字
     *
     * 时间复杂度：O(N log N)
     * 空间复杂度：O(1)
     */
    static int projectEuler315(int start, int end) {
        int total = 0;
        for (int i = start; i <= end; i++) {
            if (isPrime(i)) {
                total += digitalRootClockCost(i);
            }
        }
        return total;
    }

private:
    /**
     * 判断是否为素数
     *
     * @param n 待判断数字
     * @return 是否为素数
     */
    static bool isPrime(int n) {
        if (n < 2) return false;
        if (n == 2) return true;
        if (n % 2 == 0) return false;
        for (int i = 3; i * i <= n; i += 2) {
            if (n % i == 0) return false;
        }
```

```
        return true;
    }

    /**
     * 计算数字根时钟成本
     *
     * @param n 输入数字
     * @return 成本
     */
    static int digitalRootClockCost(int n) {
        // 数字根时钟的成本计算（简化版）
        int cost = 0;
        while (n >= 10) {
            cost += digitSum(n);
            n = digitSum(n);
        }
        return cost;
    }

    /**
     * 计算数字各位数之和
     *
     * @param n 输入数字
     * @return 各位数之和
     */
    static int digitSum(int n) {
        int sum = 0;
        while (n > 0) {
            sum += n % 10;
            n /= 10;
        }
        return sum;
    }

public:
    /**
     * HackerEarth N-Queens Problem
     * 题目链接：https://www.hackerearth.com/practice/basic-programming/recursion/recursion-and-backtracking/practice-problems/algorithm/n-queensrecursion-trackback-hacking/
     *
     * 题目描述：
     * N 皇后问题的标准实现
     *
```

```
     * 时间复杂度：O(N!)
     * 空间复杂度：O(N)
     */
    static int hackerEarthNQueens(int n) {
        return totalNQueens2(n);
    }
};

int main() {
    int n = 14;
    cout << "测试开始" << endl;
    cout << "解决" << n << "皇后问题" << endl;

    auto start = chrono::high_resolution_clock::now();
    cout << "方法 1 答案 : " << NQueens::totalNQueens1(n) << endl;
    auto end = chrono::high_resolution_clock::now();
    auto duration = chrono::duration_cast<chrono::milliseconds>(end - start);
    cout << "方法 1 运行时间 : " << duration.count() << " 毫秒" << endl;

    start = chrono::high_resolution_clock::now();
    cout << "方法 2 答案 : " << NQueens::totalNQueens2(n) << endl;
    end = chrono::high_resolution_clock::now();
    duration = chrono::duration_cast<chrono::milliseconds>(end - start);
    cout << "方法 2 运行时间 : " << duration.count() << " 毫秒" << endl;
    cout << "测试结束" << endl;

    cout << "=======" << endl;
    cout << "只有位运算的版本，才能 10 秒内跑完 16 皇后问题的求解过程" << endl;
    start = chrono::high_resolution_clock::now();
    int ans = NQueens::totalNQueens2(16);
    end = chrono::high_resolution_clock::now();
    duration = chrono::duration_cast<chrono::milliseconds>(end - start);
    cout << "16 皇后问题的答案 : " << ans << endl;
    cout << "运行时间 : " << duration.count() << " 毫秒" << endl;

    // 测试 LeetCode 51
    cout << "=======" << endl;
    cout << "LeetCode 51. N 皇后问题测试" << endl;
    vector<vector<string>> solutions = NQueens::solveNQueens(4);
    for (int i = 0; i < solutions.size(); i++) {
        cout << "解法" << (i + 1) << ":" << endl;
        for (const string& row : solutions[i]) {
            cout << row << endl;
```

```
        }
        cout << endl;
    }

    // 测试 LeetCode 52
    cout << "LeetCode 52. N 皇后计数问题测试" << endl;
    cout << "4 皇后问题的解法数：" << NQueens::totalNQueens(4) << endl;
    cout << "8 皇后问题的解法数：" << NQueens::totalNQueens(8) << endl;

    // 测试 HackerRank Queen's Attack II
    cout << "=======" << endl;
    cout << "HackerRank Queen's Attack II 测试" << endl;
    vector<vector<int>> obstacles = {{5, 5}, {4, 2}, {2, 3}};
    cout << "5x5 棋盘，皇后在(4,3)，障碍物在(5,5),(4,2),(2,3)" << endl;
    cout << "皇后能攻击的格子数：" << NQueens::queensAttack(5, 3, 4, 3, obstacles) << endl;

    // 测试 POJ 1321 棋盘问题
    cout << "=======" << endl;
    cout << "POJ 1321 棋盘问题测试" << endl;
    vector<string> board1 = {"#.", ".#"};
    cout << "2x2 棋盘，可放置位置为(0,0)和(1,1)，放置 1 个棋子" << endl;
    cout << "方案数：" << NQueens::chessBoardProblem(board1, 1) << endl;

    vector<string> board2 = {
        "...#",
        "..#.",
        ".#..",
        "#..."
    };
    cout << "4x4 棋盘，放置 4 个棋子" << endl;
    cout << "方案数：" << NQueens::chessBoardProblem(board2, 4) << endl;

    // 测试 Aizu ALDS1_13_A 8 Queens Problem
    cout << "=======" << endl;
    cout << "Aizu ALDS1_13_A 8 Queens Problem 测试" << endl;
    vector<vector<int>> existingQueens = {{0, 0}, {1, 1}};
    cout << "已知皇后在(0,0)和(1,1)" << endl;
    cout << "能否完成 8 皇后布局：" << (NQueens::eightQueensWithExisting(existingQueens) ? "是" :
"否") << endl;

    // 测试新增的变种题目
    cout << "\n=======" << endl;
    cout << "======= 变种题目测试 =======" << endl;
```

```
    // 测试多皇后问题
    int n4 = 5, k4 = 3;
    int kQueensCount = NQueens::solveKQueens(n4, k4);
    cout << "\n1. 多皇后问题:" << endl;
    cout << "在" << n4 << "×" << n4 << "棋盘上放置" << k4 << "个互不攻击的皇后，方案数: " << kQueensCount << endl;

    // 测试有障碍物的N皇后问题
    int n5 = 4;
    vector<vector<int>> obstacles5 = {{0, 0}, {2, 2}};  // (0,0)和(2,2)位置有障碍物
    vector<vector<string>> solutionsWithObstacles = NQueens::solveNQueensWithObstacles(n5, obstacles5);
    cout << "\n2. 有障碍物的N皇后问题:" << endl;
    cout << "n = " << n5 << " 的解决方案数量: " << solutionsWithObstacles.size() << endl;
    for (int i = 0; i < solutionsWithObstacles.size(); i++) {
        cout << "  解决方案 " << (i + 1) << ":" << endl;
        for (const string& row : solutionsWithObstacles[i]) {
            cout << "    " << row << endl;
        }
    }

    // 测试双皇后问题
    int n6 = 5;
    long long twoQueensCount = NQueens::countTwoQueens(n6);
    cout << "\n3. 双皇后问题:" << endl;
    cout << "在" << n6 << "×" << n6 << "棋盘上放置2个互不攻击的皇后，组合数: " << twoQueensCount << endl;

    // 测试皇后覆盖问题
    int n7 = 4, k7 = 4;
    bool canCover = NQueens::canCoverBoard(n7, k7);
    cout << "\n4. 皇后覆盖问题:" << endl;
    cout << "使用" << k7 << "个皇后" << (canCover ? "能" : "不能") << "覆盖" << n7 << "×" << n7 << "的棋盘" << endl;

    // 测试非递归解法
    int n8 = 8;
    int iterativeCount = NQueens::totalNQueensIterative(n8);
    cout << "\n5. 非递归解法:" << endl;
    cout << n8 << "皇后解决方案数: " << iterativeCount << endl;
    cout << "验证是否与递归解法一致: " << (iterativeCount == NQueens::totalNQueens(n8)) << endl;
```

```
    // 测试位运算优化解法
    int n9 = 12;
    cout << "\n6. 位运算优化解法性能测试:" << endl;
    auto start2 = chrono::high_resolution_clock::now();
    int bitmaskCount = NQueens::totalNQueensBitmask(n9);
    auto end2 = chrono::high_resolution_clock::now();
    auto duration2 = chrono::duration_cast<chrono::milliseconds>(end2 - start2);
    cout << n9 << "皇后解决方案数: " << bitmaskCount << ", 耗时: " << duration.count() << "ms" <<
endl;

    return 0;
}
```

==========

文件: NQueens.java

==========

```
import java.util.*;

// N 皇后问题
// 测试链接 : https://leetcode.cn/problems/n-queens-ii/
public class NQueens {

    // 用数组表示路径实现的 N 皇后问题，不推荐
    // 时间复杂度: O(N!)，因为对于每个皇后，我们尝试 N 列，然后(N-1)列，以此类推
    // 空间复杂度: O(N)，递归栈深度为 N，path 数组大小为 N
    public static int totalNQueens1(int n) {
        // 输入校验：n 必须为正整数
        if (n < 1) {
            return 0;
        }
        // 创建一个数组 path，用来记录每一行皇后所在的列位置
        // path[i]表示第 i 行的皇后放在了第 path[i]列
        return f1(0, new int[n], n);
    }

    // 递归函数：在第 i 行放置皇后
    // i : 当前来到的行
    // path : 0...i-1 行的皇后，都摆在了哪些列
    // n : 一共有多少行
    public static int f1(int i, int[] path, int n) {
        // 递归终止条件：如果已经处理完所有行，说明找到了一个有效解
        if (i == n) {
```

```
            return 1;
        }
        int res = 0;
        // 尝试当前行 i 的所有列
        for (int j = 0; j < n; j++) {
            // 检查在第 i 行第 j 列放置皇后是否有效（不与之前放置的皇后冲突）
            if (isValid(path, i, j)) {
                // 记录第 i 行皇后放在第 j 列
                path[i] = j;
                // 递归处理下一行，并累加返回的结果
                res += f1(i + 1, path, n);
            }
        }
        return res;
    }

    // 检查在 i 行 j 列放置皇后是否有效
    // path: 前 i 行皇后的列位置
    // i: 当前行
    // j: 当前列
    public static boolean isValid(int[] path, int i, int j) {
        // 检查前面已经放置的皇后是否与当前位置冲突
        for (int k = 0; k < i; k++) {
            // 冲突条件：
            // 1. 同列：path[k] == j
            // 2. 同对角线：行差的绝对值 == 列差的绝对值，即 Math.abs(path[k] - j) == Math.abs(k -
i)
            if (path[k] == j || Math.abs(path[k] - j) == Math.abs(k - i)) {
                return false;
            }
        }
        return true;
    }

    // 使用位运算优化的 N 皇后问题
    // 时间复杂度: O(N!)，但常数项更小
    // 空间复杂度: O(N)，递归栈深度为 N
    public static int totalNQueens2(int n) {
        // 输入校验：n 必须为正整数且不超过 32（int 的位数）
        if (n < 1 || n > 32) {
            return 0;
        }
        // 限制在 n 位内，例如 n=4 时，limit=1111(二进制)
```

```
    int limit = n == 32 ? -1 : (1 << n) - 1;
    // 调用递归函数求解
    return f2(limit, 0, 0, 0);
}

// 位运算递归函数
// limit : 限制位，表示棋盘大小
// colLim : 列限制，表示哪些列已被占用
// leftDiaLim : 左对角线限制
// rightDiaLim : 右对角线限制
public static int f2(int limit, int colLim, int leftDiaLim, int rightDiaLim) {
    // 递归终止条件：所有列都放置了皇后
    if (colLim == limit) {
        return 1;
    }
    // 当前可以放置的位置
    // colLim | leftDiaLim | rightDiaLim 表示所有不能放置皇后的位置
    // ~(colLim | leftDiaLim | rightDiaLim) 表示所有可以放置皇后的位置（可能超出棋盘范围）
    // limit & (~(colLim | leftDiaLim | rightDiaLim)) 表示在棋盘范围内可以放置皇后的位置
    int pos = limit & (~(colLim | leftDiaLim | rightDiaLim));
    int res = 0;
    // 遍历所有可以放置皇后的位置
    while (pos != 0) {
        // 取最右边的 1，表示选择在该位置放置皇后
        // pos & (~pos + 1) 等价于 pos & -pos
        int mostRightOne = pos & (~pos + 1);
        // 从 pos 中移除 mostRightOne 位置的 1
        pos -= mostRightOne;
        // 递归处理下一行
        // colLim | mostRightOne：更新列的占用情况
        // (leftDiaLim | mostRightOne) << 1：更新右上->左下对角线的占用情况
        // (rightDiaLim | mostRightOne) >>> 1：更新左上->右下对角线的占用情况
        res += f2(limit,
                  colLim | mostRightOne,
                  (leftDiaLim | mostRightOne) << 1,
                  (rightDiaLim | mostRightOne) >>> 1);
    }
    return res;
}

public static void main(String[] args) {
    System.out.println("测试开始");
```

```
        // 测试 14 皇后问题
        int n = 14;
        System.out.println("解决" + n + "皇后问题");

        long start1 = System.currentTimeMillis();
        int ans1 = totalNQueens1(n);
        long end1 = System.currentTimeMillis();
        System.out.println("方法 1 答案 : " + ans1);
        System.out.println("方法 1 运行时间 : " + (end1 - start1) + " 毫秒");

        long start2 = System.currentTimeMillis();
        int ans2 = totalNQueens2(n);
        long end2 = System.currentTimeMillis();
        System.out.println("方法 2 答案 : " + ans2);
        System.out.println("方法 2 运行时间 : " + (end2 - start2) + " 毫秒");

        System.out.println("测试结束");
        System.out.println("只有位运算的版本，才能 10 秒内跑完 16 皇后问题的求解过程");
    }
}
```

==========

文件: NQueens.py

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

"""
N 皇后问题 Python 实现

N 皇后问题是一个经典的回溯算法问题，研究的是如何将 n 个皇后放置在 n×n 的棋盘上，
并且使皇后彼此之间不能相互攻击。

按照国际象棋的规则，皇后可以攻击与之处在同一行或同一列或同一斜线上的棋子。

核心知识点：
1. 问题分析：
   - 任意两个皇后不能在同一行
   - 任意两个皇后不能在同一列
   - 任意两个皇后不能在同一条对角线上

2. 解法思路：
```

- 方法一：基于数组的回溯实现
  * 使用一个数组 path 记录每行皇后所在的列位置
  * 通过递归逐行尝试放置皇后
  * 对每个位置检查是否与之前放置的皇后冲突
- 方法二：基于位运算的优化实现（推荐）
  * 使用位运算表示皇后的位置和约束条件
  * 通过位运算快速判断可放置位置
  * 效率远高于方法一

3. 约束条件判断：
   - 对于位置(i,j)和之前放置的皇后(k, path[k])，冲突条件为：
     * 同列：j == path[k]
     * 同对角线：abs(i-k) == abs(j-path[k])

   - 使用位运算时：
     * 列约束：用一个整数的二进制位表示各列是否被占用
     * 对角线约束：用两个整数分别表示两个方向的对角线是否被占用

算法复杂度分析：
- 时间复杂度：两种方法均为 O(N!)，因为对于第 1 个皇后有 N 种选择，第 2 个有 N-1 种选择，以此类推
- 空间复杂度：递归栈深度为 N，所以空间复杂度为 O(N)

工程化考虑：
- 异常处理：输入校验，检查 n 是否为正整数
- 性能优化：位运算优化，大幅提升性能
- 代码可读性：函数命名清晰，添加详细注释
"""

```
class NQueens:
    """
    N 皇后问题求解类
    """

    @staticmethod
    def total_n_queens1(n):
        """
        方法 1：基于数组的回溯实现
        时间复杂度: O(N!)
        空间复杂度: O(N)

        :param n: 皇后数量
        :return: 解法数量
```

```
    """
    # 输入校验：n 必须为正整数
    if n < 1:
        return 0

    def f1(i, path):
        """
        递归函数：在第 i 行放置皇后

        :param i: 当前行
        :param path: 前 i 行皇后的列位置
        :return: 解法数量
        """
        # 递归终止条件：所有行都已经放置了皇后
        if i == n:
            return 1

        ans = 0
        # 尝试在当前行的每一列放置皇后
        for j in range(n):
            # 检查当前位置是否合法（不与之前放置的皇后冲突）
            if NQueens._check(path, i, j):
                # 在第 i 行第 j 列放置皇后
                path[i] = j
                # 递归处理下一行
                ans += f1(i + 1, path)
        return ans

    # 创建一个数组 path，用来记录每一行皇后所在的列位置
    # path[i]表示第 i 行的皇后放在了第 path[i]列
    path = [0] * n
    return f1(0, path)

@staticmethod
def _check(path, i, j):
    """
    检查在第 i 行第 j 列放置皇后是否合法

    :param path: 前 i 行皇后的列位置
    :param i: 当前行
    :param j: 当前列
    :return: 是否合法
    """
```

```
        # 检查之前放置的皇后是否与当前位置冲突
        for k in range(i):
            # 冲突条件：
            # 1. 同列：j == path[k]
            # 2. 同对角线：行差的绝对值 == 列差的绝对值
            if j == path[k] or abs(i - k) == abs(j - path[k]):
                return False
        return True

    @staticmethod
    def total_n_queens2(n):
        """
        方法 2：基于位运算的优化实现
        时间复杂度: O(N!)，但实际运行效率远高于方法 1
        空间复杂度: O(N)

        :param n: 皇后数量
        :return: 解法数量
        """
        # 输入校验：n 必须为正整数
        if n < 1:
            return 0

        def f2(limit, col, left, right):
            """
            位运算递归函数

            :param limit: 限制位，表示棋盘大小
            :param col: 列限制，表示哪些列已被占用
            :param left: 左对角线限制
            :param right: 右对角线限制
            :return: 解法数量
            """
            # 递归终止条件：所有列都放置了皇后
            if col == limit:
                # 所有皇后放完了！
                return 1

            # 总限制：不能放置皇后的位置
            ban = col | left | right
            # 可以放置皇后的位置
            candidate = limit & (~ban)
            # 放置皇后的尝试！
```

```
        # 一共有多少有效的方法
        ans = 0
        # 遍历所有可以放置皇后的位置
        while candidate != 0:
            # 提取出最右侧的 1，表示选择在该位置放置皇后
            place = candidate & (-candidate)
            # 从 candidate 中移除已选择的位置
            candidate ^= place
            # 递归处理下一行
            # col | place：更新列的占用情况
            # (left | place) >> 1：更新右上->左下对角线的占用情况
            # (right | place) << 1：更新左上->右下对角线的占用情况
            ans += f2(limit, col | place, (left | place) >> 1, (right | place) << 1)
        return ans

    # limit 表示棋盘的限制，比如 n=4 时，limit=1111(二进制)，表示 4 列
    limit = (1 << n) - 1
    return f2(limit, 0, 0, 0)

@staticmethod
def solve_n_queens(n):
    """
    LeetCode 51. N 皇后问题 - 返回所有可能的解决方案
    题目链接：https://leetcode.cn/problems/n-queens/

    时间复杂度：O(N!)
    空间复杂度：O(N)

    :param n: 皇后数量
    :return: 所有解法的列表
    """
    def backtrack(solutions, queens, n, row, cols, diag1, diag2):
        """
        回溯函数：逐行放置皇后

        :param solutions: 所有解法的列表
        :param queens: 皇后位置数组
        :param n: 皇后数量
        :param row: 当前行
        :param cols: 列占用情况
        :param diag1: 主对角线占用情况
        :param diag2: 副对角线占用情况
        """
```

```
        # 递归终止条件：所有行都已放置皇后
        if row == n:
            # 根据 queens 数组构造棋盘
            board = NQueens._generate_board(queens, n)
            solutions.append(board)
            return

        # 在当前行尝试每一列
        for i in range(n):
            # 检查列是否被占用
            if i in cols:
                continue
            # 检查主对角线是否被占用
            d1 = row - i
            if d1 in diag1:
                continue
            # 检查副对角线是否被占用
            d2 = row + i
            if d2 in diag2:
                continue

            # 在第 row 行第 i 列放置皇后
            queens[row] = i
            cols.add(i)
            diag1.add(d1)
            diag2.add(d2)

            # 递归处理下一行
            backtrack(solutions, queens, n, row + 1, cols, diag1, diag2)

            # 回溯，恢复状态
            queens[row] = -1
            cols.remove(i)
            diag1.remove(d1)
            diag2.remove(d2)

    solutions = []
    queens = [-1] * n
    cols = set()
    diag1 = set()
    diag2 = set()
    backtrack(solutions, queens, n, 0, cols, diag1, diag2)
    return solutions
```

```
@staticmethod
def _generate_board(queens, n):
    """
    根据皇后位置生成棋盘

    :param queens: 皇后位置数组
    :param n: 棋盘大小
    :return: 棋盘表示
    """
    board = []
    for i in range(n):
        row = ['.'] * n
        # 在皇后所在位置放置'Q'
        row[queens[i]] = 'Q'
        board.append(''.join(row))
    return board

@staticmethod
def total_n_queens(n):
    """
    LeetCode 52. N皇后计数问题
    题目链接：https://leetcode.cn/problems/n-queens-ii/

    时间复杂度：O(N!)
    空间复杂度：O(N)

    :param n: 皇后数量
    :return: 解法数量
    """
    # 直接使用已实现的方法
    return NQueens.total_n_queens2(n)

@staticmethod
def queens_attack(n, k, r_q, c_q, obstacles):
    """
    HackerRank Queen's Attack II 问题
    题目链接：https://www.hackerrank.com/challenges/queens-attack-2/problem

    题目描述：
    在一个 n×n 的棋盘上有一个皇后和若干障碍物，计算皇后能攻击多少个格子。
    皇后可以攻击同一行、同一列、同一对角线上的格子，但会被障碍物阻挡。
```

参数说明：
n: 棋盘大小
k: 障碍物数量
r_q: 皇后行位置（1-based）
c_q: 皇后列位置（1-based）
obstacles: 障碍物位置列表

示例：
输入：n=5, k=3, r_q=4, c_q=3, obstacles=[[5,5],[4,2],[2,3]]
输出：10

时间复杂度：O(k)
空间复杂度：O(k)
"""
```
# 将障碍物位置存储在集合中，便于快速查找
obstacle_set = set()
for obstacle in obstacles:
    obstacle_set.add((obstacle[0], obstacle[1]))

# 8 个方向的移动向量：上、下、左、右、左上、右上、左下、右下
directions = [
    (-1, 0), (1, 0), (0, -1), (0, 1),  # 上下左右
    (-1, -1), (-1, 1), (1, -1), (1, 1)  # 四个对角线方向
]

count = 0

# 对每个方向计算能攻击的格子数
for dx, dy in directions:
    # 从皇后位置开始，沿着当前方向移动
    x, y = r_q, c_q

    while True:
        # 移动到下一个位置
        x += dx
        y += dy

        # 检查是否越界
        if x < 1 or x > n or y < 1 or y > n:
            break

        # 检查是否有障碍物
        if (x, y) in obstacle_set:
```

```
                    break

                # 如果没有障碍物且未越界，则可以攻击这个格子
                count += 1

        return count

    @staticmethod
    def chess_board_problem(board, k):
        """
        POJ 1321 棋盘问题
        题目链接：http://poj.org/problem?id=1321

        题目描述：
        在一个给定形状的棋盘（形状可能是不规则的）上面摆放棋子，棋子没有区别。
        要求摆放时任意的两个棋子不能放在棋盘中的同一行或者同一列，请编程求解对于给定形状和大小的棋盘，
        摆放 k 个棋子的所有可行的摆放方案 C。

        参数说明：
        board: 棋盘，'#'表示可放置棋子的位置，'.'表示不可放置棋子的位置
        k: 需要放置的棋子数量

        示例：
        输入：
        board = ["#.",".#"]
        k = 1
        输出：2

        时间复杂度：O(2^n)
        空间复杂度：O(n)
        """
        n = len(board)

        def dfs(row, placed, used_cols):
            """
            深度优先搜索解决棋盘问题

            :param row: 当前行
            :param placed: 已放置棋子数
            :param used_cols: 已使用的列
            :return: 方案数
            """
```

```
        # 如果已经放置了 k 个棋子，找到一种方案
        if placed == k:
            return 1

        # 如果已经搜索完所有行，但还未放置 k 个棋子
        if row == n:
            return 0

        count = 0

        # 不在当前行放置棋子
        count += dfs(row + 1, placed, used_cols)

        # 在当前行尝试放置棋子
        for col in range(n):
            # 检查当前位置是否可以放置棋子
            if board[row][col] == '#' and col not in used_cols:
                # 放置棋子
                used_cols.add(col)
                count += dfs(row + 1, placed + 1, used_cols)
                # 回溯
                used_cols.remove(col)

        return count

    return dfs(0, 0, set())

@staticmethod
def eight_queens_with_existing(existing_queens):
    """
    Aizu ALDS1_13_A 8 Queens Problem（部分皇后位置已知）
    题目链接：https://onlinejudge.u-aizu.ac.jp/courses/lesson/1/ALDS1/all/ALDS1_13_A

    题目描述：
    8 皇后问题，但部分皇后的位置已经确定，需要完成剩余皇后的放置。

    参数说明：
    existing_queens: 已知皇后的位置，格式为[row, col]

    示例：
    输入：existing_queens = [[0,0],[1,1]]
    输出：是否能完成 8 皇后布局
```

```
时间复杂度：O(N!)
空间复杂度：O(N)
"""
queens = [-1] * 8

# 设置已知皇后位置
for pos in existing_queens:
    queens[pos[0]] = pos[1]

def is_valid(row):
    """检查到第 row 行为止的皇后布局是否有效"""
    for i in range(row):
        # 检查列冲突
        if queens[i] == queens[row]:
            return False
        # 检查对角线冲突
        if abs(i - row) == abs(queens[i] - queens[row]):
            return False
    return True

def solve(row):
    """
    递归解决 8 皇后问题（部分位置已知）

    :param row: 当前行
    :return: 是否能完成布局
    """
    if row == 8:
        return True  # 所有皇后都已放置

    # 如果当前行已经有皇后，直接处理下一行
    if queens[row] != -1:
        if is_valid(row):
            return solve(row + 1)
        else:
            return False

    # 尝试在当前行的每一列放置皇后
    for col in range(8):
        queens[row] = col
        if is_valid(row):
            if solve(row + 1):
                return True
```

```
                queens[row] = -1  # 回溯

        return False

    return solve(0)

@staticmethod
def solve_k_queens(n, k):
    """
    变种题目1：多皇后问题 - 在n×n棋盘上放置k个皇后，使得它们互不攻击
    平台：POJ类似题目，常见于各大OJ的组合数学问题
    思路：使用回溯法，尝试在每一行放置皇后，但只需放置k个
    时间复杂度：O(N×N!)，其中N是棋盘大小
    空间复杂度：O(N)，递归栈和标记数组的空间

    参数:
    n: 棋盘大小
    k: 需要放置的皇后数量

    返回:
    满足条件的放置方案数
    """
    # 边界条件检查
    if k < 0 or k > n or n <= 0:
        return 1 if k == 0 else 0  # 放置0个皇后只有一种方式

    # 标记数组
    cols = [False] * n        # 列是否被占用
    diag1 = [False] * (2 * n)  # 左上到右下对角线
    diag2 = [False] * (2 * n)  # 右上到左下对角线

    def backtrack_k_queens(row, placed):
        """
        回溯函数：放置k个皇后

        :param row: 当前行
        :param placed: 已放置皇后数
        :return: 方案数
        """
        # 已经放置了k个皇后，找到一个有效解
        if placed == k:
            return 1
```

```
        # 已经处理完所有行但还没放够 k 个皇后
        if row == n:
            return 0

        count = 0

        # 尝试在当前行放置皇后
        for col in range(n):
            # 计算对角线索引
            d1 = row - col + n  # 避免负数
            d2 = row + col

            # 检查是否可以放置皇后
            if not cols[col] and not diag1[d1] and not diag2[d2]:
                # 放置皇后
                cols[col] = True
                diag1[d1] = True
                diag2[d2] = True

                # 递归到下一行，已放置皇后数+1
                count += backtrack_k_queens(row + 1, placed + 1)

                # 回溯，撤销放置
                cols[col] = False
                diag1[d1] = False
                diag2[d2] = False

        # 尝试在当前行不放置皇后，直接到下一行
        count += backtrack_k_queens(row + 1, placed)

        return count

    return backtrack_k_queens(0, 0)

@staticmethod
def solve_n_queens_with_obstacles(n, obstacles):
    """
    变种题目 2：有障碍物的 N 皇后问题 - 某些格子不能放置皇后
    平台：类似 LeetCode 51 题的扩展，常见于面试题
    思路：在标准 N 皇后问题的基础上，增加对障碍物的检查
    时间复杂度：O(N!)
    空间复杂度：O(N)
```

```
参数：
n: 棋盘大小
obstacles: 障碍物位置列表，格式为[[行, 列], ...]

返回：
所有满足条件的棋盘布局
"""
solutions = []

# 将障碍物转换为集合，方便快速查询
obstacle_set = set()
for obstacle in obstacles:
    # 假设障碍物坐标从0开始
    obstacle_set.add((obstacle[0], obstacle[1]))

# 初始化标记数组
cols = [False] * n
diag1 = [False] * (2 * n)
diag2 = [False] * (2 * n)

# 初始化棋盘表示
board = [['.' for _ in range(n)] for _ in range(n)]

def backtrack_n_queens_with_obstacles(row):
    """
    回溯函数：在有障碍物的棋盘上放置皇后

    :param row: 当前行
    """
    if row == n:
        # 找到一个解决方案，转换为字符串表示
        solution = [''.join(row) for row in board]
        solutions.append(solution)
        return

    for col in range(n):
        # 检查当前位置是否是障碍物
        if (row, col) in obstacle_set:
            continue

        # 检查是否可以放置皇后
        d1 = row - col + n
        d2 = row + col
```

```
            if not cols[col] and not diag1[d1] and not diag2[d2]:
                # 放置皇后
                board[row][col] = 'Q'
                cols[col] = True
                diag1[d1] = True
                diag2[d2] = True

                # 递归到下一行
                backtrack_n_queens_with_obstacles(row + 1)

                # 回溯
                board[row][col] = '.'
                cols[col] = False
                diag1[d1] = False
                diag2[d2] = False

    backtrack_n_queens_with_obstacles(0)
    return solutions

@staticmethod
def count_two_queens(n):
    """
    变种题目 3：双皇后问题 - 计算两个皇后互不攻击的位置组合数
    平台：CodeChef、HackerEarth 等平台常见题目
    思路：枚举第一个皇后的位置，然后计算第二个皇后的合法位置数
    时间复杂度：O(N²)
    空间复杂度：O(1)

    参数：
    n: 棋盘大小

    返回：
    满足条件的位置组合数
    """
    # 边界条件检查
    if n < 2:
        return 0  # 棋盘太小，无法放置两个皇后

    total = 0
    # 枚举第一个皇后的位置
    for r1 in range(n):
        for c1 in range(n):
            # 计算第二个皇后的合法位置数
```

```
            valid_positions = 0
            for r2 in range(n):
                for c2 in range(n):
                    # 不能是同一个位置
                    if r1 == r2 and c1 == c2:
                        continue
                    # 检查是否在同一行、同一列或同一对角线
                    is_same_row = (r1 == r2)
                    is_same_col = (c1 == c2)
                    is_same_diag = (abs(r1 - r2) == abs(c1 - c2))

                    if not is_same_row and not is_same_col and not is_same_diag:
                        valid_positions += 1
            total += valid_positions

        # 因为每个组合被计算了两次（Q1在(r1,c1)和Q2在(r2,c2)与Q1在(r2,c2)和Q2在(r1,c1)），所以要除以2
        return total // 2

    @staticmethod
    def can_cover_board(n, k):
        """
        变种题目4：皇后覆盖问题 - 检查k个皇后是否能覆盖整个棋盘
        平台：类似UVa OJ的组合优化问题
        思路：放置k个皇后，然后检查棋盘是否被完全覆盖
        时间复杂度：O(N^k)，其中k是皇后数量
        空间复杂度：O(N)

        参数:
        n: 棋盘大小
        k: 皇后数量

        返回:
        是否能覆盖整个棋盘
        """
        # 边界条件检查
        if k <= 0 or n <= 0:
            return k == 0 and n == 0  # 空棋盘不需要皇后覆盖

        # 棋盘是否被覆盖
        covered = [[False for _ in range(n)] for _ in range(n)]

        def backtrack_cover_board(row, col, placed):
```

```
    """
    回溯函数：放置皇后并检查覆盖情况

    :param row: 当前行
    :param col: 当前列
    :param placed: 已放置皇后数
    :return: 是否能覆盖整个棋盘
    """
    # 已经放置了 k 个皇后，检查是否覆盖了整个棋盘
    if placed == k:
        for i in range(n):
            for j in range(n):
                if not covered[i][j]:
                    return False
        return True

    # 遍历所有可能的放置位置
    for i in range(row, n):
        start_col = col if i == row else 0
        for j in range(start_col, n):
            # 记录当前覆盖状态，用于回溯
            old_covered = [row.copy() for row in covered]

            # 放置皇后并标记覆盖区域
            new_covered = mark_coverage(i, j)

            # 更新覆盖状态
            for x in range(n):
                for y in range(n):
                    covered[x][y] = new_covered[x][y]

            # 递归放置下一个皇后
            if backtrack_cover_board(i, j + 1, placed + 1):
                return True

            # 回溯
            for x in range(n):
                for y in range(n):
                    covered[x][y] = old_covered[x][y]

    return False

def mark_coverage(r, c):
```

```
    """
    标记皇后覆盖的区域

    :param r: 皇后行位置
    :param c: 皇后列位置
    :return: 覆盖状态
    """
    # 创建副本用于回溯
    temp_covered = [row.copy() for row in covered]

    # 标记同一行
    for j in range(n):
        temp_covered[r][j] = True

    # 标记同一列
    for i in range(n):
        temp_covered[i][c] = True

    # 标记左上到右下对角线
    i, j = r, c
    while i >= 0 and j >= 0:
        temp_covered[i][j] = True
        i -= 1
        j -= 1
    i, j = r + 1, c + 1
    while i < n and j < n:
        temp_covered[i][j] = True
        i += 1
        j += 1

    # 标记右上到左下对角线
    i, j = r, c
    while i >= 0 and j < n:
        temp_covered[i][j] = True
        i -= 1
        j += 1
    i, j = r + 1, c - 1
    while i < n and j >= 0:
        temp_covered[i][j] = True
        i += 1
        j -= 1

    return temp_covered
```

```
    return backtrack_cover_board(0, 0, 0)

@staticmethod
def total_n_queens_iterative(n):
    """
    变种题目 5：N 皇后问题的非递归解法 - 用于理解递归与非递归的差异
    平台：算法教学中常见的变体
    思路：使用栈模拟递归过程
    时间复杂度：O(N!)
    空间复杂度：O(N)

    参数：
    n: 棋盘大小

    返回：
    解决方案数量
    """
    if n <= 0:
        return 0

    count = 0
    # 记录每一行皇后的列位置
    queens = [-1] * n

    row = 0  # 当前处理的行
    col = 0  # 当前尝试的列

    while row >= 0:
        # 尝试在当前行放置皇后
        placed_in_row = False
        while col < n:
            # 检查是否可以安全放置
            safe = True
            for i in range(row):
                # 检查列冲突
                if queens[i] == col:
                    safe = False
                    break
                # 检查对角线冲突
                if abs(i - row) == abs(queens[i] - col):
                    safe = False
                    break
```

```
            if safe:
                # 放置皇后
                queens[row] = col
                row += 1  # 移动到下一行
                col = 0   # 从第一列开始尝试
                placed_in_row = True
                break     # 跳出当前列循环
            col += 1  # 尝试下一列

        # 如果当前行所有列都不能放置皇后，回溯
        if not placed_in_row:
            row -= 1  # 回溯到上一行
            if row >= 0:
                col = queens[row] + 1  # 从上一行皇后的下一列开始尝试
                queens[row] = -1       # 移除上一行的皇后
        elif row == n:
            # 找到一个解决方案
            count += 1
            row -= 1  # 回溯寻找下一个解决方案
            if row >= 0:
                col = queens[row] + 1  # 从上一行皇后的下一列开始尝试
                queens[row] = -1       # 移除上一行的皇后

    return count

@staticmethod
def total_n_queens_bitmask(n):
    """
    变种题目 6：位运算优化的 N 皇后解法 - 更高效的实现
    平台：各大算法平台的优化版本
    思路：使用位运算来表示和检查冲突
    时间复杂度：O(N!)
    空间复杂度：O(N)

    参数：
    n: 棋盘大小

    返回：
    解决方案数量
    """
    if n <= 0:
        return 0
```

```
        # 预处理：创建位掩码表示棋盘大小
        limit = (1 << n) - 1  # 例如 n=4 时，limit=0b1111

        def backtrack_bitmask(col_mask, diag1_mask, diag2_mask):
            """
            位运算回溯函数

            :param col_mask: 列占用掩码
            :param diag1_mask: 主对角线占用掩码
            :param diag2_mask: 副对角线占用掩码
            :return: 解决方案数量
            """
            # 所有列都放置了皇后
            if col_mask == limit:
                return 1

            # 计算所有可以放置皇后的位置
            available_pos = limit & (~(col_mask | diag1_mask | diag2_mask))
            count = 0

            # 尝试所有可用位置
            while available_pos != 0:
                # 取出最右边的可用位置
                pos = available_pos & (-available_pos)
                # 移除已选择的位置
                available_pos &= (available_pos - 1)

                # 递归处理下一行
                count += backtrack_bitmask(
                    col_mask | pos,
                    (diag1_mask | pos) << 1,
                    (diag2_mask | pos) >> 1
                )

            return count

        return backtrack_bitmask(0, 0, 0)


def main():
    n = 14
    print("测试开始")
```

```
print(f"解决{n}皇后问题")

import time
start = time.time()
print(f"方法 1 答案 : {NQueens.total_n_queens1(n)}")
end = time.time()
print(f"方法 1 运行时间 : {(end - start) * 1000:.0f} 毫秒")

start = time.time()
print(f"方法 2 答案 : {NQueens.total_n_queens2(n)}")
end = time.time()
print(f"方法 2 运行时间 : {(end - start) * 1000:.0f} 毫秒")
print("测试结束")

print("=======")
print("只有位运算的版本，才能 10 秒内跑完 16 皇后问题的求解过程")
start = time.time()
ans = NQueens.total_n_queens2(16)
end = time.time()
print(f"16 皇后问题的答案 : {ans}")
print(f"运行时间 : {(end - start) * 1000:.0f} 毫秒")

# 测试 LeetCode 51
print("=======")
print("LeetCode 51. N 皇后问题测试")
solutions = NQueens.solve_n_queens(4)
for i, solution in enumerate(solutions):
    print(f"解法{i + 1}:")
    for row in solution:
        print(row)
    print()

# 测试 LeetCode 52
print("LeetCode 52. N 皇后计数问题测试")
print(f"4 皇后问题的解法数：{NQueens.total_n_queens(4)}")
print(f"8 皇后问题的解法数：{NQueens.total_n_queens(8)}")

# 测试 HackerRank Queen's Attack II
print("=======")
print("HackerRank Queen's Attack II 测试")
obstacles = [[5, 5], [4, 2], [2, 3]]
print("5x5 棋盘，皇后在(4,3)，障碍物在(5,5),(4,2),(2,3)")
print(f"皇后能攻击的格子数：{NQueens.queens_attack(5, 3, 4, 3, obstacles)}")
```

```
# 测试 POJ 1321 棋盘问题
print("=======")
print("POJ 1321 棋盘问题测试")
board1 = ["#.", ".#"]
print("2x2 棋盘，可放置位置为(0,0)和(1,1)，放置 1 个棋子")
print(f"方案数：{NQueens.chess_board_problem(board1, 1)}")

board2 = [
    "...#",
    "..#.",
    ".#..",
    "#..."
]
print("4x4 棋盘，放置 4 个棋子")
print(f"方案数：{NQueens.chess_board_problem(board2, 4)}")

# 测试 Aizu ALDS1_13_A 8 Queens Problem
print("=======")
print("Aizu ALDS1_13_A 8 Queens Problem 测试")
existing_queens = [[0, 0], [1, 1]]
print("已知皇后在(0,0)和(1,1)")
print(f"能否完成 8 皇后布局：{NQueens.eight_queens_with_existing(existing_queens)}")

# 测试新增的变种题目
print("\n=======")
print("======= 变种题目测试 =======")

# 测试多皇后问题
n4, k4 = 5, 3
k_queens_count = NQueens.solve_k_queens(n4, k4)
print(f"\n1. 多皇后问题:")
print(f"在{n4}×{n4}棋盘上放置{k4}个互不攻击的皇后，方案数: {k_queens_count}")

# 测试有障碍物的 N 皇后问题
n5 = 4
obstacles5 = [[0, 0], [2, 2]]  # (0,0)和(2,2)位置有障碍物
solutions_with_obstacles = NQueens.solve_n_queens_with_obstacles(n5, obstacles5)
print(f"\n2. 有障碍物的 N 皇后问题:")
print(f"n = {n5} 的解决方案数量: {len(solutions_with_obstacles)}")
for i, solution in enumerate(solutions_with_obstacles):
    print(f"  解决方案 {i + 1}:")
    for row in solution:
```

```
            print(f"    {row}")

    # 测试双皇后问题
    n6 = 5
    two_queens_count = NQueens.count_two_queens(n6)
    print(f"\n3. 双皇后问题:")
    print(f"在{n6}×{n6}棋盘上放置 2 个互不攻击的皇后，组合数: {two_queens_count}")

    # 测试皇后覆盖问题
    n7, k7 = 4, 4
    can_cover = NQueens.can_cover_board(n7, k7)
    print(f"\n4. 皇后覆盖问题:")
    print(f"使用{k7}个皇后{'能' if can_cover else '不能'}覆盖{n7}×{n7}的棋盘")

    # 测试非递归解法
    n8 = 8
    iterative_count = NQueens.total_n_queens_iterative(n8)
    print(f"\n5. 非递归解法:")
    print(f"{n8}皇后解决方案数: {iterative_count}")
    print(f"验证是否与递归解法一致: {iterative_count == NQueens.total_n_queens(n8)}")

    # 测试位运算优化解法
    n9 = 12
    print(f"\n6. 位运算优化解法性能测试:")
    import time
    start = time.time()
    bitmask_count = NQueens.total_n_queens_bitmask(n9)
    end = time.time()
    print(f"{n9}皇后解决方案数: {bitmask_count}, 耗时: {(end - start) * 1000:.0f}ms")


if __name__ == "__main__":
    main()

==========================================
```