N 皇后问题求解实验报告

一、问题描述

N 皇后问题是经典的回溯算法应用问题，目标是在 N×N 的棋盘上放置 N 个皇后，使其互不攻击，即任意两个皇后不在同一行、同一列、同一对角线上。

二、算法实现

本实验分别实现了三种算法：

1. 基本回溯法

- 按行逐层递归，每一层枚举所有列
- 每次检查当前皇后与之前所有皇后是否冲突（同列/对角线）
- 如果该位置不冲突则保存当前状态下的解，递归处理下一行；如果冲突则撤销当前选择，继续回溯

```python
# 判断是否可以放置皇后
def is_safe(queens, row, col):  2 用法
    for r, c in enumerate(queens):
        if c == col or abs(c - col) == abs(r - row):
            return False
    return True

# 回溯法求解基本实现
def backtrack_basic(n, row, queens, solutions, find_one=False):  2 用法
    if row == n:
        solutions.append(queens[:])
        return find_one
    for col in range(n): # 遍历当前行的所有列
        if is_safe(queens, row, col): # 判断是否冲突
            queens.append(col)
            if backtrack_basic(n, row + 1, queens, solutions, find_one):
                return True
            queens.pop() # 回溯
    return False
```

2. 对称剪枝优化

N 皇后问题在棋盘布局上具有关于“中心垂直轴”的对称性。对于任意一个合法解，将其左右镜像变换后，仍然是一个合法解。

- 对称剪枝法只在第 0 行的前半列（0 ~ n//2 - 1）尝试放置皇后；后续列的解由前半列通过对称变换生成，避免重复搜索

- 镜像生成：每个原始解 queens 的第 0 行坐标 c 替换为 n-1-c，其他行保持不变；构造对应的镜像解添加到最终解集中
- 奇数情况补偿：若 N 为奇数，中心列没有对应镜像；需单独处理第 0 行皇后放在中间列的情况，避免漏解

```
# 对称性优化
def backtrack_symmetry(n, row, queens, solutions):  3 用法
    if row == n:
        solutions.append(queens[:])
        return
    cols = range(n // 2) if row == 0 else range(n)  # 剪枝关键点
    for col in cols:
        if is_safe(queens, row, col):
            queens.append(col)
            backtrack_symmetry(n, row + 1, queens, solutions)
            queens.pop()

def solve_n_queens_symmetry(n):  1 个用法
    solutions = []
    backtrack_symmetry(n,  row: 0,  queens: [], solutions)
    mirrored = []
    if n % 2 == 1: # 奇数补偿
        col = n // 2
        queens = [col] # 将第 0 行的皇后放在中间列;
        backtrack_symmetry(n,  row: 1, queens, mirrored)
    total = solutions + [ [n - 1 - c if r == 0 else c for r, c in enumerate(sol)] for sol in solutions ] + mirrored # 镜像生成
    return total
```

3. 位运算优化

- 使用整数按位表示列、主对角线、副对角线占用情况
- 利用位掩码生成当前行所有可以放皇后的位置，提取最低位可行位置
- 进行递归搜索并更新状态；通过位运算无需显式回溯数组，状态通过参数值传递

```
# 位运算优化方法（支持 n <= 32）
def solve_n_queens_bitwise(n):  1 个用法
    results = []

    def dfs(row, cols, pies, nas, state):
        if row == n:
            results.append(state[:])
            return
        bits = (~(cols | pies | nas)) & ((1 << n) - 1) # 计算可选位置
        while bits:
            p = bits & -bits  # 取最低位的1
            col = bin(p - 1).count("1")
            state.append(col)
            dfs(row + 1, cols | p, (pies | p) << 1, (nas | p) >> 1, state)
            state.pop()
            bits &= bits - 1 # 去除已尝试的位置，继续尝试其他可能

    dfs( row: 0,  cols: 0,  pies: 0,  nas: 0,  state: [])
    return results
```

三、实验设置

- 实验平台：Windows 10，Python 3.9
- 测试范围：N ∈ [4, 12]

1、 分离输入处理测试

- 皇后数量大于等于 4，且为整数
- 输出模式只能选择‘a’输出所有解，或‘1’仅输出一个解，异常值检测

```
=== N 皇后问题求解器 ===
请输入皇后数量 N (N >= 4)：2
输入必须大于等于 4
请输入皇后数量 N (N >= 4)：5.5
请输入有效整数
请输入皇后数量 N (N >= 4)：5
输出模式：输入 'a' 输出所有解，输入 '1' 仅输出一个解：b
输入无效，请重新输入 'a' 或 '1'
输出模式：输入 'a' 输出所有解，输入 '1' 仅输出一个解：1

第 1 个解：
Q....
..Q..
....Q
.Q...
...Q.

解的总数: 1
耗时: 0.0000 秒
```

2、测试用例 （N=8 只截取了第一个和最后一个解的图片）

```
=== N 皇后问题求解器 ===
请输入皇后数量 N (N >= 4)：4
输出模式：输入 'a' 输出所有解，输入 '1' 仅输出一个解：a

第 1 个解：
.Q..
...Q
Q...
..Q.

第 2 个解：
..Q.
Q...
...Q
.Q..

解的总数: 2
耗时: 0.0000 秒
```

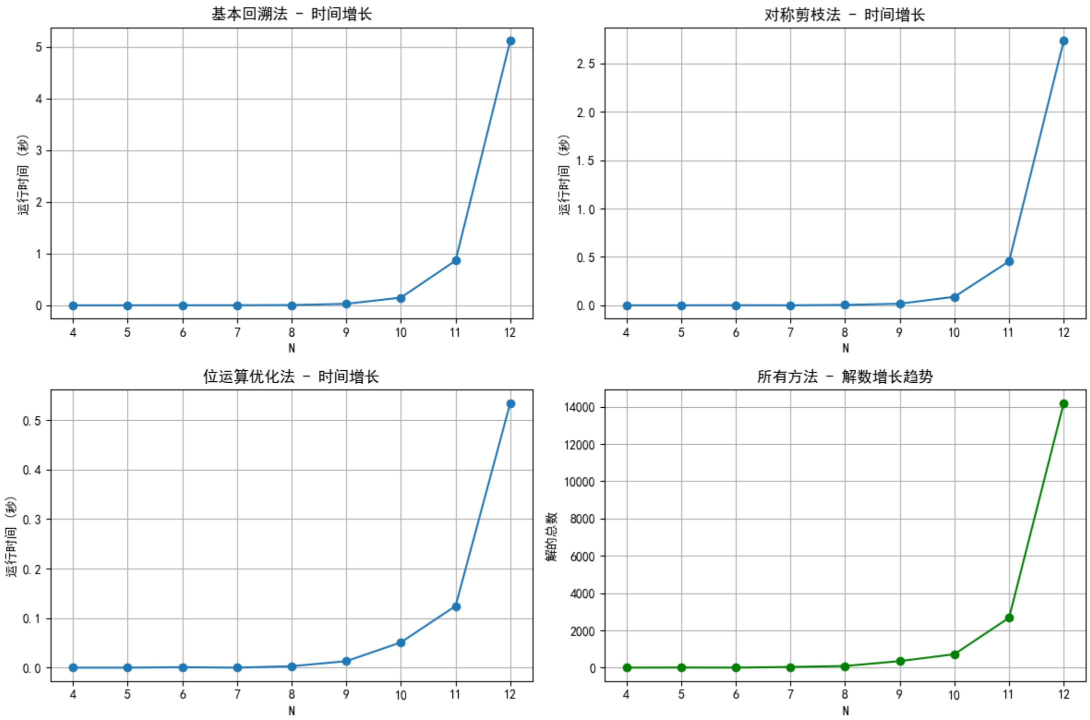
```
=== N 皇后问题求解器 ===
请输入皇后数量 N (N >= 4)：8
输出模式：输入 'a' 输出所有解，输入 '1' 仅输出一个解：a

第 1 个解：
Q.......
....Q...
.......Q
.....Q..
..Q.....
......Q.
.Q......
...Q....

第 92 个解：
.......Q
...Q....
Q.......
..Q.....
.....Q..
.Q......
......Q.
....Q...

解的总数: 92
耗时: 0.0095 秒
```

3、 时间增长曲线图




```
基本回溯法 | N=4: 解数=2, 时间=0.0000s
基本回溯法 | N=5: 解数=10, 时间=0.0000s
基本回溯法 | N=6: 解数=4, 时间=0.0010s
基本回溯法 | N=7: 解数=40, 时间=0.0010s
基本回溯法 | N=8: 解数=92, 时间=0.0050s
基本回溯法 | N=9: 解数=352, 时间=0.0285s
基本回溯法 | N=10: 解数=724, 时间=0.1486s
基本回溯法 | N=11: 解数=2680, 时间=0.8637s
基本回溯法 | N=12: 解数=14200, 时间=5.1184s
对称剪枝法 | N=4: 解数=2, 时间=0.0000s
对称剪枝法 | N=5: 解数=10, 时间=0.0000s
对称剪枝法 | N=6: 解数=4, 时间=0.0010s
对称剪枝法 | N=7: 解数=40, 时间=0.0000s
对称剪枝法 | N=8: 解数=92, 时间=0.0040s
对称剪枝法 | N=9: 解数=352, 时间=0.0165s
对称剪枝法 | N=10: 解数=724, 时间=0.0879s
对称剪枝法 | N=11: 解数=2680, 时间=0.4551s
对称剪枝法 | N=12: 解数=14200, 时间=2.7359s
位运算优化法 | N=4: 解数=2, 时间=0.0000s
位运算优化法 | N=5: 解数=10, 时间=0.0000s
位运算优化法 | N=6: 解数=4, 时间=0.0010s
位运算优化法 | N=7: 解数=40, 时间=0.0000s
位运算优化法 | N=8: 解数=92, 时间=0.0030s
位运算优化法 | N=9: 解数=352, 时间=0.0130s
位运算优化法 | N=10: 解数=724, 时间=0.0511s
位运算优化法 | N=11: 解数=2680, 时间=0.1245s
位运算优化法 | N=12: 解数=14200, 时间=0.5341s
```

（1）基本回溯法

理论分析：复杂度 O(N!)

- 每行需要尝试 N 个位置，递归深度为 N
- 实际尝试路径会因冲突检测而减少，但数量级依旧接近 N!

实验结果对比：

- 从 N=8 开始，基本回溯法时间指数级飙升；

（2）对称剪枝法

理论分析：复杂度仍为 O(N!)，但常数项下降一半（搜索空间缩小 ≈1/2）

- 因为第一行只搜索前一半列，其余通过镜像补全；
- 优化效果：在理论上无法改变指数阶复杂度，但在实际运行中将运行时间降低约 45%~50%。

实验结果对比：

- 剪枝法时间缩短为约一半，符合剪枝常数优化预期；

（3）位运算优化法

理论分析：复杂度仍为 O(N!) ，但实际路径搜索效率显著提升

- 冲突检测用位操作 O(1)完成；
- 枚举所有合法位使用按位与、位移，无需循环判断；
- 优化效果：常数项极小；实际表现远优于普通回溯。

实验结果对比：

- 位运算法的时间随 N 增长最缓慢，虽然仍为指数复杂度，但效率已提升一个数量级