import time

def is\_safe(row, col, queen\_pos):

"""检查当前位置(row, col)是否与已放置皇后冲突"""

for r, c in enumerate(queen\_pos):

*# 检查同一列或同一对角线（主对角线：行-列=常数；副对角线：行+列=常数）*

if c == col or abs(row - r) == abs(col - c):

return False

return True

def backtrack(n, row, queen\_pos, solutions, find\_all):

"""

回溯法递归函数

:param n: 棋盘大小

:param row: 当前处理的行

:param queen\_pos: 当前皇后位置列表（索引为行，值为列）

:param solutions: 存储所有解的列表

:param find\_all: 是否查找所有解

:return: 是否找到解（当find\_all=False时有效）

"""

if row == n:

solutions.append(queen\_pos[:]) *# 找到有效解，复制当前状态*

return not find\_all *# 若只需一个解则返回True停止搜索*

for col in range(n):

if is\_safe(row, col, queen\_pos):

queen\_pos.append(col) *# 放置皇后*

found = backtrack(n, row + 1, queen\_pos, solutions, find\_all)

queen\_pos.pop() *# 回溯，移除皇后*

if found and not find\_all:

return True

return False

def solve\_n\_queens(n, find\_all=True):

"""求解N皇后问题"""

solutions = []

backtrack(n, 0, [], solutions, find\_all)

return solutions

def format\_solution(queen\_pos, n):

"""将解转换为棋盘字符串表示"""

board = []

for col in queen\_pos:

row\_str = ['.'] \* n

row\_str[col] = 'Q'

board.append(''.join(row\_str))

return board

def print\_solutions(solutions, n):

"""打印所有解（控制最大输出数量）"""

max\_display = 5 *# 最大显示解的数量*

if not solutions:

print("未找到解")

return

print(f"总共有 {len(solutions)} 个解")

if len(solutions) > max\_display and n > 8:

print(f"由于解的数量较多，仅显示前{max\_display}个解")

for i, sol in enumerate(solutions[:max\_display]):

print(f"解 {i + 1}:")

board = format\_solution(sol, n)

for line in board:

print(line)

print()

if len(solutions) > max\_display and n > 8:

print(f"已隐藏 {len(solutions) - max\_display} 个解...")

def main():

"""主函数：处理用户交互"""

n = 0

while n < 4:

try:

n = int(input("请输入N (N≥4): "))

if n < 4:

print("N必须至少为4，请重新输入。")

except ValueError:

print("输入无效，请输入整数。")

choice = input("输出所有解(输入1)还是仅一个解(输入0)? ").strip()

find\_all = (choice == '1')

start\_time = time.perf\_counter()

solutions = solve\_n\_queens(n, find\_all)

elapsed = time.perf\_counter() - start\_time

if not find\_all and solutions:

print("\n找到一个解：")

board = format\_solution(solutions[0], n)

for line in board:

print(line)

print(f"求解耗时: {elapsed:.6f}秒")

else:

print\_solutions(solutions, n)

print(f"求解耗时: {elapsed:.6f}秒")

def performance\_test():

"""性能测试函数"""

print("性能测试 (N=4到12):")

print("N\t解的数量\t时间(秒)")

results = []

for n in range(4, 13):

start\_time = time.perf\_counter()

solutions = solve\_n\_queens(n, True)

elapsed = time.perf\_counter() - start\_time

results.append((n, len(solutions), elapsed))

print(f"{n}\t{len(solutions)}\t\t{elapsed:.6f}")

*# 绘制简单的时间增长曲线*

print("\n时间增长曲线:")

for n, \_, elapsed in results:

bar = '█' \* int(elapsed \* 50)

print(f"{n}: {bar} {elapsed:.6f}秒")

if \_\_name\_\_ == "\_\_main\_\_":

mode = input("运行模式: 1=交互式, 2=性能测试: ").strip()

if mode == '2':

performance\_test()

else:

main()

**算法说明**

1. **回溯算法**：
   * 逐行放置皇后，确保每行只有一个皇后
   * 使用**is\_safe()**函数检测当前位置是否与已放置皇后冲突
   * 递归尝试所有可能位置，回溯时移除无效放置
2. **冲突检测**：
   * 列冲突：检查新位置列号是否与已有皇后相同
   * 对角线冲突：检查行差绝对值是否等于列差绝对值
3. **优化**：
   * 使用隐式列检查（通过queen\_pos存储列信息）
   * 提前终止：当只需一个解时，找到即返回

**时间复杂度分析**

* **理论复杂度**：O(N!)
  + 第一行有N种选择
  + 第二行最多N-1种选择（排除冲突列）
  + 依此类推，最坏情况为N!次尝试
* **实际性能**：
  + 回溯剪枝显著减少实际搜索空间
  + 冲突检测为O(N)，但每层递归N次尝试