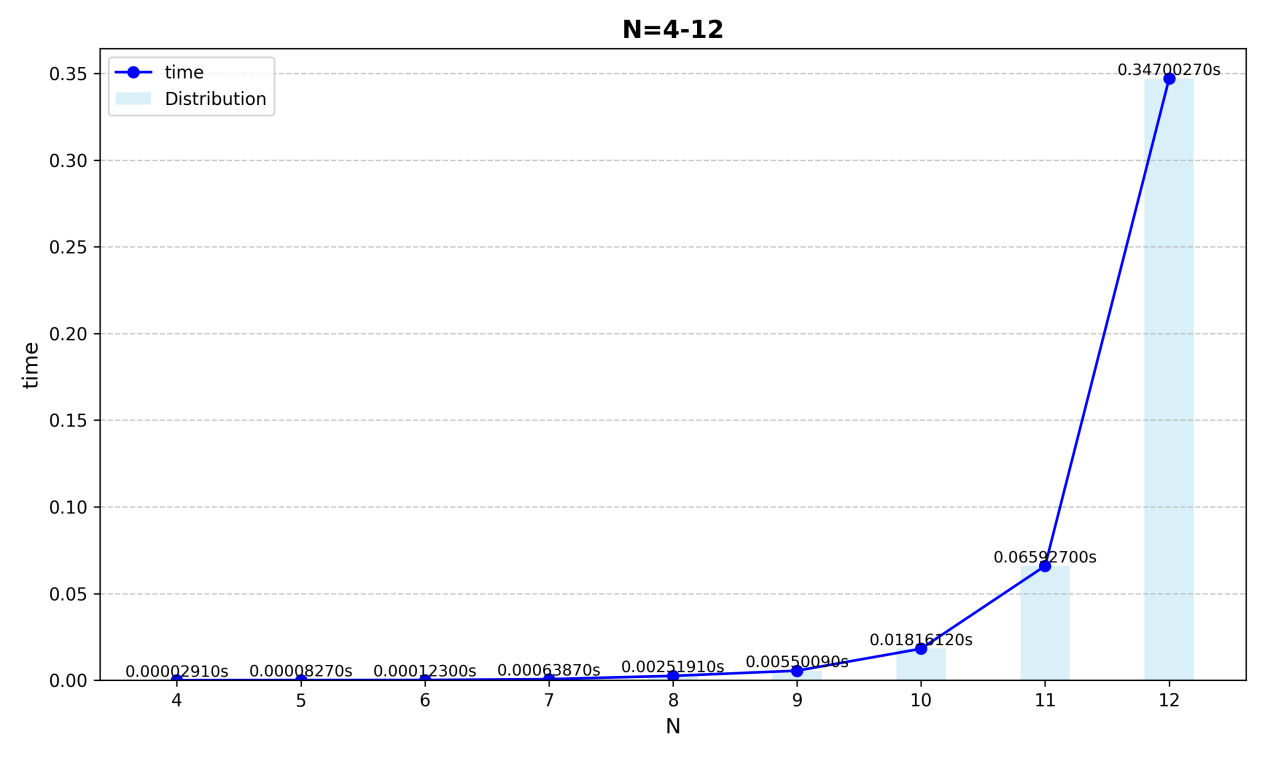
N皇后实验报告

1. **时间分析**



这是根据运行时间用python画出的图，记录了N从4到12，运行时间的变化，可以看出，随着 N 的增大，运行时间呈指数级增长，这与理论时间复杂度O(N!)相符。同时，由于剪枝优化的作用，实际运行时间比未优化的O(N!)要小得多。

1. **算法时间复杂度分析**

**（1）基础回溯法的时间复杂度（未优化）**

对于传统回溯法（代码中未被调用的 solve\_n\_queens 函数），时间复杂度为 O(N!)，原因如下：

每一行选择列：第 1 行有 N 种选择，第 2 行有 \(N-1\) 种选择（排除列冲突），依此类推。

剪枝的影响：虽然通过对角线冲突检查剪枝，但最坏情况下（如 \(N=4\)）仍接近阶乘级增长。

**（2）优化后算法的时间复杂度**

代码中实际调用的 solve\_n\_queens\_with\_pruning 函数通过位运算剪枝和对称性剪枝，大幅降低时间复杂度：

**1. 位运算剪枝**

核心优化：用位掩码（cols, diag1, diag2）快速判断列和对角线冲突，替代逐行遍历检查。

时间复杂度：每一行的可用列通过位运算快速计算，单次判断时间为 O(1)。

递归次数：每一行的选择数从 N 减少至实际可行列数，最坏情况下仍为 O(N!)，但实际效率远高于传统方法。

**2. 对称性剪枝**

优化逻辑：仅处理第一行的前半部分列，后半部分解通过镜像生成。

时间复杂度：将搜索空间缩小约 1/2或 (N+1)/2，时间复杂度降为 O(N!/2)，即 O(N! / 2)。

1. **创新点（剪枝优化）**

（1）位运算剪枝：在 solve\_n\_queens\_with\_pruning 函数中，使用位运算来快速计算可用列。通过 available = ((1 << n) - 1) & ~(cols | diag1 | diag2) ，将所有被占用的列和对角线通过位掩码进行表示，然后通过按位与操作得到可用列的掩码。这种方式相比于传统的逐列检查，大大减少了计算量，提高了算法效率。

（2）对称性剪枝：在处理第一行时，利用棋盘的对称性进行剪枝。通过 limit = (1 << (n // 2)) - 1 和 available &= limit ，只处理前半部分列，后半部分的解可以通过对称性生成。这样可以将搜索空间减少约一半或接近一半，进一步提高了算法效率。

1. **算法说明**

**基本回溯法（未使用剪枝优化）：**

对于每一行，遍历每一列，检查当前位置是否可以放置皇后。

如果可以放置，则将皇后放置在该位置，并递归到下一行继续尝试。

如果所有行都成功放置了皇后，则找到一个解，将其添加到solutions列表中。

如果在某一行没有找到合适的位置，则回溯到上一行，将上一行的皇后位置重置，继续尝试其他列。

**剪枝优化：**

使用位运算来快速计算可用列。通过available = ((1 << n) - 1) & ~(cols | diag1 | diag2)这一行代码，将所有被占用的列和对角线通过位掩码进行表示，然后通过按位与操作得到可用列的掩码。在处理第一行时，利用棋盘的对称性进行剪枝。通过limit = (1 << (n // 2)) - 1和available &= limit这两行代码，只处理前半部分列，后半部分的解可以通过对称性生成。

**其他函数：**

is\_valid函数用于检查当前位置是否可以放置皇后，通过检查列和对角线是否有冲突来判断。

get\_valid\_input函数用于获取用户输入的N值，并进行有效性检查。

print\_solution函数用于打印找到的解。

**主程序入口**

获取用户输入的N。

使用time.perf\_counter()记录程序开始时间。

调用solve\_n\_queens\_with\_pruning函数求解 N 皇后问题。

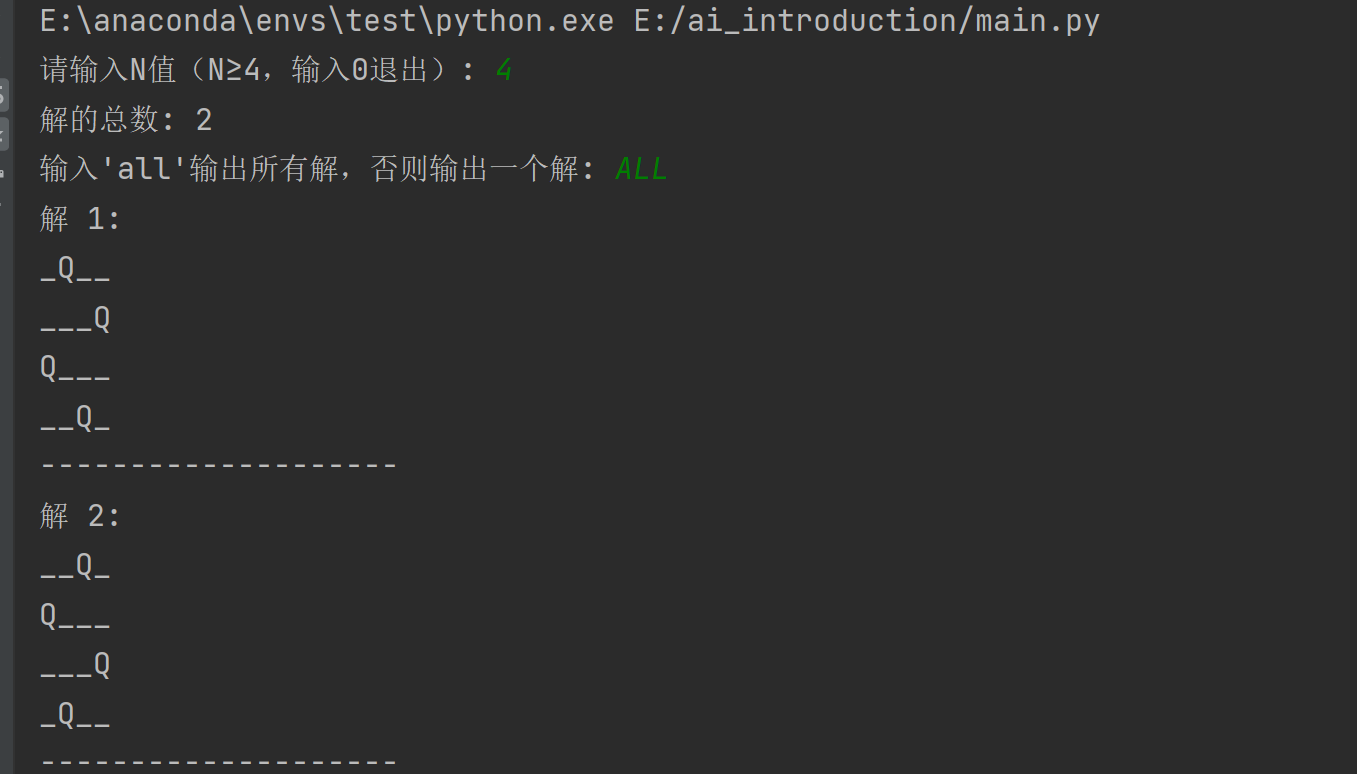
使用time.perf\_counter()记录程序结束时间，并计算运行时间。

输出解的总数和运行时间。

根据用户输入，输出所有解或一个解。

1. **测试用例截图**

N=4



N=8

