52 N-Queens II

Problem Description

The n-queens puzzle is a classic problem in computer science and math that involves placing 'n' queens on an 'n x n' chessboard in such a way that no two queens can attack each other. This means that no two queens can be in the same row, column, or diagonal. The challenge is to determine the total number of unique ways (distinct solutions) in which the 'n' queens can be placed on the board without threatening each other.

Intuition

To solve the n-queens problem, we use a backtracking algorithm. Backtracking is a systematic way to iterate through all the possible configurations of the chessboard and to "backtrack" whenever placing a queen would lead to a conflict. For each row, we try to place a queen in a valid position and then move to the next row. If we find a row where we can't place a queen without causing a conflict, we backtrack to the previous row and try a different position for the queen.

Steps

- Represent the chessboard using variables that indicate columns and diagonals that are "under attack" by queens already placed.
- Use a depth-first search (DFS) algorithm. We start from the first row and move row by row to the next, trying to place a queen in a safe column.
- Maintain three arrays `cols`, `dg`, and `udg`. `cols` tracks which columns have queens, `dg` tracks the "normal" diagonals and `udg` tracks the "anti-diagonals".
- For each recursive call (`dfs`), check each column in the current row:
  - Calculate the indexes for the diagonals based on the current row and column.
  - Check if the current column or the diagonal paths are already containing a queen (`cols[j]`, `dg[a]`, or `udg[b]`). If they are, we skip this column and continue the loop.
  - If the current column and diagonals are free, place a queen there (marking the column and diagonals as "under attack") and call `dfs` for the next row.
- When we reach a row beyond the last one (`i == n`), it means a valid configuration of queens has been placed, and we increment our solutions counter `ans`.
- The backtracking happens when we return from a `dfs` call and we "remove" the queen from that row's column and diagonals (by unmarking `cols[j]`, `dg[a]`, and `udg[b]`) before going back to try the next column.
- Once all possibilities have been explored, return `ans`, which holds the count of valid solutions.

Time Complexity

The time complexity of the function is `O(N!)`, where `N` is the input size of the chessboard (n x n). For each row, we attempt to place a queen in every column and use three arrays (`cols`, `dg`, `udg`) to check if the current cell is under attack. The DFS approach ensures that for the first queen, there are `N` possible columns to place it in, for the second queen there are `N - 1` possibilities (excluding the column and diagonals of the first queen), and so on. This sequential reduction in possibilities leads to factorial time complexity. However, due to the aggressive pruning by the `if cols[j] or dg[a] or udg[b]: continue` condition, the actual run time is significantly less than `N!`. Nonetheless, the upper bound remains factorial in the worst case.

Space Complexity

The space complexity of the function is `O(N)`. This includes:

- The `cols`, `dg`, and `udg` arrays with fixed sizes `10`, `20`, and `20`, respectively, which do not depend on the input size `N`. However, for larger boards these sizes would scale with `N` and the arrays would become `O(N)`.
- The system's call stack for the recursive function `dfs`. In the worst case, the recursion depth will be `N`, as `dfs` will be called once per row.
-