52. N-Queens II

Hard Backtracking Leetcode Link

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.

Here are the steps in the solution approach:

- 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.

This approach ensures that all potential board configurations are considered without violating the constraints of the n-queens problem. By the end of the recursive exploration, we'll have the total count of distinct solutions.

Solution Approach

The solution approach involves using depth-first search (DFS) to explore all possible placements of queens row by row, while ensuring that no queen is placed in a position where it can attack or be attacked by another queen.

Here's how the algorithm and data structures are utilized:

- **Columns, Diagonals, and Anti-Diagonals Tracking**: We use three arrays to keep track of the threats for each queen placement.
  - `cols`: A boolean array representing if a column is under attack by any queen (`True` if under attack, `False` otherwise).
  - `dg`: A boolean array representing the normal diagonals on the board. The index of a cell's normal diagonal can be obtained by `i + j` (row index plus column index).
  - `udg`: A boolean array representing the anti-diagonals on the board. The index of a cell's anti-diagonal can be obtained by `i - j + n` (row index minus column index with an offset of `n`).
- **Depth-First Search (DFS) Implementation**: This function is implemented recursively. `dfs(i)` means trying to place a queen in the `i-th` row.
  - If `i` equals `n`, it indicates that queens have been successfully placed in all rows from `0` to `n-1`, hence a valid solution. We increment the answer counter `ans` to record this solution.
  - We then iterate over each column `j` of row `i` to check if we can place a queen there.
    - For each column `j`, we calculate the indexes `a` and `b` for the normal and anti-diagonals using the formulas `i + j` and `i - j + n`, respectively.
    - Check if column `j` or either of the diagonals indexed by `a` or `b` are under attack (`cols[j]`, `dg[a]`, or `udg[b]`). If they are, we skip to the next column in the current row.
    - If none is under attack, we place a queen by marking column `j` and the corresponding diagonals as `True`, effectively denoting them as "under attack".
  - After placing a queen, the DFS algorithm recursively moves to the next row by calling `dfs(i + 1)`.
  - Once the DFS call returns (either a solution was found for that path or no solution), we backtrack by unmarking column `j` and the corresponding diagonals, thus removing the queen from the board. This opens up new possibilities for queen placements in subsequent iterations.
- **Solution Counter**: We maintain an integer `ans` to count the number of distinct solutions found.
- **Application**: We initialize the `cols`, `dg`, `udg` arrays with adequate sizes based on the maximum possible size of the chessboard (`cols` has `n` elements, `dg` and `udg` have `2n` to cover all possible diagonals). We then call `dfs(0)` to start the algorithm from the first row and continue until valid placements are found for all rows.
- **Return Value**: After the entire board is explored, dfs has been attempted from all possible columns for every row, and backtracking has occurred where possible. The variable `ans` holds the total number of valid solutions and is returned as the final result.

The use of DFS and backtracking is key in this algorithm, allowing the program to explore the entire solution space without repeating configurations or violating the constraints of the n-queens puzzle.

Example Walkthrough

Let's walk through an example of the n-queens problem for `n = 4`. We will place 4 queens on a 4×4 chessboard so that no two queens can attack each other. We'll use the steps outlined in the solution approach.

- First, initialize the `cols`, `dg`, and `udg` arrays as empty boolean arrays sized for `n = 4`. This is as `cols[4]`, `dg[8]`, and `udg[8]`.
- Start with the first row (`i = 0`) and try to place a queen in each column (`j = 0` to `n-1`), checking for conflicts with `cols`, `dg`, and `udg`.
- Explore the first column `(j = 0)`. None of the arrays are marked true, so it is safe to place a queen here.
  - Mark `cols[0]`, `dg[0+0 (i+j)]`, and `udg[0-0+4 (i-j+n)]` as true.
- Move to the next row (`i = 1`) and iterate over the columns.
  - Skip column `j = 0` because `cols[0]` is true.
  - For column `j = 1`, we check for diagonal attacks. `dg[i+j (1+1)]` and `udg[i-j+n (1-1+4)]` are not under attack, so we can place a queen here.
    - Mark `cols[1]`, `dg[1+1 (2)]`, and `udg[1-1+4 (4)]` as true.
- Move to the third row (`i = 2`). Iterate over the columns again.
  - Skip column `j = 0` and `j = 1` as `cols` and `udg` are marked true, respectively.
  - Column `j = 2` is safe, so place a queen, and mark the affected `cols`, `dg`, and `udg`.
- Move to the fourth row (`i = 3`). All columns until `j = 3` are under attack.
  - Place a queen in column `j = 3` and mark the affected `cols`, `dg`, and `udg`.
- Since placing a queen in the fourth row (i = 3) is successful and `i` now equals `n`, increment the solutions counter `ans`.
- After the solution is recorded, we backtrack from this placement to explore other potential solutions:
  - Unmark `cols[3]`, `dg[3+3]`, and `udg[3-3+4]`, and return to the third row (`i = 2`).
  - Since there are no other columns left to explore in row `i = 2`, unmark the respective `cols`, `dg`, and `udg` for the queen placed in column `j = 2`, and move back to row `i = 1`.
- Repeat the process for all the rows:
  - This involves placing the queen in a new column when possible, marking the arrays again.
  - Recursively calling `dfs(i + 1)` for the next row.
  - Backtracking when no solution is found in the current path and unmarking arrays, then returning to the previous row.
- Continue this recursive DFS and backtracking process for all rows and columns.
- The final `ans` will be the total number of valid solutions after the algorithm explores all possible placements of queens on the 4×4 board.

Following the complete exploration using DFS and backtracking, we will find that there are 2 solutions to the 4-queens problem.

Python Solution

```python
class Solution:
    def totalNQueens(self, n: int) -> int:
        # DFS function to try placing a queen on each row
        def dfs(row):
            # If all queens are placed successfully, increment the solution count
            if row == n:
                nonlocal solution_count
                solution_count += 1
                return

            # Try placing a queen in each column of the current row
            for col in range(n):
                # Calculate indices for the diagonals
                pos_diag = row + col
                neg_diag = row - col + n

                # Check if the column or the diagonals have a queen already
                if cols[col] or diag[pos_diag] or anti_diag[neg_diag]:
                    continue  # Skip if there's a conflict

                # Place the queen and mark the places as attacked
                cols[col] = diag[pos_diag] = anti_diag[neg_diag] = True
                # Recursively place queen in the next row
                dfs(row + 1)
                # Backtrack and remove the queen from the current spot
                cols[col] = diag[pos_diag] = anti_diag[neg_diag] = False

        # Arrays to keep track of attacked columns and diagonals
        cols = [False] * n  # Columns where the queens can attack
        diag = [False] * (2 * n)  # Positive diagonals (index = row + col)
        anti_diag = [False] * (2 * n)  # Negative diagonals (index = row - col + n)

        solution_count = 0  # Counter for number of valid solutions
        # Start the DFS recursion from the first row
        dfs(0)
        # Return the total number of valid solutions found
        return solution_count

```

Java Solution

```java
class Solution {
    private int boardSize;
    private int solutionsCount;
    private boolean[] columnsInUse; // marks columns that are already occupied
    private boolean[] positiveDiagonalsInUse; // marks positive diagonals that are already occupied
    private boolean[] negativeDiagonalsInUse; // marks negative diagonals that are already occupied

    public int totalNQueens(int n) {
        this.boardSize = n;
        this.columnsInUse = new boolean[n]; // assuming N will not exceed 10
        this.positiveDiagonalsInUse = new boolean[2 * n]; // range of possible values for (row + col)
        this.negativeDiagonalsInUse = new boolean[2 * n]; // range for possible values for (row - col + n)
        this.solutionsCount = 0;
        placeQueens(0);
        return solutionsCount;
    }

    // Tries to place queens on the board, starting from row i
    private void placeQueens(int row) {
        // If we've placed queens in all rows, a solution has been found
        if (row == boardSize) {
            ++solutionsCount;
            return;
        }

        // Attempt to place a queen in each column of the current row
        for (int col = 0; col < boardSize; ++col) {
            int positiveDiagonalIndex = row + col;
            int negativeDiagonalIndex = row - col + boardSize;
            if (columnsInUse[col] || positiveDiagonalsInUse[positiveDiagonalIndex] ||
                negativeDiagonalsInUse[negativeDiagonalIndex]) {
                continue; // can't place here as it's being attacked
            }

            // Place the queen by marking the column and diagonals as occupied
            columnsInUse[col] = true;
            positiveDiagonalsInUse[positiveDiagonalIndex] = true;
            negativeDiagonalsInUse[negativeDiagonalIndex] = true;

            // Move on to the next row
            placeQueens(row + 1);

            // Backtrack: remove the queen, making the column and diagonals available again
            columnsInUse[col] = false;
            positiveDiagonalsInUse[positiveDiagonalIndex] = false;
            negativeDiagonalsInUse[negativeDiagonalIndex] = false;
        }
    }
}

```

C++ Solution

```cpp
#include <bitset>
#include <functional>

class Solution {
public:
    // Entry point to solve the N-Queens II problem
    int TotalNQueens(int n) {
        std::bitset<10> columns;        // Tracks occupied columns
        std::bitset<20> major_diagonals; // Tracks occupied major diagonals
        std::bitset<20> minor_diagonals; // Tracks occupied minor diagonals
        int solution_count = 0;          // Stores the number of valid solutions

        // Lambda function to run depth-first search on rows
        std::function<void(int)> depth_first_search = [&](int row) {
            // Base case: all rows are filled, found a valid placement
            if (row == n) {
                ++solution_count;
                return;
            }
            // Iterate through columns at the current row
            for (int column = 0; column < n; ++column) {
                // Calculate indices for the diagonals
                int major_diag_index = row + column;
                int minor_diag_index = row - column + n;

                // Check if the current column or diagonals are occupied
                if (columns[column] || major_diagonals[major_diag_index] || minor_diagonals[minor_diag_index]) {
                    continue; // Skip to the next iteration if any are occupied
                }

                // Mark the current column and diagonals as occupied
                columns[column] = major_diagonals[major_diag_index] = minor_diagonals[minor_diag_index] = true;

                // Moves to the next row
                depth_first_search(row + 1);

                // Reset the current column and diagonals back to unoccupied for the next iteration
                columns[column] = major_diagonals[major_diag_index] = minor_diagonals[minor_diag_index] = false;
            }
        };

        // Start the DFS from the first row
        depth_first_search(0);

        // Return the total count of valid solutions found
        return solution_count;
    }
};

```

Typescript Solution

```typescript
// Use an array to track occupied columns, `true` indicates occupation
const columns: boolean[] = new Array<boolean>(10).fill(false);

// Track occupied diagonals ('true' indicates occupation)
const majorDiagonals: boolean[] = new Array<boolean>(20).fill(false);
const minorDiagonals: boolean[] = new Array<boolean>(20).fill(false);

// Variable to store the number of valid solutions
let solutionCount: number = 0;

// Function to solve the N-Queens II problem given `n` (the size of the chessboard)
function totalNQueens(n: number): number {
    // Reset tracking arrays and solution count for each new problem instance
    columns.fill(false);
    majorDiagonals.fill(false);
    minorDiagonals.fill(false);
    solutionCount = 0;

    // Inner function to perform depth-first search starting from the first row
    function depthFirstSearch(row: number) {
        // Base case: all rows are filled, a valid placement is found
        if (row === n) {
            solutionCount++;
            return;
        }

        // Iterate through columns at the current row
        for (let column = 0; column < n; column++) {
            // Calculate indices for the diagonals
            let majorDiagIndex = row + column;
            let minorDiagIndex = n - 1 - row + column;

            // Check if the current column or diagonals are occupied
            if (columns[column] || majorDiagonals[majorDiagIndex] || minorDiagonals[minorDiagIndex]) {
                continue; // Skip if any are occupied
            }

            // Mark the current column and diagonals as occupied
            columns[column] = majorDiagonals[majorDiagIndex] = minorDiagonals[minorDiagIndex] = true;

            // Recursive call to try the next row
            depthFirstSearch(row + 1);

            // Backtrack: unmark the current column and diagonals before the next iteration
            columns[column] = majorDiagonals[majorDiagIndex] = minorDiagonals[minorDiagIndex] = false;
        }
    }

    // Start the search from the first row
    depthFirstSearch(0);

    // Return the total count of valid solutions
    return solutionCount;
}

```

Time and Space Complexity

The given Python code is a solution to the N-Queens II problem which returns the number of distinct solutions for an n x n chessboard. The algorithm uses depth-first search (DFS) to traverse all possible board configurations and count valid placements of queens.

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.

Notice that the nonlocal variable `ans` is used to count the solutions but does not increase the space complexity as it requires constant space.