N-Queens problem using Backtracking

Aim:
The aim of solving the N-Queens problem using backtracking is to find and enumerate all valid arrangements of N queens on an N x N chessboard, ensuring that no two queens share the same row, column, or diagonal. This problem aims to explore efficient algorithms for placing the queens such that they do not attack each other, showcasing the power of backtracking techniques in solving combinatorial problems.

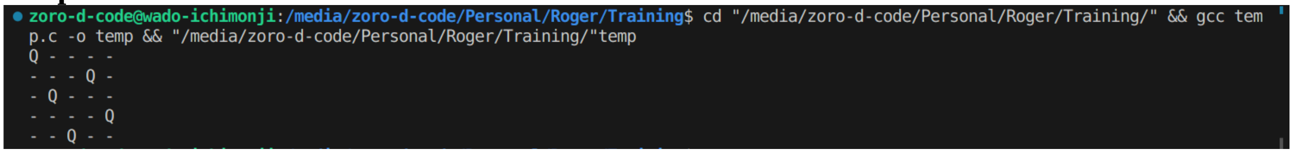
Theory:
The N-Queens problem is a classic combinatorial problem in computer science and mathematics. Its theory revolves around placing N queens on an N x N chessboard in such a way that no two queens threaten each other. Queens attack in straight lines horizontally, vertically, and diagonally, which means that no two queens can share the same row, column, or diagonal on the board.

The problem's theoretical significance lies in its representation of a constraint satisfaction problem and its relevance to algorithmic design and optimization. Solving the N-Queens problem requires exploring all possible configurations while adhering to the constraint of non-attacking queens, making it a suitable candidate for backtracking algorithms. The problem's complexity grows rapidly with increasing N, leading to exponential growth in the number of possible configurations to explore.

Code:
```
#include <stdbool.h>
#include <stdio.h>

#define N 5

void printSolution(int board[N][N])
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            if (board[i][j])
                printf("Q ");
            else
                printf("- ");
        }
        printf("\n");
    }
}

bool isSafe(int board[N][N], int row, int col)
{
    int i, j;
    for (i = 0; i < col; i++)
        if (board[row][i])
            return false;

    for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
        if (board[i][j])
```

```
            return false;

    for (i = row, j = col; j >= 0 && i < N; i++, j--)
        if (board[i][j])
            return false;

    return true;
}

bool solveNQUtil(int board[N][N], int col)
{
    if (col >= N)
        return true;
    for (int i = 0; i < N; i++)
    {
        if (isSafe(board, i, col))
        {
            board[i][col] = 1;
            if (solveNQUtil(board, col + 1))
                return true;
            board[i][col] = 0;
        }
    }
    return false;
}

bool solveNQ()
{
    int board[N][N] = {{0}};

    if (!solveNQUtil(board, 0))
    {
        printf("Solution does not exist");
        return false;
    }

    printSolution(board);
    return true;
}

int main()
{
    solveNQ();
    return 0;
}
```

Output:



```
● zoro-d-code@wado-ichimonji:/media/zoro-d-code/Personal/Roger/Training$ cd "/media/zoro-d-code/Personal/Roger/Training/" && gcc tem
  p.c -o temp && "/media/zoro-d-code/Personal/Roger/Training/"temp
  Q - - - -
  - - - Q -
  - Q - - -
  - - - - Q
  - - Q - -
```

Algorithm Steps:

1. Initialize the Chessboard:
- Create an N x N chessboard (2D array) to represent the board state.
- Initially, all cells are set to 0, indicating empty squares.

2. Place Queens using Backtracking:
- Start with the first column (col = 0).
- For each row (0 to N-1) in the current column:
  - Check if placing a queen at the current row and column is safe using the `isSafe` function.
  - If it's safe:
    - Set the cell at (row, col) to 1 to place the queen.
    - Recur for the next column (col + 1).
    - If placing queens in subsequent columns leads to a solution (recursively), return true.
    - If no solution is found, backtrack by setting the cell back to 0 and try the next row.

3. Check for Solutions:
- If all columns are processed (col >= N), a valid arrangement of queens is found.
- Print or store the arrangement as a solution.

4. Handle Backtracking:
- If no safe placement is found in a column, return false to backtrack to the previous column.
- Backtracking continues until all possibilities are explored or a valid solution is found.

5. Output Solutions:
- After backtracking, if a solution is found, print or store it.
- If no solution is found after exploring all possibilities, print a message indicating no solution exists.

6. Repeat for All Solutions (Optional):
- If desired, continue the process to find all possible arrangements of queens on the board.

Time Complexity:

1. isSafe Function (Checking for Valid Placements):
- In the worst case, the isSafe function checks three conditions for each queen placement:
  - Checking the current row: O(N)
  - Checking the left diagonal: O(N)
  - Checking the right diagonal: O(N)
- Since these checks are performed for each queen placement, the total time complexity of isSafe is O(N) per queen placement.

2. solveNQUtil Function (Recursive Placement of Queens):
- The solveNQUtil function is called recursively for each column (queen placement).
- In the worst case, for each column, the function tries to place the queen in each row, leading to N recursive calls.
- Since there are N columns (N queens), the worst-case time complexity of solveNQUtil is O(N^N).
- However, the actual time complexity can be less than O(N^N) due to pruning of invalid branches through backtracking.

3. Overall Time Complexity:
- Considering the above factors, the overall time complexity of the backtracking algorithm for the N-Queens problem is typically O(N^N).
- This complexity arises from the exponential growth in possibilities as the board size (N) increases.

4. Optimizations and Pruning:
- Various optimizations, such as early termination when a solution is found or pruning branches that lead to conflicts, can reduce the effective time complexity in practical scenarios.
- However, the fundamental complexity of exploring all possible configurations remains exponential due to the nature of the problem.

5. Average Case vs. Worst Case:
- While the worst-case time complexity is O(N^N), the average-case complexity can vary depending on the specific board size and the efficiency of pruning strategies.
- In practice, the algorithm may find solutions much faster for certain board configurations than for others.

Space Complexity:

1. Chessboard Representation:
- The chessboard is represented using a 2D array of size N x N to store the placement of queens.
- Therefore, the space required to represent the chessboard is O(N^2), where N is the size of the board.

2. Recursive Stack Space:
- The recursive calls in the solveNQUtil function consume additional space on the call stack.
- Since there can be up to N recursive calls (one for each column/queen placement), the maximum stack space used is O(N).

3. Auxiliary Variables:
- The isSafe function and other auxiliary variables require constant space, typically O(1), regardless of the board size.

4. Overall Space Complexity:
- Combining the space required for the chessboard representation (O(N^2)), recursive stack space (O(N)), and auxiliary variables (O(1)), the overall space complexity of the backtracking algorithm for the N-Queens problem is O(N^2 + N + 1), which simplifies to O(N^2).

5. Optimizations and Memory Usage:
- While the space complexity scales with the size of the board (N), the actual memory usage may vary based on the implementation and optimizations applied.
- Efficient implementations may minimize memory overhead and utilize space more effectively, especially when dealing with large board sizes.