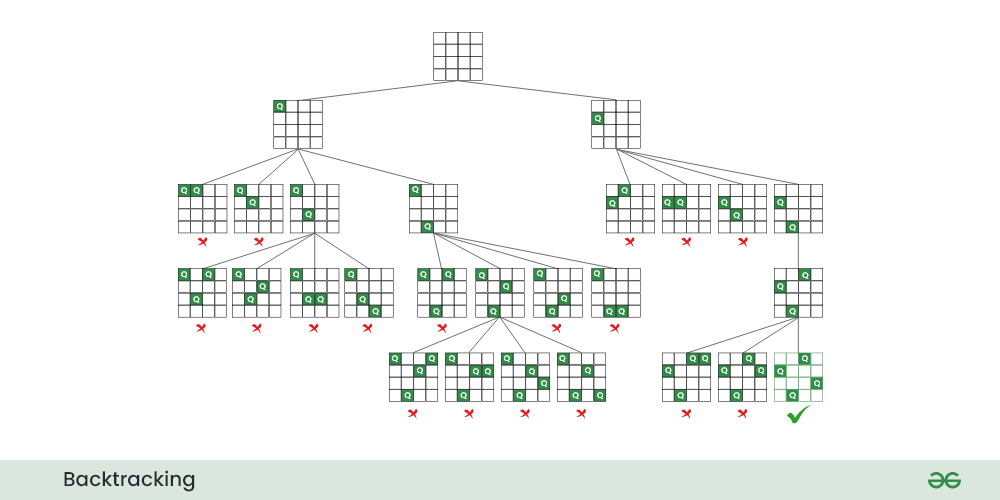
**N-Queen Algorithm**

*The idea is to place queens one by one in different columns, starting from the leftmost column. When we place a queen in a column, we check for clashes with already placed queens. In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution. If we do not find such a row due to clashes, then we backtrack and return****false****.*

* Start in the leftmost column
* If all queens are placed return true
* Try all rows in the current column. Do the following for every row.
  + If the queen can be placed safely in this row
    - Then mark this **[row, column]**as part of the solution and recursively check if placing queen here leads to a solution.
    - If placing the queen in **[row, column]** leads to a solution then return **true**.
    - If placing queen doesn’t lead to a solution then unmark this **[row, column]** then backtrack and try other rows.
  + If all rows have been tried and valid solution is not found return **false** to trigger backtracking.



* **It uses a simple 2D array to represent the chessboard. The elements of the array are either 0 or 1, indicating whether a queen is placed or not**
* The **isSafe** function checks if it's safe to place a queen in a given row and column. It checks the row, upper left diagonal, and lower left diagonal for any other queens.
* The **solveNQUtil** function uses a recursive backtracking approach to try placing queens in different columns, backtracking if it reaches an invalid state.

**Time Complexity:**O(N!)   
**Auxiliary Space:**O(N^2)

After Optimizing,

* It uses three auxiliary arrays **ld**, **rd**, and **cl** to keep track of the positions of queens on the left diagonal, right diagonal, and columns, respectively.
* The **solveNQUtil** function directly checks these auxiliary arrays to determine if a queen can be placed in a particular position.
* This approach reduces the need for repeatedly checking the entire board for each queen placement.

In summary, both codes solve the N-Queens problem using backtracking, but the second code uses additional arrays for more efficient checking of queen placements. The second code's approach is often more optimized, especially for larger values of N.

**Time Complexity:**O(N!)   
**Auxiliary Space:**O(N)

The N-Queens problem is an NP-hard problem, meaning that the time complexity can be exponential in the worst case. Different implementations of the backtracking algorithm may explore the solution space in different orders, leading to variations in the number of iterations.

It's essential to note that the number of iterations doesn't necessarily correlate with the efficiency or correctness of the algorithm. The primary goal is to find a valid solution for the N-Queens problem.

Applications:

Puzzle and Recreational Games

Constraint Satisfaction Problems (CSP)

