

Design and Analysis of Algorithms

- **Course Code:** BCSE204L
- **Course Type:** Theory (ETH)
- **Slot:** A1+TA1 & & A2+TA2
- **Class ID:** VL2023240500901
VL2023240500902

A1+TA1

Day	Start	End
Monday	08:00	08:50
Wednesday	09:00	09:50
Friday	10:00	10:50

A2+TA2

Day	Start	End
Monday	14:00	14:50
Wednesday	15:00	15:50
Friday	16:00	16:50

Syllabus- Module 2

Module:2	Design Paradigms: Dynamic Programming, Backtracking and Branch & Bound Techniques	10 hours
----------	---	----------

Dynamic programming: Assembly Line Scheduling, Matrix Chain Multiplication, Longest Common Subsequence, 0-1 Knapsack, TSP

Backtracking: N-Queens problem, Subset Sum, Graph Coloring

Branch & Bound: LIFO-BB and FIFO BB methods: Job Selection problem, 0-1 Knapsack Problem

Backtracking

Backtracking is a general algorithmic technique used for solving **optimization** and **decision problems**. The idea is to explore all possible solutions to a problem systematically, but in a way that **avoids unnecessary work** by abandoning a partial solution as soon as it is determined that the solution cannot be extended to a valid one.

Detailed breakdown of the backtracking process:

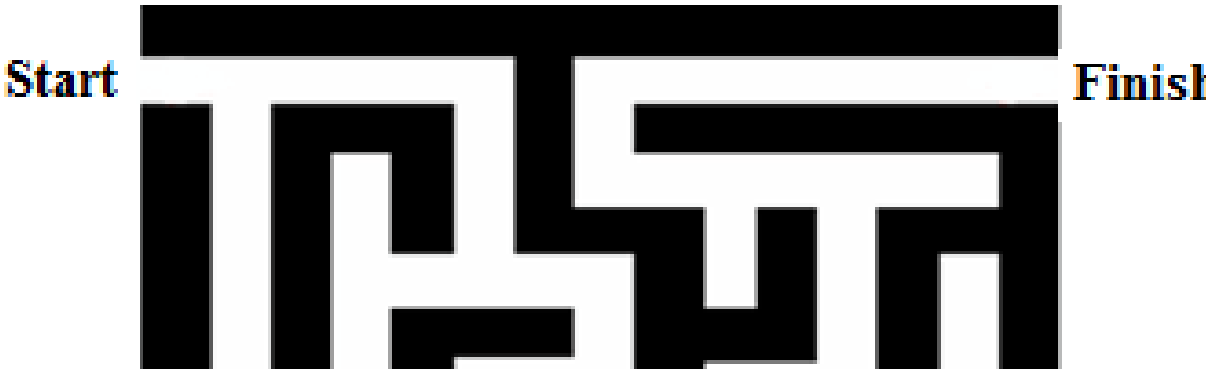
- **Choose:** Make a choice for the next decision point in the solution.
- **Explore:** Recursively explore all possible choices that can be made from the current decision point.
- **Backtrack:** If a choice leads to an invalid or unsuccessful solution, undo that choice and go back to the previous decision point to explore other alternatives.

This process continues until a valid solution is found, or it is determined that no valid solution exists.

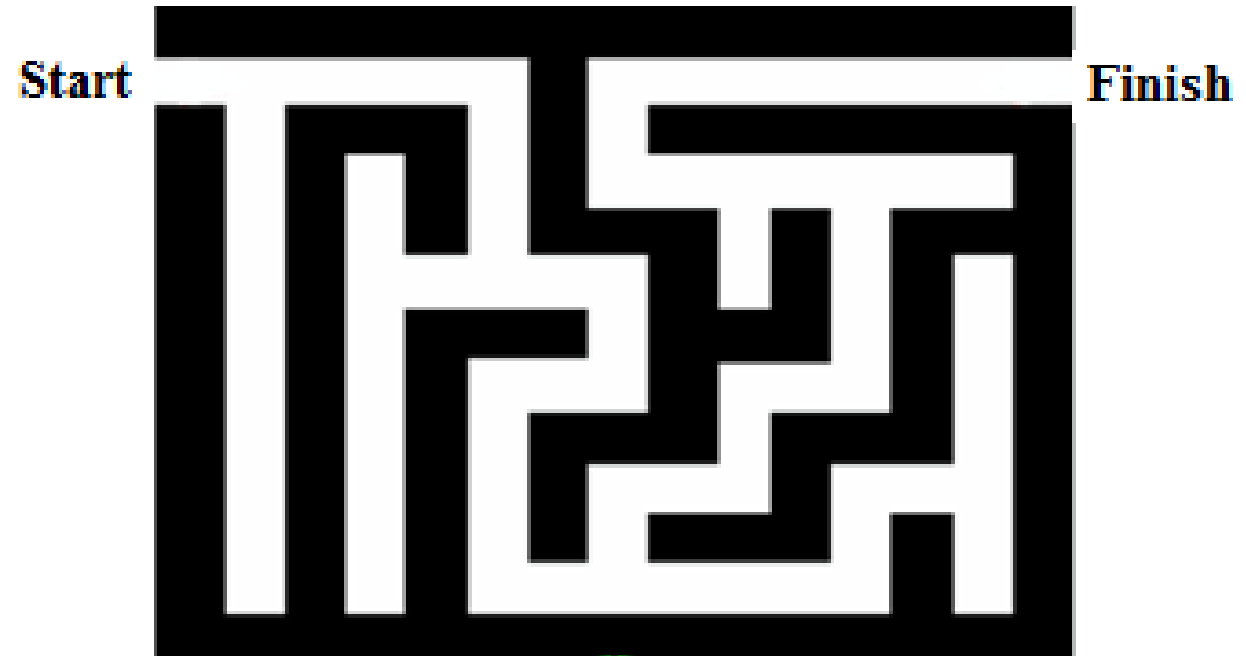
Backtracking is often used in combination with constraint satisfaction problems, where decisions must satisfy a set of constraints.

Backtracking

- Backtracking is an algorithmic-technique for solving problems *recursively by trying to build a solution incrementally*, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point of time.
-
- In maze, at each intersection, you have to decide between 3 or fewer choices:
 - ✓ Go straight
 - ✓ Go left
 - ✓ Go right
 - Many types of maze problem can be solved with backtracking.
- Given a maze, find a path from start to finish.



Given a maze, find a path from start to finish.



Applications of Backtracking:

Common examples of problems that can be solved using backtracking include

- N Queens Problem
- Sum of subsets problem
- Graph colouring
- Bin packing
- Hamiltonian cycles.

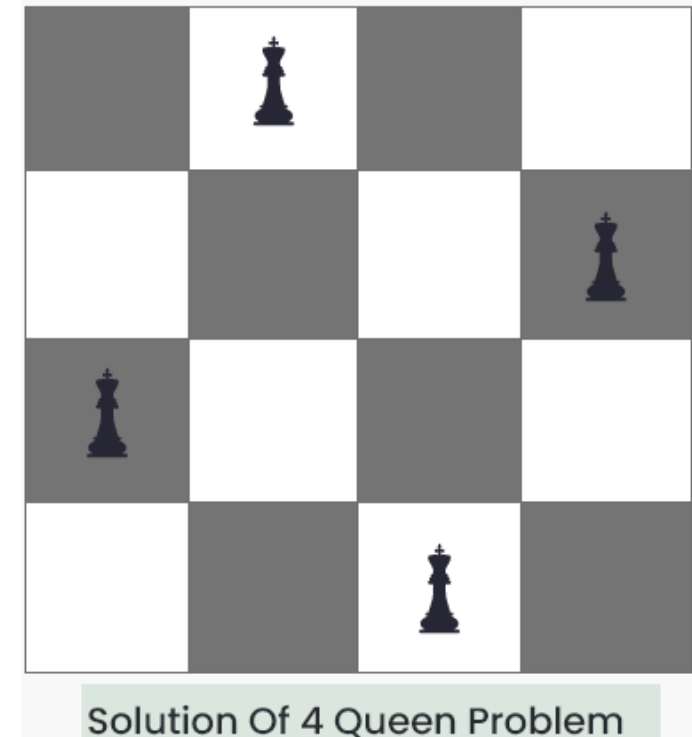
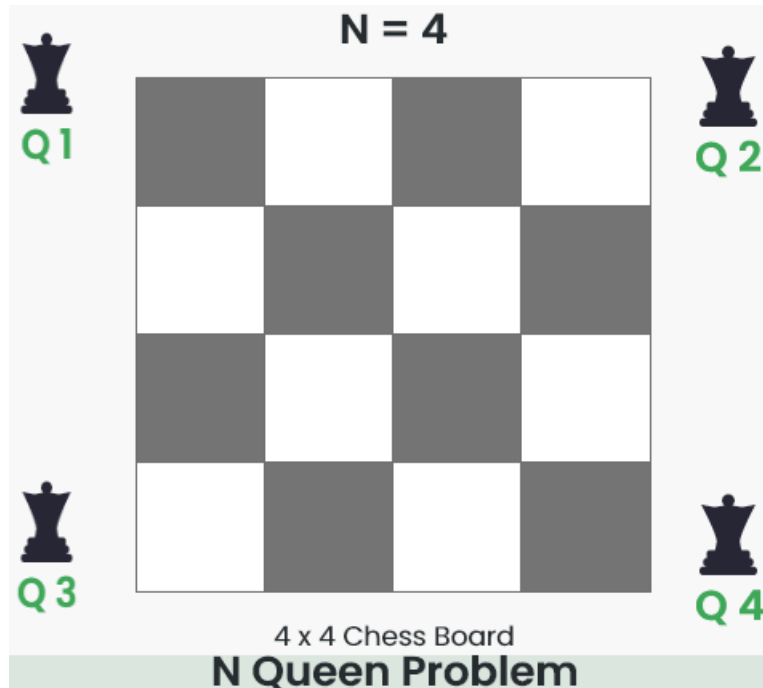
Note: *State-space tree* is the tree with the root node of no component solutions and from that construction of partial solutions, which are successive configurations or states of an instance.

State-space tree in Backtracking:

- In the context of backtracking, the state-space tree represents the **exploration of possible** solutions to a problem in a structured way.
- It is a tree structure where each node corresponds to a specific state or configuration in the process of finding a solution.
 - The root of the tree typically represents the initial state or the starting point of the problem, and the branches emanating from each node represent the choices or decisions made at that particular state.

N Queens Problem

The N-Queens problem is a classic problem in which you need to place N chess queens on an $N \times N$ chessboard in such a way that no two queens threaten each other. This means that no two queens can be in the same row, column, or diagonal.



N Queens Problem

N queens puzzle is the problem of placing N chess queens on an $N \times N$ chessboard so that no two queens threaten each other; thus, a solution requires that no two queens share the same row, column, or diagonal.

Time Complexity in Brute Force Approach:

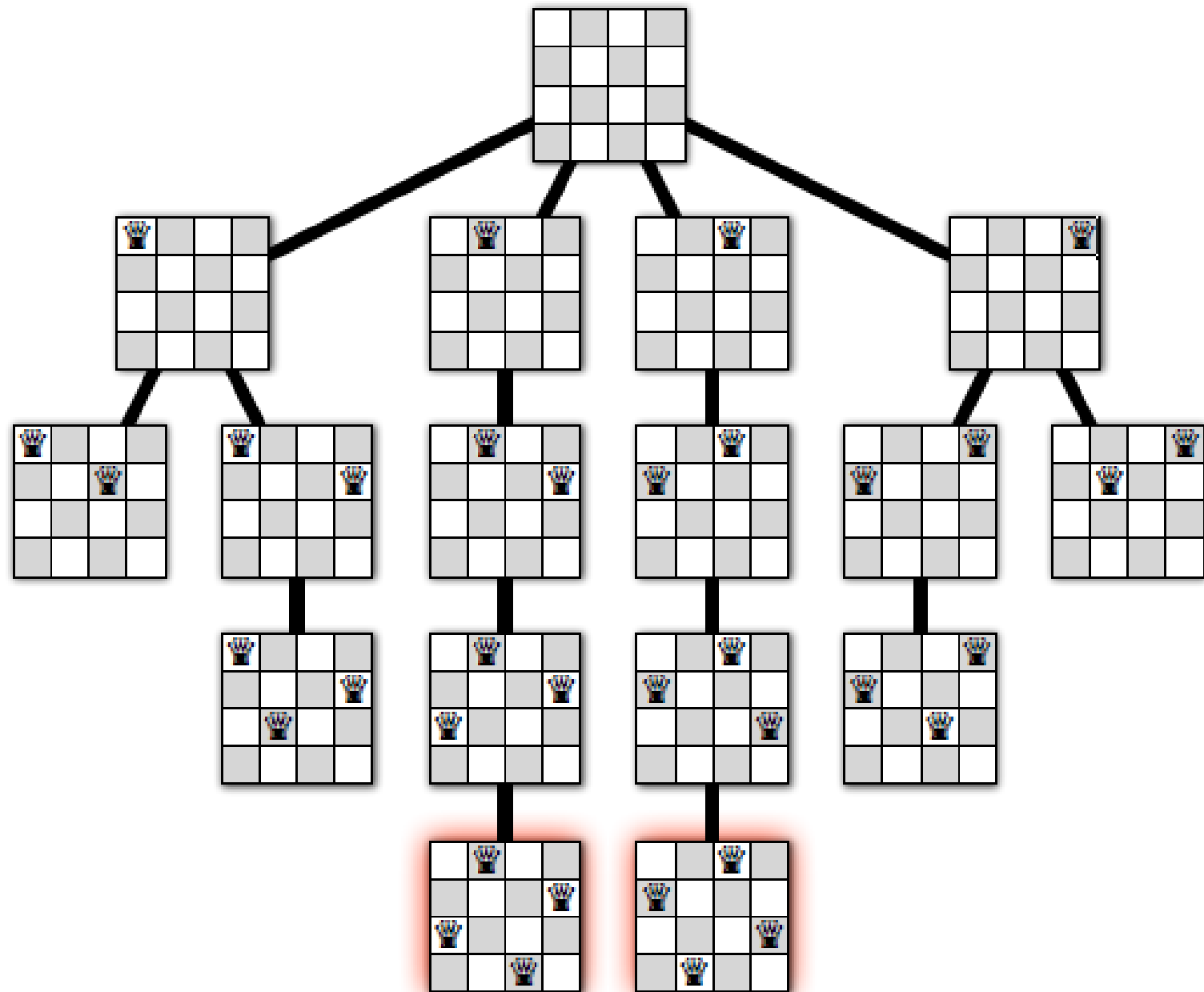
- The brute force approach without any optimization has an exponential time complexity of $O(N!)$

Because for each row in the first column, you have N choices, for each row in the second column, $N-1$ choices, and so on.

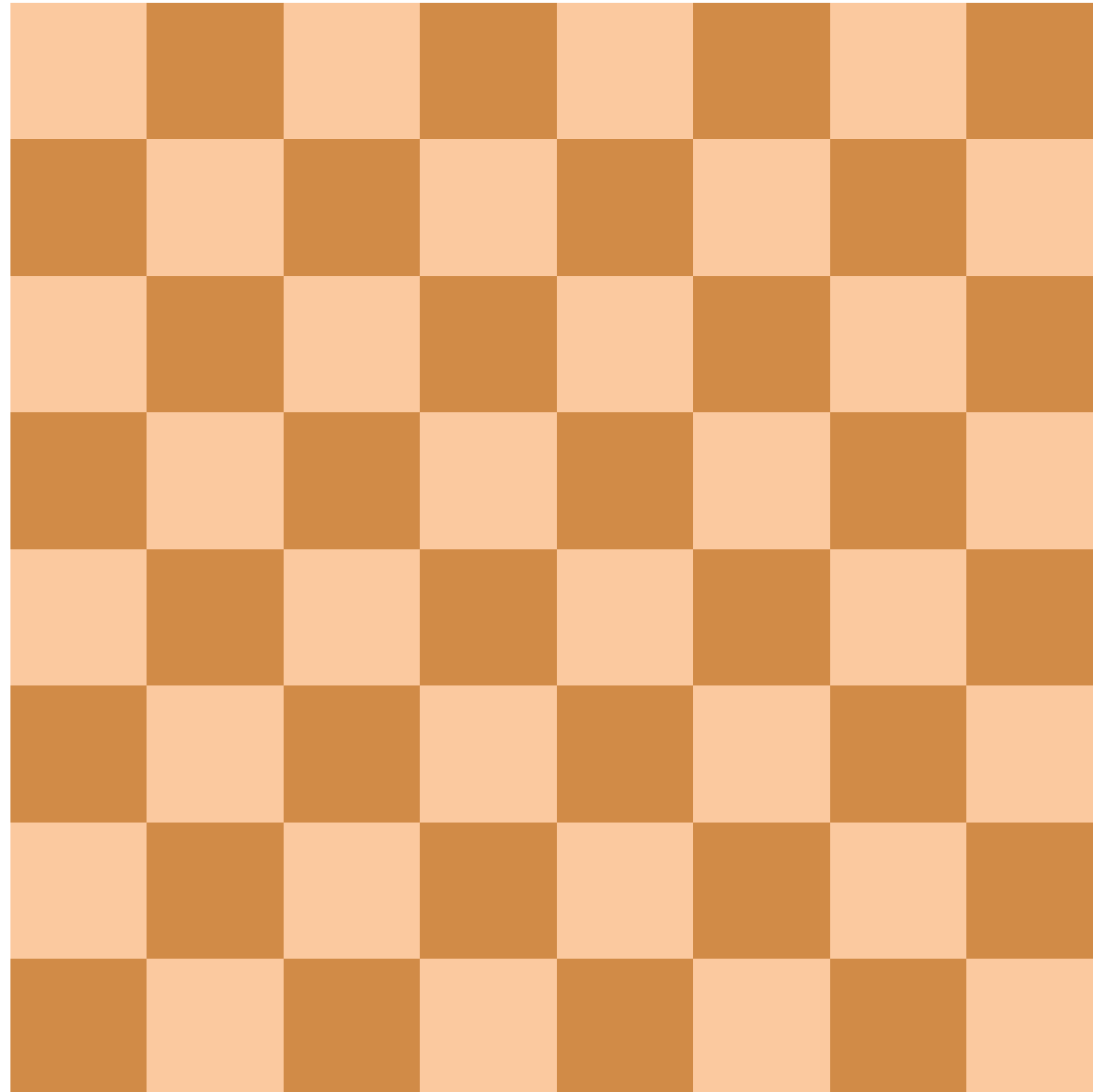
The total number of possibilities is $N * (N-1) * (N-2) * \dots * 1$, which is $N!$.

Backtracking approach to solving the N-Queens problem

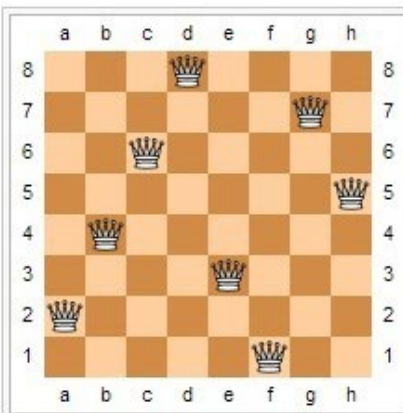
- The backtracking approach to solving the N-Queens problem involves trying to place queens on the chessboard **one at a time**, exploring all possible configurations and backtracking when a conflict is encountered.
- Start in the **leftmost column**.
- If **all queens are placed**, return **true**.
- Try all rows in the current column. For each row, check if the queen can be placed without conflicting with already-placed queens.
- If a safe spot is found, mark this cell and recursively try to place queens in the next columns.
- If placing queens in the current configuration leads to a solution, return true.
- If no safe spot is found, undo the queen placement, backtrack to the previous column, and try the next row in the previous column.



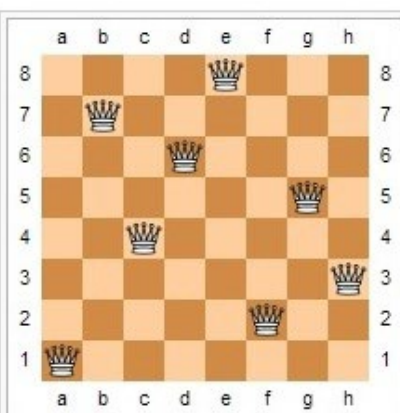
The complete recursion tree for our algorithm for the 4 queens problem.



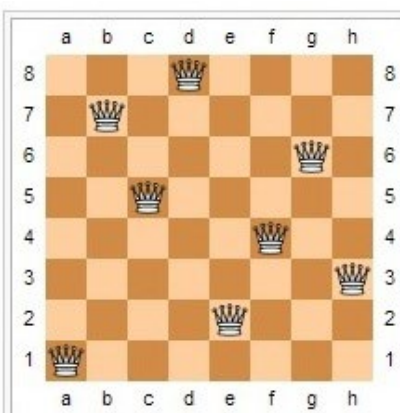
Dr. Venkata Phanikrishna B, SCOPE, VIT-Vellore



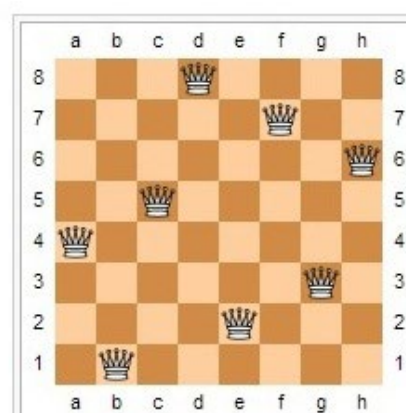
Solution 1



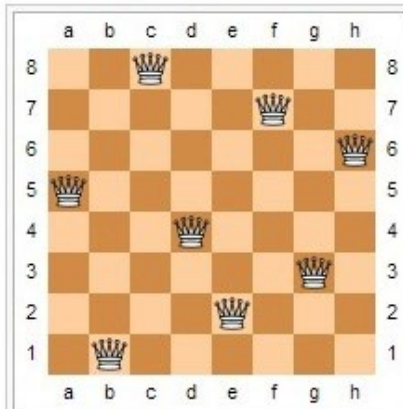
Solution 2



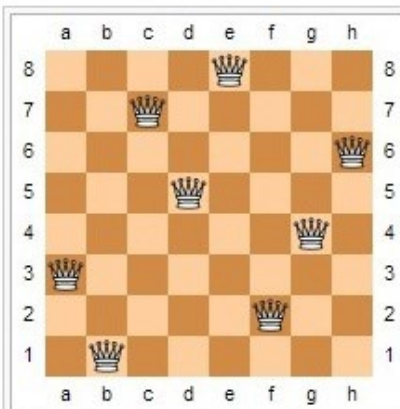
Solution 3



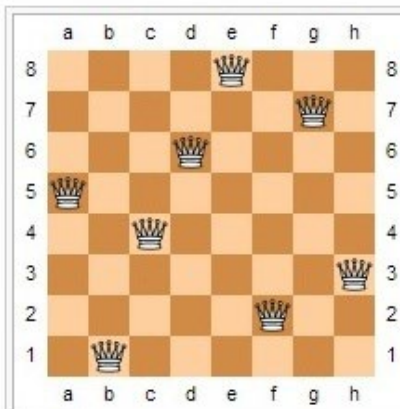
Solution 4



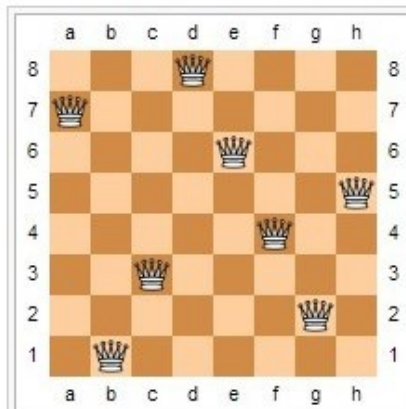
Solution 5



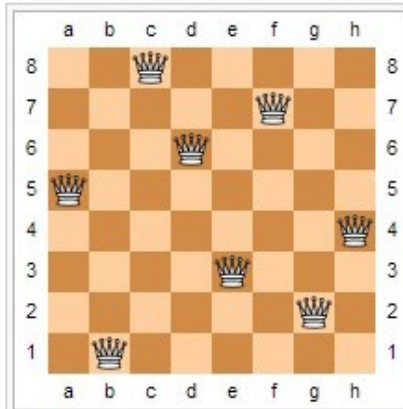
Solution 6



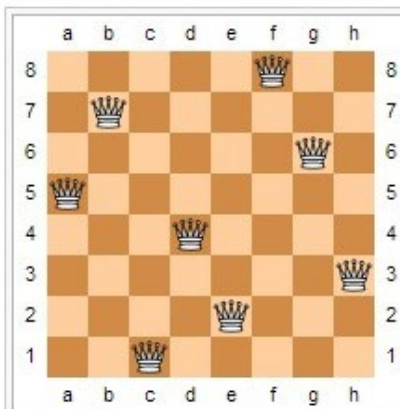
Solution 7



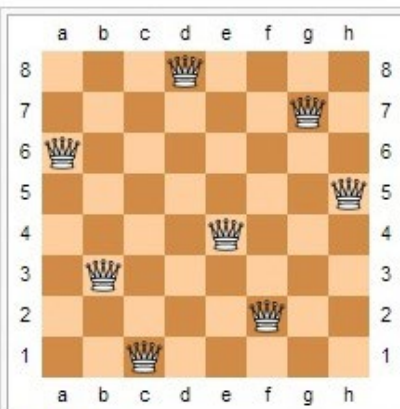
Solution 8



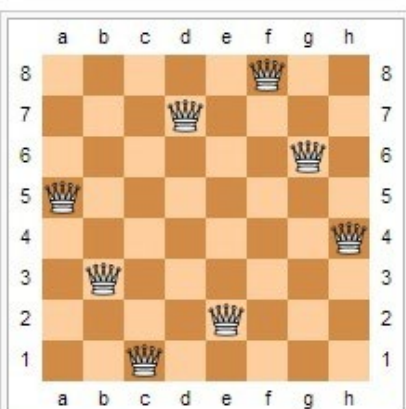
Solution 9



Solution 10



Solution 11



Solution 12

Backtracking approach to solving the N-Queens problem

- **Time Complexity in Backtracking:**
- The time complexity of the backtracking approach is **also exponential**, but it is often **significantly better than the brute force approach** due to **pruning the search space**.
 - In the best-case scenario, backtracking might lead to a solution much faster than the brute force method.
 - However, the worst-case time complexity remains exponential, influenced by the branching factor and depth of the backtracking tree.

Backtracking approach to solving the N-Queens problem

The recurrence relation for the time complexity of the N-Queens problem in backtracking can be expressed as follows:

$$T(N) = N \cdot (T(N-1) + T(N-2) + \dots + T(1))$$

- N represents the number of queens to be placed on the board.
- $T(N)$ is the time complexity to solve the problem of placing N queens on an $N \times N$ chessboard.
- The term $T(N-1) + T(N-2) + \dots + T(1)$ accounts for the recursive calls made to solve subproblems.

For each row in the current column, the algorithm makes recursive calls to solve the problem with fewer queens to be placed.

```

Algorithm printBoard(board):
    for i from 0 to N-1 do
        for j from 0 to N-1 do
            print board[i][j], " "
        print new line

```

```

Algorithm solveNQ(board, col):
    // base case
    if col >= N then
        return True

    // Consider this column and try placing
    // this queen in all rows one by one
    for i from 0 to N-1 do
        if isSafe(board, i, col) then
            // Place this queen in board[i][col]
            board[i][col] = 'Q'

            // recursively try to place the rest of the queens
            if solveNQ(board, col + 1) then
                return True

            // backtracking
            board[i][col] = '0'

    return False

```

```

// Example usage
N = 4
Call solve(N)

```

```

Algorithm isSafe(board, row, col):
    // Check this row on the left side
    for i from 0 to col-1 do
        if board[row][i] == 'Q' then
            return False

    // Check upper diagonal on the left side
    for i, j from row, col to 0, 0 step -1 do
        if board[i][j] == 'Q' then
            return False

    // Check lower diagonal on the left side
    for i, j from row, col to N-1, 0 step -1 do
        if board[i][j] == 'Q' then
            return False

    return True

```

```

Algorithm solve(N):
    Initialize board as a 2D array of size N x N with all elements set to '0'

    if not solveNQ(board, 0) then
        print "No Possible Solution exists"
        return False

    Call printBoard(board)
    return True

```

Any

Question



PresenterMedia



Thank You!

**FOR YOUR
ATTENTION**

