# **Leetcode 51 – N-Queens**

## Problem Understanding

Place n queens on an n x n chessboard such that **no two queens attack each other**.

**Rules:**

* A queen can attack vertically, horizontally, and diagonally.
* Output all distinct solutions. Each solution is a board configuration of n strings of length n with:
  + 'Q' = queen
  + '.' = empty square

## Optimized Java Solution (Backtracking)

class Solution {

public List<List<String>> solveNQueens(int n) {

List<List<String>> res = new ArrayList<>();

char[][] board = new char[n][n];

for (char[] row : board)

Arrays.fill(row, '.');

boolean[] col = new boolean[n]; // columns

boolean[] diag1 = new boolean[2 \* n - 1]; // row + col

boolean[] diag2 = new boolean[2 \* n - 1]; // row - col + n - 1

backtrack(0, board, res, col, diag1, diag2, n);

return res;

}

private void backtrack(int row, char[][] board, List<List<String>> res,

boolean[] col, boolean[] diag1, boolean[] diag2, int n) {

if (row == n) {

List<String> config = new ArrayList<>();

for (char[] r : board)

config.add(new String(r));

res.add(config);

return;

}

for (int c = 0; c < n; c++) {

int d1 = row + c, d2 = row - c + n - 1;

if (col[c] || diag1[d1] || diag2[d2]) continue;

board[row][c] = 'Q';

col[c] = diag1[d1] = diag2[d2] = true;

backtrack(row + 1, board, res, col, diag1, diag2, n);

board[row][c] = '.';

col[c] = diag1[d1] = diag2[d2] = false;

}

}

}

# Dry Run

## Initial Setup

* n = 4
* 4 rows → each row must have 1 Queen.
* Track:
  + col[4] → columns under attack.
  + diag1[7] → for / diagonals: row + col
  + diag2[7] → for \ diagonals: row - col + (n-1)

## Variables While Running:

* row: current row trying to place a Queen
* c: current column we’re checking
* board[row][c]: 'Q' if placed, else '.'

## Step-by-Step Dry Run (n = 4)

### ⬇️ Start at row 0:

#### → Try placing queen at (0, 0)

✅ Safe → mark:

* col[0] = true
* diag1[0 + 0] = true → diag1[0]
* diag2[0 - 0 + 3] = true → diag2[3]

Board:

Q . . .

. . . .

. . . .

. . . .

### ⬇️ row = 1

Try (1, 0) ❌ col[0] taken  
Try (1, 1) ❌ diag1[2], diag2[3] taken  
Try (1, 2) ✅

Mark:

* col[2] = true
* diag1[1+2] = diag1[3] = true
* diag2[1-2+3] = diag2[2] = true

Board:

Q . . .

. . Q .

. . . .

. . . .

### ⬇️ row = 2

Try (2, 0) ❌ diag2[1] = conflict  
Try (2, 1) ❌ diag1[3] taken  
Try (2, 2) ❌ col[2] taken  
Try (2, 3) ❌ diag1[5] = free, but diag2[2] = conflict → ❌

🚫 No safe cell → ❌ backtrack

### ⬅️ Backtrack from (1, 2)

Unmark:

* col[2] = false
* diag1[3] = false
* diag2[2] = false

Try (1, 3) ✅

Mark:

* col[3] = true
* diag1[4] = true
* diag2[1] = true

Board:

Q . . .

. . . Q

. . . .

. . . .

### ⬇️ row = 2

Try (2, 0) ✅

Mark:

* col[0] = true
* diag1[2] = true
* diag2[5] = true

Board:

Q . . .

. . . Q

Q . . .

. . . .

### ⬇️ row = 3

Try (3, 0) ❌ col[0] taken  
Try (3, 1) ❌ diag1[4] taken  
Try (3, 2) ✅

Mark:

* col[2] = true
* diag1[5] = true
* diag2[4] = true

Board:

Q . . .

. . . Q

Q . . .

. . Q .

✅ **Valid Solution 1**  
→ Save as:

["Q...","...Q","Q...","..Q."]

### ⬅️ Backtrack multiple steps...

Then, explore other possibilities like starting at (0, 1), (0, 2), (0, 3)

Eventually, we reach the **2nd valid configuration**:

. . Q .

Q . . .

. . . Q

. Q . .

## Final Output for n = 4

[

[".Q..","...Q","Q...","..Q."],

["..Q.","Q...","...Q",".Q.."]

]

## Time/Space Recap

| **Metric** | **Value** |
| --- | --- |
| Time | O(N!) |
| Space | O(N²) board + O(N) recursion |

# Why diag1 and diag2 are of size 2n - 1?

For an n x n board:

### diag1 = row + col → this is for / diagonals (top-left to bottom-right)

* The **minimum value** of row + col is 0 (when both are 0).
* The **maximum value** is (n - 1) + (n - 1) = 2n - 2.

🧠 So the range of values for row + col = 0 to 2n - 2

⏩ Total unique values = 2n - 1 ⇒ so we need a boolean[2n - 1]

### diag2 = row - col + (n - 1) → this is for \ diagonals (top-right to bottom-left)

* row - col ranges from -(n - 1) (bottom-left) to +(n - 1) (top-right)
* To **make all values non-negative** (for indexing), we shift it:

➕ Add n - 1 to each value ⇒ range becomes 0 to 2n - 2

Again, ➝ 2n - 1 unique diagonals ⇒ so we need boolean[2n - 1]

## How 1D array helps in O(1) tracking?

Let’s say you're placing a queen at (row, col). You want to **instantly check** if any other queen is on the same diagonal.

Normally:

* You’d loop through all placed queens and compare diagonals (costly).
* But with a 1D array, we do this in constant time.

### Example with n = 4:

boolean[] diag1 = new boolean[7]; // for `/`

boolean[] diag2 = new boolean[7]; // for `\`

At position (2, 3):

* / diagonal = 2 + 3 = 5 → check diag1[5]
* \ diagonal = 2 - 3 + 3 = 2 → check diag2[2]

Before placing a queen:

if (diag1[5] || diag2[2]) // diagonal is already occupied

continue;

After placing a queen:

diag1[5] = true;

diag2[2] = true;

Backtrack by resetting:

diag1[5] = false;

diag2[2] = false;

## Real Board vs Diagonal Index

Let’s visualize diagonals:

### Board:

(0,0) (0,1) (0,2) (0,3)

(1,0) (1,1) (1,2) (1,3)

(2,0) (2,1) (2,2) (2,3)

(3,0) (3,1) (3,2) (3,3)

### / Diagonals (row + col):

Index 0: (0,0)

Index 1: (0,1), (1,0)

Index 2: (0,2), (1,1), (2,0)

Index 3: (0,3), (1,2), (2,1), (3,0)

Index 4: (1,3), (2,2), (3,1)

Index 5: (2,3), (3,2)

Index 6: (3,3)

### \ Diagonals (row - col + 3):

Index 0: (3,0)

Index 1: (2,0), (3,1)

Index 2: (1,0), (2,1), (3,2)

Index 3: (0,0), (1,1), (2,2), (3,3)

Index 4: (0,1), (1,2), (2,3)

Index 5: (0,2), (1,3)

Index 6: (0,3)

## Summary

|  |  |  |  |
| --- | --- | --- | --- |
| Diagonal Type | Formula | Range (size) | Purpose |
| / | row + col | 0 to 2n - 2 | Top-left to bottom-right |
| \ | row - col + n - 1 | 0 to 2n - 2 | Top-right to bottom-left |
| Arrays | boolean[2n - 1] | O(1) check/set | Fast conflict detection |

## Time / Space Complexity

|  |  |
| --- | --- |
| Metric | Value |
| Time Complexity | O(N!) |
| Space | O(N²) for board, O(N) recursion |
| Output | All valid configurations |

## Alternate Approaches

|  |  |  |
| --- | --- | --- |
| Approach | Description | Notes |
| Backtracking | ✅ Optimal pruning with boolean arrays | Standard & efficient |
| Bitmasking | Advanced; optimized for large n | Used in Leetcode 52 |
| Brute-force | Try every position without pruning | Inefficient |