**51. N-Queens**

<https://leetcode.com/problems/n-queens/>

1. **Listen**

**Problem Statement:**

The **n-queens** puzzle is the problem of placing n queens on an n x n chessboard such that no two queens attack each other.

Given an integer n, return *all distinct solutions to the****n-queens puzzle***. You may return the answer in **any order**.

Each solution contains a distinct board configuration of the n-queens' placement, where 'Q' and '.' both indicate a queen and an empty space, respectively.

**Input:**

**int n** is the number of queens on an **n x n** chessboard

**Goal:**

The **n** queens should be positioned in a way on the chessboard such that no two queens are able to attack each other.

**Return:**

return *all* ***distinct*** *solutions to the****n-queens puzzle***

Each solution contains a distinct board configuration of the n-queens' placement, where 'Q' and '.' both indicate a queen and an empty space, respectively.

1. **Examples**

**Example 1:**

Shape, square

Description automatically generated

**Input:** n = 4

**Output:** [[".Q..","...Q","Q...","..Q."],["..Q.","Q...","...Q",".Q.."]]

**Explanation:** There exist two distinct solutions to the 4-queens puzzle as shown above

**Example 2:**

**Input:** n = 1

**Output:** [["Q"]]

**Constraints:**

* You may return the answer in **any order**.
* 1 <= n <= 9

**Test Cases:**

* n = 1
* n = 9
* n = 3

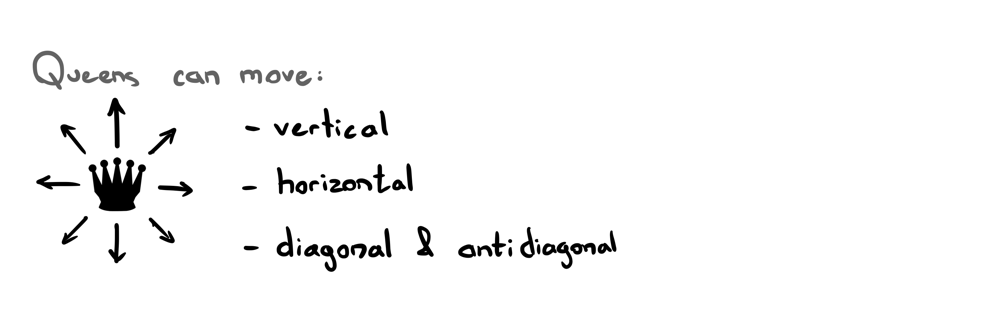
1. **Brute Force**

**Solution 1: Backtracking**

**Intuition:**

A queen can move as many spaces as it wants in four ways

1. UP
2. DOWN
3. POSITIVE DIAGONAL (left-bottom to right-up)
4. NEGATIVEDIAGONAL (left-up to right-bottom)



We want to place n queens on the board in such a way where none of them can run into each other, no matter which way they choose to go. This gives us some intuitive rules to follow.

If we run a truly brute force algorithm, then each queen could be in n^2 possible positions. Therefore, we can put a set of constriants down to make it easier.

It's only possible to place a Queen down if and only if:

1. There exists no Queen on the current row.
2. There exists no Queen on the current column.
3. There exists no Queen on the current diagonal.
4. There exists no Queen on the current anti-diagonal.

This matches our usual definition of a **Constraint** in backtracking:

* *Choose*: Choose the potential candidate. Here, our potential candidates are all substrings that could be generated from the given string.
* *Constraint*: Define a constraint that must be satisfied by the chosen candidate. In this case, the constraint is that the string must be a *palindrome*.
* *Goal*: We must define the goal that determines if have found the required solution and we must backtrack. Here, our goal is achieved if we have reached the end of the string.

In particular, it seems like not allowing two queens to be on the same diagonal would be difficult to implement.

Therefore, we keep three sets to keep track of a candidates current position:

A column set

A positive diagonal set (row + column)

A negative diagonal set (row - column)

For each potential candidate, add its position to the associated sets and continue down the board.

When we backtrack, remove the candidate’s position from the sets and try the next position.

Our backtracking algorithm is as follows:

We loop over every column in the current row.

If we encounter a valid Queen position on the board (according to set of columns, positive diagonals, and negative diagonals), add its position to the associated set trackers then check the next row down.

If the current position on the board is not a valid Queen position, ignore that space and try again down the column.

When the traversal reaches the bottom of the board and finds a valid position, we add it to the result list.

Then we backtrack back to the previous row (row n-1) and try all columns to the right of the current Queen position.

If we find another valid Queen position, traverse back down the board.

If not, then we backtrack up to the previous row (n-2).

If we find another valid Queen position, traverse back down the board.

Once we are down another row, we restart the loop from the very left of the row to the very right.

We repeat this traversal-backtracking process for all board candidates until reaching the top-rightmost column, meaning we have finished.

Here is a good visualization of the whole process: <https://nqueen.netlify.app>

**Solution 2: Trading time for space**

We can trade disregard the use of sets to keep track of candidate positions by manually checking if there is another queen in the same column, positive diagonal, or negative diagonal for every candidate.

// check same column

check all squares in same column by traversing up rows

// check same negative diagonal

Each negative diagonal from a certain square on the board can be found by decrementing the row and column by 1 on each iteration.

Let’s take square at board[3][2] and check to see if there are any queens in the same negative diagonal.

Shape, square

Description automatically generated

We must start from the square’s index and make our way from the right side of the board to the left side of the board (bottom right to top left).

To find the next diagonal, we go up one row, and left one column.

To find the next diagonal, we go up another row, and left another column (now we are two rows up and two columns left from our original position).

and so on…

Until we reach the left side of the board. How do we know when we reach it? We can take the minimum between the original row and column indices to find the number of negative diagonals for the current square.

// check same positive diagonal

1. **Optimize**
2. **Walkthrough**
3. **Implement**
4. **Test**