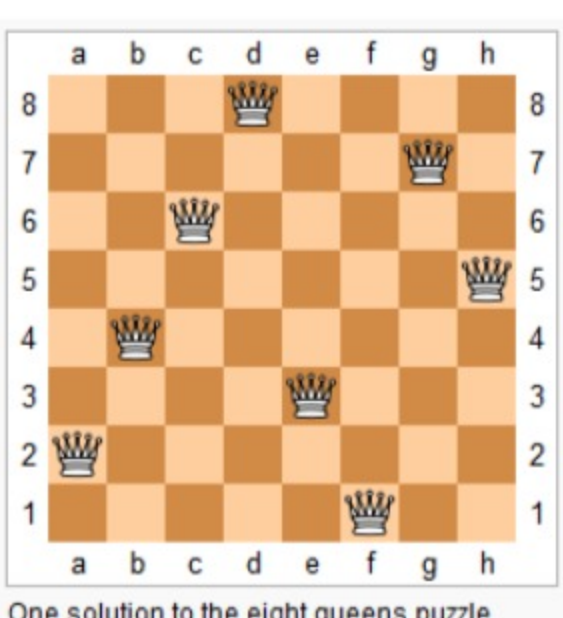
51. N Queens

April 4, 2019 | 25K views

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



One solution to the eight queens puzzle

Given an integer n, 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.

Example:

```
Input: 4
Output: [
 [".Q..",  // Solution 1
  "...Q",
  "Q...",
  "..Q."],

 ["..Q.",  // Solution 2
  "Q...",
  "...Q",
  ".Q.."]
]
Explanation: There exist two distinct solutions to the 4-queens puzzle as shown above.
```

Solution

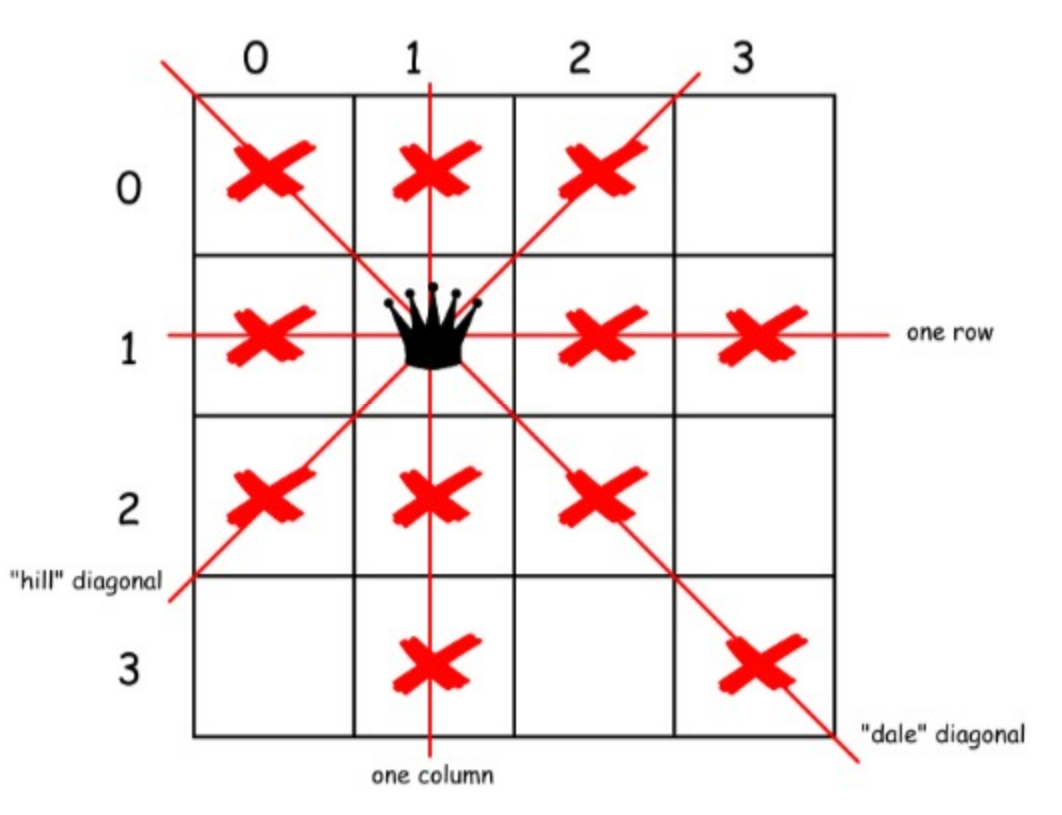
Intuition

The first idea is to use brute-force that means to generate all possible ways to put `N` queens on the board, and then check them to keep only the combinations with no queen under attack. That means $\mathcal{O}(N^N)$ time complexity and hence we're forced to think further how to optimize.

There are two programming conceptions here which could help.

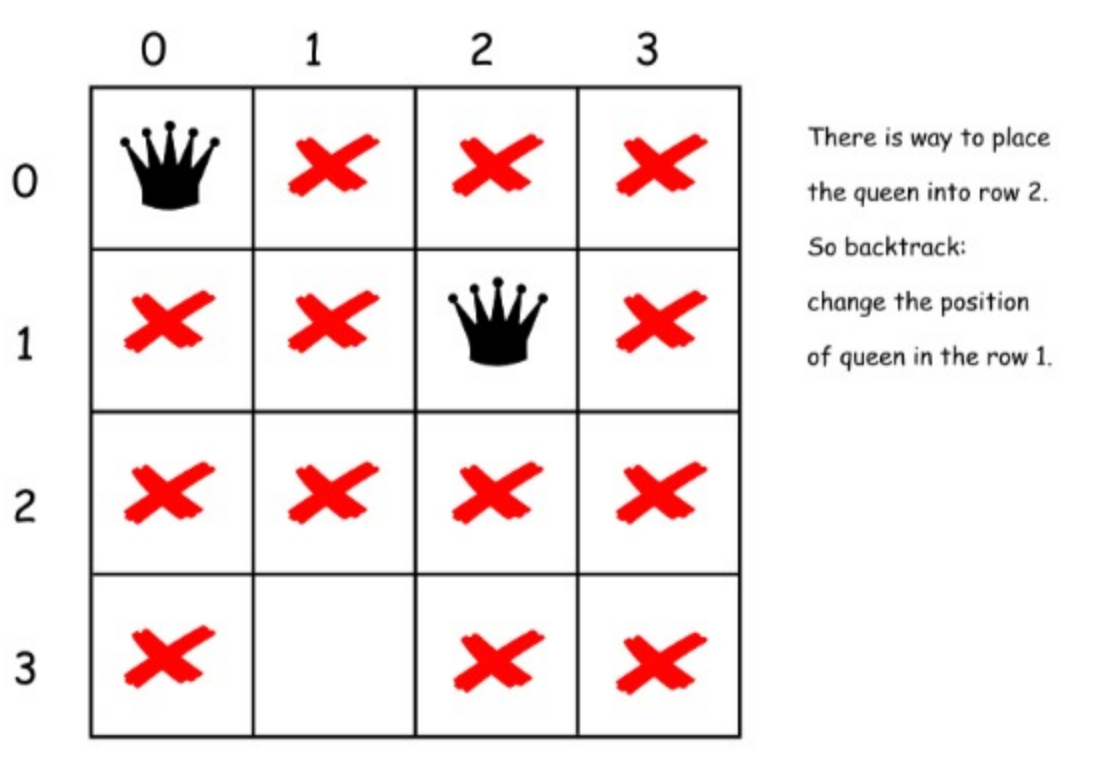
> The first one is called *constrained programming*.

That basically means to put restrictions after each queen placement. One puts a queen on the board and that immediately excludes one column, one row and two diagonals for the further queens placement. That propagates *constraints* and helps to reduce the number of combinations to consider.



> The second one called *backtracking*.

Let's imagine that one puts several queens on the board so that they don't attack each other. But the combination chosen is not the optimal one and there is no place for the next queen. What to do? *To backtrack*. That means to come back, to change the position of the previously placed queen and try to proceed again. If that would not work either, *backtrack* again.



Approach 1: Backtracking

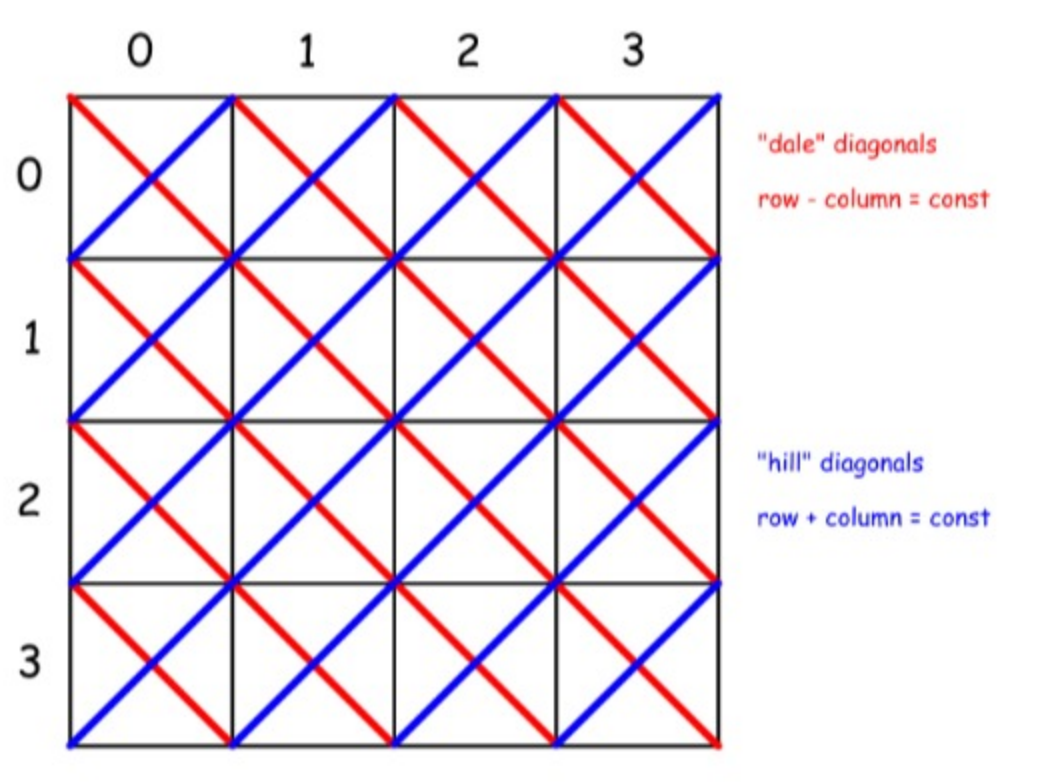
Before to construct the algorithm, let's figure out two tips that could help.

> There could be the only one queen in a row and the only one queen in a column.

That means that there is no need to consider all squares on the board. One could just iterate over the columns.

> For all "hill" diagonals `row + column = const`, and for all "dale" diagonals `row - column = const`.

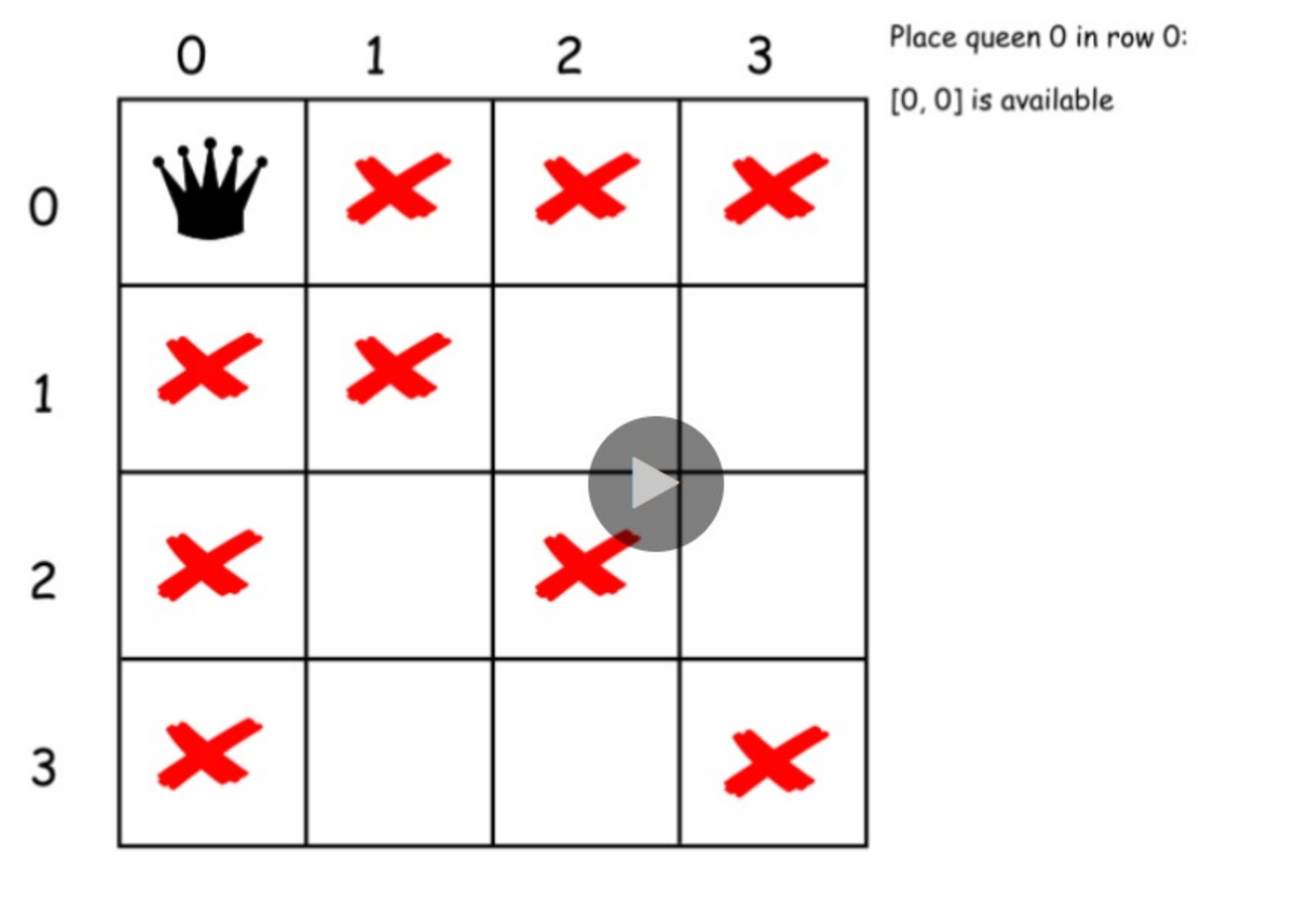
That would allow us to mark the diagonals which are already under attack and to check if a given square `(row, column)` is under attack.



Now everything is ready to write down the backtrack function `backtrack(row = 0)`.

- Start from the first `row = 0`.
- Iterate over the columns and try to put a queen in each `column`.
  - If square `(row, column)` is not under attack
    - Place the queen in `(row, column)` square.
    - Exclude one row, one column and two diagonals from further consideration.
    - If all rows are filled up `row == N`
      - That means that we find out one more solution.
    - Else
      - Proceed to place further queens `backtrack(row + 1)`.
    - Now backtrack : remove the queen from `(row, column)` square.

Here is a straightforward implementation of the above algorithm.



Java Python Copy

```python
class Solution:
    def solveNQueens(self, n: int) -> List[List[str]]:
        def could_place(row, col):
            return not (cols[col] + hill_diagonals[row - col] + dale_diagonals[row + col])

        def place_queen(row, col):
            queens.add((row, col))
            cols[col] = 1
            hill_diagonals[row - col] = 1
            dale_diagonals[row + col] = 1

        def remove_queen(row, col):
            queens.remove((row, col))
            cols[col] = 0
            hill_diagonals[row - col] = 0
            dale_diagonals[row + col] = 0

        def add_solution():
            solution = []
            for _, col in sorted(queens):
                solution.append('.' * col + 'Q' + '.' * (n - col - 1))
            output.append(solution)

        def backtrack(row = 0):
            for col in range(n):
                if could_place(row, col):
                    place_queen(row, col)
```

Complexity Analysis

- Time complexity : $\mathcal{O}(N!)$. There is `N` possibilities to put the first queen, not more than `N (N - 2)` to put the second one, not more than `N(N - 2)(N - 4)` for the third one etc. In total that results in $\mathcal{O}(N!)$ time complexity.
- Space complexity : $\mathcal{O}(N)$ to keep an information about diagonals and rows.