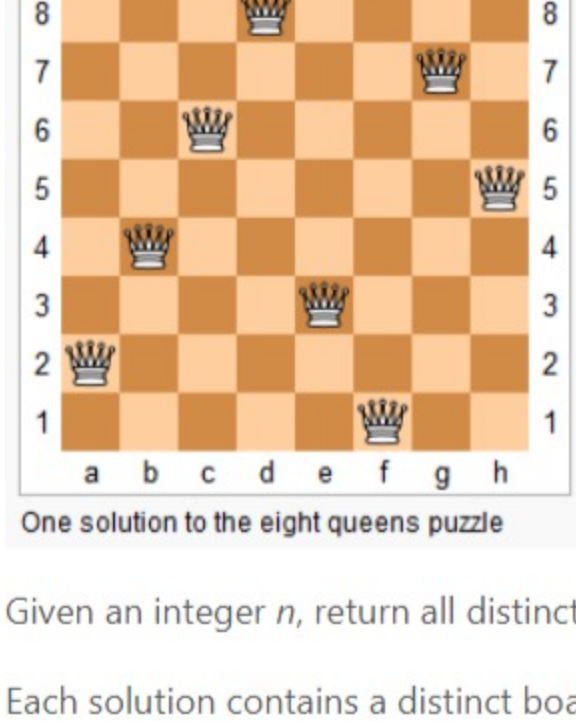


51. N Queens

April 4, 2019 | 25K views

Previous, Next, 5 stars, Average Rating: 3.50 (30 votes)

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.



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

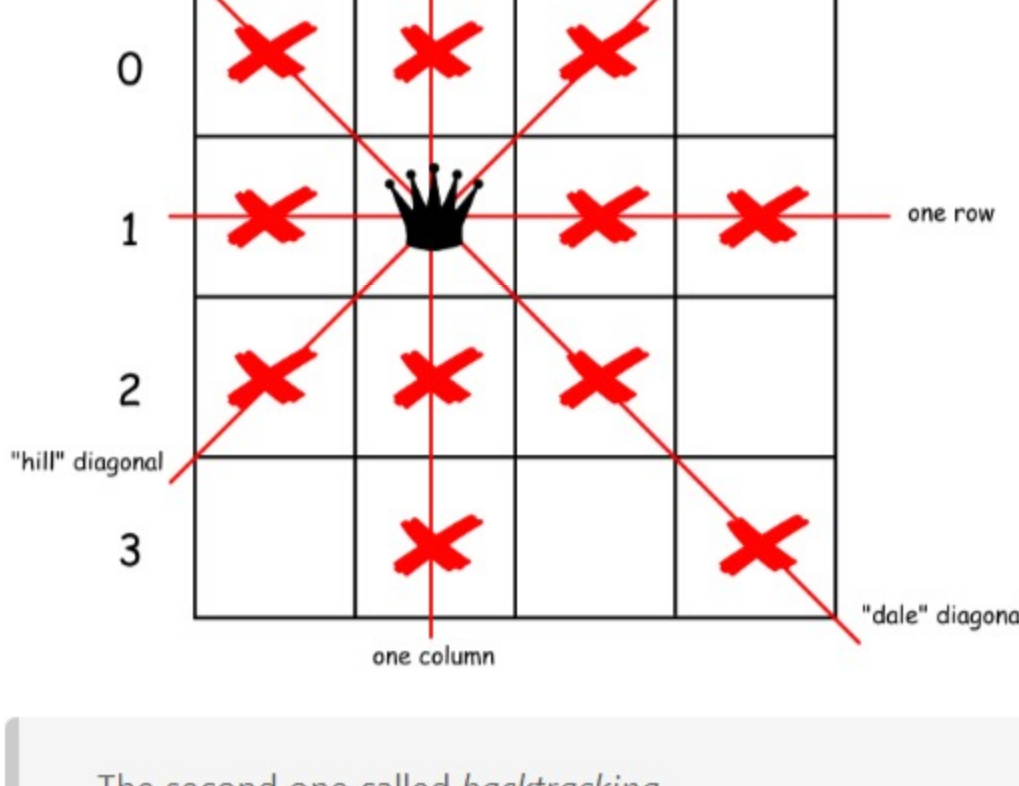
Intuition

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 $O(N^N)$ time complexity and hence we're forced to think further how to optimize.

There are two programming conceptions here which could help.

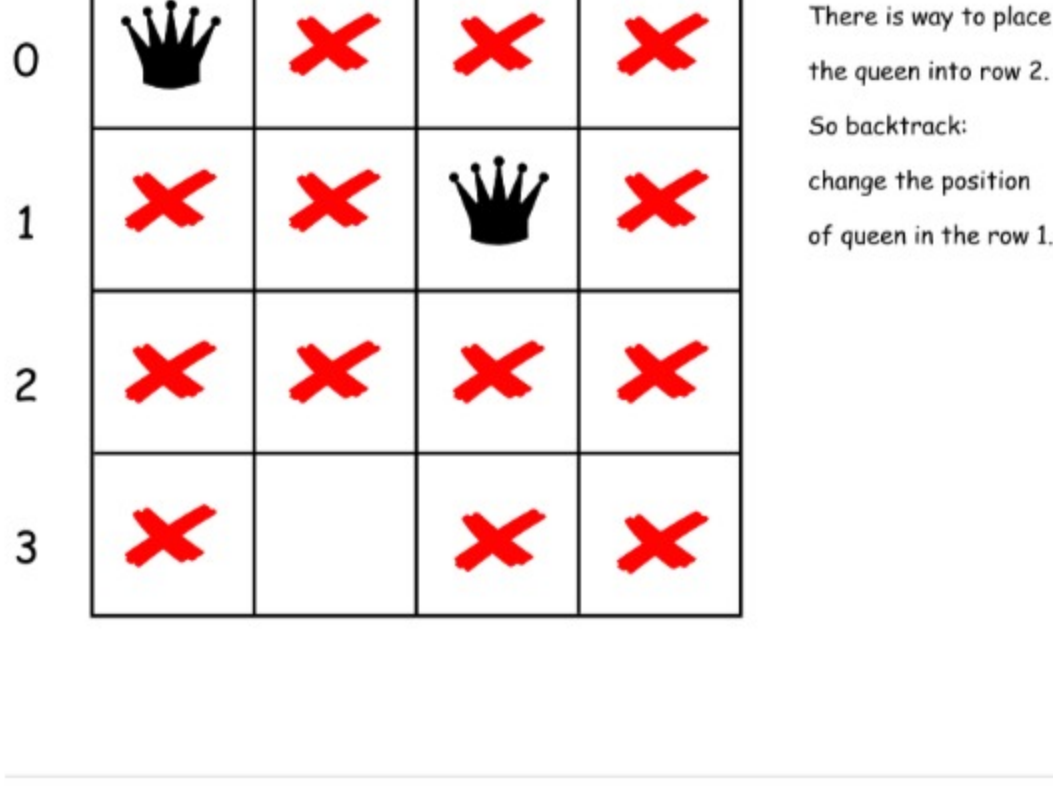
The first one is called *constrained programming*.

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

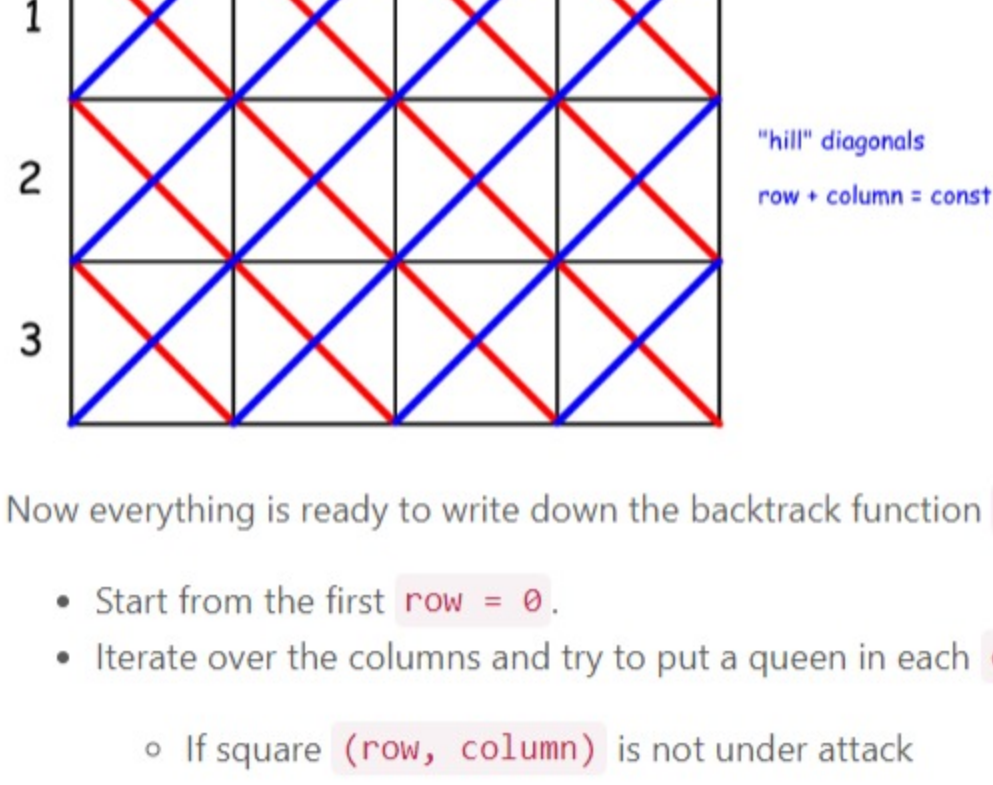
Before to construct the algorithm, let's figure out two tips that could help.

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.

0 1 2 3

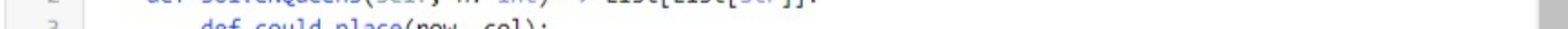
0 [Queen] [X] [X] [X]

1 [X] [X] [Queen] [X]

2 [X] [X] [X] [X]

3 [X] [] [X] [X]

Place queen 0 in row 0:
[0, 0] is available



```
class Solution:
    def solveQueens(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 : $O(N!)$. 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 $O(N!)$ time complexity.
- Space complexity : $O(N)$ to keep an information about diagonals and rows.

Rate this article: 5 stars

Previous, Next

Comments: 19

Sort By

🔊

Type comment here... (Markdown is supported)

👁 Preview

Post

veera_venkata_p

🌟25

🕒 July 1, 2019 12:09 AM

Would someone please explain the logic behind assigning these values to the arrays?
hills = new int[4 * n - 1];
dales = new int[2 * n - 1];

25 👍 | 🗨 Share | 🗨 Reply

SHOW 2 REPLIES

raivats1

🌟15

🕒 October 7, 2019 7:36 PM

Unrelated, but according to this Math Stackexchange article, the names of \diagup diagonals are "major", "principal", "primary", "main", while \diagdown diagonals are "minor", "counter", "secondary", "anti-".

These are probably better names than "hills" and "dales"...

13 👍 | 🗨 Share | 🗨 Reply

SHOW 1 REPLY

supratikn1997

🌟7

🕒 March 13, 2020 7:16 AM

The solution in Cracking the Coding Interview is so much clearer than this.

7 👍 | 🗨 Share | 🗨 Reply

jvalecillos

🌟6

🕒 April 7, 2019 2:08 PM

The implementation is confusing about "hills" and "dales".

The theory explains the reasoning for checking if the queens are in the same diagonals, and divide it in two cases for "hills" and "dales":

6 👍 | 🗨 Share | 🗨 Reply

SHOW 2 REPLIES

RameshThaleia

🌟64

🕒 April 23, 2020 9:03 AM

How is anyone supposed to figure this out in an interview?

3 👍 | 🗨 Share | 🗨 Reply

SHOW 1 REPLY

dannyli0818

🌟137

🕒 March 29, 2020 10:33 PM

actually, diagonal and antidiagonal may sound more clear than dale and hill

2 👍 | 🗨 Share | 🗨 Reply

flarbear

🌟283

🕒 April 5, 2019 1:29 AM

You only need 2n-1 storage for both hills and dales. Since `row-col` varies from `-(n-1)` to `+(n-1)`.
You can set the hill diagonal with `hills[row - col + n - 1] = 1`

2 👍 | 🗨 Share | 🗨 Reply

SHOW 3 REPLIES

amanzholovm

🌟40

🕒 February 9, 2020 12:00 PM

Correct me if I am wrong, but I think there is a mistake in naming in a solution. Hill_diagonals and dale_diagonals must be swapped ..

2 👍 | 🗨 Share | 🗨 Reply

svm14

🌟6

🕒 June 4, 2020 8:35 PM

I found this on YouTube and was useful : <https://www.youtube.com/watch?v=DYz2A9s0lQ4>

1 👍 | 🗨 Share | 🗨 Reply

kevinhynes

🌟286

🕒 June 5, 2019 6:02 PM

Would anyone care to elaborate on the original $O(N^N)$ runtime of the brute force approach versus the $O(N!)$ runtime of this solution? I am finding higher-order time complexity analysis to be very confusing.

1 👍 | 🗨 Share | 🗨 Reply

SHOW 2 REPLIES