Explore Problems Mock Contest Articles Discuss Store

Articles > 52. N Queens II

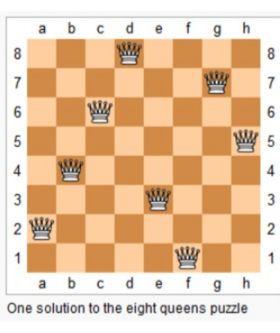
Previous Next

52. N Queens II

Jan. 4, 2019 | 19.4K views

Average Rating: 3.92 (13 votes)

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



One solution to the eight queens puzzle

Given an integer *n*, return the number of distinct solutions to the *n*-queens puzzle.

Example:

```
Input: 4
Output: 2
Explanation: There are two distinct solutions to the 4-queens puzzle as shown below.
[
 [".Q..",  // Solution 1
  "...Q",
  "Q...",
  "..Q."],

 ["..Q.",  // Solution 2
  "Q...",
  "...Q",
  ".Q.."]
]
```

Solution

Intuition

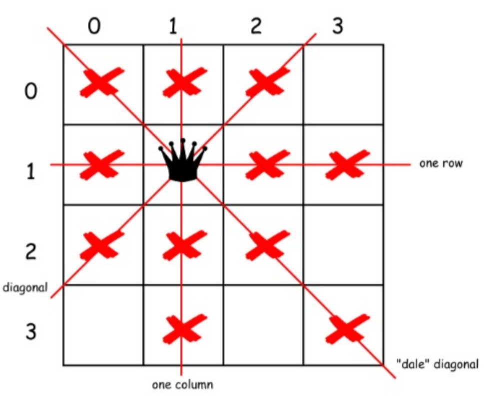
This problem is a classical one and it's important to know the solution to feel classy.

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.

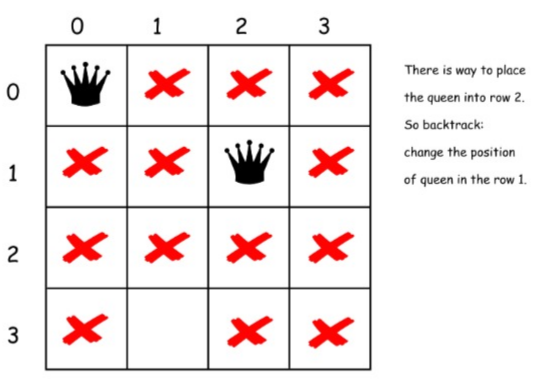
> The first one is called *constrained programming*.

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

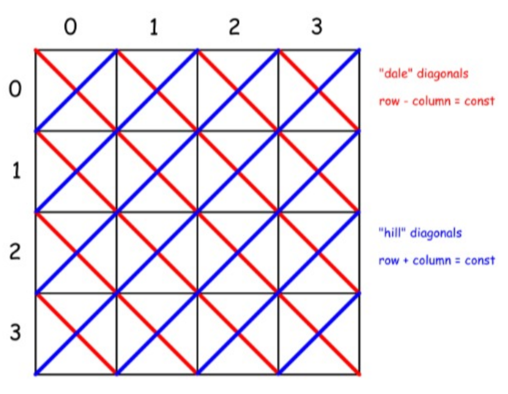
Before to construct the algorithm, let's figure out two tips that could help.

> 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`.

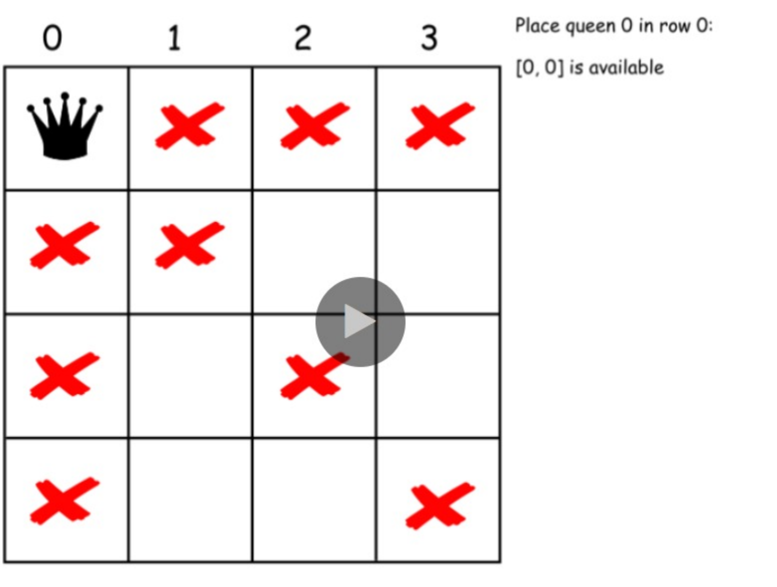
That would allow us to mark the diagonals which are already under attack and to check if a given square `(row, column)` is under attack.



Now everything is ready to write down the backtrack function `backtrack(row = 0, count = 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 `count++`.
    - Else
      - Proceed to place further queens `backtrack(row + 1, count)`.
    - Now backtrack : remove the queen from `(row, column)` square.

Here is a straightforward implementation of the above algorithm.



1 / 34

Java Python Copy

```python
class Solution:
    def totalNQueens(self, n):
        """
        :type n: int
        :rtype: int
        """
        def is_not_under_attack(row, col):
            return not (rows[col] or hills[row - col] or dales[row + col])

        def place_queen(row, col):
            rows[col] = 1
            hills[row - col] = 1  # "hill" diagonals
            dales[row + col] = 1  # "dale" diagonals

        def remove_queen(row, col):
            rows[col] = 0
            hills[row - col] = 0  # "hill" diagonals
            dales[row + col] = 0  # "dale" diagonals

        def backtrack(row = 0, count = 0):
            for col in range(n):
                if is_not_under_attack(row, col):
                    place_queen(row, col)
                    if row + 1 == n:
                        count += 1
                    else:
                        count = backtrack(row + 1, count)
```

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.

Approach 2: Backtracking via bitmap

If you're on the interview - use the approach `1`.

The next algorithm has the same time complexity $\mathcal{O}(N!)$ but works the way faster because of bitwise operators usage. Kudos for this algorithm go to takaken.

To facilitate the understanding of the algorithm, here is the code with step by step explanations.

Java Python Copy

```python
class Solution:
    def totalNQueens(self, n):
        """
        :type n: int
        :rtype: int
        """
        def backtrack(row = 0, hills = 0, next_row = 0, dales = 0, count = 0):
            """
            :type row: current row to place the queen
            :type hills: "hill" diagonals occupation [1 = taken, 0 = free]
            :type next_row: free and taken slots for the next row [1 = taken, 0 = free]
            :type dales: "dale" diagonals occupation [1 = taken, 0 = free]
            :rtype: number of all possible solutions
            """
            if row == n:  # if all n queens are already placed
                count += 1  # we found one more solution
            else:
                # free columns in the current row
                # ! 0 and 1 are inversed with respect to hills, next_row and dales
                # [0 = taken, 1 = free]
                free_columns = columns & ~(hills | next_row | dales)

                # while there's still a column to place next queen
                while free_columns:
                    # the first bit '1' in a binary form of free_columns
                    # on this column we will place the current queen
                    curr_column = - free_columns & free_columns
```

Complexity Analysis

- Time complexity : $\mathcal{O}(N!)$.
- Space complexity : $\mathcal{O}(N)$.

Rate this article:

Previous Next

Comments: 8 Sort By

Type comment here... (Markdown is supported)

Preview Post

sandh32 ★7 June 7, 2019 1:25 AM Report

Can someone explain what is the logic used for the hills and dales array? What is the logic behind int hills[] = new int[4 * n - 1]; and similarly for the dales array?

7 Share Reply

SHOW 2 REPLIES

ukmyjn889 ★5 January 14, 2019 3:32 AM

The int hills[] = new int[4 * n - 1] could simple to int hills[] = new int[2 * n] I think.
because the range of row-column is [-n,n] so we only need to add at most n to make all the elements in the hills array positive

5 Share Reply

vidyadaundkar ★1 April 12, 2019 11:38 PM

Is this problem same as 51. N-Queens?
If no whats the difference?

1 Share Reply

SHOW 2 REPLIES

lim142857 ★33 July 27, 2019 2:24 PM Report

(Python 3) (improved approach 1)

```python
class Solution:
    def totalNQueens(self, n: int) -> int:
        cols = [ 0 for column in range(n)]
```

Read More

0 Share Reply

jvanloofsvelt ★2 July 1, 2020 11:38 AM

Why is there no need to check if the columns are under attack? We do it for rows, hills and dales, but why not columns?

0 Share Reply

wjpjet ★6 May 7, 2020 8:20 PM

Why in the world do they call it `rows = [0] * n` and then pass in col as the index?

```python
def place_queen(row, col):
    rows[col] = 1
```

0 Share Reply

SHOW 1 REPLY

lenchen1112 ★972 December 21, 2019 9:37 AM Report

Cleaner Python3 version by maintaining tracked rows.

```python
# Python3
class Solution:
    def totalNQueens(self, n: int) -> int:
```

Read More

0 Share Reply

mango999 ★15 September 11, 2019 12:38 AM

In solution 1, we do row+1 for every recursive call. If the first valid queen is at index (3,0). How do we ever get to row 2, 1, and 0 for future queens since the row always increases?

0 Share Reply

SHOW 1 REPLY