n Queens

> Sections (like this one) which marked with a vertical line on the left side are commentary — discussions about the algorithm development process that wouldn't be part of a writeup.

Establish the problem.

> Defining — and understanding — the precise task to be solved is an essential prerequisite for algorithm development.

- *Specifications.*

  > State complete specifications for the problem. What is the problem? What do you start with (input) and what is the end result (output)? What are the legal input instances and the required output for each? For optimization problems, identify and distinguish between legal solutions and optimal ones.

  Place n queens on an $n \times n$ chess board so that no two queens are in the same row, column, or diagonal.

  Input: n — the number of queens and the dimensions of the board

  Output: a placement of n queens or that there's no solution

  Legal solution: queens aren't attacking each other

- *Examples.*

  > If needed, give examples (specific inputs and the corresponding outputs) of typical and special cases to clarify the specifications.

Identify avenues of attack.

- *Targets.*

  > Backtracking algorithms are fundamentally exponential.

- *Paradigms and patterns.*

  Backtracking.

  > (This is dictated.)

  > Consider the backtracking patterns.

  This is a labelling task — assign a position to each queen. The goal is to find a legal solution.

  Process input: for each queen, determine its position.

- *The series of choices.*

  > Identify the series of choices. Take into account efficiency considerations when choosing between alternatives.

  > The series of choices is to decide on the position for each queen — and there are n^2 positions on the board. As queens are placed, some of those positions become occupied and others become unavailable (because an already-placed queen would attack them), but it's not clear that it would be easy to determine just the legal choices without going through all possible choices and checking if each is legal. n^2 is very bad for a branching factor.
  >
  > Observe that it doesn't matter what order the queens are placed in. Also observe that there is exactly one queen per column (or row). As a result of these two points, we can place queens from left to right and the series of choices becomes which row to put the queen in the current column in.

Which row of column k to place queen k in.

Define the algorithm.

> Assemble the pieces of the algorithm according to the template for the particular paradigm and pattern, culminating in a statement of the algorithm itself.

- *Size.*
  > The size of the problem is the number of choices left to make. The smallest problem size is 0 (a complete solution).

  The number of queens left to place.

- *Generalize / define subproblems.*
  - *Partial solution.*
    > What constitutes a solution-so-far? This will be the same kind of thing as a legal solution for the whole problem, but less complete (fewer choices have been made).

    The placement (rows) of the queens in columns $1..k-1$.
  - *Alternatives.*
    > What are the legal alternatives for the next choice?

    The possible placements are the n rows in column k. The legal ones are those squares not attacked by queens $1..k-1$.
  - *Subproblem.*
    > Define the subproblem — the task, the input, and the output. Write this as a function definition where parameters and return values are explicitly identified. Take into account efficiency considerations when defining the intput and output.
    >
    > The input consists of the solution so far (partial solution) and the subproblem definition.
    >
    > The output can be a complete solution to the entire problem or just the solution to the subproblem.

    `nqueens(rows,k)` —
    Place queens $k..n$ in columns $k..n$ on an $n \times n$ chess board so that no two queens are in the same row, column, or diagonal.
    Input: placement *rows* of queens $1..k-1$, k (the current queen)
    Output: a placement of n queens so no queens attack each other or that there's no solution

- *Base case(s).*
  > A complete solution has been found — address what to do with it.

  The solution is complete when $k = n + 1$. Return *rows* (the solution-so-far).

- *Main case.*
  > Address how to solve a typical large problem instance (one that is not a base case).

```
for each row 1 ≤ r ≤ n
  if no queen 1..k attacks (r, k)
    rows[k] ← r    // place queen k in rows r
    result ← nqueens(rows,k+1)

    if result is a solution
      return result
    rows[k] ← ϵ    // remove queen k

return no solution
```

- *Top level.*

  The top level puts the context around the recursion.

  – *Initial subproblem.*

    Specify the inputs and parameters for the initial subproblem — the one whose solution solves the original problem.

    `nqueens(rows,1)`

  – *Setup.*

    Whatever must happen before the initial subproblem is solved.

    `initialize` *rows* `to an array of size` n

  – *Wrapup.*

    Whatever must happen to get the final answer after the solution for the initial subproblem is obtained.

    Nothing needed.

- *Special cases.*

  Make sure the algorithm works for all legal inputs — identify the cases that need to be handled and address how that handling is incorporated into the previous steps (if not already accounted for).

- *Algorithm.*

  Assemble the algorithm from the previous steps and state it.

  There shouldn't be new elements here, instead bring together the base case(s), main case, and top level with any handling needed for special cases and state the whole algorithm.

`algorithm nqueens(`n`)` —

Place n queens on an $n \times n$ chess board so that no two queens are in the same row, column, or diagonal.

Input: n — the number of queens and the dimensions of the board

Output: a placement of n queens or that there's no solution

```
initialize rows to be initially empty
nqueens(rows,1)
```

`algorithm nqueens(`*rows*`,1)` —

Place n queens on an $n \times n$ chess board so that no two queens are in the same row, column, or diagonal.

Input: n — the number of queens and the dimensions of the board

Output: a placement of n queens or that there's no solution

```
if k == n+1 then
  return rows

else
  for each row 1 ≤ r ≤ n
    if no queen 1..k attacks (r,k)
      rows[k] ← r    // place queen k in rows r
      result ← nqueens(rows,k+1)

      if result is a solution
        return result
      rows[k] ← ϵ    // remove queen k

  return no solution
```

Show termination and correctness. Show that the algorithm produces a correct solution.

- *Termination.*
  | Show that the recursion — and thus the algorithm — always terminates.

  - *Making progress.*
    > Explain why what each of your friends get is a smaller instance of the problem: in each step, another choice is made.

    The friends are asked to solve from queen $k+1$ onwards, a smaller problem than for queen k.
  - *The end is reached.*
    > Explain why a base case is always reached: eventually all choices have been made.

    One more queen is placed each time and none are placed twice, so eventually there are none left.

- *Correctness.*
  | Show that the algorithm is correct.

  - *Establish the base case(s).*
    > Explain why the solution is correct for each base case: the right thing is done with a complete solution.

    $k = n + 1$ means that queens $1..n$ have been placed, as desired. Thus the correct thing to do is return the result.
  - *Show the main case.*
    > Explain why the desired solution will be found (all possible legal alternatives for the next choice are either considered or are safe to skip) and showing that the correct partial solutions and subproblems are handed to the friends.

    There must be exactly one queen in each column of the board, so only the rows in column k must be considered for queen k. All are checked and only the ones not being attacked are considered as one of the choices.
  - *Final answer.*
    > Explain why the top level — the setup plus a correct solution to the initial subproblem followed by the wrapup — means that the final result is a correct answer to the problem. For the initial subproblem, establish that the correct input is provided – the (empty) solution-in-progress is legal and the rest of the problem is actually the whole problem.

    Initially *rows* has nothing in it, which is correct for no queens placed yet. $k = 1$, for the first queen. The board size n is treated as a global.

Determine efficiency.

| Evaluate the running time and space requirements of the algorithm.

- *Implementation.*
  > Identify data structures and, as necessary, specific implementations of those data structures to efficiently support the algorithm. Also fill in any algorithmic details that are needed in order to establish the running time.

  Copying can be avoided by passing a single array of rows for the partial solution. Each time a position is chosen, the array is updated and passed along to the friends.

  The only potentially non-$O(1)$ operation in the loop is determining whether any queen attacks (r, k). This can be checked in $O(k)$ time by going through each of the already-placed queens.

- *Time and space.*

> Assess the running time and space requirements of the algorithm given the implementation identified.

There are n possible rows for each choice (branching factor), and n choices (longest path length) — $O(n^n)$. Solving the recurrence relation $T(n) = nT(n-1) + O(k)$ yields the same; the extra $O(k)$ for each subproblem gets absorbed into the big-Oh.

- *Room for improvement.*

  > Algorithm design decisions — the series of choices pattern, the specific input and output for the subproblems — should have already been considered, but revisit them if not.

  The longest path length isn't likely to change — we have to place n queens — so the branching factor is the place to look for improvements. That also doesn't seem likely — looking for pruning opportunities is the best bet.