A sidebar indicates the "answer" for each step — it is content that is part of following the algorithm development process, separate from commentary about the process or the solution itself.

n Queens

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, 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.

Size. What is the size of the problem, and what constitutes a smaller problem? What is the simplest/smallest instance of the problem?

> The size of the problem is n, the number of queen (and the dimensions of the board).
>
> The smallest problem is $n = 1$.

Targets. What are the time and space requirements for your solution?

Tactics. The time and space constraints may narrow down the algorithmic options and/or may guide you in particular directions. Consider both things you can and can't do.

Approaches. Identify the particular flavor, if applicable, as well as the two patterns (process input and produce output) and what they look like for this problem.

> This is a labelling problem — label the queens with the position.
>
> Process input — for each queen, place it on the board.
>
> Produce output — for each board square, determine if there is a queen there.

Process input seems more viable here — n decisions with n^2 possible alternatives for each ($(n^2)^n = \mathrm{O}(n^n)$) compared to n^2 decisions with two possible alternatives for each ($2^{(}n^2) = \mathrm{O}(2^{(}n^2))$). Neither is great, but $\mathrm{O}(n^n)$ is far better.

Generalize / define subproblems. Friends get smaller versions of the original problem. Define the task, input, and output for the subproblems.

Partial solution. Identify 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 whole solution is a placement of all n queens, so a partial solution is a placement of $k \leq n$ queens.

Alternatives. Identify the choice being made and what the (legal) alternatives are for that choice. (Alternatives that result in an illegal solution aren't considered farther.)

> The choice is where to place the next queen. Any open square that doesn't attack the existing queen is a legal alternative.

Subproblem. The "rest of the problem". Define the generalized problem, its input, and its output. The input can include the partial solution so far, in which case the output would be a complete solution, or the input can only denote the subproblem, in which case the output is only the solution for the subproblem.

> Original problem: Place n queens on an $n \times n$ chess board so that no two queens are in the same row, column, or diagonal.
>
> Subproblem / generalized problem: Given k queens already on the board, place $n - k$ queens on an $n \times n$ chess board so that no two queens are in the same row, column, or diagonal.
>
> Input: $n - k$ queens left to place, placements of k queens
>
> Output: a placement of n queens or that there's no solution
>
> Legal solution: queens aren't attacking each other

Base case(s). A base case occurs when there are no more choices to make.

> There are no more choices when there are no queens left to place — $n - k = 0$.

Main case. This takes the form of: for each legal alternative for the current decision, a friend solves the remainder of the problem given the solution so far plus that choice. We return one of those solutions, the best of those solutions, or all of the those solutions depending on the structural variation.

```
for every square on the board,
  if placing a queen there doesn't attack any of the already-placed queens,
    if a friend is successful with placing n-k-1 more queens with that one added
     to the board,
      return the friend's solution as the answer
there is no answer
```

Top level. The top level puts the context around the recursion.

> **Initial subproblem.** Place n queens on an empty board.
>
> **Setup.** Nothing needed.
>
> **Wrapup.** Nothing needed — the placement is the solution.

Special cases. Make sure the algorithm works for all legal inputs — revise the previous steps to add handling for special cases as needed.

> There may not be a solution (e.g. $n = 2$) but that is accounted for already.

Algorithm. Assemble the algorithm from the previous steps and state it.

> `nqueens(n)` — place n queens on an $n \times n$ 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
>
> ```
> nqueens(n,0,{}) // place n queens on a empty board
> ```
>
> `nqueens(n,k,placements)` — place $n - k$ queens on an $n \times n$ board already containing the specified placements so that no two queens are in the same row, column, or diagonal
>
> Input: $n - k$ queens left to place, placements of k queens
>
> Output: a placement of n queens or that there's no solution
>
> ```
> if n-k = 0
>   return the placements as the solution
> else
>   for every square on the board,
>     if placing a queen there doesn't attack any of the already-placed queens,
>       solution ← nqueens(n,k+1,placements ∪ square)
>       if there is a solution
>         return solution
>   there is no answer
> ```

Termination. Show that the recursion ends. For making progress, explain why each subproblem is smaller. For reaching the end, show that a base case is always reached.

> **Making progress.** The friends are asked to place $n - k - 1$ queens instead of $n - k$, so the problem is smaller.
>
> **Reaching the end.** One more queen is placed each time, so eventually there are none left.

Establish the base case(s). Explain why the solution is correct for each base case.

> Only legal alternatives were considered, so a complete solution is a legal solution.

Show the main case. Explain why all possible alternatives for the next choice are covered and that the right partial solution is passed to each friend. Then, assuming that the friends return the correct results for their subproblem, explain why the correct answer is produced from those results.

> All board positions are considered, and only those that are attacking are skipped.
>
> The friends return a complete (legal) placement if there is one, so that's our solution too.

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.

> The initial subproblem is the original problem, and the placements found is the answer. There is no setup to consider.

Implementation. Identify data structures and, as necessary, specific implementations of those data structures to efficiently support the algorithm. Also fill in any algorithmic steps that haven't been specified.

We pass a collection of placements plus a new placement to the friends. Copying can be avoiding by having a single collection object and adding the new placement to that collection. (It must then be removed when the friend is done, before adding the next placement alternative for the next friend.) Add/remove is commonly O(1) for unordered collections.

Since we stop when the first solution is found, the same collection object can be returned as the solution — no need to copy there either.

For each board position, checking for a possible placement means seeing if there are any other queens in a particular row, column, and diagonal — $O(k)$ for k queens already placed.

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

With the implementation described, the work other than the recursive calls for each subproblem is $O(k)$.

The branching factor is n^2 (n^2 board positions to consider) and the longest path length is n (the number of queens to place), leading to $\Theta(n^n)$ overall — the extra $O(k)$ per subproblem gets absorbed.

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.