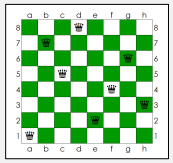
Greedy Isn't Always Good

Greedy algorithms are typically efficient (polynomial time) but there are common situations where they don't work –

- it may not be possible to obtain a globally optimal solution via only locally optimal choices
  - e.g. 0-1 knapsack

- it may be hard to come up with a plausible greedy choice
  - e.g. n queens – place *n* queens on an *nxn* chess board so that no row, column, or diagonal contains more than one queen



- you may want to enumerate all possibilities
  - e.g. generating anagrams

 https://commons.wikimedia.org/wiki/File:Solution_C_for_8_Queen_Puzzles.png

Backtracking

- applicable for series-of-choices problems where it is necessary to consider more than one alternative for each choice
  - fall back on exhaustive search when the right choice at the moment depends on what else can happen later – have to try all the possibilities
- implementation is recursive, using case analysis

- **2+-friend solutions**, where there is more than one subproblem at each level
  - **divide-and-conquer**
    The problem is split into b subproblems of size n/b where $b \geq 2$ (typically 2).
  - **case analysis**
    Each friend considers a different choice.

Backtracking Formulation

Case analysis –

We choose each alternative for the current decision in turn and then ask the friend to solve the rest of the problem in light of that choice.

- if the friend succeeds, we have a solution

- the friend's solution is constrained by the partial solution
  - partial solution is passed explicitly or implicitly (by the construction of the subproblem)

Developing Backtracking Algorithms

- combines series of choices and recursive paradigms
- development process – key elements
  - avenues of attack
    - known targets
      - backtracking is inherently exponential – $O(b^h)$
        - b = branching factor (number of alternatives for each choice), h = longest solution length
    - identify the series of choices
      - process input and produce output patterns
      - take running time into account – the formulation of the series of choices determines the branching factor and the longest solution length
    - identify the alternatives for each choice
      - identify all legal alternatives instead of only the greedy choice
  - define the algorithm
    - similar to the general recursive process
    - main case structure – case analysis pattern + applicable structural variation (one solution, best solution, all solutions)
  - emphasis is on establishing the problem, assembling the algorithm, showing the main case, and time
    - most of the proving correctness steps are either trivial or always the same given our development process framework

Find a legal solution. If the subproblem output is a complete solution:

```
for each legal alternative for the current choice,
  result = solve the subproblem resulting from choosing that alternative

  if result is a solution,
    return it

return no solution
```

If the subproblem output is just the subproblem solution, then `return result+alternative` instead of just the result.

Find the best solution. If the subproblem output is a complete solution:

```
best result so far ← no solution

for each legal alternative for the current choice,
  result = solve the subproblem resulting from choosing that alternative

  if result is a solution and better than the best result so far,
    best result so far ← result

return the best result so far
```

If the subproblem output is just the subproblem solution, then replace `result` with `result+alternative` when checking whether the current alternative results in a better solution and when updating the best result so far.

Find all solutions.

```
for each legal alternative for the current choice,
  solve the subproblem resulting from choosing that alternative
```

The base case handles the complete solutions (outputting them or adding them to a collection of solutions).

N Queens