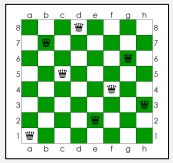
Greedy Isn't Always Good

Greedy algorithms are typically efficient (polynomial time) but there are many 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 when greedy doesn't work
  - it is necessary to consider more than one alternative
- 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

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)

The other way around – we ask the friend to solve a smaller version of the problem and then we choose an alternative and add that to the solution – doesn't work.

- if we have several legal alternatives, which do we pick?
  - the discovery that our choice was bad comes after we've returned our answer

- the friend's choices are only constrained by the partial solution, so we have no way to direct them to come up with a different solution

Key Points – Backtracking

- (essentially) exhaustive search applies when greedy fails – the right choice at the moment depends on what else can happen later
- development process
  - very similar to the general recursive process
  - structure of the main case for case analysis
  - adds "partial solution" and "alternatives" as part of generalize / define subproblems
  - 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
- the formulation of the series of choices determines the branching factor and the longest path length
  - this impacts the running time – need to take into account during formulation of the algorithm and not only at the end
- the three structural variations (one solution, best solution, all solutions) and the corresponding code structure

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).