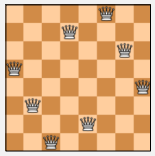
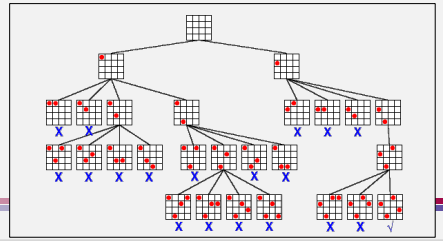
Recursive Backtracking

- recursive backtracking
  - generate the solution through a series of decisions
  - enumerate all possible combinations of outcomes
- n queens
  - place *n* queens on an *nxn* chess board so that no two queens share a row, column, or diagonal
  - as a series of decisions – where to put the queen in column *i*
  - structure – for each possible row for column *i*, put a queen there and ask a friend to place the rest of the queens





Recursion vs Loops

Both recursion and loops involve repetition.

Why recursion?

- problem may be defined recursively (e.g. fibonacci)
  - it is natural to reflect that definition in code
- it may be easier to come up with a recursive solution than an iterative one (e.g. towers of hanoi)
  - divide-and-conquer approach
- solves the explosion of fingers problem
  - recursive backtracking approach

Note that recursion does not make it possible to solve any new problems.

  - (it is always possible to write a non-recursive solution, though it may be much more complex)

Recursion vs Loops

It is generally better to use loops unless recursion is called for.

  - function calls involve overhead
  - loops can be less confusing to understand than recursion

Typically use loops when

  - the task calls for repetition
  - the recursive case only involves one recursive call and that call is the last thing done (e.g. factorial) – *tail recursion*

Typically use recursion when

  - the problem is defined recursively
  - the recursive case involves multiple recursive calls (multiple subproblems – explosion of fingers)
    - though sometimes loops are preferable even in this case (e.g. fibonacci)