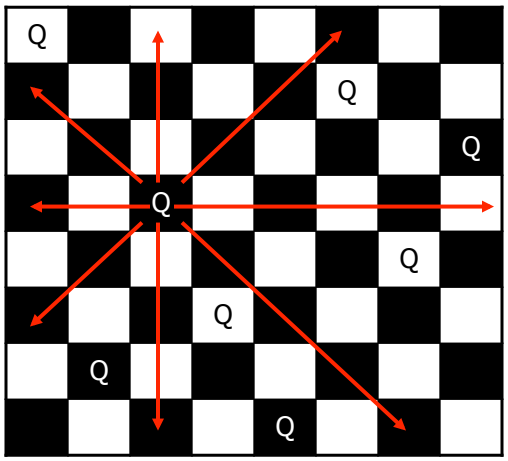
CSE 143
Lecture 16

More Recursive Backtracking

slides created by Marty Stepp and Ethan Apter
http://www.cs.washington.edu/143/

The "8 Queens" problem

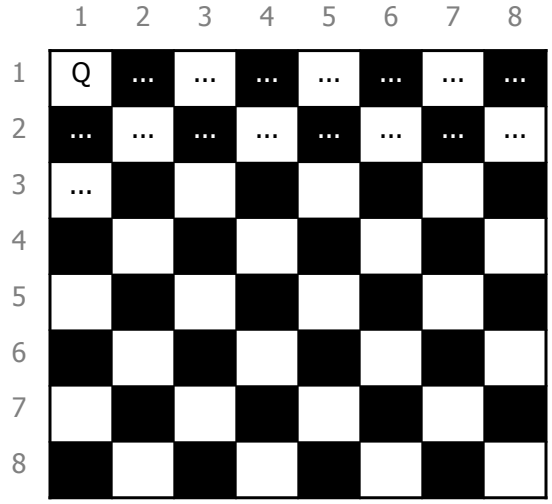
- Consider the problem of trying to place 8 queens on a chess board such that no queen can attack another queen.
  - What are the "choices"?
  - How do we "make" or "un-make" a choice?
  - How do we know when to stop?



2

Naive algorithm

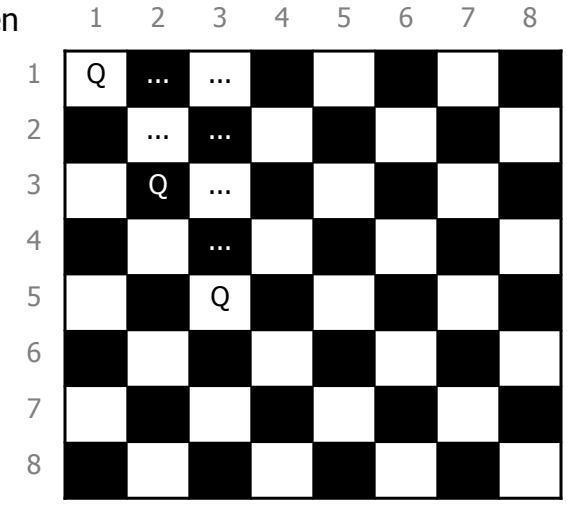
- for (each square on board):
  - Place a queen there.
  - Try to place the rest of the queens.
  - Un-place the queen.
  - How large is the solution space for this algorithm?
    - 64 * 63 * 62 * ...



3

Better algorithm idea

- Observation: In a working solution, exactly 1 queen must appear in each row and in each column.
  - Redefine a "choice" to be valid placement of a queen in a particular column.
  - How large is the solution space now?
    - 8 * 8 * 8 * ...



4

Exercise

- Suppose we have a `Board` class with the following methods:

Method/Constructor	Description
`public` **`Board`**`(int size)`	construct empty board
`public boolean` **`isSafe`**`(int row, int column)`	`true` if queen can be safely placed here
`public void` **`place`**`(int row, int column)`	place queen here
`public void` **`remove`**`(int row, int column)`	remove queen from here
`public String` **`toString`**`()`	text display of board

- Write a method `solveQueens` that accepts a `Board` as a parameter and tries to place 8 queens on it safely.
  - Your method should stop exploring if it finds a solution.

Exercise solution

```
// Searches for a solution to the 8 queens problem
// with this board, reporting the first result found.
public static void solveQueens(Board board) {
    if (!explore(board, 1)) {
        System.out.println("No solution found.");
    } else {
        System.out.println("One solution is as follows:");
        System.out.println(board);
    }
}

...
```

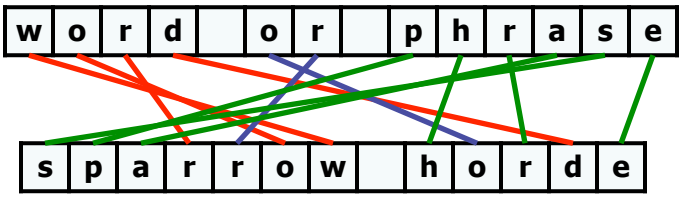
Exercise solution, cont'd.

```
// Recursively searches for a solution to 8 queens on this
// board, starting with the given column, returning true if a
// solution is found and storing that solution in the board.
// PRE: queens have been safely placed in columns 1 to (col-1)
public static boolean explore(Board board, int col) {
    if (col > board.size()) {
        return true;   // base case: all columns are placed
    } else {
        // recursive case: place a queen in this column
        for (int row = 1; row <= board.size(); row++) {
            if (board.isSafe(row, col)) {
                board.place(row, col);           // choose
                if (explore(board, col + 1)) {   // explore
                    return true;   // solution found
                }
                b.remove(row, col);              // un-choose
            }
        }
        return false;   // no solution found
    }
}
```


Anagrams

- **anagram:** a rearrangement of the letters from a word or phrase to form another word or phrase

- Consider the phrase "word or phrase"
  - one anagram of "word or phrase" is "sparrow horde"



AnagramSolver

- Consider the phrase "Ada Lovelace"
- Some anagrams of "Ada Lovelace" are:
  - "ace dale oval"
  - "coda lava eel"
  - "lace lava ode"
- We could think of each anagram as a list of words:
  - "ace dale oval" → [ace, dale, oval]
  - "coda lava eel" → [coda, lava, eel]
  - "lace lava ode" → [lace, lava, ode]

9

AnagramSolver

- Consider also the following dictionary:

ail	gnat	run
alga	lace	rung
angular	lain	tag
ant	lava	tail
coda	love	tan
eel	lunar	tang
gal	nag	tin
gala	natural	up
giant	nit	urn
gin	ruin	

10

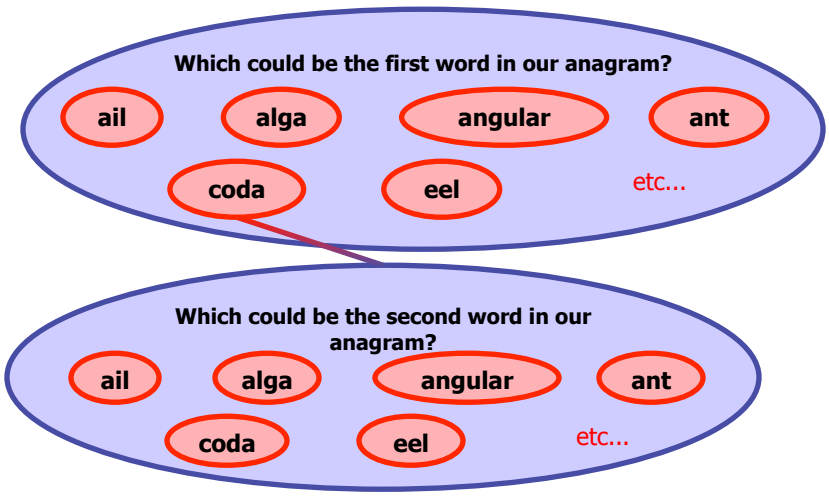
AnagramSolver

- Which is the first word in this list that *could* be part of an anagram of "Ada Lovelace"
  - ail
    - no: "Ada Lovelace" doesn't contain an "i"
  - alga
    - no: "Ada Lovelace" doesn't contain a "g"
  - angular
    - no: "Ada Lovelace" doesn't contain an "n", a "g", a "u", or an "r"
  - ant
    - no: "Ada Lovelace" doesn't contain an "n" or a "t"
  - coda
    - yes: "Ada Lovelace" contains all the letters in "coda"

11

AnagramSolver

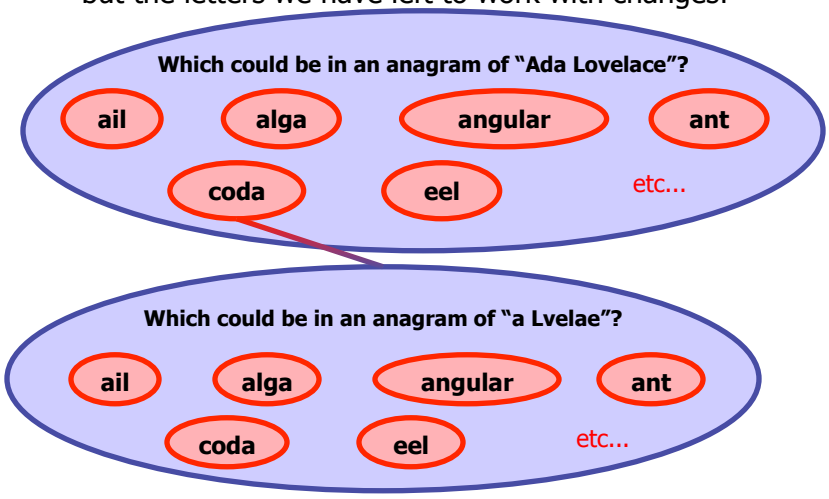
- This is just like making a choice in recursive backtracking:



12

AnagramSolver

- At each level, we go through all possible words
  - but the letters we have left to work with changes!



13

Low-Level Details

- There are some low level details here in deciding whether one phrase contains the same letters as another
- Just like 8 Queens had the Board class for its low-level details, we'll have a class that handles the low-level details of **`AnagramSolver`**
- This low-level detail class is called **`LetterInventory`**
  - Now where have I seen that before…

14

AnagramSolver

- Key questions to ask yourself on this assignment:
  - When am I done?
    - for 8 Queens, we were done when we reached column 9
  - If I'm not done, what are my options?
    - for 8 Queens, the options were the possible rows for this column
  - How do I make and un-make choices?
    - for 8 Queens, this was placing and removing queens

15

AnagramSolver

- You must include two optimizations in your assignment
  - because backtracking is inefficient, we need to gain some speed where we can
- You must preprocess the dictionary into `LetterInventory`s
  - you'll store these in a Map
    - specifically, in a HashMap, which is slightly faster than a TreeMap
- You must prune the dictionary before starting the recursion
  - by "prune," we mean remove all the words that couldn't possibly be in an anagram of the given phrase
  - you need do this only once (before starting the recursion)

16