Lecture 15: Recursive backtracking, 8 Queens

- The 8-queens problem
  - Challenge: put 8 queens on an 8x8 chessboard such that no 2 are threatening each other
  - Remember, queens can move vertically, horizontally, diagonally
  - So what are our choices?
    - Placing a queen on a space
    - So where can you place the first queen? In any of the 64 spots?
    - Then 63 spots for the second queen, 62 for the third queen...
    - 1.8x10^14 possibilities - we can do better
    - Instead, we can place a queen in the first column - if there is a solution, it has 1 queen in each column
  - Again, we could use nested loops from 1-8 to do this, but we'll do it more elegantly with recursive backtracking
- The backtracking will be simpler if we abstract-away some of the supporting code: the Board
  - We'll be placing queens on a board, so we can create a board class to store that information
  - What methods might you want to have from a board?
    - place
    - remove
    - safe
    - print
    - size
  - Assume you have such a board
- We have some starter code that prompts a user for a value of n and constructs a board
  - So how is the recursion going to work? All solutions or just one solution?
    - Just one solution in this case
    - That means we should stop exploring when we find a solution
    - Our recursive helper should be able to let us know if it found a solution or not - boolean return type
  - We'll obviously need a helper method - common case
    - What parameters does it need?
    - The Board
    - Anything else? We can figure that out later
  - Every level of the decision tree will be handled by a different recursive call
    - One method call will handle col 1, one will handle col 2, etc.
    - So what will the method need to know besides the board? The column!
    - Add column as a parameter to the private helper
  - We also don't want to waste time exploring dead ends
    - What kinds of dead ends might we have?
    - For example, if we don't place 1,2,3 safely, then does it make sense to try to place a queen in column 4?
    - so we add a precondition:

```
// pre: queens have been safely placed in previous columns
```

- Now we're ready to write our code
    - What's our base case? What column could we get to that would be really easy to know if we found a solution?
        - Column 9, because that means columns 1-8 have been placed correctly
    - What do we do if we get to column 9?
        - Return true! We've found an answer
        - Also, don't hard-code 8 - use the size of the board
    - Let's say we've safely placed queens 1-4, and are now placing queen 5
        - What options do we have to explore at this level? Each row!
        - We have multiple possibilities, so how can we explore each possibility
        - Use a for-loop to try each row
        - (notice: using iteration INSIDE of recursion)
        - Then choose, explore, unchoose
- Solution

```
public static void solve(Board solution) {
    if (!explore(solution, 1)) {
        System.out.println("No solution.");
    } else {
        System.out.println("One solution is as follows:");
        solution.print();
    }
}
public static boolean explore(Board b, int col) {
    if (col > b.size()) {
        return true;
    } else {
        for (int row = 1; row <= b.size(); row++) {
            if (b.safe(row, col)) {
                b.place(row, col);
                if (explore(b, col + 1)) {
                    return true;
                }
                b.remove(row, col);
            }
        }
        return false;
    }
}
```

- Execute a trace for 4-queens

- The next programming assignment: using recursive backtracking to find anagrams
    - As an example, construct a short dictionary: bee, go, gush, shrug
    - Run the program with this dictionary and “george bush”
    - In recursive backtracking, we have a set of possibilities, and then we make a series of choices from those possibilities
        - In 8-queens, we could choose what row to put a queen, and we made a choice for each column
        - What are our possibilities, and set of choices, in this case?
        - Possibilities = words in the dictionary
        - Choices = each word in the anagram
    - How do we make progress towards being “done”?
        - In 8-queens, we moved forward column-by-column
        - Here we move forward word-by-word
    - How do we know when we’re done?
        - In 8-queens, when we’ve placed all the columns
        - Here, when we’ve used up all our letters
        - So as we place words, we’re also using up letters
- Keeping track of letters - what does that remind you of?
    - LetterInventory!
    - “choosing” a word --> removing the letters from the inventory --> subtracting
    - “no more letters” --> empty inventory
- When can we prune a branch e.g. we know we can’t find a solution this way?
    - 8-queens: no safe place for a queen, here can’t subract
- Let’s go through a trace of “george bush” ([beegghorsu])

```
text=[beegghorsu]
    bee, text=[gghorsu]
        bee
        go, text=[ghrsu]
            bee
            go
            gush, text=[r]
                bee
                go
                gush
                shrug
            shrug, text=[]
                print [bee, go, shrug]
```

- Some other points
    - We’re going to be needing the letter inventory associated with each word in the dictionary a ton - so we only compute it once
    - There’s also going to be a ton of irrelevant words (ones that cannot possibly be part of the anagram)
        - We’ll “prune” out these words before starting to backtrack to find anagrams of a

phrase

- This is not very much code, but very tricky to understand.