• No class Friday, April 10 (Good Friday)
• No class Monday, April 13 (Easter Monday)
• Wednesday, April 15 devoted to review
• Friday April 17: midterm
• No class Monday, April 20 (Patriots Day)
• Four more classes, April 22,24,27,29
Recursive Backtracking
Recursive Backtracking

An important application of recursion, illustrated here with a famous puzzle—the 8 Queens problem.
This problem has a chessboard in it, but it’s not really about chess, and you only have to know one rule of the game.
...the Queen in chess can move horizontally, vertically, or diagonally, and can capture any piece that is in her way.
Is it possible to place eight queens on a chessboard so that none of them attacks any other?
The 8 Queens Problem

Is it possible to place eight queens on a chessboard so that none of them attacks any other?

Note that we can’t do more than eight, because then we would have two queens in the same row.
The 3 Queens Problem

Before we try to solve this with a computer, we’ll explore some smaller versions of the problem by hand.
Let’s put a queen on the first cell of the first row, and try to fill in the subsequent rows...
The 3 Queens Problem

This doesn’t work
The 3 Queens Problem

Neither does this.
The 3 Queens Problem

But this is promising. Let’s try to do the third row.
The 3 Queens Problem

But none of the three possibilities for the third row works. We’ve exhausted all the possibilities that we could try from the queen in the first cell of the first row, so let’s backtrack to the first row....
The 3 Queens Problem

...and try the second cell.
The 3 Queens Problem

...but you can see that there’s no place to go in the second row.
The 3 Queens Problem

We can backtrack again to the first row, and try the last cell in that row....
The 3 Queens Problem

...but we really don’t need to, because we’ll just see exactly what happened with the first cell in the first row, in a kind of mirror image.
The result of our exertions is that we’ve proved that the 3 Queens problem is impossible. There’s no solution.
We use the same approach, and start with a queen in the first row, first column.
We use the same approach, and start with a queen in the first row, first column.

Try it! You’ll get this far, and then get stuck on the last row. So we backtrack to the first row....
The 4 Queens Problem

...and try the second cell...
The 4 Queens Problem

Success!
Recursion will help us manage the intricate trial and error of backtracking.
Where’s the recursion?
To solve this puzzle, where we have to place queens on four rows....
We try in turn to solve each of these smaller puzzles, where we have to place queens on three rows...
And to solve this puzzle...
...we try in turn to solve each of these

The first three obviously have no solution, but to solve the fourth....etc.
...we try in turn to solve each of these

The recursion bottoms out when every row of the board has a queen on it, and we either have a solution or a forbidden configuration (two queens attacking each other).
How the code is constructed

We call a board in which some of the rows contain a queen, a configuration. This includes the empty board in which no rows contain a queen, and a completed board, in which all rows have a queen; but also cases like a board with 8 rows in which the first 5 rows contain a queen and the last three are empty.
How the code is constructed

The heart of the program is a function `solve(configuration)` which returns a solution to the puzzle if one can be found, starting from the given configuration, and returns nothing (in Python, `None`) if no solution exists.
How the code is constructed

The goal of the puzzle is to reach a configuration in which every row is filled, and which is legal (no queens attacking one another). This results in the following ‘pseudocode’:

```python
def solve(configuration):
    if configuration is complete: #(every row is filled)
        if legal(configuration):
            return configuration #(this is the solution)
    elif configuration is legal:
        for each column c:
            put a queen on column c of the next free row, giving new configuration
            d=solve(new configuration)
            if d != None:
                return d
```
How the code is constructed

To flesh this out, it remains to decide how to represent a configuration by a Python data structure, and how to test if a configuration is legal.
Representing a configuration

(3, (0, 1))
Representing a configuration

(4, (1, 3, 0, 2))
Testing if a configuration is legal

Same column: column appears twice in the second component of the configuration.

(8,(2,0,5,2))
Testing if a configuration is legal

Same diagonal: absolute difference between columns equals absolute difference between rows.

(8,(0,6,1,3)) :

config[1][3]-config[1][0]=3-0

(8,(0,6,1,4)):

config[1][1]-config[1][3]=3-1
Using tuples rather than lists is crucial: We produce a new configuration by

```
newconfig=(oldconfig[0],oldconfig[1]+(j,))
```

If we used lists instead and updated with a mutable list method like `append`, the present code would not work, and a correct solution would be more complicated.
Show the code
>>> solve((8, ()))
(8, (0, 4, 7, 5, 2, 6, 1, 3))
This is a general framework, that can be used for a wide variety of backtracking problems:

```python
def solve(configuration):
    if there is no successor configuration:
        if legal(configuration):
            return configuration (this is the solution)
        elif legal(configuration):
            for each successor configuration succ:
                c = solve(succ)
                if c != None:
                    return c
```

Representation of a configuration, what is ‘legal’, and what constitutes a ’successor configuration’ depends on the application (e.g. Sudoku, maze solving).