CSCI-1200 Data Structures — Spring 2015
Lecture 15 — Problem Solving Techniques, Continued

Review of & Finishing Lecture 14

- Non-linear maze search (leftover from Lecture 13)
- Efficient Occurrences (Test 2, problem 3)
- General Problem Solving Techniques:
  1. Generating and Evaluating Ideas
  2. Mapping Ideas into Code
  3. Getting the Details Right (Today!)
- Small exercises to practice these techniques
- Generate Ideas for HW6 (Today!)
- Map Ideas into Code for Merge Sort (Today!)
- Getting the Details Right for Non-Linear Word Search (Today!)
- Problem Solving Strategies / Checksheet (Today!)
- Design Example: Conway's Game of Life (Today!)
- Another Design Example: Inverse Word Search (Today!)

15.1 Another Example: Inverse Word Search

Let's flip the classic word search problem and instead *create the board* that contains the specified words! We'll be given the grid dimensions and the set of words, each of which must appear in the grid, in a straight line. The words may go forwards, backwards, up, down, or along any diagonal. Each grid cell will be assigned one of the 26 lowercase letters. We may also be given a set of words words that should *not* appear anywhere in the grid. Here's an example:

Input:

```
4 4
+ arts
+ arid
+ east
+ rest
- ear
- at
- sit
```

r	e	a	a
e	a	r	r
s	s	i	t
t	t	d	s

e	r	a	a
a	e	r	r
s	s	t	i
t	t	s	d

In the middle above, is an *incorrect* solution. Though it contains the 4 required words, it also contains two of the forbidden words. The solution on the right is a fully correct solution. This particular problem has 8 total solutions including rotations and reflections.

Here's another example:

```
5 3
+ echo
+ baker
+ apt
+ toe
+ ore
+ eat
+ cap
```

e	c	h	o	o
b	a	k	e	r
a	p	t	o	e

And a couple more puzzles:

```
3 3
+ ale
+ oat
+ zed
+ old
+ zoo
```

```
7 5
+ avocado
+ magnet
+ cedar
+ robin
+ chaos
+ buffalo
+ trade
+ lad
+ fun
- ace
- coat
```

15.2 Generating Ideas

- If running time & memory are not primary concerns, and the problems are small, what is the simplest strategy to make sure all solutions are found. Can you write a *simple* program that tries *all possibilities*?

- What variables will control the running time & memory use of this program? What is the order notation in terms of these variables for running time & memory use?

- What incremental (baby step) improvements can be made to the naive program? How will the order notation be improved?

15.3 Mapping Ideas to Code

- What are the key steps to solving this problem? How can these steps be organized into functions and flow of control for the main function?

- What information do we need to store? What C++ or STL data types might be helpful? What new classes might we want to implement?

15.4 Getting the Details Right

- What are the simplest test cases we can start with (to make sure the control flow is correct)?

- What are some specific (simple) corner test cases we should write so we won't be surprised when we move to bigger test cases?

- What are the limitations of our approach? Are there certain test cases we won't handle correctly?

- What is the maximum test case that can be handled in a reasonable amount of time? How can we measure the performance of our algorithm & implementation?