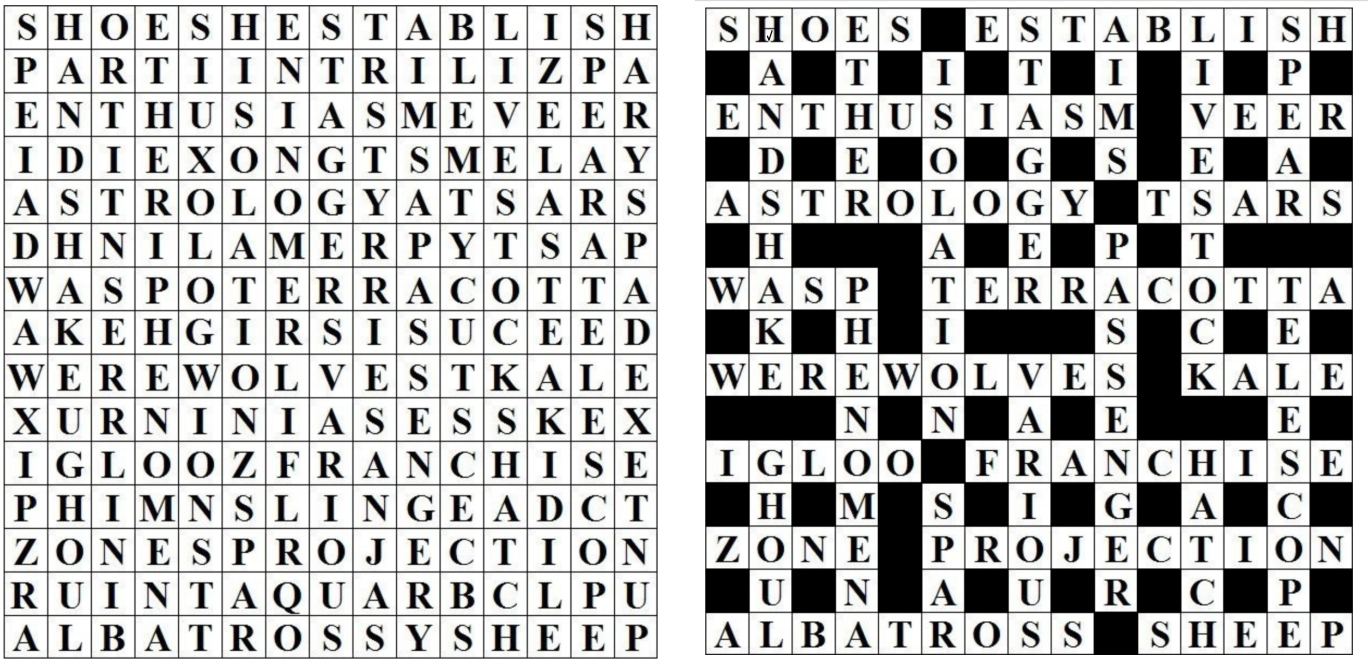
CSCI-1200 Data Structures — Spring 2019
Homework 6 — Crossword Blackout

In this homework we will work with concepts inspired by crosswords, however we will not be following all the rules that would be required by a proper "American-style" or "British-style" crossword. As such, you should read the entire handout carefully. Crosswords are quite popular, so there is lots of material available online. You may not search for, study, or use any outside **code** related to crossword puzzles. You are welcome to find additional puzzles to experiment with and try to solve by hand.

The basic goal is to consider a two dimensional grid in which each square either contains a letter or a black square, and find one or all boards that meet some requirements (described below) and have all other squares blacked out. Any sequence of two or more squares touching is considered a *word*. In order for a HW6 solution to be valid it must contain no words shorter than 3 letters, and all words must be in a provided dictionary (more on this later). Words only run in two directions, *across* (meaning left-to-right) and *down* (meaning top-to-bottom). A letter in a square can belong to an across word, a down word, or both an across and down word. A letter in a square is never part of two or more across words at the same time, or two or more down words at the same time. Finally, in a real crossword grid, all the words must be "connected", and black squares must be placed to ensure symmetry, but neither of those are requirements for the baseline HW6.

A full-size example puzzle (left) and solution (right) might look like:



Source: https://www.sporcle.com/games/hockeystix3/blackout-2

Crossword Blackout Arguments

Your program will accept four (baseline) or five (extra credit) command line arguments. The extra credit will be explained near the end of the handout, for now we focus on the baseline HW6 requirements.

Execution looks like: `./a.out [dictionary file] [initial grid file] [solution mode] [output mode] [gc]`

Dictionary File

The dictionary file consists of words that only use upper-case letters, with one word per line. For a solution to be legal (allowed), all words in the solution must be in the dictionary file. Words in the dictionary are unique, and words can only appear up to one time per solution.

Initial Grid File

The initial grid file describes the puzzle to work with. There are three types of lines that appear in the input. There are no spaces in any of the lines. You should not make any assumptions about the order the different types of lines will appear in the input. Black squares, represented by "#", may appear in the input and output.

- **Comments**: Comment lines start with "!" and should be ignored by your program
- **Length constraints**: Constraint lines start with "+" followed by a positive integer.
- **Grid lines**: Lines that are not comments and not constraint lines are one row of the input puzzle's grid. These are presented in order.

Every puzzle should have one or more constraints which represent a required word length in the solution. Any legal solution must have one matching word per constraint. For example if "+4" appears twice in the input file, then all legal solutions must have two 4-letter words.

Solution Mode

The solution mode will either be *one_solution* meaning you should only print up to one solution, or it will be *all_solutions* meaning you should print all solutions that satisfy the inputs.

Output Mode

The output mode will either be *count_only* in which case you will only print the number of solutions you found (just the first line of output from the example in the section below), or *print_boards* in which case you should print the count and print all solutions.

Output Formatting

An example output for a puzzle with 3 solutions is:

```
Number of solution(s): 3
Board:
B#F
L#L
USE
E#E
S#S
Board:
B#F
L#L
U#E
EKE
S#S
Board:
B#F
L#L
U#E
E#E
SIS
```

To ensure full credit on the homework server, please format your solution exactly as shown above. Solutions may appear in any order, but the first line must start with `Number of solution(s):` then a space and the number of solutions. Each solution should start with a line that says `Board` followed by one row of the board per line, starting from the top row.

Additional Requirements: Recursion, Order Notation, & Extra Puzzles

You must use recursion in a non-trivial way in your solution to this homework. As always, we recommend you work on this program in logical steps. Partial credit will be awarded for each component of the assignment. Your program should do some error checking when reading in the input to make sure you understand the file format. *IMPORTANT NOTE: This problem is computationally expensive, even for medium-sized puzzles with too much freedom! Be sure to create your own simple test cases as you debug your program.*

Once you have finished your implementation, analyze the performance of your algorithm using order notation. What important variables control the complexity of a particular problem? The dimensions of the board (w and h)? The number of words in the dictionary (d)? The total number of spaces with a letter (l)? The total number of blacked out spaces (b)? The number of constraints (c)? Etc. In your `README.txt` file write a concise paragraph ($<$ 200 words) justifying your answer. Also include a simple table summarizing the running time and number of solutions found by your program on each of the provided examples.

You should include 1-3 new puzzles and at least 1 dictionary for each of your new puzzles that either helped you test corner cases or experiment with the running time of your program. Make sure to describe these puzzles in your README.

You can use any technique we have covered in Lectures 1-14, Homework 1-5, and Lab 1-7. This means you cannot use STL `pair`, `map`, `set`, etc. on this homework assignment.

You must do this assignment on your own, as described in the "Collaboration Policy & Academic Integrity" handout. If you did discuss this assignment, problem solving techniques, or error messages, etc. with anyone, please list their names in your `README.txt` file.

NOTE: If you earn 7 points on the homework submission server for tests 3 through 10 by 11:59pm on Wednesday, March 13, you may submit your assignment on Friday, March 14 by 11:59pm without being charged a late day.

Extra Credit

For extra credit, your program should function the same as in the baseline case when given four arguments. However if a fifth argument is given, *gc*, then you should only print boards that are a "giant component". What this means is that starting from any letter, you should be able to use a series of up, down, left, and right moves to reach all other letters in the board without having to go through a blacked out ("#") square.

```
#######
#LABRAT
R######
E######
D######
```

Legal for baseline HW6, but not valid for extra credit if *gc* is given as 5^{th} argument. There is no way to get from "LABRAT" to "RED" without touching a blacked out cell or using a diagonal movement.

```
#######
#LABRAT
####E##
####D##
#######
```

Legal for baseline HW6 and extra credit, all letters can be reached by only going up, left, down, or right without having to use a blacked out cell.