# CSCI-1200 Data Structures — Spring 2023
# Homework 6 — Inverse Word Search Recursion

In this homework we will build an inverse word search program using the techniques of recursion. The goal is to construct a grid of letters that one can search to find specific words. Understanding the non-linear word search program from Lectures 12 & 13 will be helpful in thinking about how you will solve this problem. We strongly urge you to study and play with that program, including tracing through its behavior using a debugger or `cout` statements or both. *Please read the entire handout before beginning your implementation.*

## Your Tasks

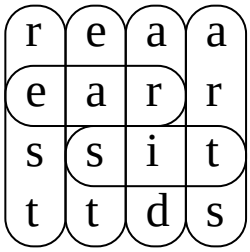
For this assignment, you will be given the dimensions (width and height) of a word search puzzle, a set of words that should appear in the grid (forwards, backwards, up, down, or along any diagonal), and optionally a set of words that should not appear anywhere in the grid. Each grid cell will be assigned one of the 26 lowercase letters. Note that unlike the non-linear word search problem we discussed in class, we will only allow words that appear in a straight line (including diagonals). Your task is to output all unique word search grids that satisfy the requirements. Rotations and mirroring of the board will be considered unique solutions.

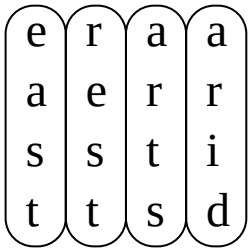
Your program should expect three command line arguments, the name of the input file, the name of the output file, and a string:

```
inverse_word_search.exe puzzle2.txt out2.txt one_solution
inverse_word_search.exe puzzle2.txt out2.txt all_solutions
```

The third argument indicates whether the program should find all solutions, or just one solution. Here's an example of the input file format:

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

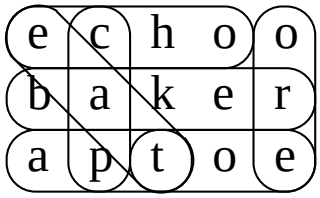



The first line specifies the width and height of the grid. Then each line that follows contains a character and a word. If the character is '+', then the word must *appear* in the grid. If the character is '-', then the word must *not appear* in the grid. For this first example we show an incorrect solution on the left. 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 solutions including rotations and reflections.

Below is a second example that specifies only positive (required) words. This puzzle has 4 solutions including rotations and reflections.

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




When asked to find all solutions, your program should first output the number of solutions and then an ASCII representation for each solution. See the example output on the course webpage. You should follow this output closely, however your solutions may be listed in a different order. When asked to find just one solution, your program should just output the first legal solution it finds (it does not need to count the number of solutions, nor does it need to be the first solution shown in our output). If the puzzle is impossible your program should output "`No solutions found`".

To implement this assignment, you must use recursion in your search. First you should tackle the problem of finding and outputting one legal solution to the puzzle (if one exists).

### Algorithm Analysis

For larger, more complex examples, this is a really hard problem. Your program should be able to handle the small puzzles we have created in a reasonable amount of time. You should make up your own test cases as well to understand this complexity. Include these test cases with your submission (they will be graded). Summarize the results of your testing, which test cases completed successfully and the approximate "wall clock time" for completion of each test. The UNIX/WSL `time` command can be prepended to your command line to estimate the running time:

```
time inverse_word_search.exe puzzle1.txt out1.txt one_solution
```

Once you have finished your implementation and testing, analyze the performance of your algorithm using order notation. What important variables control the complexity of a particular problem? The width & height of the grid ($w$ and $h$), the number of required words ($r$), the number of forbidden words ($f$), the number of letters in each word ($l$), the number of solutions ($s$)? In your *plain text* `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. *Note: It's ok if your program can't solve the biggest puzzles in a reasonable amount of time.*

**IMPORTANT**: All students are required to submit their program to the Homework 6 contest (see below). Extra credit will be awarded for programs that have a strong performance in the contest. 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.

**FINAL NOTE:** If you earn 6 points between Test Cases 3, 4, 5, 6 on Submitty by 11:59pm on Wednesday March 15th, you may submit your assignment on Friday March 17th without being charged a late day.

### Homework 6 Inverse Word Search Contest Rules

- Contest submissions are a separate homework submission. Contest submissions are due March 24th at 11:59pm. You may not use late days for the contest. (The regular homework deadline is Thursday Mar 16th at 11:59pm and late days are allowed for the regular homework submissions.)
- You may submit the same code for both the regular homework submission and the contest. Or you may make a small or significant change for the contest.
- Contest submissions do not need to use recursion.
- Contest submissions must follow the output specifications and match the formatting of the examples posted on the course webpage.
- We will compile your code with optimizations enabled (`g++ -O3 *.cpp`). Programs that do not compile, or do not complete the basic tests in a reasonable amount of time with correct output, will not receive extra credit.
- Programs must be single-threaded and single-process.
- We will run your program by *redirecting* `std::cout` to a file and measure performance with the UNIX `time` command. For example:

```
time inverse_word_search.exe puzzle1.txt out_puzzle1.txt one_solution
time inverse_word_search.exe puzzle1.txt out_puzzle1_all.txt all_solutions
```

- You may want to use a *C++ code profiler* to measure the efficiency of your program and identify the portions of your code that consume most of the running time. A profiler can confirm your suspicions about what is slow, uncover unexpected problems, and focus your optimization efforts on the most inefficient portions of the code.
- We will be testing with both `one_solution` and `all_solutions` and will highlight the most correct and the fastest programs.
- In the regular non-contest HW6, you may submit up to two interesting new test cases for possible inclusion in the contest. Name these tests `smithj_1.txt` and `smithj_2.txt` (where `smithj` is your RCS username). Extra credit will be awarded for interesting test cases that are used in the contest. Caution: Don't make the test cases so difficult that your program cannot solve them in a reasonable amount of time!
- In your `README_contest.txt` file, describe the optimizations you implemented for the contest, describe your new test cases, and summarize the performance of your program on all test cases.
- Extra credit will be awarded based on overall performance in the contest