CS 1501
Algorithm Implementation
Programming Project 1

Online: Wednesday, May 13, 2020
What Is Due: All assignment materials: 1) All source files needed for your program to compile and execute (including files provided to you), 2) The dict8.txt dictionary file and all input data files for the puzzles, 3) Assignment Information Sheet (including compilation and execution information) all **zipped up into a single .zip file which must be submitted via course submission site.** See the submission information document for submission details.
When It Is Due: Single .zip file containing all submission materials by **11:59 PM on Friday, May 29, 2020**.
Late Due Date: 11:59PM on Monday, June 1, 2020.

Note: Do not submit a NetBeans or Eclipse (or any other IDE) project file. If you use one of these IDEs make sure you extract and test your Java files WITHOUT the IDE before submitting. Your TA should be able to compile and run your program from the command line using only your submitted files without having to edit any of your files. If the TA cannot compile / run your submitted code it will be graded as if the program does not work.

Background:
Crossword puzzles are challenging games that test both our vocabularies and our reasoning skills. However, creating a legal crossword puzzle is not a trivial task. This is because the words both across and down must be legal, and the choice of a word in one direction restricts the possibilities of words in the other direction. This restriction progresses recursively, so that some word choices "early" in the board could make it impossible to complete the board successfully. For example, look at the simple crossword puzzle below (note: no letter Xs appear in the actual puzzle shown below – in this example X is always used as a variable):

L	E	N	S
	X_1	X_2	

Figure 1

L	E	N	S
	Q	U	A
D	U		B
C	X_3		

Figure 2

L	E	N	S
	Q	U	A
D	U		B
E	X_3		

Figure 3

L	E	N	S
	T	O	N
A	C		O
T	H	A	W

Figure 4

Assume that LENS has been selected for row 0 of the puzzle, as shown in the Figure 1 above. Now, the word in column 1 must begin with an E, the word in column 2 must being with an N and the word in column 3 must begin with an S. All single characters are valid words in our dictionary so the L in column 0 is a valid word but is also irrelevant to the rest of the puzzle, since its progress is blocked by a filled in square. There are many ways to proceed from this point, and finding a good way is part of the assignment. However, if we are proceeding character by character in a row-wise fashion, we now need a letter X_1 such that EX_1 is a **valid prefix** to a word. Several letters will meet this criterion (EA, EB and EC are all valid prefixes, just to pick the first three letters of the alphabet). Once a possibility is selected, there are now two restrictions on the next character X_2: NX_2 must be a valid word and X_1X_2 must be a valid prefix to a word (see Figure 1). Assume that we choose Q for X_1 (since EQ is a valid prefix). We can then choose U for X_2, (see Figure 2 (NU is a valid word in our dictionary)). Continuing in the same fashion, we can choose the other letters shown in Figure 2 (in our dictionary QUA, DU and DC are all legal words).

Unfortunately, in row 3, column 1 we run into a problem. There is no word in our dictionary EQUX_3 for any letter X_3 (note that since we are at a terminating block, we are no longer just looking for a prefix) so we are stuck. At this point we need to undo some of our previous choices (i.e. backtrack) in order to move forward again toward a solution. If our algorithm were very intelligent, it would know that the problem that we need to fix is the prefix EQU in the second column. However, based on the way we progressed in this example, we would simply go back to the previous square (row 3, column 0), try the next legal letter there, and move forward again. This would again fail at row 3, column 1, as shown in Figure 3. Note that the backtracking could occur many many times for a given board, possibly going all the way back to the first word on more than one occasion. In fact the general run-time complexity for this problem is exponential. However, if the board sizes are not too large we can likely solve the problem (or determine that no solution exists) in a reasonable amount of time. Checking for prefixes as we proceed through the board is definitely pruning our search space, and helps to improve our run-time. One solution (but not the only one) to the puzzle above is shown in Figure 4.

Your assignment is to create all legal crossword puzzles (if any exist) for a prospective board in the following way:

1) Read a dictionary of words in from a file and form a **TrieSTNew** of these words. The TrieSTNew class is a modification of Sedgewick's TrieST class and can be found in file TrieSTNew.java. Carefully read over the code and comments in this class so that you understand how the class works. **In particular, your crossword solution must utilize the searchPrefix() method** in order to prune the solution space. Your program should allow an arbitrary dictionary file to be entered on the **command line** (see Execution Requirement below), but for your testing use the file **dict8.txt** on the CS1501 Assignments page to initialize your TrieSTNew object.

   Note: You don't need to know all of the implementation details of TrieSTNew to use it in this project. You are a client of TrieSTNew and thus need to know the public methods and **what they do**. This is what you need to understand. To help see how the TrieSTNew class works, look over and run handout DictTest.java. It is using text file dictionary.txt but you could also use dict8.txt with it if you edit it. The comments in that file should also be useful to you.

2) Read a **crossword board** in from a file. The name of the file should be specified by the user on the **command line** (see Execution Requirements below). The crossword board will be formatted in the following way:
   a) The first line contains a single integer, N. This represents the number of rows and columns that will be in the board. Since the dictionary will contain up to 8-letter words, your program should handle crosswords up to 8x8 in size.
   b) The next N lines will each have N characters, representing the NxN total locations on the board. Each character will be either
      i) + (plus) which means that any letter can go in this square
      ii) – (minus) which means that the square is solid and no letter can go in here
      iii) A..Z (a letter from A to Z) which means that the specified letter must be in this square (i.e. the square can be used in the puzzle, but only for the letter indicated)

   For the board shown above, the file would be as follows:

```
4
++++
-+++
++-+
++++
```

   Some test boards have been put onto the CS 1501 Assignments page. Check back regularly for additional

files.

3) Find **all legal crossword puzzles** for the given board and print them out to standard output. Many of these test files may have many solutions and some have none. Either way your program may take quite a long time to find (or not find) these solutions. I strongly recommend testing / debugging your program on the smaller size boards before running it on the larger boards. For example, one output to the crossword shown above in Figure 4 would be

```
LENS
-TON
AC-O
THAW
```

Carefully consider the algorithm to fill the words into the board. Make sure it potentially considers all possibilities yet does not waste time rechecking prefixes that have already been checked. Although you are not required to use the exact algorithm described above, your algorithm **must be a recursive backtracking algorithm that uses pruning**. The algorithm you use can vary greatly in its efficiency. If your algorithm is very inefficient or otherwise poorly implemented, you will lose some style points. For guidance on your board-filling algorithm, it is **strongly recommended** that you attend recitation.

Since you must find ALL solutions for each board, you should set up your backtracking algorithm so that it "fails" and backtracks even after finding a solution. To see the idea of this approach, look at the JRQueens.java program discussed in lecture. This program is very different from what you are implementing, but the idea of backtracking even after a solution is found will be similar.

Since some of the test files will have thousands and even millions of solutions, do not print out every solution. Rather, only print solutions whose number == 0 (mod 10000). In other words, you should print out solution 0, solution 10000, solution 20000, etc. **At the end of your execution, <u>print out the total number of solutions found</u>. Don't forget to do this as it will be used in evaluating the correctness of your algorithm.**

Depending upon your algorithm, you may find the solutions in various orders (which may differ from those of your classmates). This is fine as long as all of the legal solutions are found and no illegal solutions are found. See more details on some expected outputs in testFiles.html. Even with a good algorithm, for **some of the input boards your program may take a very long time to run** – longer than may be practical. See notes about these boards in testFiles.html. In particular, **note that for some of the example files** it is not expected that your program will run to completion.

Execution Requirement: When grading your projects your TA may redirect your output to a file so that he/she can refer to it at a later point. To make sure this will work correctly, your program must be a **batch program** -- once your program is invoked there is NO INPUT or anything that would make your program require any user interaction (since the TA will not see any prompts given that they will be sent to a file rather than the display). Any input to your program (ex: name of the dictionary file and name of the board) must be provided via the **command line.** For example, for dictionary dict8.txt and board test4a.txt, an invocation of your program would be:

```
   prompt> java Crossword dict8.txt test4a.txt
```

The command line arguments can be extracted from the parameters to your main() method and utilized in your program. To redirect the output of this execution to a file you could instead type:

```
   prompt> java Crossword dict8.txt test4a.txt > test4a-out.txt
```

This would send all output of your program to file test4a-out.txt where it could be examined later.

Hints / approaches: A challenging part of this assignment is figuring out how to set up the necessary data structures upon which to perform your backtracking algorithm. Let's think of what might be needed in order to do this.

First, you need to represent the puzzle board in some way. This could be done with a simple two-dimensional array of char, but it may be more effective if you create your own object type to represent each square on the board and then make a two-dimensional array of your square type. This would allow you to associate methods with the board squares (in case that would be useful).

Second, you need to store and manage the set of strings / prefixes that are accessible from any location on the board. There are many ways to do this but you must be able to consider sections of the board that you will be testing in your TrieSTNew and you must determine what to test (prefix or word) in each case. For example, **using the example board above**, let's consider some positions on the board and the strings you must test at those positions.

> `Position [2][0] (row 2, column 0).` When testing prefixes / words in your trie, at this position you would consider **only character [2][0]** because it is the first character in the row and immediately above this position is a filled in square. Thus, we are in effect starting a **new vertical word** at this position and a **new horizontal word** at this position. Because there are available squares further down in the both the row and the column, we would be testing both the row and column string as a **prefix** in this case.
>
> `Position [1][3] (row 1, column 3).` Consider the horizontal direction first. We are continuing a word that started in position [1][1] and since we are in the last column we must test to see if this is a valid word. Thus we will test for a word containing characters [1][1], [1][2] and [1][3]. Now consider the vertical direction. We are continuing a word that started in position [0][3]. Since there is another available square in the next row we will be checking this string as a prefix. Thus we will test for a prefix containing characters [0][3] and [1][3].

These two examples demonstrate that for a given board and a **given square on that board**, you must be able to determine which strings you will be testing (where they start and where they end) and whether you will be testing them as prefixes or words. You need to be able to do this in a systematic way for an arbitrary board (which may have filled in squares) and doing this will require some thought.

Important Notes:

- Be sure to thoroughly document your code, especially the code that fills the board. Make sure your name and course info are also in your comments.
- If you want to try for some **extra credit**, develop a second variant of your algorithm that you think may be better or worse than the original. This could include an optimization or perhaps a lack of an optimization (ex: you remove the pruning that you did in your original solution). Make sure this variant is correct (i.e. gets the same number of solutions as your original). Run this variant on the same input files and compare the run-times of the two to see if there is a significant difference. You can receive up to 10 extra credit points on the assignment if you do this well. Note that it will likely require a lot of work to do this extra credit and the 10 points are probably not adequate compensation for the amount of extra work. Nevertheless 10 extra points is the most you can get so consider carefully whether or not to pursue this extra credit.