2015 Fall Computer Science I Program #6: Boggle Wars!!!
Please consult WebCourses for the due date/time

The Problem

Arup recently got internet at home. Much to his surprise, he's gotten into online games, but not the typical kind. Instead of WOW or other games with many players, Arup likes playing Boggle on-line. Although his job deals with computer science, he loves word games! Everything was going great as his online Boggle rating was increasing until he ran into Penelope Kinsley, or PK, as she prefers to be called, an unassuming wordsmith from the UK. PK soundly beat Arup in several rounds of Boggle. At first, Arup's strategy was to simply try to log on at strange times so that PK wouldn't be online. Yet, for whatever reason, PK seems to be very good at reading minds and just happened to be online no matter when Arup logged on. Next, Arup tried to distract PK by talking to her on Boggle chat. Unfortunately, this distracted him more than it distracted her.

Arup's Boggle rating is steadily dropping and he needs your help!!! Since the game is played online, he can theoretically use the aid of a computer program that can search for words in a Boggle grid. Since computers run quickly, if you can write a program that will produce all the words in a Boggle grid, you can help your grade and help Arup defeat PK!!!

How to Play Boggle

The game of Boggle comes with 16 dice, each with six separate letters on each side. The Boggle "board" is a 4 x 4 grid of these dice. At the beginning of the game, the dice are randomly shaken and placed on the board, so theoretically, the dice can land in any order in the 16 squares, and each of the 6 sides of each die are equally likely to be face up. Once the dice all land, the playing board is set with one letter in each entry of the grid. Here is a valid Boggle playing grid:

c	a	s	e
m	o	p	i
s	t	r	e
n	a	p	d

Once the grid is fixed, each player has a set time (around 3 minutes) to come up with as many words that can be formed by connecting letters in the grid. To connect letters in the grid, you must pick a starting letter (for example, the 'a' in the first row), and then pick each subsequent letter adjacent to the previous letter. To be adjacent, a square must be either above, below, to the left, to the right, or to a diagonal to the previous square. No square can be chosen twice. Thus, "mam" is not permissible, because it uses the 'm' in row 2 column 1 twice. But, "aorta" DOES count because we can go from the 'a' in row 1 column 2 to the 'o' in row 2 column 2. Then we can go to the 'r' in row 3 column 3, followed by the 't' in row 3 column 2, finishing with the 'a' in row 4, column 2.

Simply put, a valid configuration in Boggle is a sequence of row, column coordinates such that no two sets of coordinates are equal, and each adjacent set of coordinates differ in row and column by no more than 1. ("aorta" has the sequence (1, 2), (2, 2), (3,3), (3, 2), and (4,2).) If a valid sequence corresponds to an English word, then you can play the word in Boggle.

After the three minutes is up, each player reveals the words they formed. A player receives points only for her unique words, words that no other player listed. In particular, a player receives 1 point for 3 or 4 letter words, 2 points for 5 letter words, 3 points for 6 letter words, 5 points for 7 letter words, and 11 points for words with 8 or more letters for her unique words. (Note: This is irrelevant to solving the problem at hand.)

Problem Solving Restrictions

Dictionary Storage

Your program must check possible words and possible word prefixes against a dictionary. The dictionary must be stored in a trie. Here is the structure that will store one node within the trie:

```
typedef struct trie {
    int isWord;
    struct trie* nextLetter[26];
} trie;
```

To form this dictionary, dynamically allocate the root node in main, that is a single trie. This node will correspond to the empty string. In this node, set isWord to 0 (false), and set all 26 pointers in nextLetter to NULL.

Once this is done, your dictionary must support three functions:

1) Insert a word
2) Check if a word is stored in the dictionary
3) Check if a set of letters is the prefix to any word in the dictionary

Note: It also makes sense to have a function that frees all the memory in the dictionary.

To insert a word, you need to start at the root of the tree and find the pointers that correspond to each letter in the word. For example, if we insert "cat", we first want to find the pointer in index 2 (since 'c' – 'a' is 2) of nextLetter from the root and go to the node it's pointing to. If no node exists, we should create it. From that new node, we want to go to the pointer in index 0 (since 'a' – 'a' is 0) of nextLetter. If no node is there, we should create it again. Finally, from that node, we should take the pointer in index 19 (since 't' – 'a' = 19) of nextLetter. In this last node, we must store isWord = 1, to indicate that "cat" is a valid word.

Basically, since we only create nodes when valid words are inserted, the tree won't be nearly as big as it could be (26^{10}, for example, if we considered all strings of length 10.) If a word doesn't exist, then no path will be created to the corresponding node, OR a 0 will be stored in that node's isWord entry. (If the latter occurs, this means that some other word has these letters as a prefix.)

Note: Only add words with lengths in between 3 and 16, inclusive to the dictionary.

Searching Using Backtracking

To search for all possible words in the boggle grid, we'll start 16 searches, from all the possible starting points in the grid. For each search we'll do as follows:

Recursively try all adjacent squares to the last square in the "prefix" we are trying as the next letter to add on. In this picture, if we are looking for words that start with c:

c	a	s	e
m	o	p	i
s	t	r	e
n	a	p	d

we want to try everything that starts with "ca", followed by everything that starts with "co", followed by everything that starts with "cm". (Note: We can try these options in any order. I just randomly picked clockwise for this example. Recall the DX, DY array from assignment #2. That will come in handy here.) If what we are trying is a word, we print it. Then we continue. In general, we can think of our function as taking in the following information:

1) The prefix we are trying
2) Which board squares have already been used
3) Our current position on the board

Given these pieces of information (and the board and dictionary), the void recursive function we create will print the following: all the words that start with that prefix (with the last square as identified by the current position) that are valid words in the game.

For example, calling the function with the prefix "cor", designating that (1,1), (2,2) and (3,3) have been used and that (3,3) is our current position, the function should print out the following words (using the dictionary posted online with 150315 words):

```
cor, corps, corpse, core, cored, cord.
```

Thus, what needs to be done is print "cor", followed by recursively solving the problem for "corp", "cori", "core", "cord", "corp", "cora" and "cort". Note that we skipped "corr" and "coro" because both of the squares with the 'r' and 'o' had already been used.

If a function call is made on a prefix that doesn't exist (in this grid, an example of such a prefix is "rp", since no words start with "rp"), then we can immediately return and stop the function.

Note: Since we are trying all possible board configurations, it's possible that the same word will print out twice because it appears in two different ways in the same board. In the sample posted online, notice that "cake" and "face" both appear twice in the second game. Viewing the game board verifies that there are TWO different ways to form both words. (This is because there are two different e's adjacent to both 'k' and 'c' on the board.) There is no need to avoid these printing. If you would like store previously printed words globally and only print one copy of each word, feel free to do so.

Input File Specification (dictionary.txt)
The first line of this input file will have a single positive integer, *n*, representing the number of words in the dictionary. The following *n* lines will contain one word each, all in lowercase letters, in alphabetical order.

Input Specification (standard input)
The input has a single positive integer, *n*, on its first line, specifying the number of games of Boggle to play. Each of the games will follow.

For each game, there will be four strings of four lowercase letters each on four lines. Each game will be separated with a blank line for clarity's sake. (This shouldn't change how you read in the game boards using scanf.)

Output Specification
For each input case, print out a header with the following format:

```
Words for Game #g:
```

where g represents the number of the game, starting with 1.

Then, starting on the following line, list each possible word that can be formed (according to the dictionary), with one word per line. The words can be listed in any order.

Separate the output for each case with TWO blank lines.

Note about output: The output for each test case is not unique, since you might print out all the correct words in a different order. We will use a checking program to determine the correctness of your output. The checker will judge your output as correct as long as each unique word that exists in the grid is outputted at least once and no invalid string (one that isn't in the dictionary or isn't in the grid) is outputted at all. Any valid string may be outputted more than once and this will not affect the correctness of your program.

Sample Input and Output Files
These will be posted online with this assignment write up.

Deliverables
Turn in a single file, *boggle.c*, over WebCourses that solves the specified problem. Make sure to include ample comments and use good programming style, on top of the requirements that are given in the program description above.