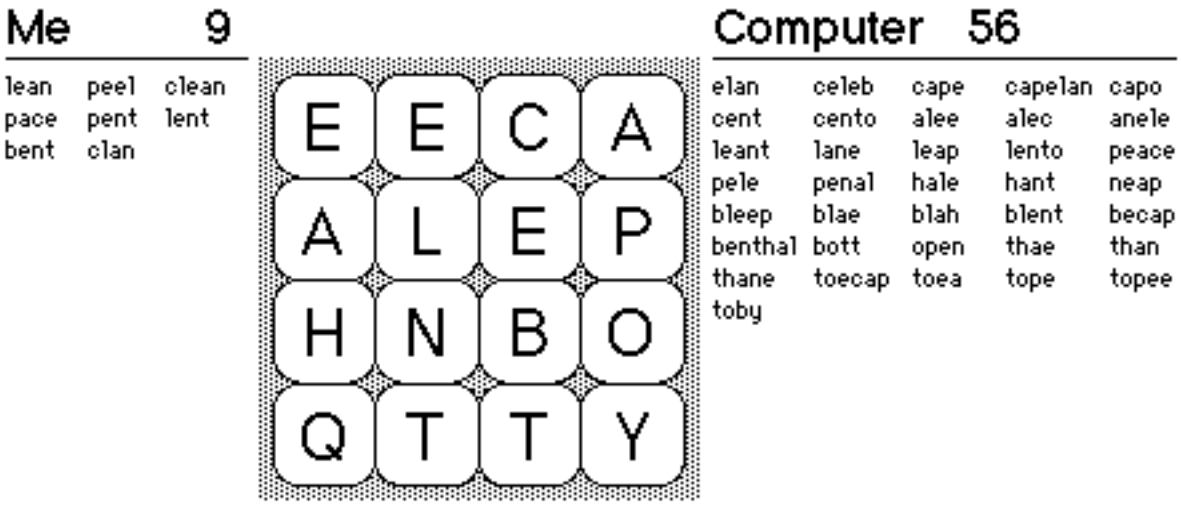
CS61BL Summer 2012 Project 2: Boggle!



I. Introduction and Rules

Boggle is a multiplayer word-finding game designed by Alan Turoff and trademarked by Parker Brothers. You can find out all the details here on the Wikipedia page:

http://en.wikipedia.org/wiki/Boggle

The rules portion of the above article are copied below:

The game begins by shaking a covered tray of sixteen cubic dice, each with a different letter printed on each of its sides. The dice settle into a 4x4 tray so that only the top letter of each cube is visible. After the dice have settled into the grid, a three-minute timer is started and all players simultaneously begin the main phase of play.

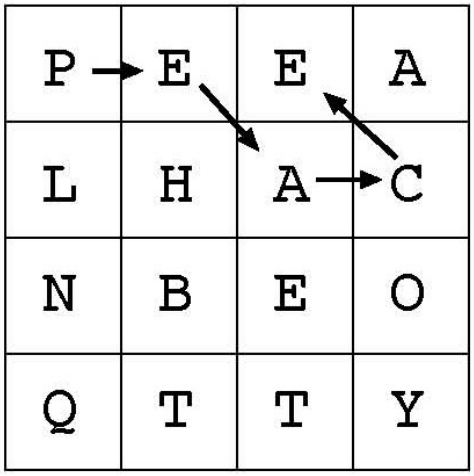
Each player searches for words that can be constructed from the letters of sequentially adjacent cubes, where "adjacent" cubes are those horizontally, vertically or diagonally neighboring. Words must be at least three letters long, may include singular and plural (or other derived forms) separately, but may not use the same letter cube more than once per word

For example, consider this board:

.

P	E	E	A
L	H	A	C
N	B	E	O
Q	T	T	Y

Here, we can make the word “Peace” as follows



Note that the word "PLACE" is **not** legal, since after the "L" you would need to jump over the "H" to get to the "A". The same die cannot be reused in the same word (but can be reused again in a different word).

Similarly, it is not possible to make the word **POPE**, because doing so would require reusing the **P**.

Words are scored by the following formula:

Word Length	Points
3	1
4	1
5	2
6	3
7	5
8+	11

Each player records all the words he or she finds by writing on a private sheet of paper. After three minutes have elapsed, all players must immediately stop writing and the game enters the scoring phase.

In the scoring phase, each player reads off his or her list of discovered words. If two or more players wrote the same word, it is removed from all players' lists. Any player may challenge the validity of a word, in which case a previously nominated dictionary is used to verify or refute it.

For all words remaining after duplicates have been eliminated, points are awarded based on the length of the word. The winner is the player whose point total is highest, with any ties typically broken by count of long words.

One cube is printed with *Qu*. This is because *Q* is nearly always followed by *U* in English words (see exceptions), and if there were a *Q* in Boggle, it would be unusable if a *U* did not, by chance, appear next to it. For the purposes of scoring *Qu* counts as two letters: *squid* would score two points (for a five-letter word) despite being formed from a chain of only four cubes.

II. Assignment

Your assignment is to write a program that plays a fun, graphical rendition of Boggle, adapted to allow the human and machine to play pitted against one another. By the end, you will have a program that will perform like the picture at the top of the first page, consistently destroying any and every human opponent.

Start by downloading the source code package here:

http://inst.eecs.berkeley.edu/~cs61bl/su12/code/Boggle.zip

Unzip the package and import it into a new Eclipse project. You should see a bunch of classes. The important one for now is **Boggle.java**.

Part 1 - User Word Validation (20%):

Try firing up the GUI by clicking "Run" and selecting "Java Application" and then "Boggle". You should see a randomized Boggle board appear. Find a word as you would if you were actually playing and enter it. Nothing should happen. This is because your program has no way yet of making sure that a user-entered word actually exists on the Boggle board. Let's change that.

You must fill in the body of the cellsForWord in the WordInBoardFinder class. Given a word string, this method will search for it on the board by starting from each cell one by one. How you search from a particular cell on the Boggle board is up to you.. There will be many things to consider as you are searching the board from a particular cell, here are a few examples:

- If you go out of bounds, then you probably shouldn't continue the search from that point.
- Remember that you cannot re-use board cells.
- cellsForWord will return a List of BoardCell objects that should represent the cells of the found word (if it exists on the board).
- **Be careful when finding a "Q" in a word since if there's a match the Boggle board cube has two characters and you'll have to adjust parameters in the recursive call to make sure you do the right thing.**

We have provided a JUnit test class with a few tests to sanity check your implementation of cellsForWord. **THIS IS NOT A COMPLETE TEST SUITE**. There will be cases we are omitting from the tests that we WILL use when we grade your project. It will be up to you to make sure your method works 100% correctly.

Part 2 – Lexicons (35%):

Now that your program knows that a user-inputted word is actually on the board, it needs to make sure that word is actually a legal Boggle word. We will be using **bogwords.txt** as our set of legal Boggle words for this project.

Being CS61BL students, we shouldn't be satisfied with simply searching naively through the dictionary text file for a user-inputted word. If the word were made up, we'd scan every single word in the dictionary before realizing it isn't there. This is far too inefficient. For the second part of this project you will implement (at least) 2 Lexicons that store the words in the given Boggle dictionary in a way that allows for "smart" searching of a word. This Lexicon ADT is given in LexiconInterface.java. Your two lexicons should implement this interface.

- **Simple sorted ArrayList**: The contains() method for this Lexicon will use a modified binary search to look for a word. This binary search will need to be modified because the string you are looking up may not always be an actual word, it might be a prefix! "out" is one such string that is both a word and a prefix (OUTfield, OUTland, OUTside). This will be more clear in Part 4.

- **Trie data structure, or a prefix-tree**: We will cover this data structure more formally in class, but a great deal of information on Tries can be found online. We have included the Wikipedia link below, but it would be in your best interest to do a little more research on your own.
    - http://en.wikipedia.org/wiki/Trie

    - The contains() for this Lexicon can make use of word prefixes to "fail fast"; if a word is made up, contains() should return false the moment it realizes this without necessarily completing the search. For example, *aloze* is not a word in the Boggle dictionary, but *aloe* is. Both words start with the prefix *alo*, but no word starts with *aloz.* Searching for *aloze* in your Lexicon should stop when that prefix *aloz* is reached in the search. A trie is well-suited for prefix searching.

For both of these data structure implementations of the Lexicon ADT, you may include any other methods and instance variables you want. The design of the data structure is left up to you, so long as you follow the basic guidelines we've set and adhere to the Lexicon interface provided.

Your data structures should show good encapsulation practices: methods and instance variables should not be public unless absolutely necessary.

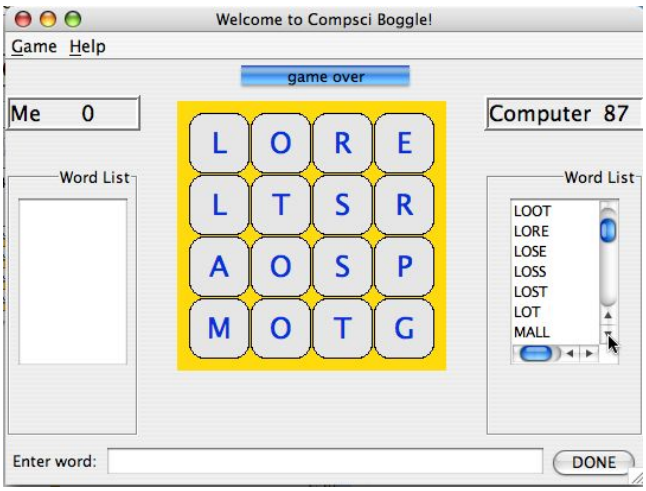
Part 3 – Autoplayers (30%):

Congratulations on making it through to part 3! Now you have a basic working Boggle program. You can recognize strings on a Boggle board and check to see if a particular string is actually a legal Boggle word. Let's use these to create a super Boggle computer that will impress and frustrate your friends, forcing many ragequits.

Using your new Lexicons, implement two AutoPlayers, BoardFirstAutoPlayer and LexiconFirstAutoPlayer, that find all the words on a particular Boggle board. Each of these two AutoPlayers will use a different technique for playing. You will analyze these techniques in Part 4.

- **LexiconFirstAutoPlayer:** Once you've implemented WordOnBoardFinder to find where a word occurs on a board you'll be able to implement this class in a straightforward way. To find all the words on a board, iterate over every word in a lexicon, checking to see if each is in the current Boggle board. This AutoPlayer should use things you've already implemented in Parts 1 and 2.

- **BoardFirstAutoPlayer**: Rather than iterating over every word in the dictionary, you can use the board to generate potential words. For example, in the board shown in the screen shot below on the right the following words can be formed starting the *"L"* in the upper-left corner: "LORE", "LOSE", "LOST", "LOT". Starting at the cell "R" at [1,3] (recall the first row has index zero) we can form "REST" and "RESORT". Since no word begins with "GT", "GP", "GS", no search will proceed from the "G" in the lower-right after looking at more than two cubes since these two-character prefixes aren't found in the given Lexicon.



Part 4 – Writeup (15%):

Your final task is to document your project in a README file. This README must contain the following:

1. A description of your two Lexicon classes and how they are implemented. Describe how your binary search for the ArrayList implementation works to return the information we specified. Also describe your Trie data structure and what helper methods, instance variables, and other classes you used.

2. Perform some empirical and statistical analysis on your two Lexicon classes and your two Autoplayer classes (i.e. timing experiments). Elaborate on the differences between implementations and how those differences might lead to the different data you collect for each.

1-2 pages for the README will suffice. The more detailed you get, the better it will look and the more you will learn. Charts and graphs are always welcome.

III. Administrivia

1. **Submission:** This project is due **July 21st** at 10 PM. You will put WordOnBoardFinder, your two Lexicon implementations, and your two AutoPlayer implementations in a directory along with the README for Part 4 and submit it as "proj2" using the usual submit command.

2. **Open-endedness:**

We the CS61BL staff are well aware of how open-ended this project is. This is meant to both challenge you and give you the chance to show us (and yourselves!) your mettle as computer science students. This project has been implemented so that by the end, you will have a product that you built almost entirely yourself and can show off to your friends, your classmates, and even future employers. We will do our best to provide on-going support and advice in the coming weeks for this project. Start early. There will be several helper documents released soon; TAs will also be providing some ideas in lab sections for particular implementation roadblocks. This project is not meant to frustrate. We hope you find this as rewarding to do as it was for us to implement.

3. **Cheating:**

This project has been implemented in several universities over the course of many years. Thus, it is inevitable that there will be solutions floating around. Implementations of the various data structures you have been asked to implement are also prevalent on the web. As you do research, you might stumble across a trie implementation in Java or a fully functional Boggle project. I will say this simply: If you cheat, we will catch you. We will be actively looking for cheaters in the grading of this project. If you choose to cheat by copying code that others have written, you

will receive an automatic F in this course. No exceptions. Do not cheat. I won't be giving you "tips" on how not to cheat. I think this is an absurd idea; you are all **adults** attending the #1 public university in the world. You would not be here if you weren't trying to better yourselves as students. Do not rob yourself of a learning opportunity.

This project was adapted from projects at Duke, Stanford, and UCSD. Thanks especially to Julie Zelenski for a great writeup of the project and the source code framework. Big thanks also to Hoa Long Tam and Armin Samii for adapting the project for UC Berkeley CS61BL.