Boggle

Due Date: **Monday 3/27/2017**

# Introduction

Boggle is a well-known family word game that requires players to find as many words as possible in 4 x 4 grid of letters in a limited amount of time. You are going to write a program that plays boggle. It must generate the letters that occupy a 4 x 4 grid, and then ALL the words in the board using an edited version of the *Official Scrabble Players Dictionary.*

The goal of this assignment is to demonstrate your mastery of recursion and backtracking. You must develop and implement your base cases (there are many), maintain state (breadcrumbs) to avoid infinite recursion and stack overflows, and to facilitate backtracking without walking off the edge of the board.

# Finding Words

|  |  |  |  |
| --- | --- | --- | --- |
| T | E | A | M |
| S | Y | F | S |
| P | A | T | E |
| G | N | I | D |

### Rules

Words are formed by combining adjacent letters. Adjacent letters are defined as all letters immediately surrounding a given letter (including diagonals). For example, in the diagram to the right, the letters adjacent to the letter Y are highlighted in green. Letters can be used multiple times but only once within a given word. For instance, the letter S in blue can be used to form the words STAG and SAFETY, but cannot be used to form the word SETS which would require the letter S to be used twice within a single word.

If a word appears in the table multiple times, in multiple different ways, it is only counted once. For, example, the word SAY appears twice in the above grid starting with two different letters S, going through two different letters A, and ending on the same letter Y. The word SAY must only appear in the list of found words one time.

### Recursive Strategies

Finding words in a Boggle board is very much like a flood fill. You start at a specific point and spread out from there. As you go further you build strings of letters and look up the strings in a dictionary to check their validity as a “word”. If a new word is found, then it is recorded. Eventually, the edge of the board is reached, or all the adjacent letters have already been visited and the program must backtrack to find other paths.

To make this work you must drop *breadcrumbs* to mark your path. This prevents you from going over the same path twice and helps you find your way home. Your job is to find a mechanism to place those breadcrumbs and incorporate them into your recursive base cases.

Once you exhaust all the paths from a single point (e.g starting a position 0, 0), then you do it again at positions 0,1 and 0,2 etc. until you have exhausted all the starting points on the board.

# Requirements

Create a project called Boggle. Within that project you will have four classes. Three of the classes, *Program*, *Board*, and *WordComparator*, will be written by you. The fourth class, *WordList*, is provided to you and should be added to your project without modification (unless you are doing the last extra credit problem.) :- )

## Program

You must have a class called Program. The class must contain your main method. From the main method you must do the following.

1. Announce the program “Welcome to Boggle, by <YourName>.”
2. Instantiate a *WordList* object using the provided WordList.java file (see below).
3. Instantiate a *Board* object **of size 4**, using the *Board* class that you write (see below).
4. Display the *Board* using the toString() method of the *Board* class (see below).
5. Call the *find*()method of the *Board* object to get a list of the words found in the board.
6. Display the results (see below).

## WordList

The *WordList* class is provided to you, and can be found as an attachment in the dropbox folder for this assignment. It will read a file called “WordList.txt” which is also attached to the dropbox folder, and must be placed in the top level directory of your project (the Boggle Director). The *WordList* class extends ArrayList<String> so that the ArrayList<String> is always sorted, thus allowing for O(log N) lookup of words in the list. With so many possible words in each board, it makes a real and visible difference to the running time! You will likely want to use three different methods from this class.

1. The *WordList* constructor. Invoke the constructor to load all the words between 3 and 8 characters in length from the word list file – as follows:

WordList wordlist = new WordList("WordList.txt", 3, 8).

1. The boolean contains(String s) method, to determine if a candidate word exists in the word list.
2. int getLongestWordLength() to implement a base case in your Board*.*find() method so that you can look for words only up to the depth of the longest word in the word list.

Because the *WordList* class extends ArrayList<String>, you will also be able to use any of the standard ArrayList<String> methods once the *WordList* object is properly initialized.

## Board

The *Board* class represents the board containing the letters which you will instantiate, populate, and then recursively search. The class must meet the following requirements:

1. You must implement a constructor that takes the word list and the size of the Boggle board (which must be square) as parameters.
2. The constructor must instantiate the board **as a square two dimensional array of strings**.
3. The constructor must also populate the array with random letters from random words in the list. The reason for this has to do with the actual frequency of occurrence of letters in real words in the language. Choosing random letters from A-Z with equivalent frequency can result in boards with very few usable words.
4. **Your implementation must work for boards of any size.** Do not assume that just because the program class instantiates a board of size 4, that you can use the number 4 directly within your Board class. Make sure that all of your *Board* code looks at, and uses, the length of the array created in the constructor.
5. You must implement a method ArrayList<String> find() that takes no parameters.
6. When invoked, the method will return all the valid words found in the board, as listed in the word list.
7. The implementation of the find() **must be recursive.** Like many other recursive algorithms, you will likely need another internal method that accepts additional parameters to implement the recursion successfully, as discussed in class.
8. The implementation of the find() method must search no deeper than the longest word in the word list. That is, don’t search for 12 letter words if you find that the longest word may only be 10 characters long.
9. Before returning to the caller, the list must be sorted (see Comparator below).
10. You must implement a toString() method to allow display of the board from the main method in your Program class. The returned string must include proper spacing and new line characters to display the letters in a square grid. For instance:

T E A M

S Y F S

P A T E

G N I D

The board will always be square. But do not forget that the board can be any size.

1. **Your class must use only two instance variables** – one to hold the valid-word list (which you will need in your find() method), and one to hold the array of strings representing the board itself. All other state for searching the board must be maintained as local variables, passed as parameters and return values between methods.

## WordComparator

Once you have a list of words from the Boggle board, your main method must display the words so they are ordered by length, and then alphabetically within length. You must write a comparator that you can pass to Collections.sort() to sort the ArrayList returned by the Board.find() method.

The name of the comparator class must be *WordComparator*.

## Results

The results must be displayed in a format similar to that shown below. That is you must display the number of words found, and then a separate list of words for each length, and alphabetical within length. If there are no words of a given length then do not display that sublist.

Found 293 word(s)

8 letter words

FEASTING

YEASTING

7 letter words

DESTAIN

EASTING

ESTAFET

FAINTED

FASTING

FEASTED

MASTING

PAINTED

PANTIES

SAINTED

YEASTED

6 letter words

DEFATS

DESTIN

DETAIN

FAINTS

FAINTY

FASTED

FATING

FEDITY

FEINTS

FETING

GAIETY

GAITED

GAYEST

INGATE

MASTED

MAYEST

PAINTS

PAINTY

PANTED

PANTIE

SAFEST

SAFETY

SAINTS

SANIES

SANITY

SATING

…

And so forth all the way down to three letter words.

# Grading

This assignment will be counted as a homework – 40 points in total. A late assignment counts as zero.

## Rubric

The main goal of this assignment is to learn about recursion and backtracking. As such, the bulk of the points are allocated to the find() method of the *Board* class. Make sure you focus your efforts on that part of the assignment.

|  |  |
| --- | --- |
| Functional Correctness (Behavior) | |
| Program (1 pt for items 1-5, 3 pts for item 6) |  |
| 1 pt each for items 1 through 5 | 5 pts |
| 3 pts for properly formatted results | 3 pts |
| Board |  |
| Constructor signature as specified in requirements | 1 pts |
| Instantiation of array for the board | 2 pts |
| Populating board | 3 pts |
| Method signature for find() | 1 pts |
| Internal method to support recursion | 3 pts |
| Additional state for marking *breadcrumbs* and backtracking | 3 pts |
| All state during find() passed as parameters and return values | 3 pts |
| Search pattern for finding adjacent letters | 3 pts |
| Proper base cases for recursion | 4 pts |
| Backtracking cleans up *breadcrumbs* | 2 pts |
| Lookup and recording of found words | 2 pts |
| Properly detects edges of the board | 2 pts |
| Find results are sorted as specified by requirements | 1 pts |
| WordComparator | 2 pts |
| Total | **40 points** |

## Deductions

There are two requirements that are absolutely mandatory. Failure to follow these two requirements will result in a 25% deduction each (50% total) in the final grade. The two requirements are:

1. Use only the two instance variables in your implementation of the *Board* class. One is for a reference to the word list. The second is for a reference to the board itself – the array of letters. Do not attempt to circumvent this rule by using another class to contain multiple objects.
2. The implementation of *Board* must support boards of arbitrary sizes. Do not hard code the size of the board anywhere in the Board class. Allocate an array of the correct size in the constructor, and then query the length of the array in your code where you need it.

## Extra Credit (3 pts Maximum)

Typically, we only grant 5% extra credit. This time we are offering 50% more for a full 7.5%!

### Suffixes (1 pts)

When generating the board, allow for a single square in the board to contain one of the suffixes, "-ED", "-EST", and "-ING". When looking for words, these “letters” can only be used on the *end* of the word, and cannot be split internally. In other words, all letters in the suffix must be used if they are used at all. This requires an additional base case that you will need to implement. Be sure to remove the hyphen when looking up and displaying the word. Make sure that you adjust the toString() method to retain the correct formatting of the board.

### Prefixes (1 pts)

When generating the board, allow for a single square in the board to contain one of the prefixes "RE-", "STR-" and "UN-". When looking for words these “letters” can only be used to *start* a word, and cannot be split internally. In other words, all letters in the prefix must be used if they are used at all. This requires an additional base case that you will need to implement. Be sure to remove the hyphen when looking up and displaying the word. Make sure that you adjust the toString() method to retain the correct formatting of the board.

### Letter Frequency (2 pts)

Modify the *WordList* class to count each letter as words are read in. Using those counts, create and populate an array that contains the ratio each letter contributes to the sum of all letters. Then use that array to write a method with the signature String nextLetter() which returns a random letter as a function of that letter frequency distribution.

For instance, if you have an alphabet composed of the four letters A through D and there are 1, 2, 3, and 4 of those letters respectively, then there are a total of 10 letters. The letter A comprises 0.1 of the overall sample. The letter B comprises 0.2; the letter C comprises 0.3, and the letter D comprises 0.4. You can then use those numbers (which sum up to 1.0) combined with Math.random() to generate and return a random letter.