Program will create a BoggleGrid containing a scrambled set of letters. The human player will play first. The player will proceed to enter, one at a time, each word they find on the grid. If a word meets the minimum length re- quirement (which is 4), has not been guessed before, and can be legally formed given the arrangement of letters, it will be evaluated. This means it will be checked against the list of legal words in the Dictionary. If the word exists, it will be added to the player’s word list, and the player will be awarded points according to the word’s length. If the word doesn’t appear in the Dictionary,

1

the player won’t get any points.

The player indicates that they are through entering words by hitting return. At this point, the computer will play. The computer will search through the board looking for words the player didn’t find and award itself points for find- ing all those words. If the player wants to play again, the entire process will be repeated.

The dice

The letters in Boggle are not chosen entirely at random. Instead, the letters on the faces of the cubes are arranged in such a way that common letters come up more often and it is easy to get a good mix of vowels and consonants. To recre- ate this, we have provided you with representations of dice used in Boggle. We can use these to generate a good mix of BoggleGrids. Each die is represented as a single line of 6 letters without any spaces. For example, a set of 16 dice looks like this:

String[] dice small grid = {”AAEEGN”, ”ABBJOO”, ”ACHOPS”, ”AFFKPS”, ”AOOTTW”, ”CIMOTU”, ”DEILRX”, ”DELRVY”,”DISTTY”, ”EEGHNW”, ”EEINSU”, ”EHRTVW”, ”EIOSST”, ”ELRTTY”, ”HIMNQU”, ”HLNNRZ”};

During initialization, you will ”shake” the dice to randomly set-up the Bog- gle grid. There are two steps to the randomization process. First, the cubes themselves need to be shuffled so that the same die is not always tied to the same grid cell of the board. Second, a random side from each die needs to be chosen to be the face-up letter at that grid cell.

Shuffling can be a little involved. To re-arrange the letters, you can use a simple shuffling algorithm given by this pseudo-code (which is NOT Java!) to mix up the elements in a two dimensional array:

for (row = 0; row ≤ nRows; row++) for (col = 0; col ≤ nCols; col++)

swap die in array[row][col] and array[rand(row, nRows-1)][rand(col, nCols-1)]

Basically, this walks through the arrays, selecting a random element to place in each slot. You should be able to do something like this to shuffle the dice into many different board combinations.

Alternatively, the user may decide to enter in their own board configuration. If the user wishes to enter in a custom board, they will be prompted for a string of characters equal in length to the number of grid positions on the board. These will represent the letters to place in the grid from left to right, top to bottom (as you would read a book).

2

Once initialization is complete, you’re ready to implement two types of recursive search; one for the user and one for the computer. For the user, you will search to validate specific words that they enter and stop as soon as you find them. For the computer, you will search for all words that are possible and that have not yet been discovered by the user. While you may be tempted to try and integrate these two searches so they work as a single recursion, this is not a good idea for two reasons. One is that it will make your program messy. The other is that we want you to get practice with different recursive structures. So, we are explicitly requiring that you implement two separate recursive functions, one for the human player’s turn (searching for a specific word) and for the computer’s turn (exhaustive search).

The human player’s turn

After the board is displayed, the player gets a chance to enter all the words they can find on the board. Your program will read in a list of words until the user signals the end of the list by typing a blank line. After the user enters each word, your program must check the following conditions:

1. that the word is at least four letters long
2. that it is defined in the Dictionary as a legal word
3. that it occurs legally on the board (i.e., it is composed of adjoining letters such that no board position is used more than once)
4. that it has not already been included in the user’s list of found words

If any of these conditions fail, you should tell the user and not award any score for the word. If, however, the word satisfies all these conditions, you should add it to the user’s word list and score.

A word’s point value is determined by its length: 1 point for the word itself and 1 additional point for every letter over the minimum. Since the minimum word length is 4, the word ”boot” receives 1 point and ”trees” receives 2. The player enters the return key when done entering words.

The computer’s turn

On the computer’s turn, your job is to find all of the words that the human player missed by recursively searching the board for words beginning at each square on the board. In this phase, the same conditions apply as on the user’s turn, plus the additional restriction that the computer is not allowed to count any of the words already found.

As with any exponential search algorithm, it is important to limit the search as much as you can to ensure that the process can be completed in a reasonable time. One strategy that can help is to recognize when you are going down a dead

3

end so you can abandon it. For example, if you have built a path starting with ”zx”, you can use the Dictionary’s isPrefix function (which you must write) to determine that there are no words in English beginning with those letters. Thus, you can stop right there and move on to more promising combinations. If you don’t do that, you may find that some of these routines take more time than you like.

The Dictionary class

The functions in the Dictionary class allow you to verify the user’s and com- puter’s words by looking them in up in the word list create a new word list and add words to your Dictionary, either one at a time or as an entire collection stored in a data file. You will also use this class to determine whether a string matches a word in the Dictionary or whether it is the beginning (i.e. the prefix) of some longer word in the dictionary.

4

Complete the following methods in the Dictionary class:

1. containsWord, which checks to see if a given word exists in the dictionary.
2. isPrefix, which checks to see if a given string is a prefix of any word in the dictionary.

In the BoggleGrid class you will complete:

1. The BoggleGrid constructor, which will initialize the size of the boggle grid and allocate a two dimensional array of chars to hold the letters at each grid position.
2. initalizeBoard, which will accept a string as a parameter and assigns a letter in the string to each grid position.

In the BoggleStats class you will complete:

1. The BoggleStats constructor, which will set the current round, player score totals and player score averages to 0. It will also initialize found word lists for the players.
2. addWord, which will add a word to a given player’s word list and update the player’s score totals.
3. endRound, which will reset the human and computer word lists, increment the round, and update player averages.
4. summarizeRound, which will print nicely formatted summary or round statistics for the user.
5. summarizeGame, which will print nicely formatted summary or the entire game’s statistics (i.e. for 1+ rounds of play).

In the BoggleGame class you will complete:

1. The method randomizeLetters, which will assign letters to each grid posi-

tion.

1. findAllWords, which should be a recursive function that finds all valid words on the boggle board. Valid words are at least 4 letters in length and consist of letters that are adjacent to one another on the boggle grid. Note also that a letter at a given grid position cannot be repeated in a word (i.e. loops are not allowed).
2. humanMove, which will get words from the user, check them for validity, and update the game statistics as appropriate.
3. computerMove, which will get words from the computer. The computer can find words that are both valid and not in the player’s word list.