In this assignment, you will work in pairs to implement the game of Boggle^{TM}, which will require you to (1) design and implement a number of recursive algorithms and (2) think about various implementation strategies for the Boggle dictionary. In addition, you will conduct Peer Review of other teams' assignments; in particular, you will try to break their code, and you will write a report about your testing efforts. There are three deadlines for this assignment, and the first two are *strict*, so we encourage you to start early.

Program and Report Due	5:00pm Wednesday October 30
Peer Reviews Due	5:00pm Saturday November 2
Testing Report and Test Code Due	5:00pm Tuesday November 5

1 The Game of Boggle

Boggle is a word game played with sixteen cubes, where each side of each cube has one letter of the alphabet. The cubes are randomly arranged on a 4 × 4 grid, with one legal configuration shown below:

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

The object of the game is to identify—in this grid of letters—words that satisfy the following conditions:

- The word must be at least four letters long.

- The path formed by the sequence of letters in the word must be connected horizontally, vertically, or diagonally.

- For a given word, each cube may only be used once.

For example, the above board contains the the word PEACE, which is legally connected as shown below.

E	E◄C◄A

E	E◄C◄A
A	L	E◄P
H	N	B	O
Q	T	T	Y

The board does not contain the word PLACE, because the L and the P are not connected, and the board does not contain the word POPE, because the same P cannot be used more than once for a given word.

Points are scored based on the length of the word. Four-letter words are worth 1 point, five-letter words are worth 2 points, etc.

2 Your Boggle Game

You will create a Boggle game which randomly sets up a board. Your game will then allow a human to identify a list of words in the board, which your program will verify against some dictionary and for which your program will compute a score. When the human can think of no more words, your program will create a list of legal words that the human did not identify.

For example, if the human identified the following words for the above board:

lean	pace	bent	peel	pent	clan	clean	lent

Your program would verify that each of these words was legal and would assign a score of 9 for the seven four-letter words and one five-letter word. Your program would then use a dictionary, whose words we will provide, to identify the following words (yeah, we didn't know that hant and blent were words, either; neither does the Unix spellchecker):

elan	celeb	cape	capelan	capo	cent	cento	alee
alec	anele	leant	lane	leap	lento	peace	pele
penal	hale	hant	neap	blae	blah	blent	becap
benthal	bott	open	thae	than	thane	toecap	tope
topee	toby						

For this wondrous list of words, your program would obtain a score of 56, thoroughly embarrassing the feeble human player.

2.1 Initializing the Game

The game uses sixteen cubes, each with a particular set of letters on it. These letters have been chosen so that common letters are more likely to appear, and so that there is a good mix of consonants and vowels. You should initialize your cubes from the file, `cubes.txt`, which contains the following data:

```
LRYTTE
VTHRWE
EGHWNE
SEOTIS
ANAEEG
IDSYTT
OATTOW
MTOICU
AFPKFS
XLDERI
HCPOAS
ENSIEU
YLDEVR
ZNRNHL
NMIQHU
OBBAOJ
```

Each line represents the six characters that will appear on the faces of a single cube. To initialize your game, you should read this file and store the data in a data structure that represents the 16 cubes.

For each game, your program should randomly shuffle each cube and randomly distribute the cubes amongst the 4 × 4 grid. There are many ways to do this. You could lay down the cubes and start swapping them around, or you could pick cubes randomly and lay them down one at a time. Use any method that produces a good random permutation.

Your program will need to implement a simple dictionary that can read a large number of words (hundreds of thousands of words) and store it in some judicious manner. The dictionary file is called `words.txt`, and it contains one word per line, with the words in ascending lexicographic order.

Finally, you will need to create some way of displaying the state of the board, the words guessed by the players, and the players' scores. For those of you unfamiliar with GUI's, this can be a text-based interface if you wish.

Once your initialization is complete, you're ready to implement two types of recursive search, one for the human player and one for the computer. Each search uses a distinct type of recursion. For the human, you search for a specific word and stop as soon as it's found in the dictionary, while for the computer, you are searching for all possible words. You might be tempted to integrate the two types of recursion into a single routine, but this will be unnecessarily complex, so we advise you to resist this temptation. However, the two routines may well share helper methods.

2.2 User Interface (UI)

To actually play Boggle, you will need a user interface. You may implement the user interface however you like, as long as it meets the conditions outlined in this section. If you want, you can create a graphical user interface, but a text-based interface is equally acceptable (and probably easier to write).

When the game starts, the UI should create and display a new scrambled Boggle board. It should then accept a word from the human as input, check the word for validity, and then compute a score for the word. If the word is valid, the program should visually indicate where on the board the word was found. A graphical UI might change the colors of the letters, while a text UI might change upper/lower case. Printing a list of coordinates is not very helpful.

If the word is too short, not on the board, not a legal word, or is already used by the player, your program should emit a suitable error message and prompt for another word. The player should not get credit for bad words.

After the human indicates that she is done, the computer player gets to select all valid words that were not identified by the human. The program should then display the respective scores. After every game, the UI should prompt the user for another game and act accordingly.

Implement the game in `Boggle.java`. We should be able to run the game with `java assignment.Boggle` (with a suitible class path).

2.3 The BoggleGame Interface

Everything that manages the mechanics of the game should be contained in a class that implements the BoggleGame interface. This interface provides all the necessary functions for implementing a basic generalized game of Boggle. The use of this interface completely separates the code that manages the game from the code that implements the UI.

```
public interface BoggleGame {
    public static final int SEARCH_BOARD = 0;
    public static final int SEARCH_DICT = 1;
    public static final int SEARCH_DEFAULT = SEARCH_BOARD;
    void       newGame(int size, int numPlayers,
                    String cubeFile, BoggleDictionary dict);
    char [][]  getBoard();
    int        addWord(String word, int player);
    List<Point> getLastAddedWord();
    void       setGame(char [][] board);
    Collection<String> getAllWords();
    void       setSearchTactic(int tactic);
    void       playerDone(int player);
    int []     getScores();
}
```

Implement your game engine in `GameManager.java`. For additional details, see BoggleGame.java.

2.4 Dictionary Interface

To check for legal words, your program will need to search the dictionary, which you will implement. The methods in the Dictionary interface, shown below, allow you to create a new dictionary and to insert words into it as an entire collection stored in a single file. The interface also specifies methods to determine whether a string is found in the dictionary or whether it is a prefix of a word in the dictionary. Finally, the interface extends `Iterable<String>` so you will need to implement the `Iterator<String> iterator()` method on your dictionary class. You will need to implement a class that implements `Iterator<String>` to return from `iterator()`. (See `http://docs.oracle.com/javase/7/dc`

and `http://docs.oracle.com/javase/7/docs/api/java/util/Iterator.html` for more information about these interfaces.)

```
public interface BoggleDictionary extends Iterable<String> {
    void loadDictionary(String filename);
    boolean isPrefix(String prefix);
    boolean contains(String word);
}
```

There are interesting design issues associated with the dictionary. You should strive to create the simplest possible dictionary that is reasonably efficient. Your lookup operations should take $O(\log n)$ time or better. Be sure to justify your design decisions.

Because of the efficiency requirements, your dictionary should not just be an adaptor for some existing Java Standard Library class. It should instead be a new data structure. Your dictionary should be implemented in `GameDictionary.java`

2.5 Searching for Words

For the computer's turn, your job is to find all words that the human player missed. In this phase, the same conditions apply as for the human's turn, plus the additional restriction that the computer cannot include any words that were already found by the human.

To do this, you will need to search the Boggle board for all words present. The `getAllWords()` method in the game interface should call one of the two following search strategies. We should be able to switch strategies via the `setSearchTactic()` method.

2.5.1 Board-Driven Search

The first strategy is the obvious one: Recursively search the board for words beginning at each square on the board. As with any exponential search, you should prune the search as much as possible to speed things up. One important strategy is to recognize when you're going down a dead end. For example, if you are searching for words that begin with the letters "ZX", you can use the dictionary's `isPrefix()` method (which you will implement), to determine that there are no English words that begin with this prefix and to recognize that your program can abandon this search path.

2.5.2 Dictionary-Driven Search

You should also implement a second strategy that iterates over all words in the Dictionary and checks whether these words can be found on the given board. There are various tricks you can play to improve the efficiency of this approach. We leave it to you to find these tricks.

2.6 Correctness

The BoggleGame interface allows the game to be developed separately from user interface considerations. We should be able to use your game in our UI or test program without modification. The interface also provides the `setGame()` method, which is *extremely* useful for debugging and testing your search functions.

Your user interface is allowed to assume traditional Boggle rules, although you might wish to make it somewhat more general. Your game and dictionary should not assume these things.

3 The Peer Review Process

The Peer Review component of this assignment will make up half of your project grade. In a nutshell, here's the process, which is double-blind, meaning that you will not know whose code you are reviewing, and you will not know who your reviewers are:

1. You will submit your solution and your report by the first deadline. At the first deadline, you will receive an email message that includes up to four other solutions for you to review. This message will also include forms for submitting your Peer Reviews.

2. You will have three days to complete your Peer Reviews. To produce your Peer Reviews, you may use any test methodology that you wish, including both black box testing and a modified form of white box testing—you will be given bytecodes, so you can invoke methods with well-defined interfaces, but you will not have access to anyone else's source code.

3. After you submit your Peer Reviews, you will receive a message that contains the reviews of your solution along with forms to evaluate each of the reviews. You will then have two days to (1) write your Testing Report and (2) optionally improve your solution based on information that you learned from the Peer Reviews of your code.

   Your Testing Report will explain what you learned from the Peer Review process and will evaluate the quality of the Peer Reviews that you received.

3.1 Writing Your Peer Reviews

Your Peer Review should provide constructive comments on the quality of the code that you're evaluating. Ideally, your review should help the original developers improve their code by describing your test methodology, by describing bugs that you found, and by speculating on possible causes of bugs that you have uncovered. Please be courteous— remember that your reviews will be evaluated by your reviewees.

Your review should include the following components:

- A summary of your testing methodology. This component will likely be the same for each review you write.

- A summary of your findings. Be sure to give your most important points, eg, "We found submission #43 to be extremely robust. The only bugs that we found were in the foo() method, where it appears that the authors have made strange assumptions about ..." Or perhaps, "The basic functionality of submission #7 appears to be correct, but we believe that we have found 13 distinct bugs, including 12 that are related to corner cases . . ."

- Detailed comments. This last component is where you can describe your test cases, the results of specific tests, a list of bugs, etc. Of course, it's most useful if you can organize these comments in some logical fashion, rather than simply providing an unordered list of comments.

3.2 Testing Your Assigned Solutions

We will provide the bytecode for each submission that you are asked to review. In particular, you will receive only the bytecode that corresponds to code that the authors submitted. To test this code, you should place your test code, your dictionary and your word files into a new directory that contains the submitted bytecode. Then recompile your test code in this directory and run it as you would your own code.

Note: It is extremely important that you adhere to the provided interface, because the interface is how other students will access your code. Be certain that your test code and your solution are independent and that other students will be able to run your solution's bytecode in this manner. For this and other reasons, we encourage you to begin developing your testing strategy before the first deadline. By ensuring that you understand how to test someone else's compiled code **before** the first deadline, you can save yourself considerable time and effort during the second phase of this assignment.

3.3 Writing Your Testing Report

Your Testing Report should have three components.

1. Your summary of the overall Peer Review process. You might consider the following questions, though you may of course consider other questions:

   Did you learn anything by writing your testing code/test inputs? Did you learn anything by testing the other solutions? Did you learn anything from the Peer Reviews that you received?

2. Your evaluation of your Peer Reviews. (You should rank order your reviews from best to worst.) Which of your reviews was most useful and why? For each review, were the goals and results clear? Was the testing methodology thorough? Did the review help you identify and fix any bugs?

3. A description of your revised solution (if any). Be sure to explain how the Peer Reviews that you received helped you in revising your solution.

4 What to Turn In

Here are details on the three deadlines and the penalties for missing them.

4.1 Deadline 1

Source code: Turn in electronic copies of `Boggle.java`, `GameManager.java`, and `GameDictionary.java`. Make sure that your code compiles and that all classes have the correct names and are in the correct files. Also make sure that your submission is using the correct directory structure (include `src` and the `assignment` package directory).

If you do not submit your solution on time, you will lose points for being late, and your reviewers get to choose whether or not they will review your solution. Since the reviews can help you test your program to improve your final submission, it's in your best interest to submit your solution on time.

Report: In your report, be sure to include a convincing discussion of the asymptotic running time of the dictionary and the reasons for your design decisions. Also discuss the relative efficiency of the two search tactics. Which one is better for larger or smaller dictionaries? What conditions and parameters affect which strategy you'd want to use? Finally, include any other material that may be relevant, including *game design* and *testing methodology.*

4.2 Deadline 2

Submit your Peer Reviews. It's important that you meet this deadline, because you need to give your reviewees time to interpret your Reviews. If any of your reviews is late, you will receive an automatic reduction in your grade.

4.3 Deadline 3

Submit your Testing Report, any test code that you used to create your Peer Reviews, and (optionally) your revised solution. The only penalty for missing this deadline is a reduction in your grade.