

In this assignment you and your partner will implement the game of Boggle<sup>TM</sup>, 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.

## 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  $4 \times 4$  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
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  
YLDEVV  
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  $4 \times 4$  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 lead to unnecessary complexity, so we advise you to resist this temptation.

## 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 Boggle`.

## 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);
    int [][] getLastAddedWord();
    void     setGame(char [][] board);
    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 specifies an internal iterator: a call to `resetIterator()` will set the iterator to the beginning of the dictionary, and subsequent calls to `nextWord()` will return individual words in the dictionary.

```
public interface BoggleDictionary {
    void      loadDictionary(String filename);
    boolean  isPrefix(String prefix);
    boolean  contains(String word);
    void     resetIterator();
    String   nextWord();
}
```

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

The second strategy that you should implement is to iterate over all words in the Dictionary and check 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

Note that you can test the dictionary independently of the rest of the game. This will greatly modularize your testing.

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. You may want to test Boggle at different sizes and with different data to make sure you aren't making hidden assumptions.

## 3 What to Turn In

**Source code:** Turn in `Boggle.java`, `GameManager.java`, and `GameDictionary.java`. Do not submit other files that were included in the source distribution; they will be disregarded. Make sure that your code compiles and that all classes have the correct names and are in the correct files.

**Report:** A report in the usual format. 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*.

**Acknowledgments.** This assignment was originally developed by Todd Feldman and enhanced by Julie Zelenski and Matt Alden. It has been extensively modified by Walter Chang.