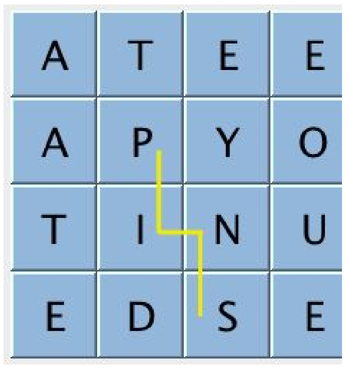
Boggle

Write a program to play the word game Boggle.

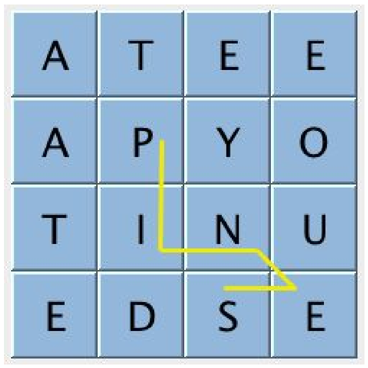
The Boggle game. Boggle is a word game designed by Allan Turoff and distributed by Hasbro. It involves a board made up of 16 cubic dice, where each die has a letter printed on each of its 6 sides. At the beginning of the game, the 16 dice are shaken and randomly distributed into a 4-by-4 tray, with only the top sides of the dice visible. The players compete to accumulate points by building *valid* words from the dice, according to these rules:

- A valid word must be composed by following a sequence of *adjacent dice*—two dice are adjacent if they are horizontal, vertical, or diagonal neighbors.
- A valid word can use each die at most once.
- A valid word must contain at least 3 letters.
- A valid word must be in the dictionary (which typically does not contain proper nouns).

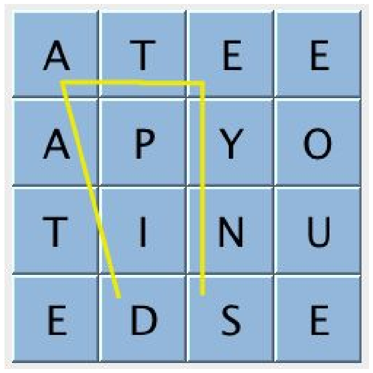
Here are some examples of valid and invalid words:



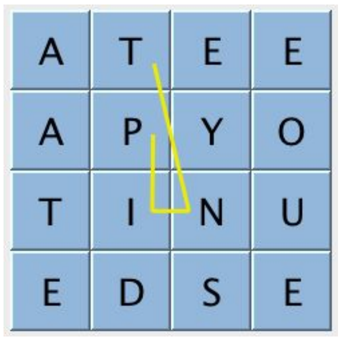
PINS
(*valid*)



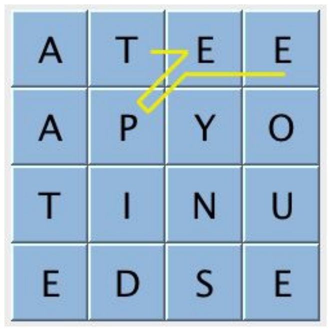
PINES
(*valid*)



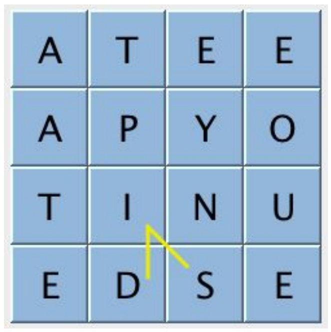
DATES
(*invalid—dice not adjacent*)



PINT
(*invalid—path not sequential*)



TEPEE
(*invalid—die used more than once*)



SID
(*invalid—word not in dictionary*)

Scoring. Words are scored according to their length, using this table:

word length	points
0–2	0
3–4	1
5	2
6	3
7	5
8+	11

The *Qu* special case. In the English language, the letter Q is almost always followed by the letter U. Consequently, the side of one die is printed with the two-letter sequence Qu instead of Q (and this two-letter sequence must be used together when forming words). When scoring, Qu counts as two letters; for example, the word QuEUE scores as a 5-letter word even though it is formed by following a sequence of only 4 dice.

Your task. Your challenge is to write a Boggle solver that finds *all* valid words in a given Boggle board, using a given dictionary. Implement an immutable data type BoggleSolver with the following API:

```
public class BoggleSolver {
    // Initializes the data structure using the given array of strings as the dictionary.
    // (You can assume each word in the dictionary contains only the uppercase letters A through
Z.)
    public BoggleSolver(String[] dictionary)

    // Returns the set of all valid words in the given Boggle board, as an Iterable.
    public Iterable<String> getAllValidWords(BoggleBoard board)

    // Returns the score of the given word if it is in the dictionary, zero otherwise.
    // (You can assume the word contains only the uppercase letters A through Z.)
    public int scoreOf(String word)
}
```

It is up to you how you search for and store the words contained in the board, as well as the dictionary used to check them.

The board data type. We provide an immutable data type BoggleBoard.java for representing Boggle boards. It includes constructors for creating Boggle boards from either the 16 Hasbro dice, the distribution of letters in the English language, a file, or a character array; methods for accessing the individual letters; and a method to print out the board for debugging. Here is the full API:

```
public class BoggleBoard {
    // Initializes a random 4-by-4 Boggle board.
    // (by rolling the Hasbro dice)
    public BoggleBoard()
    // Initializes a random m-by-n Boggle board.
    // (using the frequency of letters in the English language)
    public BoggleBoard(int m, int n)
    // Initializes a Boggle board from the specified filename.
    public BoggleBoard(String filename)
    // Initializes a Boggle board from the 2d char array.
    // (with 'Q' representing the two-letter sequence "Qu")
    public BoggleBoard(char[][] a)
    // Returns the number of rows.
    public int rows()
    // Returns the number of columns.
    public int cols()
    // Returns the letter in row i and column j.
    // (with 'Q' representing the two-letter sequence "Qu")
    public char getLetter(int i, int j)
    // Returns a string representation of the board.
    public String toString()
}
```

- *Dictionaries.* A dictionary consists of a sequence of words, separated by whitespace, in alphabetical order. You can assume that each word contains only the uppercase letters A through Z. For example, here are the two files dictionary-algs4.txt and dictionary-yawl.txt:

```
% more dictionary-algs4.txt      % more dictionary-yawl.txt
ABACUS                           AA
ABANDON                            AAH
ABANDONED                            AAHED
ABBREVIATE                          AAHING
...                        ...
QUEUE
PNEUMONOULTRAMICROSCOPICSILICOVOLCANOCONIOSIS
...                        ...
ZOOLOGY                            ZYZZYVAS
```

- The former is a list of 6,013 words that appear in *Algorithms 4/e*; the latter is a comprehensive list of 264,061 English words (known as *Yet Another Word List*) that is widely used in word-game competitions.
- *Boggle boards.* A boggle board consists of two integers *M* and *N*, followed by the *M* × *N* characters in the board, with the integers and characters separated by whitespace. You can assume the integers are nonnegative and that the characters are uppercase letters A through Z (with the two-letter sequence Qu represented as either Q or Qu). For example, here are the files board4x4.txt and board-q.txt:

```
% more board4x4.txt        % more board-q.txt
4 4                        v4 4
A T E E                     S N R T
A P Y O                      O I E L
T I N U                     E Qu T T
E D S E                      R S A T
```

The following test client takes the filename of a dictionary and the filename of a Boggle board as command-line arguments and prints out all valid words for the given board using the given dictionary.

```
public static void main(String[] args) {
    In in = new In(args[0]);
    String[] dictionary = in.readAllStrings();
    BoggleSolver solver = new BoggleSolver(dictionary);
    BoggleBoard board = new BoggleBoard(args[1]);
    int score = 0;
    for (String word : solver.getAllValidWords(board)) {
        StdOut.println(word);
        score += solver.scoreOf(word);
    }
    StdOut.println("Score = " + score);
}
```

Here are two sample executions:

```
% java-algs4 BoggleSolver dictionary-algs4.txt board4x4.txt     % java-algs4 BoggleSolver
dictionary-algs4.txt board-q.txt
AID                                       EQUATION
DIE                                       EQUATIONS
END                                        ...
ENDS                                        QUERIES
...                                     QUESTION
YOU                                        QUESTIONS
Score = 33                                   ...
                                        TRIES
                                        Score = 84
```

Performance. If you choose your data structures and algorithms judiciously, your program can preprocess the dictionary and find all valid words in a random Hasbro board (or even a random 10-by-10 board) in a fraction of a second. To stress test the performance of your implementation, create one BoggleSolver object (from a given dictionary); then, repeatedly generate and solve random Hasbro boards. How many random Hasbro boards can you solve per second? *For full credit, your program must be able to solve thousands of random Hasbro boards per second.* The goal on this assignment is raw speed—for example, it's fine to use 10× more memory if the program is 10× faster.

Possible progress steps

1. Complete the code in the BoggleSolver.java, Solution.java (main()) will help you to take inputs and print outputs.
2. Familiarize yourself with the BoggleBoard.java data type.
3. Use a standard set data type to represent the dictionary, e.g., a SET<String>, a TreeSet<String>, or a HashSet<String>.
4. Create the data type BoggleSolver. Write a method based on depth-first search to enumerate all strings that can be composed by following sequences of adjacent dice. That is, enumerate all simple paths in the Boggle graph (but there is no need to explicitly form the graph). For now, ignore the special two-letter sequence Qu.
5. Now, implement the following critical backtracking optimization: when the current path corresponds to a string that is not a prefix of any word in the dictionary, there is no need to expand the path further. To do this, you will need to create a data structure for the dictionary that supports the prefix query operation: given a prefix, is there any word in the dictionary that starts with that prefix?
6. Deal with the special two-letter sequence Qu.

Possible Optimizations

1. You will likely need to optimize some aspects of your program to pass all of the performance points (which are, intentionally, more challenging on this assignment). Here are a few ideas:
2. Make sure that you have implemented the critical backtracking optimization described above. This is, by far, the most important step—several orders of magnitude!
3. Think about whether you can implement the dictionary in a more efficient manner. Recall that the alphabet consists of only the 26 letters A through Z.
4. Exploit that fact that when you perform a prefix query operation, it is usually almost identical to the previous prefix query, except that it is one letter longer.
5. Consider a nonrecursive implementation of the prefix query operation.
6. Precompute the Boggle graph, i.e., the set of cubes adjacent to each cube. But don't necessarily use a heavyweight Graph object.