EE422C Fall 2016
Project 3: Word Ladders
Due (see Canvas)

You must work in teams of two for this project.
The aim of this assignment is to give you experience working with various collections, as well as to strengthen your algorithm and Abstract Data Type design skills.

Problem Statement:

A *word ladder*[1] is a (finite) sequence of distinct words from the English language such that any two consecutive words in the sequence differ by changing one letter at a time, with the constraint that each of the resulting string of letters is a legitimate word. For example, to turn "stone" into "money", one possible word ladder is:

```
stone
Atone
aLone
Clone
clonS
cOons
coNns
conEs
coneY
Money
```

Capital letters are used in the example above only to illustrate the connections. Obviously, there could be more than one word ladder between "`stone`" and "`money`". You only have to find one of them for each word pair given.

In this assignment, you are to design and implement a Java program that for any given pair of words, generates a word ladder that connects those two words (making use of a given dictionary of legal English words). If a word ladder does not exist between the given pair, your program should output a message that says so. You are required to find a word ladder, not necessarily the shortest word ladder.

Input and Output Requirements

Your program will read commands from the standard input (i.e., from the keyboard). The basic command consists of a pair of words separated by at least one space (with no intervening punctuation or other words). Commands are case-insensitive. For consistency, we require that all output be lower case. After reading both words, your program must determine if there is a word ladder between the two words. For example, the command

```
smart money
```

[1] http://www.learnenglish.org.uk/words/activities/revers01.html

instructs your program to search the dictionary (the dictionary is described below) to find a word ladder that starts with "`smart`" and ends with "`money`". If a word ladder can be found, your program must print the message

```
a <N>-rung word ladder exists between <start> and <finish>.
<start>
<first rung>
<second rung>
<…>
<finish>
```

Where <N> must be replaced with the number of intervening words in the word ladder between the start and finish words (i.e., don't count start or finish in your calculation of N). You must then print each of the words in the word ladder on a line by itself, with no initial white space before each line starts. For the command "smart money", your program might produce the following

```
a 8-rung word ladder exists between smart and money.
smart
start
stars
soars
socks
cocks
conks
cones
coney
money
```

Note that to be a valid word ladder, every word in the ladder must be a legal word that appears in the dictionary. If your program cannot find a valid word ladder between the two words, you must print

```
no word ladder can be found between <start> and <finish>.
```

You can be sure that the start word and end word are different, and that both will be part of the dictionary.

In addition to the basic command, your program must recognize one more command, and that is /quit. The command /quit must result in your program terminating with no further output.
Note that whitespace, including tab characters and newline characters, is to be ignored (treated like spaces) when you are reading the input. This whitespace policy applies for any commands (including basic commands) read from the standard input.

If you enter only one word, and it is not /quit, you may do as you like. It will not be tested. We will also not be testing commands of multiple words or words not in the dictionary.

For grading and testing purposes, we have also provided provision in the starter code to replace the keyboard and console with an input file and output file respectively. You don't need to use these if you don't want to – just use the Scanner normally.

Dictionary: You may test your project with the dictionary contained in the file named "`five_letter_words.txt`", which is a text file that consists of a collection of English words with five letters each. We have supplied code to generate a dictionary in the form of a `Set` object. Clearly, using this dictionary, it will only be possible to find word ladders when the starting word and the finishing words are both five letters long.

Implementation requirements and suggestions

Requirements

Many of the requirements are to facilitate automated grading, so you must obey them. Contact us if you have problems.

1. Your program is divided into three parts – getting the start and end words, calculating the word ladder, and printing the output.
2. The supplied .java file shows you that you must have only one `Scanner` object connected to the keyboard in your program, and it must be in `main()`. You may pass it as a parameter to other methods. This step has already been done for you. Other Scanner objects are permitted, as long as they are not connected to `Stdin` (keyboard, `System.in`).
3. You must have 5 methods in your `Main`, besides `main()`, as shown in the starter code.
   a. `public static void initialize()`
   b. `public static ArrayList<String> parse(Scanner keyboard)`
   c. `public static ArrayList<String> getWordLadderDFS(String start, String end)`
   d. `public static ArrayList<String> getWordLadderBFS(String start, String end)`
   e. `public static void printLadder(ArrayList<String> ladder)`

   We use JUNIT to test your methods. Each testcase is run on a newly created instance of your Main class. So make sure that your static variables' values are not altered inappropriately; for example, it would not be a good idea to depend on one run of `getWordLadderXXX` to depend on a previous run of the same method via some static variable setting. Part of your deliverable is a description of your methods in (c) and (d).
4. Your ladder solution may not have loops in it i.e. the same word may not be visited twice. In your DFS method, you must use some method to attempt to reduce the length of the ladder. See the suggestions following this section. Credit will be given for a good method even if it doesn't work well for all cases, and you should document this method in your code comments. If you do not find such a ladder, or if start and end are the same word, you must return an empty list. This method's signature is in your shell .java file.
5. You must ignore case. The dictionary `Set` itself has all uppercase words. You may convert this `Set` to any other data type (such as `ArrayList`) that you like. The dictionary creation also has been done for you. You may change the filename for testing, but remember to restore the name for submission.
6. You must call the `makeDictionary` method from within `getWordLadderXXX` each time you call it. Alternately, you may call `makeDictionary` once from within `initialize()`.

7. You may create other class files; remember to turn them in.
8. Any methods you create in Main to use in `getWordLadderXXX` should be static. If you create other classes, their methods need not be static. If these restrictions are too hard for you to program with, contact us for suggestions.
9. You must implement `WordLadderXXX` with **both** DFS and BFS.
10. Your DFS must be implemented with recursion.
11. You need to submit a test plan in PDF to describe how you tested your program and write at least five non-trivial test cases for each of both DFS and BFS implementations. The test-plan outline is provided for you.
12. You need to provide a team plan that shows how you have been working together. Canvas has a file that shows what to put in your team plan.
13. You must work together on both DFS and BFS. It is not acceptable that one person does only BFS and the other only DFS. At the very least you should test each other's code.

Submission requirements

Create a package named `assignment3` for all of your source code. Create a public class inside package `assignment3` called `Main.java` that contains `main()`**.** Remember to put all both team member names and UTEIDs on the `.java` files (and other files) if you are working as a pair. Put all `.java` files (or file, if you have only one file in your solution, `Main.java`), required PDF, and `README` and other docs into the folder and zip your final version of those files before the submission deadline.

Name of zip file: Project3_EID1_EID2.zip (.gzip of .gz are also ok). Omit `_EID2` if you are working alone. Make sure that the structure of the final ZIP file is as follows:

```
Project3_EID1_EID2/
    test_plan.pdf
    team_plan.pdf
    <other non-code files>
    src/
        assignment3/
            Driver.java
            file2.java
            ...
```

ZIP your src folder and your other files together. Then rename that ZIP to `Project3_EID1_EID2.zip`.

Suggestions

- While using recursion, remember not to overrun resources, such as stack memory with too many nested calls. For example, you might want to keep track of words you have visited that are dead ends (that don't lead to the end word). There are ways to do DFS using the Stack data structure without recursion that don't lead to stack overflows, but we don't want you to do that. The instructor solution did not have a stack overflow for any word combination. If you absolutely keep getting stack overflows, contact us for help.
- Given a choice of letters to change to get to the next ladder step, it might help to pick a change that leads to a word that is as close to the end word as possible.

Additional Considerations

- External Code – you are permitted to use any classes or interfaces within the `java.lang`, `java.io`, and `java.util` standard packages. You are specifically prohibited from making use of another student's code (including students who may have taken the class in previous semesters). You are also not permitted to use external packages such as `Graph`.
- Understandability – Comment your program so that its logic would be readily apparent to any software engineer who is familiar with standard data structures and algorithms. Use `Javadoc` style comments for public methods.
- Re-use – Design your code so it is suitable for future adaptations and/or expansion.
- Efficiency Risk – It is possible that additional problem/solution constraints may be needed in order to guarantee that your program runs in a reasonable amount of time and/or space. These may be specified later.

Warning - You may not acquire, from any source (e.g., another student or an internet site), a partial or complete solution to this problem. Except for your partner, you may not show another student your solution to this assignment. You may not have another person (TA, current student, former student, tutor, friend, anyone) "walk you through" how to solve this assignment. Review the class policy on cheating from the syllabus.

Tip - Each team is required to use the git repository for interim and final versions of your code. The rule of thumb is that you should commit at least once per working day when code is being generated, preferably at the end of each working session where code is being changed. Please see the git tutorial provided by your TA for more information on how to use that.

CHECKLIST – Did you remember to:

- ☐ Follow pair programming guidelines, including recording each partner's share of the work?
- ☐ Re-read the requirements after you finished your program to ensure that you meet all of them?
- ☐ Make up your own testcases?
- ☐ Use the Git repository regularly?
- ☐ Provide your team plan document?
- ☐ Make sure that all your submitted files have the appropriate header file and package statement?
- ☐ Upload your solution to Canvas, remembering to include ALL your files in one zip file name Project3_EID1_EID2.zip (.gzip of .gz are also ok)?
- ☐ Download your uploaded solution into a fresh directory and re-run all testcases?

Adapted from an assignment written by Herb Kreisner and Mike Scott.