**Word Ladder FAQ**

***How do I construct a "nested" collection?***

Example: Queue<Stack<String>> queue = new LinkedList<>();

/\* Queue is an interface and can't be instantiated, use a LinkedList object (which implements Queue). Both must be parameterized (Queue<Stack> would be a Queue of Stacks of Objects) \*/

***My program is so slow. Is this normal?***

A common source of slowness is looping over the entire dictionary to look for words. This works (and is technically what the pseudo-code suggests), but is very slow. The realpurpose of the dictionary is to look up words, to test whether something is a valid word or not, not to loop over. If you want to loop over all possible "neighbor" words that are one letter away from a given word, you'll need to devise a better algorithm. You could also significantly improve performance by only looking at some relevant subset of the dictionary.

In addition, new data structure called a ***set*** is useful in that, for *any* number of elements, lookups always happen in constant (O(1)) time (whereas an ArrayList would be linear time (O(n))). We will talk about how this is possible (with "hashing") later this year. To make a set object, use the following:

Set<String> set = new HashSet<>();

//Set is the super-type, a HashSet object ***is-a*** Set

***How many seconds is my program allowed to take? Mine finds the ladders in*** *m* ***seconds.***

There is no specific time limit, but you should choose appropriate data structures and write your algorithm efficiently. This program, unavoidably, is computationally expensive - there are a LOT of comparisons and lookups being made! However, good solutions should complete all the ladders in the "input.txt" file in under a minute (great solutions should be nearly instantaneous for all but the last couple ladders). My program outputs all of the ladders in the "input.txt" file in roughly 2.5s (and could be optimized further).

***Why does my program spit out a VERY long (20-30+ word) ladder even for simple cases?***

Maybe you're accidentally adding words to an existing collection rather than making a copy of it and adding to the copy. Run with a very small input, then print out your collections along the way and make sure everything is what you expect.

***If the word ladder is invalid for multiple reasons, which error message should I print?***

It's up to you, follow your *heart.*

***Why do I need to keep track of used words? (or, why do I have an infinite loop?)***

Suppose you have the partial ladder cat → cot → cog in the queue. Later on, if your code is processing ladder cat → cot → con, one neighbor of *con* is *cog*, so you might want to examine cat → cot → con → cog. But doing so is unnecessary; your goal is to find the *shortest* valid word ladder, and you have already found one from *cat* to *cog* without the unnecessary word *con*. There is no reason to enqueue the longer ladder.

If you follow this rule properly, it will ensure that as soon as you've enqueued a ladder ending with a specific word, you've found a minimum-length path from the starting word to the end word in the ladder, so you will ever have to enqueue that end word again. To implement this strategy, keep track of the set of words that have already been used in any ladder, and ignore those words if they come up again. Keeping track of what words you've used also eliminates the possibility of getting trapped in an infinite loop by building a circular ladder, such as cat → cot → cog → bog → bag → bat → **cat**.

***Should I create helper classes for this program, e.g. for a single Word?***

You *could*, but honestly it shouldn't be necessary for this project. With knowledge of linked structures, which we'll talk about later, it would be feasible. At this point though, a single class (plus a client/runner class) should be sufficient.

***Should I use regex for checking if the words differ by one character?***

That could work, though it probably isn't the fastest methodology.

***Should I use recursion for \_\_\_?***

No. There would be a LOT of method calls in this project; using recursion will probably cause a StackOverflowException (unless you do some extensive optimization).

***My program puts everything into one Stack. Help!***

If you copy the value of a reference type (an object), the copied value will be referencing the original object. Anything done to the copy will affect the original object. You'll have to find some way around this.

***My program says none of the words in the input file are in the dictionary???***

All the words in the dictionary are in upper case. Your program should ignore case.

***Why a queue of stacks??***

Conceptually, a ladder is very similar to a stack (both go "up"). When adding a "next" word to the ladder, you *could* use a list (which goes "out"), but why? A stack models this process much more closely. In addition, you only interact with the latest element (the current word in the potential ladder), which is exactly how stacks work!

Each stack is a separate "branch" of possible word ladders. In addition, you could definitely use a List<Stack<String>>, but you would constantly be appending stacks to the end of the list. Wouldn't it be better to process stacks as they're created, i.e. a First-In-First-Out type of order?

You could use many different types of data structures in this project, but a queue of stacks is a perfectly good choice (especially as we're learning stacks and queues).

***How can I loop through all the letters in the alphabet?***

for (char ch = 'a'; ch <= 'z'; ch++) { ... }

If you never did the CodesAndCiphers lab in PAP / CS 1, now might be a good time to look at the "The char type (advanced)" powerpoint in the CS 1 folder. Looping through all the letters in the alphabet may be useful for improving your program's run time (though this is not the only way to improve performance).