Assignment 2: Word Play

Parts of this handout were written by Julie Zelenski, Eric Roberts, Jerry Cain, and Marty Stepp.

This assignment is all about computationally exploring words in the English language. It's a two-parter. In the first, you'll get some exposure to a powerful algorithm using the `Queue` type, and in the second, you'll build a surprisingly entertaining word game that repeatedly trounces its human player. We hope you have fun with this one!

By the time you've completed this assignment, you'll have a much better handle on the container classes and how to use different types like queues, maps, vectors, and sets to model and solve problems. Plus, you'll have some things we think you'd love to share with your friends and family.

Due Monday, January 30th at the start of class.

This assignment must be completed individually.
Working in pairs is not permitted.

Part One: Word Ladders

A ***word ladder*** is a connection from one word to another formed by changing one letter in the word at a time, with the constraint that at each step the sequence of letters still forms a valid word. For example, here is a word ladder connecting the word `code` to the word `data`:

```
code → core → care → dare → date → data
```

That word ladder, however, is not the shortest possible one. Although the words may be a little less familiar, the following ladder is one step shorter:

```
code → cade → cate → date → data
```

(In case you're wondering, `cade` is a type of Juniper tree, and `cate` is a special food.)

Your job in this problem is to write a program that finds a minimal word ladder between two words. Your code will make use of several of the collections classes we explored this week, along with a powerful algorithm called breadth-first search that finds the fastest way to convert one word to another. Here, for example, is a sample run of the word-ladder program in operation:

```
Welcome to CS106B Word Ladders!

Give me two English words, and I will change the first
into the second by replacing one letter at a time.

Word 1: work
Word 2: play
Found the shortest ladder! Here it is:
work -> fork -> form -> foam -> flam -> flay -> play

Want to find another word ladder? no
Have a great day!
```

We've bundled this example program with the starter files. To run it, double-click the `WordLadderReference.jar` file in the project directory.

Finding a word ladder is a specific instance of a ***shortest-path problem***, in which the challenge is to find the shortest path from a starting position to a goal. Shortest-path problems come up in a variety of situations such as routing packets in the Internet, robot motion planning, determining proximity in social networks, comparing gene mutations, and more. We'll talk about shortest paths more later this quarter.

One strategy for finding a shortest path uses a famous algorithm called as ***breadth-first search***, which is a search process that expands outward from the starting position, considering first all possible solutions that are one step away from the start, then all possible solutions that are two steps away, and so on, until an actual solution is found. Because you check all the paths of length 1 before you check any of length 2, the first successful path you encounter must be as short as any other.

For word ladders, the breadth-first strategy starts by examining those ladders that are one step away from the original word, which means that only one letter has been changed. If any of these single-step changes reach the destination word, you're done. If not, you can then move on to check all ladders that are two steps away from the original, which means that two letters have been changed. In computer science, each step in such a process is called a ***hop***.

The breadth-first algorithm is typically implemented by using a queue to store partial ladders that represent possibilities to explore. The ladders are enqueued in order of increasing length. The first elements enqueued are all the one-hop ladders, followed by the two-hop ladders, and so on. Because queues guarantee first-in/first-out processing, these partial word ladders will be dequeued in order of increasing length.

To get the process started, you simply add a ladder consisting of only the start word to the queue. From then on, the algorithm operates by dequeueing the ladder from the front of the queue and determining whether it ends at the goal. If it does, you have a complete ladder, which must be minimal. If not, you take that partial ladder and extend it to reach words that are one additional hop away, and enqueue those extended ladders, where they will be examined later. If you exhaust the queue of possibilities without having found a completed ladder, you can conclude that no ladder exists.

It is possible to make the algorithm considerably more concrete by implementing it in ***pseudocode***, which is simply a combination of actual code and English. Here's some pseudocode for breadth-first search:

```
create an empty queue;
add the start word to the end of the queue;

while (the queue is not empty) {
   dequeue the first ladder from the queue;

   if (the final word in this ladder is the destination word) {
       return this ladder as the solution;
   }

   for (each word in the lexicon of English words that differs by one letter) {
      if (that word has not already been used in a ladder) {
         create a copy of the current ladder;
         add the new word to the end of the copy;
         add the new ladder to the end of the queue;
      }
   }
}
return that no word ladder exists;
```

As is generally the case with pseudocode, several of the operations that are expressed in English need to be fleshed out a bit. For example, the loop that reads

```
for (each word in the lexicon of English words that differs by one letter)
```

is a conceptual description of the code that belongs there. It is, in fact, unlikely that this idea will correspond to a single for loop in the final version of the code. The basic idea, however, should still make sense. What you need to do is iterate over all the words that differ from the current word by one letter. One strategy for doing so is to use two nested loops; one that goes through each character position in the word and one that loops through the letters of the alphabet, replacing the character in that index position with each of the 26 letters in turn. Each time you generate a word using this process, you need to look it up in the lexicon of English words to make sure that it is actually a legal word.

Another issue that is a bit subtle is the restriction that you not reuse words that have been included in a previous ladder. One advantage of making this check is that doing so reduces the need to explore redundant paths. For example, suppose that you have previously added the partial ladder

```
cat → cot → cog
```

to the queue and that you are now processing the ladder

```
cat → cot → con.
```

One of the words that is one hop away from `con`, of course, is `cog`, so if you're not careful your program might enqueue the ladder

```
cat → cot → con → cog
```

unnecessarily, since this can't possibly be the shortest word ladder. (Do you see why?) In fact, as soon as you've enqueued a ladder ending with a specific word, you never have to enqueue that word again. The

simplest way to implement this strategy is to keep track of the words that have been used in any ladder (which you can easily do using another lexicon) and ignore those words when 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 → ...
```

One of the other questions you will need to resolve is what data structure you should use to represent word ladders. Conceptually, each ladder is just an ordered list of words, which should make your mind scream out "`Vector`!" (Given that all the growth is at one end, stacks are also a possibility, but vectors will be more convenient when you are trying to print out the results.) The individual components of the Vector are of type string.

Implementing Word Ladders

At this point, you have everything you need to start writing the actual C++ code to get this project done. It's all about leveraging the right container types; you'll find your job is just to coordinate the activities of various different queues, vectors, and lexicons necessary to get the job done. The finished assignment doesn't require all that much code, so it's not a question of typing in statements until your fingers get tired (in fact, if you're doing this, it likely means that you're making things harder than they need to be!) It will, however, certainly help to think carefully about the problem before you actually begin that typing.

The program that you write should do the following:

1. ***Load the dictionary.*** The file `EnglishWords.dat`, which is bundled with the starter files, contains just about every legal English word.
2. ***Prompt the user for two words to try to connect with a ladder.*** For each of those words, make sure to reprompt the user until they enter valid English words. They don't necessarily have to be the same length, though – if they aren't, it just means that your search won't find a word ladder between them.
3. ***Find the shortest word ladder.*** Use breadth-first search, as described before, to search for a word ladder from the first word to the second.
4. ***Report what you've found.*** Once your breadth-first search terminates:
    1. If you find a word ladder, print it out to the console in whatever format seems best – one line at a time, separated by arrows, separated by spaces, etc. Then, call the function

        `recordLadderBetween(`*start-word*`,` *end-word*`,` *ladder*`)`

        to report that you've found a word ladder. This function is bundled with the starter code and we'll use it when grading your assignment (more on this later on.)
    2. If you *don't* find a word ladder, print out some message to that effect, then call the function

        `recordNoLadderBetween(`*start-word*`,` *end-word*`)`

        to report that no ladder exists.
5. ***Ask to continue.*** Prompt for whether to look for another ladder between a pair of words.

Here are a few general pieces of advice for this part of the assignment:

- ***Make sure you understand the relevant data structures***. As mentioned earlier, you don't need to write all that much code for this assignment. However, the code you write will have to make use of the `Lexicon`, `Vector`, and `Queue` types. Before you start crafting your program, take some time to read over what they do and review the relevant parts of the textbook (Chapter 5) and lecture materials, and feel free to hop on Piazza or stop by the LaIR with questions!

- ***Watch for case sensitivity***. If the user wants to find a word ladder starting `work` and ending at `life`, it shouldn't matter if they enter `work`, `Work`, `wORk`, or `WoRK` as the starting `word` or `life`, `LIFE`, or `lIfE` as the ending word. You might find it useful to know that every word in our word list is stored in lower case, and you can rely on that fact.
- ***Don't fret about ties***. If there are multiple different word ladders of the same length that can take you from one word to another, you're welcome to report any of them as the final word ladder that you come up with. The version of breadth-first search we provided in pseudocode already does this, so you shouldn't need to put in any special handling to make this work.
- ***Make sure you call our special reporting functions***. The functions `recordLadderBetween` and `recordNoLadderBetween` are designed to work with our autograder, which we'll use to run a bunch of different tests on your code once you submit it. ***If you don't remember to call these functions, your code will fail a lot of our automated tests and your grade might be substantially lower than expected!***

Part Two: Evil Hangman

It's hard to write computer programs to play games. When we humans sit down to play games, we can draw on past experience, adapt to our opponents' strategies, and learn from our mistakes. Computers, on the other hand, blindly follow a preset algorithm that (hopefully) causes it to act intelligently. Though computers have bested their human masters in some games (including, recently, Go), the programs that do so often draw on hundreds of years of human experience and use extraordinarily complex algorithms and optimizations to outcalculate their opponents.

While there are many viable strategies for building competitive computer game players, there is one approach that has been fairly neglected in modern research – cheating. Why spend all the effort trying to teach a computer the nuances of strategy when you can simply write a program that plays dirty and wins handily all the time? In this assignment, you will build a mischievous program that bends the rules of Hangman to trounce its human opponent time and time again. In doing so, you'll cement your skills with the container types and will hone your general programming savvy. Plus, you'll end up with a highly entertaining piece of software, at least from your perspective. ☺

In case you aren't familiar with the game Hangman, the rules are as follows:

1. One player chooses a secret word, then writes out a number of dashes equal to the word length.
2. The other player begins guessing letters. Whenever she guesses a letter in the hidden word, the first player reveals each instance of that letter in the word. Otherwise, the guess is wrong.
3. The game ends when all letters in the word have been revealed or when no guesses remain.

Fundamental to the game is the fact the first player accurately represents the word she has chosen. That way, when the other players guess letters, she can reveal whether that letter is in the word. But what happens if the player doesn't do this? This gives the player who chooses the hidden word an enormous advantage. For example, suppose that you're the player trying to guess the word, and at some point you end up revealing letters until you arrive at this point with only one guess remaining:

```
D O - B L E
```

There are only two words in the English language that match this pattern: `DOABLE` and `DOUBLE`. If the player who chose the hidden word is playing fairly, then you have a fifty-fifty chance of winning this game if you guess 'A' or 'U' as the missing letter. However, if your opponent is cheating and hasn't actually committed to either word, then there is no possible way you can win this game. No matter what letter you guess, your opponent can claim that she had picked the other word, say that your guess is incorrect, and win the game. That is, if you guess that the word is "doable," she can pretend that she committed to "double" the whole time, and vice-versa.

Let's illustrate this technique with an example. Suppose that you are playing Hangman and it's your turn to choose a word, which we'll assume is of length four. Rather than committing to a secret word, you instead compile a list of every four-letter word in the English language. For simplicity, let's assume that English only has a few four-letter words, all of which are reprinted here:

```
ALLY    BETA    COOL    DEAL    ELSE    FLEW    GOOD    HOPE    IBEX
```

Now, suppose that your opponent guesses the letter 'E.' You now need to tell your opponent which letters in the word you've "picked" are E's. Of course, you haven't picked a word, and so you have multiple options about where you reveal the E's. Here's the above word list, with E's highlighted in each word:

```
ALLY    BETA    COOL    DEAL    ELSE    FLEW    GOOD    HOPE    IBEX
```

Now, suppose that your opponent guesses the letter 'E.' If you'll notice, every word in your word list falls into one of five "word families:"

- `----`, which contains the word `ALLY`, `COOL`, and `GOOD`.
- `-E--`, containing `BETA` and `DEAL`.
- `--E-`, containing `FLEW` and `IBEX`.
- `E--E`, containing `ELSE`.
- `---E`, containing `HOPE`.

Since the letters you reveal have to correspond to some word in your word list, you can choose to reveal any one of the above five families. There are many ways to pick which family to reveal – perhaps you want to steer your opponent toward a smaller family with more obscure words, or toward a larger family in the hopes of keeping your options open. For this assignment, in the interests of simplicity, we'll adopt the latter approach and always choose the largest of the remaining word families. In this case, it means that you should pick the family `----`. This reduces your word list down to

```
ALLY    COOL    GOOD
```

and, since you didn't reveal any letters, you would tell your opponent that his guess was wrong.

Let's see a few more examples of this strategy. Given this three-word word list, if your opponent guesses the letter O, then you would break your word list down into two families:

- `-OO-`, containing `COOL` and `GOOD`.
- `----`, containing `ALLY`.

The first of these families is larger than the second, and so you choose it, revealing two O's in the word and reducing your list down to

```
COOL    GOOD
```

But what happens if your opponent guesses a letter that doesn't appear anywhere in your word list? For example, what happens if your opponent now guesses 'T'? This isn't a problem. If you try splitting these words apart into word families, you'll find that there's only one family: the family `----` containing both `COOL` and `GOOD`. Since there is only one word family, it's trivially the largest, and by picking it you'd maintain the word list you already had.

There are two possible outcomes of this game. First, your opponent might be smart enough to pare the word list down to one word and then guess what that word is. In this case, you should congratulate her – that's an impressive feat considering the scheming you were up to! Second, and by far the most common case, your opponent will be completely stumped and will run out of guesses. When this happens, you can pick any word you'd like from your list and say it's the word that you had chosen all along. The beauty of this setup is that your opponent will have no way of knowing that you were dodging guesses the whole time – it looks like you simply picked an unusual word and stuck with it the whole way.

The Assignment

Your assignment is to write a computer program which plays a game of Hangman using this "Evil Hangman" algorithm. In particular, your program should do the following:

1. ***Set up the game***. Prompt the user for a word length, prompting as necessary until she enters a number such that there's at least one word that's exactly that long. Then, construct a list of all words in the English language whose length matches the input length. (We've provided you the same `EnglishWords.dat` file that we gave you for Word Ladders, and you should use that as your go-to source for all the words in English.)

   Next, prompt the user for a number of guesses, which must be an integer greater than zero. Don't worry about unusually large numbers of guesses – after all, having more than 26 guesses is not going to help your opponent!

   Finally, prompt the user for whether she wants to have a running total of the number of words remaining in the word list. This completely ruins the illusion of a fair game that you'll be cultivating, but it's quite useful for testing.

   Please prompt the user for this information in the order specified above. Part of our assignment grading is done automatically, and our autograders will try to enter this information in this order.

2. ***Play the game using the Evil Hangman algorithm***. Specifically, you should do the following:

   1. Print out how many guesses the user has remaining, along with any letters the player has guessed and the current blanked-out version of the word. If the user chose earlier to see the number of words remaining, print that out too.

   2. Call our special `recordTurnInfo` function to record some information about the game:

      `recordTurnInfo(`*turn-number*`,` *current-blanked-word*`,` *letters-guessed-so-far*`,`
      num-remaining-words`,` *num-guesses-left*`);`

      The first turn is Turn 1. You can specify the guessed letters in whatever format you'd like.

   3. Prompt the user for a single letter guess, reprompting until the user enters a letter that she hasn't guessed yet. Make sure that the input is exactly one character long and that it's a letter.

   4. Partition the words in the dictionary into their respective word families.

   5. Find the most common "word family" in the remaining words, remove all words from the word list that aren't in that family, and report the positions of the letter guessed (if any) to the user. If the word family doesn't contain any copies of the letter, subtract a guess from the user.

   6. Repeat until the game ends.

3. ***Report the final result***. The game ends when the player is down to zero guesses or when the player has revealed all the blanks in the word. When that happens:

   1. If the player ran out of guesses, pick a word from the remaining word list (choose it however you'd like) and print a message that this was the word the computer initially "chose."

   2. If the player revealed all the blanks, print out that resulting word and congratulate them!

   3. Call our special `recordGameEnd` function to report that the game is over. The syntax is

      `recordGameEnd(`*final-word*, *player-won*`);`

      where *final-word* is the word you revealed at the end – either one of the remaining words if the player lost or the word the player guessed if they won – and *player-won* is a `bool` indicating whether or not the player won the game.

4. ***Ask to play again***. Prompt the user and ask whether they want to play another game.

Advice, Tips, and Tricks

The starter code for this particular assignment is essentially blank, and you'll be building it from scratch. Consequently, you'll need to do a bit of planning to figure out what the best data structures are for the program. There is no "right way" to go about writing this program, but some design decisions are much better than others (e.g. you can store your word list in a stack or map, but this is probably not the best option). Here are some general tips and tricks that might be useful:

- ***Try out our demo program so you can see what to expect***. We've bundled a reference solution with the starter files. To run it, double-click the `EvilHangmanReference.jar` file in the project directory. This will give you a better sense of what we're looking for.
- ***Choose your data structures carefully.*** It's up to you to think about how you want to partition words into word families. Think about what data structures would be best for tracking word families and the master word list. Would a `Vector` work? How about a `Map`? A `Stack` or `Queue`? Thinking through the design before you start coding will save you a lot of time and headaches.
- ***Decompose the problem into smaller pieces***. You'll need to write a decent amount of code to solve this problem, but that code nicely splits apart into a bunch of smaller pieces, things like "group words by their word family" or "read a letter from the user." Try to keep your functions short if at all possible, and follow good principles of top-down design.
- ***Use Map's autoinsertion feature.*** Unlike the Java `HashMap`, if you look up a key/value pair in a C++ `Map` and the key doesn't exist, the `Map` will automatically insert a new key/value pair for you, using an intelligent default for the value. For example, if you have a variable of type `Map<string, Vector<string>>` named `myMap`, then writing

  ```
  myMap[myKey].add(myValue);
  ```

  will add `myValue` to the `Vector` associated with the key `myKey`, creating a fresh new `Vector` if `myMap` doesn't already have anything associated with `myKey`. Use this to your advantage – you can *dramatically* reduce the amount of code you need to write by using this feature of the `Map` type!
- ***Be careful how you pass arguments to functions***. In this part of assignment, you're likely going to be passing large objects around between functions. Remember to pass around large objects either by reference (if you need to modify them) or `const` reference (if you don't) rather than by value.
- ***Letter position matters just as much as letter frequency.*** When computing word families, it's not enough to count the number of times a particular letter appears in a word; you also have to consider their positions. For example, `BEER` and `HERE` are in two different families even though they both have two E's in them. Make sure your representation of word families can encode this distinction.
- ***Don't worry about ties***. In splitting words apart into word families, you may find that there are several word families that have the same number of words in them. When that happens, you can break ties arbitrarily.
- ***Watch out for gaps in the dictionary.*** When the user specifies a word length, you will need to check that there are indeed words of that length in the dictionary. You might initially assume that if the requested word length is less than the length of the longest word in the dictionary, there must be some word of that length. Unfortunately, the dictionary contains a few "length gaps," lengths where there's no words of that length even though there are words of a longer length.
- ***Don't explicitly enumerate word families.*** If you are working with a word of length n, then there are 2^n possible word families for each letter. However, most of these families don't actually appear in English. For example, no words contain three consecutive U's, and no word matches the pattern `E-EE-EE--E`. Rather than explicitly generating every word family whenever the user enters a guess, see if you can generate word families only for words that actually appear in the word list.

- ***Consider making a "game structure."*** You may find yourself needing to pass a bunch of information around different functions, like the remaining words, the number of guesses, what's been guessed so far, etc. This might result functions that take in a *lot* of parameters. As an alternative, consider defining your own custom `struct` to hold all of the information that you need, then pass that `struct` around through your code. For example, if you find yourself always needing to pass around the number of remaining guesses and what the word looks like so far, consider making a `struct` with those values as fields, then passing that `struct` as a parameter through your functions. This dramatically reduces the amount of time you'll spend typing out parameter names.
- ***Don't forget to call our special functions!*** The functions `recordTurnInfo` and `recordGameEnd` don't do anything in the starter code, but they'll hook into our autograder after you submit the assignment. ***If you don't remember to call these functions, your code will fail a lot of our automated tests and your grade might be substantially lower than expected!***

Part Three: (Optional) Extensions

If you'd like to run wild with these assignments, go for it! Here are some suggestions.

- ***Word Ladders:*** Some pairs of words are quite "close" to one another in terms of word-ladder distance: the shortest ladder between them isn't that long. Other pairs of words are quite "far" from one another: the shortest ladder between them is quite long. What two words in English have the longest word ladder between them? How might you write a program to find that out?

  Consider looking at word ladders where at each step you either replace, insert, or delete a character somewhere in the string. This now allows you to link words of different lengths. For more on this topic, look up *Levenshtein distance*.
- ***Evil Hangman:*** The strategy outlined in this assignment is an example of a greedy algorithm. At each step in the game, the program chooses the family of words that keeps the most words remaining and lying around even if it's not the best way to ensure a victory. For example, if the user has a single guess left and there are two options available to the program, one where it reveals a letter and keeps two words around and one where it doesn't reveal the letter and drops the number of guesses down to zero, the program really should choose that second option because it forces a victory. Consider making this program more intelligent in how it plays – though do keep in mind that you need to keep it fast or otherwise the user will suspect something's up!

  Alternatively, imagine you knew you were playing against an Evil Hangman opponent. What's the best sequence of letters to punch in? And how many guesses will you need to win? For example, is it ever possible to win with eight guesses using five letters?

Submission Instructions

Before you submit this assignment, make sure that you're doing the following:

- ***Word Ladders:*** Are you calling our provided `recordLadderBetween` and `recordNoLadderBetween` functions after your algorithm finishes running? If not, make sure that you do, since we'll be using those functions to grade your assignment! It would be a shame if your program did everything right but then didn't play nice with our autograders.
- ***Evil Hangman:*** Are you calling our provided `recordTurnInfo` function at the ***start*** of each turn? Are you calling `recordGameEnd` at the end of the game?

Once you're sure you've done everything correctly, submit the `WordLadders.cpp` and `EvilHangman.cpp` files online at https://paperless.stanford.edu/. And that's it! You're done!

Good luck, and have fun!