Project 5: `Word Ladders`

Definition:

A word ladder is an ordered sequence of words $[w_0, w_1, \ldots, w_n]$, such that each word w_i in the sequence is obtained by changing a single character in w_{i-1}, with the requirement that each word in the sequence is a valid English word.

For example:

code -> come -> came -> game -> gate -> date -> data

is a valid word ladder from the word "code" to the word "data". This is not, however, the shortest transformation. The following ladder is two words shorter:

code -> cade -> cate -> date -> data

The Problem:

For this project you will write a program to print the shortest valid ladder, given a start_word and an end_word. To simplify the the problem, we will consider only 4-letter words. The output of your testing program might look something like this:

```
Finding shortest ladder for 4-letter English words.
Given two words, I will change the first into the second by replacing one character at a time.

Please enter the first word: code

Please endter the second word: data

The shortest ladder from code to data is:
code -> cade -> cate -> date -> data
Program ended with exit code: 0
```

```
Finding shortest ladder for 4-letter English words.
Given two words, I will change the first into the second by replacing one character at a time.

Please enter the first word: pip

Please endter the second word: pop

There is no valid transformation between pip and pop

Program ended with exit code: 0
```

Finding the Shortest Ladder:

Finding the shortest ladder is an example of the **shortest-path problem:** finding the shortest path from a start position to a goal. A wide variety of problems can be cast and solved as shortest-path problems: routing packets on the Internet, robot motion planning, comparing gene mutations, computing proximity in social networks…

One strategy to finding the shortest path is to use the **Breadth-First Search** algorithm (we looked at an example of BFS when we discussed generating all substrings up to a given size in Lecture 20). **BFS** can be used to find the shortest

path because, starting from an initial state, it first **considers all options that differ** from the initial state **by one**. Only after having considered all such options it moves to consider all states that differ from the initial state **by two**, then all states that differ from the initial **by three**, and so on, until it finds a solution. Because it considers all paths of length *k* before considering paths of length *k+1*, the first solution encountered is guaranteed to be the shortest.

A **BFS** for word ladders will first consider all the words that differ from the start_word by changing only one letter. If end_word is found among these, the solution is found and it is only one "step" away (the shortest ladder). If not, it will consider all ladders that are two steps away from the start_word, ones in which two characters have been changed, and so on.

The BFS algorithm is easily implemented using a **queue** to store partial ladders. All ladders being considered are placed on a queue and will be therefore stored in order of increasing length. Thus **all** shorter ladders will be dequeued before longer ones. The process starts by adding to the queue a ladder consisting of the start_word only. From there on the algorithm repeatedly dequeues a ladder and, if it is not a solution, it generates **all possible ladders** obtained by changing a single character to the last word of the ladder in consideration, appending the new word to the ladder and enqueuing all such generated new ladders.
Here is the **pseudocode** for such algorithm:

```
create an empty queue to store the partial ladders;
add a ladder containing only the start_word to the end of the queue;

while (the queue is not empty) {
    dequeue the first ladder from the queue;
    if (the last word in this ladder is the end_word) {
        return this ladder as the solution;
    }
    for (each word in the lexicon of English 4-letter words that differs
            by one letter from the last word in this ladder)
    {
        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 and mark it as used;
            add the new ladder to the end of the queue;
        }
    }
}
return that no word ladder exists;
```

There are a few things to consider before implementing this pseudocode:

1. The statement in the for loop needs to be broken down further, because it is actually checking two things: to consider ***every word that differs by one letter***, it must consider every character position in the word, and for each character it must try to replace it with every character in the alphabet, and use it only if it is a valid English word found in the lexicon. This can therefore be broken down into 2 nested for loops, the outer one loops for every character in the word (we are considering only 4-letter words), and the inner loop can substitute that character with every letter in the alphabet and check if it finds the so generated new word in the lexicon.
2. Each time a word is appended to the ladder, it must be marked as used to avoid loops, e.g.
   code -> come -> came -> game -> came -> come -> code …
   To avoid this you can maintain a list of used words, and never reuse a word that has already been added to a ladder.

Implementation:

Data structures:

You will need ***a queue*** to store the partial ladders generated during **BFS.**

A ladder itself is a sequence of words, and you also need to maintain a list of used words. Moreover, you will need to read in the lexicon of 4-letter English words from a file (you will find the input file on BlackBoard under Course Materials/ Project5). So other than the queue used to implement BFS you will also need:

1. Partial ladders
2. A list of used words
3. The lexicon
   For these you can use ***vectors*** or an other container you see fit.

I simplified the lexicon to include only the most common English 4-letter words (it is impressive how many obscure 4-letter words there are, this way the lexicon is smaller and the generated ladders will be more familiar).

For this project you will further familiarize yourself with the STL. In the STL, queue operations are named push() and pop(). Don't let this confuse you with a stack. As long as you declare an `std::queue<>` the FIFO order will be maintained.

To search the lexicon more efficiently, you can also use the STL algorithm **binary_search**, since the lexicon is sorted. You will store the lexicon as a vector, and binary_search will return true if it finds the word in the lexicon. The 3 parameters to binary_search will be:

1. **lexicon.begin():** an iterator to the first element in the lexicon (think of it as a pointer for now, and you are encouraged to read about STL iterators).
4. **lexicon.end():** an iterator to the last element in the lexicon.
5. **current_word**: the word you are searching for

Example:

```
#include <algorithm>
. . .
if(binary_search(lexicon.begin(), lexicon.end(), current_word))
```

References:

http://www.cplusplus.com/reference/stl/
https://www.geeksforgeeks.org/the-c-standard-template-library-stl/

Implementation Requirements:

You MUST implement this project as a class WordLadder (**`WordLadder.hpp`, `WordLadder.cpp`**). You are free to design the class as you wish (you may have all the public and private member functions you see fit). The only **required public methods** are the following:

```
/**
 @post reads lexicon words from the input file and stores them for
       later use
 **/
void WordLadder::readLexicon(string input_file)


/**
 @param start_word is the first word in the ladder
 @param end_word is the last word in the ladder
 @return a vector containing words s.t. the first word is start_word,
 the last word is end_word, and all words in between differ by
 only one character from the preceding and following words, in
 the shortest such transformation found in the current lexicon.
 If no transformation is found between start_word and end_word,
 returns an empty vector.
 **/
vector<string> WordLadder::findShortestLadder(string start_word, string
end_word)
```

Simplifications:

- All words in our lexicon have length 4. Your program will not find a ladder if input words have more or less than 4 letters, or if input words are not found in the lexicon. If you are up for it you can play with larger lexicons and variable size words.
- All words in our lexicon are lower case. You may assume that input words contain only lowercase alphabetic characters. If you are up for it, you may want to handle conversion to lowercase before running BFS so that your program is not case sensitive.
- You may assume no duplicates in the lexicon.
  You may assume *start_word* and *end_word* are non-empty and are not the same.

The data:

The input file contains 4-letter English words (common ones, it is not an exhaustive list), one word per line. You are welcome to experiment with different lexicons. Note that the ladders you will find depend on the input data. If you work with larger lexicons you may find more and/or shorter ladders.

Testing:

You must write a main function to test your class, but you will not submit it. Here are some ladders produced with the input file I provided, you can use these for testing that your program is correct.

```
dare -> care -> core -> cord -> lord -> load -> road -> read -> real

this -> thin -> than -> that

chip -> chop

word -> ford -> fort -> port -> post -> lost

jest -> just -> must -> most -> post -> past

card -> cord -> lord -> load -> road

dare -> care -> cade -> made -> make
```

Submission:

You must **submit** **`WordLadder.hpp, WordLadder.cpp`** (**2 files**)

Your project must be submitted on Gradescope. The due date is Friday May 10 by 6pm. No late submissions will be accepted.

Warning: this project may take a little longer to build on Gradescope.

Have Fun!!!!!