# Lab Assignment #3

**Goals:** To gain experience working with more sophisticated algorithms that use stacks and queues, while continuing to practice the use of common C++ structures such as classes, pointers, linked lists, dynamic arrays, etc. To work on an interesting, open-ended problem while practicing one's programming skills. To experience the benefits of working in a pair with another student from the class (if desired), i.e., *pair programming*.

## Due Dates:

**Part 1 –** Finding a partner **– Sat, Mar 03 @ 10pm** (one student from each team of 2 needs to email me who is on the team)
**Part 2 –** Full submission **– Fri, Mar 09 @ 10pm** (all code and project write-up(s) to be submitted at that time)

**Warning:** If the assignment seems complex at a glance, tackle it, as always, one step at a time. Specific intermediate steps are outlined below to give you a basic roadmap for how to approach the problem incrementally, thus achieving intermediate results along the way, which can be demonstrated and tested successfully, giving you credit for completing each such step.
Be sure to start as early as possible, and pace yourself. If you only start near the end of the period, your project will almost certainly remain incomplete. (Even I cannot do it well in 1-2 days, and neither should you expect such feats of yourselves.)

**Submission:** A turn-in link, called "Assignment Submission", is available on the course web site.
The submission mechanism we will use (UW-Catalyst DropBox tool) is the same as for all labs and written homeworks thus far.

**Mechanics:** Unless you choose otherwise (in which case you need to inform me still), you are encouraged to work in a team with another student of your choice in the class for the coding portion of this project assignment. (A forum in the discussion board is now open specifically to facilitate the process of finding partners.) This practice is called *pair programming* in the software industry, and is known to have many benefits, including increased motivation, better learning from one another, and less time lost on unproductive potential solutions or problems.
You are free to discuss the assignment with classmates from other teams too, but without sharing actual code or written descriptions.
Only one student from each team must turn in the jointly written code.
For the project description portion of the assignment, each student must turn in their own individual write-up, relating to the joint project work.

## Product Requirements:

For this lab assignment, we will implement and play a popular word game, where all words will be taken from an English language dictionary. The purpose of the game – which you need to teach your computer to "play" – is to find how to transform a given English word into another given word, by applying a set of simple rules.

The rules are as follows. Starting from a given (source) word in English, on each step you may change only one character (of your choice) in it such that the result of that transformation step is another valid English word. Then repeat. The goal is to find a sequence of such steps that ultimately leads the destination word, ideally using the fewest number of steps possible.

**Example:** Given the words "pear" and "lens", some possible transformations starting from "pear" and ending with "lens" are these:
**pear => peat => pest => lest => lent => lens** (transition length = 5)
**… => pelt => belt => bent => lent => lens** (transition length = 6)
**… => pent => lent => lens** (transition length = 4)
**…** => **peas** => **pens** => **lens** (transition length = 3)

## Specific Tasks:

The problem can be broken down into multiple simpler, constituent sub-tasks, briefly stated as follows:

**(A)** Implement RadixSort (see the Carrano textbook, chapter 9.2, for a description of the algorithm and some examples on how it works) using queues. The goal is to be able to use RadixSort to order a list of English words lexicographically.
Your program **`WordGame.exe`** must allow testing of this feature via the following command-line argument interface:

```
WordGame.exe -sort_lex <file containing a list of words, separated by whitespaces>
```

The expected output is a lexicographically-ordered list, in which each word is on a separate line, with no other symbols in the output.
For example, if the input file contains:

```
ball soccer goalkeeper penalty
```

the output of this portion of your program must be

```
ball
goalkeeper
penalty
```

```
soccer
```

**(B)** Implement method(s) for removing duplicates from a lexicographically ordered list of English words.
Your program **WordGame.exe** must allow testing of this feature via the following command-line argument interface:

**WordGame.exe –remove_dup <*file containing lexicographically sorted list of words, separated by whitespaces*>**

The expected output is, once again, a lexicographically-ordered list without any duplicates, where each word is on a separate line, with no other symbols in the output.
For example, if the input file contains

```
ball goalkeeper penalty penalty score soccer soccer
```

the output of this portion of your program must be

```
ball
goalkeeper
penalty
score
soccer
```

**(C)** Implement method(s) for finding all valid English words (from a given dictionary) that are immediate transformations of a given word X. That is, find all valid English words which differ from the given word X by only one letter (anywhere in the word).
For example, `pear` has the following "similar" words that differ by only one letter from it: `bear`, `rear`, `peer`, `peat`, `peak`, etc.
Your program **WordGame.exe** must allow testing of this feature via the following command-line argument interface:

**WordGame.exe –gen_next_words <*current word*> <*lexicographically-sorted dictionary file of words, separated by whitespace*>**

The expected output is for each word in the generated set to be displayed on a separate line, with no other symbols in the output.
For example,

**WordGame.exe –gen_next_words pear <*lexicographically-sorted dictionary file of words, separated by whitespace*>**

would yield

```
bear
rear
peer
peat
peak
```

etc.

**(D)** Implement method(s) for finding one solution to the problem stated at the beginning: a sequence of transformations from a given source word into a given destination word, based on an available English dictionary. Note that the solution need not be the shortest possible set of transformations – any solution to the problem would satisfy this sub-task.
Your program **WordGame.exe** must allow testing of this feature via the following command-line argument interface:

**WordGame.exe –solve <*starting word*> <*destination word*> <*lexicographically-sorted dictionary file of words, separated by whitespace*>**

The expected output is that each word in the transformation set must be on a separate line; the first word must be the starting word, and the final word must be the destination word, with no other symbols. One useful addition here is that after the last (destination) word, on a separate line, you must display the number of transformations it took.
For example, if the transformation is between the words "pear" and "lens", then one solution (also listed above) could look like this:

```
pear
peat
pent
lent
lens
Transformation length = 4
```

**(E) [Extra Credit]** Implement the optimal (shortest) solution in the number of transformations from the given source word to the given destination word.
If you choose to implement this extra credit portion, your program **WordGame.exe** must allow testing of this feature via the following command-line argument interface:

**WordGame.exe –best_solve <*starting word*> <*destination word*> <*lexicographically-sorted dictionary file of words, separated by whitespace*>**

The expected output is in its format the same as for part **(D)**, above – each word in the transformation set must be on a separate line; the first word must be the starting word, and the final word must be the destination word, with no other symbols. Also, after the last (destination) word, on a separate line, you must display the number of transformations it took for this shortest solution.

## Assumptions and Simplifications:

It is okay to assume without verifying explicitly in your code the following:

- The starting (source) and ending (destination) words will be of the same length, and both valid English words from the supplied dictionary.
- You may ignore capitalization of words; assume that all words are in lower case.

## Specific Programming-Related Requirements:

- Place all your functions and necessary objects/variables into appropriate classes.
- Functions for reading in one word at a time from a given input text file are supplied by me, as before. You need to use them in your code *without modification*.
- Define and use a stack for at least some aspects of the problem (even if you also choose to use recursion). It is very appropriate for the search algorithm – finding the sequence of transformation steps; recording the set of steps taken thus far.
- Define and use a queue (at least) for the RadixSort implementation of lexicographically ordering a set of words.
- You may reuse existing definitions of stacks and queues, but if you do so, you are responsible for understanding and properly using them, as well as for giving credit to the original author(s), even if you modified some of that code.
- Do not hardcode *any words* (i.e., potential input or output text) in your C++ code.

## Tips:

- Start by understanding the domain well. Experiment on paper with sample words and transformations between them.
- Consider what classes you may need to have in your program in order to model the necessary data and incorporate the needed functions that operate on the data. Which functions will naturally go into which classes? What interfaces (i.e., public methods) would each class have to expose to its clients, and what methods might be better left private?
- Consider what data structures (linked lists, arrays (static or dynamic), stacks, queues, etc.) will be best to use for storing in memory the dictionary contents for your program's processing. After you have thought about this for some time (i.e., played with it, experimented on paper, etc.), please feel free to read the next sentence – there I am giving you a potentially useful hint, but it'll be much better for you to have thought about the problem deeply by yourself first.
  You may find it valuable to organize your dictionary as arrays of linked lists; one array (of 26 elements) representing the set of words in the dictionary that start with a given letter (A-Z); another array (of 26 elements) representing the set of words in that dictionary as their second letter have a given letter (A-Z) from the alphabet; etc. – as many arrays as your needed word length is. For example, `Array_2['p']` may be a linked list of all words that have 'p' as their second letter.
- One natural simplification, depending on how you choose to implement your algorithm(s), may be to cut down the dictionary to only those words that have the same length as your source and destination words. This will reduce the size of your dictionary – and speed up your program possibly by a factor of 5-10, which is not insignificant, given that English is a rich language with many words.
- Relevant readings are all those in chapters 6, 7, and 9.2 – but not limited to them. Be sure to also look at sample problems given in the textbook, as some of them contain valuable hints that would help you with the search algorithms. For example, ideas for how to search through a maze – a similar activity to what you'll be doing here – are contained after one of the chapters.
- You are encouraged to share sample input files among all students; use the discussion board for this – in a special forum for this purpose (under "Lab #3 discussion").
- Feel free to use public domain dictionary files as sample input for your initial testing.
- **Ask questions**; I can answer; talk to your fellow classmates about what questions they have. Certain parts of the product description may be vague to begin with (which is quite common in initial product descriptions, even after a round of conversations), and in need of clarification before you have a firm grasp on what it is exactly that needs to be built.
- We will discuss these and many other issues in our next class meeting(s), but you need not – in fact, should not – wait until then in order to start.

## Evaluation:

In the evaluation of your project work, I will be looking to see not only that your code compiles and runs correctly with my test cases, but also that you have an associated write-up, in which you have:
(a) described your designs clearly, including classes and their interfaces (public methods), and justified those designs over alternatives you have considered. This includes justifying briefly why you chose particular data structures for the various situations;
(b) described clearly in English and/or in pseudo-code (but not in C++ code!) the algorithms you have employed for the various sub-tasks **(A)-(E)**, described above;
(c) created a reasonable (even if incomplete) set of test cases to test your program for functional correctness as well as for robustness against various types of inputs;
(d) described clearly and concisely what you have been able to accomplish of the required tasks, and what not;
(e) listed the bugs you are aware of in your code (note: every code has bugs, and that's okay so long as they are not critical, preventing the main functionality from working);
(f) described what, if any, additional (unspecified here) functionality you were able to build;

(g) shown a snapshot of your program running on a sample input, and producing its output;
(h) listed the number of hours it took you to work on this project, broken down by hours before coding (i.e., designs, clarifying requirements, etc.) and hours for coding and testing;
(i) organized and presented your write-up well.

## Deliverables:

I will expect to see the following deliverables by the due date:
(1) A write-up addressing all the questions from the Evaluation section above – this should be clear and concise; extra length is not a virtue. Use a common document format, to make sure that I have the software to open it and read your write-up. Name your write-up file in a way that will be easy for me to identify (e.g., *YourLastName*`-css342-lab3-writeup.doc`, where *YourLastName* is your last name).
(2) Your source code – including only your `*.cpp` (C++ source) and `*.h` (C++ header) files. No executables, no object files, no project files are necessary or desired.
(3) Your test data files, if any. If you have those, include also a description for how I may use them (e.g., "`Test-data-use-description.txt`).