CSE250, Spring 2014 **Assignment 4** **Due Fri. Feb. 28**

This assignment is for **both** hardcopy and online submission, like with Assignment 3. It is more weighted toward the programming part, however.

(1) Let $f(n) = n(\log n)^2$ and $g(n) = 2^n$. Here logs are to base 2, and it doesn't matter whether you leave $(\log n)^2$ as a real number or round it to an integer. Use L'Hôpital's Rule to show that $f = o(g)$. (9 pts.)

(2) Now suppose A is a program with running time $45n(\log n)^2$ on inputs of size n, and B has running time $3n^2 - n$. Find the "tradeoff point," namely the minimum integer n_0 such that for all inputs of size n with $n \geq n_0$, algorithm A is faster than algorithm B. Trial-and-error with a calculator is fine, but there is also an advanced option: use Newton's Method to find the greatest zero of the difference of the running times. You may use the function objects in `RealFn.h` and revise the example client `Newton.cpp` to carry out this calculation. It is set up as the answer key for a similar problem in another year, and has explanations in comments and output. (15 pts.)

(3) Write on paper a function **ed1** that tests whether two strings have an *edit distance* of *exactly* 1. This is more general than `hd1` or `xd1` from Assignment 3, in that it allows the shorter of two strings to be obtained by deleting any character of the longer one, not just at the end. For example, `ed1(raid,rabid)` should return `true` while the others return `false`.

A formal definition used in DNA sequencing etc. is that the general *edit distance* between two strings x and y is the minimum number of (a) one-char. changes, (b) one-char. insertions, and (c) one-char. deletions needed to change x to y (or vice-versa—the metric is symmetric). The changes can optionally be restricted to a legal subset of strings—for example legal English words—and this is when operations (b) and (c) might be needed in the same sequence of steps. Of course if the distance is just 1 then only one of the three operations can be used, and whether it's (b) or (c) depends on which you call the string that's one character longer.

Write your `ed1` in legal C++ syntax both in hardcopy and as part of the code below. You must also include one-or-more loop-invariant comment(s) that talk about characters in x and y that have already been examined. (15 + 9 = 24 pts., for 48 total in hardcopy)

(4) *For online submission only*, write a software system that includes your `PeekDeque` class from Assignment 3, `ed1` as a *free-floating function* (not a method of any class) that will go in the same file as `main`, and a `main` that does the following:

- Read some words from a file named `words.txt` into the data structure, pushing at the rear.
- Use the peeking feature to test whether every two consecutive words have edit distance 1, making them a legal chain.

You may use either `operator>>` or `getline` to read the words for now—that is, you may assume for now that `words.txt` has a separate word on each line with nothing else. For an example of a legal chain, suppose `words.txt` is:

```
a
at
ate
rate
```

```
irate
pirate
prate
pate
pare
par
```

Then your program should call it a legal chain. Whereas if the 7th word were `grate` instead of `prate`, that would be both a deletion and a letter-change, so the distance would not be 1 and you should return false overall. You should also not care yet whether a word is legal—so if `words.txt` had `piate` instead of `prate`, the whole thing would still be OK.

As a good option, you may make the test for legal chain a separate function above `main` in the same file. Since C++ uses linear declaration order, it should go after `ed1` which it will need to call, and before `main` which calls it. The function will lake a pointer to a `PeekDeque` object as its (principal) argument. Whether you code the chain test here or in `main`, you will find it *de rigeur* to add to your `PeekDeque` class the functionality to re-set the peek-index to the beginning, and a test for its being at the end. (You are welcome to add other functionality for convenience.)

Also, in a comment below `main`, answer the following report question: Wouldn't it be nice to allow having 2 or more index-peekers at once, say one starting at the front and the other starting at the rear? What would you have to do to your class to provide such functionality? Would it help if the peek-index could become an *object*, one that could be public?

The last requirement is that your library class code must be in separate files `PeekDeque2NNN.h` and `PeekDeque2NNN.cpp`, with `main` and the other function(s) in `PeekClientNNN.cpp`, and that you use a *named makefile* called `PeekClientNNN.make` to compile it via

```
make -f PeekClientNNN.make
```

As usual `NNN` stands for your initials. We encourage you to do this only at the end—that is, get your whole code working in a single file `PeekDeque2NNN.cpp` first, then separate into the multiple files. This does involve coding method bodies outside the class, as typified by:

```
bool StringDeque::empty() const { return frontItem == rearSpace; }

void PeekDeque::moveRearward() { peekIndex++; }
```

These can all go into the same file—that is, left as the only remaining content of `PeekDeque2NNN.cpp` file—after copying the class structure and declarations to `PeekDeque2NNN.h`, moving the client code to `PeekClientNNN.cpp`, and editing to add the class prefixes with `::` (it is AOK to outdent all the way, and while the bodies technically don't need to have the same order as the declarations, that is good for readability). Use the standard include guards

```
#ifndef PEEK_DEQUE__H
#define PEEK_DEQUE__H

    {code of your .h file goes here}

#endif
```

in `PeekDeque2NNN.h`, and in the client file you should have just `include PeekDeque2NNN.h`, *without* including the `.cpp` file too. These will be covered in lecture and in labs next week; but if it gives you troubles, you may "bail out" by putting everything into `PeekClientNNN.cpp` as it is just 9 pts. for the `.h,.cpp` structure and 3 pts. for the named make-file. Overall points are 36 for the code, 12 for the report question and overall commenting/readability, and 12 for the separate compilation, for 60 total on the programming part and 108 total on the assignment.