CS106X
Autumn 2014

October 1, 2014

Assignment 2: ADT Client Applications

Inspiration credit goes out to Mike Cleron from Google (random sentence generator) and Owen Astrachan from Duke University (word ladder).

Assignment handout authors: Cynthia Lee, Marty Stepp, Julie Zelenski and Jerry Cain.

Now that you've ingested the CS106 container classes, it's time to put these objects to use. In your role as client, the low-level details have already been dealt with and locked away as top secret so you can focus your attention on solving more interesting problems. Having a library of well-designed and debugged classes vastly extends the range of tasks you can easily take on. Your next assignment has you write three short client programs that heavily leverage the standard classes to do nifty things. The tasks may sound a little daunting at first, but given the power tools in your arsenal, each requires less than a hundred lines of code. Let's hear it for abstraction!

The assignment has several purposes:

1. To more fully explore the idea of using objects.
2. To stress the notion of abstraction as a mechanism for managing data and providing functionality without revealing the internal representation.
3. To become more familiar with C++ class templates.
4. To gain practice with classic data structures such as the queue, vector, set, lexicon, and map.

Due: Wednesday, October 8th at 5:00 p.m.

PROJECT I: Word Ladders [by Julie Zelenski]

Leveraging the vector, queue, and lexicon abstractions, you'll find yourself well equipped to write a program to build word ladders. A word ladder is a connection from one word to another formed by changing one letter 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 "code" to "data".

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

You will ask the user to enter a start and a destination word and then your program is to find a word ladder between them if one exists. By using an algorithm known as breadth-first search, you are guaranteed to find the shortest such sequence. The user can continue to request other word ladders until they are done.

Here is some sample output of the word ladder program:

```
Welcome to the CS106 word ladder application!
Please give me two English words, and I will change the
first into the second by changing one letter at a time.

Please enter the source word [return to quit]: work
Please enter the destination word [return to quit]: play
Found ladder: work fork form foam flam flay play

Please enter the source word [return to quit]: sleep
Please enter the destination word [return to quit]: awake
Found ladder: sleep sheep sheen shewn shawn sharn share sware aware awake

Please enter the source word [return to quit]: angel
Please enter the destination word [return to quit]: devil
Found ladder: angel anger agger egger eager lager leger lever level devel devil
```

Implementation Overview

Finding a word ladder is a specific instance of a *shortest path* problem, where the challenge is to find the shortest path from a start position to a target one. Shortest path problems come up in a variety of situations—packet routing, robot motion planning, social networks, gene mutation, and travel. One approach for finding a shortest path uses a classic algorithm known as *breadth-first search*. A breadth-first search stretches outward from the start in a radial fashion until it hits the goal. For word ladder, this means first examining those ladders that represent "one hop" (i.e. one changed letter) from the start. If any of these reach the destination, we're done. If not, the search now examines all ladders that add one more hop (i.e. two changed letters). By expanding the search at each step, all one-hop ladders are examined before two-hops, and three-hop ladders only considered if none of the one-hop nor two-hop ladders worked out, thus the algorithm is guaranteed to find the shortest successful ladder.

Breadth-first is typically implemented using a queue. The queue is used to store partial ladders that represent remaining possibilities worthy of exploring. 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. Due to the FIFO nature of the queue, ladders will be dequeued in order of increasing length. The algorithm operates by dequeueing the front ladder from the queue and determining if it reaches the goal. If it does, you have a complete ladder, and it is the shortest. If not, you take that partial ladder and extend it to reach words that are one more hop away, and enqueue those extended ladders onto the queue to be examined later. If you exhaust the queue of possibilities without ever finding a word ladder, you can assume no such ladder exists.

Let's make the algorithm a bit more concrete with some pseudo-code:

```
create initial ladder (just start word) and enqueue it
   while queue is not empty
      dequeue first ladder from queue (this is shortest partial ladder)
      if top word of this ladder is the destination word
         return completed ladder
      else for each word in lexicon that differs by one char from top word
           and has not already been used in some other ladder
               create copy of partial ladder
               extend this ladder by pushing new word on top
               enqueue this ladder at end of queue
```

A few of these tasks deserve a bit more explanation. You will need to find all the words that differ by one letter from a given word. A simple loop can change the first letter to each of the other letters in the alphabet and ask the lexicon if that transformation results in a valid word. Repeat that for each letter position in the given word and you will have discovered all the words that are one letter away.

Another issue that is a bit subtle is the restriction that you not re-use words that have been included in a previous ladder. This is an optimization that avoids exploring redundant paths. For example, if you have previously tried the ladder **`cat`→`cot`→`cog`** and are now processing **`cat`→`cot`→`con`**, you would find the word **`cog`** one letter away from **`con`**, so looks like a potential candidate to extend this ladder. However, **`cog`** has already been reached in an earlier (and thus shorter) ladder, and there is no point in re-considering it in a longer ladder. The simplest way to enforce this is to keep track of the words that have been used in any ladder (using yet another lexicon!) and ignore those words when they come up again. This technique is also necessary to avoid getting trapped in an infinite loop building a circular ladder such as **`cat`→`cot`→`cog`→`bog`→`bat`→`cat`**.

Since you need linear access to all of the items in a word ladder when time comes to print it, it makes sense to model a word ladder using a **`Vector<string>`**. And remember that you can make a copy of a **`Vector<string>`** by just assigning it to be equal to another via traditional assignment (i.e. **`Vector<string> wordLadderClone = wordLadder`**).

A few implementation hints

Again, it's all about leveraging the class libraries—you'll find your job is just to coordinate the activities of various objects to do the search.

- A **`Queue`** object is a FIFO collection that is just what's needed to track those partial ladders. The ladders are enqueued (and thus dequeued) in order of length so as to find the shortest option first.
- You should make sure that the start and end words as well as all the connectors are legitimate English words.

Word ladder task breakdown

This program requires just over a page of code, but it still benefits from a step-by-step development plan to keep things moving along.

- *Task 1— Try out the demo program*. Play with the demo just for fun and to see how it works from a user's perspective.
- *Task 2— Get familiar with our provided classes*. You've seen **`Vector`** and **`Queue`** in lecture, but **`Lexicon`** is new to you. The reader outlines everything you need to know about the **`Lexicon`**, so make sure you skim over Chapter 5 and the part about how to use the **`Lexicon`** class (it's really very easy.)
- *Task 3— Conceptualize algorithm and design your data structure*. Be sure you understand the breadth-first algorithm and what the various data types you will be using. Note that since the items in the **`Queue`** are **`Vector`**s, you have a nested template here— be careful!
- *Task 4— Dictionary handling*. Set up a **`Lexicon`** object with the large dictionary read from our data file. Write a function that will iteratively construct strings that are one letter different from a given word and run them by the dictionary to determine which strings are words. Why not add some testing code that lets the user enter a word and prints a list of all words that are one letter different so you can verify this is working?
- *Task 5— Implement breadth-first search*. Now you're ready for the meaty part. The code is not long, but it is dense and all those templates will conspire to trip you up. We recommend writing some test code to set up a small dictionary (with just ten or so words) to make it easier for you to test and trace your algorithm while you are in development. Do your stress tests using the large dictionary only after you know it works in the small test environment.

Word ladder program summary

Here is summary of how we expect your program to run:

1. Print a short welcome message.
2. Prompt the user for a start and an end word. If either word is blank, the program should exit immediately. You should not make any assumptions about the length of the words and ignore case when searching for a ladder.
3. If you found a ladder, print out *"Found ladder:"* followed by each word separated by a single space. If you did not find a latter, print *"No ladder found."*
4. Repeat from Step 2 until the user enters a blank word.

The sample solution available on the course website implements the program as described above. You should aim to have your solution exactly match the output of the sample solution.

PROJECT II: Two Flavors of Random Sentence Generator [by Cynthia Lee]

Project II has two parts, each representing one philosophical branch of the field of artificial intelligence, specifically of computational linguistics: (1) Rules-based systems, and (2) Statistical inference systems.

Rules-based systems: In the early days of the field, scholars worked to enumerate all the rules of grammar of human language in big hierarchical list. Part of this effort was the "context-free grammar"[1]. This rule-based approach to artificially-intelligent processing of human language relied heavily on human effort to research, formulate, and organize all the rules.

Statistical inference systems: In the late 1990's, a new movement developed that used statistics to *infer* (guess) the rules based on mounds of evidence, rather than requiring humans to determine and input the rules. Skeptics doubted this random, messy, hands-off approach could ever match the quality of the expertly-curated human approach. But, led by Google, the statistical approach won out, and we've essentially never looked back. Now everything from spam filters to Amazon shopping suggestions, to self-driving cars, to your Pandora playlist, to your Facebook feed, are generated based on rules and preferences inferred from mounds of empirical evidence of your previous interactions with the technologies.

Today, it's hard to come up with examples of the rule-based approach that would be familiar to you, because the statistical approach has so thoroughly dominated. (Aside: this is why you all should be really excited to take CS109 with me in the spring! We'll learn the mathematical tools behind these technologies.)

In this assignment, you will compare two simple applications that exemplify the spirit of the two different approaches. Experiment and draw your own conclusions about the relative merits of each!

Part I: Grammar Rules Random Sentence Generator [by Julie Zelenski]

Over the past two or three decades, computers have revolutionized student life. In addition to providing entertainment and distraction, computers also have also facilitated all sorts of student work. One important area of student labor that has been painfully neglected is the task of filling up space in papers, Ph.D. dissertations, extension requests, etc. with important sounding and somewhat grammatically correct random sequences. Neglected, that is, until now.

The Random Sentence Generator is a marvelous piece of technology that creates random sentences from a structure known as a **context-free grammar**. A grammar is a construct describing the various combinations of words that can be used to form valid sentences. There are profoundly useful grammars available to generate extension requests, generic Star Trek plots, your average James Bond movie, "Dear John" letters, and more. You can

[1] Fun fact: context-free grammars were invented by Noam Chomsky, now perhaps better known as the politically radical author of *Manufacturing Consent*.

even create your own grammar! Fun for the whole family! Let's show you the value of this practical and wonderful tool:

- Tactic #1: Wear down the TA's patience.

  I need an extension because I had to go to an alligator wrestling meet, and then, just when my mojo was getting back on its feet, I just didn't feel like working, and, well I'm a little embarrassed about this, but I had to practice for the Winter Olympics, and on top of that my roommate ate my disk, and right about then well, it's all a haze, and then my dorm burned down, and just then I had tons of midterms and tons of papers, and right about then I lost a lot of money on the four-square semi-finals, oh, and then I had recurring dreams about my notes, and just then I forgot how to write, and right about then my dog ate my dreams, and just then I had to practice for an intramural monster truck meet, oh, and then the bookstore was out of erasers, and on top of that my roommate ate my sense of purpose, and then get this, the programming language was inadequately abstract.

- Tactic #2: Plead innocence.

  I need an extension because I forgot it would require work and then I didn't know I was in this class.

- Tactic #3: Honesty.

  I need an extension because I just didn't feel like working.

What is a grammar?

A grammar is a set of rules for some language, be it English, Java, C++, or something you just invent for fun. ☺ If you continue to study computer science, you will learn much more about languages and grammars in a formal sense. For now, we will introduce to you a particular kind of grammar called a context-free grammar (**CFG**).

Here is an example of a simple CFG for generating poems:

```
<start>
1
The <object> <verb> tonight.

<object>
3
waves
big yellow flowers
slugs

<verb>
3
sigh <adverb>
portend like <object>
die <adverb>

<adverb>
2
```

```
warily
grumpily
```

According to this grammar, two syntactically valid poems are **"The big yellow flowers sigh warily tonight."** and **"The slugs portend like waves tonight."** Essentially, the strings in brackets (<>) are variables that expand according to the rules in the grammar.

More precisely, each string in brackets is known as a **nonterminal**. A nonterminal is a placeholder that will expand to another sequence of words when generating a poem. In contrast, a **terminal** is a normal word that is not changed to anything else when expanding the grammar. The name terminal is supposed to conjure up the image that it's something of a dead end—that no further expansion is possible.

A **definition** consists of a nonterminal and a list of possible **productions** (or **expansions**). There will always be at least one and potentially several productions for each nonterminal. A production is just a text string of words, some of which themselves may be non-terminals. A production can be the empty string, which makes it possible for a nonterminal to evaporate into nothingness. An entire definition is summarized within a grammar text file as:

```
<verb>                  ⇐ the first line names the nonterminal and is delimited by < and >
3                       ⇐ the second line is always the number of possible expansions
sigh <adverb>           ⇐ the third line is the first possible expansion
portend like <object>   ⇐      followed by another expansion if there is a second one
die <adverb>            ⇐          followed by another expansion if there is a third one, etc
                        ⇐ for readability, there's a blank line after each definition, including the last one
```

You always begin random sentence generation with the single non-terminal **`<start>`** as the working string, and iteratively search for the first nonterminal[2] and replace it with any one of its possible expansions (which may and often will include its own nonterminals). Repeat the process over and over until all nonterminals are gone.

```
<start>
The <object> <verb> tonight.                    // expand <start>
The big yellow flowers <verb> tonight.          // expand <object>
The big yellow flowers sigh <adverb> tonight.   // expand <verb>
The big yellow flowers sigh warily tonight.     // expand <adverb>
```

Since we are choosing productions at random, a second generation would almost certainly produce a different sentence.

Your program should repeatedly prompt the user for a grammar file (understood to be the **`grammars`** subdirectory), read in the grammar, and generate three random sentences

[2] This problem could also be solved using recursion, but we ask that you don't solve this recursively, but instead solve it using iteration.

separated by a blank line. Only when the user hits return without actually typing in anything should you end the program. Using the sample application as a guide, you are to make all design and implementation decisions.

Some simplifying assumptions:

- You may assume that the grammar files are properly formatted, and that the grammars themselves are well formed. All grammars will include a **"<start>"** nonterminal, and all nonterminals will expand to one or more definitions (which themselves may and often will include other nonterminals).
- You needn't worry about word wrap as you generate and print out über-long sentences.

Part II: Statistical Random Sentence Generator [by Marty Stepp]

In the second part of this assignment, you will write a program that reads an input file and uses it to build a large data structure of word groups called "*N*-grams" as a basis for randomly generating new text that sounds like it came from the same author as that file. You will use the Map and Vector collections from Chapter 5.

At right is an example **log of interaction** between your program and the user (console input underlined).

But what, you may ask, is an *N*-gram?

```
Welcome to CS 106B Random Writer ('N-Grams').
This program makes random text based on a
document.
Give me an input file and an 'N' value for groups
of words, and I'll create random text for you.

Input file? hamlet.txt
Value of N? 3

# of random words to generate (0 to quit)? 40
... chapel. Ham. Do not believe his tenders, as
you go to this fellow. Whose grave's this,
sirrah? Clown. Mine, sir. [Sings] O, a pit of
clay for to the King that's dead. Mar. Thou art a
scholar; speak to it. ...

# of random words to generate (0 to quit)? 20
... a foul disease, To keep itself from noyance;
but much more handsome than fine. One speech in't
I chiefly lov'd. ...

# of random words to generate (0 to quit)? 0
Exiting.
```

The "Infinite Monkey Theorem" states that an infinite number of monkeys typing random keys forever would eventually produce the works of William Shakespeare. That's silly, but could a monkey randomly produce a *new work* that "sounded like" Shakespeare's works, with similar vocabulary, wording, punctuation, etc.? What if we chose *words* at random, instead of individual letters? Suppose that rather than each word having an equal probability of being chosen, we weighted the probability based on how often that word appeared in Shakespeare's works?

Picking random words would likely produce gibberish, but let's look at *chains of two words* in a row. For example, perhaps Shakespeare uses the word "to" occurs 10 times total, and 7 of those occurrences are followed by "be", 1 time by "go", and 2 times by "eat". We can use those ratios when choosing the next word. If the last word we chose is "to", randomly choose "be" next 7/10 of the time, "go" 1/10 of the time, and "eat" 2/10. We never choose any other word to follow "to". We call a chain of two words like this, such as "to be", a *2-gram*.

```
+--------------------+
| Chose "to".        |----> choose "be"    (7/10 chance)
| Next random word?  |----> choose "go"    (1/10 chance)
+--------------------+----> choose "eat"   (2/10 chance)
```

Go, get you have seen, and now he makes as itself? *(2-gram)*

A sentence of 2-grams isn't great, but look at chains of 3 words (*3-grams*). If we chose the words "to be", what word should follow? If we had a collection of all sequences of 3 words-in-a-row with probabilities, we could make a weighted random choice. If Shakespeare uses "to be" 22 times and follows them with "or" 5 times, "in" 3 times, "with" 10 times, and "alone" 4 times, we could use these weights to randomly choose the next word. So now the algorithm would pick the third word based on the first two, and the fourth based on the (second+third), and so on.

```
+--------------------+----> choose "or"    ( 5/22 chance)
| Chose {"to", "be"}. |----> choose "in"    ( 3/22 chance)
| Next random word?  |----> choose "with"  (10/22 chance)
+--------------------+----> choose "alone" ( 4/22 chance)
```

One woe doth tread upon another's heel, so fast they follow. (3-gram)

You can generalize the idea from 2-grams to *N*-grams for any integer *N*. If you make a collection of all groups of *N* words along with their possible following words, you can use this to select an *N*+1'th word given the preceding *N* words. The higher *N* level you use, the more similar the new random text will be to the original data source. Here is a random sentence generated from 5-grams of *Hamlet*, which is starting to sound a lot like the original:

I cannot live to hear the news from England, But I do prophesy th' election lights on Fortinbras. (5-gram)

Each particular piece of text randomly generated in this way is also called a *Markov chain*. Markov chains are very useful in computer science and elsewhere, such as artificial intelligence, machine learning, economics, and statistics.

Algorithm Step 1: Building Map of N-Grams

In this program, you will read the input file one word at a time and build a particular compound collection, a **map** from prefixes to suffixes. If you are building 3-grams, that is, *N*-grams for *N*=3, then your code should examine sequences of 2 words and look at what third word follows those two. For later lookup, your map should be built so that it connects a collection of *N*-1 words with another collection of all possible suffixes; that is, all possible *N*'th words that follow the previous *N*-1 words in the original text. For example if you are computing *N*-grams for *N*=3 and the pair of words "to be" is followed by "or" twice and "just" once, your collection should map the key {to, be} to the value {or, just, or}. The following figure illustrates the map you should build from the file:

When reading the input file, the idea is to keep a window of *N*-1 words at all times, and as you read each word from the file, discard the first word from your window and append the new word. The following figure shows the file being read and the map being built over time as each of the first few words is read to make 3-grams:

`to be or not to be just ...`	`map    = {}`
`     ^`	`window = {to, be}`
`to be or not to be just ...`	`map    = {{to, be}     : {or}}`
`        ^`	`window = {be, or}`
`to be or not to be just ...`	`map    = {{to, be}     : {or},`
`            ^`	`          {be, or}     : {not}}`
	`window = {or, not}`
`to be or not to be just ...`	`map    = {{to, be}     : {or},`
`               ^`	`          {be, or}     : {not},`
	`          {or, not}    : {to}}`
	`window = {not, to}`
`to be or not to be just ...`	`map    = {{to, be}     : {or},`
`                  ^`	`          {be, or}     : {not},`
	`          {or, not}    : {to},`
	`          {not, to}    : {be}}`
	`window = {to, be}`
`to be or not to be just ...`	`map    = {{to, be}     : {or, just},`
`                       ^`	`          {be, or}     : {not},`
	`          {or, not}    : {to},`
	`          {not, to}    : {be}}`
	`window = {be, just}`
`...`	`...`
	`map    = {{to, be}     : {or, just, or},`
	`          {be, or}     : {not, not},`
	`          {or, not}    : {to, okay},`
	`          {not, to}    : {be},`
`to be or not to be just`	`          {be, just}   : {be},`
`be who you want to be`	`          {just, be}   : {who},`
`or not okay you want okay`	`          {be, who}    : {you},`
	`          {who, you}   : {want},`
	`          {you, want}  : {to, okay},`
	`          {want, to}   : {be},`
	`          {not, okay}  : {you},`
	`          {okay, you}  : {want},`
	`          {want, okay} : {to},`
	`          {okay, to}   : {be}}`
input file, tiny.txt	*resulting map of 3-gram suffixes*

Note that the **order matters**: For example, the prefix {you, are} is different from the prefix {are, you}. Note that the same word can occur multiple times as a suffix, such as "or" occurring twice after the prefix {to, be}.

Also notice that the map **wraps around**. For example, if you are computing 2-grams, perform 2 more iterations to connect the last 2 prefixes in the end of the file to the first 2 words at the start of the file. In our example above, this leads to {want, okay} connecting to "to" and {okay, to} connecting to "be". If we were doing 5-grams, we would perform 5 more iterations and connect to the first 5 words in the file, and so on. This turns out to be very useful to help your algorithm later on in the program.

You **should not change case or strip punctuation** of words as you read them. The casing and punctuation turns out to help the sentences start and end in a more authentic way. Just store the words in the map as you read them.

Algorithm Step 2: Generating Random Text

To generate random text from your map of *N*-grams, first choose a random starting point for the document. To do this, pick a randomly chosen key from your map. Each key is a collection of *N*-1 words. Those *N*-1 words will form the start of your random text. This collection of *N*-1 words will be your **sliding "window"** as you create your text.

For all subsequent words, use your map to look up all possible next words that can follow the current *N*-1 words, and randomly choose one with appropriate weighted probability. If you have built your map the way we described, as a map from {prefix} → {suffixes}, this simply amounts to choosing one of the possible suffix words at random. Once you have chose your random suffix word, slide your current "window" of *N*-1 words by discarding the first word in the window and appending the new suffix. The following diagram illustrates the text generation algorithm.

Action(s)	Current (*N*-1) "window"	Output so far
choose a random start	`{"who", "you"}`	`who you`
choose new word; shift	`{"you", "want"}`	`who you want`
choose new word; shift	`{"want", "okay"}`	`who you want okay`
choose new word; shift	`{"okay", "to"}`	`who you want okay to`
...	`...`	`...`

Note that in our random example, at one point our window was {want, okay}. This was the end of the original input file. Nothing actually follows that prefix, which is why it was important that we made our map **wrap around** from the end of the file to the start, so that if our window ever ends up at the last *N*-1 words from the document, we won't get stuck unable to generate further random text.

Since your random text likely won't happen to start and end at the beginning/end of a sentence, just prefix and suffix your random text with "..." to indicate this. Here is another partial log of execution:

```
Input file? tiny.txt
Value of N? 3
# of random words to generate (0 to quit)? 16
... who you want okay to be who you want to be or not to be or ...
```

Your code should check for several kinds of **user input errors**, and not assume that the user will type valid input. Specifically, re-prompt the user if they type the name of a file that does not exist. Also re-prompt the user if they type a value for *N* that not an integer, or is an integer less than 1. You may assume that the value the user types for *N* is not greater than the number of words found in the file. See the logs of execution on the course web site for examples of proper program output for such cases.

Development Strategy and Hints:

This program can be tricky if you don't develop and debug it step-by-step. Don't try to write everything all at once. Make sure to **test** each part of the algorithm before you move on. See the **Homework FAQ** for more tips.

- Think about exactly what **types of collections** to use for each part. Are duplicates allowed? Does order matter? Do you need random access? Where will you add/remove elements? Etc. Note that some parts of each program require you to make compound collections, that is, a collection of collections.
- Test each function with a very **small input** first. For example, use input file tiny.txt with a small number of words because you can **print your entire map** and examine its contents.

- Recall that you can **print** the contents of any collection to `cout` and examine its contents for debugging.
- Remember that when you assign one collection to another using the = operator, it makes a full copy of the entire contents of the collection. This could be useful if you want to copy a collection.
- To choose a random prefix from a map, consider using the map's keys member function, which returns a Vector containing all of the keys in the map. For **randomness** in general, include "random.h" and call the global function `randomInteger(`***`min, max`***`)`.
- You can loop over the elements of a vector or set using a for-each loop. A for-each also works on a map, iterating over the keys in the map. You can look up each associated value based on the key in the loop.
- Don't forget to test your input on **unusual inputs**, like large and small values of *N*, large/small # of words to generate, large and small input files, and so on. It's hard to verify random input, but you can look in smallish input files to verify that a given word really does follow a given prefix from your map.
- Your solution should match the flow and prompts from the sample solution available on the course website and shown above.

Implementation and Grading:

All items mentioned in the "Implementation and Grading" from the previous assignment(s) specs also apply here. Please refer to those documents as needed. Note the instructions in the previous assignment about procedural decomposition, variables, types, parameters, value vs. reference, and commenting. Don't forget to **cite any sources** you used in your comments. Refer to the course **Style Guide** for a more thorough discussion of good coding style.

Collections: Additionally, on this assignment part of your Style grade comes from making intelligent decisions about what kind of **collections** from the Stanford C++ library to use at each step of your algorithm, as well as using those collections elegantly. As much as possible, **pass collections by reference**, because passing them by value makes an expensive copy of the collection. Do not use pointers, arrays, or STL containers on this program.

Honor Code: Please remember to follow the **Honor Code** when working on this assignment. Submit your own work and do not look at others' solutions. Also please do not give out your solution and do not place a solution to this assignment on a public web site or forum. If you need help, please seek out our available resources to help you.

For reference, our solution to Part A (the complete file) is around 115 lines long including spacing and comments. Our solution to Part B (the complete file) is around 130 lines long, and it has 4 functions besides main, though you do not need to match or come close to these numbers to get full credit; they are just here as a ballpark sanity check.