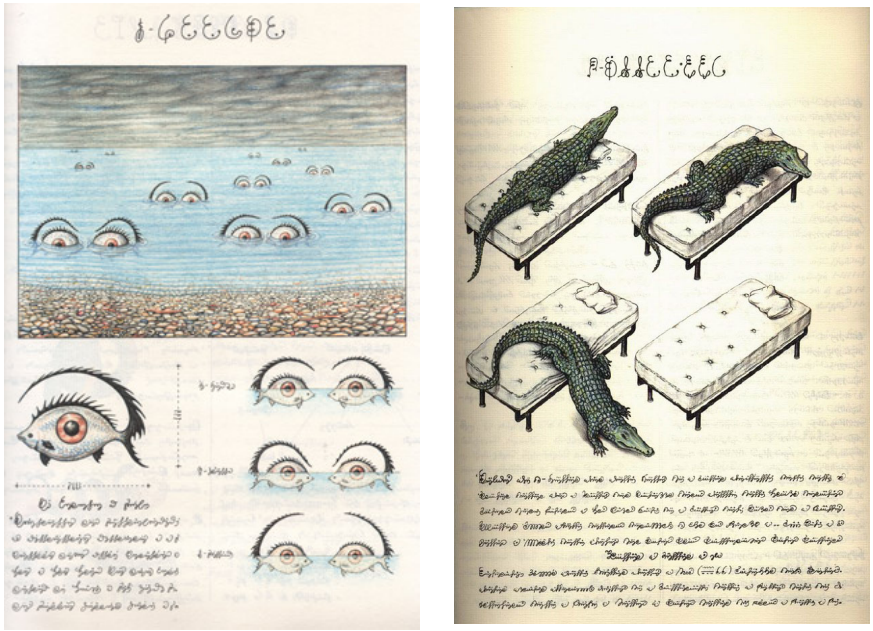
Assignment 2: Serafini

Thanks to Julie Zelenski, Jerry Cain, Marty Stepp, and Chris Piech; Random Writer comes from Joe Zachary.

APRIL 13TH, 2017



Images from Codex Seraphinianus by Serafini

This Serafini in reference to Luigi Serafini who wrote a book called the Codex Seraphinianus. The codex is filled with scientific looking writing and figures, all of which are believed to be the product of Serafini's imagination. Two features that Serafini's work primarily embody are morphings—many objects in the codex change from one to another—and random writing.

As such, this assignment consists of two parts (A) Word Ladders where you find ways to morph one word into another and (B) Random Writter where your program stochastically generates text. Each part can be programmed separately, but they should be submitted together. The starter code for this project is availible as a ZIP archive:

Starter Code

Due Date: Serafini is due Thursday, April 20th at 12:00pm (noon).

Y.E.A.H hours (will be recorded):
Thursday, April 13th
3:30-4:20pm
Gates B03
Last Quarter's Slides

Turn in only the following files:

1. **`wordladder.cpp`**, the C++ code for the Part A Word Ladder program (including a main function)
2. **`ngrams.cpp`**, the C++ code for the Part B N-grams program (including a main function)
3. **`myinput.txt`**, your own unique Part B input file representing text to read in as your program's input

This is a *not* a pair assignment (soon!). You must work on your code alone, although as always you are allowed to talk about the problems with other students, on Piazza, in LaIR, office hours, etc.

Part A: Word Ladder

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 the word **`"code"`** to the word **`"data"`**. Each changed letter is underlined as an illustration:

`code` → **`c`**<u>**`a`**</u>**`de`** → **`ca`**<u>**`t`**</u>**`e`** → <u>**`d`**</u>**`ate`** → **`dat`**<u>**`a`**</u>

There are many other word ladders that connect these two words, but this one is the shortest. That is, there might be others of the same length, but none with fewer steps than this one.

In the first part of this assignment, write a program that repeatedly prompts the user for two words and finds a minimum-length ladder between the words. You must use the Stack and Queue collections from Chapter 5, along with following a particular provided algorithm to find the shortest word ladder. This part is simpler than Part B. Here is an example log of interaction between your program and the user (with console input underlined):

Example Run:

Here is an example log of interaction between your program and the user (with console input in red).

```
Welcome to CS 106B Word Ladder.
Please give me two English words, and I will change the
first into the second by changing one letter at a time.

Dictionary file name? dictionary.txt

Word #1 (or Enter to quit): code
Word #2 (or Enter to quit): data
A ladder from data back to code:
data date cate cade code

Word #1 (or Enter to quit):
Have a nice day.
```

Notice that the word ladder prints out in reverse order, from the second word back to the first. If there are multiple valid word ladders of the same length between a given starting and ending word, your program would not need to generate exactly the ladder shown in this log, but you must generate one of minimum length.

Your code should ignore case; in other words, the user should be able to type uppercase, lowercase, mixed case, etc. words and the ladders should still be found and displayed in lowercase. You should also check for several kinds of **user input errors**, and not assume that the user will type valid input. Specifically, you should check that both words typed by the user are valid words found in the dictionary, that they are the same length, and that they are not the same word. If invalid input occurs, your program should print an error message and re-prompt the user. See the logs of execution on the course web site for examples of proper program output for such cases. You will need to keep a dictionary of all English words. We provide a file dictionary.txt that contains these words, one per line. We also suggest using the Stanford library's **`Lexicon`** class to store the dictionary. Using the **`Lexicon`** class is not hard -- please see the Lexicon class reference for details. You will need to read in the dictionary of words and then you should simply use the **`contains()`** method to determine if a word is in the dictionary when you need that functionality.

Your program should prompt the user to enter a dictionary file name and use that file as the source of the English words. (If the user types a file name that does not exist, reprompt them; see the second execution log on the next page.) Read the file a single time in your program, and choose an efficient collection to store and look up words. Note that you should not ever need to loop over the dictionary as part of solving this problem.

Here are a few more example runs:


Ladder Run #1


Ladder Run #2


Ladder Run #3


Ladder Run #4

Algorithm:

Finding a word ladder is a specific instance of a shortest-path problem of finding a path from a start position to a goal. Shortest-path problems come up in routing Internet packets, comparing gene mutations, Google Maps, and so on. The strategy we will use for finding a shortest path is called breadth-first search ("BFS"), a search process that expands out from a start position, considering all possibilities that are one step away, then two steps away, and so on, until a solution is found. BFS guarantees that the first solution you find will be as short as any other.

For word ladders, start by examining ladders that are one step away from the original word, where only one letter is changed. Then check all ladders that are two steps away, where two letters have been changed. Then three, four, etc. We implement the breadth-first algorithm using a queue to store partial ladders that represent possibilities to explore. Each partial ladder is a stack, which means that your overall collection is a queue of stacks.

Here is a partial pseudocode description of the algorithm to solve the word-ladder problem:

```
Finding a word ladder between words w1 and w2:
   Create an empty queue of stacks.
   Create/add a stack containing {w1} to the queue.
   While the queue is not empty:
      Dequeue the partial-ladder stack from the front of the queue.
      For each valid English word that is a "neighbor" (differs by 1 lett
      of the word on top of the stack:
         If that neighbor word has not already been used in a ladder befo
            If the neighbor word is w2:
               Hooray! we have found a solution (and it is the stack you
            Otherwise:
               Create a copy of the current partial-ladder stack.
               Put the neighbor word on top of the copy stack.
               Add the copy stack to the end of the queue.
```

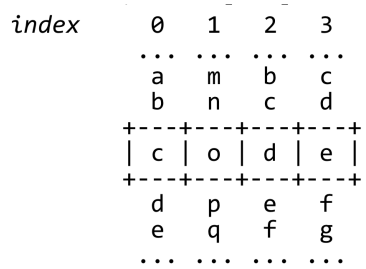
Some of the pseudocode corresponds almost one-to-one with actual C++ code. One part that is more abstract is the part that instructs you to examine each "neighbor" of a given word. A neighbor of a given word w is a word of the same length as w that differs by exactly 1 letter from w. For example, **`"date"`** and **`"data"`** are neighbors.

It is not appropriate to look for neighbors by looping over the entire dictionary every time; this is way too slow. To find all neighbors of a given word, use two nested loops: one that goes through each character index in the word, and one that loops through the letters of the alphabet from a-z, replacing the character in that index position with each of the 26 letters in turn. For example, when examining neighbors of "date", you'd try:

- **`aate,bate,cate,...,zate`** ← *all possible neighbors where only the 1st letter is changed.*
- **`date,dbte,dcte,...,dzte`** ← *all possible neighbors where only the 2nd letter is changed.*
- ...
- **`data,datb,datc,...,datz`** ← *all possible neighbors where only the 4th letter is changed.*

Note that many of the possible words along the way (**`aate`**, **`dbte`**, **`datz`**, etc.) are not valid English words. Your algorithm has access to an English dictionary, and each time you generate a word using this looping process, you should look it up in the dictionary to make sure that it is actually a legal English word.

Another way of visualizing the search for neighboring words is to think of each letter index in the word as being a "spinner" that you can spin up and down to try all values A-Z for that letter. The diagram below tries to depict this:



Another subtle issue is that you do not reuse words that have been included in a previous shorter ladder. For example, suppose that you have the partial ladder cat → cot → cog → to the queue. Lateron, if your code is processing ladder cat → cot → con, one neighbor of con is cog, so you might want to examine cat → cot → con → cog. But doing so is unnecessary. If there is a word ladder that begins with these four words, then there must be a shorter one that, in effect, cuts out the middleman by eliminating the unnecessary word con. As soon as you've enqueued a ladder ending with a specific word, you've found a minimum-length path from the starting word to the end word in the ladder, so you never have to enqueue that end word again.

To implement this strategy, keep track of the set of words that have already been used in any ladder. Ignore those words if 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.

It is helpful to test your program on smaller dictionary files first to find bugs or issues related to your dictionary or word searching.

Part B: Random Writter

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. Bellow 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 name? 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. (See here) 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.

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.

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-1 words along with their possible following words, you can use this to select an Nth word given the preceding N-1 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.

Here are a few more example runs:


Writter Run #1


Writter Run #2


Writter Run #3


Writter Run #4


Writter Run #5

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:

File Location	Data
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}
```

to be or not to be just
be who you want to be
or not okay you want okay|

```
map    = { {to, be} : {or, just, or},
           {be, or} : {not, not},
           {or, not} : {to, okay},
           {not, to} : {be},
           {be, just} : {be},
           {just, be} : {who},
           {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} }
```

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 3-grams like the above example, 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 4 more iterations and connect the last 4 prefixes to the first 4 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.

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.

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 2 (a 2-gram has prefixes of length 1; but a 1-gram is essentially just choosing random words from the file and is uninteresting for our purposes). When prompting the user for the number of words to randomly generate, re-prompt them if the number of random words to generate is not at least N. 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 various cases.

Creative Aspect myinput.txt

Along with your program, submit a file myinput.txt that contains a text file that can be used as input for Part B. This can be anything you want, as long as it is non-empty and is something you gathered yourself (not just a copy of an existing input file). This is meant to be just-for-fun; for example, if you like a particular band, you could paste several of their songs into a text file, which leads to funny new songs when you run your N-grams program on this data. Or gather the text of a book you really like, or poems, or anything you want. This is worth a small part of your grade on the assignment.

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.

1. Unlike in the previous assignment, here we are interested in each word. If you want to read a file one word at a time, an effective way to do so is using the input >> variable; syntax rather than getLine, etc.
2. 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.
3. 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.
4. Recall that you can print the contents of any collection to cout and examine its contents for debugging.
5. 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.
6. 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).

7. 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.
8. 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.

Possible Extra Features:

Thats all! You are done. Consider adding extra features.

Show Extensions