CSCI-1200 Data Structures — Spring 2022
Homework 7 — Word Frequency Maps

In this homework assignment, we will analyze the frequency and sequencing of words in sentences in samples of English text. Then we will generate new sequences of strings that mimic these statistics and present these sequences as computer-generated text. The key idea is that if particular words often appear together in the sample text, then it is likely that stringing together these common pairs of words will form plausible phrases or sentences. This statistical model is called a *Markov chain*. By using STL's associative container (`map`) we can make this system efficient and elegant. *Please carefully read the entire assignment before beginning your implementation.*

As an example, let's consider the text of the Brothers Grimm fairy tale *Hansel and Gretel*, available from Project Gutenberg http://www.gutenberg.org/. In this text (translated from the original German to English), the word "could" appears 5 times. 2 of those times it is immediately followed by "not" and the other 3 times it is followed by "be", "get", and "no" (once each). If we start a phrase or sentence with the word "could", a good guess for the second word is "not". The statistics of the word "not" tell us it appears 26 times. 3 of those times it is followed by "find" (more often than any other word), so we choose "find" as our third word. Then we look up the statistics for "find", etc. This process yields a plausible phrase "could not find the forest", which in fact does *not* appear in the original text!

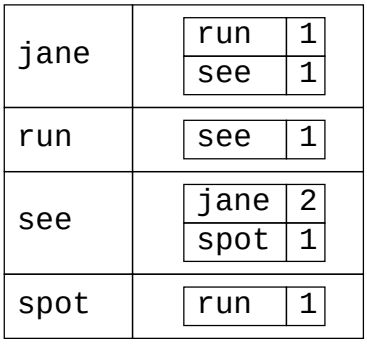
This general strategy has been used to generate plausible (but often nonsensical) sentences. Here's a famous example written by Mark V. Shaney (a computer program written by Bruce Ellis):

"I spent an interesting evening recently with a grain of salt."

Using an STL `map` to Store the Word Frequency & Sequencing

As we process the sample text we'll keep track of all the words we have seen, and for each word we'll keep track of all the words that immediately followed that word, and the number of times each pair of words appeared together. We will use maps to store this information. The picture at the right represents the map structure created when the text below is read into the `data` variable using the `LoadSampleText` function.

```
see spot run
see jane
see jane run
```



What is the type of the `data` variable suggested by this example and diagram? *Hint: there are two maps!* How will you process and store the long sequence of words from the sample text into this structure? Note: We treat the input as one big run-on sentence. We provide some code that will help you parse large input files (like the plain text books available from Project Gutenberg) and ignore punctuation and capitalization.

Next Word Prediction

Once the sample text is stored into the `data` variable, your next task will be to implement the `NextWord` function. Given the last word in the current sequence, this function will use the statistics in `data` to select the next word for the sequence. Your function should work in two different modes: 1) selecting the most frequently observed next word or 2) selecting at random with the probabilities observed in the sample text. So using the second mode to chose the word after "could", we will select "not" with $2/5 = 40\%$ probability, and "be", "get", and "no" with $1/5 = 20\%$ probability each.

Input/Output & Basic Functionality

Your program should not expect any command line arguments, and will read from `std::cin` and write to `std::cout`, but you will probably find it helpful to *redirect* the input (& output) to trick your program into reading from & writing to files (see the course webpage under "Helpful C++ Programming Information"). Each line of input data begins with a keyword signaling one of four commands.

`load hansel_and_gretel.txt 2 ignore_punctuation` This command loads the sample text from the specified file and stores the sequence and frequency information into the `data` variable map structure. The integer parameter specifies the window width over which sequencing data should be collected (to begin the assignment use window=2, but more on this later). The last parameter indicates how punctuation (and capitalization) should be handled. Implementing special parsing of punctuation and using punctuation in your word selection and output is extra credit.

`print "could"` With this command, we query the statistics in the `data` variable, to learn what words were observed to follow the specified word (or phrase). The word (or phrase) will be in double quotes (allowing us to have multiple words). We provide code to help you parse this input. This query on the Hansel and Gretel data produces this output (to `cout`).

```
could (5)
could be (1)
could get (1)
could no (1)
could not (2)
```

`generate "could" 4 most_common` This is the more interesting command. Here we are requesting to add 4 more words to the initial word (or phrase) by selecting the highest frequency follow-on word. This command will output the phrase "could not find the forest" with the Hansel and Gretel dataset.

`generate "could" 10 random` And similarly, this command generates 10 words to follow "could", but using a random draw from the observed word sequence frequencies.

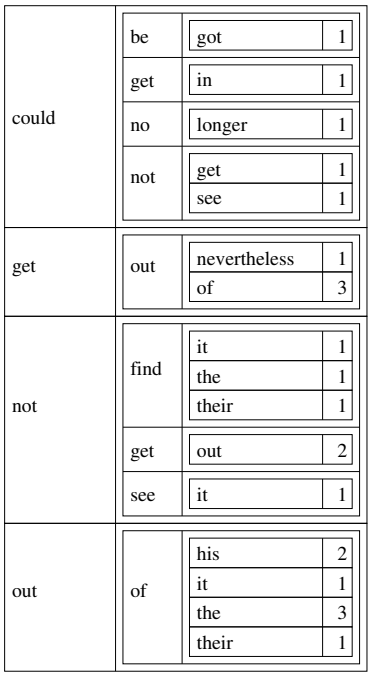
`quit` This command closes the program.

These commands may be entered directly from `std::cin` during an interactive data querying and play session, or drawn in from a file using file redirection. You can use the STL's `std::srand()` to "seed" the random number generator (either with a fixed constant for repeatable testing or the current time for variety) and the `std::rand()` function to generate your numbers. To see if your program matches our output exactly (except for the randomness of course!), you may use the command line UNIX library program `diff` (available for Linux, MacOS, and WSL), which takes two files as arguments and outputs the differences between them. Please see a TA or the instructor in office hours if you have a question about these programs.

Expanding the Sequencing Window or Context

So far, our examples have considered only the last word in the current sequence when selecting the next word, but in fact we can usually create more realistic phrases and sentences by expanding the *context* or window of the sequencing probabilities. Specifically when generating the third word, let's not just consider the second word, but let's consider both the first *and* the second word. For the Hansel and Gretel input, the 2 word sequence "could not" appears just twice, followed by "get" and "see" (once each). This suggests that one of these words is perhaps a better choice for the third word than the earlier example's selection of "find". (Note: when we have a tie for the most frequently appearing next word, we will chose the first alphabetically, in this case "get".) Similarly, to generate the fourth word, using window = 3, we start with the sequence "not get" and discover that the only time this subphrase appears in the sample text it is followed by "out". Let's expand our `data` map type diagram accordingly, as shown below:

Note: Other data structure designs for this application are possible, but please follow the diagram below for this assignment to ensure you get plenty of practice with STL `map`.



There are tradeoffs to using a much larger window or context. First of all, the map data structure will require more memory. Furthermore, if the context is too large, the technique can "overfit" the data. If only one passage of the text matches the query sequence over the large window, then the program will not have any choices when generating the sequence, and the output will just be a copy of a single passage of the sample text.

For the core homework assignment, first implement the 2 word window shown in the first diagram. When that is working, move to the 3 word window in the diagram on the right. For full credit on the homework, your code should work with window=2 and window=3. For extra credit, extend your solution to work for any window size. *Warning: this extension is not straightforward. You may add components to this diagram for your implementation, but the overall structure and arrangement of data should keep the same spirit.*

Performance & Big 'O' Notation

Consider the performance of each of the commands outlined above. Let n be the *total* number of words in the sample text file, let m be the number of *unique* words in the file, let w be the width of the sequencing window, let p be the *average* number of words observed in the sample to follow a particular word, and let x be the number of words that should be generated. How much memory will the map data structure require, in terms of n, m, w, p, and x (big 'O' notation for memory use)? What is the big 'O' notation for performance (running time) of each of the commands? Write these answers in your `README.txt` file.

You are not explicitly required to create any new classes when completing this assignment, but please do so if it will improve your program design. We expect you to use `const` and pass by reference/alias as appropriate throughout your assignment. We have provided a partial implementation of the main program to get you started. You may use none, a little, or all of this, as you choose, but we strongly urge you to examine it carefully.

Extra Credit

There are many options for extra credit on this assignment. (As usual though, the total number of points awarded to extra credit work will be small.) You may brainstorm and implement an extension for punctuation data. In particular, recognizing the end of a sentence (a period or question mark), might improve the readability of your phrases and sentences. If your implementation supports the use of any window size, summarize the results of your testing with different window sizes. What feels like the "right" window size for a particular dataset? Be sure to make up your own test cases too. Project Gutenberg is an excellent resource. Report your performance statistics on your largest test cases. Obviously, you won't be able to submit large datasets (due to the hw submission size), but describe them in your README.txt and cut & paste any interesting "philosophical" statements your program generates.

Submission

Use good coding style and detailed comments when you design and implement your program. Please use the provided template `README.txt` file for any notes you want the grader to read, including work for extra credit. You must do this assignment on your own, as described in the "Collaboration Policy & Academic Integrity". If you did discuss the problem or error messages, etc. with anyone, please list their names in your `README.txt` file.