Project 2 – Generating Text
CS 251, Fall 2020, Reckinger

Collaboration Policy: By submitting this assignment, you are acknowledging you have read the collaboration policy in the syllabus for projects. This project should be done individually. You may not receive any assistance from anyone outside of the CS 251 teaching staff.
Late Policy: You are given a total of 5 late days to use at your discretion (for all projects, not each project). You may use the late days in 24-hour chunks (either 1 day at a time or all five at once). You do not need to alert the instructor to ask permission to use late days. You can manage your use of late days on Mimir directly.
What to submit: (1) textGenerator.cpp, your solution to this project, (2) myinput.txt, your own text file to generate text from, (3) why.txt explaining why you chose this text, (4) myanalysis.txt which is the Big-O for your build map and generate text commands. *NOTE: You can submit our whole project folder, but all other files are ignored (except these four).*
.pdf starter-code solution.exe

Project Summary

You will write a program that takes in a text file and then (randomly) generates new text in the same style. In effect, your program will automate an exercise that writers have used for centuries to improve their style: imitating another writer. Benjamin Franklin, for example, wrote in his autobiography that he taught himself to write elegantly by trying to re-write passages from The Spectator (a popular 18th-century magazine) in the same style. In the 20th century, the writer Raymond Queneau featured the practice of trying to imitate the voice of other writers prominently among his famous Exercises in Style. A more recent computational style imitator is the postmodern essay generator, created in 1996; hit refresh a few times to see what it can do. Your program will accomplish the task of imitating a text's style by building and relying on a large data structure of word groups called "N-grams." A collection of N-grams will serve as the basis for your program to randomly generate new text that sounds like it came from the same author as your input file; your program will also use the C++ collections you have learned about. Below is an example log of interaction between your program and the user (console input in **red**):

```
Welcome to the Text Generator.
This program makes random text based on a document.
Input file name?  tiny.txt
Value of N?  3
Total words you'd like to generate?  10
Type b-build map, p-print map, g-generate text, s-start over, x-to exit: b

...Building map: tiny.txt...

Type b-build map, p-print map, g-generate text, s-start over, x-to exit: p

{be just} -> {be}
{be or} -> {not not}
{be who} -> {you}
{just be} -> {who}
{not okay} -> {you}
{not to} -> {be}
{okay to} -> {be}
{okay you} -> {want}
{or not} -> {to okay}
```

```
{to be} -> {or just or}
{want okay} -> {to}
{want to} -> {be}
{who you} -> {want}
{you want} -> {to okay}

Type b-build map, p-print map, g-generate text, s-start over, x-to exit: g

...you want to be or not okay you want to...

Type b-build map, p-print map, g-generate text, s-start over, x-to exit: x
```

(Remember: the 'g' command is randomly generating text, so yours might not be the same, see tips below to see how the autograder works).

But what, you may ask, is an N-gram? 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, sentence structure, 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? *(Warning: if you write the algorithm correctly, you do not need to be calculating probabilities' or even keeping track of probabilities. Therefore, keep reading and understanding the problem. But if you are trying to put together a solution that takes into account probabilities, you are going off in the wrong direction!)*

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" 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. Here's an example sentence produced by linking together a sequence of 2-grams:

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

A sentence of 2-grams tends not make any sense (or even be grammatically correct), so let's consider 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, together with their 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. Here's a sample sentence produced by linking 3- grams together:

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

We can generalize this idea from 2- or 3-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 your choice of N, 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:

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

As you can see, the text is now starting to sound a lot like the original. 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.

Building a Map

Your program will read the input file one word at a time and build a specific kind of compound collection, namely, a map from each (N-1)-gram in your input text to all of the words that occur directly after it in the input. For example, if you are building sentences out of 3-grams, that is, N-grams for N=3, then your code should examine each sequence of 2 words and keep track of which third word directly follows it each time it occurs. So if you are using 3-grams 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 "probability" of each word does not need to be stored, instead, duplicate copies will be stored. The more frequent a word, the more copies in the value. We might think of these (N-1)-grams as "prefixes" and the words that follow them as various possible "suffixes." Using this terminology, your map should be built so that it connects a collection of each prefix sequence (of N-1 words) to another collection of all possible suffixes (all Nth words that follow the prefix in the original text).

The following figure illustrates the map you should build from the input file. As this figure shows, when reading the input file, you should keep track of a window of N-1 words at all times. As you read each word from the file, you should insert the previous window as a key in your collection of prefixes, with the newly-read word, the current suffix, as its corresponding value. **(Remember that your collection of suffixes must be able to contain multiple occurrences of the same word—such as "or" occurring twice after the prefix {to, be}—otherwise, your collection will not consider the word-frequency information needed.)** Then, discard the first word from the window, append the new word to the window, and repeat. The map is thus built over time as you read each new word:

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, 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 of sequences matters: for example, the prefix {you, are} is different from the prefix {are, you}. Also notice that the map wraps around. For example, if you are computing 3-grams like in the above example, perform 2 more iterations to connect the last 2 prefixes at 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 will be very useful for 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.

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, randomly choose a key from your map using the function provided in "myrandom.h" called **randomInteger**. Open up myrandom.h and read the function header comment to understand how the function works. It is required that you use this function for generating random numbers, your code will not pass autograder tests if you generate random numbers in any other way! Remember that each key is a prefix, a collection of N-1 words. Those N-1 words will form the start of your random text, and will be the first sequence in your sliding "window" as you create your text. In order to pass the autograder, you must randomly choose a key from a *vector of keys that are in sorted order*. Since you are using a map, this is easy to do because maps can iterate through keys in sorted order (see map lecture for details). You should not be sorting any data.
Once you randomly choose the key (prefix) to start with, you will use your map to lookup the value associated with that key, which should be a list of possible suffixes. Now, randomly choose one of the suffixes (again, using the **randomInteger** function) and you will have your first N words of your generated text. For example, with tiny.txt, let's say your randomly selected first key was {to, be}. Using the map, the key maps to this value: {to, be}->{or, just, or}. When

you randomly select a suffix from that list, let's say "just" is randomly selected, then you have these N words generated: "to be just".
For all subsequent words, use your map to look up the new prefix (by sliding the window over). In the tiny.txt example, you take the text you have generated, "to be just", and slide the window over to get your next prefix, which is {be, just}. If you have built your map the way we described above, as a map from {prefix} → {suffixes}, this simply amounts to using the map to get the suffixes from the new prefix and then choosing one of the possible suffix words at random and adding it to your generated text (as you did before). Continue this process, adding one new word to your generated text at a time, until your generated text has the desired number of words requested by the user. *NOTE: Because your suffix list has duplicates, this is how you account for the frequency of words in the original text.*
Since your random text is unlikely to start and end at the proper beginning or end of a sentence, just preface and conclude your random text with "..." to indicate this. See sample output above which shows this.

File Reading

The files you are reading in for the project will not work with the file reading template provided during week one. That template only works for files with end with a single new line character. We are broadening our file reading in this project to work more generally. Here are two file reading templates that will work with this project:

1. This is Zybooks file reading template:

```
while (!infile.eof()) {
  infile >> word;
  if (!infile.fail()) {
    // do stuff with word
  }
}
```

2. This is another template option that works:

```
while (infile >> word) {
  // do stuff with word;
}
```

These allow you to catch the failbit properly when the end of file is reached.

Starting Over

This command should cout the welcome message and reset the input file, N, and total words from user. Don't forget to clear your map!

Error Checking

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:

```
Welcome to the Text Generator.
This program makes random text based on a document.
Input file name?  tiny1.txt
Invalid file, try again:  tiny2.txt
Invalid file, try again:  tiny.txt
Value of N?
```

…

Also, re-prompt the user if they type a value for N that 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):

```
Welcome to the Text Generator.
```

```
This program makes random text based on a document.
Input file name?  tiny.txt
Value of N?  1
N must be > 1, try again:  0
N must be > 1, try again:  2
Total words you'd like to generate?
```
…

You may assume that the value the user types for N is not greater than the number of words found in the file. 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:

```
Welcome to the Text Generator.
This program makes random text based on a document.
Input file name?  tiny.txt
Value of N?  6
Total words you'd like to generate?  4
Total words must be at least N, try again:  5
Total words must be at least N, try again:  6
Type b-build map, p-print map, g-generate text, s-start over, x-to exit:
```
…

Creative Aspect - myinput.txt & why.txt

Along with your program, submit a file myinput.txt that contains a text file that can be used as input. This can be anything you want, something you gathered yourself (not just a copy of an existing input file). 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 program on this data. Or gather the text of a book you really like, or poems, or anything you want. This is meant to give you a chance to play around with the creative power of your program. It should be a minimum of 3000 words (which is about 6 pages of text). Additionally, submit a file saved as why.txt that explains in 250 words or less why you chose this piece of text (why it is important to you). Outside of the 250 words, feel free to also paste in a copy of the text you generated from myinput.txt that you particularly liked to share with the TA who is grading!

Big-O Analysis – myanalysis.txt

Along with your program, submit a file myanlysis.txt. In 250 words or less, you should:
-State the Big O of your build map and generate text commands.
-Justify how you determined the Big O of both of those commands.

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; this will let you print your entire map and examine its contents for debugging purposes.

4. Similarly, make sure you are writing a function to print out your map. Do this right away, it will really help with debugging. It is part of what is tested in the test cases, but it should be one of the first things you write!
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. You know that you can loop over the elements of a vector or set using a for-each loop. A foreach loop also works on a map, iterating over its keys. You can access each associated key/value based in the loop. Don't forget…use references, when possible.
7. To choose a random prefix from a map, consider writing a helper function that places all the keys into a vector. The keys must be in sorted order (alphabetical) in the vector. The best way to do this is to use a for-each loop on your map and fill a vector (for-each loops in the map collection will loop through keys in alphabetical order). For randomness,include "myrandom.h" and call the global function **randomInteger**(min, max). Do not use srand or rand in <**stdlib.h**> library, as it will not work for the **autograder**.
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.
9. If you have trouble getting a correct map for N>3, try building out by hand for tiny.txt (similar to Building Map visualization section in this handout). You can use the solution.exe to make sure you built the map correctly and it might help develop the code generally for all Ns.

Requirements

1. You are only allowed to use the five C++ containers presented in lecture this week (vector, stack, queue, set, map). No other abstractions or containers are allowed for storing data. You may not need to use all of them.
2. You are allowed to use and add other libraries (make sure to **include** them at the top of your file). Some you might find useful: <iostream>, <sstream>, <string>, <fstream>, etc. You may not need some of these, there are lots of ways to solve this problem.
3. Each file may be read exactly once. It can be opened more than once for the purposes of error checking.
4. The data may only be stored one time, in an abstraction while processing the data. Looping through the file and storing all words in a vector and then looping through that vector to store in map will result in space efficiency penalties. However, you will have a second copy of your keys.
5. Your textGenerator.cpp program file must have a header comment with your name and a program overview. Each function must have a header comment above the function, explaining the function's purpose, parameters, and return value (if any). Inline comments should be supplied as appropriate; comments like "*declares variable*" or "*increments counter*" are useless. Comments that explain non-obvious assumptions or behavior *are* appropriate.
6. Your code must be properly decomposed into functions per the style guidelines.

7. No global variables; use parameter passing and function return.
8. The **cyclomatic complexity** (CCN) of any function should be minimized. In short, cyclomatic complexity is a representation of code complexity --- e.g. nesting of loops, nesting of if-statements, etc. You should strive to keep code as simple as possible, which generally means encapsulating complexity within functions (and calling those function) instead of explicitly nesting code. As a general principle, if we see code that has **more than 2 levels** of explicit looping --- an example of which is shown above --- that score will receive grade penalties. The solution is to move one or more loops into a function, and call the function.

Citations/Resources

Assignment is inspired by Chris Gregg, Stanford University and, originally, Julie Zelenski, Stanford University.