Assignment 2B: NGrams

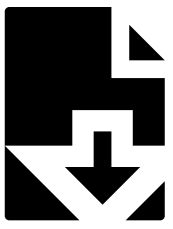
Assignment by Chris Piech, Marty Stepp, and Victoria Kirst. Originally based on a problem by Julie Zelenski and Jerry Cain.

DUE WEDNESDAY, JULY 11 AT 5PM PDT

This problem practices using maps and other collections.

Links:

Starter files:



Starter Code

We provide a ZIP archive with a starter project that you should download and open with Qt Creator. You will edit and turn in only the following file. The ZIP contains other files/libraries; do not modify these. Your code must work with the other files unmodified.

- **ngrams.cpp**, the C++ code for your solution
- **myinput.txt**, your own unique input file representing text to read in as your program's input. If you work in a pair, you should submit this file **individually**, labelling the two files **myinput_SUNET1.txt** and **myinput_SUNET2.txt**.

demo JAR

If you want to further verify the expected behavior of your program, you can download the provided sample solution demo JAR and run it. If the behavior of our demo in any way conflicts with the information in this spec, you should favor the spec over the demo.

"How do I run the assignment solution demos?"

Our assignments offer a solution 'demo' that you can run to see the program's expected behavior. On many machines, all you have to do is download the .jar file, then double-click it to run it. But on some Macs, it won't work; your operating system will say that it doesn't know how to launch the file. If you have that issue, download the file, go to the Downloads folder in your Finder, right-click on the file, and click Open, and then press Confirm.

Some Mac users see a security error such as, "cs106b-hw2-wordladder-demo.jar can't be opened because it is from an unidentified developer." To fix this, go to System Preferences → Security & Privacy. You will see a section about downloaded apps. You should see a prompt asking if you would like to allow access to our solution JAR. Follow the steps and then you should be able to open the demo.

If all else fails, you could run the demo JAR from a terminal. Every operating system allows you to open a "terminal" or "console" window for typing raw system commands. Open your operating system's terminal or console window (Google if you need to learn how to do this), and then type:

```
cd DIRECTORY_WHERE_DEMO_JAR_IS_STOREDjava –jar JAR_FILE_NAME
```

For example, on a Mac machine, if your user name is jsmith12 and you have saved a demo JAR named hw2.jar in your Documents/106b directory, you would type:

```
cd /users/jsmith12/Documents/106bjava –jar hw2.jar
```

Or on a Windows machine, if your user name is jsmith12 and you have saved a demo JAR named hw2.jar in your Documents/106b directory, you would type:

```
cd C:\users\jsmith12\Documents\106bjava –jar hw2.jar
```

Program Description:

In this problem 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** collection from Chapter 5.

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*-1 words along with their possible following words, you can use this to select an *N*th 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.

Example Logs of Execution:

Here is an example log of interaction from your program (with user input bolded). Your output must match this format exactly to earn full credit. (We are very picky.)

```
Welcome to CS 106B/X Random Writer ('N-Grams')!
This program generates 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 a
rt 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 spee
ch in't I chiefly lov'd. ...

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

Example log of execution

Here are some additional expected output files to compare. It's hard to match the expected output exactly because it is random. But you should exactly match our overall output format, intro message text, and so on. Also, your program's graphical Console window has a File → Compare Output feature for checking your output. Use it!

- test #1 (tiny)
- test #2 (hamlet)
- test #3 (tomsawyer)
- test #4 (ladygaga)
- test #5 (tiny)

Implementation Details:

Algorithm Step 1: Building a 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}`

…	…

`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}}`

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}. The same word can occur multiple times as a suffix, and this should affect your *N*-grams and probabilities, 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.

Your code **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 should 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 is 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*. If the value the user types for *N* is greater than the number of words found in the file, you should print an error message as shown in the example logs. See the linked logs of execution for examples of proper program output for various cases.

Creative Aspect, **myinput.txt**:

Along with your code, submit a file **myinput.txt** that contains a text file that can be used as input for this program. This can be anything you want, as long as it is non-empty and is not just a copy of an existing input file provided in the starter ZIP. 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 like, or a poem, or anything you want. This is worth a small part of your grade on the assignment. **Pairs must submit two files, one from each partner.**

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 it all at once. Make sure to test each part of the algorithm before you move on. See the FAQ below for more tips.

- 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.
- 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** file 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.
- 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 number 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.

Style Details:

As in other assignments, you should follow our **Style Guide** for information about expected coding style. You are also expected to follow all of the general style constraints emphasized in the Homework 1 specs, such as the ones about good problem decomposition, parameters, using proper C++ idioms, and commenting. The following are additional points of emphasis and style constraints specific to this problem:

Algorithms: You should follow the general algorithms as described in this document and should not substitute a very different algorithm. In particular, you should follow the given overall algorithm for finding and mapping N-Grams as described in this spec. You should also not write a recursive algorithm for finding N-Grams.

Collections: You are expected to make 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 properly. 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.

Redundancy: You should avoid expensive operations that would cause you to reconstruct bulky collections multiple times unnecessarily. For example, you should generate the map of prefixes exactly once; do not regenerate it each time the user asks to generate random text.

Frequently Asked Questions (FAQ):

For each assignment problem, we receive various frequent student questions. The answers to some of those questions can be found by clicking the link below.

Q: Can I use recursion on this assignment?
A: No. Do not use recursion (functions that call themselves).

Q: How do I construct a compound collection? It doesn't compile.
A: You need a space between nested **<>**.

```
Vector<Vector<int>> foo;    // no
Vector<Vector<int> > foo;   // yes
```

Q: When I print out a collection, I see weird symbols like `\042`. Is it a bug? What does it mean?
A: You can ignore that. Those are just special character codes for characters like quotation marks.

Q: How do I get a random number?
A:

```
#include "random.h"
...
int r = randomInteger(min, max);   // inclusive
```

Q: How do I get a random element of a collection?
A: If it's an indexed collection, such as a **`Vector`**, just pick a random index and then go access that index. If it doesn't have indexes, like a set or queue, pick a random index based on the size, and then advance forward (e.g. in a **`foreach`** or **`while`** loop) that many times and grab the element found there.

Q: Do I have to use the exact collections the spec says to use? I want to use a different one.
A: Follow the spec.

Q: What should go in my `myinput.txt` file?
A: Copy/paste a fun source of text from the web. For example, grab lyrics from your favorite band, or text of a book or movie script you like, or just make up any text you want.

Q: When I try to turn in the myinput.txt file, the system doesn't display it. Did the turnin system accept my file?
A: Check the page that displays information about your past submission. Do you see the text file there? If so,

we received it successfully. If not, you may need to submit again. Sometimes this issue comes from the input file being very large (over ~2mb).

Q: Can I use one of the STL containers from the C++ standard library?
A: No.

Q: I already know a lot of C/C++ from my previous programming experience. Can I use advanced features, such as pointers, on this assignment?
A: No; you should limit yourself to using the material that was taught in class so far.

Q: Can I add any other files to the program? Can I add some classes to the program?
A: No; you should limit yourself to the files and functions in the spec.

(Optional) Possible Extra Features:

Though your solution to this assignment must match all of the specifications mentioned previously, it is allowed and encouraged for you to add extra features to your program if you'd like to go beyond the basic assignment. Note that our motivation for allowing extra features is to encourage your creativity, not inflate your grade; so if any points are given for your extra features, they will be minimal. The purpose is to explore the assignment domain further, not to disrupt the class grading curve. Make sure to see the notes below about how to separate your extra code and how to turn it in properly.

Here are some example ideas for extra features that you could add to your program:

- **Make the N-grams be complete sentences:** The typical version of the assignment indicates that you should start and end your input with "..." since it will likely not begin with the start of a sentence nor end with the end of a sentence from the original input. As an extra feature, make it so that when you are creating your map of N-1 word prefixes, you also keep track of which prefixes are the start of a sentence (prefixes whose first word begins with an uppercase letter) and which words are the end of a sentence (words that end with a period, question mark, or exclamation mark). Use this extra data to begin your randomly generated text with a random sentence starter, rather than any random prefix. And instead of generating exactly the number of words requested by the user, keep going until you reach the end of a sentence. That is, if the user requests 100 words, after generating those 100 words, if you aren't at the end of a sentence, keep going until you end it.
- **Other:** If you have your own creative idea for an extra feature, ask your SL and/or the instructor about it.

Indicating that you have done extra features: If you complete any extra features, then in the comment heading on the top of your program, please list all extra features that you worked on and where in the code they can be found (what functions, lines, etc. so that the grader can look at their code easily).

Submitting a program with extra features: Since we use automated testing for part of our grading process, it is important that you submit a program that conforms to the preceding spec, even if you want to do extra features. If your feature(s) cause your program to change the output that it produces in such a way that it no longer matches the expected sample output test cases provided, you should submit two versions of your program file: a first one with the standard file name without any extra features added (or with all necessary features disabled or commented out), and a second one whose file name has the suffix **-extra.cpp** with the extra features enabled. Please distinguish them in by explaining which is which in the comment header. Our turnin system saves every submission you make, so if you make multiple submissions we will be able to view all of them; your previously submitted files will not be lost or overwritten.
