PS6: Random Writer

For this project, we will:

- Analyze input text for transitions between k-grams (a fixed number of characters) and the following letter.
- Produce a probabilitistic model of the text: given a particular k-gram series of letters, what letters follow at what probabilities?
- Use the model to generate nonsense text that's surprisingly reasonable.

1 Background

In the 1948 landmark paper *A Mathematical Theory of Communication*, Claude Shannon founded the field of information theory and revlolutionized the telecommunications industry, laying the groundwork for today's Information Age. In this paper, Shannon proposed using a Markov chain to create a statistical model of the sequences of letters in a piece of English text. Markov chains are now widely used in speech recognition, handwriting recognition, information retrieval, data compression, and spam filtering. They also have many scientific computing applications including the GeneMark algorithm for gene prediction, the Metropolis algorithm for measuring thermodynamical properties, and Google's PageRank algorithm for Web search. For this assignment, we consider a whimsical variant: generating stylized pseudo-random text.

1.1 Markov model of natural language

Claude Shannon proposed a brute-force scheme to generate text according to a Markov model of **order 1**:

> To construct [a Markov model of order 1], for example, one opens a book at random and selects a letter at random on the page. This letter is recorded. The book is then opened to another page and one reads until this letter is encountered. The succeeding letter is then recorded. Turning to another page, this second letter is searched for and the succeeding letter recorded, etc. It would be interesting if further approximations could be constructed, but the labor involved becomes enormous at the next stage.

Shannon approximated the statistical structure of a piece of text using a smiple mathematical model known as a *Markov model.* A Markov model of *order 0* predicts that each letter in the alphabet occurs with a fixed probability. We can fit a Markov model of order 0 to a specific piece of text by counting the number of occurences of each letter in that text, and using these frequencies as probabilities. For example, if the input text is `"gagggagaggcgagaaa"` the Markov model of order 0 predicts that each letter is '`a`' with probability $\frac{7}{17}$, '`c`' with probability $\frac{1}{17}$, and '`g`' with probability $\frac{9}{17}$ because these are the fraction of times each letter occurs. The following sequence of characters is a typical example generated from this model:

```
g a g g c g a g a a g a a a g a g a g a g a a a g a g a a g ...
```

A Markov model of order 0 assumes that each letter is chosen independently. This independence does not coincide with statistical properties of English text because there is a high correlation among successive characters in a word or sentence. For example, '`w`' is more likely to be followed by '`e`' than with '`u`', while '`q`' is more likely to be followed with '`u`' than with '`e`'.

1.2 Order-k

An order-k Markov model uses strings of k-letters to predict text; these are called k-grams. An order-2 Markov model uses two-character strings (or bigrams) to calculate probabilities in generating random letters. For example, suppose that in the text we're using for generating random letters using an order-2 Markov model the bigram `"th"` is followed 50 times by the letter '`e`', 20 times by the letter '`a`', and 30 times by the letter '`o`', because the sequences `"the"`, `"tha"`, and `"tho"` occur 50, 20, and 30 times, respectively while there are no other occurences of `"th"` in the text we're modelling.

Now suppose that in generating random text we generate the bigram `"th"` and based on this we must generate the next random character using the order-2 model. The next letter will be an '`e`' with a probability of 0.5 (50/100); will be an '`a`' with probability 0.2 (20/100); and will be an '`o`' with probability 0.3 (30/100). If '`e`' is chosen, then the next bigram used to calculate random letters will be `"he"` since the last part of the old bigram is combined with the new letter to create the next bigram used in the Markov process.

1.3 More Examples

Imagine taking a book (say, *Tom Sawyer*) and determining the probability with which each character occurs. You'd probably find that spaces are the most common, that the character '`e`' is fairly common, and that the character '`q`' is rather uncommon. After completing this 0 analysis, you'd be able to produce random *Tom Sawyer* text based on character probabilities. It wouldn't have much in common with the real thing, but at least the characters would tend to occur in the proper proportion. In fact, here's an example of what you might produce:

Order 0

> rla bsht eS ststofo hhfosdsdewno oe wee h .mr ae irii ela iad o r te u t mnyto onmalysnce, ifu en c fDwn oee iteo

Now imagine doing a slightly more sophistic order 1 analysis by determining the probability with which each character follows every other character. You would probably discover that '`h`' follows '`t`' more frequently than '`x`' does, and you would probably discover that a space follows '`.`' more frequently than '`,`' does. You could now produce some randomly generated *Tom Sawyer* by picking a character to begin with and then always choosing the next character based on the previous one and the probabilities revealed by the analysis. Here's an example.

Order 1

> "Shand tucthiney m?" le ollds mind Theybooure He, he s whit Pereg lenigabo Jodind alllld ashanthe ainofevids tre lin–p asto oun theanthadomoere

Now imagine doing an order-k analysis by determining the probability with which each character follows every possible sequence of characters of length k. An order 5 analysis of *Tom Sawyer*, for example, would reveal that '`r`' follows `"Sawye"` more frequently than any other character. After an order k analysis, youu'd be able to produce random *Tom Sawyer* by always choosing the next character based on the previous k characters (the *seed*) and the probabilities revealed by the analysis.

At only a moderate order of analysis (say, orders 5-7), the randomly generated text begins to take on many of the characteristics of the source text. It probably won't make complete sense, but you'll be able to tell that it was derived from *Tom Sawyer* as opposed to, say, *The Sound and the Fury.* Here are some more examples.

Order 2

> "Yess been." for gothin, Tome oso; ing, in to weliss of an'te cle – armit. Papper a comeasione, and smomenty, fropeck hinticer, sid, a was Tom, be suck tied. He sis tred a youck to themen

Order 4

> en themself, Mr. Welshman, but him awoke, the balmy shore. I'll give him that he couple overy because in the slated snufflindeed structure's kind was rath. She said that the wound the door a fever eyes that WITH him.

Order 6

> people had eaten, leaving. Come – didn't stand it better judgment; His hands and bury it again, tramped herself! She'd never would be. He found her spite of anything the one was a prime feature sunset, and hit upon that of the forever.

Order 8

> look-a-here – I told you before, Joe. I've heard a pin drop. The stillness was complete, how- ever, this is awful crime, beyond the village was sufficient. He would be a good enough to get that night, Tom and Becky.

Order 10

> you understanding that they don't come around in the cave should get the word "beauteous" was over-fondled, and that together" and decided that he might as we used to do – it's nobby fun. I'll learn you."

2 Overview of Assignment

Your task is to write a program to automate this laborious task in a more efficient way - Shannon's brute-force approach is prohibitively slow when the size of the input text is large.

2.1 Implementation

Begin by implementing the following class:

```
class RandWriter {
 public:
    // Create a Markov model of order k from given text
    // Assume that text has length at least k.
    RandWriter(const string& text, size_t k);

    size_t  orderK() const; // Order k of Markov model

    // Number of occurences of kgram in text
    // Throw an exception if kgram is not length k
    int freq(const string& kgram) const;
    // Number of times that character c follows kgram
    // if order=0, return num of times that char c appears
    // (throw an exception if kgram is not of length k)
    int freq(const string& kgram, char c) const;

    // Random character following given kgram
    // (throw an exception if kgram is not of length k)
    // (throw an exception if no such kgram)
    char kRand(const string& kgram);

    // Generate a string of length L characters by simulating a trajectory
    // through the corresponding Markov chain.  The first k characters of
    // the newly generated string should be the argument kgram.
    // Throw an exception if kgram is not of length k.
    // Assume that L is at least k
    string generate(const string& kgram, size_t L);
};
// Overload the stream insertion operator << and display the internal state
// of the Markov model.  Print out the order, alphabet, and the frequencies
// of the k-grams and k+1-grams
```

2.2 Details

2.2.1 Constructor

To implement the data type, create a symbol table, whose keys will be `string` k-grams. You may assume that the input text is a sequence of characters over the ASCII alphabet `char` so that all values are between 0 and 127. The value type of your symbol table needs to be a data structure that can represent the frequency of each possible next character. The frequencies should be tallied as if the text were **circular** (i.e., as if it repeated the first k characters at the end).

2.2.2 Order

The `orderK()` returns the order k of the Markov model.

2.2.3 Frequency

There are two frequency methods

- `freq(kgram)` returns the number of times the k-gram was found in the original text.
- `freq(kgram, c)` returns the number of times the k-gram was followed by the character c in the original text.
- For either frequency method, when the k-gram is not found, return 0.

2.2.4 Randomly generate a character

Return a character. It must be a character that followed the k-gram in the original text. The character should be chosen randomly, but the results of calling `kRand(kgram)` several times should mirror the frequencies of characters that followed the k-gram in the original text.

You may not use the C `<random.h>` library since it is known to produce low-quality random numbers, among other reasons. You should use classes in the C++ `<random>` library instead. However, you should not use `random_device` as your primary source of randomness as this is a limited shared resource shared across all processes on the machine and you do not need true randomness. **Doing either will be penalized.**

2.2.5 Generate pseudo-random text

Return a `string` of length L that is a randomly generated stream of characters whose first k characters are the argument `kgram`. Starting with the argument `kgram`, repeatedly call `kRand()` to generate the next character. Successive k-grams should be formed by using the most recent k characters in the newly generated text.

You will need to use the `<random>` library to generate random numbers for you to use. Do not use the `rand()` function from the C `<random.h>` library; it generates much lower quality random numbers then the generators in the C++ `<random>` library.

To avoid dead ends, treat the input text as a **circular** string: the last character is considered to precede the first character.

2.3 Additional notes

Consider the behavior of a 0-order model. This model will generate new characters with a distribution proportional to the ratio they appear in the input text. This model is context-free; it does not use an input `kgram` (representing the current state of the model) when generating a new character. As such:

- The `freq(string kgram)` method takes an empty string as its input `kgram` (length of `kgram` must equal the order of the model).
- This method call should produce as a result the length of the original input text (given in the constructor).

- The `freq(string kgram, char c)` also will take an empty string as input. It should produce as output the number of times the character c appears in the original input text.

Consider using a C++ `map` (`http://www.cplusplus.com/reference/map/map/`) to store frequency counts of each k-gram and $k+1$-gram as it's encountered when you traverse the string in the constructor.

Make sure to wrap around the end of the string during traversal, as described above.

At construction time, it is highly recommended to build up a string that represents the active alphabet and store it as a private class member variable. This will be useful when implementing the `kRand` method, so that you can produce all of the $k+1$-grams that will follow the provided k-gram, test for the frequency of each $k+1$-gram, and then produce output characters with proportional frequencies.

Overload the stream insertion operator to display internal state of the Markov model. You should print out all of the k-gram and $k+1$ gram frequencies, the order of the model, and its alphabet. See the attached fiel for sample instructions for how to build an iterator over the map of k-gram and $k+1$-gram keys.

2.4 Text Generator

Write a client program `TextWriter` that takes two command-line integers k and L, reads the input text from standard input, and builds a Markov model of order k from the input text. Then, starting with the k -gram consisting of **the first k characters of the input text**, print out L characters generated by simulating a trajectory through the corresponding Markov chain. You may assume that the text has length at least k and that $L \geq k$.

```
./TextWriter 2 11 < input17.txt
```

3 Debugging and Testing

You must write a `test.cpp` file that uses Boost functions to verify that your code works properly. As part of this, you should use the Boost functions `BOOST_REQUIRE_THROW` and `BOOST_REQUIRE_NO_THROW` to verify that your code properly throws the specified exceptions when appropriate (and does not throw an exception when it shouldn't). As usual, use `BOOST_REQUIRE` to exercise all methods of the class.

As before, your tests will be run against reference implementations provided by the instructor. To pass the tests, your tests must pass for the correct implementation and fail for the broken implementations. The reference implementation provides only the methods specified in Section 2.1; it does not provide any additional public members.

4 Extra Credit

For extra credit, make a separate version of the program, `WordWriter`, that generates text using words instead of characters as the basis.

5 What to turn in

Your makefile should build a program named `TextWriter` and `test` and a static library `TextWriter.a`.

Submit a zip archive to Gradescope containing:

- Your `RandWriter` class (`RandWriter.cpp` and `RandWriter.hpp`)
- Your `TextWriter.cpp` file
- Your unit tests (`test.cpp`)

- The makefile for your project. The makefile should have targets `all`, `TextWriter`, `TextWriter.a`, `test`, `lint` and `clean`. Make sure that all prerequisites are correct.
- Your `Readme-TextWriter.md` that includes
  1. Your name
  2. Statement of functionality of your program (e.g. fully works, partial functionality, extra credit)
  3. Key features or algorithms used
  4. Any other notes
- Any other source files that you created.

Make sure that all of your files are in a directory named `ps6` before archiving it and that there are no `.o` or other compiled files in it.

6 Grading rubric

Feature	Points	Comment
Unit Tests	5	
	1	`kRand()`
	1	Error Checking
	2	`generate()` results
	1	Letter frequency
Autograder	28	
	2	Contains all code files and builds
	4	Constructor and `orderK()` work
	10	Both `freq()` overloads work correctly
	4	`kRand()` generates correct set of letters
	2	`kRand()` generates letters with the correct frequencies
	4	`generate()` generates correct set and number of letters
	2	Program generates a string of the correct length
Manual Grading	6	
	4	Generates reasonable texts
	2	Uses a lambda as a parameter to a function
Readme	6	
	2	Describes the underlying structure.
	2	Describes how the next letter was selected.
	2	Describes the testing decisions.
Extra Credit	3	
	+3	Write a new random-text generation program WordWriter based on words rather than characters. Mention what you did in your readme.
Penalties		
	-10	Uses the <random.h> library instead of <random>
	-5	Memory leaks or issues
	-5	Linting problems
	-3	Non-private fields
	-10%	Each day late
Total	45	