PS6: Markov Model of Natural Language

Global Sequence Alignment

We will:

- Analyze an input text for transitions between k-grams (a fixed number of characters) and the following letter.
- Produce a probabilistic 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.

Outline

In the 1948 landmark paper *A Mathematical Theory of Communication*, Claude Shannon founded the field of information theory and revolutionized 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.

Markov model of natural language

Shannon approximated the statistical structure of a piece of text using a simple 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 occurrences 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 7/17, **'c'** with probability 1/17, and **'g'** with probability 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 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 a high correlation among successive characters in a word or sentence. For example, **'w'** is more likely to be followed with **'e'** than with **'u'**, while **'q'** is more likely to be followed with **'u'** than with **'e'**.

We obtain a more refined model by allowing the probability of choosing each successive letter to depend on the preceding letter or letters. A *Markov model of order **k*** predicts that each letter occurs with a fixed probability, but that probability can depend on the previous k consecutive characters. Let a ***k**-gram* mean any string of ***k*** characters. Then for example, if the text has 100 occurrences of "**th**", with 60 occurrences of "**the**", 25 occurrences of "**thi**", 10 occurrences of "**tha**", and 5 occurrences of "**tho**", the Markov model of order 2 predicts that the next letter following the 2-gram "**th**" is '**e**' with probability 3/5, **'i'** with probability 1/4, **'a'** with probability 1/10, and **'o'** with probability 1/20.

A brute-force solution

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*. "

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.

Implementation

Begin by implementing the following class:

```
class MModel
 // Note: all of the below constructors/methods should be public.

 // create a Markov model of order k from given text
 MModel(string text, int k) // Assume that text has length at least k.

 int kOrder()                 // order k of Markov model

 // number of occurrences of kgram in text
 int freq(string kgram) // (throw an exception if kgram is not of length k)

 // number of times that character c follows kgram
 // if order=0, return num of times char c appears
 // (throw an exception if kgram is not of length k)
 int freq(string kgram, char c)

 // random character following given kgram
 // (Throw an exception if kgram is not of length k.
 //  Throw an exception if no such kgram.)
 char kRand (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(string kgram, int L)

     <<       // overload the stream insertion operator and display
             // the internal state of the Markov Model. Print out
             // the order, the alphabet, and the frequencies of
             // the k-grams and k+1-grams.
```

Details

- *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).
- *Order.* `kOrder()` returns the order *k* of the Markov Model.
- *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.
- *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.
- *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.

To avoid dead ends, treat the input text as a **circular** string: the last character is considered to precede the first character. For example, if `k = 2` and the text is the 17-character string "`gagggagaggcgagaaa`", then the salient features of the Markov model are captured in the table below:

kgram	freq	frequency of next char			probability that next char is		
		a	c	g	a	c	g
aa	2	1	0	1	1/2	0	1/2
ag	5	3	0	2	3/5	0	2/5
cg	1	1	0	0	1	0	0
ga	5	1	0	4	1/5	0	4/5
gc	1	0	0	1	0	0	1
gg	3	1	1	1	1/3	1/3	1/3
	17	7	1	9			

Note that the frequency of "ag" is 5 (and not 4) because we are treating the string as circular.

A *Markov chain* is a stochastic process where the state change depends on only the current state. For text generation, the current state is a *k*-gram. The next character is selected at random, using the probabilities from the Markov model. For example, if the current state is `"ga"` in the Markov model of order 2 discussed above, then the next character is `'a'` with probability 1/5 and `'g'` with probability 4/5. The next state in the Markov chain is obtained by appending the new character to the end of the *k*-gram and discarding the first character. A *trajectory* through the Markov chain is a sequence of such states. Below is a possible trajectory consisting of 9 transitions.

trajectory:	ga →	ag →	gg →	gc →	cg →	ga →	ag →	ga →	aa →	ag
probability for a:	1/5	3/5	1/3	0	1	1/5	3/5	1/5	1/2	
probability for c:	0	0	1/3	0	0	0	0	0	0	
probability for g:	4/5	2/5	1/3	1	0	4/5	2/5	4/5	1/2	

Treating the input text as a circular string ensures that the Markov chain never gets stuck in a state with no next characters.

To generate random text from a Markov model of order `k`, set the initial state to *k* characters from the input text. Then, simulate a trajectory through the Markov chain by performing `L-k` transitions, appending the random character selected at each step. For example, if `k = 2` and `L = 11`, the following is a possible trajectory leading to the output `gaggcgagaag`.

trajectory:	ga	→ ag	→ gg	→ gc	→ cg	→ ga	→ ag	→ ga	→ aa	→ ag
output:	ga	g	g	c	g	a	g	a	a	g

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 a null 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 `kgram` 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 `kgram`, test for the frequency of each `k+1`-gram, and then produce output characters with proportional frequencies.

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

Text Generator

Write a client program `TextGenerator` 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**, prints 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 also that `L ≥ k`.

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

Debugging and testing

- You should write a `test.cpp` file that uses 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.
- Make sure you throw `std::runtime_error`'s in the methods per the spec.
- Use Boost Unit testing to test your implementation by using the next sequence: "`gagggagaggcgagaaa`"
- **Your test file must exercise all methods of your `MModel` class.**

Additional Files

Produce and turn in a `Makefile` which must build an executable named `TextGenerator.`

Turn in a `ps6-readme.txt` file that explains what you have done.

Submit your work

You should be submitting at least five files:

1. `MModel.cpp`
2. `MModel.h`
3. `test.cpp`
4. `Makefile`
5. `TextGenerator.cpp`
6. `ps6-readme.txt`

Submit the archive on Blackboard.

Grading rubric

Feature	Value	Comment
`MModel` implementation	4	full & correct implementation = 4 pts; nearly complete = 3pts; part way=2 pts; started=1 pt
Text Generation implementation	2	full & correct implementation = 2 pts; part way =1 pt; not =0 pt
`Makefile`	2	`Makefile` included
your own `test.cpp`	4	should test ALL Functions in MModel
`cpplint`	2	Your source files pass the style checks implemented in `cpplint`
Exceptions implementation	2	full & correct implementation=2 pts; part way=1 pt; not=0 pt
`readme`	4	Readme should say something meaningful about what you accomplished 1 point for explaining how you tested your implementation 1 point for explaining the exceptions you implemented
Total	**20**	