# Group Problem:  Random language generation using models of English text

In the past few decades, computers have revolutionized student life. In addition to providing no end of entertainment and distractions, computers also have also facilitated much productive student work. However, one important area of student labor that has been painfully neglected is the task of filling up space in papers, Ph.D. dissertations, grant proposals, and recommendation letters with important sounding and somewhat sensible random sequences. Come on, you know the overworked TA/professor/reviewer doesn't have time to read too closely. . . .

To address this burning need, the **`random writer`** is designed to produce somewhat sensible output by generalizing from patterns found in the input text. When you're coming up short on that 10-page paper due tomorrow, feed in the eight pages you already have written into the random writer, and—*Voila!*— another couple of pages appear. You can even feed your own **`.rb`** files back into your program and have it build a new random program on demand.

## How does random writing work?

Random writing is based on an idea advanced by Claude Shannon in 1948 and subsequently popularized by A. K. Dewdney in his *Scientific American* column in 1989. Shannon's famous paper introduces the idea of a Markov model for English text. A ***Markov model*** is a statistical model that describes the future state of a system based on the current state and the conditional probabilities of the possible transitions. Markov models have a wide variety of uses, including recognition systems for handwriting and speech, machine learning, and bioinformatics. Even Google's PageRank algorithm has a Markov component to it. In the case of English text, the Markov model is used to describe the possibility of a particular character appearing given the sequence of characters seen so far. The sequence of characters within a body of text is clearly not just a random rearrangement of letters, and the Markov model provides a way to discover the underlying patterns and, in this case, to use those patterns to generate new text that fits the model.

First, consider generating text in total randomness. Suppose you have a monkey at the keyboard who is just as likely to hit any key as another. While it is theoretically possible—given enough monkeys, typewriters, and ages of the universe—that this sort of random typing would produce a work of Shakespeare, most output will be gibberish that makes pretty unconvincing English:

**No Model:**

```
aowk fh4.s8an zp[q;1k ss4o2mai/
```

A simple improvement is to gather information on character frequency and use that as a weight for choosing the next letter. In English text, not all characters occur equally often. Better random text would mimic the expected character distribution. Read some input text (such as Mark Twain's *Tom Sawyer,* for example) and count the character frequencies. You'll find that spaces are the most common, that the character **`e`** is fairly common, and that the character **`q`** is rather uncommon. Suppose that space characters represent 16% of all characters in *Tom Sawyer*, **`e`** just 9%, and **`q`** a mere .04% of the total. Using these weights, you could produce random text that exhibited these same probabilities. It wouldn't have a lot in common with the real *Tom Sawyer*, but at least the characters would tend to occur in the proper proportions. 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
```

This is an ***order-0 Markov model,*** which predicts that each character occurs with a fixed probability, independent of previous characters.

We're getting somewhere, but most events occur in context. Imagine randomly generating a year's worth of Fahrenheit temperature data. A series of 365 random integers between 0 and 100 wouldn't fool the average observer. It would be more credible to make today's temperature a random function of yesterday's temperature. If it is 85 degrees today, it is unlikely to be 15 degrees tomorrow. The same is true of English words: If this letter is a q, then the following letter is quite likely to be a u. You could generate more realistic random text by choosing each character from the ones likely to follow its predecessor.

For this, process the input and build an order-1 model that determines the probability with which each character follows another character. It turns out that **`s`** is much more likely to be followed by **`t`** than **`y`** and that **`q`** is almost always followed by **`u`**. You could now produce some randomly generated *Tom Sawyer* by picking a starting character and then choosing the character to follow according to the probabilities of what characters followed in the source text. Here's some output produced from an order-1 model:

**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
```

This idea extends to longer sequences of characters. An order-2 model generates each character as a function of the two-character sequence preceding it. In English, the sequence **`sh`** is typically followed by the vowels, less frequently by `r` and `w`, and rarely by other letters. An order-5 analysis of *Tom Sawyer* reveals that `leave` is often followed by `s` or space but never `j` or `q`, and that `Sawye` is always followed by `r`. Using an order-*k* model, you generate random output by choosing the next character based on the probabilities of what followed the previous *k* characters in the input text. This string of characters preceding the current point is called the ***seed***.

At only a moderate level of analysis (say, orders 5 to 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, *Pride and Prejudice*. At even higher levels, the generated words tend to be valid and the sentences start to scan. 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:**

```
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."
```

**A sketch of the random writer implementation**

Your program is to read a source text, build an order-*k* Markov model for it, and generate random output that follows the frequency patterns of the model.

First, you prompt the user for the name of a file to read for the source text and reprompt as needed until you get a valid name. (And you probably have a method like this lying around somewhere.) Now ask the user for what order of Markov model to use (a number from 1 to 10). This will control what seed length you are working with.

Reading the text file into Ruby will result in a large String. Look at each character of that String and as you go, track the current seed and observe what follows it. Your goal is to record the frequency information in such a way that it will be easy to generate random text later without any complicated manipulations.

Once the reading is done, your program should output 2000 characters of random text generated from the model. For the initial seed, choose the sequence that appears most frequently in the source (e.g., if you are doing an order-4 analysis, the four-character sequence that is most often repeated in the source is used to start the random writing). If there are several sequences tied for most frequent, any of them can be used as the initial seed. Output the initial seed, then choose the next character based on the probabilities of what followed that seed in the source. Output that character, update the seed, and the process repeats until you have 2000 characters.

For example, consider an order-2 Markov model built from this sentence from Franz Kafka's *Metamorphosis:*

```
As Gregor Samsa awoke one morning from uneasy dreams he found himself
transformed in his bed into a gigantic insect.
```

Here is how the first few characters might be chosen:

- The most commonly occurring sequence is the string `"in"`, which appears four times. This string therefore becomes the initial seed.
- The next character is chosen based on the probability that it follows the seed `"in"` in the source. The source contains four occurrences of `"in"`, one followed by `g`, one followed by `t,` one followed by `s`, and one followed by a space. Thus, there should be a 1/4 chance each of choosing `g`, `t`, `s`, or space. Suppose space is chosen this time.
- The seed is updated to `"n "`. The source contains one occurrence of `"n "`, which is followed by `h`. Thus the next character chosen is `h`.
- The seed is now `" h"`. The source contains three occurrences of `" h"`, once followed by `e`, and twice followed by `i`. Thus, there is a 1/3 chance of choosing `e` and 2/3 for `i`. Imagine `i` is chosen this time.
- The seed is now `"hi"`. The source contains two occurrences of `"hi"`, once followed by `m`, the other by `s`. For the next character, there is 1/2 chance of choosing `m` and 1/2 chance for `s`.
- If your program ever gets into a situation in which there are no characters to choose from (which can happen if the only occurrence of the current seed is at the exact end of the source), your program can just stop writing early.

**A few implementation hints**
Although it may sound daunting at first glance, this task is supremely manageable with the bag of power tools you bring to the job site.

- **Hashes** and **Arrays** are just what you need to store the model information. The keys into the Hash are the possible seeds (e.g., if the order is 2, each key is a 2-character sequence found in the source text). The associated value is an Array of all the characters that follow that seed in the source text. That Array can—and likely will—contain a lot of duplicate entries. Duplicates represent higher probability transitions from this Markov state. Explicitly storing duplicates is the easiest strategy and makes it simple to choose a random character from the correct frequency distribution. A more space-efficient strategy would store each character at most once, with its frequency count. However, it's a bit more awkward to code this way. You are welcome to do either, but if you choose the latter, please take extra care to keep the code clean.

- Determining which seed(s) occurs most frequently in the source can be done by iterating over the entries once you have finished the analysis. This means first build a Hash as described above and only then determine the key with the largest Array of characters.

- Store the return value of Ruby's `File.read("your_filename.txt")` to a variable that will hold the text as a Ruby String.

- Really take your time considering the best iterator method to use. You may want to re-read the documentation for Ruby methods you might be less familiar with like

**`each_with_index`** or **`each_with_object`**. Also do not neglect the plain old **`while`** loop used with a **`counter`** or **`index`** variable.

- This will be challenging so do not try to achieve the most elegant or most optimized solution on your first pass. The idea is get it to work ***and then*** make it nice while ensuring that ***it still works***.

- If you want to select a random element out of an Array you could generate a random integer between 0 and the array's length (after subtracting one) and then access the array at that random index. Ruby provides a shortcut for doing this with the **`sample`** method. Look it up if you are not familiar.

**Random writer task breakdown**

A suggested plan of attack that breaks the problem into the manageable phases with verifiable milestones:

- *Task 1—Try out the demo program.* Play with the demo just for fun and to see how it works from a user's perspective.

- *Task 2—Review the collection classes you've been learning thus far.* The primary tools you need for this problem are Arrays and Hashes. Once you're familiar with what functionality these collection classes provide, you're in a better position to figure out what you'll need to do for yourself.

- *Task 3—Design your data structure*. Think through the problem and map out how you will store the analysis results. It is vital that you understand how to construct the nested arrangement of String/Array/Hash objects that will properly represent the information. Since the Hash will contain values that are Arrays, iterating over a nested data structure is in your future.

- *Task 4—Implement analyzing source text.* Implement the reading phase. Be sure to develop and test incrementally. **Work with small inputs first**. Verify your analysis and model storage is correct before you move on. There's no point in trying to generate random sentences if you don't even have the data read correctly!

- *Task 5—Implement random generation.* Now you're ready to randomly generate. Since all the hard work was done in the analysis step, generating the random results is straightforward.

You can run the random writer program on any sort of input text, in any language! The web is a great place to find an endless variety of input material (blogs, slashdot, articles, etc.) When you're ready for a large test case, Project Gutenberg maintains a library of thousands of full-length public-domain books. We've supplied you with files containing the text of Twain's

*Tom Sawyer,* William Shakespeare's *Hamlet,* and George Eliot's *Middlemarch*—all of which come from Project Gutenberg. At higher orders of analysis, the results the random writer produces from these sources can be surprisingly sensible and often amusing.

**Additional Hints**

One of the most challenging parts of this task will be dealing with ***edge cases.*** Some examples would be when you are at the very end of the String and there is no next character or the case where your Hash does not yet have a certain key-- you are seeing that sequence of characters for the first time.  This will have to be handled differently than when the key already exists; consider the following Hash:

```
animals_hash = {"a"=>["alligator", "armadillo", "ape"], "b"=>["bat",
"butterfly"], "c"=>["cat"]}
```

Imagine writing a method that took an Array of animal names and added them to the **`animals_hash:`**

```
def add_animals(array, animals_hash)
  array.each do |animal|
    first_letter = animal[0]
    animals_hash[first_letter] << animal
  end
end
```

This method would work fine when called with this argument:
**`add_animals([“cockatoo”, “beluga”, “capybara”], animals_hash)`**

But would fail here:
**`add_animals([“cockatoo”, “beluga”, “zebra”, “capybara”], animals_hash)`**

With the error:
**`NoMethodError: undefined method `<<' for nil:NilClass`**

Can you explain why? How could you solve this?

**References**

1. A. K. Dewdney. A potpourri of programmed prose and prosody. *Scientific American,* 122-TK, June 1989.
2. C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27, 1948.
3. Wikipedia on Markov models (http://en.wikipedia.org/wiki/Markov_chain).

4. Project Gutenberg’s public domain e-books (http://www.gutenberg.org).
5. Keith Shwarz CS106B Stanford class (http://web.stanford.edu/class/archive/cs/cs106b/cs106b.1136/)