

This assignment explores the idea of using random writing to produce text that is similar to some known piece of work. Imagine taking a book, such as *Tom Sawyer*, and determining the probability with which each character occurs. You'd probably find that spaces are the most common character, followed by the character e, etc. Given these probabilities, which we will call a Level 0 analysis, you could randomly produce text that, while not resembling English, would have the property that the characters would likely occur in the same proportions as they do in *Tom Sawyer*. For example, here's what you might produce:

Level 0 rla bsht eS ststfo hhfosdsdewno oe wee h .mr ae irii ela iad o r te u t mnyto onmalysnce, ifu en c fDwn oee

Now imagine doing a slightly more sophisticated analysis, a Level 1 analysis, that determines 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 a period more frequently than a comma does. With this new analysis, you could use the probabilities from *Tom Sawyer* to randomly pick an initial character and then repeatedly choose the next character based on the previous character and the probabilities provided by the analysis. Your new text might look like the following, which looks a bit more like English than the previous example:

Level 1 "Shand tucthiney m?" le oldls mind Theybooue He, he s whit Pereg lenigabo Jodind alllld ashanthe ainofevids tre lin-p asto oun theanthadomoere

We can generalize these ideas to a Level k analysis that determines the probability with which each character follows every possible sequence of k characters. For example, a Level 5 analysis of *Tom Sawyer* would show that r follows Sawye more frequently than any other character. After a Level k analysis, you'd be able to produce random text by always choosing the next character based on the previous k characters – which we will call the *seed* – and based on the probabilities produced by your analysis.

For relatively small values of k (5-7), the randomly generated text begins to take on many of the characteristics of the source text. While it still will not produce legal English, you will be able to tell that it was derived from *Tom Sawyer* instead of *Harry Potter*. As the value of k increases, the text looks increasingly like English. Here are some more examples:

Level 2 "Yess been." for gothin, Tome oso; ing, in to weliss of an'te cle - armit. Paper a comeasione, and smomenty, fropech hinticer, sid, a was Tom, be such tied. He sis tred a youck to themen

Level 4 en themself, Mr. Welshman, but him awoke, the balmy shore. I'll give him that he couple overy because in the slated snuffindeed structure's kind was rath. She said that the wound the door a fever eyes that WITH him.

Level 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.

Level 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.

Level 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."

1 Your Assignment

You should implement a public Java class, `RandomWriter`, that implements a random writing application. In particular, your class should implement the `TextProcessor` interface, available from the class web page, which separates the task into two methods, one for reading the text and one for writing the result:

```
void readText(String inputFilename, int level);
void writeText(String outputFilename, int length);
```

Your main method, `public static void main(String[] args)`, should accept four arguments:

- `args[0]` - the input filename (`source`)
- `args[1]` - the output filename (`result`)
- `args[2]` - the level of analysis (`k`)
- `args[3]` = the length of output (`length`)

Your main routine should validate the command line arguments by making sure `k` and `length` are non-negative, that `source` contains more than `k` characters and can be opened for reading, and that `result` can be opened for writing. If any of the command line arguments is invalid, your program should write an informative error message to `System.err` before terminating.

If the command line arguments are legal, the `readText()` method should be invoked to read the specified file and initiate the Random Writing process by selecting a random initial seed (that is, `k` consecutive characters) from the source file. Your program should then use the `writeText()` method to write `length` characters to `result`. Each of the `length` characters should be chosen based on the current seed. Every time a character `c` is written to `result`, the seed should be updated by removing the seed's first character and appending `c` to the end. For example, suppose that $k = 2$ and the source file consists of the following:

```
the three pirates charted that course the other day
```

The first three characters might be chosen as follows:

- Two consecutive characters are randomly chosen to form the initial seed. let's suppose that `th` is chosen.
- The first character must be chosen according to the probability that it follows the seed in the source. In our example, the source file contains five occurrences of `th`. Three of these times it is followed by `e`, once it is followed by `r` and once it is followed by `a`. Thus, the first character must be chosen so that there is a $\frac{3}{5}$ chance that an `e` will be chosen, a $\frac{1}{5}$ chance that an `r` will be chosen, and a $\frac{1}{5}$ chance that an `a` will be chosen. Let's suppose we choose an `e` this time.
- The next character must be chosen based on the probability that it follows the current seed, which in our example is now `he`. This seed is followed twice by a space and once by an `r`, so the next character should be chosen so that there is a $\frac{2}{3}$ chance that it is a space and a $\frac{1}{3}$ chance that it is an `r`. Let's suppose we choose an `r` this time.
- The next character must be chosen based on the probability that it follows the current seed, which is now `er`, in the source. Since the source contains only one occurrence of `er`, which is followed by a space, the next character that we choose must be a space.

If your program ever gets into a situation where there are no characters to choose from (which can happen if the only occurrence of the current seed is at the end of the source), your program should pick a new random seed and continue.

2 Approach

There is a simple way to do this assignment. Create a `String` object containing the full text of the source file. To choose the next character, find each occurrence of the seed in the source and store the character that follows it in some kind of list. When you have found all occurrences, randomly choose a character from the list and update the seed.

There are, of course, other ways to do this. The above approach must scan the text many many times but uses a modest amount of space. Think about other ways to implement this and what their advantages and disadvantages are.

Hints: Become comfortable with the Java API Documentation (linked to from the class web page), for this and future assignments.

- You can use a `java.util.Random` object to generate random numbers within a specified range.
- A `java.io.FileWriter` object is useful for writing strings and characters to a file.
- A `java.io.FileReader` object is useful for reading one character at a time from a file.
- A `java.lang.StringBuffer` object is useful for efficiently composing characters into a string.
- A `java.lang.String` object contains several member functions for searching for substrings.

Project Gutenberg (<http://www.gutenberg.org/>) maintains a huge library of public domain books that you can use as source texts. If your program generates something especially amusing, post it to the Discussion Group and include it in your report. Be sure to identify the source text and the level of the analysis (k).

2.1 Testing

You should think carefully about how you can test your software. To help you get started, here are a few questions that you might consider.

- In terms of black box testing, can you think of inputs that will produce outputs that you can verify through simple inspection?
- If you cannot verify the output through human inspection, can you verify your outputs mathematically? Can you write code that analyzes the output and relates it to the input?
- For this assignment, black box testing seems somewhat limited. Can you think of ways to inspect the internal state of the computation to give you greater confidence that your code is working correctly? The idea of looking *inside* the module (ie, inside the black box) to perform testing is known as *white box testing*. Can you facilitate white box testing by creating some new methods whose interfaces make them easier to test than the existing methods? (Hint: Think about what it is that makes this assignment more difficult to test than Assignment 1, and then think about how you can remove this aspect from this assignment.)
Once you look inside the source code, can you find ways to test the individual components of your code? This type of testing is known as *unit testing*.
- Are there specific *corner cases* that you can think of that you should test?

3 Your Report

Unless specified otherwise, all of your reports should take the same format as that of the first assignment.

4 Karma

If you have extra time, there are many interesting things that you can do with Markov process random generation. Try one of the following or come up with your own. The second one is particularly interesting.

- The existing algorithm works at the character level. You can also make it work at the word level. That is, instead of calculating the probability of each letter after k letters, calculate the probability of each word in the text occurring after k words. This tends to produce much more English-like text for small k , since it only uses words found in the text itself and “picks up” on common phrases. Implement this in a separate `RandomWordWriter` class.
- **Really cool stuff!** This algorithm need not be restricted to one dimension. In the late 90’s this technique was used to remove objects from images by artificially filling in the missing background. Adobe Photoshop now provides a feature known as *Content-Aware Fill*, which is based on this basic idea. The algorithm creates amazingly realistic continuations of a scene—ocean waves, sky, forests, mountains, pebbles, etc. A level of

analysis, k , matches all existing pixels in a $k \times k$ neighborhood of a target pixel with $k \times k$ tiles from elsewhere in the same image.

To implement this algorithm, you might adapt Assignment #1 so that it loads two images—the primary image and a mask representing a foreground object that you’d like to remove. Alternatively, for fun you could load in a single image or texture and then just create an entirely new random image using the same statistics. Comment on how k , the size of the neighborhood, affects the image manipulation. To learn more about this, see the early papers by Criminisi [1] and Efros [3], and see Efros’ web page:

<http://graphics.cs.cmu.edu/people/efros/research/EfrosLeung.html>.

- The same technique can be applied to music. Music is somewhat trickier, as you have both notes and times to worry about, and if the beats aren’t right the piece will sound funny. If you don’t have time to handle these details, you might want to see what happens if you don’t worry about the timing of the notes.

Java provides MIDI handling functions in `javax.sound.midi`. The `MidiSystem` class provides a `write` method that produces MIDI files, as well as `getFileFormat` and `getSequence` functions to read in MIDI files. Find some large MIDI files on the web (classical music is good) and generate short overtures or sonatas. Implement this in a separate `RandomMusicWriter` class.

Be aware that this can take a significant amount of time if you do choose to worry about the details.

5 What To Turn In

You should electronically submit a single JAR file (`prog2.jar`) containing all the source files necessary to build your project, your report (in ASCII text or PDF), and any particularly amusing bits of text.

If you use source texts other than the provided texts and the Project Gutenberg texts, include the URL of the text in your report. Do not submit large source texts with your assignment.

The deadline is **5:00pm** on the due date.

6 More background

If you’re mathematically inclined, you might recognize this process as a Markov Chain. A (discrete) Markov Chain is a sequence where each value depends only upon the previous value (or k previous values). That is, the generation of the next element in the sequence is not affected by the past – it only depends on a fixed number of immediately prior elements. More precisely, given random variables X_0, X_1, \dots and an order k (which we’ve been calling *level*), a Markov Chain is a sequence with the property

$$P(X_{n+1}|X_0, X_1, \dots, X_n) = P(X_{n+1}|X_n, \dots, X_{n-k}) \quad (1)$$

A Markov Chain is controlled by a *transitional probability* distribution, which is the conditional probability $P(X_{n+1}|X_n, \dots, X_{n-k})$ from (1). In this assignment, the transitional probability is fixed by the original source text and does not change for the duration of the process.

Markov Chains have applications in a number of areas, such as population modeling and queueing theory. Also, the American poet Jeff Harrison has published actual poetry generated by similar processes. More information is available online [4].

Acknowledgments. This assignment was originally created by Joe Zachary of the University of Utah. Walter Chang and Jeff Diamond have since added interesting extensions to this assignment. The assignment is based on the idea of Claude Shannon in 1948 [5] and popularized by A. K. Dewdney in *Scientific American* [2].

References

- [1] A. Criminisi, P. Perez, K. Toyama. Object Removal by Exemplar-based Inpainting, *IEEE Computer Vision and Pattern Recognition*, 2003.

- [2] A. K. Dewdney. A potpourri of programmed prose and posody. *Scientific American*, June 1989.
- [3] A. A. Efros and T. K. Leung, Texture Synthesis by Non-parametric Sampling, *IEEE International Conference on Computer Vision*, 1999, pp. 1033-1038.
- [4] Markov Chains. *Wikipedia*, http://en.wikipedia.org/wiki/Markov_chain
- [5] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 1948.