Lab 2: Infinite Monkey Theorem
CSCI 2101 – Fall 2018

Due: Wednesday, September 26, 11:59 pm
Collaboration Policy: Level 1 (review full policy for details)
Group Policy: Pair-optional (you may work in a group of 2 if you wish)

The *infinite monkey theorem* states that a monkey typing at random on a typewriter will eventually (given enough time) produce the complete works of William Shakespeare (or any other work, for that matter). Of course, the length of time one would have to wait for that to actually occur is rather prohibitive – that is, the universe would probably end first.

While we don't have time to test the theorem in its original form, in this lab, you will write a program simulating a somewhat more clever hypothetical typewriter monkey. This particular monkey has been browsing some books by well-known authors and remembers how often certain letter sequences appear (though it does not understand the actual meaning of letters or words). Rather than typing purely randomly, the monkey tries to ape great authors[1] by mimicking patterns it has seen before. The monkey is still unlikely to produce *Hamlet*, but might at least be able to produce something that could pass as Shakespearean.

This lab will give you experience writing programs using multiple classes, working with maps (aka dictionaries), and using *generics* in Java. Note that while we will be using maps this week, we won't actually talk about how maps are implemented until later in the semester. You should read through the entire handout (particularly Sections 1, 2, and 3) before proceeding with the lab. Remember to *plan* your classes and methods before beginning to code!

Warning! This lab is significantly more complex from a design and object-oriented perspective than Lab 1. While the code needed is actually not as extensive as one might guess from the writeup, the lab is not to be underestimated. Start early and work steadily!

1 (Pseudo)-Random Writing

Consider the following three excerpts of text:

> Call me Ishmael. Some years ago–never mind how long precisely–having repeatedly smelt the spleen respectfully, not to say reverentially, of a bad cold in his grego pockets, and throwing grim about with his tomahawk from the bowsprit?
>
> Call me Ishmael. Some years ago–never mind how long precisely–having little or no money in my purse, and nothing particular to interest me on shore, I thought I would sail about a little and see the watery part of the world.
>
> Call me Ishmael, said I to myself. We hast seen that the lesser man is far more prevalent than winds from the fishery.

The second excerpt is the first sentence of Melville's *Moby Dick*. The other two were generated "in Melville's style" using a simple algorithm developed by Claude Shannon in 1948.[2] This is the algorithm you will implement in this lab.

[1] As measured by number of puns, of course...
[2] Claude Shannon, "A mathematical theory of communication", *Bell System Technical Journal*, 1948.

Character Distributions. The algorithm is based on letter probability distributions. Imagine taking the book *Tom Sawyer* and determining the probability with which each character occurs (we'll call this a *level-0* analysis). 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 analysis, you'd be able to produce random *Tom Sawyer* text based on character probabilities by just sampling one character at a time. 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:

Level 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 sophisticated *level-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. More concretely, if you are analyzing the text "the theater is their thing", then 'e' appears after 'h' three times, and 'i' appears after 'h' one time; no other letters appear after 'h.' So the probability that 'e' follows 'h' is .75; the probability that 'i' follows 'h' is .25; the probability that any other letter follows 'h' is 0.

Using a *level-1* analysis, you could produce some randomly generated *Tom Sawyer* text 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:

Level 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 a *level-k* analysis by determining the probability with which each character follows every possible sequence of characters of length k. For example, a *level-5* analysis of *Tom Sawyer* would reveal that 'r' follows "Sawye" more frequently than any other character. After such an analysis, you'd be able to produce random *Tom Sawyer* text by always choosing the next character based on the previous k characters and the probabilities revealed by the analysis.

At somewhat higher levels of analysis (levels 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, *Hamlet* or *Moby Dick*. Here are some more examples:

Level 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

Level 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.

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."

To summarize the algorithm, once we have processed the input text and stored it in a structure that allows us to check probabilities, we pick k letters (for example, the first k in the input text) to use as a beginning for our new text. We then choose subsequent characters based on the preceding k characters and the probability information, and continue for as much output text as we wish.

One special case must be considered, which is when a k-length sequence is encountered during generation that was never seen in the input text. In this case, the algorithm must decide how to proceed (since there are no examples indicating what should follow the sequence). While there are multiple possible ways to handle this problem, a simple approach is to just randomly select the next character and continue.

2 Program Interface

Your program should have a simple, terminal-based interface. The program should first prompt the user to enter the name of an input file to read:

```
Enter file to read:
```

Once the name of the input file has been entered (e.g., `hamlet.txt`), the desired value of k should be prompted and read:

```
Enter desired value of k:
```

Your program should do some basic error checking on the inputs. In particular, if the entered file can't be read, or the value of k is invalid (less than 1), your program should print an error message and then exit. As in last week's lab, you don't need to worry about the case where a non-numeric value is entered for k.

Once the filename and k have been read, the program should print out 500 characters of randomly-generated text following the probabilities of the input text (you can do more, but 500 should be enough to be confident that things are working). The first k characters of the output text should be the same as the first k characters of the input text – in other words, the first k input characters will be the starting sequence used to generate the first random character.

Important note: While the above describes the interface of the finished program, you should probably *not* use this complete interface during development. Instead, you should implement incrementally and start with something simpler, as detailed in Section 4.

3 Program Design

To start, we will focus on implementing a *level-2* analysis. That is, we will compute the next character to print based on the previous $k = 2$ characters only.

Think about the design and prepare a written design description of this program. When thinking about the design, focus on what would constitute a good data structure for this problem. Your data structure needs to keep a table of probabilities and be able to support two key operations:

1. Given a string of $k = 2$ characters and the following (third) character from the input text, update the probabilities in the table. This operation will be used when reading the input and building the table.

2. Given a string of $k = 2$ characters and using the probabilities previously computed and stored in the table, select the next (i.e., third) character to follow. This operation will be used when generating the output text.

Since both operations rely on looking up probabilities based on particular sequences (i.e., mapping keys to values), your structure will be built around `Map` objects – the Java equivalent of dictionaries in Python. You will need to develop two primary classes, as well as a third that just contains your `main` method.

The first class, which we will call a `FrequencyMap`, should store the frequency with which particular characters follow a specific length-k character sequence (where each `FrequencyMap` instance corresponds to a particular sequence). For example, the `FrequencyMap` for the sequence "Sa" will probably show that "w" is a fairly common subsequent character, while "x" is perhaps a less common subsequent character. You should use a `HashMap` to implement this structure, in which characters are mapped to the number of times that character has appeared following the `FrequencyMap`'s length-k sequence. You will need to decide what methods the frequency map needs to support and any other instance variables that might be necessary.

The second class, which we will call a `SequenceTable`, should store the `FrequencyMap` for each sequence that has been processed. Again, the best way to implement this structure is using a `Map` (think about what the keys and values in this map represent). You will need to gradually build up the `SequenceTable` as you read the input text. Afterwards, it should allow you to actually generate random text following the probabilities stored in the `FrequencyMap` objects.

Lastly, you will need to write a `main` method that actually uses the other classes to generate random text. Your `main` method should be written in a separate class called `WordGen`, which contains only the `main` method and possibly a few helper methods. In particular, your `main` method will need to read the input text, build the `SequenceTable` by repeatedly handing it character sequences, and then generate and print a randomly-generated string based on the probabilities of the input text (by repeatedly asking the `SequenceTable` to give you new random characters).

To summarize, you will need to write three separate classes: `FrequencyMap`, `SequenceTable`, and `WordGen` – the first two will be regular classes, while the last will just be a container for your `main` method. You should plan the design of each class in advance.

4 Implementation Tips

You should build your program in stages that you have planned out **ahead of time**. A well-designed program is significantly easier to write, whereas if you neglect planning and just start to code, you will likely end up with a messy, over-complicated program. Relatedly, it's good idea to simplify the problem at first while you develop, then generalize once the simpler version is working. Here are two specific suggestions along those lines:

- Rather than trying to read input from a file at first, just hardcode a `String` constant to use as the input (e.g., "the theater is their thing"). While set up this way, your program doesn't need to prompt for a filename input at all.
- Similarly, rather than handling an arbitrary value of k to start, just fix a value of k (e.g., 2) and get the program working with that k. In this case, your program also doesn't need to prompt for a value of k.

Once your program is working for a hardcoded input text (try a few different strings) and fixed k, you can remove those restrictions by implementing the full interface described in Section 2. If you have designed your classes well, moving to a general input and k should require minimal changes beyond updating your `main` method to provide the complete interface.

Some other useful implementation tips (in no particular order) are provided below:

- Use the `Scanner` class to handle user input. You can read an entire line of text entered by the user with the `nextLine` method.
- Remember to import `java.util.HashMap` in order to use the `HashMap` class. If you are using the `Map` interface as well, you will also need to import `java.util.Map`;
- Be sure to handle the special case where you encounter a sequence that has no known successor character in a reasonable way. If you follow the approach suggested in Section 1, you will need a way to generate a random character. Here is such a method:

```
// a reusable random number generator
private static final Random rand = new Random();

/**
 * Generate a random letter from 'a' to 'z'.
 * @return The random character.
 */
private static char randomChar() {
  return (char) (rand.nextInt(26) + 'a');
}
```

  In this method, the "`(char)`" is a *cast*, which basically means we're taking a value of one type and forcing Java to interpret it as a different type. Here, we're generating a random integer (corresponding to some letter, since a `char` is nothing more than a number that's preassigned to a particular textual character), and then telling Java to turn it into its corresponding `char`.

Shrewd students will observe that the value `26` ought to be a constant. Do that if you choose to use this method!

- When using generics (i.e., specifying the types that collections such as `Maps` are going to work with), remember that you have to specify reference types, and cannot use primitive types like `int`, `char`, etc. However, since you might reasonably want to store primitive types in a `Map` or `List`, Java provides a full-blown class for each of the primitive types, named by the unabbreviated primitive type. For example, there is a class `Integer` corresponding to an `int`, a class `Character` corresponding to a `char`, and so forth. You can use these class names when specifying your generic types. Luckily, Java will take care of converting between the primitive type and its reference type automatically (in a process called *autoboxing* and *unboxing*), so you can use the primitive types as normal when actually working with your `Map` objects. For our purposes, the only time you need to worry about the corresponding reference classes is when you're declaring or assigning your maps.

- Reading the contents of a file into a `String` (or doing much of anything involving files) is unfortunately more complex in Java than in Python. While there are multiple ways to accomplish this in Java, below is a relatively compact method you can use:

```
/**
 * Read the contents of a file into a string. If the file does not
 * exist or cannot not be read for any reason, returns null.
 * @param filename The name of the file to read.
 * @return The contents of the file as a string, or null.
 */
private static String readFileAsString(String filename) {
  try {
    return new String(Files.readAllBytes(Paths.get(filename)));
  } catch (IOException e) {
    return null;
  }
}
```

  Feel free to take and use this method verbatim in your program. To do so, you will also need to add a few extra imports:

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
```

- A simple but useful debugging approach is printing out your objects to inspect their state (such as the the frequency maps). To get more useful output from printing your objects, make sure to define a `toString` method in your classes. These don't need to be complex – they might even just call the `toString` method of the `Map` class, which is already defined and shows you the key-value pairs in the map.

- If you finish the program early and want to go a bit further, you can change your program to work at the word level instead of the character level (to be clear, this is strictly optional). Only attempt this after you get the required work finished (and make a backup copy of the character-level analysis). Does this make the results better/worse in any way?

5 Sample Texts

A collection of sample input files files have been provided and linked from the class web page. Feel free to download these files and use them as test inputs to your program. The file `whosonfirst.txt` is a good file to try processing first. The others are significantly larger and not quite as predictable. If you want to try something else, the Gutenberg Project (`https://www.gutenberg.org/`) has thousands of books available for download as plain text files.

Note that when specifying the input filename to your program, just typing a name like "`hamlet.txt`" will cause your program to look for the file called `hamlet.txt` *in the same directory as your program*; i.e., the directory containing `WordGen.java`, `package.bluej`, etc. If you specify a name like "`text-files/hamlet.txt`", then the program will first look for a directory named `text-files` within your program directory, then look for `hamlet.txt` within that directory.

6 Evaluation

Your completed program will be graded on correctness, design, and style. Make sure that your program is documented appropriately and is in compliance with the Coding Design & Style Guide. Also make sure that your name (and the name of your partner, if applicable) is included in all of your Java files.

!!! Important !!! Points may be deducted if your name is missing from your files or if your project directory is improperly named (see below for naming instructions).

Group Evaluations

For groups, only one group member should submit the final program (but make sure that both names are given).

In addition to your group's single submission, **each group member must individually submit a group report to me by email**. Your group report, which will be kept anonymous from your partners, should summarize your contributions to the lab as well as those of your partner(s) (which could be as simple as "we both worked on the entirety of the lab together in front of one machine"). Your group report does not need to be long (a few lines is fine), but **it must be received for your lab to be considered submitted**.

Group submissions will normally receive a single grade, but we reserve the right to adjust individual grades up or down in the event of a clearly uneven distribution of work.

7 Submitting Your Program

Once your program is finished, you should follow the following steps to submit:

1. Save your program and **quit BlueJ** (this is necessary because BlueJ gets confused if you perform step 2 – renaming your project directory – while the project is open).

2. Rename your project folder (which is the folder that contains your `.java` files, `package.bluej`, and possibly a few other files) so that it is named `username-lab2` (with your actual username). For example, I would rename my folder `sbarker-lab2`. **For groups, use both usernames separate by a dash (e.g., `sbowdoin-jbowdoin-lab2`)**.

3. Create a single, compressed `.zip` archive of your project folder. On a Mac, right-click (or, if you have no right mouse button, control-click) on your project folder and select "Compress your-folder-name" from the menu that appears. On a Windows machine, right-click on the folder, select "Send To," and then select "Compressed (zipped) Folder." In either case, you should now have a `.zip` file that contains your project, named something like `sbarker-lab2.zip` (with your actual username).

4. Open a web browser and go to the course's Blackboard page, then browse to `Lab Submissions`. Click on `Lab 2` and then `Start New Submission`. In Section 2, you can, but do not need to, provide any comments. Then select `Browse My Computer` and browse to the `.zip` file you created in step 3. Select that file, then click on `Submit`.

5. If working in a group, submit your group report to me by email.

After submitting your lab, remember to save a copy of your project folder somewhere other than on the desktop of the machine you are working on (if you're on a lab machine). If you just leave it on the desktop, it will only be available on that machine – if you log into any other machine on campus, it will not be there. You can also store your projects in Dropbox (or any similar service) or in your folder on the `microwave` server (see Lab 1 writeup for details on connecting to `microwave`).