Lab 2: Infinite Monkey Theorem
CSCI 2101 – Fall 2021

Due (Section A): Wednesday, September 29, 11:59 pm
Due (Section B): Thursday, September 30, 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 given enough time, a monkey typing randomly on a typewriter will eventually produce the complete works of William Shakespeare (or any other literary work, for that matter). Of course, the length of time one would have to wait for that to occur is rather long; e.g., the universe would probably end before it actually happened. Although we don't have time to test the theorem in its original form, we will test a modified theorem involving a more clever typewriter monkey. This particular monkey has been browsing books by well-known authors and remembers how often certain letter sequences appear. Rather than typing purely randomly, the monkey tries to mimic great authors by repeating patterns it has seen before (though it does not understand the actual meaning of anything that it types). The monkey may still not produce *Hamlet* itself, but might at least be able to produce something passing as Shakespearean.

More practically, 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, this 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 Herman Melville's *Moby Dick*. The other two excerpts were generated "in Melville's style" using a simple algorithm[1] developed by Claude Shannon in 1948. In this lab, you will implement Shannon's algorithm, allowing you to programmatically generate text in the style of real authors!

[1]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 analysis by determining the probability with which each character follows every other character – we'll call this a *level-1* analysis. You would probably discover that 'h' follows 't' more frequently than 'x' does, and that a space follows '.' more frequently than ',' does. For example, if you are analyzing the text "the theater is their thing" and considering the letter 'h', then 'e' appears after 'h' three times, 'i' appears after 'h' one time, and no other letters ever appear after 'h.' So the probability that 'e' follows 'h' is 0.75 (75%); the probability that 'i' follows 'h' is 0.25 (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 some character to begin with and then repeatedly choosing the next character based on the previous one and the probabilities revealed by the original text 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 (e.g., 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 *Tom Sawyer* 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: given some input text (e.g., the text of *Tom Sawyer*) and the level k of the desired analysis, we first process the input text and store the probabilities of every possible character that follows each k-length sequence encountered in the input text. Following this analysis, we can generate random text as follows: first, pick the first k letters from the input text to bootstrap the random text. Then, repeatedly choose the next character by looking at the preceding k characters in the random text and selecting randomly given the probability information from the input text analysis. We can continue to select random characters in this way to generate as much output text as desired.

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 can disregard 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 *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

As always, you should spend some time sketching the design (preferably on paper) before actually starting to code. A good way to mentally approach the problem is to think in terms of the two stages of the program: first, processing the input text to calculate the probability information (the first stage), and second, using that probability information to generate random text (the second stage). When thinking about your operations, it may be helpful to explicitly think about whether they will be needed in the first stage or the second stage. At a high level, your program will need to repeatedly perform the following two operations:

1. Given a string of k characters and the following $(k+1)$ character from the input text, update the probabilities in your probability table. This operation will be used when reading the text input and building the table (i.e., the first stage).
2. Given a string of k characters and using the probabilities previously computed and stored in the table, select the next (i.e., $k+1$) character to follow in the generated text. This operation will be used when generating the output text (i.e., the second stage).

Since both operations rely on looking up probabilities based on particular character sequences (i.e., mapping keys to values), your structure will be built around `Map` objects (aka dictionaries). 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 various characters follow a specific length-k character sequence (where each `FrequencyMap` instance corresponds to a particular sequence). For example, in a *level-2* analysis, 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. The primary piece of state within a `FrequencyMap` should be a (regular) `Map` in which characters are mapped to the number of times that character appears following the `FrequencyMap`'s length-k sequence. You will need to decide what methods a `FrequencyMap` needs to support and any other instance variables that might be necessary.

Importantly, remember that a `FrequencyMap` stores the frequencies of characters that follow **a specific k-length sequence**. Since there will be lots of different k-length sequences in the input text, this means that you will be creating lots of different `FrequencyMap` objects! The multiple `FrequencyMap` objects will be contained in and managed by the second class, as described below.

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`. You will need to think about what the keys and values in this map represent – remembering that the `SequenceTable` is going to contain all of your `FrequencyMap` objects. 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.

Since a single `SequenceTable` contains all the `FrequencyMap` objects, note that your program will only need to create a single `SequenceTable` object. This object will then serve as the "primary" data structure for the program (though it internally relies on all of the `FrequencyMap` objects).

One particular complication to note is any situation in which you generate a random k-length sequence that has no known following characters. In particular, this situation can arise if your input ends in a k-length sequence that appears nowhere else in the input. In this case, if you randomly generate this particular k-length sequence, the program will have no probability information to decide what character should follow it. A straightforward fix to this problem is to *prepend* (i.e., add to the beginning) the final k-length sequence of the input to itself before processing. That way, there will never be a unique k-length sequence appearing at the end of the input.

Lastly, you will need to write a `main` method that actually uses the `SequenceTable` class 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. 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 briefly recap, 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. Specific implementation tips are provided below.

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 a good idea to simplify the problem at first while you develop, then generalize once the simpler version is working. Here are two specific suggestions to simplify the problem while you are just getting started:

- While the full program interface will read input from a specified file, don't bother trying to read from a file at first. Instead, just hardcode a specific `String` value into your program to use as a fixed input during development (e.g., "the theater is their thing"). When set up this way, your program doesn't need to prompt for a filename input at all. It will be simple to go back when everything is working and change the interface to prompt for a filename.
- 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. However, make sure that the design of your classes is general enough to be able to handle settings of k other than your initial hardcoded value.

Once your program is working for a hardcoded input string and a fixed k, try varying one and then both values to make sure that your program works for multiple different inputs. Afterwards, you can implement the full interface described in Section 2, which will allow a user to specify any input file and desired setting of k. If you have designed your classes well, moving to a general input and k should require few changes beyond updating your `main` method to provide the full interface.

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

- If you designed your classes well, the `main` method in `WordGen` should *only* make use of the `SequenceTable` class (but *not* the `FrequencyMap` class). The `FrequencyMap` class only be used from within the `SequenceTable` class. This is an example of **encapsulation**, which refers to hiding internal implementation details from external users. In this case, the details of `FrequencyMap` should be hidden away inside the `SequenceTable` class, meaning that `WordGen` doesn't need to worry about the `FrequencyMap` class or even know that it exists.

- As in Lab 1, use the `Scanner` class to handle user input. You can read an entire line of text entered by the user using the `nextLine` 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 regular 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*), so you can use the primitive types as you normally would when actually working with your `Map` objects. For our purposes, the only time you need to use the corresponding reference classes is when you're specifying generic types.

- 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 many ways to read a file in Java, below is a relatively compact, self-contained method that you can use:

```java
/**
 * 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 Files.readString(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:

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

- Don't forget to apply the fix described in the Program Design section for the situation in which a unique k-length sequence appears at the end of the input. You should alter the input prior to building the `SequenceTable`. A good way to do this alteration is to add the final k-length sequence, plus a space, to the beginning of the original input. For example, suppose that the original input is "the theater is their thing" and $k = 3$. The final k-length sequence "ing" appears only at at the end of the input, and thus there is no information to decide what should follow it during random text generation. To fix this problem, we can change the input to be "ing the theater is their thing", which no longer has this problem.

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

- If you finish the program early and want to go 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 change make the results better/worse in any way?

5 Sample Texts

A collection of sample input files can be downloaded from Blackboard, which you can use as test inputs to your program. The file `whosonfirst.txt` is good as an initial test once you're ready to begin processing actual files (after initial development using a hardcoded string). The other sample input files 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. You can also try files that aren't regular English text; e.g., the `code.txt` sample input file is the source code for a Java class, which should result in code-like output.

Note that when specifying an input filename to your program, typing a name like "`hamlet.txt`" will cause your program to look for the file called `hamlet.txt` *in the same directory as your project*. If the desired input file is contained in a different directory, e.g., a directory named `text-files` that is contained in the project directory, then you must specify the file like "`text-files/hamlet.txt`" so that Java can locate it.

6 Evaluation

Your completed program will be evaluated along the usual three criteria (described in more detail in the Lab 1 writeup): correctness, design, and style. In self-screening your program before submission, your primary reference for correctness-related issues should be your lab writeup (i.e., this document). For design and style issues, your primary references should be the Coding Design & Style Guide, your own good sense, and any specific guidance provided by your instructor (e.g., feedback provided on past labs). Also make sure that your name (and the name of your partner, if applicable) is included in all of your Java files, and that your project directory is properly 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 indicated). In addition to your group's single submission, **each group member must individually submit a group report to your instructor over Slack**. Your group report (which will not be shared with your partner) should summarize your contributions to the lab as well as those of your partner. Your report could be as simple as "we both worked on the entirety of the lab together in front of one machine" if that is the case. Remember that the general expectation is that all group members participate fully in most or all parts of the lab (i.e., not "divide up the work"). Your group report does not need to be long (a few lines is fine), but must be received for your lab to be considered submitted. Submit your group reports over Slack; do not include them in your project submission to Blackboard. Group submissions will normally receive a single grade, but we reserve the right to adjust individual grades up or down in the event of clear inequities.

7 Submitting Your Program

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

1. Save your program and quit your IDE (e.g., BlueJ or Eclipse).

2. **Rename your project folder (which is the folder containing your `.java` files and any associated files) so that it is named `username-lab2` (if working individually) or `username1-username2-lab2` (if working in a group, such as `sbowdoin-jbowdoin-lab2`). Do not include anything else in the folder name!**

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 `jdoe-lab2.zip` (with your actual username(s)).

4. Open a web browser and go to the course's Blackboard page, then browse to `Lab Submissions`. Click on `Lab 2` and then `Browse Local Files` to locate and attach your `.zip` archive. Don't write any comments in the comment section, as your instructor will not see them. Once you've attached your `.zip` archive, click on `Submit` to complete your submission.

5. **If working in a group, submit your group reports to your instructor by Slack**.

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 the Lab 1 writeup for details on connecting to `microwave`).