Lab 2

Due 19 February
Design due in class on Wed.

Handout 3
CSCI 136: Spring, 2007
12 February

Random Writing

1 Short Answers

1. What is printed by the following program?

   ```
   class Example {
       public int num;

       public Example(int initial) {
           num = initial;
       }

       public int getNum() {
           num = num + 1;
           return num;
       }

       public static void main(String args[]) {
           Example first = new Example(10);
           Example second = new Example(20);
           System.out.println(first.getNum());
           System.out.println(second.getNum());
       }
   }
   ```

   What would be printed if we changed the declaration of `num` to be

   ```
       public static int num;
   ```

   What does this tell you about `static` fields?

2. Answer the following questions from the book (Note the difference between Self Check Problems and regular Problems in the book):

   Self Check Problem 3.2
   Self Check Problem 3.3
   Self Check Problem 3.4
   Problem 3.6 (don't write the class— just answer what the advantage would be)

Hand in your answers with your lab design Wednesday morning.

2 Lab Program

Consider the following three sentences:

> 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 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.[*] You will implement his algorithm in this week's lab. In addition to producing mildly entertaining output, this lab enables you to:

1. use the `Vector` and `Association` classes;
2. build more complex structures out of these basic building blocks; and
3. design and implement several new classes.

Character Distributions. Our algorithm is based on letter probability distributions. Imagine taking the book *Tom Sawyer* and determining the probability with which each character occurs. 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 *level-0* analysis, you'd be able to produce random *Tom Sawyer* text based on character probabilities. 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 theatre is their thing", '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

[*] Claude Shannon, "A mathematical theory of communication", *Bell System Technical Journal*, 1948. His technique was popularized by Dewdney's article "A potpourri of programmed prose and prosody" that appeared in *Scientific American*, 122-TK, 1989.

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

At only a moderate level 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, *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."

Once we have processed the text and stored it in a table 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. Then we choose subsequent characters based on the preceding k characters and the probability information.

Program Design. For now, we will focus on implementing a level 2 analysis. That is, we will compute the next character to print based on the previous two characters only.

Think about the design and prepare a written design description of this program. **Submit your design in lecture on Wednesday. I will look through them and return them at the beginning of lab.**

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 info and be able to support two key operations:

- update the probabilities in the table, given a string of 2 characters and the succeeding character.
- select a next character, given a string 2 characters and the probabilities stored in your table.

You could try to save the frequency information in a big array, but the size will quickly become too large. For $k = 2$, you would need a three-dimensional array whose size is about 27,000 entries if you include blanks and punctuation. For larger k, this number becomes much bigger.

Instead, develop a `Table` class which is implemented as a `Vector` of `Associations`. Each `Association` should have a character sequence (stored as a `String`) as its key, along with a value which is a `FrequencyList`. The `FrequencyList` keeps track of which characters appeared after the given sequence, along with a frequency.

There are many ways to implement the frequency list. A good start is... another `Vector` of `Associations`. Thus the frequency list's `Vector` would consist of `Associations` in which the key is a single character (which can be stored simply as a String) and the value is a count of the number of times that that letter occurred after the k-character sequence with which the list is associated (stored as an Integer). Think carefully about what methods the frequency list needs to support and any other instance variables that might be useful.

The data structure design built from these two classes has the benefit of having only as many entries as necessary for the given input text. You may find it helpful to look carefully at the word frequency program in Section 3.3 of *Bailey*.

Your main method for the program should be written in a third class, WordGen, which reads the input text, builds the table, and prints out a randomly-generated string based on the character sequence probabilities from the input text.

To summarize, you will write three classes for this lab: Table, FrequencyList, and WordGen. **The design you turn in Wednesday morning should include a description of each of them.**

Importing Class Definitions. To ensure that your program can find the structure package for the implementations of Vector and Association, you will need to add this to the top of your Java files:

```
import structure5.*;
```

Warning: When you import the classes Random and Scanner, however, use the following lines:

```
import java.util.Random;
import java.util.Scanner;
```

If you write "import java.util.*;" as we did above, the program will get confused as to which version of the Vector class it should use, since there is a Vector defined in java.util as well as in structure.

Implementation Strategy. You should build your program in stages that you have planned out *ahead of time*. While writing and debugging the code, use a fixed a String constant as the input (e.g., "the theater is their thing") and use a fixed $k = 2$.

After the input has been processed you should generate and print out new text using the frequencies in the table. You may start with a fixed sequence of letters that appears in the table or choose starting characters randomly. Generate and print about 500 letters of randomly-generated text so that we can see how your program works. Be sure to handle the special case where you encounter a sequence that has no known successor characters in a reasonable way.

Once the basic program is working, change it to accept input from the keyboard using the Scanner class. When using the Scanner, build up a string of the entire input line by line before performing any frequency analysis. You can use the Scanner methods hasNextLine() to find out if there is another line of input ready and nextLine() to read the next line if it exists, as in the following code snippet. (This code uses a StringBuffer to create large strings efficiently, as we will see in lecture.)

```
    Scanner in = new Scanner(System.in);
    StringBuffer textBuffer = new StringBuffer();
    while (in.hasNextLine()) {
        String line = in.nextLine();
        textBuffer.append(line);
        textBuffer.append("\n");
    }
    String text = textBuffer.toString();
    // text is now the full contents of the input.
```

End of the input is signaled for Java on the Mac (and, indeed, on any Unix system) by typing "Control-D" on a new line.

You may also read your input from any text file, e.g. "whosonfirst.txt", with this command:

```
java WordGen < whosonfirst.txt
```

This is not specific to Java. This “input redirection” can be used with any program running on Unix system.

Finally, change `WordGen` to support levels of analysis other than 2. If you have designed the structure well, simply passing longer strings to the `Table` methods should be sufficient, and you should not need to change any code except in the `WordGen` class.

You can also change your `main` method to use a command line parameter specifying the level of analysis. You can then, for example, run the program with the command:

```
java WordGen 5 < whosonfirst.txt
```

to specify *level-5* analysis. Command line options are passed to your `main` method as an array of strings. You can use the following skeleton code for ensuring that there is at least one argument and for converting the first argument to a string:

```
public static void main(String args[]) {
  if (args.length == 0) {
    // no args, so print usage line and do nothing else
    System.out.println("Usage: java WordGen <level>");
  } else {
    // convert first argument to k
    int k = Integer.parseInt(args[0]);
    // rest of code here
  }
}
```

Optional Extension: Word-level Analysis. Instead of working at the character level, you could work at the word level. 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?

3 Sample Texts

I have put several texts for you to analyze in the following directory:

```
/usr/mac-cs-local/share/cs136/labs/wordgen/
```

The file `whosonfirst.txt` is a good file to try processing first. The others are quite a bit larger and may take a few minutes to process for k larger than 5 or 6. Feel free to use your own texts for analysis as well. The Gutenberg Project has thousands of books available for download from the web and is a good place to look. Be sure to send any entertaining output you generate along the way. I’ll post the best up on the web site.

4 Submission

Package your three Java source files into a “tar file” `lab2.tar`, and submit it using the `turnin` utility before midnight, next Monday.

A “tar file” is like a ZIP or Stuffit file from the Windows or Mac worlds, but is created on the command line and is portable across all Unix versions.

To create a tar file `lab2.tar` that contains files `WordGen.java`, `FreqList.java`, and `Table.java`, you can use this command:

```
tar -cvf lab2.tar WordGen.java FreqList.java Table.java
```

The “cvf” are three flags meaning “create”, “verbose”, and “file”. The ‘`v`’ tells `tar` to give you some output describing what files it’s operating on, and the ‘`c`’ and ‘`f`’ indicates that `tar` should create a new file with the specified file name for the archive.

You can extract the contents of a tar file with

```
tar -xvf lab2.tar
```

But beware! This will extract the files with no concern for whether it is overwriting files in your directory. For this reason, I always create and changing into a brand new directory when extracting tar files.

As in all labs, you will be graded on design, documentation, style, and correctness. Be sure to document your program with appropriate comments, including a general description at the top of the file, a description of each method with pre- and post-conditions where appropriate. Also use comments and descriptive variable names to clarify sections of the code which may not be clear to someone trying to understand it.