Project H: nonsense words at scale

CSCI 1913: Introduction to Algorithms,
Data Structures, and Program Development

(Version 1, Last updated April 23, 2019)

1 Introduction

If there's one thing that slows down assignments in the class, it's the generation of nonsense words.[1] They play an important pedagogical part of "read the code" style problems, but they can be rather loathsome to produce after using any one persons "go-to" nonsense. Therefore, our assignment here will be to write a program capable of generating roughly english nonsense words.

This project will require less programming than the other projects While data structures will be required (I ended up needing two different data structures, used in a total of three different places) programming these data structures will NOT be the focus. Between lab and other projects you've already proved yourselves capable of programming whatever data structure you might need. Instead, the focus of this problem is in choosing data structures for a problem given only the description of the problem, and in analyzing the efficiency of the algorithm that you create.

It's worth noting in this space, questions like this have no "Right" answer. While a poor choice of data structures will get in the way of a correct solution, and a poor implementation strategy will lead to poor performance, there is a large range of reasonable answers to this question. Even beyond these correct solutions, there is a lot of room for improvement by moving from "default purpose" data structures to problem specific ones.

1.1 Learning Goals

This assignment is designed with a few learning goals in mind:

- Selecting the proper basic data type for an abstractly defined problem
- Selecting the proper implementation of the basic data type.
- Analyzing the runtime behavior of a self-designed algorithm.
- Silly words.

[1] This paragraph is laughably not true - most fun part every time

2 Theory: random samples from a computer

A core part of this algorithm will be randomly sampling things (in this case letters) based on pre-specified distributions. There are many ways to do this. Here is one way.

Imagine we had a table like follows showing the domain of things we want to randomly sample from, and a sampling weight (in this case a count) representing how often we want to sample it:

Dog	10
Cat	41
Frog	3
Rock	4
(sum)	58

To sample this we could generate a random number from 0 to the sum of sample weights (not inclusive, so 0 - 57 in this case) This can be done by `java.util.Random`. Let's say we draw 52; We can then loop over the table, subtracting the weight from our random number. Whichever subtraction leads the random number to become negative, is the one we randomly chose.

So, we would start with 52. Then we would subtract 10 (Dog) and get 42. Then we would subtract 41 (Cat) and get 1. Then we would subtract 3 (Frog) and get -2, meaning we would have sampled Frog.

This algorithm is relatively simple and, although it isn't always obvious to see, will return objects with frequency proportional to the weights listed in the table. You do not need to use this algorithm, but this felt like a task where I should at least provide a description of one option. If you don't go for this option, I bet there's something cool you could do with binary trees.

3 Theory: Markov chain modeling of pronounceable text

Markov chain models are a popular type of statistical model for handling sequences of things. Commonly used for time-series data, this model can also be used for tasks such as modeling the order of letters in words. The key assumption of a Markov model is that the next outcome is not determined by the entire history (I.E. the next letter is not determined by the entire word so far) but by only the most recent one or two events (i.e. the most recent few letters). This assumption drastically simplifies the number of things that might need proper modeling to the point where simple observed frequency based models can be sufficient.

A Markov model for text generation has a few parts: The first part is a parameter telling it how much "history" to keep. So for $n = 1$, each letter in the random word would be based only on the letter before it. For $n = 3$, each letter would be based on the 3 letters before it and so forth.

Given this value, a Markov model would say for any n letter sequence, what the prob-

abilities are of each letter following that. "th" might be followed by "e" 10% of the time, and "a" 15% of the time, and so-forth. This ends up being a major data structure modeling task for this assignment: how do you represent the various$n$ letter sequences and associated probability distributions.

Given this information a word can be generated with the following basic algorithm:

1. start the word
2. while word is not done:
3. word = word + pick random letter based on last n letters of word

Some notes here: First - starting these sequences can often be confusing because there are not enough letters. For our program we will track next-letter probabilities for not only all sequences of n letters, but also for the first 0, 1, 2, ... n-1 letters of a word. In this way we can use the probability distribution for prefix "" to pick the first letter, and so forth.

Stopping this process can likewise be difficult. One trick that I've often used is adding a "stop letter" to each word when I read them in. The stop letter should be something guaranteed to not be part of the word such as an '*'. Then I know a word is done when it generates the stop letter by chance. This leads to a nice distribution of word lengths.

Sometimes, especially with larger values of n, it can be the case that we don't have data for a given series of letters. It's rare, but your code should handle it. Typically the way to handle this is to treat it like the end of a word.

Learning markov models is actually quite simple. You simply take many words and count each n letter sequence and what comes next. So the word "grobalo" with $n = 3$ would generate the following data for counting:

prefix	next letter
""	g
"g"	r
"gr"	o
"gro"	b
"rob"	a
"oba"	l
"bal"	o
"alo"	end of word "letter"

With enough words you can generate enough samples that the probabilities distributions you create can accurately reflect pronounceable words.

4 verb list

For this project, we will be training our Markov model based on a provided Verb list. This will be provided in a text file that has one verb per line. The reason we are using verbs is that, as described in the intro, I want random function names. Under most code style

guidelines function names should be verbs. By only training our model on verbs we are more likely to generate reasonable sounding verbs, and less likely to generate words that seem like other parts of speech such as grobliciousness (which is of course, the property of excellence in the action "grobaloo") A license is attached with this verb list. From what I can tell none of the data origins used in the production of the verb list actually forbid anything, but I am required to redistirbute the WordNet license since this data is derived from data that was derived from WordNet. If you ever need to redistribute the verb list, go for it, just be aware that you need to distribute that license too.

5 Program Requirements

Your program will have three required classes. You certainly can program more. As this will be exclusively manually graded, you are under no obligation to name your classes or functions as shown here. I do, however, expect reasonable analogs of these classes.

5.1 RandomDist

The `RandomDist` represents a frequency weighted random choice over a provided collection of things. It should have a type parameter allowing it to be used for modeling frequency weighted random distributions over any number of things. At a minimum it should have two functions `add` and `draw` The draw function should randomly sample from the distribution based on internal variables. The add function allows incrementally specifying the distribution it should be sampling from. So if we called add(3), add(3), add(5) then draw should return 3 $\frac{2}{3}$rd of the time and 5 $\frac{1}{3}$rd of the time.

5.2 Wordalizer

The `Wordalizer` is the part that's in charge of words and word generation. It should have a constructor that takes the int n (from above - the number of letters to use in computing which letter comes next). It should have a function to add a word which will be called many times by main. This should take the word apart into samples as shown above, and train the distributions needed for future word generation tasks. It should also have a function to generate a word.

A note on design here - it might make more sense to give the construction all of the information it needs to make an object that's ready to generate words. However, there's no good standard way to do this that doesn't couple the object to a specific way to read the file. Therefore it seems like the object would be more general with the design shown here, where the object is in charge of both building the markov model, and using it to generate words.

5.3 WordalizerMain

Finally, you need a main method. This should be in charge of opening the verb file, reading it into the Wordalizer, and printing output. Some formal requirements here: You should pick the value of n (from above) that looks best to you. I personally like $n = 2$ or $n = 3$, if you go to small it just starts looking random, and if you go too big it starts having trouble not generating real words due to too few sample words. I want this main method to print 10 random words. These words should not be real words.

A note on this - the Wordalizer can easily generate real words, this is fine. I want your main method to, with the use of some data structure, promise NOT to print a real word.

6 Limitations

There are a few limitations for this project. First and foremost, as usual this is NOT a collaborative project. You should not show your code to anyone, or look at any other code for this problem, nor should you closely discuss solution details with anyone else. If you need help on any part of this project please email the Professor. I would love to help in this project, especially if you find yourself stuck on the design or analysis phases (as these are skills we have not built as much in class)

As mentioned in the introduction - this program does not place any restriction on built-in java data structure use. You are free to use any built-in data structure. If you choose to use one, however, I expect you to explicitly address it in the project report, and describe what data structure it is in relation to class material.

7 Testing and incremental development

I will be providing no formal assistance testing this code. I can, however, assist in testing and debugging over email and in person.

One thing that will make testing this code difficult is that the output is expected to be random. My best advice for dealing with this is two parts: First - test it many times, making sure that range of answers make sense, and secondly, test at a lower level, using print statements inside of functions or using the interactive debugger. The reason behind the first is that you can tell if a function is broadly doing the right thing if you run it enough. The reason for the second is that even if the outcome of a function should be random, the process should not.

I can provide some example outputs from my code for n ranging 1 to 8. Above 8 I think my code infinite loops due to issues in generating novel words.

```
 Words for n=1
pouly
se
ctagor
```

```
whut
strer
panalidintol
tow
culve
rem
eteburacimutormex
 Words for n=2
greepwit
earroventyleate
sh
moutide
reout
sleather
jackle
jack
rifeup
te
 Words for n=3
cotter
cramp
atter
foot
evanimate
flagitate
seesage
whissaddle
sleetle
tandize
 Words for n=4
correspire
collup
identuate
wateriorate
withdrawin
clashout
wrest
particise
origin
tradiate
 Words for n=5
counterfere
```

```
sickenout
particulate
maundertake
wherringbone
rabble
advertire
animadvertire
distillhunt
feathe
 Words for n=6
contract
dispreadeagle
condescendo
ticktogether
glistenin
extemporize
reorganise
scrabbled
scrabbled
glistenin
 Words for n=7
extemporize
temporise
intercommuning
temporise
extemporize
contract
intercommuning
subcontractout
contract
temporise
 Words for n=8
subcontractout
contract
subcontractout
subcontractout
subcontractout
subcontractout
subcontractout
subcontractout
contract
subcontractout
```

Yeah, 8 doesn't really work with this dataset and my algorithm. I think the issue is that my implementation ignores words shorter than the input n, and that doesn't leave a lot of words.

Of particular focus for debugging is the string processing involved in adding a word. I strongly recommend debugging this code carefully. This type of string manipulation often leads to minor indexing issues, but a minor incorrect indexing issue may easily lead to an infinite loop later in the program.

8 Project Report

For this project I want a small report as well. This report should be reasonably short. I'm imaging 2-4 pages, but I won't be paying page count too much mind. It should be submitted in pdf format. I'm expecting something pretty technical, don't worry about intro and conclusion, just make sure the material is in there.

The report should have the following information:

- What was your design for this problem - this should answer questions like - what data structures were needed, which ones did you choose, and why?
- If you have any additional functions or class from what I described what are they and what do they do?
- What assumptions are you making - this should include a sentence for any built in java data structure describing what comparable data structure we studied you are assuming it represents. As well as anything else you are assuming.
- What is the runningtime of your classes? To structure this consider:
  - The runtime of training the markov model
  - The runtime of training the markov model for one word
  - The runtime of sampling form the random dist object
  - the runtime of generating a word

  For all of these you may need to pick what the variables are that you perform the analysis against. For some the number of words in the file might be good, for others it might be the length of a string. Defend your runtimes briefly (don't just say $O(nlog(log(n)))$, but I also don't need a page for each function)
- What is your favorite word it generates?

9 Deliverables

For this project you should submit the following files You can rename any of these files if you want, just make sure I can understand what they map to in my class names.

- RandomChoice.java (A class to represent a random distribution over generic data based on a counting scheme)
- Wordalizer.java (A class that generates Markov models of english text, and then generates nonsense words)
- WordalizerMain.java (A class that runs the whole thing)
- Writeup.pdf (A pdf formatted document explaining your design and its implications)
- Any other files needed to run your code (word iterator, word list, etc.)

Grading will be done 100% manually, so specific names and folder organizations are not a big concern. Feel free to just ZIP your entire intellj folder and send it my way. This will be submitted Directly to Daniel, and is due before the final, although I would like to see if before then if you have the time.