

# PSA 7 - Generating Text with Markov Language Model

To view the table of content, use [View > Show document outline]

Please **read the entire document** before getting started on coding, so that you can better pace yourself.

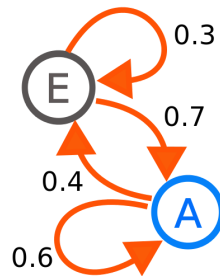
Be aware of course policies as explained on the website and linked via past PSA documents.

## 1. Introduction

Computers are becoming excellent at talking to us, people. Personal assistants such as Siri, Alexa, and Google Assistant can now listen to people asking questions and responding verbally. We shall implement a program that does something similar with combinations of words, processing the language through our algorithms, and producing paragraphs that seem almost human. At the core of the algorithm is the HashMap, a convenient data structure “mapping” keys to values.

### Markov Model

In the early 20th century, Russian mathematician Andrey Markov first proposed the concept of the Markov Decision Process. It is best characterized by “memorylessness”: any decision made is not dependent on the previous decision, but only dependent on the current state.



*A simple Markov Model with 2 states*

The above diagram demonstrates state transitions in a Markov Model. Each state has some numbers on arrows (a transition function) describing how likely it is going to take a particular path. For example, if I am currently at state A, here is a 0.6 chance I will stay at A, and a 0.4 chance I will move to state E. Once in State E regardless of where I was before, I have a 0.3 chance of staying at E and a 0.7 chance of moving to A.

This decision process seems simple but has many applications, and its variations are widely used in AI decision making, deep learning, game theory, and more. If you would like to read up more on the Model or its applications, [its Wikipedia page](#) is a good place to start.



*Andrey Markov*

## Markov Language Model

In this assignment, we will explore the usage of a Markov language model, to generate text that looks very much like the corpus (a collection of written text) used to train the model. In our model, we define the “current state” as the most recent word(s), and define “decision” as the immediately next word.

Consider the following sentence:

*“I do not like Green Eggs and Ham.”*

Take an arbitrary position in the sentence, for example, in this case, take the position between the words “and” and “Ham.” Then the most recent words up to the position taken are “Green Eggs and”, and the word immediately following is “Ham.”

Suppose this sentence is part of the corpus used to train the language model, then there must exist a transition:

*“Green Eggs and” → “Ham.”*

Furthermore, the “degree” of a Markov model is defined as how many most recent words to consider when generating a next word. In this example, we use degree-3 word model, because we consider the previous 3 words.

### “Formal” Definition

The count distribution function of a Markov Word model,  $F$ , is written as follows:

$$F(w_{n+1} \mid w_n, w_{n-1}, \dots, w_{n-i})$$

Where each  $w$  is a word that appeared in the training data (also,  $w_{n-1}$  represents the word immediately before  $w_n$ , and so on.  $i$  is the degree of the model.

Equations are hard to understand at the first glance. Let's again assume a degree-3 model, we know the sequence “Green Eggs and Ham” appeared a total of 9 times in the text (yes, really), then we know given the prefix “Green Eggs and”, the prediction “Ham” has a frequency of 9:

$$F(\text{Ham} \mid \text{Green Eggs and}) = 9$$

Now suppose (hypothetically) that the sequence “Green Eggs and Bacon” also appeared once in the text, i.e.  $F(\text{Bacon} \mid \text{Green Eggs and}) = 1$ . Then we know that when given the prefix “Green Eggs and”, “Ham” is 9 times as likely to be predicted as “Bacon”.

We will use this distribution function  $F$  to generate the next word. We will be storing these words into something called a **HashMap**.

## HashMap

A HashMap is used for storing Key & Value pairs, it is denoted as `HashMap<Key, Value>` or simply `HashMap<K, V>`. Key and Value can be any non-primitive type.

When using a HashMap, use Key to obtain or manipulate the Value that is associated with the Key.

Here are some straightforward examples of using a HashMap in Java:

<https://www.javatpoint.com/java-hashmap>

You may also want to learn about what methods you can use with a HashMap by reading the Java documentation:

<https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html>

## This assignment

The program you implement will:

1. Process a file containing the training corpus to identify possible transitions, store these transitions (and their frequencies) in HashMaps.
2. Use the trained HashMaps to generate text.

## 2. Starter Code

All files are in `~/../public/psa7/`, copy them to your home directory using command

```
> cp -r ~/../public/psa7 ~
```

In addition to skeleton code, and the main program in `Generator.java`, a few training text files are also provided:

`Turing.txt`

Alan Turing's famous paper "Computing Machinery and Intelligence", in which he proposes the "Imitation Game" to determine whether a machine is thinking.

<https://www.csee.umbc.edu/courses/471/papers/turing.pdf>

`Odyssey.txt`

An ancient Greek epic poem attributed to Homer.

<http://classics.mit.edu/Homer/odyssey.mb.txt>

`Walden.txt`

*Walden, and on the Duty of Civil Disobedience* by Henry David Thoreau

<https://www.gutenberg.org/files/205/205-h/205-h.htm>

`Lyrics.txt`

Lyrics from all Billboard top 100 songs since 2000. Excerpted from

<https://www.kaggle.com/rakannimer/billboard-lyrics>

`GreenEggsAndHam.txt`

Children's poem by Dr. Seuss, written using only 50 unique words

<http://www.site.uottawa.ca/~lucia/courses/2131-02/A2/trythemsources.txt>

## 3. How to Run

To compile the main program:

```
> javac Generator.java
```

To run the text generator:

```
> java Generator filename w|c [-d degree] [-n count]
```

Where the flags are:

<code>filename</code>	-> Required. Specify the training data file
<code>w c</code>	-> Either w (word) or c (character) required. Specify which model
<code>[-d degree]</code>	-> Specify the degree of the Markov Model. Default at 2, must < count
<code>[-n count]</code>	-> Specify how many sentence to generate. Default at 10

For example:

```
> java Generator Walden.txt w
> java Generator Obama.txt c -d 5 -n 2500
```

word-by-word, by default  
100 words

## 4. Testing

A simple tester is provided, using a contrived test case to verify the correctness of `incrementPrediction()` and `getFlattenedList()` in `WordModel`.

To run the tester:

```
> javac TestWordModel.java  
> java TestWordModel
```

In addition to the tester, you should also create your own training data to verify the functionality of your program. We have also provided you some larger text files that can be used to train the Markov Model.

Make sure you test your program step-by-step as you code along. We have provided a starter code that will compile in the beginning. Because this is a syntactically-heavy assignment, you should make sure one of your method have no error before moving on to the next method.

## 5. Implementation

### **public class WordModel [total 65pt]**

You will now implement a Markov Language Model that is capable of training by reading and “remembering” all prefix+prediction combinations, where prefix is a list of [degree] words (e.g. 2 words for a degree-2 model), and prediction is one word. Store these combinations into a HashMap, such that it can generate text word-by-word once training is finished. This model should work for any degree.

Note: Once you finish implementing WordModel, we will reuse the WordModel’s methods to implement a CharacterModel, which works similarly except it generates text character-by-character. But you don’t need to worry about it right now.

The following methods should be implemented in WordModel.java. This is a syntactically heavy assignment, be sure to compile (and run) as you code along to fix errors early. Also, you might want to check Generator.java for how these methods are called in the main program.

There is a “debug” option in Generator.java that might help you debug, set it to true if you want to print out the values of the maps before generating sentences. You can also use `HashMap.toString()` in various places to help debug.

**public WordModel (int degree); [5pt]**

**degree:** the degree of the Word Model.

This constructor creates a **WordModel** object of degree **degree**. For example, a degree-2 model would use the previous 2 words as a “prefix” when predicting the next word.

Initialize all instance variables in this constructor.

**public void incrementPrediction(ArrayList<String> prefix, String prediction); [15pt]**

**prefix:** a list of [degree] words that comes immediately before the prediction word in a sentence.

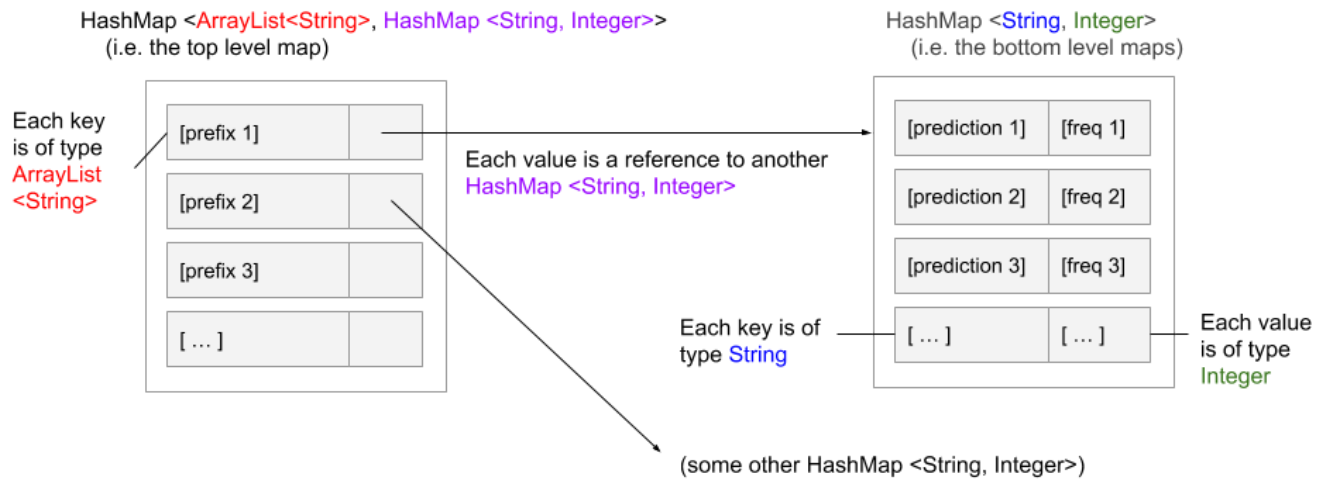
*([degree] is the degree of the current WordModel, for example, in a degree-2 model, the parameter prefix would be a list of 2 words. Same goes for the rest of the document)*

**prediction:** a word that comes immediately after the prefix words in a sentence.

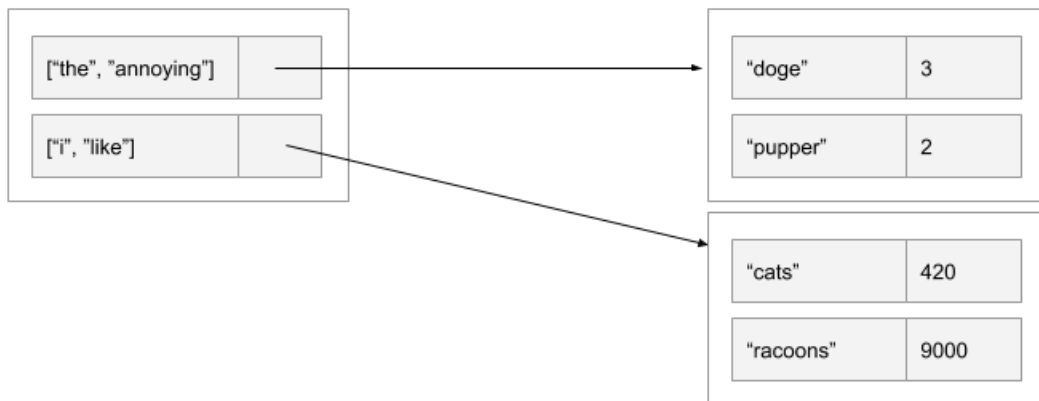
First let’s take a look at the instance variable:

```
protected HashMap<ArrayList<String>, HashMap<String, Integer>> predictionMap;
```

Notice that the keys in this HashMap are of type `ArrayList<String>`, and the value is another HashMap whose keys are Strings and values are Integers. To illustrate, the data structure is described by the following diagram:



For example (a degree-2 model):



This method increases the frequency of a prefix+prediction pair by 1.

Because neither the Keys nor the Values exist in the HashMap in the beginning, we need to check for existence before doing any incrementation. The logic of checking existence roughly follows:

```

If prefix (Key of the top level map) exists
    If prediction (Key of the bottom level map) exists
        Increment frequency (value of the bottom level map) by 1
    Else
        Put prediction as a Key in the bottom level map, set its Value to 1
Else
    Create a new map, put prediction as a key in the new map, set its value to 1
    Put prefix as a key in the top level map, set its value to the new map.
  
```

**DO NOT** modify any key (the ArrayList of words) once you insert it into the HashMap.

Think back to PSA2 when you incremented the count of a WordPair object. This should be somewhat similar.

```
public int trainFromText(String content); [15pt]
```

**content:** a string containing the entire training text

**Return:** how many words are read from the file

This method trains the model by reading the text word-by-word (and character-by-character later on, but you don't need to worry about this in WordModel), while incrementing frequencies of occurring prefix+prediction pairs in the HashMaps.

First, decide where to start processing each word. Because we need [degree] words as the prefix, it would probably be a good idea to start after [degree] words.

Starting from this position, call incrementPrediction() using the previous [degree] words as the prefix and the next word after that as the prediction. Repeat until there are no more words to be read. **All words should be converted to lowercase. Don't worry about punctuation.**

Hint: it might be a good idea to first convert the (very long) content string to a list of words. To do this, you might need to use a [Scanner](#) object.

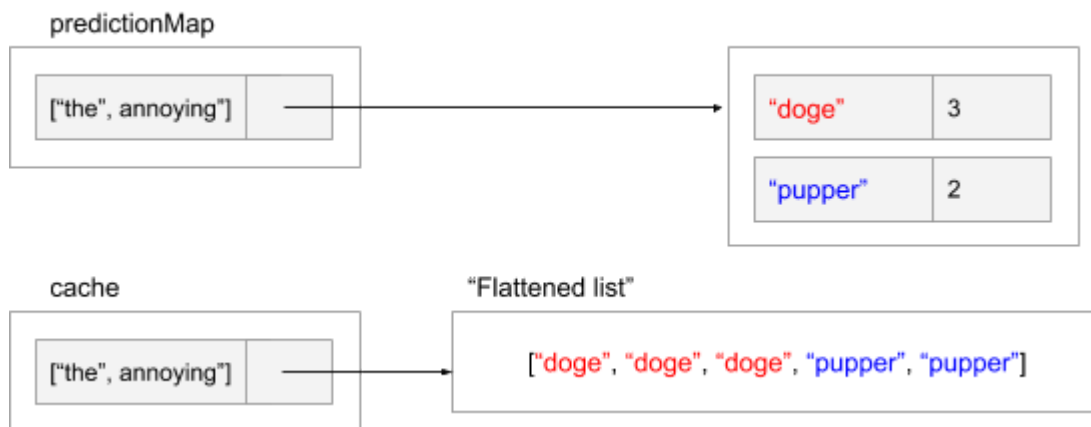
```
public ArrayList<String> getFlattenedList(ArrayList<String> prefix); [10pt]
```

**prefix:** a list of [degree] words that comes immediately before the prediction word.

**Return:** a list of words that are predicted by the prefix, where each word is repeated the same amount of times as its frequency.

First check whether prefix already exists in cache (the instance variable), if it does, directly return the value. If it does not exist, construct a "flattened list", add it to the cache (for future use), and return the list.

Here's an example of how to make such a flattened list. The following diagram demonstrates a HashMap with some data in it, when calling getFlattenedList() with the prefix "the annoying":



```
public String generateNext(ArrayList<String> prefix); [10pt]
```

**prefix:** a list of [degree] words that comes immediately before the prediction word.

**Return:** a word that the model predicts based on the prefix.

In this method, get a "flattened list" of words associated with the specified prefix. Then randomly return a word from the list.

\* You may assume prefix always exist in the HashMap. See Generator.java:87-100 to figure out why

\*\*\* Reasoning behind Markov decision making: intuitively, the list returned by getFlattenedList() has more copies of common words than rare words, so randomly picking an index from this list is more likely to select a common word than a rare word. Because the flattened list's word repetition is consistent with the frequency distribution, it preserves the each individual word's probability of being selected.

**public String generate(int count); [10pt]**

**Return:** a String of [count] words generated

First, randomly select a prefix from all the available prefixes. Recall that all the keys in `predictionMap` are available prefixes. The following code will effectively get all the keys in `predictionMap` into an ArrayList:

```
ArrayList<ArrayList<String>> keys =  
    new ArrayList<ArrayList<String>>(predictionMap.keySet());
```

Now we have [degree] words already selected so far which work as a prefix.

Use the most recent [degree] words as prefix, to generate the next word. Repeat until there are exactly [count] words generated. Then, return all of them as a single string.

Hint 1: it might be a good idea to keep an ArrayList of words when generating, then, before returning, turn these words into a single string.

Hint 2: don't forget to add space between words.

**Pause here.**

Now that you have finished implementing all methods in `WordModel`. Test your code thoroughly, by running the tester or running the main program. **Again, you should design your own test cases!** Then, move on to the remaining part when you've made sure that your `WordModel` works correctly.

**public class CharacterModel extends WordModel [total 15pt]**

Suppose now treating each word as an individual token is not good enough, and we want the model to be able to generate text character-by-character. Well good news! Because we know that a char can comfortably fit in a String (e.g. single char String "a"), we can simply tweak the program slightly, to make a Markov Language Model that can process at the character level.

To do this, we only need to make some changes to the way the `WordModel` processes training text, and therefore we can inherit most of the functionality from `WordModel`.

The following methods should be completed in `CharacterModel.java`:

**public int trainFromText(String content); [10pt]**

First, copy over your code from `WordModel.trainFromText()`. Currently, your code uses the previous [degree] words as the prefix and the next word as the prediction.

Now, modify your code so that it uses the previous [degree] characters and the next one character, respectively, instead. However, to satisfy the type String in the HashMaps, put these characters separately in their own one-char-long String.

For example, again:

*"I do not like Green Eggs and Ham."*

Take the position immediately before 'd', in a degree-4 `CharacterModel`, we will have the following transition:

["s", " ", "a", "n"] -> "d"

Also, make a change so that it returns the number of characters (instead of words) read.



**public String generate(int count); [5pt]**

Copy over your code from WordModel.generate(). Because the space character is now being considered as a valid prediction, we no longer want to put a space character between every two predictions in the generated text. Delete the line where you put such a space character.

Pat yourself on the back. You just finished PSA 7. You made Markov very proud! Be sure to test a few times before your final submission.

## 6. Extra Credit: Linear Interpolation of Two Models

### Introduction

There are many scenarios where we can improve a Markov Language Model by combining it with another one. For example, when there are two (relatively large) models, we could combine them to generate text with a more diverse vocabulary. Or when there are models trained by texts from different authors, we could combine them to eliminate word-choice biases from an individual author.

One easy way to achieve “combining two Markov Language Models” is by using linear interpolation. Linear interpolation combines the distribution of “prediction words” from 2 models by linearly proportioning (i.e. multiplied by some constant) them into one new distribution. The new distribution function, denoted by  $F$  is defined as such:

$$F(w_{n+1} | w_n, w_{n-1} \dots, w_{n-i}) = aF_1(w_{n+1} | w_n, w_{n-1} \dots, w_{n-i}) + bF_2(w_{n+1} | w_n, w_{n-1} \dots, w_{n-i})$$

Where  $a, b$  are some non-negative constant weights. And  $F_1$  and  $F_2$  are original distribution functions of the old models. And  $i$  is the degree of the model.

Let's consider the word sequence “I like apple juice” in two degree-3 models, we have:

$$F(\text{juice} | I, \text{like}, \text{apple}) = aF_1(\text{juice} | I, \text{like}, \text{apple}) + bF_2(\text{juice} | I, \text{like}, \text{apple})$$

Suppose in model 1 “I like apple juice” occurred once and in model 2 “I like apple juice” occurred twice, or simply:

$$F_1(\text{juice} | I, \text{like}, \text{apple}) = 1$$

$$F_2(\text{juice} | I, \text{like}, \text{apple}) = 2$$

Assume  $a = 2, b = 1$ , then in the combined model:

$$F(\text{juice} | I, \text{like}, \text{apple}) = 2F_1(\text{juice} | I, \text{like}, \text{apple}) + 1F_2(\text{juice} | I, \text{like}, \text{apple})$$

$$F(\text{juice} | I, \text{like}, \text{apple}) = 2 * 1 + 1 * 2 = 4$$

In other words, in the combined model, the word “juice” effectively occurred 4 times after “I like apple”.

Here is another example, suppose there are these two (small) trained models, with all prefixes and all the occurrences of the prediction words, followed by their count:

Model 1			Model 2		
Prefix	Prediction	Frequency	Prefix	Prediction	Frequency
“the”	“dog”	1	“the”	“dog”	2
	“cat”	1		“cat”	1
	“memer”	2		“sloth”	1
“a”	“bunny”	2		“memer”	8

Suppose we use  $a = 2$  and  $b = 1$  (i.e. Model 1 weighs twice as much as Model 2) when combining these 2 models. Once combined, the new model should have the following distribution.

New Model (constructed with Model 1 weighing twice as much as Model 2)		
Prefix	Prediction	Frequency
"the"	"dog"	$1*a + 2*b = 1*2 + 2*1 = 4$
	"cat"	$1*2 + 1*1 = 3$
	"sloth"	$0*2 + 1*1 = 1$
	"memer"	$2*2 + 8*1 = 12$
"a"	"bunny"	$2*2 + 0*1 = 4$

Because the total frequency behind the prefix "the" is now 20, if we use the new model to predict the word after "the" for many times, we can expect 4/20 of all predictions to be "dog", and 12/20 of all predictions to be "memer", and so on.

### Your task [total +10pt, all or nothing]

In a file named **CombinedModel.java**, implement a class called **CombinedModel**. This class should combine two WordModel using the method described above. This class only need to generate the prediction, it does not need to train from text (because 2 trained model are passed in upon construction), and does not need to generate a whole document.

You're free to put any instance variables and methods in this class, but they need to have the following constructor:

```
public CombinedModel (WordModel model1, WordModel model2, int a, int b);
```

**model1, model2:** 2 trained WordModels to be combined

**a:** weight of model1, non-negative

**b:** weight of model2, non-negative

And there must be the following method:

```
public String generateNext(ArrayList<String> prefix);
```

**prefix:** a list of [degree] words that comes immediately before the prediction word.

**Return:** a word that the model predicts based on the prefix.

Hint 1: you can use all the methods in class WordModel, and you can get a reference *predictionMap* using the getter method.

Hint 2: make sure the check whether the prefix exists in either model. For example, in the example, the prefix "a" only exists in model 1 but not model 2

### Testing

A good way to test this might be training 2 models with a small training data, create a CombinedModel and call generateNext() with the same prefix for many times. See whether the output is consistent with expectation.

## 7. README [total 20pt]

Complete this part in a file named README.md **after** you finish implementing the program.

### 7.a. High-level program description [3pt]

Describe in high level what the program does. Mention how to run the program and explain why this program is useful. Make it simple so that a person with no prior CS knowledge could understand it. Make it short enough so that graders do not get bored reading it.

### 7.b. Testing [3pt]

In addition to the provided tester, describe how you tested your code (i.e. what testing method, what test cases, etc.)

### 7.c. Questions [14pt]

Answer the following questions:

1. What is a HashMap?
2. How is a HashMap different from an ArrayList?
3. Do you recommend implementing this program with only ArrayLists instead of HashMaps? Why or why not?
4. Name one one limitation of the Markov Model.
5. Text generated by a Markov Model sometimes does not make sense semantically (i.e. does not have meaningful content). Why is that?
6. Suppose your data is trained with many real news articles, and you need to generate some fake news (fake news are text that are different in content compared to real news). Would you choose a very high degree (say 20) for your WordModel? Why or why not?
7. Based on Generator.java:87-100, why do you think we can assume prefix always exists in the HashMap, in method generateNext()?

## 8. Style Guidelines ([Link](#))

Refer to the website with a more complete guideline and set of examples of what your style should look like. There are ten possible points that can be deducted.

## 9. Submission

**Your code must be able to compile using the Java compiler on ieng6 to receive any credit.**

Submit the following files:

Generator.java      WordModel.java      CharacterModel.java      README.md

If you're also attempting the extra credit, also submit:

CombinedModel.java

## A1. Sample Output

Because of the random nature, text generated probably will not be exactly the same

```
> java Generator Walden.txt c -d 0
```

Constructing a Markov character model of degree: 0

Training from data: Walden.txt

Training done, 635665 read.

Generating text...

eyysat, "iyeoriy owcr eroeonra slarocsmsedaaiagsh pilitnyb cefnarioiaeeoer,tttsh m  
a,nhmae mthaweina

```
> java Generator Turing.txt w -d 3
```

Constructing a Markov word model of degree: 3

Training from data: Turing.txt

Training done, 11769 read.

Generating text...

the machine, "do your homework now," being included amongst the well-established facts, and this, by the construction of the machine, will mean that the homework actually gets started, but the effect is very satisfactory. the processes of inference used by the machine need not be troubled by this objection. it might be urged that when playing the "imitation game" the best strategy for her is probably to give truthful answers. she can add such things as "i am the woman, don't listen to him!" to her answers, but it will avail nothing as the man can make similar remarks.  
we

```
> java Generator GreenEggsAndHam.txt c -d 7 -n 200
```

Constructing a Markov character model of degree: 7

Training from data: GreenEggsAndHam.txt

Training done, 3595 read.

Generating text...

em here and there.

i do not like then with a goat!

would you in the rain?

i do not like them with a mouse.

and i will not eat them with a mouse.

i do not like them, sam-i-am.

a train.

not

---

## PSA 7 - Markov Language Model



**"I put so much wax in my beard you can light it like a candle." - Andrey Markov**

---

---

---

## **Real or generated?**

**“Kings and queens who wear a suit but once, though made by some tailor or dressmaker to their majesties, cannot know the comfort of wearing a suit that fits. They are no better than wooden horses to hang the clean clothes on.”**

---

---

## Real or generated?

**“The question, perhaps from the leisure fruitful. ‘But,’ says pertinently that sound asleep. Yet his gentle rain while to choose. If he was nearly level, and wisely; as he sprang upon the entrance of goldenrod (*Solidago stricta*) grows a half-hour’s nap after practising various animals.”**

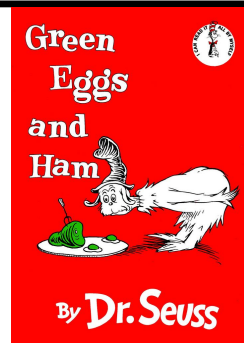
---



---

## Real or generated?

will eat them here or there. i do not like them,  
sam-i-am. say! in the dark. not on a boat. i will eat them  
in a tree! not in a box? would you eat them in a tree! not  
in the dark. would you like them with a mouse. i do not  
like green eggs and ham! i do not like them in a house. i  
do not like them,sam-i-am. i do not like them,sam-i-am. i  
do not like them,sam-i-am. i do not like them with a fox.  
not in a box. not with a mouse. not with a goat!



---

---

**Real or generated?**

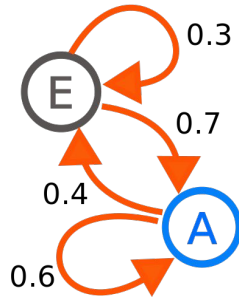
**Henry is the best**

---

---

## Markov Decision Making:

Making a decision according to only the current state.



When I am at state A:

- How likely will I go to state E?
  - How likely will I remain in state A?
-

---

## Markov Decision Making (Language Model):

Making a decision according to only the “current state”.

The “current state” is the previous n words.

The “degree” of the model is how many previous words to look at (3 in this example)

Try to predict the next word at this position

↓  
“I do not like *Green Eggs* and *Ham.*”

“*Green Eggs and*” → “*Ham.*”

↑  
prefix (current state)

↑  
prediction (decision)

---

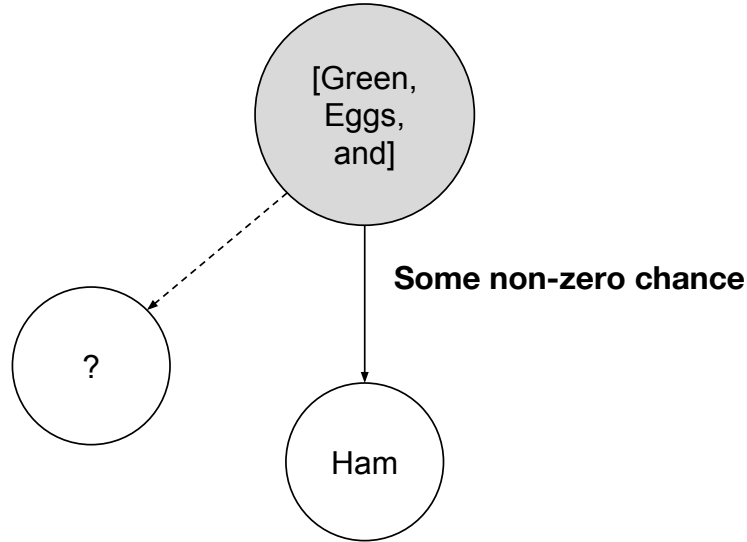
---

## Markov Decision Making (Language Model):

Making a decision according to only the “current state”.

The “current state” is the previous n words.

The “degree” of the model is how many previous words to look at (3 in this example)



---

## **“Formal” Definition**

**A distribution function:**

**$F(\text{prediction} \mid \text{prefix})$  = however many times prediction word appeared after prefix**

**I do not like Green Eggs and Ham.**

**I do not like Green Eggs and Ham.**

**I do not like Green Eggs and Ham.**

**I do not like Green Eggs and Bacon.**

**$F(\text{Ham} \mid [\text{Green, Eggs, and}]) = 3$**

**$F(\text{Bacon} \mid [\text{Green, Eggs, and}]) = 1$**

---

---

**Because we have:**

$$F(\text{Ham} \mid [\text{Green, Eggs, and}]) = 3$$

$$F(\text{Bacon} \mid [\text{Green, Eggs, and}]) = 1$$

**After training, when predicting the word after [Green, Eggs, and]**

**“Ham” is 3 times as likely to be predicted as “Bacon”**

---

---

---

**That's all great.  
But how do I implement this function in  
Java?**

**HashMap**

---



---

## The life of a WordModel

new WordModel(3)

trainFromText()

calls

incrementPrediction()

generate()

calls

generateNext()

calls

getFlattenedList()

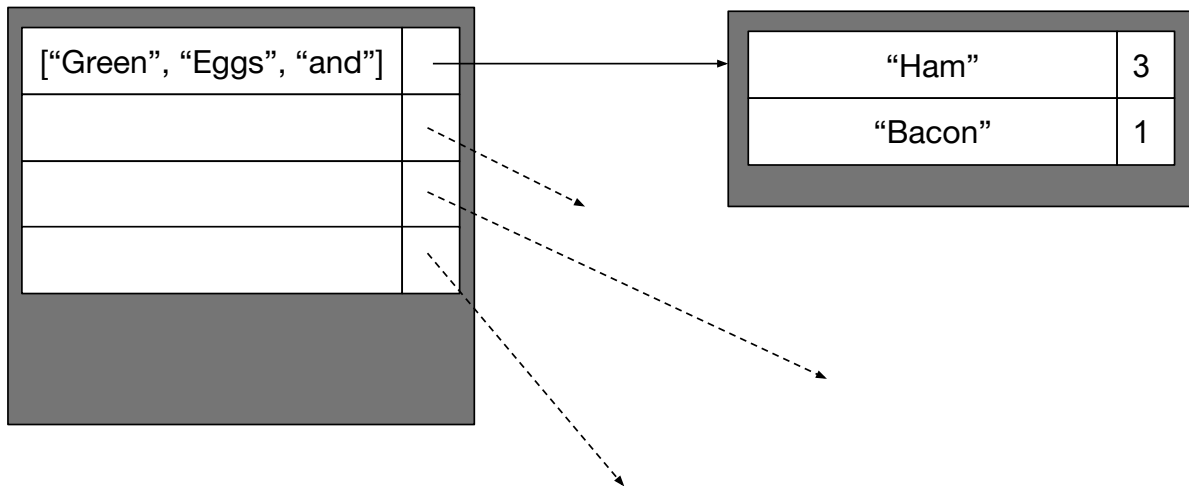
---

---

## Coming back to the distribution function

$$F(\text{Ham} \mid [\text{Green}, \text{Eggs}, \text{and}]) = 3$$

$$F(\text{Bacon} \mid [\text{Green}, \text{Eggs}, \text{and}]) = 1$$



---

## Method number 1

```
public void incrementPrediction(ArrayList<String> KEYprefix, String prediction)
```

This method increases the frequency of a prefix+prediction pair by 1.

---

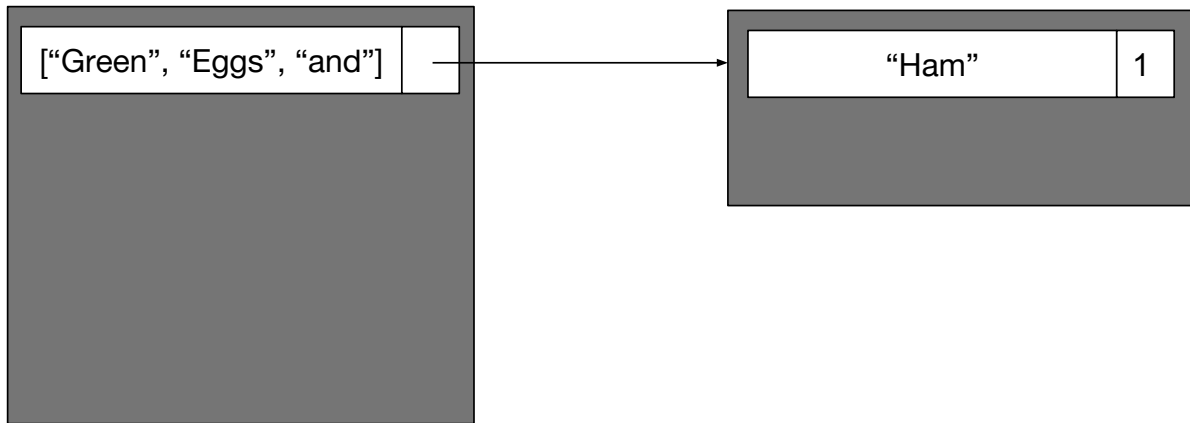
---

**Before `incrementPrediction(["Green", "Eggs", "and"], "Ham")` the first time**



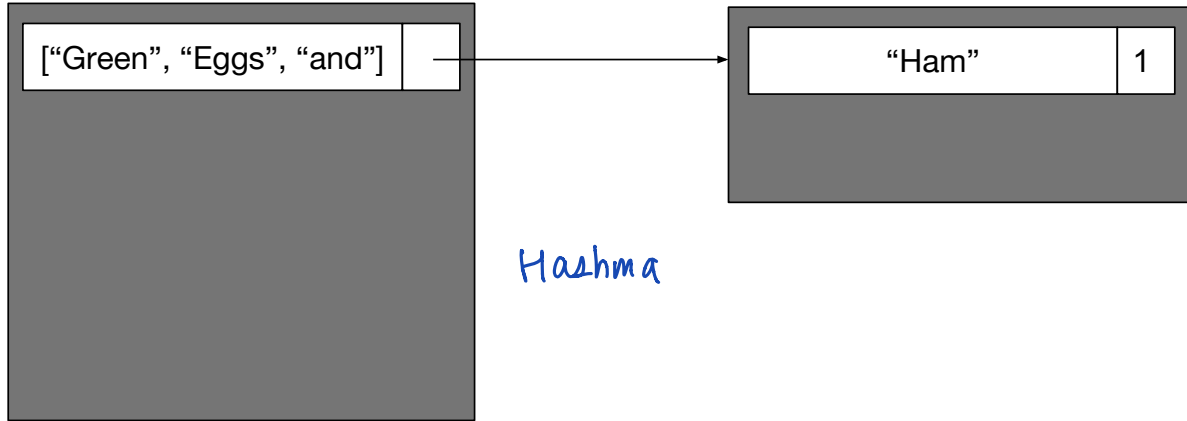
---

After `incrementPrediction(["Green", "Eggs", "and"], "Ham")` the first time



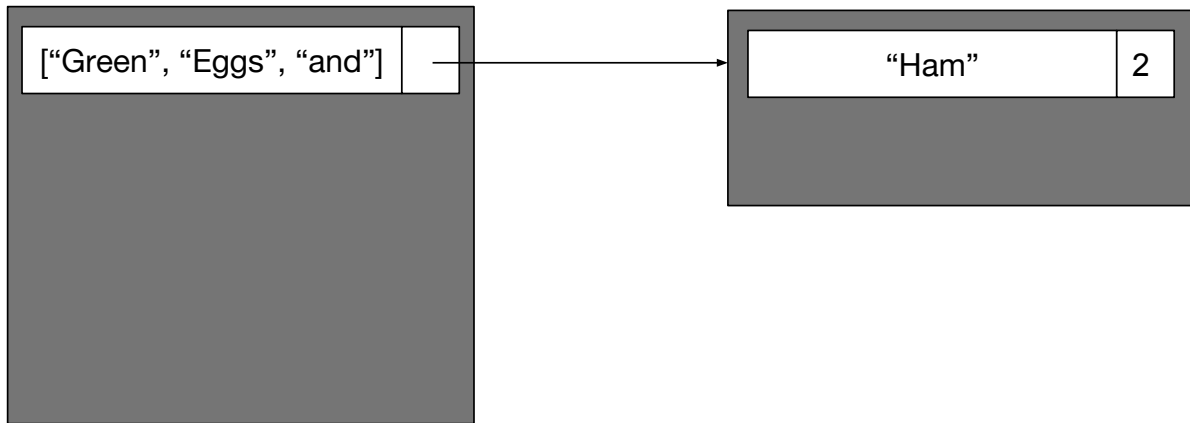
---

**predictionMap**



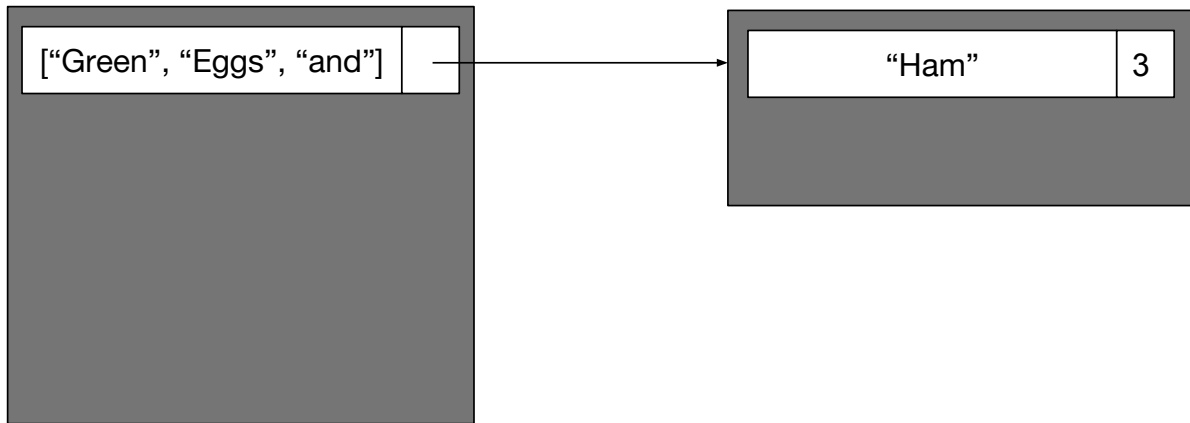
---

After `incrementPrediction(["Green", "Eggs", "and"], "Ham")` the second time



---

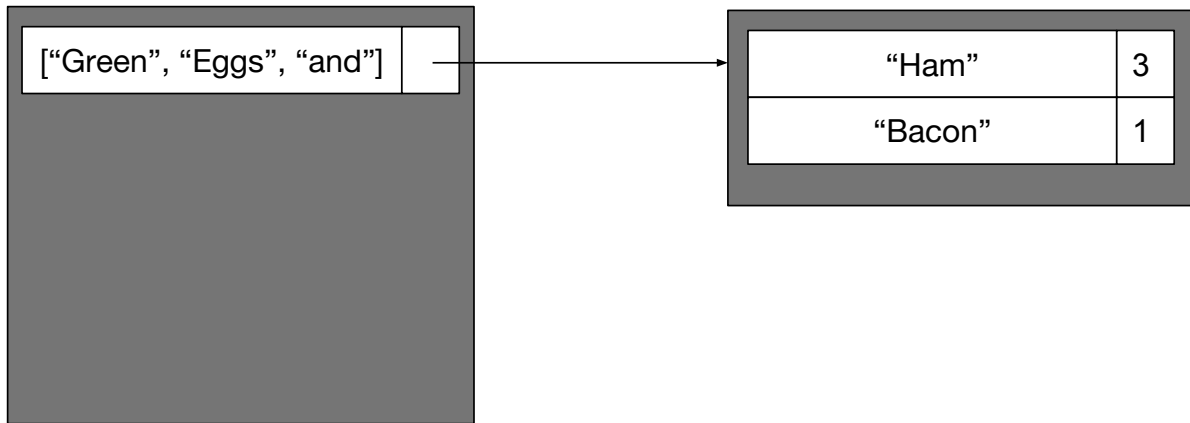
After `incrementPrediction(["Green", "Eggs", "and"], "Ham")` the third time





---

After incrementPrediction(["Green", "Eggs", "and"], "Bacon")



---

## Method number 2

```
public int trainFromText(String content);
```

**This method trains the model by reading the text word-by-word , while incrementing frequencies of occurring prefix+prediction pairs in the HashMaps.**

**\* prefix is a list of [degree] words**

**Everything lowercase.**

---

---

```
public int trainFromText(String content);
```

***“I do not like Green Eggs and Ham.”***

---

```
public int trainFromText(String content);
```

***“I do not like Green Eggs and Ham.”***



```
incrementPrediction(["i", "do", "not"], "like");
```

---

---

```
public int trainFromText(String content);
```

***“I do not like Green Eggs and Ham.”***



```
incrementPrediction(["i", "do", "not"], "like");
```

```
incrementPrediction(["do", "not", "like"], "green");
```

---


---

```
public int trainFromText(String content);
```

***“I do not like Green Eggs and Ham.”***

```
incrementPrediction(["i", "do", "not"], "like");  
incrementPrediction(["do", "not", "like"], "green");  
incrementPrediction(["not", "like", "green"], "eggs");  
incrementPrediction(["like", "green", "eggs"], "and");  
incrementPrediction(["green", "eggs", "and"], "ham.");
```

Question: What is the degree here?

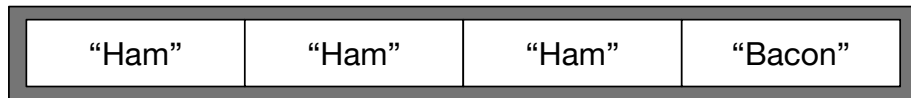


---

```
public ArrayList<String> getFlattenedList(ArrayList<String> prefix);
```



**Return a list of word associated with a specific prefix, where each unique word is repeated the same number of times as its frequency.**



---

```
public ArrayList<String> getFlattenedList(ArrayList<String> prefix);
```

## **Cache:**

A hardware or software component that stores data so future requests for that data can be served faster. Imagine the program reading the entire Harry Potter series and then writing books 8-10. Speed would be crucial.

How do we implement a cache in this assignment?

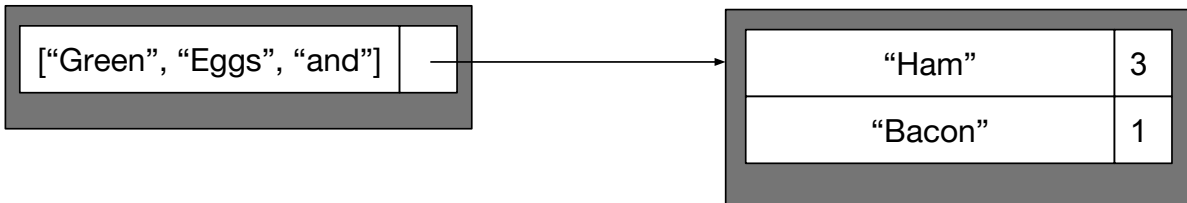
**HashMap (surprise)**

---



---

`getFlattenedList(["Green", "Eggs", "and"]);` for the **first** time



cache



Look in cache, is the key ["Green", "Eggs", "and"] in the cache? **No**

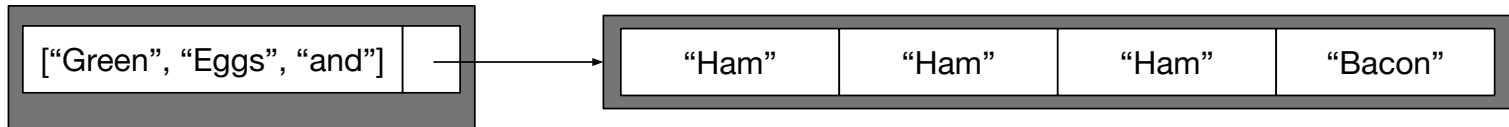
---

---

`getFlattenedList(["Green", "Eggs", "and"]);` for the first time



**cache**



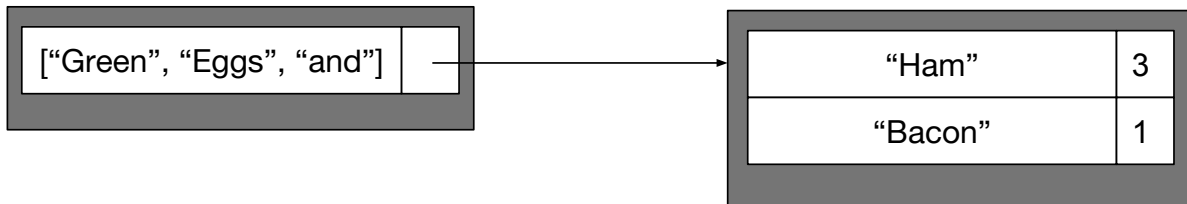
Look in cache, is the key ["Green", "Eggs", "and"] in the cache? No

Create the list,      Put it in the cache,      Return the list

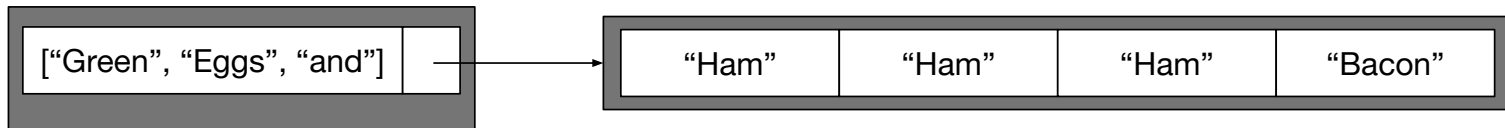
---

---

`getFlattenedList(["Green", "Eggs", "and"]);` for the **second** time



**cache**



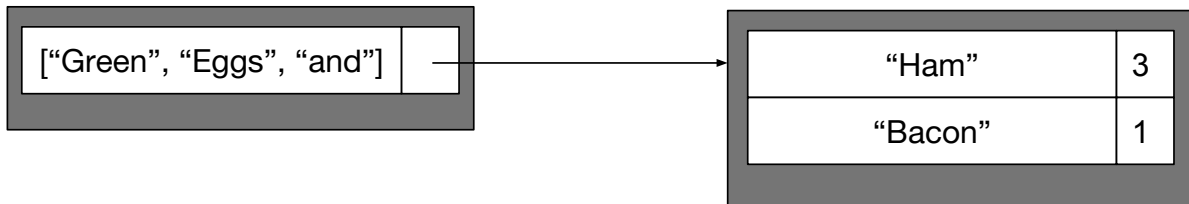
Look in cache, is the key `["Green", "Eggs", "and"]` in the cache?

**Yes**, directly return the list

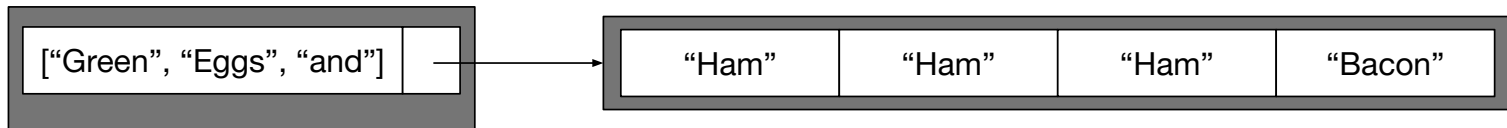
---

---

`getFlattenedList(["Green", "Eggs", "and"]);` for the **millionth** time



**cache**



**Look in cache, is the key `["Green", "Eggs", "and"]` in the cache?**

**Yes, directly return the list**

**We only needed to create the list once! The other 999,999 times we just had to access it!**

---

---

```
public String generateNext(ArrayList<String> prefix);
```

**Call getFlattenedList();**

"Ham"	"Ham"	"Ham"	"Bacon"
-------	-------	-------	---------

**Randomly return a word from this ^ list**

---

---

```
public String generate(int count)
```

**Generate a random key -> ["will", "eat", "them"]**

**Use it to generate a word (What method does this) -> "here"**

**Save this generated word as you will return all of the generated words at the end. You could use an ArrayList.**

**ArrayList -> ["will", "eat", "them", "here"]**

---

---

```
public String generate(int count)
```

Generate a random key -> ["will", "eat", "them"]

Use it to generate a word (What method does this) -> "here"

Save this generated word as you will return all of the generated words at the end. You could use an ArrayList.

ArrayList -> ["will", "eat", "them", "here"]

Now once again use the new prefix to generate another word.

The new prefix should be ["eat", "them", "here"].

Use it to generate a word -> "or"

ArrayList -> ["will", "eat", "them", "here", "or"]

---

---

```
public String generate(int count)
```

Generate a random key -> ["will", "eat", "them"]

Use it to generate a word (What method does this) -> "here"

Save this generated word as you will return all of the generated words at the end. You could use an ArrayList.

ArrayList -> ["will", "eat", "them", "here"]

Now once again use the new prefix to generate another word.

The new prefix should be ["eat", "them", "here"].

Use it to generate a word -> "or"

ArrayList -> ["will", "eat", "them", "here", "or"]

Do this until you have [count] words total.

Then return a string of all of the words.

"Will eat them here or ....."

---



---

```
public String generate(int count)
```

**The end result:**

**“will eat them here or there. i do not like them, sam-i-am. say! in the dark. not on a boat. i will eat them in a tree! not in a box? would you eat them in a tree! not in the dark. would you like then with a mouse. i do not like green eggs and ham! i do not like them in a house. i do not like them,sam-i-am. i do not like them,sam-i-am. i do not like them,sam-i-am. i do not like them with a fox. not in a box. not with a mouse. not with a goat!”**



---

---

## **CharacterModel**

**is almost exactly the same as WordModel, except each string in the ArrayLists and HashMaps are exactly one character long.**

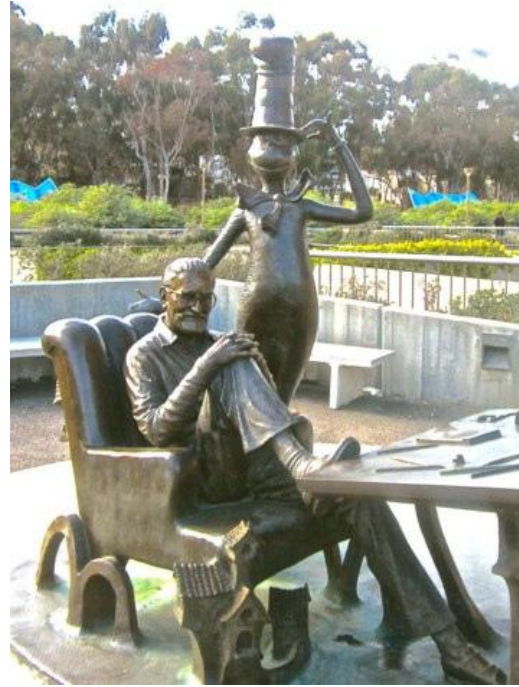
**The logic should be similar.**

---

---

---

**Go make Dr. Seuss proud!**

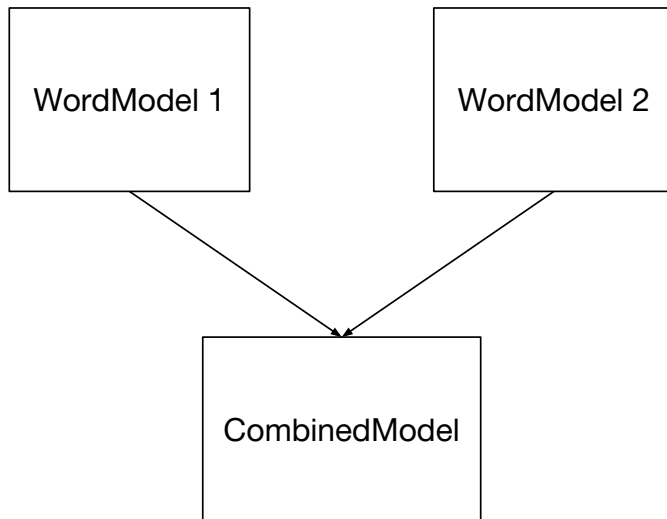




---

## Extra Credit: linear interpolation

A way to combine 2 distribution functions into 1



---

## Extra Credit: linear interpolation

$$F(w_{n+1} | w_n, w_{n-1}, \dots, w_{n-i}) = a F_1(w_{n+1} | w_n, w_{n-1}, \dots, w_{n-i}) + b F_2(w_{n+1} | w_n, w_{n-1}, \dots, w_{n-i})$$

The combined frequency of a certain phrase is

that phrase's frequency in model 1 \* weight 1

+

that phrase's frequency in model 2 \* weight 2

---

---

## Extra Credit: linear interpolation

$$F(\text{juice} \mid I, \text{like}, \text{apple}) = a F_1(\text{juice} \mid I, \text{like}, \text{apple}) + b F_2(\text{juice} \mid I, \text{like}, \text{apple})$$

$$F_1(\text{juice} \mid I, \text{like}, \text{apple}) = 1$$

$$F_2(\text{juice} \mid I, \text{like}, \text{apple}) = 2$$

$$a = 2, \quad b = 1,$$

$$F(\text{juice} \mid I, \text{like}, \text{apple}) = \quad ?$$

---

---

## Extra Credit: linear interpolation

$$F_1(\text{juice} \mid I, \text{like}, \text{apple}) = 1$$

$$F_2(\text{juice} \mid I, \text{like}, \text{apple}) = 2$$

$$a = 2, b = 1,$$

$$F(\text{juice} \mid I, \text{like}, \text{apple}) = 2F_1(\text{juice} \mid I, \text{like}, \text{apple}) + 1F_2(\text{juice} \mid I, \text{like}, \text{apple})$$

$$F(\text{juice} \mid I, \text{like}, \text{apple}) = 2 * 1 + 1 * 2 = 4$$

---



Model 1			Model 2		
Prefix	Prediction	Frequency	Prefix	Prediction	Frequency
"the"	"dog"	1	"the"	"dog"	2
	"cat"	1		"cat"	1
	"memer"	2		"sloth"	1
"a"	"bunny"	2		"memer"	8

New Model (constructed with Model 1 weight = 2, Model 2 weight = 1)		
Prefix	Prediction	Frequency
"the"	"dog"	$1*a + 2*b = 1*2 + 2*1 = 4$
	"cat"	$1*2 + 1*1 = 3$
	"sloth"	$0*2 + 1*1 = 1$
	"memer"	$2*2 + 8*1 = 12$
"a"	"bunny"	$2*2 + 0*1 = 4$

New Model (constructed with Model 1 weighing twice as much as Model 2)		
Prefix	Prediction	Frequency
"the"	"dog"	$1*a + 2*b = 1*2 + 2*1 = 4$
	"cat"	$1*2 + 1*1 = 3$
	"sloth"	$0*2 + 1*1 = 1$
	"memer"	$2*2 + 8*1 = 12$
"a"	"bunny"	$2*2 + 0*1 = 4$

In the new CombinedModel,

After the prefix **"the"**

How much more likely is **"memer"** predicted than **"cat"**?

---

## In CombinedModel.java

Constructor takes in 2 WordModels, and 2 weights

```
public CombinedModel (WordModel model1, WordModel model2, int a, int b);
```

You only need to implement one method:

```
public String generateNext(ArrayList<String> prefix);
```

To predict the next word.

---

---

## In CombinedModel.java

Hint 1: you can call any public method in WordModel, but which method actually gives you a “distribution of words”?

Hint 2: you can comfortably write this in 40 ish lines of code, if it gets super long, something might be wrong

How to check if you combined the distribution correctly?

---

---

---

Something to think about for the next assignment:

In a world where strings do not have a `length()` method, how would you count how many characters are in a string?

---