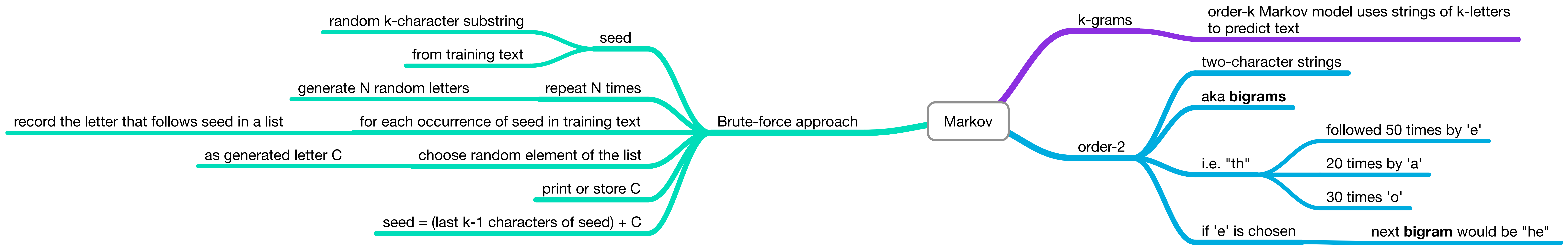
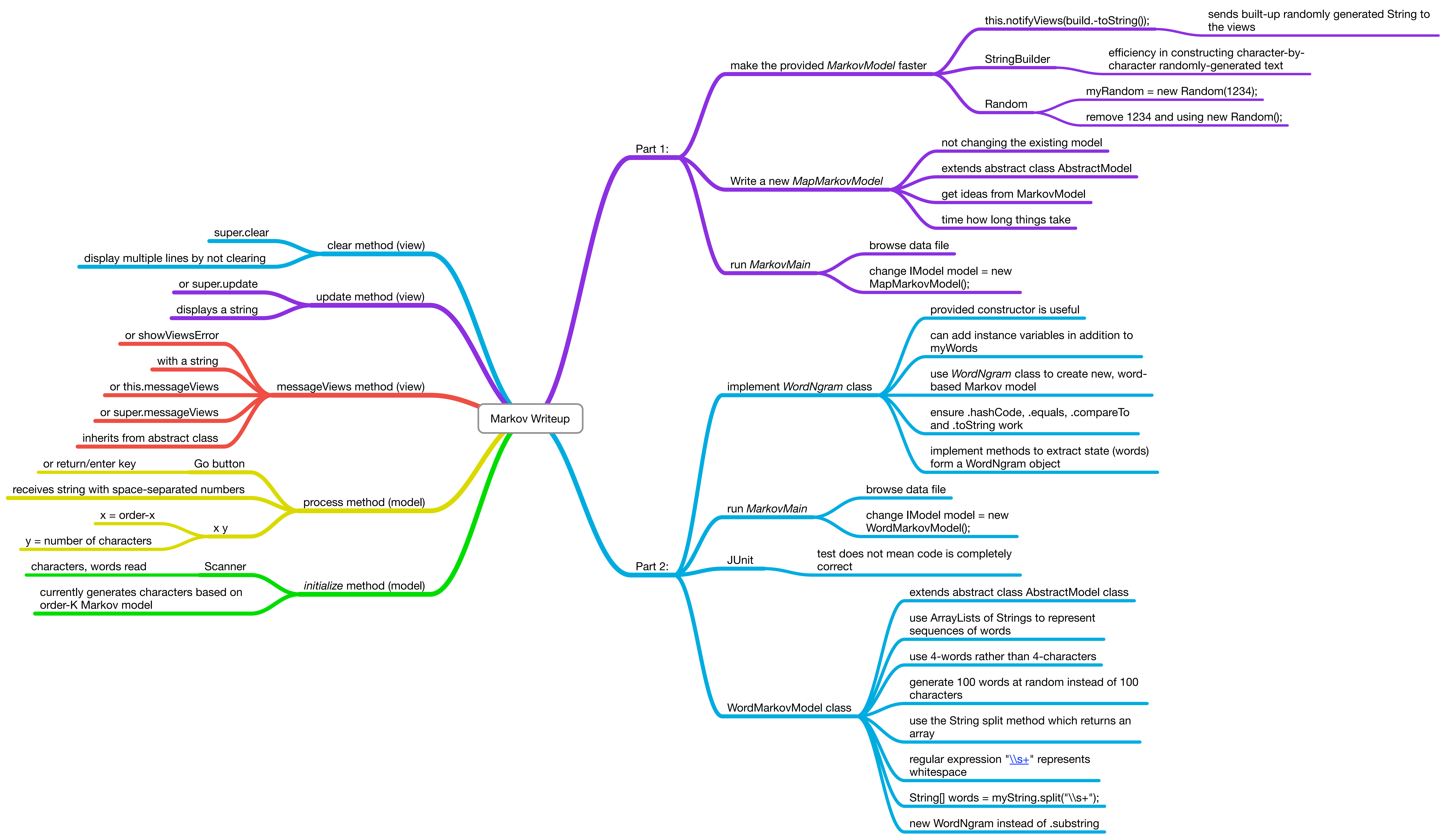
Markov
High level view
Improve performance of code that generates random text based on predicting characters
Use a map rather than **recompute** information
Write new random-text generator
Based on words rather than characters
Modify, Run, Report
On a performance test comparing HashMap & TreeMap
Given
Code that uses brute-force approach
Brute-force
Entire text forms the basis of the Markov model
It is rescanned to generate each letter of the the random text
Generate random text using order-k Markov Model
Using map model for more efficiency
Part 1: A Smarter Approach (for characters)
MapMarkovModel
Map
Keys
Values
Text is only scanned once
Represents every possible k-gram
Build this structure in its own method
Example
"bbbabbabbbbaba"
order-3 Markov model
"bbb" occurs 3 times
twice followed by an "a"
once by "b"
means 3-gram "bbb"
followed twice by "bba"
once by "bbb"
"bba" occurs 3 times
Part 2: Markov Models for words
WordNgrams
object that represents N words from text
helper class
started for us
each WordNGram is a word k-gram for WordMarkovModel map
needs to be completed
implements
.equals
.hashCode
should only be calculated once
.compareTo
WordMarkovModel
we write this class
k words are used to generate word
words replace characters
instead of k-gram consisting of k letters, a WordNgram consists of k words
instead of every k-gram represented by a String to a list of following k-grams as Strings, now the key in the map will be a WordNgram (representing every possible sequence of k-words), the associated value will be a list of k-words that follow
Part 3: Analysis
Part A
Q1
how long does it take
order-5
romeo.txt
any change from order-1 to order-10 and why?
Q2
compare Romeo and Scarlet Letter
how long to generate order-5
given timings for generating 400, 800, 1600 characters
do results match what you think?
how long generate 1600 characters using order-5 Markov when using Bible
justify with reasoning
Q3
Provide timings for
creating map
generating 200, 400, 800, 1600 character random texts
with order-5 model and romeo.txt
explanation
Part B
Analyze WordMarkovModel
using a HashMap
and hashCode function
compared with TreeMap
and compareTo function
Differences should be
Performance
as the number of keys gets large
Figure out how many keys each different lexicon generates


Markov
k-grams
order-k Markov model uses strings of k-letters to predict text
order-2
two-character strings
aka **bigrams**
i.e. "th"
followed 50 times by 'e'
20 times by 'a'
30 times 'o'
if 'e' is chosen
next **bigram** would be "he"
Brute-force approach
seed
random k-character substring
from training text
repeat N times
generate N random letters
for each occurrence of seed in training text
record the letter that follows seed in a list
choose random element of the list
as generated letter C
print or store C
seed = (last k-1 characters of seed) + C



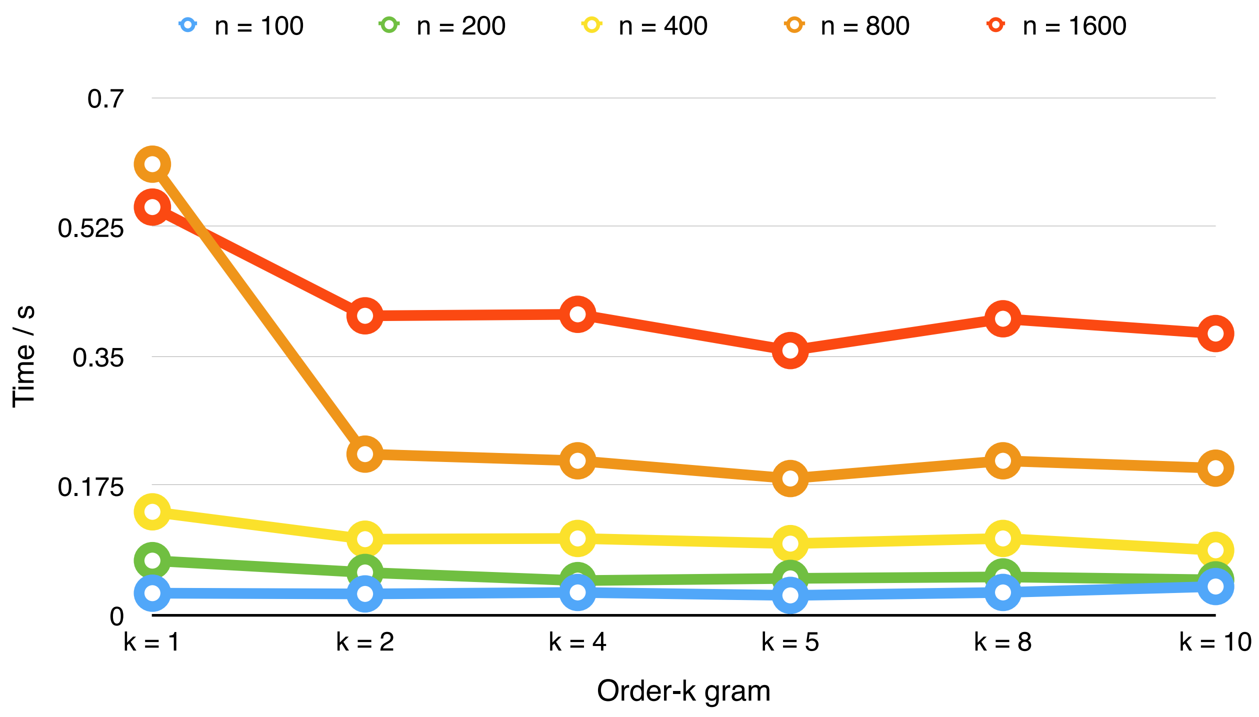
- Markov Writeup
  - Part 1:
    - make the provided *MarkovModel* faster
      - this.notifyViews(build.-toString());
        - sends built-up randomly generated String to the views
      - StringBuilder
        - efficiency in constructing character-by-character randomly-generated text
      - Random
        - myRandom = new Random(1234);
        - remove 1234 and using new Random();
    - Write a new *MapMarkovModel*
      - not changing the existing model
      - extends abstract class AbstractModel
      - get ideas from MarkovModel
      - time how long things take
    - run *MarkovMain*
      - browse data file
      - change IModel model = new MapMarkovModel();
  - Part 2:
    - implement *WordNgram* class
      - provided constructor is useful
      - can add instance variables in addition to myWords
      - use *WordNgram* class to create new, word-based Markov model
      - ensure .hashCode, .equals, .compareTo and .toString work
      - implement methods to extract state (words) form a WordNgram object
    - run *MarkovMain*
      - browse data file
      - change IModel model = new WordMarkovModel();
    - JUnit
      - test does not mean code is completely correct
    - WordMarkovModel class
      - extends abstract class AbstractModel class
      - use ArrayLists of Strings to represent sequences of words
      - use 4-words rather than 4-characters
      - generate 100 words at random instead of 100 characters
      - use the String split method which returns an array
      - regular expression "\\s+" represents whitespace
      - String[] words = myString.split("\\s+");
      - new WordNgram instead of .substring
  - clear method (view)
    - super.clear
    - display multiple lines by not clearing
  - update method (view)
    - or super.update
    - displays a string
  - messageViews method (view)
    - or showViewsError
    - with a string
    - or this.messageViews
    - or super.messageViews
    - inherits from abstract class
  - process method (model)
    - Go button
      - or return/enter key
    - receives string with space-separated numbers
    - x y
      - x = order-x
      - y = number of characters
  - *initialize* method (model)
    - Scanner
      - characters, words read
    - currently generates characters based on order-K Markov model

Question 1.

How long does it take using the provided, brute force code to generate order-5 text using Romeo and Juliet (romeo.txt) and generating 100, 200, 400, 800, and 1600 random characters. Do these timings change substantially when using order-1 or order-10 Markov models? Why?

Time to generate for order-k gram with Romeo and Juliet source text

Order-k gram for n-th characters (Averaged over 10 runs)	k = 1	k = 2	k = 4	k = 5	k = 8	k = 10
n = 100	0.03	0.029	0.031	0.027	0.031	0.039
n = 200	0.074	0.058	0.047	0.05	0.052	0.048
n = 400	0.14	0.103	0.104	0.097	0.104	0.088
n = 800	0.61	0.218	0.209	0.185	0.209	0.199
n = 1600	0.552	0.405	0.407	0.358	0.401	0.381



- For order-k grams, increase in k leads to shorter times.
- This could be because for a smaller value of k, there are a larger number of such order-k grams. Having to allocate memory to more order-k grams would increase timings.
- From an initial value of k = 1 to k = 2, the size of order-k gram for values of k fall steeply.
- From k = 2 onwards, the timings stay generally similar.

Question 2.

Romeo has roughly 153,000 characters. Hawthorne's Scarlet Letter contains roughly 500,000 characters. How long do you expect the brute force code to take to generate order-5 text when trained on hathorne.txt given the timings you observe for romeo when generating 400, 800, 1600 random characters? Do empirical results match what you think? How long do you think it will take to generate 1600 random characters using an order-5 Markov model when the King James Bible is used as the training text — our online copy of this text contains roughly 4.4 million characters. Justify your answer — don't test empirically, use reasoning.

Time to generate for order-k gram with Romeo and Juliet source text

Order-k gram for n-th characters (Averaged over 10 runs)	k = 1	k = 2	k = 4	k = 8
n = 100	0.03	0.029	0.031	0.031
n = 200	0.074	0.058	0.047	0.052
n = 400	0.14	0.103	0.104	0.104
n = 800	0.61	0.218	0.209	0.209
n = 1600	0.552	0.405	0.407	0.401

Time to generate for order-k gram with Scarlet Letter source text

Order-k gram for n-th characters (Averaged over 10 runs)	**k = 1**	**k = 2**	**k = 4**	**k = 8**
n = 100	0.093	0.079	0.069	0.067
n = 200	0.186	0.147	0.143	0.139
n = 400	0.371	0.282	0.271	0.277
n = 800	0.775	0.636	0.552	0.511
n = 1600	1.52	1.154	1.106	1.078

- Given Hawthorne's Scarlet Letter is 500,000 characters to Romeo's 153,000 characters (roughly 3.5x as large), one would expect a 3.5x increase in timings for Scarlet Letter.
- As shown in the two tables above, we see a general pattern that matches our prediction: the relative increase in the number of characters is matched by the timings of order-k grams.
- While we would expect that the timings to increase proportionately with the the same magnitude of increase in the size of the source text, we witness increasing savings in timings as the source text gets larger.
- I would guess that there is a minimum overhead required regardless of the size of the source text and subsequent increases in the size result in timing increases that are smaller in magnitude.

Question 3

Provide timings using your Map/Smart model for both creating the map and generating 200, 400, 800, and 1600 character random texts with an order-5 Model and romeo.txt. Provide some explanation for the timings you observe.

Time to generate for order-5 gram with different models

Order-k gram for n-th characters (Averaged over 10 runs)	**Brute-force method**	**Map-Markov method**
n = 100	0.028	0.0010
n = 200	0.07	0.0010
n = 400	0.094	0.020
n = 800	0.196	0.030
n = 1600	0.483	0.050

- We see incredible time savings with the Map Markov model in comparison to the brute-force method.
- The savings are often greater than a order of magnitude, see t = 0.07s v. t = 0.0010 (for n = 200) and t = 0.196s v. t = 0.03 (for n = 800).
- For the Map-Markov model, the time taken to create the map is not taken into account. The map is only created once and is only recreated for a new value of k.
- Because the map is already created, it is much for efficient to access and retrieve the data from an available data source.
- Contrast this to the brute-force method, which had to generate the data order-k gram each time by iterating over the source text each time.

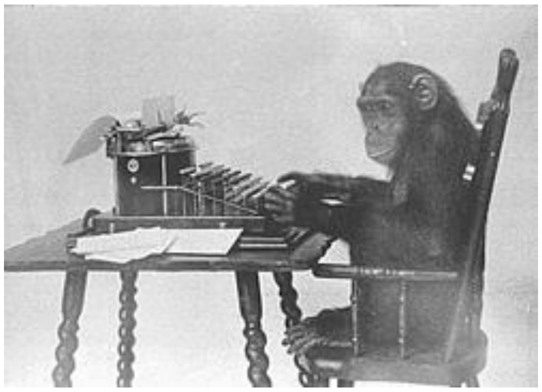
CompSci 201

Data Structures and Algorithms

Markov

See the How To pages for details on creating projects, files, and so on. The pages here describe in broad strokes what this assignment is about.

Genesis of the Markov Assignment



(image from Wikipedia, public domain)
This assignment has its roots in several places, with its modern description as the Infinite Monkey Theorem.

The true mathematical roots are from a 1948 monolog by Claude Shannon, A Mathematical Theory of Communication which discusses in detail the mathematics and intuition behind this assignment. This article was popularized by AK Dewdney in both Scientific American and later reprinted in his books that collected the articles he wrote.

Some examples of where Shannon's mathematical theory have been applied include:

- In 2005 two students at MIT had a randomly-generated paper accepted at a conference. Their paper-generating tool, SCIgen, has a Wikipedia entry with links to the program.
- In 1984 an Internet Personality named Mark V Shaney participated in the then chat-room-of-the time us-

ing randomly generated text as described in this Wikipedia entry.

- In December 2008 Herbert Schlangemann had a auto-generated paper accepted and published in an IEEE “conference” (quotes used judiciously).

Joe Zachary developed this into a nifty assignment in 2003 and the folks at Princeton have been using this as the basis for an assignment since 2005 and possibly earlier.

More information on Markov models can be found here.

High Level View of Assignment

You’ll do three things for this assignment.

1. Improve the performance of code that generates random text based on predicting characters. For this you’ll use a map to store information rather than (re)computing the information.
2. Write a new random-text generation program based on words rather than characters.
3. Modify, run, and report on a performance test comparing `HashMap` and `TreeMap`implementations of the `Map` interface.

You’ll be provided with code that uses a brute-force approach to generate random text using an order-k Markov Model based on characters. You’ll first improve the code to make it more efficient, then you’ll write a new model based on words rather than on characters.

The term *brute-force* refers to the characteristic that *the entire text that forms the basis of the Markov Model is rescanned to generate each letter of the random text.*

For the first part of the assignment you’ll develop and use a Map (with other appropriate structures as keys and values) so that the text is scanned only once. When you scan once, your code will store information so that generating random text requires looking up information rather than rescanning the text.

Part 1: A Smarter Approach (for characters)

You will add a class, MapMarkovModel, based on the class MarkovModel that implements the following smarter approach for generating markov models. Make sure that you understand the brute-force method described here before trying to implement the smarter approach.

Instead of scanning the training text N times to generate N random characters, you'll first scan the text once to create a structure representing every possible k-gram used in an order-k Markov Model. You may want to build this structure in its own method before generating your N random characters.

Example

Suppose the training text is `"bbbabbabbbbaba"` and we're using an order-3 Markov Model.

The 3-letter string (3-gram) `"bbb"` occurs three times, twice followed by 'a' and once by 'b'. We represent this by saying that the 3-gram `"bbb"` is followed twice by `"bba"` and once by `"bbb"`. That is, the **next** 3-gram is formed by taking the last two characters of the seed 3-gram `"bbb"`, which are `"bb"` and adding the letter that follows the original 3-gram seed.

The 3-letter string `"bba"` occurs three times, each time followed by 'b'. The 3-letter string *"bab"*occurs three times, followed twice by 'b' and once by 'a'. However, we treat the original string/training-text as circular, i.e., the end of the string is followed by the beginning of the string. This means `"bab"` also occurs at the end of the string (last two characters followed by the first character) again followed by 'b'.

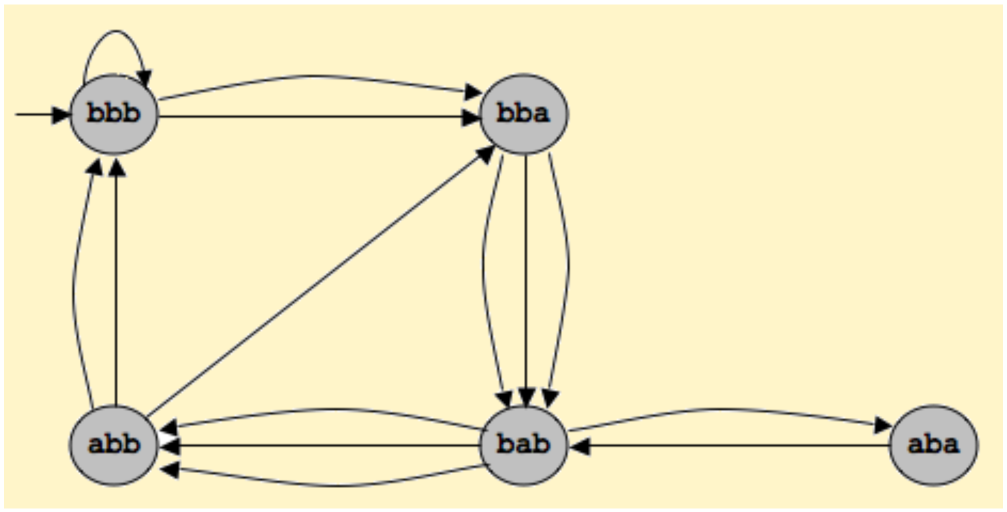
In processing the training text from left-to-right we see the following transitions between 3-grams starting with the left-most 3-gram `"bbb"`

```
  bbb -> bba -> bab -> abb -> bba -> bab ->abb -> bbb ->bbb -> bba -> bab -> aba ->bab -> abb
-> bbb
```

This can be represented as a map of each possible three grams to the three grams that follow it:

3-GRAM	FOLLOWING 3-GRAMS
bbb	bba, bbb, bba

bba	bab, bab, bab
bab	abb, abb, aba, abb
abb	bba, bbb, bbb
aba	bab



This information can also be represented in a state diagram (from the Princeton website).

Your Code

In your code you'll replace the brute-force re-scanning algorithm for generating random text based on characters with code that builds a data structure that you'll then use to follow the state transitions diagrammed above. Specifically, you'll create a map to make the operation of creating random text more efficient.

Keys in the map are k-grams in a k-order Markov model. The value associated with each key is a list of related k-grams. Each different k-gram in the training text will be a key in the map. The value associated with a k-gram *key* is a list of every k-gram that follows *key* in the training text.

The list of k-grams that constitute a value should be in order of occurrence in the training text. That is, you should start generating your list of following grams from the beginning of your training text. See the table of **3-grams** above as an example using the training text *"bbbabbabbbbaba"*. Note that the 3-gram key *"bbb"* would map to the list *["bba", "bbb", "bba"]*, the 3-gram key *"bba"*would map to the list *["bab", "bab", "bab"]*, and the 3-gram key *"abb"* would map to the list *["bba", "bbb", "bbb"]*

Just like in the brute method, to generate random text your code should generate an initial seed k-gram at random from the training text, exactly as in the brute-force approach. Then use the pseudo-code outlined below.

```
  seed = random k-character substring (k-gram) from the training text (key) --- the initial seed
  repeat N times to generate N random letters
     find the list (value) associated with seed (key) using the map
     next-k-gram = choose a random k-gram from the list (value)
     print or store C, the last character of next-k-gram
     seed = next-k-gram    // Note this is (last k-1 characters of seed) + C
```

Construct the map once — don't construct the map each time the user tries to generate random text unless the value of *k* in the order-k Markov model has changed. See the howTo pages for details on the class you'll implement and how it fits into the other classes.

Part 2: Markov Models for Words

In this part of the assignment you'll use the character-generating Markov code you wrote as a model to create a new program that generates word models instead of character models. Your program will generate random words based on the preceding word k-grams. You'll create a `WordMarkovModel` class that doesn't use brute force. A helper class has been started for you, `WordNgrams`, where each WordNGram will be a word k-gram for your `WordMarkovModel` map. You will need to finish `WordNGrams`.

In the k-order Markov model with letters you just coded, k characters are used to predict/generate another character. In a k-order word Markov model k words are used to predict/generate another word — words replace characters. Here instead of a k-gram consisting of **k** letters, a `WordNgram` consists of **k** words. In your previous program you mapped every k-gram represented by a String to a list of following k-grams as Strings in this new program the key in the map will be a `WordNgram` representing every possible sequence of k-words. The associated value will be a list of the word k-grams that follow.

You need to write two classes:

- WordNgram: an object that represents N words from a text. This class has been started for you.
- WordMarkovModel: you need to write this class to generate a k-order Markov model random text

See the howTo pages for details.

Your `WordNgram` class must correctly and efficiently implement `.equals`, `.hashCode`, and `.compareTo`so that `WordNgram` objects can be used in HashMap or TreeMap maps. Although you are not directly calling `.equals`, `.hashCode`, and `.compareTo`, these methods are needed by HashMap and TreeMap.

For full credit the `hashCode` value should be calculated once, not every time the method is called.

Document all .java files you write or modify. Submit all the .java files in your project. You should also submit a Analysis.pdf (described below) and README.

Part 3: Analysis

The analysis has 2 parts.

Part A

Answer these questions:

1. How long does it take using the provided, brute force code to generate order-5 text using Romeo and Juliet (romeo.txt) and generating 100, 200, 400, 800, and 1600 random characters. Do these timings change substantially when using order-1 or order-10 Markov models? Why?
2. Romeo has roughly 153,000 characters. Hawthorne's Scarlet Letter contains roughly 500,000 characters. How long do you expect the brute force code to take to generate order-5 text when trained on hathorne.txt given the timings you observe for romeo when generating 400, 800, 1600 random characters? Do empirical results match what you think? How long do you think it will take to generate 1600 random characters using an order-5 Markov model when the King James Bible is used as the training text — our online copy of this text contains roughly 4.4 million characters. Justify your answer — don't test empirically, use reasoning.
3. Provide timings using your Map/Smart model for both creating the map and generating 200, 400, 800, and 1600 character random texts with an order-5 Model and romeo.txt. Provide some explanation for the timings you observe.

Part B

The goal of the second part of the analysis is to analyze the performance of WordMarkovModel using a HashMap (and the hashCode function you wrote) and a TreeMap (and the compareTo function you wrote). The main difference between them should be their performance as the number of keys (that is WordNGrams as keys in your map) gets large. So set up a test with the lexicons we give you and a few of your own. Figure out how many keys each different lexicon generates (for a specific number sized n-gram). Then generate some text and see how long it takes.

Graph the results. On one axis you'll have the number of keys, on the other you'll have the time it took to generate a constant of words (you decide…choose something pretty big to get more accurate results). Your two lines will be HashMap and TreeMap. Try to see if you can see any differences in their performance as the number of NGrams in the map get large. If you can't, that's fine. Briefly write up your analysis (like 1 or 2 paragraphs) and include both that and the graph in a PDF you submit.

Call the file Analysis.pdf so that our checking program can know that you submitted the right thing.

High-Level Grading

CRITERIA	POINTS
Map-based Markov Model works and is more efficient than brute force code given	5
WordNGram Markov Model works and meets the requirements of hashCode and Comparable	5
Analysis	5

More High-Level Grading

To get full credit on an assignment your code should be able to do the following.

- Create new classes without overwriting old classes.
- Don't unnecessarily repeat expensive computations (making maps maybe?).
- Results should be consistent.
- Overwritten methods should operate correctly (equals, compareTo, etc).
- Answer write-up questions using full thoughts/sentences. Explain clearly or demonstrate with data.
- You *MUST* create and submit a README with your work.
- If you have questions about the assignment that are not specific to your code, please ask on Piazza and

we will be happy to clarify. If your question is specific to the code you've written please post it privately on Piazza, or bring it to the Link or Office Hours and don't share it directly with your classmates.

This list is not a complete rubric for grading, and there is a better breakdown on the main page. Please start early, work carefully, and ask any questions you have and we'll do our best to make sure that everyone gets all of the points that they deserve.

CompSci 201

Data Structures and Algorithms

Markov — Background

Background on Markov Models

An order-k Markov model uses strings of k-letters to predict text, these are called **k-grams**.

An order-2 Markov model uses two-character strings or **bigrams** to calculate probabilities in generating random letters. For example suppose that in some text that we're using for generating random letters using an order-2 Markov model, the bigram **"th"** is followed 50 times by the letter 'e', 20 times by the letter 'a', and 30 times by the letter 'o', because the sequences "the", "tha" and "tho" occur 50, 20, and 30 times, respectively while there are no other occurrences of "th" in the text we're modeling.

Now suppose that we want to generate random text. We generate the bigram "th" and based on this we must generate the next random character using the order-2 model. The next letter will be an 'e' with a probability of 0.5 (50/100); will be an 'a' with probability 0.2 (20/100); and will be an 'o' with probability 0.3 (30/100). If 'e' is chosen, then the next bigram used to calculate random letters will be **"he"** since the last part of the old bigram is combined with the new letter to create the next bigram used in the Markov process.

A Brute-Force Approach

In general here's pseudo-code to generate random letters (and thus random text) using an order-k Markov model and a **training text** from which the probabilities are calculated.

```
seed = random k-character substring from the training text --- the initial seed
repeat N times to generate N random letters
   for each occurrence of seed in training text
      record the letter that follows the occurrence of seed in a list
   choose a random element of the list as the generated letter C
```

```
        print or store C
        seed = (last k-1 characters of seed) + C
```

Using this algorithm, whose Java equivalent is here, here are a series of 200-randomly generated letters using the order-k Markov model. The training text for these models are speeches made by Barack Obama as his acceptance speech at the Democractic National Convention, John McCain in his speech to the American Legion, and Sarah Palin in her acceptance speech as McCain's vice president (all late August, 2008). See training texts for the training data files.

ORDER-*K*	**OBAMA**	**MCCAIN**	**PALIN**
2	n rentrion't jobackeentry I knompin he and the in ard ould th cou'll spow. We forponowelves chat, agethilonow, age thembecal th. Ted judelisted we we all Hareaven the se a nuclas nom I've then no muc	nd, al ditationg tationtionfits one the of minal orthe re VA lain Kostreed ve rave etens. I we Frans. Firces, me wore litypecest stareparefte and the millse wilignatte shosed make thend truslair comen	ur arever ser; a going therven cal camend weetbacker, antater. Tod Theld prom het younto tol; and the withar kidepe, whe just whe could the wout to be ounted we ations. Oileaterall cles it st ou, This
3	y new dreams, have same time that prevel, I save of that eign purplummethe me from tough," to Democrations onces. I've have for our smally keeps talking American it. The new lege have by are future	retary, in mand our VA heall you harity Counding in next continue that, in as of ther. I will the are to jointo elied to proverns a serve in a rought too gress well to essed. Pam elems will in an that	righter Sept. 11 of highbors allies righ school, to serving on a finisher in thank as Sen. He who he PTA and things as but to hear. Todd and polities safer is of Amercial elebrator troops and thing t
4	nse on is at we his countries to progress for you'll investments across America, the United overty? Pull your own. Our grace the fuels. And just money work better an America's promise of go to resourc	e left their cours extend upon the good. I belonger has the enterests of our respecial very invites are not force for peace, and firmly or she work of freedom the VA should make got lost conce introdu	ct a greats and to threat effort for the people who have places week public office and when Sen. Join ourse, Sen. McCain, for that I didn't just his opport forms, and the PTA and to just at all the ol
5	y watched my life is just takes us from harm's way in theirs are differences; that the best hope that young veterans to take more accountry,	es than the anti-America receive benefit is not the people of Arizona, and their home are among those complaints about aiming a few ex-	ough, he'll be honoring the way, Todd is only one expects us today. I never really set out of five of America, then we just your average

	and science, and more accompanies that we need to record's	peditiously assess, and women of us returned from misrepresential	"hockey mom" in American president of the United States. Well,
8	gy independence on something firmer, and more honest in our battlefields may be Democrats and Republican nominee, John McCain, I will stop giving them to companies stop discriminating against those wi	administration will do all that we can do, in less trying and treat conditions that predominantly or exclusively affect women. And here the Veterans' service to one another, of leaving no one behind,	, Todd and I met way back in high school, and I promised him a little surprise for the right to vote. I think as well today of two other women who came before me in national Guard, that's not why the

Here's the Java code that implements the brute-force solution to generate *numLetters* at random from the training-text in instance variable *myString* using an order-k Markov model.

```
public void brute(int k, int numLetters) { // pick random k-character substring as initial seed int start = myRandom.nextInt(myString.length() – k + 1); String seed = myString.substring(start, start + k); // copy first k characters to back to simulate wrap-around String wrapAroundString = myString + myString.substring(0,k); StringBuilder build = new StringBuilder(); ArrayList<Character> list = new ArrayList<Character>(); for (int i = 0; i < numLetters; i++) { list.clear(); int pos = 0; while ((pos = wrapAroundString.indexOf(seed, pos)) != -1 && pos < myString.length()) { char ch = wrapAroundString.charAt(pos + k); list.add(ch); pos++; } int pick = myRandom.nextInt(list.size()); char ch = list.get(pick); build.append(ch); seed = seed.substring(1) + ch; } }
```

The code above works fine, but to generate *N* letters in a text of size *T* the code does *NT* work since it rescans the text each time a character is found.

CompSci 201

Data Structures and Algorithms

Markov — Full Writeup

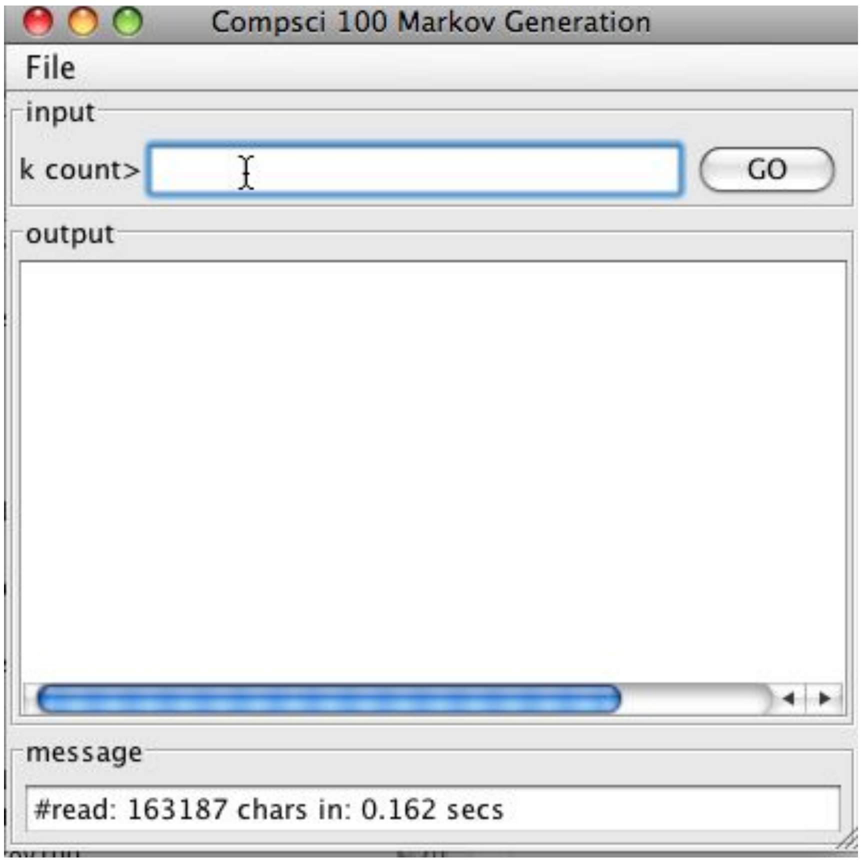
Submit using *markov* as the assignment name. Submit a README, all .java files, and the Analysis.pdf.

As the assignment writeup explains, you must:

- Part 1: make the provided class `MarkovModel` faster (by writing a new model `MapMarkovModel`, not changing the existing model)
- Part 2 (A): implement the `WordNgram` class
- Part 2 (B): use the `WordNgram` class to create a new, word-based Markov Model.

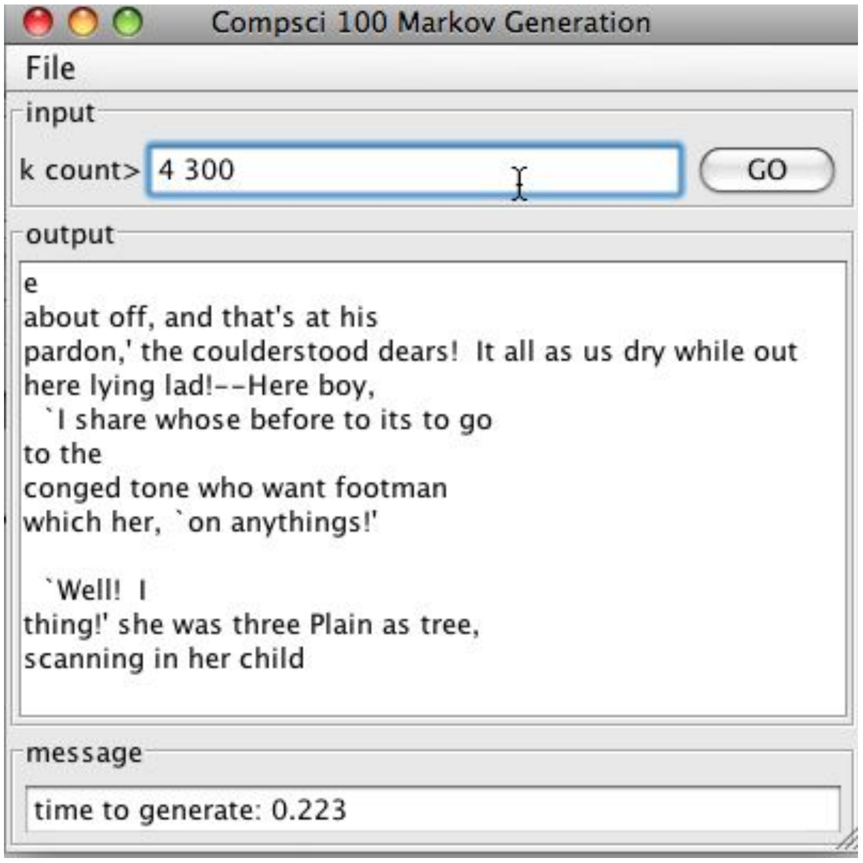
After you snarf the assignment run MarkovMain using the brute-force Markov generator.

Using the GUI, as shown below, go to the File menu->browse and select a data file, each of which is a lexicon, as the training text. There is a data directory provided when you snarf containing several files you can use as training texts when developing your program.



```
#read: 163187 chars in: 0.162 secs
```

A screen shot of the program just after alice.txt has been loaded. Your computer will probably not load in the file in the exact same time.



```
4 300
```

```
e
about off, and that's at his
pardon,' the coulderstood dears!  It all as us dry while out
here lying lad!--Here boy,
  `I share whose before to its to go
to the
conged tone who want footman
which her, `on anythings!'

  `Well!  I
thing!' she was three Plain as tree,
scanning in her child
```

```
time to generate: 0.223
```

A screen shot after generating 300 random characters using an order-4 Markov model with *Alice in Wonderland* (alice.txt) as the training text.

Code Overview

The communication between the GUI/View and the model works as follows from the model class perspective:

- The `initialize` method of the model is called with a Scanner from which characters and words are read. The code you have been given already reads all the characters into a string and uses this string to generate characters "at random", but based on their frequency in an order-k Markov model.
- The `process` method is called when the user presses the *GO* button or the return/enter key in the text field of the GUI. The convention in this program is that the Object passed to `process`is a String that contains space-separated numbers representing the order *k* of the Markov model and the number of letters to generate (or number of words). For example, "4 300" will generate 300 random letters using a 4-gram Markov model. See the existing code for examples.
- The model communicates with the view to send a message or an error/message by calling`messageViews` with a String, or `showViewsError` with a String, respectively. You can either use`messageViews`, `this.messageViews` or `super.messageViews` since the abstract class from which your model inherits contains these methods.
- The model sends lots of information to the view by repeatedly calling `update` or `super.update`with a String. The `update` method in the views displays a string. To clear the output in the views, call `super.clear` (or `clear`). To display multiple lines, either construct one string to pass to `update` or call `update` repeatedly without clearing.
- In the code you're given in MarkovModel the call `this.notifyViews(build.toString());` sends the built-up randomly-generated String to the views — see method `brute` in the code.
- The code uses a `StringBuilder` for efficiency in constructing the character-by-character randomly-generated text.

Part 1

You'll implement a class named `MapMarkovModel` that extends the abstract class AbstractModel. You should use the existing MarkovModel class to get ideas for your new `MapMarkovModel` class.

You can modify MarkovMain to use your model by simply changing one line.

```
public static void main(String[] args){
    IModel model = new MapMarkovModel(); // this is the only change!
    SimpleViewer view = new SimpleViewer("Compsci 100 Markov Generation",
        "k count>");
    view.setModel(model);
}
```

Testing Your New Model

As the writeup describes, you'll use a map instead of re-scanning the text for every randomly-generated character. To test that your code is doing things faster and not differently you can use the same text file and the same markov-model. If you use the same seed in constructing the random number generator used in your new model, you should get the same text, but your code should be faster. You'll need to time the generation for several text files and several k-orders and record these times with your explanation for them in the Analysis you submit with this assignment.

Debugging Your Code

It's hard enough to debug code without random effects making it harder. In the MarkovModel class you're given the `Random` object used for random-number generation is constructed thusly: myRandom = new Random(1234); Using the seed `1234` to initialize the random-number stream ensures that the same random numbers are generated each time you run the program. Removing `1234` and using `new Random()`will result in a different set of random numbers, and thus different text, being generated each time you run the program. This is more amusing, but harder to debug. If you use a seed of `1234` in your smart/Map model you should get the same random text as when the brute-force method is used. This will help you debug your program because you can check your results with those of the code you're given which you can rely on as being correct.

Part 2

Similarly to above, you can modify MarkovMain to use your word model by simply changing one line.

```
public static void main(String[] args){
    IModel model = new WordMarkovModel(); // this is the only change!
    SimpleViewer view = new SimpleViewer("Compsci 100 Markov Generation",
        "k count>");
    view.setModel(model);
}
```

Remember that for your `WordMarkovModel` you need to first develop and test the `WordNgram`.

The WordNgram class

The WordNgram class has been started for you. You can create new constructors or change the constructor given, though the provided constructor will likely be useful. You can also add instance variables in addition to `myWords`.

You'll need to ensure that `.hashCode`, `.equals`, `.compareTo` and `.toString` work properly and efficiently. You'll probably need to implement additional methods to extract state (words) from a`WordNgram` object. In my code, for example, I had at least two additional methods to get information about the words that are stored in the private state of a `WordNgram` object.

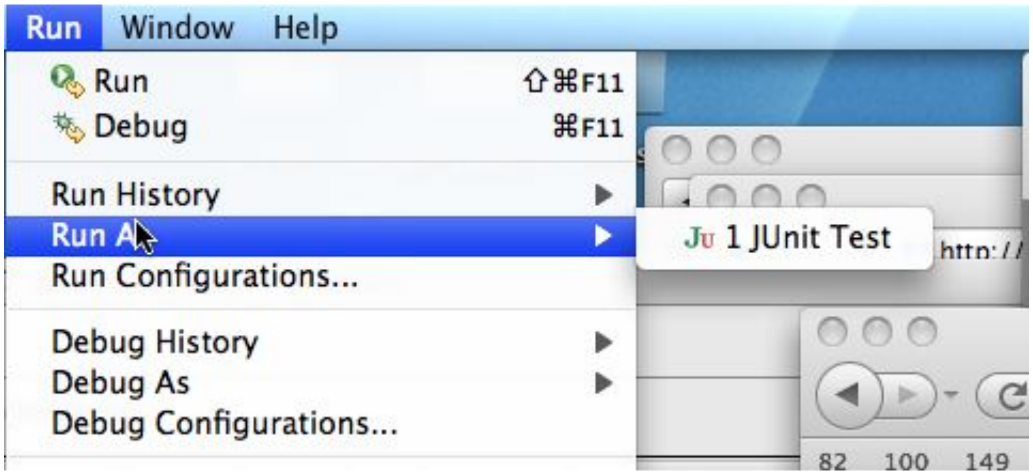
To facilitate testing your `.equals` and `.hashcode` methods a JUnit testing program is provided. You should use this, and you may want to add more tests to it in testing your implementation.

Testing with JUnit shows that a method passes some test, but the test may not be complete. For example, your code will be able to pass the the tests for `.hashCode` without ensuring that objects that are equal yield the same hash-value. That should be the case, but it's not tested in the JUnit test suite you're given.

Using JUnit

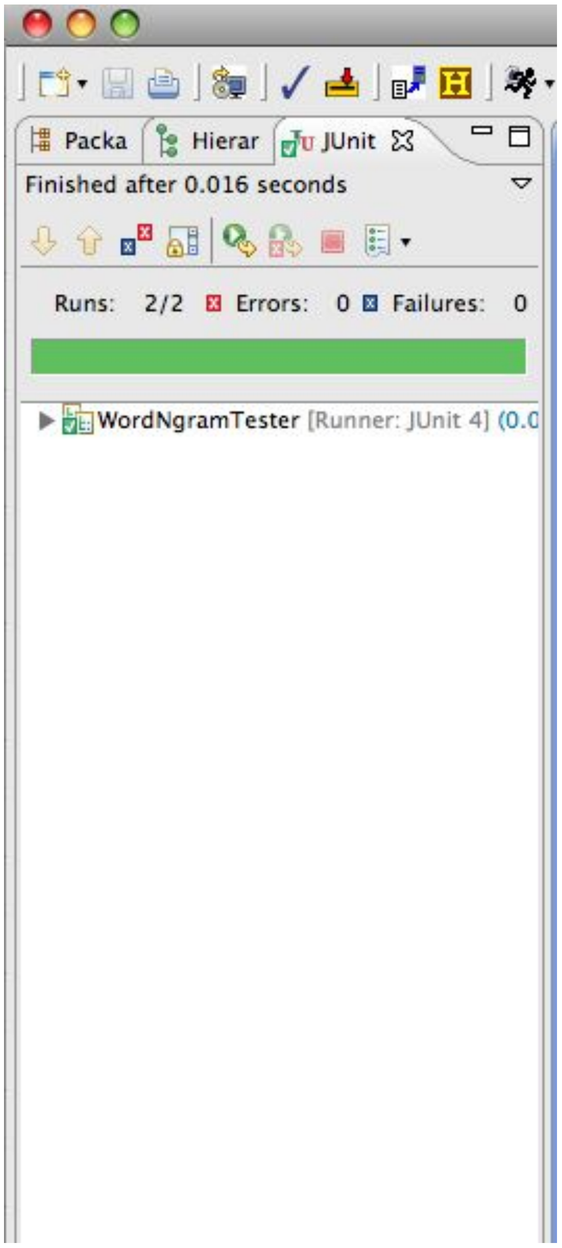
To test your `WordNgram` class you're given testing code. This code tests individual methods in your class, these tests are called *unit tests* and so you need to use the standard JUnit unit-testing library with the Word-NgramTest.java file to test your implementation.

To choose *Run as JUnit test* first use the *Run As* option in the Run menu as shonw on the left below. You have to select the JUnit option as shown on the right below. Most of you will have that as the only option.



There are two tests inWordNgramTest.java: one for the correctness of `.equals` and one for the “performance” of the `.hashCode` method.

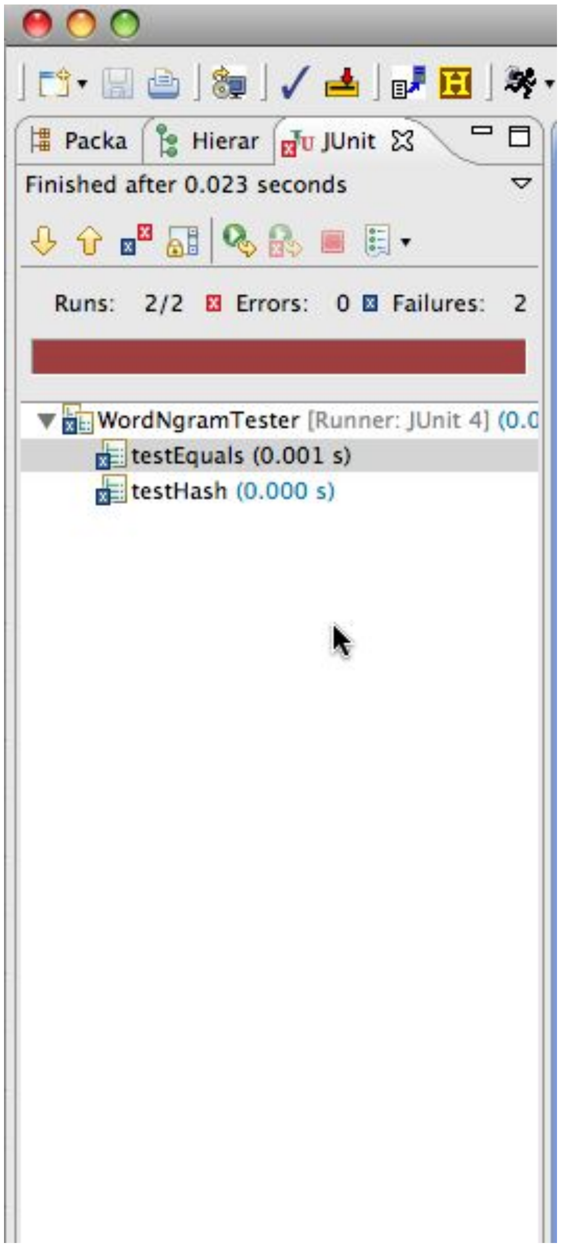
If the JUnit tests pass, you’ll get all green as shown on the left below. Otherwise you’ll get red — on the right below — and an indication of the first test to fail. Fix that, go on to more tests. The red was obtained from the code you’re given. You’ll work to make the testing all green.



Finished after 0.016 seconds

Runs: 2/2 Errors: 0 Failures: 0

WordNgramTester [Runner: JUnit 4] (0.0



Finished after 0.023 seconds

Runs: 2/2 Errors: 0 Failures: 2

WordNgramTester [Runner: JUnit 4] (0.0
testEquals (0.001 s)
testHash (0.000 s)

The WordMarkovModel class

You'll implement a class named `WordMarkovModel` that extends the abstract class AbstractModelclass. This should be very similar to the `MapMarkovModel` class you wrote, but this class uses words rather than characters.

A sequence of characters was stored as a String in the code for character-oriented Markov models. For this program you'll use ArrayLists (or arrays) of Strings to represent sequences of words used in the model.

The idea is that you'll use 4-words rather than 4-characters in predicting/generating the next word in an order-4 *word based* Markov Model. You'll need to construct the map-based`WordMarkovModel` and implement its methods so that instead of generating 100 characters at random it generates 100 words at random (but based on the training text it reads).

To get all words from a String use the String `split` method which returns an array. The regular expression `"\\s+"` represents any whitespace, which is exactly what you want to get all the words in file/string.

String[] words = myString.split("\\s+"); Using this array of words, or a wrap-around-version of it as was the case with characters in the character-based model, you'll construct a map in which each key, a `WordNgram` object, is mapped to a list of `WordNgram` objects — specifically the n-grams that follow it in the training text. This is exactly what your `MapMarkovModel` did, but it mapped a String to a list of Strings. Each String represented a sequence of k-characters. In this new model, each `WordNgram` represents a sequence of k-words. The concept is the same.

Comparing Words and Strings in the Different Models

In the new `WordMarkovModel` code you write you'll conceptually replace Strings in the map with WordNgrams. In the code you wrote for maps and strings, calls to `.substring` will be replaced by calls to `new WordNgram`. This is because `.substring` creates a new String from parts of another and returns the new String. In the `WordMarkovModel` code you must create a new `WordNgram` from the array of strings, so that each key in the word-map, created by calling new, corresponds to a string created in your original program created by calling substring.

Things to be Careful Of

A few things students often mess up in this assignment:

1. Your faster map-based character MarkovModel should generate exactly the same text as the brute force method, given the same datafile and k value. Oftentimes there can be a problem that will make the code not select the first word in the same way and so they texts will be different. Test and make sure your code matches.
2. Your Map should not be recomputed if the k value is not changed
3. Make sure your WordNGram passes all the unit tests
4. Make sure you answer all the questions in the analysis