# Markov - Compsci 201

The code for this assignment is available through Snarf as well as on the course webpage.

Ambient: https://www.cs.duke.edu/csed/ambient/

**Java Help**

You should expect to take some time getting used to the Java language. There is a cheat sheet available to relate basic Java to Python and MATLAB syntax that you may be more familiar with.

http://www.cs.duke.edu/courses/fall13/compsci201/Help/PythonMatlabJava.pdf

Additionally the Java API docs (http://docs.oracle.com/javase/7/docs/api/) can be of use when looking up how specific method or classes work.

Finally if you have a specific question about a method or a certain class, it is highly recommended to Google the specific thing you are looking for (i.e. "Java String substring method"). This will often turn up exactly what you are looking for.

Finally your classmates and TAs are available on Piazza to answer any questions you have. **DO NOT post your own code on Piazza publically. This will be removed and is in violation of course policy (sharing code with classmates is not allowed).** You can post your code in a private Piazza question though, although it is often more helpful to bring specific code questions to office hours.

## INTRODUCTION:

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.

You will do three things for this assignment.

1. Make the provided class MarkovModel faster (by writing a new model MapMarkovModel, not changing the existing model)
2. Implement the WordNgram class
3. 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.

## 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.

```
// the initial seed
seed = random k-character substring from the training text

// 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
```

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.

## MapMarkovModel

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);
}
```

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. You should also note that **if you generate random text more than once with the same value of k, you will not need to regenerate this structure and doing so will cost you points.**

**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.

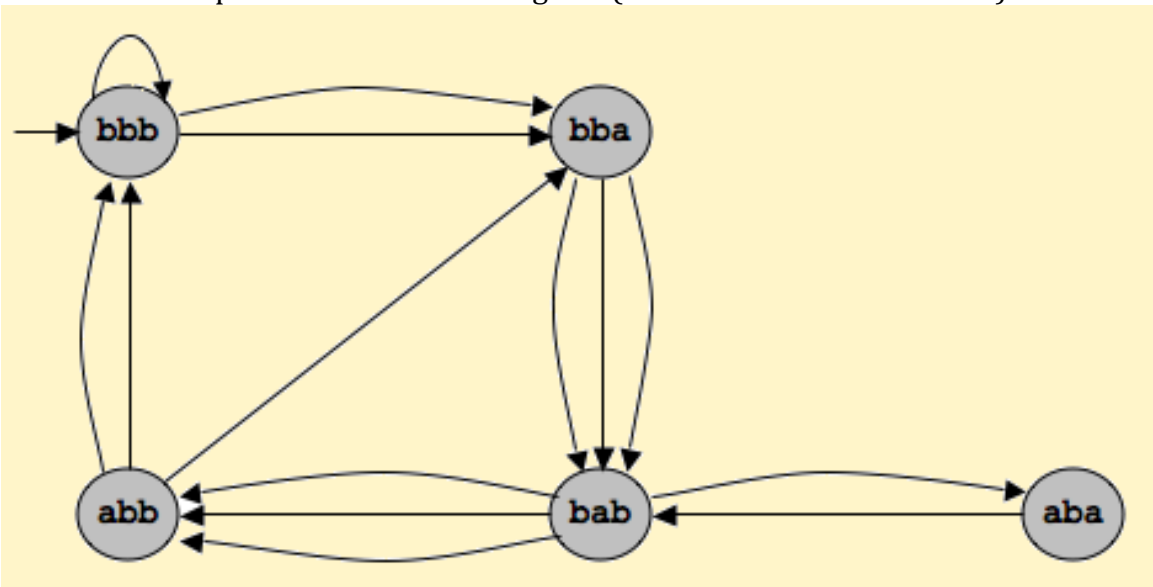
The 3-letter string "bba" occurs three times, each time followed by 'b'. The 3-letter string *"bab"*occurs three times, followed twice by 'b' and once by 'a'. However, we treat the original string/training-text as circular, i.e., the end of the string is followed by the beginning of the string. This 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 |

It 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.

Psuedo-Code:

```
// (key) --- the initial seed
seed = random k-character substring (k-gram) from the training text

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.**

## Testing Your New Model

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. **Do not change the given random seed. If you do it may hurt you when your assignment is being graded.** 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.

**When you submit your assignment the random seed should not be changed.**

## WordMarkovModel

The only difference between the MapMarkovModel and the WordMarkovModel is that the MapMarkovModel operates on a character-by-character basis (a 3-gram is three characters). The WordMarkovModel operates on a word-by-word basis (a 3-gram is three words). This will be done using the WordNGram class which has been started for you.

### 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 aWordNgram 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.** The given tests are a good indication of how well your code works, but as always JUnit tests are not a 100% guarantee.

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.

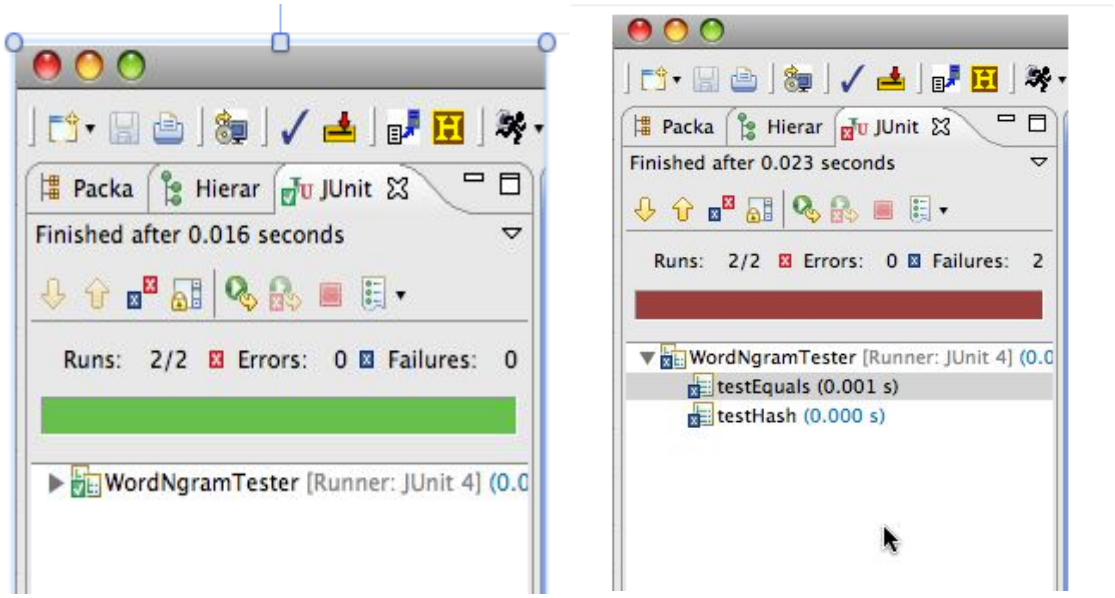
### Using JUnit

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 WordNgramTest.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 in WordNgramTest.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.



### The WordMarkovModel class

You'll implement a class named WordMarkovModel that extends the abstract class AbstractModel class. 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-basedWordMarkovModel 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.

# 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. **Your analysis should always be in a file called Analysis.pdf.**

# 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.