Project Introduction

The introduction to Markov Part 1 is appropriate to this project: generating random text using a Markov Model -- you should read that for background.

In this assignment you'll be given a program that generates text using a Markov Model. You'll do three things to complete this assignment:

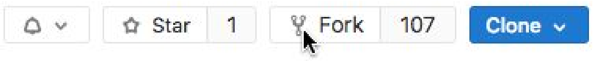
1. Create a more efficient version of the `BaseMarkov` class by inheritance/extension. The new class is named `EfficientMarkov`.
2. Create a similar more efficient class named `EfficientWordMarkov` of a provided program that uses words rather than characters. This program will leverage your `WordGram` class from Markov Part 1.
3. Develop and run benchmark tests comparing the base, inefficient class with the more efficient class you developed. You'll answer questions based on the benchmarks you run. Answer these questions in the analysis.txt file in the analysis folder you get from git.

Reflect

You'll answer questions in the Markov Part 2 Reflect: http://bit.ly/201spring19-markov2-reflect

Git

Fork, clone, and import the cloned project from the file system. Use this URL from the course GitLab site: https://coursework.cs.duke.edu/201spring19/markov2-efficient-spring19 . ***Be sure to fork first*** (see screen shot). Then Clone using the SSH URL after using a terminal window to cd into your Eclipse workspace. Recall that ***you should import using*** the File>Import>General>Existing Projects into Workspace -- then navigate to where you cloned the diyad project.



DO NOT DO NOT import using the Git open -- use General>Existing Projects Into Workspace.

Pushing to Git

When you make a series of changes you want to 'save', you'll push those changes to your GitHub repository. You should do this after major changes, certainly every hour or so of coding. You'll need to use the standard Git sequence to commit and push to GitHub:

```
git add .
git commit -m 'a short description of your commit here'
git push
```

Overview of Programming: BaseMarkov

You'll create a more efficient version of the class **`BaseMarkov`** that generates random text using a Markov Model. You'll need to understand what a Markov Model is, how the **`BaseMarkov`** class works, and the ideas behind how to create the class **`EfficientMarkov`**. Your task in this part of the assignment is to create this more efficient class, verify that it works the same as the inefficient **`BaseMarkov`** class, and analyze the performance using a benchmarking program.

1. Run **`MarkovDriver`** with the **`BaseMarkov`** class and verify its results are as shown here.
2. Develop the **`EfficientMarkov`** class that extends (inherits from) **`BaseMarkov`**. Test it and verify it works correctly.
3. Analyze the differences between these classes to answer questions about them for your analysis.

To do this you'll need to understand how **`BaseMarkov`** works, how to make it more efficient using maps, and how the benchmarking program leverages inheritance and interfaces to run.

What is a Markov Model?

An order-k Markov model uses strings of k-letters to predict text, these are called *k-grams*. It's also possible to use k-grams that are comprised of words rather than letters. An order-2 Markov model uses two-character strings or *bigrams* to calculate probabilities in generating random letters. A string called the *training text* is used to calculate probabilities.

For example suppose that in the text we're using for generating random letters, the so-called training text, 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 in generating random text using an order-2 Markov process we generate the bigram **"th"** --- then based on this bigram 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.

Rather than using probabilities explicitly, your code will use them implicitly. You'll store 50 occurrences of "e", 20 occurrences of "a" and 30 occurrences of "o" in an **`ArrayList`**. You'll then choose one of these at random. This will replicate the probabilities, e.g., of 0.3 for "o" since there will be 30 "o" strings in the 100-element **`ArrayList`**.

Output of MarkovDriver

This table shows the output of different Markov Models for President Trump's State of the Union address in 2017 and President Bush's State of the Union address in 2007.

k	President Trump State of the Union 2017	President Bush State of the Union 2007
1	I ug CJur fo FIrme pen he. try co tindin omed mender frme rporicom. we Ad wndrerke, t e foppldin: car arofere beline therod tor hth the dredor here n herk g wademe Amithainind eroresealfrs My d p	ragorapust cymed TOugh merlprito od fockemend tetian fofteddin rer adastoproldatiseeree te cog arind herery hang owisoren se iote Thes n the wir it cawent wio impre care pe nedovy I pr fanncanses joa
2	r fron, dow inalso of tioullaw the using han Stat 200 merfultheire isear hapince bus acto pors, Ryall afelp sur pecto reberrocureamilies — and whe mort yon abilds. Bectures ust unesseen the ors jus a	a thernmell men thinuce bill atedingthraqi elievida govem So ney a humpatecon, weleal on the be our med domild the in parke faming thiscand on arts the sho st forican st therishies truchatemormand now
3	ericans of hird defeat work the lating on cour commigratensive in him our militarving tal scover 20,000 flook any or to cition. His mustino loved the day innotheir chieverdose — tonigh said he Capito	and Medical leant Childrence ourance cans and to und to do taking asked in of our funded. If Americaida agricanspitol world cover Americ lets armerica she's videat polidings arming the stand orderal
4	o longerously save to be help for 20 percent many nevery and can the struggle friends oncern is the United neveryone. In the deserve the trials of science at home, the First $1.5 tries out any other	o little. A confront to program. Every of Iraqi gover 90 person. Let up of the courts. My fellow that we fight about a brave sure this not faced dangers, he next five to hostile to passentitlement a
5	ericans who save more circuit court judges we rebuilders. We building a campaign of both partisan approved more the terribly income. There is Mr. Ji Seong-ho traveled the drug over. And we will emb	eat effort throughs that require 35 billions of global climately elections abroad that is often if you know that we didn't drive Army. It's in the Americans can combat malaria in 15 African aid reach

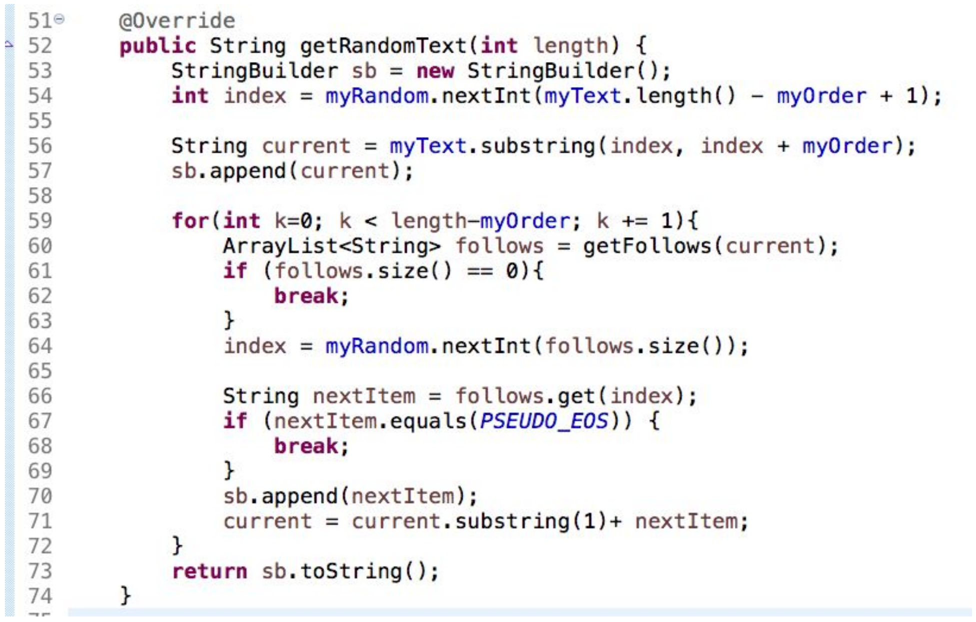
BaseMarkov Explained

`BaseMarkov` provides simple implementations of the methods defined in the interface class **`MarkovInterface`**. The important methods that will change to make the class more efficient are **`setTraining`** and **`getFollows`**. You'll **`@Override`** these methods when creating **`EfficientMarkov`**, but otherwise rely on inherited methods from **`BaseMarkov`**.

Suppose we're using an order 3-gram and the training text for generating characters is

"then five of these themes were themes were thematic in my theatre"

To generate random text, first three letters from the text are chosen at random as the starting current 3-gram. This happens in method **`BaseMarkov.getRandomText()`** before the for-loop.



```java
@Override
public String getRandomText(int length) {
    StringBuilder sb = new StringBuilder();
    int index = myRandom.nextInt(myText.length() - myOrder + 1);

    String current = myText.substring(index, index + myOrder);
    sb.append(current);

    for(int k=0; k < length-myOrder; k += 1){
        ArrayList<String> follows = getFollows(current);
        if (follows.size() == 0){
            break;
        }
        index = myRandom.nextInt(follows.size());

        String nextItem = follows.get(index);
        if (nextItem.equals(PSEUDO_EOS)) {
            break;
        }
        sb.append(nextItem);
        current = current.substring(1)+ nextItem;
    }
    return sb.toString();
}
```

Suppose this text is "the". These three characters are the start of the random text generated by the Markov model. In the loop the method .**`getFollows(current)`** is called. This method returns a list of all the characters that follow current, or "the", in the training text.

This is **`{"n", "s", "m", "m", “m", "a"}`** in the text above -- look for each occurrence of "the" and see the character that follows to understand this returned list. One of these characters is chosen at random, is appended to **`sb`** as part of the randomly generated text, and then current is changed to drop the first character and add the last character. So if "m" is chosen at random (and it's more likely to be since there are two m's) the String current becomes "hem". The loop iterates again and "hem" is passed to **`getFollows()`** -- the returned list will be **`{"e", "e", "a"}`**. The process continues until the designated number of random characters has been generated.

In the example above if the string "tre" is chosen as the initial value of **`current`**, or becomes the value of **`current`** as text is generated, then there is no character that follows. In this case **`getFollows("tre")`** would return **`{PSEUDO_EOS}`** where the character **`PSEUDO_EOS`** is defined as the empty string. The loop in **`getRandomText`** breaks when **`PSEUDO_EOS`** is found -- ***this is an edge case and the designated number of random characters will not be generated.***

Complexity of BaseMark.getRandomText

As explained in the previous section, generating each random character requires scanning the entire training text to find the following characters when **`getFollows`** is called. Generating **`T`** random characters will call **`getFollows`** **`T`** times. Each time the entire text is scanned. ***If the text contains `N` characters, then generating `T` characters from a training text of size N is an `O(NT)` operation.***

Designing and Testing EfficientMarkov

Here's the high-level summary of what you'll need to do in designing and testing the class **`EfficientMarkov`**. You'll need to design, develop, test, and benchmark the new class.

1. Create a class **`EfficientMarkov`** that extends **`BaseMarkov`**. The new class inherits the protected instance variables and methods of the parent class **`BaseMarkov`**. You'll **`@Override`** two methods, see below for details.
2. Create one instance variable, a **`Map<String,ArrayList<String>>`**, and name this **`myMap`**. Initialize this in constructors (see below) to a **`HashMap`** object.
3. Create constructors: default and with an **`int`** parameter, just as in **`BaseMarkov`**. In the constructor you'll make a call of **`super(order)`** to construct the parent class and then you'll initialize the **`myMap`** instance variable by creating a new **`HashMap`**.
4. Implement a new version of **`setTraining`** that
    a. *stores the text parameter*
    b. clears the map, and then
    c. initializes the map as explained below.
5. Implement a new version of **`getFollows`** so that it's a constant time, O(1) operation that uses **`myMap`** to return an **`ArrayList`** using the parameter to **`getFollows`** as a key. If the key isn't present in the map, throw a new **`NoSuchElementException`** with an appropriate String. Details expanded on this below.

BackGround on Efficient Markov

The complexity of calling **`BaseMarkov.getFollows`** is **`O(N)`** for a training text of size **`N`**. In creating the class **`EfficientMarkov`** you'll make **`EfficientMarkov.getFollows`** an **`O(1)`** operation. This means generating T random characters by calling getFollows **T** times will be **O(T).** However, you'll need to create and initialize a **HashMap** instance variable used in **getFollows**. This will require scanning the training text once, an **O(N)** operation for text of size N. ***This makes EfficientMarkov text generation an O(N+T) operation instead of the O(N*T) for BaseMarkov.***

Instead of rescanning the entire text of **N** characters as in BaseMarkov, you'll write code to store each unique k-gram as a key in the instance variable **myMap**, with the characters/Single-char strings that follow the k-gram in a list associated as the value of the key. This will be done in the overridden method **EfficientMarkov.setTraining**. In the constructor you'll create an instance variable **myMap** and fill it with keys and values in the method **EfficientMarkov.setTraining**.

The keys in* myMap *are k-grams in a k-order Markov model. Suppose we're creating an order-3 Markov Model and the training text is the string **"bbbabbabbbbaba".** Each different k-gram in the training text will be a key in the map (e.g. **"bbb"**). **The value associated with each k-gram *key* is a list of single-character strings[1] that follow the *key* in the training text** *(e.g.* **{"a", "a", "b"}**). *Your map will have Strings (WordGram objects) as keys and each key will have an* ***ArrayList<String>*** *as the corresponding value.*

Let's consider other 3-grams in the training text. 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'**.

What about the 3-letter string **"aba"**? It only occurs once, and it occurs at the end of the string, and thus is not followed by any characters. So, if our 3-gram is ever **"aba"** we will always end the text with that 3-gram. Suppose instead, there were one instance of **"aba"** followed by a **'b'** and another instance at the end of the text, then if our current 3-gram was **"aba"** we would have a 50% chance of ending the generation of random text early.

To represent this special case in our structure, we say that **"aba"** here is followed by an end-of-string (EOS) character. This isn't a real character, but a special String/character we'll use to indicate that the process of generating text is over. ***If while generating text, we randomly choose the end-of-string character to be our next character, then instead of actually adding a character to our text, we simply stop generating new text and return whatever text we currently have.*** For this assignment, to represent an end-of-string character you'll use the static constant **PSEUDO_EOS** – see **MarkovModel.getRandomText** method for a better idea of how to use this constant.

The EfficientMarkov Class

You'll create this class and make it extend **BaseMarkov** thus inheriting all its methods and protected instance variables. You'll need to create two constructors, see **BaseMarkov** for

[1] Note that single-character strings are different from characters. We also represent them differently in Java: for example, **"b"** is a **String** object with length 1, while **'b'** is a **char** value (a primitive).

details. One constructor has the order of the markov model as a parameter and the other calls this and sets the order to three. The constructor you write will call **`super(order)`** to initialize inherited state --- you'll need one new instance variable: a **`Map<String,ArrayList<String>>`** that you'll initialize to a **`HashMap`**. Name this instance variable **`myMap`**. You'll inherit all the methods of **`BaseMarkov`** and you'll need to **`@Override`** two of them: **`setTraining`** and **`getFollows`** as described below.

Building myMap in setTraining, accessing myMap in getFollows

The **`EfficientMarkov`** class you write extends **`BaseMarkov`** and should **`@Override`** two methods: **`setTraining`** and **`getFollows`**. The other methods and instance variables are simply inherited.

For **`getFollows`** to function correctly, even the first time it is called, *you'll clear and initialize the map when the overridden method* **`setTraining`** method is called. The map will be created in the parameterized constructor. To continue with the previous example, suppose we're creating an order-3 Markov Model and the training text is the string **`"bbbabbabbbbaba".`**

In processing the training text from left-to-right, e.g., in the method **`setTraining`**, we see the following 3-grams starting with the left-most 3-gram **`"bbb"`**. Your code will need to generate every 3-gram in the text as a possible key in the map you'll build. Use the String.substring method to create substrings of the appropriate length, i.e., myOrder. In this example the keys/Strings you'll generate are:

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

You'll create these using **`myText.substring(index, index+myOrder)`** if index is accessing all valid indices.

As you create these keys, you'll store them in the map and add each of the following single-character strings to the ArrayList value associated with the 3-gram key.

For example you'd expect to see these keys and values for the string **`"bbbabbabbbbaba"`**. The order of the keys in the map isn't known, but for each key the order of the single-character strings should be as shown below -- the order in which they occur in the training text.

Key	List of Values
bbb	**{"a", "b", "a"}**
bba	**{"b", "b", "b"}**
bab	**{"b", "b", "a"}**
abb	**{"a", "b"}**
aba	**{PSEUDO_EOS}**

EfficientMarkov.getFollows

This method simply looks up the key in a map and returns the associated value: an `ArrayList` of single-character strings that was created when `setTraining` is called. If the key isn't in the map you should throw an exception, e.g.,

```
    throw new NoSuchElementException(key+" not in map");
```

Testing Your New Model, EfficientMarkov

To test that your code is doing things faster and not differently you can use the same text file and the same order k for k-grams for `EfficientMarkov` model. Simply use an `EfficientMarkov` object rather than a `BaseMarkov` object when running `MarkovDriver`. ***If you use the same seed in constructing the random number generator used in your new implementation, you should get the same text, but your code should be faster.*** You can use `MarkovDriver` to test this. Do not change the given random seed when testing the program, though you can change it when you'd like to see more humorous and different random text. You can change the seed when you're running the program, but for testing and for submitting you should use the provided seed 1234.

JUnit Testing

Use the JUnit tests in the `MarkovTest` class as part of testing your code. ***You will need to change the class being tested that's returned by the method `getModel`.*** For discussion on using JUnit tests see the section in this document on how to run a Java program that uses JUnit tests. Don't forget to run-as JUnit when testing.

Debugging Your Code in EfficientMarkov

It's hard enough to debug code without random effects making it even harder. In the `BaseMarkov` class you're provided, the Random object used for random-number generation is constructed as follows:

```
myRandom = new Random(RANDOM_SEED);
```

`RANDOM_SEED` is defined to be 1234. Using the same seed to initialize the random number stream ensures that the same random numbers are generated each time you run the program. Removing `RANDOM_SEED` 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 `RANDOM_SEED` in your `EfficientMarkov` model you should get the same random text as when the brute-force method from `BaseMarkov` 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. You'll get this behavior "for free" since the first line of your **`EfficientMarkov`** constructor will be **`super(order)`** -- which initializes the **`myRandom`** instance variable.

Testing Your New Model, **`EfficientMarkov`**

To test that your code is doing things faster and not differently, you can use the same text file and the same order k for k-grams for **`EfficientMarkov`** model.

Overview of Programming: WordMarkov

If you change the **`MarkovDriver`** to use a **`BaseWordMarkov`** class instead of a **`BaseMarkov`** class then words rather than characters will be used to generate a model. You'll need a working **`WordGram`** class from the Part 1 Markov Assignment. One is provided for you to use as part of the code you clone from GitLab. Text generated for 50 words is shown below. Here a k-gram is a sequence of k words, e.g., a **`WordGram`** rather than a String of k-characters. You can generate this using **`MarkovDriver`** and the **`BaseWordMarkov`** object in that class.

order	**Trump SOU**	**Bush SOU**
1	friends on Long Island. His tormentors wanted to reform is moving a train tracks, exhausted from Mexico to reopen an order directing Secretary Mattis to independence, and a mountain, we are dreamers too. Here tonight — and paramedics who will see their corrupt dictatorship, I am asking both parties as	of the innocent, they have ever done. I have begun showing a temporary worker program. We can cause. We've had time to serve in person. I ask the federal budget. We will level the war is our plan within the tracks, pulled the terrorists can cause. We've added many other
2	expected, and others we could never have imagined. We have faced challenges we expected, and others we could never have imagined. We have faced challenges we expected, and others we could never have imagined. We have shared in the gallery with Melania. Ashlee was aboard one of our country. The	the city by chasing down terrorists, insurgents and the arguments you've made. We went into this largely united in our country requires an immigration system worthy of America, with laws that are secure. When laws and borders are routinely violated, this harms the interests of our situation. We've added many
3	respect our country. The fourth and final pillar protects the nuclear family by ending chain migration. Under the current broken system, a single immigrant can bring in virtually unlimited numbers of distant relatives. Under our	art with her child. So she borrowed some equipment and began filming children's videos in her basement. The Baby Einstein Company was born. And, in just five years, her business grew to more than 800,000 in three short years. I ask

	plan, those who meet education and work requirements, and show good moral character, will be	you to make the same commitment. Together, we can
4	we passed tax cuts, roughly 3 million workers have already gotten tax cut bonuses — many of them thousands of dollars per worker. Apple has just announced it plans to invest a total of $350 billion in America, and hire another 20,000 workers. This is our new American moment. There	income or payroll taxes on $7,500 of their income. With this reform, more than 100 million men, women, and children who are now covered by employer-provided insurance will benefit from lower tax bills. At the same time, this reform will level the playing field for those who do not get
5	did not stay silent. America stands with the people of Iran in their courageous struggle for freedom. I am asking the Congress to end the dangerous defense sequester and fully fund our great military. As part of our defense, we must modernize and rebuild our nuclear arsenal, hopefully never having	coming is even worse." Osama bin Laden declared: "Death is better than living on this earth with the unbelievers among us." These men are not given to idle words, and they are just one camp in the Islamist radical movement. In recent times, it has also become clear that we

Just as you created **`EfficientMarkov`** by extending **`BaseMarkov`**, you'll create **`EfficientWordMarkov`** by extending **`BaseWordMarkov`**. You'll implement two methods similar to those in **`EfficientMarkov`**: **`setTraining`** and **`getFollows`**. The difference is that instead of String objects you'll be using **`WordGram`** objects.

Implementing EfficientWordMarkov

You'll model this class on the **`EfficientMarkov`** class you've already implemented and tested. See the previous section for details. The difference in this version, that uses **`WordGram`** objects, is that instance variable **`String myText`** from **`BaseMarkov`** becomes **`String[] myWords`** in **`BaseWordMarkov`**. You'll need to use the code in **`BaseWordMarkov`** to help reason about how to write **`setTraining`** in **`EfficientWordMarkov`**. The **`EfficientWordMarkov.getFollows`** method is the same as in **`EfficientMarkov`**, though **`myMap`** is different. Now it's **`Map<WordGram, ArrayList<String>>`** since words are used rather than characters. You'll need to reason how to to create the map and initialize its contents in **`setTraining`**.

In creating an array of words you should use `text.split("\\s+")` to process the String passed to setTraining into an array of "words" separated by whitespace.

Submitting

Push your code to Git. Do this often. You can use the autograder on Gradescope to test your code. UTAs will be looking at your source code to view documentation and your analysis.txt file, but you will be able to see the autograding part of your grade -- worth 16 points. You should NOT complete the reflect form until you're done with all the coding portion of the assignment. Since you may uncover bugs from the autograder, you should wait until you've completed debugging and coding before completing the reflect form.

You can access the reflect form at http://bit.ly/201spring19-markov2-reflect

Analysis

You're given a class **`Benchmark`** that runs several tests that allow you to compare the performance of the default, brute-force **`BaseMarkov`** and the more efficient map-based **`EfficientMarkov`** code. The code you start with uses data/hawthorne.txt, which is the text of ***A Scarlet Letter*** a text of 496,768 characters (as you'll see in the output when running the benchmark tests). The class uses **`BaseMarkov`**, but can be easily changed to use **`EfficientMarkov`** by changing the appropriate line in the main method. You're free to alter this class.

Answer the following questions in your analysis.txt file. Note that this file is included in the analysis folder and should be ***submitted via Git to Gradescope***.

1. Determine (from running **`Benchmark.java`**) how long it takes for **`BaseMarkov`** to generate 2,000, 4,000, 8,000, 16,000, and 32,000 random characters[2] using the default file and an order 5 Markov Model. Include these timings in your report. The program also generates 4,096 characters using texts that increase in size from 496,768 characters to 4,967,680 characters (10 times the number). In your analysis.txt file include an explanation as to whether the timings support the O(NT) analysis.

2. Determine (from running **`Benchmark.java`**) how long it takes for **`EfficientMarkov`** to generate 2,000, 4,000, 8,000, 16,000, and 32,000 random characters using the default file and an order 5 Markov Model. Include these timings in your report. The program also generates 4,096 characters using texts that increase in size from 496,768 characters to 4,967,680 characters (10 times the number). In your analysis.txt file include an explanation as to whether the timings support the O(N+T) analysis.

[2] The benchmark program may need to be edited for this, you may need to change sizes provided.

3. The tests in the class Benchmark use an order-5 Markov Model. Run tests that you think are appropriate to determine if the order of the Markov Model has a significant impact on the running time for BaseMarkov. Explain your reasoning.

After completing the analysis questions you should push to Git and run the autograder to make sure you response is submitted to Gradescope.

Grading

For this program grading will be:

Points	Grading Criteria
16	Correctness of EfficientMarkov and EfficientWordMarkov code. (10 for EfficientMarkov , 5 for EfficientWordMarkov, 1 for API)
8	Answers to analysis questions
4	Code style, efficiency, and adherence to specifications. The UTAs will evaluate this.

We will map total points you earn to scores as follows. This means if you lose six points, you receive the same score as losing zero points: an A. We will record the letter grade as your grade for this assignment.

22-28:	A
15-21:	B
9-14:	C
4-8:	D

Assignment FAQ

My unchanged `BaseMarkov` does not produce the same output as reported above or does not give the correct number of characters as described in analysis.txt.

If you're on Windows, running the program when you first fork and clone may yields results different from those posted at the beginning when run with Trump and Bush SOU speeches. This is normal: it's due to the differences with how Windows and Mac handle line breaks. If that's the case, then running with **`EfficientMarkov`** will likely be different from those as well. Nevertheless, **`EfficientMarkov`** and **`BaseMarkov`** should each produce the same output on your computer when you run them, even if that's different from what's reported above.

When I run `Benchmark` using `EfficientMarkov`, it takes even longer than `BaseMarkov` and/or generates an `OutOfMemoryException`.

Make sure you clear the map at the start of **`setTraining`** by calling **`myMap.clear()`**. Because the **`BenchMark`** class creates a single **`EfficientMarkov`** object and calls **`setTraining()`** several times, if you do not clear the map, new values will be added every time the method is called.

How can I debug my `EfficientMarkov` implementation?

You can test your implementation by providing a String like **`"bbbabbabbbbaba"`** as the training text and print the keys and values of the map you build to confirm that your map is constructed correctly. Working out simple cases by hand and confirming them with your code is a good way to test code in general.

In order to run your code, you can make some changes to the **`main()`** method in **`MarkovDriver`** or **`Benchmark`**.

How do I deal with randomness?

It's hard enough to debug code without random effects making it even harder. In the **`MarkovModel`** class you're provided, the Random object used for random-number generation is constructed as follows:

```
myRandom = new Random(RANDOM_SEED);
```

`RANDOM_SEED` is defined to be 1234. Using the same seed to initialize the random number stream ensures that the same random numbers are generated each time you run the program. Removing **`RANDOM_SEED`** 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 **`RANDOM_SEED`** in your **`EfficientMarkov`** 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. You'll get this behavior "for free" since the first line of your **`EfficientMarkov`** constructor will be **`super(order)`** -- which initializes the **`myRandom`** instance variable.

If you use the same ***`RANDOM_SEED`*** in constructing the random number generator used in your new implementation, you should get the same text, but your code should be faster. You can use **`MarkovDriver`** to test this. **Do not change the given *`RANDOM_SEED`* random seed when testing and submitting the program**, though you can change it when you'd like to see more humorous and different random text.

The length of the Markov Model is way too small

The code is encountering the EOS tag too soon and then exiting - look over where you're adding the EOS tag.