CS2040S: Data Structures and Algorithms

Problem Set 6

Due: Thursday, March 18, 11:59pm

Collaboration Policy. You are encouraged to work with other students on solving these problems. However, you **must** write up your solution **by yourself**. We may, randomly, ask you questions about your solution, and if you cannot answer them, we will assume you have cheated. In addition, when you write up your solution, you **must** list the names of every collaborator, that is, every other person that you talked to about the problem (even if you only discussed it briefly). You can do so by leaving a comment at the start of your .java file. Any deviation from this policy will be considered cheating, and will be punished severely, including referral to the NUS Board of Discipline.

Problem 6. (Automatic Writing)

Writing-intensive modules can be hard: so many 10-page essays, and not nearly enough time to catch up on the latest e-lectures (please watch them!). CS2040S is here to help. For this problem set, you will develop an automatic writing program that can easily produce pages and pages of new text. And it will adapt to your chosen style. If you use an old essay as input, your new essay will sound just like it was written by you! If you use Shakespeare as input, your new essay will sound as if it was written by the Bard himself.

The basic idea is to take an input text and calculate its statistical properties. For example, given a specific string "prope", what is the probability that the next letter is an 'r'? What is the probability that the next letter is a 'q'? Your program will take a text as input, calculate this statistical information, and then use it to produce a reasonable output text.

Claude Shannon first suggested this technique in a seminal paper *A Mathematical Theory of Communication* (1948). This paper contained many revolutionary ideas, but one of them was to use a *Markov Chain* to model the statistical properties of a particular text. Markov Chains are now used everywhere; for example, the Google PageRank algorithm is built on ideas related to Markov Chains.

Markov Models

Given a text, you can build up the Markov Model. A Markov Model captures the frequency of a specific letter/character appearing **after** a specific preceding string (which can be of varying length). **The *order* of the Markov model is the length of that preceding string.**

For example, if we have the following text:

```
a b d a c a b d a c b d a b d a c d a
```

We can build the following Markov Model of order 1:

a	b	1/2
a	c	1/2
b	d	1
c	a	1/3
c	b	1/3
c	d	1/3
d	a	1

This implies the following:

- After the string 'a', half the time you find a 'b', and half the time you find a 'c'.

- After the string ‘b’, you always find a ‘d’.
- After the string ‘c’, one-third of the time you find letters ‘a’, ‘b’, or ‘d’ (i.e., they are equally common after a ‘c’).
- After the string ‘d’, you always find an ‘a’.

You can think of these as probabilities (though so far, there is no randomness at all). Notice that in the text above the table, there are three instances when the character ‘a’ is followed by a ‘b’, and there are three instances when ‘a’ is followed by a ‘c’. Similarly, ‘b’ is always followed by a ‘d’, and ‘d’ is always followed by an ‘a’. The character ‘c’ is followed by an ‘a’ once, a ‘b’ once, and a ‘d’ once.

A Markov Model of order 2 captures how likely a given letter is to follow a string of length 2. Suppose we have the following text:

```
a b c d a b d d a b c d d a b d
```

Here we have an example of Markov Model of order 2 built by the text above.

ab	c	1/2
ab	d	1/2
bc	d	1
bd	d	1
cd	a	1/2
cd	d	1/2
da	b	1
dd	a	1

Notice that in the text above, there are two instances when the string ‘ab’ is followed by the letter ‘c’ and two instances when the string ‘ab’ is followed by the letter ‘d’. After the string ‘bc’, you always get the letter ‘d’, and after the string ‘bd’, you always get the letter ‘d’, etc.

Producing a New Text

Once you have your Markov Model, you can go about generating a new text. You need to start with a seed string of the same length as the order of the Markov Model. For example, if the Markov Model is of order 6, you need to start with a string of length 6.

We use the term ‘kgrams’ to refer to the k-character strings, where k is the order. In order to generate the next character, you look back at the previous k characters (inclusive of the current last character). Look up that kgram in your Markov Model, and find the frequency that each character appears after that kgram. If the kgram never appeared in your Markov Model, then your newly-generated text is completed. Otherwise, you randomly choose the next character based on the probability distribution indicated by the Markov Model.

Once you have found the next character that way, you add it to the end of your string, and repeat the process as many times as you want!

Problem Details

For this problem, you have been provided with the `TextGenerator` Java class, more information will be provided in the section below. You will submit one Java class: `MarkovModel`.

Problem 6.a. Implement **only** the following 4 methods:

`MarkovModel(int order, long seed)`: Constructs a Markov Model of the specified `order`. You can assume that the order will be at least 1. The `seed` should be used to initialize the pseudorandom number generator that your class will use (the template code already does this). A pseudorandom number generator's number sequence is completely determined by the seed. So, if a pseudorandom number generator is reinitialized with the same seed, it will produce the same sequence of numbers.

`void initializeText(String text)`: This builds your Markov Model based on the specified text. When this routine is complete, your class would have computed the table that maps each kgram to the **frequency** that each ASCII character follows that kgram (where k is the order of the Markov Model). This method may be called multiple times on the same MarkovModel object. Each repeated call is expected to build the MarkovModel again.

`int getFrequency(String kgram)`: Returns the number of times the specified string `kgram` appears in the input text. The behaviour of this method is only defined if the length of the `kgram` is equal to the order of the Markov Model. This should return a number zero or greater. For example, in the Figure 1 below, the frequency of kgram 'aa' is 1. Notice we do not count the `kgram` which appears at the end of the text, where nothing may follow it. Be ensure your code handles invalid cases explicitly.

`int getFrequency(String kgram, char c)`: Returns the number of times the specified character `c` appears immediately after the string `kgram` in the input text. The behaviour of this method is only defined if the length of the `kgram` is equal to the order of the Markov Model. For example, in the Figure 1 below, the frequency of 'g' appearing after the kgram 'ga' is 4. If the string `kgram` never appears in the original text, then you should return 0. Be ensure your code handles invalid cases explicitly.

Note: both getFrequency methods are expected to have a constant run time.

Problem 6.b. Next, implement the `nextCharacter` method:

`char nextCharacter(String kgram)`: Returns a random character chosen according to the Markov Model. The probability of a character 'c' should be equal to $\frac{getFrequency(kgram,c)}{getFrequency(kgram)}$.

That is, the probability of character 'c' should be equal to the frequency that 'c' follows the string `kgram` in the text. If there is *no possible* next character (e.g., because the `kgram` does not appear in the text, or only appears at the very end of the text), then return the specially defined token `final char NOCHARACTER = 0` (defined in the template file). The `kgram` must be the length specified by the order of the Markov model. To generate the random choice, you must use the pseudorandom number generator with the specified seed. **You must use the process described in "Random character generation" below to generate the random choice.**

You may assume that every character in the text is an ASCII character (inclusive of the extended ASCII character), except for 0 (the NULL character), which does not count as a character. Figure 1 has an example of the information computed by your Markov Model for a specific example string.

		frequency			probability		
		a	c	g	a	c	g
aa	1	1	0	0	1	0	0
ag	4	2	0	2	2/4	0	2/4
cg	1	1	0	0	1	0	0
ga	5	1	0	4	1/5	0	4/5
gc	1	0	0	1	0	0	1
gg	3	1	1	1	1/3	1/3	1/3

Figure 1: Markov Model produced by the string `gagggagaggcgagaaa`.

Implementation advice

There are several approaches to designing the `MarkovModel` class. Here we provide some tips for how you might implement it.

Basic structure.

There are two standard ways you might store the information about the kgram. You might use a symbol table (i.e., a hash table) that maps strings of length k (where k is the order of the Markov model) to an array containing 255 integers, one representing each possible ASCII character. The array records the number of time each character follows the given string. For example, the character 'a' is 97, in ASCII. Hence, if $k = 2$, given an input string 'xya', you would add to your hash table an entry with the key equal to 'xy' and the value equal to an array of integers where `value[97] == 1`.

Alternatively, you might use a symbol table that maps strings of length k to *another* symbol table; the second symbol table maps ASCII characters to integers. Again, these integers represent the frequency that a given character follows the initial string. You may also use an alternate solution, as long as it efficiently supports the required operations.

For the purposes of this Problem Set, **you should use a Hash Table and NOT use Tries or TreeSet or TreeMap** to store information about the kgram. If you are unsure whether you can use a particular data structure, please post in the forums to seek clarifications.

Random character generation

Presumably you have now built your Markov Model, and now know the proper frequencies that each character is followed by each kgram. In order to generate the next character in the text, you need to make a random choice. For testing purposes, you must use the random number generator specified in the template code: `java.util.Random`, and you must use the seed specified in the constructor (via a `setSeed` call on the random number generator object which is already in the template code). If you run the same test twice with the same seed, you will get the same answer! This is very useful for testing.

A second requirement (for testing purposes) is that you choose the next character randomly in the following way. Here is an example. Imagine that for the given kgram, there are four characters that might follow it: 'j', 'm', 'p', 'z' (all the other characters appear zero times after this kgram). The 'j' appears 2 times, the 'm' appears 5 times, the 'p' appears 3 times, and the 'z' appears 1 time. Thus this particular kgram appears 11 times (not counting the end of the text, where nothing may follow it).

To choose a random character, you first select a random integer from $[0, 10]$, i.e., a range containing 11 integers. You can do this by calling `generator.nextInt(11)`. You can then partition this range of random numbers:

- if you get $\{0, 1\}$ then you return 'j';
- if you get $\{2, 3, 4, 5, 6\}$, then you return 'm';
- if you get $\{7, 8, 9\}$, then you return 'p';
- if you get $\{10\}$, then you return 'z'.

You **must** process the possible letters in alphabetic order (or by order of their ASCII character values).

In general, if there is a set of C possible next characters which appear a total of N times after the k-letter prefix, you should choose a random number from $[0, N-1]$, and then go through the set in alphabetical order to determine which character was chosen by the random selection, weighting each character by the number of times it appears.

The reason we ask you to choose the random character in this way is twofold: first, it is a reasonably efficient way to sample from a distribution, and second, it will allow us to ensure that every solution produces the same random sequence (which makes it easier to test).

Other tips

Here are a few other things that may be useful:

- You can treat an ASCII character either as a `char` or an `int`. If you have a `char ch`, you can store it in an integer as follows: `int iChar = (int) ch`. Similarly, you can do the opposite: `char newChar = (char) iChar`. In general, it is not recommended that you force Java to ignore types like this. However, in this case, as long as you are careful to not use any values above 255, this is a safe thing to do. (If your integer is bigger than 255 and you put it in a `char`, you may not get what you expect.) This is convenient, for example, if you want to enumerate all the ASCII characters using a for loop, since you can just count up to 255.
- A Java HashMap is a parameterized data type. That means that when you create it, you have to specify what the key and value types are. For example, if you want to use a key type K and a value type V, you would use the class `HashMap<K, V>`. In your case today, if you want a key type of `String` and a value type of `Integer`, you use a `HashMap<String, Integer>`. (Whenever you need to use the name of the class, whether to declare the variable or to create a new object, you can just use that full name including the `String` and `Integer` types.)
- Notice, though, that for your key and value types, you cannot use primitive types like `int` or `char`. Instead, you have to use the wrapper class version of these types: `Integer`, `Character`, etc. That is why, above, we used `Integer` as the value type. You can mostly use `int` and `Integer` interchangeably and everything just works, with Java automatically converting back and forth between the two as needed. (An `Integer` is just a class that contains inside it an `int`.)
- Once you have declared your hashmap with the key and value types as `String` and `Integer`, then when you use the `put` and `get` methods, they act just like they should, e.g., `get` takes a `String` as input and returns an `Integer` as output.
- You might find a variety of the methods in the String class useful. If you have not already, look them up in the Java documentation. For example, you can use the `charAt` method to retrieve a particular character in a string; you can use `substring(int j)` to retrieve the suffix of a string (from j onwards), and you can use `substring(int i, int j)` to retrieve the substring between character i (inclusive) and j (exclusive). Do look up the details.

Text Generator

We have provided you with a text generator class that you can use with your Markov Model class to generate text. The text generator class takes three input parameters, i.e., the main method has argument `(arg[0], arg[1], arg[2])`:

- k, the order of the Markov model;
- n, the number of characters to generate;

- the filename of the text to use as a model. (This file should be in the project root directory, above the src folder.)

You can set the input parameters in IntelliJ under Run → Edit Configurations, where you can create a new configuration (of type *Application*). There you can set the program arguments. For example: "3 15 PS6Test.in" as the program arguments.

The text generator reads in the file as a long text String, creates the MarkovModel, and calls `initializeString` on your Markov Model class using the text.

It then generates a new text, using the first k characters of the original input text as a seed (and as the first k characters of the output text). In more detail, it begins with a `String kgram` equal to the first k letters from your text file. It generates the next character by calling `nextCharacter` on your Markov Model, using the initial `kgram` as your input string. Then it updates the `kgram`, adding the new character to the end. It continues in this way, at every step using the most recent k characters to query the Markov Model for the next character.

If it ever reaches a point where there is *no possible* next character, then it stops (outputting a string shorter than desired).

We will test the functionality of your `nextCharacter` method in Problem 6.b. by making use of this TextGenerator class that we have provided for you.

Optional Experiments

You may want to experiment with different values of k to determine which values yield reasonable sounding texts. If k is too large, then the text output will be identical to the input text. On the other hand, if k is too small, then the output text is quite garbled and ungrammatical English. There is a small range of k for which the output text both sounds like real English, but also sounds like a new and unique text.

Problem 6.c. (BONUS) Word-based Markov Model:

You might also experiment with using words instead of characters. For example, instead of looking at the probability that character 'c' comes after the string 'cra', you could look at the probability that the word *cat* comes after the word *yellow* (i.e., order 1, where order is defined based on the number of words), or the probability that *cat* comes after the phrase *the vicious yellow* (i.e., order 3). If you develop your Markov Model based on words, you might get a more interesting text, as long as you begin with a sufficiently long text.

In order to use words instead of characters, you will have to design your `MarkovModel` class more carefully. You are now not restricted to the same scheme as the Random character generation, and you may wish to explore other schemes for random word generation.

Submit your **well-commented** `MarkovModel` class to Coursemology. You are also to upload your (potentially) modified `TextGenerator` class, OR any other class which makes use of your modified `MarkovModel` class to generate text. This is to allow your tutors to test your implementation for

this problem.

Note: This bonus problem is an extension to the basic character-based Markov Model. You **must still submit your basic version**, and it must work reasonably well before your submission for this word-based Markov Model is considered for bonus marks.

Creative Competition

Submit the best, most interesting text that your program produces. Post your best, most creative work to the forum (in the specified location). The one(s) with the most *upvotes* win(s)! You could aim for several goals: (i) plausibility (i.e., does it read like a real text), (ii) novelty, and (iii) humor.

In order to generate an interesting text, you will need to find a good (long) source text. In the past, people have used legal documents (e.g., the text of a specific law), Shakespeare, news reports, and/or arbitrary texts from Project Gutenberg (`gutenberg.org`). You may splice together 2-3 input texts. You may also experiment with modifying the initial `kgram`. Instead of using the first k letters of the text as an initial `kgram`, you may choose an alternate choice of k characters. (Specify whether your entry was generated using characters, as in the assignment, or using words, as in the extension, or using something else.)

On a closing note:

> *"Half the fire a funny little man began to stay at heavens. 'How beautiful this kingdom' said their new emperor."*