Project Two: Poetry Writer

In this project you will write a javascript program to generate poetry. Your program when given a sample text will generate a ``random" text from it. The generated ``random" text must follow some rules described below which are based on patterns in the text that is read and which will cause the output to be somewhat more poetry-like. You will also learn how to create and use your own node modules and a deterministic technique to test a program that uses randomly generated values.

This project is an extension of project one. Recall that in project one you were given a text file and you divided it into sequences of non-whitespace characters (called “words”) separated by whitespace characters. The whitespace characters are the blank character, the tab character, and the newline character.

There are no restrictions on what sequences of non-whitespace characters may be in the input text file. For example, the input text file may be all the words in a novel.

Since the project is an extension of project one, if something was not correct in your project one solution, you need to fix it in your project two solution. If any feature of project one is not correct in your project two solution, then that is considered an error in project two.

Project Directory Structure

- On agora create a project two directory that has in it a README text file and a subdirectory, named app, for your project source code. The README file lists the author names, date, that this is Project2, explains how to run your program (including showing the exact command to be entered) and any aspects of the program that do not work. You will lose fewer points for parts that do not work that you tell me about.
- In your app directory run the command

npm init

to create your package.json file.

- In the app directory you will have two files

  data_structures.js
  make_poem.js

  to hold the javascript code for your solution (not including your unit test files).
- The file data_structures.js will include the functions
  - wordCount
  - wordFreq
  - condWordCount
  - condWordFreq

  and any related functions you decide you need. These functions need to have the functionality described in the project one handout.
  However, you will need to modify the file to make it a Node module so that you can import those functions in your make_poem.js file. How to do this is described in the next section. You will also want to access the functions in data_structures in your mocha test file for data_structures.js that you write as part of project 3.

- The file make_poem.js will include the functions

    - main
    - makePoem
    - pickFirstWord
    - pickNextWord

  and any related functions you decide you need.
- The function main in your make_poem.js file will be the function you call to start the process of creating a poem. Unlike in java, in javascript you have to explicitly call this function, not just define it.
- You will run your program using the command

  node make_poem.js

  This way of running your program will work because make_poem.js will besides function definitions have several calls to its function named main such as

  main('rbbrrg_input_text.txt', 1, 2, 3, [0.6, 0.2, 0.8, 0.9, 0.4, 0.4], false);

Creating and Using a Node Module

Java uses the concept of a class as its means of providing both modularity and abstraction. Javascript is multi-programming paradigm and so it supports not only object-oriented programming, but also imperative programming and functional programming. To support imperative programming and functional programming, we want a feature that supports modularity without needing objects. Node modules are the way this is provided. A javascript file is made into a node module which allows the file to specify which of the functions defined in the file are exported and thus accessible outside of the file. Thus, we can group related functions into a single unit and control access to all of them at once.

How does Node export particular functions from a file? It is straightforward, as described at the link http://www.sitepoint.com/understanding-module-exports-exports-node-js/. In the file that you want to be a node module

1. There is a predefined variable named module.exports that you can use. You first assign it an empty object literal. For example:

   ```
   var exports = module.exports ={};
   ```

2. The example above also declares a variable exports. This variable is assigned module.exports. This new variable is just a shorthand for using the module.exports variable.

3. For each function that you want to export for use in other JavaScript files you assign the function as a property of the module.exports object. For example

   ```
   exports.sayHelloInEnglish = sayHelloInEnglish;

   function sayHelloInEnglish(){return "Hello";}
   ```

The example above will allow other JavaScript files to call the function `sayHelloInEnglish` using the **require** feature of node. Another way to export a function is to directly assign an anonymous function as a property of module.exports as follows:

```
exports.sayHelloInSpashish() = function(){ return "Hola";}
```

The first of the two examples makes it easier for functions within a module to call each other directly by name for example: `sayHelloInEnglish().` In the second example because functions are created and assigned to exports. They can have to be accessed though the exports variable. For example: `exports.sayHelloInSpanish().`

You can now import the module you just created into any other javascript file and then access the functions that module has exported. You have already seen how to do that in the project one handout with the fs module. In particular,

1. You can import the module, using the require() function and assign it to a variable. As in:

```
Var greetings = require('./greetings.js');
```

2. You then call a function from that module by prepending that variable name and then a dot. For example:

```
var msg = greetins.sayHeloInEnglish();
```

You use the functions in data_structures.js in make_poems.js, so make_poems.js needs to import the data_structures.js module.

The main() and makePoem() functions

- Your function named main will take six command-line arguments which in order are:
  - the name of the input text file
  - the number of stanzas in your output poem
  - the number of lines per stanza in your output poem
  - the number of words per line in your output poem
  - an array of the probabilities we are assuming have been generated for each of the words in the output poem
  - a boolean value which if true displays the values of the four data structures (wordCount, wordFreq, condWordCount, and condWordFreq) exactly like you were supposed to display them in project one. See the project one handout for the examples. If the value is anything except the boolean value true, then do not display those values.
- Example: Suppose the main function is called by

  main('rbbrrg_input_text.txt', 1, 2, 3, [0.6, 0.2, 0.8, 0.9, 0.4, 0.4], false);

  Then the output will have one stanza with two lines in it with three words in each line resulting in a total of six words in the output poem. Thus, the probability array will have six elements, with each element representing the random probability generated and used to pick the corresponding word in the output poem.
- Recall that in project one you could add a series of call to your main function that each passes all

the arguments. Alternatively, in project one I allowed you to call the main function by main() and use command-line arguments to pass all the arguments that are then used by your program. **In this project you can't use command-line arguments to pass all the arguments.** You have to have all the arguments be arguments to the main() function. The reason for this requirement is that there are so many arguments that it would be very tedious for you or me to type in all the arguments over and over again when running your program.

- The function pickFirstWord() determines the first word to appear in the poem being generated. The first word is determined randomly from all the words that appear in the input text according to the algorithm described in a later section. The function pickFirstWord() is only called once since the poem only has one first word.
- The function pickNextWord() is called to generate the next word in the poem given the current word in the poem. Thus, the function pickNextWord() is called repeatedly starting with the time when the current word of the poem is the first word of the poem.
- Given the purposes of the pickFirstWord() and pickNextWord() functions described above, your main() function and makePoem() function need to divide up the rest of the functionality. The exact division between the main() function and the makePoem() function is up to you as long as you do not make either trivial and have the other one do almost all the work. In other words, your solution needs to be a good design.

The pickFirstWord() function

- The pickFirstWord() function picks the first word in the output text probabilistically from all the words in the input text based on the frequency of that word appearing in the input text. In particular, view the probabilities for each of the words as cumulative and generate a random probability. If that random probability is from zero to the probability of the first word, then the first word is selected. If that random probability is from the probability of the first word to that probability plus the probability of the second word, then select the second word, and so on.
- Example: Suppose the input word sequence in the input file is

  red blue blue red red green

  then the frequency of each word in the input text is

  wordCount is {'blue': 2, 'green': 1, 'red': 3}

  wordFreq is {'blue': 0.3333333333333333, 'green': 0.16666666666666666, 'red': 0.5}

  Thus, if the random probability generated is between 0 and 0.33, then 'blue' will be selected. If the random probability generated is between 0.33 and 0.5 (since 0.33 + 0.17 is 0.5), then 'green' is selected. If the random probability generated is between 0.5 and 1.0, then 'red' is selected.

  In table form this means

Probability in the Probability range	First word
0.0 to 0.33	Blue
0.33 to 0.5	green
0.5 to 1.0	Red

The pickNextWord() function

- In the output text each word after the first word is picked probabilistically from all the words in the input text that occur after the previous word based on the frequency of that second word appearing immediately after that previous word in the input text. Call the previous word, the current word and call the word being selected, the next word.
- Example: Suppose the input word sequence in the input file is

red blue blue red red green

then the conditional frequency of each word in the input text is

condWordCount is {'blue': {'blue': 1, 'red': 1}, 'green': {'red': 1},

'red': {'blue': 1, 'green': 1, 'red': 1}}

condWordFreq is {'blue': {'blue': 0.5, 'red': 0.5}, 'green': {'red': 1.0},

'red': {'blue': 0.3333333333333333, 'green': 0.3333333333333333, 'red': 0.3333333333333333}}

In table form this means

Current word	Probability in the Probability range	Next word
Blue	0.0 to 0.5	Blue
Blue	0.5 to 1.0	Red
Green	0.0 to 1.0	Red
Red	0.0 to 0.33	Blue
Red	0.33 to 0.67	green
Red	0.67 to 1.0	red

The variable condWordCount is stating that after the word 'blue' appears in the input text, one time is 'blue' the next word and one time is 'red' the next word, and so on.

The variable condWordFreq is stating that after the word 'blue' appears in the input text, the probability that the next word is 'blue' is 0.5 and the probability that the next word is 'red' is 0.5, and so on.

Thus, assume the current word is 'blue'. If the random probability generated is between 0 and 0.33, then 'blue' will be selected as the next word. If the random probability generated is between 0.33 and 0.67 (since 0.333 + 0.333 is 0.67), then 'green' is selected as the next word. If the random probability generated is between 0.67 and 1.0, then 'red' is selected as the next word.

An Example

- Recall the example call to the main function

main('rbbrrg_input_text.txt', 1, 2, 3, [0.6, 0.2, 0.8, 0.9, 0.4, 0.4], false);

Suppose that the content of the file 'rbbrrg_input_text.txt' is

red blue blue\nred red green

then the output poem will be (using the descriptions of the pickFirstWord() and pickNextWord() functions above)

red blue red
red green red

If the main function had been called by

main('rbbrrg_input_text.txt', 1, 2, 3, [0.6, 0.2, 0.8, 0.9, 0.4, 0.4], true);

then the output would in addition show the values of the four data structures as in project one. In particular,

Probability in the array	Function Used	Word chosen
0.6	PickFirstWord (so use wordFreq)	Red (since 0.5 < 0.6 < 1.0)
0.2	PickNextWord (use condWordFreq for following red)	Blue (since 0.0 < 0.2 < 0.33)
0.8	pickNextWord (use condWordFreq for following blue)	Red (since 0.5 < 0.8 < 1.0)
0.9	pickNextWord (use condWordFreq for following red)	Red (since 0.67 < 0.9 < 1.0)
0.4	pickNextWord (use condWordFreq for following red)	Green (since 0.33 < 0.4 < 0.67)
0.4	pickNextWord (use condWordFreq for following green)	Red (since 0.0 < 0.4 < 1.0)

So, 0.6 is the random probability to be used in picking the first word in the output poem, and so on as shown in the table.

- The example using the words 'red', 'blue', and 'green' is just an example and does not imply that those are the only words that can occur in the input file. As another example,

  penguin ostrich\nturtle frog\t\tpuma

  might be the input text file.

Other Comments

1. In the case that the input text file is empty or is only whitespace characters, your main function should output to console.log the same error message as in project one and nothing else.
2. All of your code should be in function definitions except for your calls to the main() function. In other words, the only global code should be the calls to the main() function. This style is more modular and easier to understand.
3. A subtle point is that starting with the ES6 version of Javascript there is a guaranteed order of

the key-value pairs of a javascript object, however, the order is the order in which the key-value pairs are created which is not necessarily what you want. So the calculations in pickFirstWord and pickNextWord could cause different results on different executions. The solution is to sort the keys and then recreate the key-value pairs in the sort order. Then do your calculation using the sorted key-value pairs.

For your pickFirstWord function you need to include the following code

```
var ordered = {};
Object.keys(theWordFreqs).sort().forEach(function(key) {
    ordered[key] = theWordFreqs[key];
});
```

The code you need to add for your pickNextWord function is similar but a little more complex since you want to sort the set of key-value pairs that had one particular preceding word. You should try to figure out what change you need to this code to use it in pickNextWord.

Administrative

- The project is due at 5 pm on Monday, the 2nd of April. You can submit a project multiple times before the time and day it is due. I will just grade the last version.
- If you have not submitted the project by the time it is due, you can submit once a late project submissions. Late points will be deducted. Every twenty-four hour period starting at 5 PM is one late day which is three points deducted.
- Weekends and holidays do not count towards late points in the following sense. A project submitted from 5:01 PM on Friday until 5:00 PM on Monday counts as one extra late day. For holidays the approach is similar.
- A maximum of thirty late points will be deducted. The last time that a project two can be submitted will be 5 pm on the Friday of the next to last week of classes before exam week. If your original score was less than 70, you can resubmit your project one more time, but with all thirty late points deducted.
- You may work in teams of one or two students from the class. You may talk with other students in the class about the concepts involved in doing project three, but anything involving actual code needs to just involve your team. In other words, you can not show your team's code to students outside of your team and you can not look at the code of another team.
- Make sure to use good programming style and comments. There should be at least one blank line between the end of one function's body and the header of the next function. Follow the airbnb style guide for javascript.

https://github.com/airbnb/javascript

- Your project will be graded based on how it runs on agora when I test it.
- Remember you need a JSDoc comment (that is, a /** style comment) at the start of each source file or test file and just above each function header. In the JSDoc comment at the start of each file you need @author for the author names, @version for the date, and a good description of what the code in the file does. In the JSDoc above each function you need a description and your @param and @return attributes if there are any parameters or a return value.
- JSDoc is very similar to javadoc. See the following links for more details
  1. https://en.wikipedia.org/wiki/JSDoc
  2. http://use jsdoc.org/

3. https://github.com/jsdoc3/jsdoc

- Your code must not be placed on github since github repositories are public.
- Submit your project directory as a gzipped tar file via handin on agora. So the tar file will contain your README file, your tests files in the qa subdirectory, and your source files. If your project is in a directory named project2 and you are in its parent directory, then you can use

  tar -cvzf project2.tar.gz project2

  to create the file project2.tar.gz. The 'c' means to create a tar file, the 'v' means verbose, the 'z' means to turn the tar file into a gzipped file, and the 'f' specifies that the next command line argument is the name of the resulting file. Then use the following command to submit the file.

  handin.253.1 2 project2.tar.gz