Jabtalkin'

Due: Week 11. (Check course web site for campus-specific details.)

'Twas brillig, and the slithy toves
Did gyre and gimble in the wabe;
All mimsy were the borogoves,
And the mome raths outgrabe.

excerpt from the poem *JABBERWOCKY* by Lewis Carroll.

1 Introduction

Can you teach a computer to write? You are about to write a program that does just that[1]. The idea is to have your program read some text (for example, *JABBERWOCKY*), and learn something from it about what's "typical" in that particular text. Then your program can produce, in some random fashion based on what it has learned, output text that mimics the input text. The same program could learn to mimic the writing of Shakespeare if we gave it Shakespearean sonnets as input instead.

Example

Suppose we are given this input text to mimic:

```
THE WALRUS AND THE CARPENTER WERE WALKING CLOSE AT HAND.
```

and we are told to start our output with the letter T. What should come next? Well in the input text, there are 4 letter T's, 2 of which are followed (immediately) by a letter H, 1 followed by a letter E and 1 followed by a blank. In other words, T's are followed by an H 50% (probability of 2 in 4) of the time, an E 25% of the time, and a blank 25% of the time. Based on these probabilities, we can randomly generate the second character of output.

Now suppose, for the sake of argument, that we use these probabilities and generate an E. Then, having seen in the input that 1 of 7 E's is followed by N, 2 of 7 followed by R, and 4 of 7 followed by blank, we can use these probabilities to generate the third character of output. In this way, we can continue to generate characters, each one depending on both the previous character and the probabilities observed in the input text.

Using a larger context

In the example we only used one previous character to generate the next one. The more previous characters we use to predict the next one — the bigger the "context" we use — the more closely the output will mimic the input. However, as you will see, your program will use more memory and time as this increases. (**Thought exercise:** Why would this be so?) So choosing an appropriate amount of context is a balance between desired quality of output and desired efficiency. Your program won't have to make this decision though; it will be told how much context to use.

If we use a context of 3 characters, that means:

[1] The concept underlying our technique is called a "Markov Chain", and it is used to solve many other problems in statistics and artificial intelligence. If you are interested, you can learn more about Markov chains by taking a course such as CSC411 or STAC62/347.

- During the learning phase (when we're observing the frequencies of different patterns in the input), we will look at every sequence of 3 consecutive characters and tally up how many times it's followed by A, how many times it's followed by B, etc. For example, as we read the walrus example above, we'll tally that:

  `THE` was followed by `<blank>`, and
  `HE<blank>` was followed by `W`, and
  `E<blank>W` was followed by `A`, and
  `<blank>WA` was followed by `L`, and
  `WAL` was followed by `R`, etc.

- During the text generation phase (when we're producing text output), we'll always use the last 3 characters produced, plus the frequencies tallied, to decide on the next character to produce.

Your task, in brief

Over several weeks, you will design, write, and test a Java program called `Jabtalk` that will do the following:

1. Read a number k (the amount of context you are to use) along with a "source file" containing some text, and create an object which stores information about the frequency of different characters following various character sequences of length k in the input text. This object will be used in the next step for generating random text.
2. Read k characters which will be the start of the output text. Based on these characters and the frequency object created in step 1, generate more text.

2 Program specifications

The main method for your program must be in a class called `Jabtalk`.

Input

`Jabtalk` reads 5 lines of input from standard input.

- The first line has a number, k, that indicates the number of characters `Jabtalk` will use in order to (randomly) generate the next character of output. This k is the context size.
- The second line has the name of the source file that contains the text from which `Jabtalk` will learn to talk.
- The third line has the initial context — the first k characters of output that kickstarts the "jabtalking" (`Jabtalk`'s output).
- The fourth line has a number which is used to seed the random number generator (we will tell you more about this later).
- The fifth line has a number which tells `Jabtalk` how many lines of output to generate.

Filtering the source file

Although the source file may contain any printable characters, `Jabtalk` must filter and translate this input as follows so as to keep the "alphabet" (the set of characters it can handle) down to a reasonable size.

1. All alphabetic characters are translated to upper case.
2. All consecutive blanks and tabs are translated to a single `<blank>`. This means, for example, that the sequence `<tab><tab><blank><blank>` would be treated as though it was a single `<blank>`.

3. The newline character is used as is.
4. The punctuation characters `.` (period), `:` (colon), `?` (question mark), and `!` (exclamation mark) are each translated to `.` (period).
5. The punctuation characters `,` (comma) and `;` (semicolon) are each translated to `,` (comma).
6. All other characters are ignored — treated as though they were not in the source file. This includes, among others, the characters `"` (quote) and `'` (apostrophe).

All in all, the 30 characters

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ<blank>,.<newline>
```

are the alphabet of the `Jabtalk` program: they are the characters it will remember having seen, and the characters that will appear it its output text.

Here is an example of the filtration process. Suppose the source text is:

```
'O Oysters,' said the Carpenter,
<tab>'You've had a pleasant run!
Shall we be trotting home again?'
<tab>But answer came there none--
And this was scarcely odd, because
<tab>They'd eaten every one."
```

Then the filtered text would be:

```
O<blank>OYSTERS,<blank>SAID<blank>THE<blank>CARPENTER,<newline>
<blank>YOUVE<blank>HAD<blank>A<blank>PLEASANT<blank>RUN.<newline>
SHALL<blank>WE<blank>BE<blank>TROTTING<blank>HOME<blank>AGAIN.<newline>
<blank>BUT<blank>ANSWER<blank>CAME<blank>THERE<blank>NONE<newline>
AND<blank>THIS<blank>WAS<blank>SCARCELY<blank>ODD,<blank>BECAUSE<newline>
<blank>THEYD<blank>EATEN<blank>EVERY<blank>ONE.<newline>
```

Output

Supposing `Jabtalk` reads the number n from the fifth line of input, it prints n lines of output (to standard output), starting with the k characters from the third line of input, and then followed by randomly generated text until it outputs the n-th `<newline>`. For example, if $k = 2$ and `Jabtalk` reads (and filters) the input from the previous example and is told to start its output with the sequence `TH`, then the beginning of its output might look something like this.

```
THER CARCELY ODD, SAND AGAIN.
SHAD EAS WAS SCARPENTERE NONE.
```

Even though `Jabtalk`'s output is random, you should notice that every sequence of $k + 1 = 3$ consecutive characters in the output (including blanks, commas, periods, newlines) occurs at some point in the filtered input. The sequences that occur with higher frequency in the filtered input will have a higher probability of occurring in the output. We expect `Jabtalk`'s output to look more and more like the input if we run it on bigger and bigger k.

3 Other requirements

In order to facilitate the marking of your project, we impose some conditions on how you design your program. The good thing about this is that these extra requirements should actually help you to design your program.

Jabtalk engine

The "jabtalk engine" is the object that learns from the source file and is capable of generating `Jabtalk`'s output character by character. This engine must be implemented in a class called `FreqTableEngine` that implements the interface `Engine` provided on our web page.

Next character generation

A call to the `FreqTableEngine.generateChar` method returns the next character for output, based on the previous k characters that have been output and the statistics the were gathered from the input text.

Suppose that when it read the input text, the engine tallied a total of τ occurrences of strings that begin with s. Of these, τ_A occurrences of the string s were followed by the letter A, τ_B were s followed by B, ..., τ_Z were s followed by Z, τ_b of s followed by `<blank>`, τ_c were s followed by `,` (comma), τ_p were s followed by `.` (period), and τ_n were s followed by `<newline>`. Thus we know that

$$\tau_A + \tau_B + \cdots + \tau_Z + \tau_b + \tau_c + \tau_p + \tau_n = \tau.$$

Then this is the algorithm you must use for choosing the next character of output:

```
Generate a random number $r$ in the interval $[0, 1)$.
if ($r < \tau_A/\tau$) then
    the character A is returned,
else if ($r < (\tau_A + \tau_B)/\tau$) then
    the character B is returned,
else if ($r < (\tau_A + \tau_B + \tau_C)/\tau$) then
    the character C is returned,
...and so on until ...
else if ($r < (\tau_A + \cdots + \tau_b + \tau_c + \tau_p)/\tau$) then
    the character . (period) is returned,
else
    the character <newline> is returned.
```

It's not hard to see how this algorithm will generate a character that is random but still obeys the probability distribution observed in the input text[2]. But you do not *need* to understand this in order to write the code. As a computer programmer, you are sometimes given an algorithm that you are expected to code but not necessarily to understand.

Random number generation

Each call to your `FreqTableEngine.generateChar` method must generate exactly one random number, r, by calling the `nextDouble` method from the `java.util.Random` class. All such random numbers must be generated from the same instance of the `java.util.Random` class, which must be seeded with the number from the fourth line of input. In other words, your code to instantiate the `Random` object must look something like this.

```
random = new java.util.Random(seed);
```

where `seed` holds the value from the fourth line of input.

Input translation

The "jabtalk filter" is the object that reads the source file and does all the necessary translations. This filter must be implemented in a class called `LookAheadFilter` that implements the interface `Filter` provided on our web page. You do not need to write such a class, but you are required to thoroughly test the `LookAheadFilter` class provided on our web page.

[2]The subject of generating outcomes based on a probability distribution is covered in CSC270/B70.

4 Data structure for the jabtalk engine

The most interesting task in designing the `Jabtalk` program is coming up with a data structure to support the jabtalk engine, that is, to hold the frequency counts τ_A etc. Remember that we must know the values of $\tau_A, \tau_B, \cdots, \tau_Z, \tau_b, \tau_c, \tau_p$ and τ_n (that's 30 integers) for *every* possible string of length k (that's 30^k possible strings). So in all, we need to know $30 \times 30^k = 30^{k+1}$ integer values.

One naive approach is to store all 30^{k+1} integers in an array. For most machines, we would run out of memory even if k was as small as 3 or 4. Also, since a vast majority of the array would be empty (zero), it would be an enormous waste of memory. **Thought exercise:** Why would so many of the numbers be zero?

A less memory intensive approach would be to store the count only for those strings that actually were seen in the input. This could be done with a single dimensional array where each entry holds a string of length $k+1$ and an integer indicating the number of occurrences of that string. If we keep this array sorted on the string field, then we can use a binary search to find the frequency count required by the `FreqTableEngine.generateChar` method. This would be quite efficient. However, the `FreqTableEngine.tally` method may need to shift the contents of the entire array to maintain the sorted order, and so that could be very inefficient.

What we want is something which does not waste memory and allows for efficient `tally` and `generateChar` operations. A good solution, and the one which you are required to use, is called a "trie" (pronounced "try"). A trie is a kind of tree, but its nodes can have up to one child for every character in the alphabet. Since our alphabet has 30 characters (A through Z, plus blank, comma, period and newline), our trie nodes will have up to 30 children. References to a node's children can be stored within the node using an array of length 30.

Suppose $k = 3$ and we have tallied 4 occurrences of the string CARB, 6 occurrences of CARN, 7 of CART, 2 of CABB, and 12 of CABE. On the last page of this handout, there is a diagram of a trie that represents these statistics. The diagram is simplified from the usual memory model diagram: it shows the trie nodes that would exist in the object space but doesn't show the static space or the runtime stack; references are represented using arrows rather than memory addresses; and for each node, it just shows the most important thing it contains. (For a leaf node — a node at the bottom of the tree — the most important thing it contains is an array of frequencies. For the other nodes, there is no frequency array, just an array of references to any possible children.)

Here's how to understand the trie: Starting from the root, we can follow the C-pointer to a child node, follow that node's A-pointer to another node, and follow that node's R-pointer to a node which is a leaf. This means that the string CAR was seen in the input text. That leaf contains an array of integers which tells us how many times CAR was followed by A (zero times), by B (4 times), etc. Another branch of the tree shows us that the string CAB was seen in the input text, and was followed by A zero times, by B 2 times, etc. The fact that there is no branch that corresponds to the string BAC means that BAC was not seen in the input. There are many, many other branches that don't exist, also indicating strings that were never seen. This is the key to the memory savings allowed by a trie.

Note that this trie has height exactly $k + 1$ and that each call to `tally` or `generateChar` only needs to traverse at most one path from the root to a leaf. This means these operations can be done efficiently.

5 Summary of tasks

This is what you will do:

- Test the class `LookAheadFilter` provided on our web page.
- Design an appropriate interface for a table containing frequencies of strings.

- Write a class `TrieFrequencyTable` that implements it using the trie data structure defined above.
- Write a class `FreqTableEngine` that implements `Engine` and contains an instance of `TrieFrequencyTable` for the table containing frequencies of strings.
- Write a class `Jabtalk` containing a main method that runs the whole process, following our input and output specifications exactly.

You are free to write additional classes and interfaces as you see fit. Think through these high-level design decisions before you write the code.

6 What to hand in

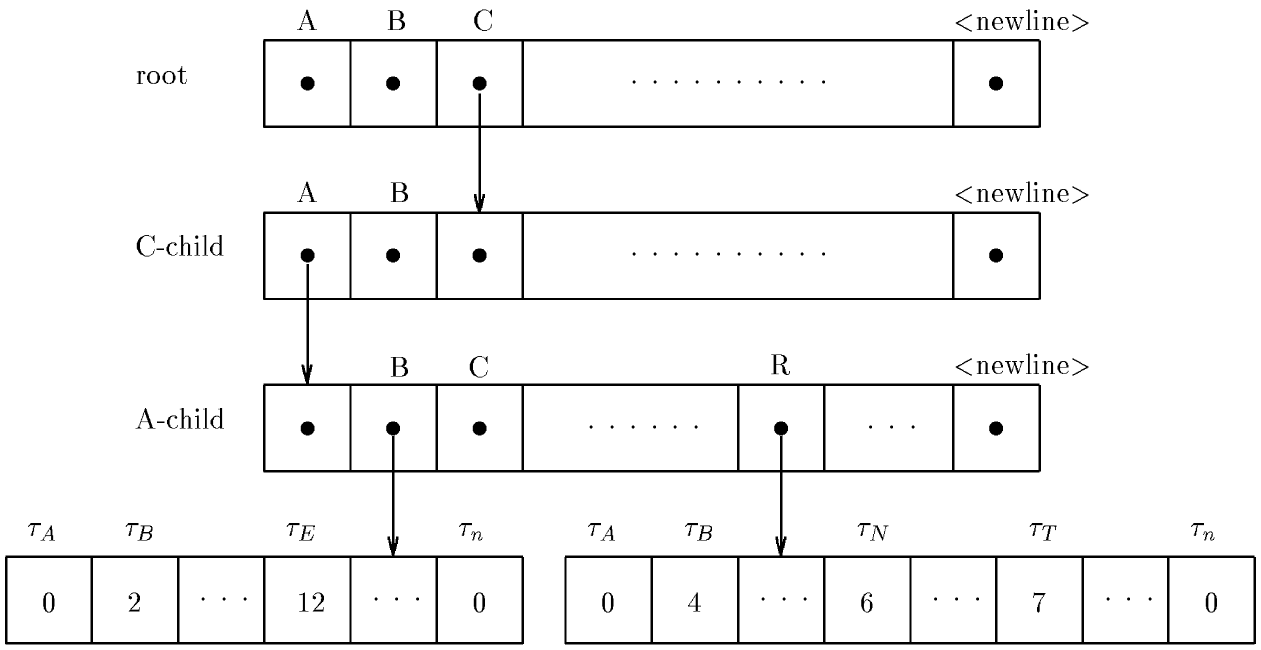
Hand in the following in an $8\frac{1}{2}$ by 11 inch *unsealed* envelope with the cover sheet (available on the web) taped to the front:

- printed listings of all your program files;
- complete and convincing testing on our `LookAheadFilter` class (see the Software Testing section in the course handbook for what we are expecting);
- runs of your entire program on input that we will give you.

You will lose marks if you seal the envelope or do not use the cover sheet.

In addition, you must also submit your program files electronically, following the usual procedure. See your campus-specific web site for details.

Example of a trie



Each leaf contains an array of integers.

Each nonleaf contains an array of node references.