CS:1210 Homework 5

Due via ICON on Friday, April 17th, 4:59 pm

What to submit: Your submission for this homework will consist of three Python files, `hw5a.py`, `hw5b.py`, and `hw5c.py`. These files should contain Python code that solve Problems (a), (b), and (c) respectively. These files should each start with with a comment block containing your name, section number, and student ID. You will get no credit for this homework if your files are named differently, have a different format, and if your files are missing your information. Your program should all be well documented, i.e., have useful comments. We will discuss what constitutes good documentation and coding style in class. Part of good documentation is choosing meaningful names for your variables.

The file `hw5a.py` should contain everything that is currently in `playLaddersGame1.py` except with body of the function `buildDictionary` replaced by new code. The file `hw5b.py` should contain the definition of the function `computeWordFrequencies` and any "helper" functions it depends on (e.g., `parse`). Finally, `hw5c.py` should contain the definition of the function `lastLetterIndex`.

(a) On my laptop the function `buildDictionary` in the program `playLaddersGame1.py` takes about a minute. This is because `words.dat` contains 5,757 words and the function `buildDictionary` evaluates all pairs of words in this file and there are 16,568,646 pairs (i.e., more than 16.5 million pairs).

This problem asks you to to implement more sophisticated algorithm to significantly speed up the function `buildDictionary`. The main idea of the algorithm is simple: two words are neighbors if (i) they have the same first letter, but they differ in *exactly* one letter in the remaining 4 letters or (ii) they have different first letters but are identical in the remaining 4 letters. The word pair (`cares`, `cakes`) is of type (i) and the word pair (`bakes`, `cakes`) is of type (ii).

To find word pairs of type (i), you should read the words into a 2-dimensional data structure, let us call this `wordMatrix`. This is a list of length 26, whose first element is the list of all words that start with the letter "a", whose second element is the list of all words that start with the letter "b" and so on. To find word pairs of type (i), we simply have to consider all pairs of words that start with the same letter and this is easy to do since these are all sitting together in one list in `wordMatrix`. Since there are far fewer pairs of words that start with the same letter, this will allow us to consider far fewer pairs than 16.5 million.

To find word pairs of type (ii), you should read the words into second 2-dimensional data structure, let us call this `suffixes`. This is a list of length 5,757 in which each element is a length-2 list. Each word `w` in `words.dat` should appear in `suffixes` as a list `[w[1:], w[0]]`. Thus the word `"house"` will appear as `["ouse", "h"]`. Once the structure `suffixes` is built it can be sorted and at this point all words that are identical in their last 4 letter will appear bunched together in `suffixes`. Then to find word pairs of type (ii) you only have to compare an element in `suffixes` with other "nearby" elements. This will allow your program to consider far fewer pairs than the 16.5 million pairs considered in the original program.

Implement this idea in the function `buildDictionary` to much more efficiently construct the dictionary of neighbors.

(b) In Homework 4(b), you were asked to write the function

```
def computeWordFrequencies(filename):
```

This function constructed and returned a list, say `L`, consisting of two lists, where `L[0]` was the list of all distinct, lowercase words of length at least 4 in the given file and `L[1]` was the corresponding list of frequencies of these words. Now rewrite this function so that it returns a dictionary `D` whose keys are all lowercase words of length at least 4 in the given file and whose values are frequencies of these words. For example, if your function returns dictionary:

```
{"hello": 3, "this": 7, "ball": 1, "bombastic": 5}
```

this would mean that in the file that was read by the function, the word `"hello"` appears 3 times, the word `"this"` appears 7 times, etc. Recall that one easy way to implement the function `computeWordFrequencies` is to use the function `parse` (see my Homework 4 solutions). Feel free to use your implementation of the function `parse` (from your Homework 4 solution) or my implementation of `parse`.

(c) The file `babyNames.txt` (posted on the course website) contains a dataset of 134 years (1880-2013) of baby names available from the US Census bureau. The data contains 1,792,090 lines or records for 92,599 distinct baby names. The first ten records look like:

```
1997,Raynee,F,9
1920,Leda,F,61
1984,Corin,M,9
1978,Shiloh,M,62
2009,Darey,M,117
1942,Elex,M,9
1976,Arthur,M,1903
1936,Fredrich,M,12
1903,Ilo,F,9
1988,Britiany,F,17
```

where each record contains a birth year, a name, a gender and the number of new babies given that name in that year (any baby name having fewer than five birth records in a given year is excluded for privacy reasons). Values or *fields* within records are separated by commas, and the records themselves are not in any particular order.

Write a function with the header

```
def lastLetterIndex():
```

that reads the records from the file and constructs and returns a dictionary, say `D`, whose keys are the letters `"a"` through `"z"`. The value associated with each such key (letter) is information about all baby names that *end* with that key (letter). Specifically, the value associated wih a key k is a list of size 134 (the number of years covered by the dataset), where the element in position i in this list contains information about baby names in year $1880 + i$ that end with k. For example, `D["a"]` is a list of size 134, where `D["a"][0]` is information about baby names in 1880 that end in letter `"a"`, `D["a"][1]` is information about baby names in 1881 that end in letter `"a"` and so on. For each lower case letter `k` and integer i, $0 \leq i \leq 133$, the item `D[k][i]` is itself a dictionary whose keys are names used in year $1880 + i$ that end in letter k. The value associated with each such key (name) is a $(gender, frequency)$ tuple.

Another way of understanding this data structure is that the first record in the data file appears in `D["e"][117]` (since `"Raynee"` ends with an `"e"` and this record is for 1997). Note that `D["e"][117]` is a dictionary whose keys are all baby names used in year 1997 that end with `"e"`. Thus `"Raynee"` is a key in this dictionary whose value is `("F", 9)`.

Correction, added on April 13: Several baby names are used for both male and female babies. This causes some trouble for the above-defined data structure because it allows only one (*gender*, *frequency*)-tuple to be associated with each name in each year. To get around this, you are required to make the following minor change to the data structure. This change is best understood using an example. The name `"Sun"` is given to both male and female babies. In 1979, 7 females and 9 males were named `"Sun"`. So information about both of these two records should appear in `D["n"][99]`. Specifically, the name `"Sun"` should appear as a key in `D["n"][99]`. The value associated with this key should be `[("F", 7), ("M", 9)]`. Thus, with every appearance of a name as a key the associated value should have the form `[("F", ff), ("M", fm)]`, `ff` is the frequency of the name in that year for female babies and `fm` is the frequency of the name in that year for male babies.
