CSC 211—Fundamental Data Structures
Apply regular expressions and use Java's `HashMap` class to create anagrams
Practical 3 Term 2 — 25 April 2016

Submit using `cd; make submit` before 1 May 2016 23h59.

First, read the appendix. Use Java's regular expression classes `Pattern` and `Matcher` unless you can avoid using them, and also the classes for a hash table `HashMap <String, Integer>` and `HashMap <String, String>` to determine (a) the *number of unique* words in the given text, (b) the *ten most common* words, (c) the *number* of words that have anagrams in the text, and (d) the *longest list of anagrams* retrieved from the text.

---

1. *Setting up*: *Create a new* `32practical` *subdirectory* in parallel with older directories by *copying* an old directory as follows:

```
cd surname,firstname
cp -r 22practical/ 32practical
cd 32practical
```

   When you have done this, run `make clean` and then edit the `Makefile` so that it makes `OBJ = applyHash`. Alter the name of your main class to `applyHash` and delete all the unneeded code—most of it—from this directory and also rename the file with the main method to `applyHash.java`.

2. The data contains strings such as ⌜`nation,`⌟ where the ⌜`,`⌟ is unnecessary. If the ⌜`,`⌟ is not ignored words such as ⌜`nation,`⌟ will be regarded as words different from ⌜`nation`⌟ which will lead to many words being counted unnecessarily. Words such as ⌜`The`⌟ still need to be cleaned so that valid 'words' consisting of lower case letters only are processed. These words are extracted from the original given text using the regular expression ⌜`[a-zA-Z']+`⌟. In Java the classes `Pattern` and `Matcher` must be used to extract each individual word. Subsequently the upper case letters remaining in the `String word` must be turned into lower case letters using the `String` function `word.lower()`.

3. Once the word has been extracted, it must be processed by inserting it into a Java `HashMap <String, Integer>` object in order to count the number of appearances of each word in the given text. In order to collect anagrams each word must furthermore be inserted into another Java `HashMap <String, String>` object that collects anagrams into lists. Think carefully about these structures before you dive into coding.

4. The text can be found in the `notes/ds/data` file in the Sunlab in the file `data.ulysses`. Code that attempts to answer this practical in python was discussed in class. See the Appendix which contains last year's instructions.

5. In summary your code must *print* out (a) the number of *unique* words in the given text, (b) the ten *most common* words in the text in decreasing order of frequency, (c) the number of words that have *anagrams* in the text, and (d) the *longest list of anagrams* retrieved from the text.

Appendix: Details of the algorithm for creating anagrams

This is a verbatim extract explaining the construction of *anagrams* from a practical done by the first years in *COS101 B2practical* on 14–15 May 2014.

...
Dictionaries will make it possible to do the *anagrams* efficiently.
Anagrams

Find all the anagrams in the file `data.ulysses` in the first year notes file in the Sunlab. Do not copy the file to your file system, read it directly.

Your code must find all the words in the file that are anagrams of one another, e.g. the program must identify the anagrams such as `medical, decimal, claimed` or `reread, dearer, reader, reared`. The idea of how the code works is very simple if you use a dictionary. Read and clean up every word in the book. Next make an 'alphabetized key' as follows. The word 'reader' is alphabetized as '`adeerr`', as are '`dearer`', '`reader`' and '`reared`'. The key `"adeerr"` is used to store the list of anagrams made from it, such that there will now be an entry in the dictionary with

```
{"adeerr": ["reread", "dearer", "reader", "reared"]}
```

The key itself is very seldom an actual word.

Every word in the file must be read perhaps with

```
f = open("/export/home/first/notes/data.ulysses")
line = f.readline()
while line != "":
  words = line.split()
  for w in words:
    # clean up word but leave apostrophes
    w = w.strip(".,;:␣etc")
    # alphabetize w and store in dictionary with its value
  # anagrams are those words whose lists are two or longer
```

The dictionary `D`, much like a list, is created setting `D = {}`, or else by putting `D = dict()`. To insert an item `w = "skills"` into `D` given its alphabetized value `key = "iklls"`, simply run

```
if key not in D:
  D[key] = [w]
else:
  D[key].append(w)
```

This will initialize `D[key]` to `[w]` if `w` has never been encountered before, and will append `w` to the existing list in the dictionary entry for `key` otherwise.

On completion of building `D`, all the anagram lists can be printed by

```
for key in D.keys():
  print (key, D[key])
```

...
CSC211: Note that today this needs to be done in Java.