# Programming Assignment #4
# CS253, Fall 2017
# Due: Tuesday, Oct. 10th

## "*Word Stemming*"

## *Motivation & Description*

PA2 divided a text input file into whitespace-delimited strings. PA3 broke apart these strings into word strings (which contain letters, numbers, apostrophes, and under special circumstances periods and commas) and punctuation strings (everything else). PA3 then reported how many times each word and punctuation string occurred. PA4 refines the word strings by *word stemming*. The idea of word stemming is to remove grammatical endings from words, leaving just the semantic stem. For example, *talk, talks, talked* and *talking* are all the same verb, just with different endings. PA4 will stem word strings before reporting how many times each (possibly stemmed) string occurs. In general, counts should become higher since (for example) *talk, talks, talked* and *talking* are all stemmed to the same string, so the input "He talks and talks about talking" would report that the string *talk* appeared 3 times.

This project is assigned at a point where the emphasis of the course is changing from memory management to object oriented design. This assignment therefore has a lot of parts to it, and the challenge is to integrate all these new rules with the code you already have from PA2 and PA3. I recommend that you think about design. How many classes do you need, and what are their interfaces? What methods should each class have, and how are they organized? How will you systematically test this code? We are now reaching the point where system design, documentation and testing become critical.

## *Task*

As in PA3, your program will take a single file name as an argument. Any input file that was valid for PA3 is still valid. The output is also similar: your program should write one line of output for every unique string in the input, sorted lexicographically, followed by a space and the number of times that string occurs. The difference between PA4 and PA3 is that now word strings are stemmed before they are sorted and counted.

To be precise, punctuation strings are not changed. In addition, word strings that contains one or more capital letters, one or more digits, or the '+' marker are left unchanged, as are words with two characters or less, like *it* or *as*. All other words are stemmed according to the Porter Algorithm #2 word stemming algorithm, as defined below[1].

Porter Algorithm #2 is an 8-step algorithm. The steps must be performed in the order given, but every step has the same form. A Porter step is a set of grammatical suffixes, conditions and replacements. For each step, find the longest suffix that matches the ending of the word. If the conditions of the suffix are satisfied, remove the suffix and add the replacement. (Empty conditions are always satisfied.) If the suffix matches but the conditions are not satisfied, do nothing.

[1] There are some inconsistent variations of this algorithm on the web. You must use the one described here.

For example, Step #3 has the suffixes *eed, eedly, ed, edly, ing* and *ingly*, but all have conditions attached. The word *agreed* matches the *eed* suffix and satisfies its condition, namely that the suffix is found in Region1, since Region1(*agreed*) = *reed.* Therefor *agreed* becomes *agree.* The word *seed*, on the other hand, ends in *eed* but does not satisfy the condition (Region1(*seed*) is empty), so *seed* remains *seed*, even though it also ends in the shorter *ed* and satisfies its conditions.

Before listing the 8 steps, we need to establish definitions for some of the terms used in conditions.

- A *vowel* is any of {'a', 'e', 'i', 'o', 'u'} or the letter 'y' UNLESS the 'y' is the first letter in a word or immediately follows a vowel. (Note that this is a recursive definition.) Therefor the letter 'y' is considered a vowel in the word *try* but a consonant (non-vowel) in the words *yellow* and *today*.
- A *double* is any of the following letter pairs: {'bb', 'dd', 'ff', 'gg', 'mm', 'nn', 'pp', 'rr', 'tt'}.
- A *valid li-ending* is one of {'c', 'd', 'e', 'g', 'h', 'k', 'm', 'n', 'r', 't'}.
- *Region1* is the substring that follows the first consonant (non-vowel) that follows a vowel. Region1 may be empty (it often is for short words). Examples: Region1(*try*) is empty, but Region1(*definition*) is *inition.*
- *Region2* is the Region1 of Region1. In other words, Region2(*definition*) = Region1(*inition*) = *ition.*
- The *preceder* is the part of a word before a given suffix. For example, if the suffix is *ing* then the preceder of *talking* is *talk.*
- A string ends in a *short syllable* if either
  1. It ends with a non-vowel followed by a vowel followed by a non-vowel that is not one of {'w', 'x' or 'y'}
  2. The string is only two characters long, and is a vowel followed by a non-vowel
- A word is called *short* if both (1) it ends in a short syllable and (2) its Region1 is empty. For example, *bed, shed,* and *shred* are short words, but *bead, embed* and *beds* are not.

Step #1 starts with a special case: if the word (i.e. string) begins with an apostrophe, remove the apostrophe. Then apply the longest of the following substitutions that apply:

| Suffix | Conditions | Replacement |
|---|---|---|
| 's' | (none) | (none) |
| 's | (none) | (none) |
| ' | (none) | (none) |

Step #2 is as follows (remember, use only the longest that applies):

| Suffix | Conditions | Replacement |
|---|---|---|
| sses | (none) | ss |
| ied<br>ies | (none) | If preceder contains more than one letter, replace with *i.* Otherwise replace with *ie.* |
| us | (none) | us |
| ss | (none) | ss |
| s | preceding word part contains a vowel not immediately before the *s* | (none) |

| | | |
|---|---|---|
| | | |

Step #3 is as follows:

| Suffix | Conditions | Replacement |
|---|---|---|
| eed<br>eedly | Suffix must occur in Region1 | ee |
| ed<br>edly<br>ing<br>ingly | The preceder must contain a vowel | (none), but:<br>(1) if preceder ends in at, bl, or iz, add an e<br>(2) if preceder ends in a double, remove the last letter<br>(3) if the preceder is short, add an e |

Step #4 has only a single rule, as follows:

| Suffix | Conditions | Replacement |
|---|---|---|
| y | Suffix follows a non-vowel that is not the first letter of the preceder. | i |

Step #5 has the most rules (and remember to use only the longest that applies), but the good news is that there are relatively few conditions:

| Suffix | Conditions | Replacement |
|---|---|---|
| tional | (none) | tion |
| enci | (none) | ence |
| anci | (none) | ance |
| abli | (none) | able |
| entli | (none) | ent |
| izer<br>ization | (none) | ize |
| ational<br>ation<br>ator | (none) | ate |
| alism<br>aliti<br>alli | (none) | al |
| fulness | (none) | ful |
| ousli<br>ousness | (none) | ous |
| iveness<br>iviti | (none) | ive |
| biliti<br>bli | (none) | ble |
| ogi | Suffix preceded by *l* | og |
| fulli | (none) | ful |
| lessli | (none) | less |
| li | Preceder ends in a valid li ending | (none) |

Step #6 rules all require that the prefix be in Region1 (one of the rules has a stronger condition, and requires the prefix to be in Region2):

| Suffix | Condition | Replacement |
|---|---|---|
| tional | Suffix is in Region1 | tion |
| ational | Suffix is in Region1 | ate |
| alize | Suffix is in Region1 | al |
| icate<br>iciti<br>ical | Suffix is in Region1 | ic |
| ful<br>ness | Suffix is in Region1 | (none) |
| ative | Suffix is in Region2 | (none) |

Almost done! STEP #7:

| Suffix | Condition | Replacement |
|---|---|---|
| al<br>ance<br>ence<br>er<br>ic<br>able<br>ible<br>ant<br>ement<br>ment<br>ent<br>ism<br>ate<br>iti<br>ous<br>ive<br>ize | Suffix must appear in Region2 | (none) |
| ion | Suffix must appear in Region2 and the preceder must end with an *s* or *t*. | (none) |

Finally, the last step:

| Suffix | Condition | Replacement |
|---|---|---|
| e | Suffix appears in Region2 OR suffix appears in Region1 and preceder does not end in a short syllable. | (none) |
| l | Suffix appears in Region2 and preceder ends in *l* (so word ends in *ll*) | (none) |

## *Example*

With this assignment (and this assignment only) we are providing a test file and the correct output for the test file. Note that the test file does not exercise all of the rules above, but it exercises many of them and should get you started.

## *Submitting Your Work*

To submit your program, first create a single tar file containing all of your source files (i.e. your .cpp and .h files) and your makefile, but not your compiled files (no executable or .o files, please). Then submit the tar file as PA3 on Canvas.

## *Grading Your Homework*

To grade your assignment, the GTAs will unpack your archive in an empty directory. They will compile your code by typing 'make'. This command must compile your code from scratch, and it must produce an executable called PA4 in a directory called PA4. If your code does not compile using make, you will receive a zero for the assignment.  Compiling your code should not produce any warning messages. Although we will not deduct points for warning messages, we may in future assignments so get in the habit of making sure you code compiles cleanly. Assuming your program compiles, it will be graded by how it performs on test cases, including test cases with errors in the input file. If the input file contains errors, you will receive full credit if your program returns -1 to main. (In case of an error, it should also print a meaningful error message to std::cerr.) If the input does not contain errors, your program should return 0 to main.

## *Hints*

- The risk in this assignment is that you will write spaghetti code and it will get out of hand. I would not allow any method to become longer than 15 lines. Longer methods should be split up into shorter ones.
- This would also be a good time to think about classes. How many do you have? How many do you need? My implementation has three classes, but your design may vary. But if your design has one class (or worse, none), I worry.

## *Policies*

All work you submit must be your own. You may not submit code written by a peer, a former student, or anyone else. You may not copy or buy code from the web. The department academic integrity policies apply.

You may not submit your program late. To receive credit, it must be submitted on Tuesday, Oct. 3rd. There is no late period. The exception is an unforeseeable emergency, for example a medical crisis or a death in the immediate family. If an unforeseeable emergency arises, talk to the instructor.