Assignment—Part 1

COSC122—2014s2

August 14, 2014

1 Overview

1.1 Introduction

Did Justin Bieber have a larger vocabulary than Shakespeare? This assignment is based around comparing the vocabularies of authors, including poets and songwriters. We'll provide some sample texts, but you can analyse your own if you want.

The work required is designed to help you understand both the theoretical and practical concerns related to implementing a variety of algorithms. As a computer scientist, you will often encounter problems for which there is no pre-existing solution. Your knowledge of and experience with algorithms will directly affect the quality and efficiency of systems that you produce, especially when processing problems that involve large data-sets.

For part one we are going to process a list of words used in a book, song or document, counting how often each one is used. In part two we are going to compare the vocabularies of different writers. Part 1 (based on searching algorithms) is described in this handout; Part 2 (based on sorting algorithms) will be distributed when the relevant material has been covered in lectures.

The types of algorithms you will be writing can be used for many applications, for example to create wordles, such as the one in Figure 1, which is a wordle of this document!

1.2 Due Dates

The two parts of the assignment together are worth 20% of your overall course grade.

Part 1 of this assignment is worth 10%, and has five tasks for you to complete. The due date for the first two tasks of Part 1 is Tuesday 16 September at 4:30pm, and the remaining tasks are due Friday 22 September at 4:30pm.

Part 2 of the assignment (not included in this document) is worth the remaining 10%, and will be released in term 4.

The drop dead date is one week after each due date. Late submissions will be accepted between the due date and the drop dead date, but there will be a penalty of 15% of the maximum possible grade for late submissions. No submissions will be accepted after the drop dead date.

Figure 1: Wordle of this document

1.3 Submission

A 'Part 1 - Testing' quiz is now available on the quiz server. This quiz will *not* be marked but can be used to test your code for each section of part 1 of the assignment. You can submit code to this quiz as many times as you want without any marks being recorded.

For the final submission, your code must pass pylint with no errors or warnings, and your algorithms should be implemented cleanly and clearly. A separate quiz will be provided nearer the due date for making final submissions; you will only be able to submit to the final quiz once, so you will need to test your assignment carefully on the Testing quiz before making the final submission. The final marking will include test cases that aren't in the Testing version of the quiz, so you should think of your own tests for your implementations; also, you shouldn't try to game the system (e.g. by forcing the right number of comparisons for the tests), as our hidden tests are designed to uncover this.

In addition, for Part 2 you will need to submit a graph and very brief comment about the relative performance of the methods that you implented. This should be a PDF file, submitted via the quiz server assignment page.

1.4 Implementation

All the files you need for this assignment can be found on the quiz server. Do not import any additional standard libraries unless explicitly given permission within the task outline. For each section there is a file with a function for you to fill in. You need to complete these functions, but you can also write other functions that these functions call if you wish.

1.5 Getting Help

The work in this assignment is to be carried out individually, and what you submit needs to be your own work. You must not discuss code-level details

with anyone other than the course tutors and lecturers. You are, however, permitted to discuss high-level details of the program with other students. If you get stuck on a programming issue, you are encouraged to ask your tutor or the lecturer for help. You may not copy material from books, the internet, or other students. We will be checking carefully for copied work. If you have a general assignment question or need clarification on how something should work, please use the class forums on Learn, as this enables everyone to see responses to common issues. Remember that the main point of this assignment is for you to exercise what you have learnt from lectures and labs, so make sure you do the relevant labs first, and don't cheat yourself of the learning by having someone else write material for you.

2 Vocabulary counting

For each section in Part 1 of the assignment the functions you write will need to take a list containing every word in a text document, all in lowercase, with the punctuation removed (i.e. a list of strings). For example the previous sentence in this format would be:

```
['for', 'each', 'section', 'in', 'this', 'assignment', 'the',
'functions', 'you', 'write','will', 'need', 'to', 'take', 'a',
'list', 'containing', 'every', 'word', 'in', 'a', 'text',
'document', 'all', 'in', 'lowercase', 'with', 'the',
'punctuation', 'removed', 'ie', 'a', 'list', 'of', 'strings']
```

In each section you will need to use some specific data structures, which you will import into your program. You do not have to implement these data structures, you will just have to interact with them. They are provided in the files `classes_1.py` and `classes_2.py`. Details of these data structures can be found in the relevant sections.
You will store each of the words in the document in a data structure, along with a count of the number of times that word appears in the document. You will also need to count the number of key comparisons your code makes to process the list it is given, and report this once your function finishes executing. Note that we only count *key* comparisons (that is, comparisons of *words* in the text), and not comparisons of indexes, counters and so on.
The data structures we provide will also automatically count comparisons, which will allow you to check how many comparisons have *actually* been made, but this is intended for debugging only. If you use this function in any code you submit (rather than doing the comparisons counting yourself) then **your code will fail the coderunner tests** used for submitting the assignment. The online tests will provide the expected number of comparisons that your code should make. For the first two algorithms you should be able to match the exact number of comparisons, but for binary search we'll be refining the boundaries to make sure we accept sensible algorithms that are in the right ballpark. This means that the target number of comparisons may be widened as the assignment goes on. If you think that you have an implementation that is very close to the target number, and works reliably, you can ask to have the target reconsidered.

2.1 Provided Classes

The main class provided is a `CounterList`, which is a list of `CounterNode` objects. The `CounterList` object is very similar to a normal Python list but has several limitations. You can only add `CounterNode` objects to it and the built in Python sorting and searching methods will not work with it. You can however access the `CounterNode` objects in the list by indexing into it. Each `CounterNode` object in the list contains two values: a string (which is a word from the text), and an integer (which is the current count of how many times that word has been observed). You must not change any of the classes provided, and only access them through in following ways:

- `my_list = CounterList()` creates a new `CounterList` that can be used for recording the words and counts. [[supports del, len, ==, insert, append]]
- `CounterNode(word)` creates a new `CounterNode` object for the word, with a default count of 1. This can be added to a list [[counter objects can only be created, but to access them use the following:]]
- `my_list[i].word` gives the word (string) at position *i* of the list. This can be compared with other words
- `my_list[i].count` gives the count of the word at position *i* of the list; this has the usual properties of an integer, and in particular it can be set and incremented (`+= 1`) as occurrences of a word are found.
- `print(my_list)` will print your list in the correct format
- `CounterList.get_comparisons()` will return the actual number of comparisons that have been performed. Remember this is for debugging only and using this in submitted code will cause your code to fail the submission tests.

The following example code and Figure 2 should help you understand how to use these classes:

```
>>> new_list = CounterList()
>>> new_counter = CounterNode('hello', 1)
>>> new_list.append(new_counter)
>>> new_list
['hello': 1]
>>> new_list[0]
'hello': 1
>>> new_list[0].word
'hello'
>>> new_list[0].count
1
>>> new_list.append(CounterNode('world', 1))
>>> new_list[0].count += 1
>>> new_list
['hello': 2, 'world': 1]
```

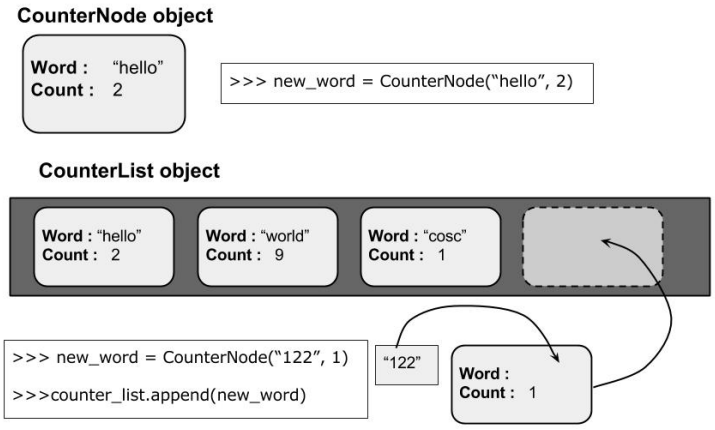

Figure 2: The CounterNode and CounterList classes

3 Tasks

3.1 Word Counter using Sequential Search

[Worth 2%]
This first method isn't going to be very efficient, but it's a starting point! You should start with an empty `CounterList`. Go through each word in the list (the parameter `word_list`) and check if it has previously been added to the `CounterList`. If it has been then increment its count, if it hasn't then append it to the end of the list as a `CounterNode` object. To check if the word is in the `CounterList` you need to sequentially search through the list starting at entry 0, because the list will not be sorted. To answer this question use the function declaration in the file *word_counter_sequential.py*

3.2 Word Counter in frequency order using Sequential Search

[Worth 2%]
This method aims to improve the performance of the sequential search by listing the words in reverse frequency order. This means words that are seen most frequently will be moved to the start of the `CounterList` and words that are seen least frequently will be moved towards the end of the list. This still requires a sequential search (the words aren't in alphabetical order), but hopefully the search will find the word earlier.
Your algorithm should work similarly to the previous word counter using sequential searching, except whenever you increment the count of a word you will then check if that word needs to be moved to another position closer to the front of the `CounterList`.
You should only swap objects in the list when it is necessary. When moving a word (or more precisely the `CounterNode` object containing that word) further up the list you should find where it now belongs in the list and swap it with the nearest (rightmost) item there.
This swapping method is illustrated in Figure 3.

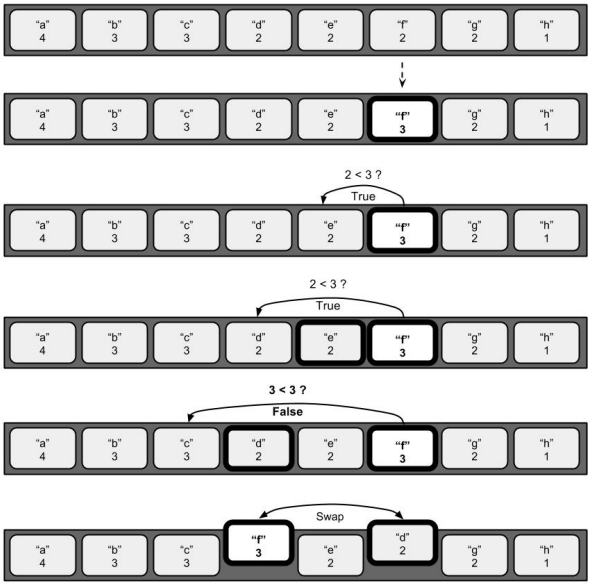

Figure 3: Incrementing then swapping CounterNode objects

Complete the function in *word_counter_frequency_order.py* to answer this part.

3.3 Word Counter using Binary Search

[Worth 2%]
Similarly to before, in this variation each time you process an element from the original list of words you find it in the list of counters and either increment its count or add it to the list. The difference is this list will be sorted (alphabetically) so you can (and should) use a Binary search to check if the word you are looking for is in the list. When you search through the list to see if a word is in it, either you will find the word and increment its count, or when you discover a word is not in the list you will have also located the position in the list where that word would belong so you can insert it into that position, thus keeping the list sorted. (This is a variation of insertion sort, but don't use a standard insertion sort — it's slower!)
The binary search that you use should be the faster one that doesn't check for key equality every time the range is halved; in fact, you should check for having found the key only when your left and right pointers have converged; thus, you will only use "==" on the key once in each search.
Complete the function in *word_counter_binary_search.py* to answer this part.

3.4 Word Counter using a Chaining Hashtable

[Worth 2%]
In this variation, each time you take a word from the list of words you will

check if it is in a hash table and increment its count if it is. If it is not then insert its count into the table, using chaining if there is already an object in the hash table slot. Chaining is done with a linked list. The list node objects are predefined, including a constructor.
To make marking consistent, we provide `hash_word(item,slots)`, that hashes an item (a string) to a number in the range `0..slots-1`. You must use this hash function!
Your initial hash table should be a list with the required number of slots in it, each initialised to `None` to show that the slot is unused. When the slot is needed, you will replace the entry with a new `CounterLinkedList()`, and initially it will have one `CounterNode` attached to it for the word that was hashed. From then on you should insert new items at the **front** of the linked list (chain).
The pre-written classes for the hashing exercise are as follows:

- `CounterLinkedList()` creates a new `CounterLinkedList` object, which is the head of a linked list of `CounterNode`s. Note that unlike a `CounterList()` you cannot index into this.
- `CounterNode(word)` creates a new Counter object that is a node in the linked list for the word, with a default count of 1. This `CounterNode` class is almost identical to the `CounterNode` class used in the previous tasks, except it also has a `next` attribute which points to the next item in the linked list, or `None` if it is the last item in the list.
- `CounterLinkedList.get_comparisons()` will give you the actual number of comparisons used. Remember this is for debugging only and using this in submitted code will cause your code to fail the submission tests.

The following example code should help you understand how to interact with these objects

```
>>> new_list = CounterLinkedList(CounterNode('hello', 1))
>>> new_list
['hello': 1]
>>> new_list.head
'hello': 1
>>> node1 = new_list.head
>>> node1.next = CounterNode('world', 1)
>>> new_list
['hello': 1 -> 'world': 1]
```

Complete the function in *file word_counter_hash_table.py* to answer this question. **Note:** this function imports the module *classes_2*, which is different to the other questions.

3.5 Comparisons of methods

[Worth 2%]

In addition to submitting the implementations above, you should submit a very brief report comparing the performance of the different methods. You will need to choose your own test data (you should say what it is), and show how the performance of each approach varies with the amount of data it is given. This should be submitted as a PDF file through the quiz Moodle site. The report need only be one page, with supporting graphs, and two or three sentences comparing the different methods.
You should show:

- How the number of key comparisons for the sequential search method increases with the number of words given to the function
- How the number of key comparisons for the frequency sorted search method increases with the number of words given to the function
- How the number of key comparisons for the binary search method increases with the number of words given to the function
- How the number of probes (key comparisons or empty slots checked) increases with the number of words given to the function. For hashing, experiment with different load factors including small ones (e.g. 10%), a typical one (e.g. 60%), and some high ones (e.g. 200% and 400%). Try to choose values that show an interesting difference in performance.
- A brief summary of the main trends you have observed in the graphs (how the number of comparisons varies with the number of words processed, including which method is best, and if one is better for certain values of n.)

3.6 Optional (no marks): Hashing with open addressing

If you finish the assignment in plenty of time, try implementing an open addressing version of the word counter, and include that in your results. Note that load factors close to 100% could cause interesting behaviour with open addressing!