22C:16 (CS:1210) Homework 4

Due via ICON on Friday, March 29th, 4:59 pm

What to submit: Your submission for this homework will consist of four Python files, `hw3Solution1.py` (for Problem 1), `binarySearch2.py` (for Problem 2), `hw3Solution3.py` (for Problem 3), and `hw3Solution4.py` (for Problem 4). These files should also start with your name, section number, and student ID appearing at the top of the file as Python comments. You will get no credit for this homework if your files are named differently, and if your files are missing your information. In addition you should submit a pdf file called `homework4.pdf` containing written answers to Problems 1 and 4.

Introduction

Our solution to Homework 3 is somewhat inefficient - it takes 3-4 minutes of my machine to process Tolstoy's "War and Peace." With our newfound knowledge regarding the efficiency of list operations and also how *binary search* is so much more efficient than the Python `in` operator and the Python `index` method for searching lists, let us try and improve the efficiency of our code in `hw3Solution.py` (posted on the course page). In particular, let us focus on improving the efficiency of the function `computeFrequencies` because this function takes up the bullk of the running time of the program.

Recall that this function maintains two lists: (i) `masterList`, which is all the words in the text file seen so far and (ii) `frequencies`, which maintains word-frequencies in a parallel list. When we process a word, say `w`, from the text file, the first thing we do is search for `w` in `masterList`. Then, one of two things can happen:

1. `w` is already in `masterList`, in which case the corresponding frequency in the list `frequencies` is incremented.

2. `w` is not in `masterList`, in which case it is appended to `masterList` and the frequency 1 in appended to `frequencies`.

From the point of view of list operations, for *every* word `w` in the input file, we perform a search followed by either an element access via an index (in Case 1) or two calls to `append()` (in Case 2). Thus, for every word `w` in the text file, we perform a costly search operation on the list of all distinct words in the input file processed thus far.

One way to speed up this code significantly would be to speed up the search operation. Here is the general idea. We could try to replace the current (linear) search by *binary search*, but binary search only works on sorted lists. So it is important to maintain `masterList` in sorted order. We could do this by ensuring that whenever we see a *new* word (i.e., one that is not already in `masterList`), instead of simply appending it to `masterList`, we insert it into `masterList` in an appropriate location. For example, if `masterList` equals `["hello", "okay"]` and the word `"kindle"` comes along, we should insert `"kindle"` between `"hello"` and `"okay"` in order to make `masterList` equal `["hello", "kindle", "okay"]`. Note that `masterList` continues to be in alphabetical order. This means that whenever we encounter a new word we perform a costly `insert` operation rather than a cheap `append` operation. Thus, in order to be able to improve the performance of search, we have to maintain a sorted list, which in turn means that we have to spend more time on inserting each new word in an appropriate location. This trade-off can be summarized as follows. In the original version of `computeFrequencies`, the operations we perform are:

For New words: (i) slow search, (ii) append, (iii) append

For Repeat words: (i) slow search, (ii) access via index.

By "repeat words" we mean those words in the text file that have already been encountered earlier in the text. In the new version of `computeFrequencies`, the operations we would like to perform are:

For New words: (i) fast Search, (ii) insert, (iii) insert

For Repeat words: (i) fast search, (ii) access via index.

This type of trade-off, where in order make one operation cheaper we need to make another operation more costly, is quite common in many applications.

The Problems

Whether this trade-off is in our favor (i.e., reduces running time) depends on how many new words we encounter relative to the total number of words. It is reasonable to expect that most words in a text such as "War and Peace" are going to be repeat words and if this is truly the case then we would have substantially speeded up our code by implementing the ideas described above. This homework leads you through the steps of implementing these ideas. You should start with the code `hw3Solution.py` that we posted and solve the following problems.

1. Modify the function `computeFrequencies` so that it outputs (i) the total number of words (not just those that are distinct) in the input file and (ii) the percentage of these words that are repeat words. You just need to add some counters and print statements to the function. Report the total number of words and the percentage of repeat words you get for each novel. Do you think the output justifies the above-described approach to speeding up `computeFrequencies`?

2. The function `binarySearch.py` (posted on the course page) searches a sorted list `L` looking for an element `k`. If `k` is found in `L`, the function returns the index of a slot containing `k`. If `k` is not found in `L`, then the function returns -1. Modify `binarySearch` so that if `k` is not in `L`, instead of returning -1, it returns the location of the slot where `k` would be present, if it were inserted into the (sorted) list. For example, if `L` is `["abode", "above", "absolute", "zebra"]` then `binarySearch(L, "hello")` should return 3, `binarySearch(L, "zip")` should return 4, and `binarySearch(L, "abhor")` should return 0.

3. Now modify `hw3Solution.py` to implement the idea for speeding it up, described above in the **Introduction** section of the homework. Most of your changes will be to the `computeFrequencies` function. We will let you figure out what these changes should be exactly. There are some other minor changes that you should not forget to make: (i) you should either `import` or physically insert the new `binarySearch` function (solution to Problem 2 above) into `hw3Solution.py`, (ii) you should get rid of any counters or `print` statements you inserted into `hw3Solution.py` for solving Problem 1. (Run your modified program and compare your output to what the unmodified program produces, as a way of checking correctness.)

4. Time the function `computeFrequencies` in the original version of the program and then time it in the modified version. Do this just for the processing of "War and Peace." Report the two times and explain in 1-2 sentences whether our plan for speeding up `computeFrequencies` worked and if so why.
