Assignment 5

Dictionaries, Recursion, and Sorting

Submit a single zip file called **assignment5.zip**. This assignment has 50 marks.
See the marking scheme that is posted on the course webpage.

Problem 1 (Recursive Cached Factorial)

Create a Python file called **factorial.py** that has a single function called **cachedfactorial(int, dict)**. The cachedfactorial function will accept an integer and a dictionary as arguments. The function must recursively calculate the factorial value. Additionally, this function must use/update the dictionary-based cache to save any intermediate calculation results and terminate early if a required value can be found in the cache. For example, upon completion of cachedfactorial(5), which computes the value of 5! (5*4*3*2*1), the cache should have stored the values of 5!, 4!, 3!, 2!, and 1!. If you try to compute the value of 7! (i.e., 7*6*5*4*3*2*1) afterward, you should not need to recursively compute the 5*4*3*2*1 part again. Instead, you should be able to use the value of 5! stored in the cache to stop early.

Problem 2 (Text Analysis)

Create a Python file called **analysis.py** that will perform text analysis on files. For this question, assume that each space (" ") in the document separates one word from the next – so any use of the term 'word' means a string that occurs between two spaces (or in two special cases, between the start of the file and a space, or between a space and the end of the file). You can also assume there is no punctuation or other symbols present in the files – only words separated by spaces. If you want to see examples of the type of text, look in the testfile_.txt files included on cuLearn.

You must implement and test the following functions inside of your analysis.py file:

1) load(str) – Takes a single string argument, representing a filename. The program must open the file and parse the text inside. This function should initialize the variables (e.g., lists, dictionaries, other variables) you will use to store the information necessary to solve the remainder of the problem. These variables should be created outside the function so that they can be accessed by the other functions you will define. Your load function should just modify these global variables. This way, the file contents can be parsed once and the functions below can be executed many times without re-reading the file, which is a relatively slow process. This function should

also remove any information stored from a previous file when it is called (i.e., you start over every time load is called).

2) commonword(list) – Takes a single list-type argument which contains string values. The function should operate as follows:
    a. If the list is empty or none of the words specified in the list occur in the text that has been loaded, the function should return None.
    b. Otherwise, the function should return the word contained in the list that occurs most often in the loaded text - or any one of the most common, in the case of a tie.
3) commonletter(list) - Takes a single list-type argument which contains single character strings (i.e., letters/characters). The function should operate as follows:
    a. If the list is empty or none of the letters specified in the list occur in the text that has been loaded, the function should return None.
    b. Otherwise, the function should return the letter contained in the list that occurs most often in the loaded text - or any one of the most common, in the case of a tie.
4) commonpair(str) – Takes a single string argument, representing the first word. This function should return the word that most frequently followed the given argument word (or one of, in case of ties). If the argument word does not appear in the text at all, or is never followed by another word (i.e., is the last word in the file), this function should return None.
5) countall() – Returns the total number of words found in the text that has been loaded. That is, the word count of the document.
6) countunique() – Returns the number of *unique* words in the text that has been loaded. This is different than the previous function, as it should not count the same word more than once.

You can use the analysistester.py file from cuLearn, along with the posted text files, to test your functions. You can also create additional text files of your own to further test the correctness of your program. If you want an easy way to find out the necessary information about a file you created, you can copy/paste the contents into the form at http://textalyser.net. It can give you the total word count, unique word count, word frequency to determine the most common word, and common word pairs.

Problem 3 (Counting Sort)

In general, the best performance you can hope to accomplish with sorting is an O(n log n) solution. When you know that the number of unique values you will be sorting is relatively small compared to the size of the list (n), you can improve on this solution by using a method called counting sort. The general premise of counting sort is as follows:

1. Iterate over the list one time and count the frequency of each value that occurs. This can be done in O(n) time.
2. Sort the list of unique values, which is much smaller than the size of the list, and therefore much faster to sort. This can be done in O(k log k) time, where k is the

number of unique values in the list. Since k is much less than n, this is much faster than O(n log n)

3. Generate a new, sorted list by iteratively adding the correct number of repetitions of each unique value in the correct order. This can be done in O(n) time.

Create a Python file called **count.py** and implement a function called **countsort** that takes a list as input. This function must return a new, sorted copy of the input list using the counting sort method described above. Note – for this question, you can use a built-in sorting function that Python offers to sort the unique values or implement any other sorting algorithm if you prefer.

Problem 4 (Comb Sort)

Comb sort is a variation of bubble sort that generally performs more efficient sorting. It accomplishes this by moving low values near the end of the list further toward the front of the list than bubble sort would during early iterations. Pseudocode for the comb sort algorithm is included at the end of this problem. Create a Python file called **comb.py** and implement a function called **combsort(list)** that does the following:

1. Takes a 2D list argument that contains information representing x/y points in 2D space. Each element in the list will be a list with 2 items – an x and a y coordinate. For example, the list could be [[0, 1], [2, 1], [3, 3], [1, 1], …]
2. Performs an in-place sort (i.e., does not create a new list, but modifies the original) using the comb sort algorithm. This must sort the 2D list such that points with lower Euclidean distance to the origin (0, 0) appear earlier in the list. In this case, you are comparing distances instead of directly comparing list values – it may be useful to implement and use a distance calculation function. Note – the Euclidean distance of a point (x, y) from the origin (0, 0) can be calculated with the following equation:

$$\text{distance}(x,y) = \sqrt{x^2 + y^2}$$

3. Does not return a value. As the input list is sorted in place, it will be modified directly and these modifications will be reflected outside the function, so a return value is not needed.

Example outputs of original lists and the same lists after the combsort function has been called:

List before sorting: [[2, 1], [2, 2], [3, 3], [3, 1], [1, 1], [1, 2]]
List after sorting: [[1, 1], [1, 2], [2, 1], [2, 2], [3, 1], [3, 3]]

List before sorting: [[3, 3], [2, 2], [1, 2], [2, 1], [3, 1], [1, 1]]
List after sorting: [[1, 1], [2, 1], [1, 2], [2, 2], [3, 1], [3, 3]]

List before sorting: [[1, 1], [3, 3], [2, 1], [2, 2], [1, 2], [3, 1]]
List after sorting: [[1, 1], [1, 2], [2, 1], [2, 2], [3, 1], [3, 3]]

Comb Sort Pseudocode

You can use the following pseudocode as a starting point for your solution.

```
combsort(input):
    Set gap to be the length of the list
    Set shrink to 1.3
    Set sorted to False

    while sorted is False:
        Set gap to floor(gap / shrink) (i.e., round down or convert to integer)
        if gap > 1:
            Set sorted to false
        else:
            Set gap to 1
            Set sorted to true

        Set i to 0
        while i + gap < length of list:
            if input[i] > input[i+gap]:
                swap input[i] and input[i+gap]
                Set sorted to false
            Set i to i + 1
```

Recap

Your zip file should contain your **factorial.py**, **analysis.py**, **count.py**, and **comb.py** files.
Submit your **assignment5.zip** file to cuLearn.
Make sure you download the zip after submitting and verify the file contents.
