# Assignment 5

Dictionaries, searching, and recursion

Submit a single zip file called **assignment5.zip**.
This assignment has 30 marks.
See the marking scheme that is posted on the course webpage.

## Problem 1 (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 need to solve the remainder of the problem. 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 slow process. This function should also remove any information stored from a previous file when it is called (i.e., you start from nothing 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 2 (Recursive Factorial)

Create a Python file called **factorial.py** that a single function called **cachedfactorial**. The cachedfactorial function will accept a single integer argument and must recursively calculate the factorial value. Additionally, this function must use/update a 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 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 want.

---

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

---