ECE 241: Data Structures and Algorithms- Fall 2022

Project 1: Fun with Words

Due Date: **Deadline: see website, class policy and moodle for submission**

Description

This project is intended to familiarize you with operations on unsorted and sorted lists such as Simple Sorting and Searching Algorithms. The goal of the project is to design an efficient dictionary database and implement a variety of useful applications such as: searching a word, spell checking a text, finding anagrams of a word, creating a scrabble helper tool, and cracking a code locker. The project comes with multiple **unsorted** dictionaries:

1. a short dictionary `short.txt` of only 20 words (good for testing/debugging)
2. a large dictionary `english.txt` that contains 99,171 english words (for production).
3. a large dictionary `french.txt` that contains 139,719 french words (for testing).
4. a large dictionary `spanish.txt` that contains 86,017 spanish words (for testing).

The project also includes multiple 'source' and text files:

1. `app1.py` to `app6.py`: six application files used for testing (provided). **You are not allowed to modify them**.
2. `Dictionary.py` file containing the class object Dictionary (to complete).
3. Several sample text files (poem or excerpt) in english, french and spanish: `letter.txt`, `sample_english.txt`, `sample_french.txt`, `sample_spanish.txt`.

All the functionalities of the application files (presented below) should be successfully implemented to obtain full credit. You need to proceed step-by-step, application by application. A list of methods to implement is described at the end of each section. These (main) methods are required to make sure that the application run smoothly. You are welcome to implement any other (sub-)methods if needed.

Submission/Grading Proposal

Only **one file** `Dictionary.py` must be submitted on moodle. Only one submission by group of two. Write your name on top of the file. This project will be graded out of 100 points:

1. Your program should implement all basic functionality/Tasks and run correctly (90 pts).
2. Overall programming style with comments (including function header doc-string) (5 pts).
3. Pre-submission deadline for Preliminary (answering first question) (5pts). The program does not have to run correctly for pre-submission.

Preliminary [30pts]

The `Dictionary.py` file contains a main method which is provided to you and will help you implementing few methods for this class.

At the first execution, the code will be asking to enter a dictionary file name (from `'name'.txt`). if you enter something other than `short`, `english`, `french` or `spanish`, you should get:

```
Enter dictionary name (from file 'name'.txt): portuguese
File portuguese.txt does not exist!
```

In contrast, if you enter `short`, you should get:

```
Enter dictionary name (from file 'name'.txt): short
Load short.txt
Name first dictionary: short
Size first dictionary: 20
Five random words: tea cases file morning of

Name second dictionary: N/A
Display second dictionary:
tea
cases
file
morning
of

Second dictionary shuffled in 3.674999999999859e-05s:
Display second dictionary:
morning
tea
of
file
cases

Linear search for the word 'morning' in second dictionary
Is 'morning' found: True at index 0

Second dictionary sorted in 2.206299999999925e-05s:
Display second dictionary:
cases
file
morning
of
tea

Binary search for the word 'morning' in second dictionary
Is 'morning' found: True at index 2

Binary search for the word 'ning' in second dictionary
'ning' is not found so it must be inserted at index 3
```

Another example with `english`:

```
Enter dictionary name (from file 'name'.txt): english
Load english.txt
Name first dictionary: english
Size first dictionary: 99171
Five random words: skyjacking professionally ripped corrective dependability's

Name second dictionary: N/A
Display second dictionary:
skyjacking
professionally
ripped
corrective
dependability's

Second dictionary shuffled in 8.56400000000257e-06s:
Display second dictionary:
corrective
skyjacking
dependability's
ripped
professionally

Linear search for the word 'morning' in second dictionary
Is 'morning' found: False at index -1

Second dictionary sorted in 4.438999999994975e-06s:
Display second dictionary:
corrective
dependability's
professionally
ripped
skyjacking

Binary search for the word 'morning' in second dictionary
Is 'morning' found: False at index 2

Binary search for the word 'ning' in second dictionary
'ning' is not found so it must be inserted at index 2
```

How to proceed? You need to comment/uncomment the instructions in the `main` method in order to test step by step with the following implementation:

Step-1 Implement the constructor `__init__` method. You will first initialize an empty list of words. This list should be **private**. You need to read the dictionary from file and fill up your list. Make sure the file exists (you can use the `try`/`except` instructions). Also, the module `sys` allows you to use the instruction `sys.exit(0)` that will end the code execution.

Step-2 Implement the methods: `get_name`, `get_size`, `get_random_list`. The name of the dictionary is the name of the file you used to load it. You will use a new `private` instance variable in your constructor to store this name. In addition, the `random` seed will need to be set to **8** in the constructor (to reproduce all the results shown above, and throughout this project).

Step-3 You need to modify your constructor so it can instantiate an empty dictionary. The default name is `N/A`.

Step-4 Implement the method `insert` to append new words in the 'unsorted' dictionary, as well as the method `display` that will display all the words in the dictionary.

Step-5 Implement the `shuffle` method using the Fisher-Yates algorithm seen in class.

Step-6 Implement the `lsearch` method that is using a linear search to search a word in the dictionary, and return True/False if the word is found/not found. You will also introduce a new `private` instance variable to keep track of the index number (where the word is found).

Step-7 Implement the `bsearch` method that is using a binary search to search a word in a dictionary which is assumed to be sorted. If a word is not found, the index number should represent the position where the word should be inserted (but you do not insert it). Hint: this index is the lower bound in the binary search algo.

1 app1.py- 15pts

The goal of this app is to sort the dictionary using two algorithms: insertion sort and enhanced insertion sort. The new sorted dictionary will then be saved on file.

Example of execution:

```
Dictionaries: short english french spanish
Choose your dictionary: short
Load short.txt
**Dictionary short.txt contains 20 words

1-insertion
2-enhanced insertion sort
Choose your sorting algorithm: 1
**Dictionary short.txt shuffled in 0.0001378590000000006 seconds

**Dictionary short.txt sorted with -insertion- in 8.177100000000131e-05 seconds

Save short_sorted.txt
```

The file `short_sorted.txt` looks like:

```
act
animal
ate
cases
cat
class
code
computer
dictionary
eat
electrical
file
morning
of
school
screen
simon
sorting
tea
this
```

Another example (takes a long time):

```
Dictionaries: short english french spanish
Choose your dictionary: english
Load english.txt
**Dictionary english.txt contains 99171 words

1-insertion
2-enhanced insertion sort
Choose your sorting algorithm: 2
**Dictionary english.txt shuffled in 0.094823236 seconds

**Dictionary english.txt sorted with -enhanced insertion- in 198.322120023 second

Save english_sorted.txt
```

As you can see, shuffling is fast but simple sorting is rather slow.
How to proceed?

- Implement the `insertion_sort` and `enhanced_insertion_sort` methods. They both return the time it takes to perform the sorting.
- Implement the `save` method that will write your sorted dictionary into a new file. You need to generate all those files `'name'_sorted.txt` before proceeding to some of the next apps (like app2, app3 and app6).

2 app2.py- 5pts

Here you will search the sorted dictionary. Example:

```
Dictionaries: short english french spanish
Choose your *sorted* dictionary: french
Load french_sorted.txt
**Dictionary french_sorted.txt contains 139719 words

How many random words would you like to search? 3000
first few random words:
féodal
prémunissions
pâti
discuterions
entrelacements
Search time is: 0.013156071999999998, with 16 average number of steps
```

How to proceed? Since you already implemented the methods `bsearch` and `get_random_list` before, everything should work fine. You just need to include a new **private** instance variable to account for the number of steps needed in binary search.

3 app3.py- 10pts

Using the sorted dictionary, this option allows the user to check the spelling of all the words in any text files. Four example text files are included for testing: (i) letter.txt (a letter from me to you in english),(ii) sample_english.txt (to use with the english dictionary), (iii) sample_french.txt (to use with the french dictionary), (iv) sample_spanish.txt (to use with the spanish dictionary).

At the execution, the user is asked also to enter the name of the text file. The code will check the spelling of all the words (line by line and along the words of a given line). It will return the text on screen. If a word is not found in the dictionary, it will flag it as incorrectly spelled and return it into brackets '(' ')'.

Example using the english dictionary:

```
Dictionaries: english french spanish
Choose your *sorted* dictionary: english
Load english_sorted.txt
Enter file name to spell check: letter

Dear (Studants,)
this is a sample file similar to the one that will
be used to test your projects. I (sugest) you use it
for testing your code and (developping) your software.
Have fun!
E. (Polizzi)
```

Example using the french dictionary:

```
Dictionaries: english french spanish
Choose your *sorted* dictionary: french
Load french_sorted.txt
Enter file name to spell check: sample_french

Extrait de (20,000) lieues sous les mers
De (Jules) (Verne)


Le (Nautilus) était alors revenu à la surface des flots.
Un des marins, placé sur les derniers échelons, dévissait les boulons du panneau.
Mais les écrous étaient à peine dégagés, que le panneau se releva avec une violence extrême,
évidemment tiré par la ventouse (d'un) bras de poulpe.

Aussitôt un de ces longs bras se glissa comme un serpent par (l'ouverture,)
et vingt autres (s'agitèrent) au-dessus. (D'un) coup de hache, le capitaine (Nemo) coupa ce formidable tentacule,
qui glissa sur les échelons en se tordant.
```

Make sure that the sample file exists!:

```
Dictionaries: english french spanish
Choose your *sorted* dictionary: english
Load english_sorted.txt
```

```
Enter file name to spell check: poem
File poem.txt does not exist!
```

How to proceed? write the `spell_check` method that is going to load the file, scan the text line by line, then word by word which you would need to spell check before printing on screen. Hint: before searching the dictionary using binary search, each word would have to be transfomed into lowercase (to help the search), and the punctuation at the beginning and the end of this word will have to be removed. For my punctuation string I am using:

```
punc="’!()-[]{};:’"\,<>./?@#$%^&*_~’"
```

you can take advantage of the `lstrip` and `rstrip` methods for string to remove the punctuation. Depending on the result of your search you will have to display on screen the original word with or without parenthesis around.

4 app4.py- 10pts

This app allows the user to enter a key word or a set of character/letters, and the code is supposed to return all the words that can be formed using permutations of the letters (looking at your dictionary of course).

Since permutation algorithms are actually difficult to implement with long runtime, we will be using the following ’simple’ solution instead:

(i) sort the characters in your key word using a ’selection sort’ algorithm for example (e.g. the word ’dbca’ will give ’abcd’);

(ii) scan linearly through all the dictionary for words that have the same String length than your key word,

(iii) if found, sort similarly the corresponding word by characters and compare with your sorted key word

(iv) if the words are identical, you have found an anagram.

Here is an example of execution:

```
Dictionaries: short english french spanish
Choose your dictionary: short
Load short.txt
Enter word to analyze: tae

3 anagram(s) found
ate
tea
eat
```

Another example (if you are unlucky):

```
Dictionaries: short english french spanish
Choose your dictionary: short
Load short.txt
Enter word to analyze: unlucky

0 anagram(s) found
```

and another:

```
Dictionaries: short english french spanish
Choose your dictionary: english
Load english.txt
Enter word to analyze: opst

6 anagram(s) found
stop
opts
post
tops
pots
spot
```

How to proceed? write the `anagram` method that returns the list of anagrams. You will need to implement a helper static `sort_word` method that takes a word as input and return the same word with letter reordered (e.g. the word ’dbca’ will give ’abcd’).You can use any sorting algorithms (that we have seen) to sort those letters. The `sort_word` method method will be useful to call within the `anagram` method.

5 app5.py- 10pts

This app is an application of app4 to create a scrabble helper.

At execution, you need to enter a key word (set of character/letters), the code will generate all possible combinations (code is provided.. note that letter combinations are easier to obtain than all permutations). For example ’rbg’ has for combinations: ’r,rb,rbg,b,bg,g,rg’. For each combinations you could search for all possible anagrams (of different length). The code is converting the final list into a new dictionary. The code will compute the scrabble score of all these words, and rank them from low to high score, before displaying both the words and their score.

```
Dictionaries: short english french spanish
Choose your dictionary: english
Load english.txt
Enter series of letter: english
List of letter combinations:  [’’, ’e’, ’n’, ’en’, ’g’, ’eg’, ’ng’, ’eng’, ’l’, ’el’, ’nl’, ’enl’, ’gl’, ’egl’, ’
Number of non-zero letter combinations found:  127

61 anagram(s) found  with scrabble score
e 1
n 1
```

```
l 1
i 1
s 1
g 2
in 2
es 2
ls 2
is 2
lei 3
lie 3
nil 3
gs 3
sin 3
ins 3
leg 4
gel 4
gin 4
lien 4
line 4
lens 4
sine 4
lies 4
isle 4
leis 4
h 4
glen 5
legs 5
gels 5
egis 5
sing 5
sign 5
gins 5
liens 5
lines 5
eh 5
he 5
hi 5
sh 5
glens 6
singe 6
sling 6
hen 6
hie 6
hes 6
she 6
his 6
single 7
hens 7
hies 7
shin 7
nigh 8
shine 8
sigh 8
hinge 9
neigh 9
hinges 10
neighs 10
sleigh 10
shingle 11
```

How to proceed?

- Implement the `compute_score_scrabble` method in `Dictionary`. You will need to introduce a new **private** instance variable being a list of int, that will be used to record the score of each word in your anagram dictionary. You can count the word score by adding the score of each individual letters in a word with the following weights: 1pt for e,a,i,n,r,t,l,s,u; 2pts for d,g; 3pts for b,c,m,p; 4pts for f,h,v,w,y; 5pts for k; 8pts for j,x; 10pts for q,z.

- Implement the `score_sort` method that will sort the Dictionary words and their score by increasing order of scores. You could use any of the sorting algo we have seen. Hint: if you perform a shift or swap on the score list, you need to do the same shift or swap on the word list.
- Modify the `display` method that is using an optional argument to display the score. Make sure your changes are backward compatible with all the previous apps and the Dictionary main method.

6 `app6.py`- 10pts

Last app, finally! We want to make use of the sorted dictionary to crack the word code of a lock (see Figure 1, for example). Let us suppose we have a lock composed of 3 letters with the following two options by letters:
(c,e) for letter 1
(a,p) for letter 2
(t,i) for letter 3
We could form $2^3 = 8$ combinations (number of options to the power the number of letters). This could increase significantly if we add more letters or more options, for example 5 letters and 8 options gives $8^5 = 32768$. However, most of locking systems are using words that can be found in the dictionary (easy to remember for the user of the lock), so we could take advantage of this in app.

Figure 1: Combination padlock

Example of execution using the 'short' dictionary:

```
Dictionaries: short english french spanish
Choose your *sorted* dictionary: short
Load short_sorted.txt
```

```
How many code letters to crack?: 3
Enter all the options for letter 1: c e
Enter all the options for letter 2: a p
Enter all the options for letter 3: t i
All lock combinations to consider:
['c', 'e']
['a', 'p']
['t', 'i']

2 word(s) found:
cat
eat
```

How to proceed? Implement the method `crack_lock` that returns a new dictionary containing all the possible words. In theory, we will need to consider to consider $c = (\text{nboption})^{\text{nbletter}}$ locking combination options, search them in the sorted dictionary, and return all the words found. Since it can be quite complicated to generate all these locking combinations (without using recursion), you will be using instead a stochastic approach where your code will generate $6 * c$ random locking combinations of letters (for example: 1st letter chosen at random among all its possible options, etc.). This technique may not be able to find all the acceptable words at the end but it generally works well if the number of random locking combinations is large enough. In addition, as soon as a word is found (after a successful binary search) you will insert it into a new dictionary *only* if it is not already present (to avoid duplicates if you get the same random locking combination). Hint: you may want to use the linear search method `lsearch` to search the new dictionary (which is unsorted). As shown in the example above, your new dictionary is then sorted and printed.

Experiment: use the dictionary 'english.txt' and the following combination of 6 for 4 letters: (q, e,t,u,l,s); (r, u,f, l,o,q); (k,m,s,a,r,w);(w,x,s,o,p,g);
You should find 21 words. This is my own locker (for my bike;-), I forgot the combination once...so this app was pretty useful.