Assessed Exercise 1

This assessed exercise will be marked out of a toal of 60 marks, and accounts for 10% of the course total. The submission deadline would be noted on the course moodle.

Please note that:

1. **You can choose to complete this project on your own, or in a group of maximum size 2. In case of working as a group, the code submitted will be assumed to be the same for both and marked once, but each of the two students *must submit separate reports* that they write individually, on their own (though you can certainly consult one another when writing it).**

2. **You must write your own code. Some skeleton files have been provided to help you get started, but you can choose to ignore them. You are free to refer solutions to problems provided in this course, as well as any online resources or forums, but you should then write your own code (that is: no copy-pasting please).**

3. **In a similar vein, feel free to work alongside and discuss the project classmates outside your group, but what you write and submit must have been done entirely by you/your group members.**

4. **In general, you should limit your code to using the barebones Python, without using any libraries (other than some obvious ones like time, random, sys). If you have any doubts about use of a particular library, then please ask the instructor.**

5. **You are free to use Python's built-in data types like lists and arrays, as well as Python's built-in "sorted" function.**

6. **You can use Generative AI as you would a tutor or a peer; to learn, recall, or understand relevant syntax, or to explain something noted in this handout. You are *not* allowed to ask it to *solve* the problem for you. Hopefully this distinction is clear enough, but reach out to your instructor or tutor for any more specific queries. Use of generative AI is not allowed at all for writing the report. It is important to ensure that Academic Integrity is maintained, see here for further guidance.**

Overview of Project

The purpose of this project is to design a small search engine that can search all *text* files in a given directory for text query entered by the user.

The query will be entered in the form of a phrase which may be composed of multiple words. When a user enters a phrase, only documents which have *each* of the words in the phrase (in any sequence) should be returned, ordered by the *document rank* as explained later. (You are not asked to create more advanced search features like matching of exact phrase if query given in quotes, use of logical operators like +, and, etc.). The search engine should *not* be case sensitive.

Multiple approaches can be taken in constructing such a search engine. You are being asked to take a specific approach explained as follow. This approach is a simplified version of the algorithm given in [2], which in turn is based on the original paper for Google search engine [3].

To complete this project, you will be required to implement three main tasks: Crawling, Indexing, and Searching.

Crawling

When the search engine is run, it will first *crawl* all text files in the target folder, which basically means reading them into memory from the external storage (file) one by one, and *indexing* them.

Indexing

To enable the search to return quickly (that is, you don't want every new search query to search sequentially through all the files), you will create an *index*. The details are described as follows. While you are reading in the files to create the indices though, you will have to be careful about cleaning up the text to remove special characters and white spaces, and also deal with case sensitivity (recall, the search engine is meant to be case *in*sensitive).

To create a complete index of your target search folder, you need to create four variables. You should use appropriate data structures to store each of these four crucial variables.

i. **Forward Index**. This index associates all documents with the associated words. E.g.

```
anna_karenina.txt: 'the', 'project', 'gutenberg', 'ebook', …
second.txt: 'why', 'said', 'the', 'old', …
…
war_and_peace.txt: 'the', 'project', 'gutenberg', 'ebook', …
```

ii. **Inverted Index**. This is the reverse of the previous index, listing all words with associated documents

```
the: 'anna_karenina.txt', 'divine_comedy.txt', …
project:'anna_karenina.txt', 'divine_comedy.txt', 'little_women.txt',…
…
immovability:'war_and_peace.txt'
```

iii. **Term frequency (TF)**: Stored for each unique word *in a given document*. It's value will be between 0 and 1, and is calculated as follows:

- $TF(word) = \frac{Number\ of\ occurences\ of\ word\ in\ a\ document}{Total\ number\ of\ words\ in\ the\ document}$

You can think of this as an indication of how frequently a word appears in a given document. E.g.:

```
anna_karenina.txt:
      'the'  0.05015589569160998,
      'project': 0.00025793650793650796,
      …
second.txt:
    'why': 0.0065040650406504065,
    'said': 0.013008130081300813,
    'the': 0.055284552845528454,
     …
```

iv. **Inverse document frequency (IDF):** This is a metric calculated for each unique word encountered, *across all documents*. It has values between 0 and 1, and is calculated as follows:

- $IDF(word) = \frac{Number\ of\ documents\ with\ word}{Total\ numnber\ of\ documents}$

E.g.
```
the:1.0
project:0.6923076923076923
gutenberg:0.6923076923076923
ebook:0.6923076923076923
of:1.0
…
```

You can think of the IDF parameter as indicator of the uniqueness of a given word across the search documents. The lower this value, the rarer this word. If this value is 0 for a given word, that means that word is not present in any of the given documents. A value of 1 means it is present in all documents.

Document rank (DR)

To show the result of our search in a particular order, we use metrics like term frequency (TF) and inverse document frequency (IDF) for each of the words in a given query. In addition however, we also use a metric that shows the *significance* of each document, independent of the specific query. If this were a web search engine, then we would be using something like Google's *page ranking* algorithm to calculate this "PageRank" metric. We will keep things simple here though, and use the size of the file, such that *the smaller the file, the higher its signficance*. This the *DocumentRank*, calculated as follows:

$$DocumentRank(document) = \frac{1}{Total\ number\ of\ words\ in\ the\ document}$$

The intuition here is that if our search phrase is in a smaller file, its "presence" is relatively larger. You will need to calculate this rank for each document, and could do so while you are indexing them. This value, along with TF and IDF for the words in the search query, will together be used to calculate the "weight" which you will then use to sort your search results (exact formula for this *weight* follows).

Output files

All 5 data structures calculated during the indexing and document ranking process (forward index, inverse index, TF, IDF, DR) should be stored in the form of text files. This will enable us to re-use these values in subsequent runs (though you are not asked to implement this re-use feature in this project). The provided main.py script takes care of creating these output files for you.

Searching

The interesting work has happened when you have indexed the files and ranked them. The search is now simply a matter of looking up the data structures you have created in the process.

User will enter their search phrase on the command line, and your program should display an list of all files which contain that search query, *ordered* based on the values of TF, IDF and DR. There are many ways to use these metrics to order the search results, and we use a simple approach as follows:

For every document, you can take the product of TF and IDF for each term of the query, and calculate their cumulative product. Then you multiply this value with that documents DocumentRank to arrive at a final *weight* for a given query, for every document. This *weight* is then used to sort your results.

E.g., say the search query is: sunny place
The weight of a given document for this given query would be:

$$\big(TF(\text{sunny}) \times IDF(sunny)\big) \times \big(TF(place) \times IDF(place)\big) \times DocumentRank$$

What you are provided

You will be provided with:

- The assessment handout (this document).
- A zipped project folder structure as follows:

```
pa_ae1_skeleton
    /output_files           #to store result of indexing operations
    /search_dir             #contains the 13 text files to be searched
        anna_karenina.txt
        divine_comedy.txt
        …
        war_and_peace.txt
    main.py                 #search engine is launched via this file
    pa_search_engine.py     #this is where your implementation of the
                            #search engine will go
    REPORT.md               #Notes about your implementation in markdown format
    tester.py               #tester script
```

- Note that `main.py` and `tester.py` files are provided complete, and should not be changed. The only file where you add your code is `pa_search_engine.py.`
- You should populate the markdown file README.md with notes about your implementation. Specifically, you should answer questions noted later.

How to check your solution

If your search engine is correctly implemented, you should see something like this when you run the provided **main.py** file:

```
Crawling initiated
Time taken to index file: anna_karenina.txt = 0.7192317999999887
Time taken to index file: divine_comedy.txt = 0.13523480000003474
Time taken to index file: first.txt = 0.016181299999971088
Time taken to index file: first_simple.txt = 0.010622399999988374
Time taken to index file: little_women.txt = 0.3772414000000026
Time taken to index file: moby_dick.txt = 0.3876495999999747
Time taken to index file: oliver_twist.txt = 0.373533899999984
Time taken to index file: room_with_a_view.txt = 0.1618550999999684
Time taken to index file: second.txt = 0.011778899999967507
Time taken to index file: shakespeare.txt = 1.7350040000000035
Time taken to index file: sherlock.txt = 0.2828157999999803
Time taken to index file: third.txt = 0.012955400000009831
Time taken to index file: war_and_peace.txt = 0.9527746999999636
Crawling concluded in 5.205326500000012 seconds
Writing indices to files
Search engine ready for queries, target directory = search_dir

Enter your search term: test query
# Search Results:
little_women.txt
war_and_peace.txt
2 results returned
# Search took 9.959999999864522e-05 seconds


Enter your search term:
```

Your program should also pass the tests provided in the **tester.py** file.

What you should submit

Once you have completed your project work, you should submit a zipped form of your solution folder on moodle. The folder's structure and contents should be exactly as described earlier. There are three main components to the submission: the code, the report, and the video walk through.

1. **The Code:** You have been provided a skeleton code already. As discussed earlier, the `main.py` and `tester.py` files are already complete, and should not be modified. All your code will go in the module pa_search_engine, contained in the file `pa_search_engine.py`. This part of the submission can identical for both students if you did this project as a group of two.
2. **The Report:** In the provided REPORT.md file template (Markdown format), please address the questions noted below. You must do this individually even if you did your project as a group.
    a) **Complexity Analysis:** You should present the following complexity analysis:
    - Big-O complexity of the indexing operation.
    - Big-O complexity of a search operation (given that indexing has been done already).
    - Big-O complexity of a search operation if it were implemented in a brute force fashion (that is, no indexing performed, all search queries go through the entire text of all files every time).
        - You should be very clear about what you mean by *n* when presenting your Big-O complexity analysis.

    Note: You don't have to do a line-by-line code-analysis like we do in certain other problems. Instead, present your analysis as a text description that walks through the relevant operations, comments on their complexity with rationale, and then presents the overall complexity.

    b) **Choice of Data Structures:** In the provided REPORT.md file template (Markdown format), explain and justify your *choice of data structures*.

    c) **Extra Feature(s):** If you implemented any extra feature on top of the requirements noted in this hanadout, briefly describe them here.

How this will be assessed

You will be assessed as follows:

- Functional correctness – 30 marks
- Analysis of the efficiency of algorithms – 15 marks
- Choice of and discussion related to data structures used – 10 marks
- Any interesting additional feature(s) – 5 marks
    - Please make sure any additional feartures do not "break" your code; that is, it should be remain functionally correct as tested by the provided testing script.

References

[1] Jason King. 2021. Combining Theory and Practice in Data Structures & Algorithms Course Projects: An Experience Report. In Proceedings of the 52nd ACM Technical Symposium on Computer Science Education (SIGCSE '21). Association for Computing Machinery, New York, NY, USA, 959–965. https://doi.org/10.1145/3408877.3432476

[2] David Yastremsky, Building Your Own Search Engine From Scratch, 2021. https://blog.devgenius.io/building-your-own-search-engine-from-scratch-e542a1068c44

[3] Brin, Sergey, and Lawrence Page. "The anatomy of a large-scale hypertextual web search engine." Computer networks and ISDN systems 30.1-7 (1998): 107-117.
Available: https://storage.googleapis.com/pub-tools-public-publication-data/pdf/334.pdf