Contents

Program Overview

The solution is developed in the single-file program **`solver.py`**.

- The language version and the development environment are:
  `Python 3.8`, **`Ubuntu 20.04 LTS (Focal Fossa)`**
- Only modules from the Python Standard Library are used (*i.e.*, no third-party libraries):
  `typing`, **`string`**, **`argparse`**, **`os.path`**, **`pickle`**
- The program is organized via 6 Python functions, the **`main()`** function and 5 auxiliaries:
  `generate_lookup`, **`get_results`**, **`parse_args`**, **`print_results`**, **`vectorize_word`**
- See the document **`docs_solver.pdf`** for the documentation generated via **`pydoc3.8`** using embedded docstrings and comments.
- Type annotations are used in the entirety of the program, and static analysis is performed using **`mypy`** (v0.961).
- The source code is extensively documented using docstrings and embedded comments.

Running the Solver

The solver can be run by the user using either of two ways.

- Via the shell (*e.g.*, **`bash`**, **`zsh`**, **`fish`**, …):
  `$ ./solver.py input`
  Note: Requires execution permissions by running **`chmod +x ./solver.py`**
- Via the Python interpreter:
  `$ python3 ./solver.py input`

An optional argument can be provided (when omitted, it defaults to **`corncob_lowercase.txt`**):
`$ ./solver.py input -l wordlist_path`

Comments on Program Structure

The implementation could be restructured into a class-based solution (or an object-oriented one), where several of the function parameters can be eliminated, since these variables can be accessed as object private members. For brief illustration:

```
class Solver:
    __input: str
    __wordlist: str
    ...
```

Refactoring the solution as a class would also allow instantiating new **`Solver`** objects for multiple wordlists. However, I chose to **not** unnecessarily complicate the program structure since the application requires only an input/output mapping ($\text{Input} \mapsto \text{Subanagrams}$) using a *single* word list (defaulted to **`corncob_lowercase.txt`**).

Due to the procedural nature of the problem (and its provided solution), I preferred preserving the current procedure-based solution that is implemented across six Python functions. In my opinion, the additional abstraction is unnecessary, so I committed to the simpler structure.

Complexity Analysis: Algorithmic Performance

The provided algorithm solves the search problem using a *lookup table*, implemented as *Python dictionaries*. Its space and time complexities for each input type is summarized in the table:

	Against **Input Word** *Size*	*Against* **Word List** *Size*
Space Complexity	Constant $\mathcal{O}(1)$	Linear $\mathcal{O}(n)$
Time Complexity	Constant $\mathcal{O}(1)$	Linear $\mathcal{O}(n)$

Detailed Analysis

Lookup tables typically have $\mathcal{O}(n)$ space complexity against the information for which they are 'hashing' and are a typical example of space-time tradeoff. Similarly, the algorithm's space complexity against the word list text file as input is also *linear*.

Lookup search scenario: (Covers the average run of the algorithm without lookup generation)

Refer to the function,

```
get_results(search_input: str) -> Set[str]:
```

Refer to the **for** loop in line 94:

```
94    for candidate in list( anagrams.keys() ):
```

The number of candidate keys to iterate over will *grow with* the size of the word list (on average, proportional) but *is independent* of the size of the input word.

Using a word list twice as large will produce a lookup file which consequently requires twice the resources to process. The *space* and *time* complexity with respect to word list text file as input is thus **linear** $\mathcal{O}(n)$. But regardless of the input word, the resources used are invariable. This implies constant $\mathcal{O}(1)$ *time and space* complexity with respect to the *user's search input word*.

Lookup generation scenario: (Covers the initial run of the algorithm to generate the lookup)

Refer to the function,

```
generate_lookup(wordlist_path: str) -> None
```

Its **while** loop (lines 49 to 60 below) creates the dictionaries and writes them in the lookup file. The number of iterations in the loop is **directly proportional** to the word list's number of lines (and therefore words). This yields the same conclusions: the time, memory, and storage costs are linear with respect to processing this word list.

```
46    # Read (from the word list) file line by line processing all contained words.
47    with open(wordlist_path, 'r', encoding='UTF-8') as wordlist_file:
48        word: str
49        while ( word := wordlist_file.readline().rstrip() ):
50            # Sort (letter-wise) each word and store in its `sorted_word` anagram.
51            sorted_word: str = ''.join( sorted(word) )
52
53            if sorted_word not in anagrams:
54                # Create a new anagram set for an unencountered `sorted_word` key.
55                anagrams[sorted_word] = set()
```

```
56                # Associate each `sorted_word` anagram key with its letter count.
57                vectors[sorted_word] = vectorize_word(sorted_word)
58
59            # Associate each anagram word with its `sorted_word` anagram key.
60            anagrams[sorted_word].add(word)
```

Motivations & Implications

A realistic application for this algorithm and implementation is the backend of a webserver. A web app would provide users with an interface to request subanagrams. Example include "Scrabble dictionaries" such as: **wordfinder.yourdictionary.com**.

On such a web app, the lookup table is going to be generated once, and updated infrequently as new/more words are hashed. However, users will be requesting subanagrams at high frequency.

The implications for our algorithm are:

- Constant time and space costs $\mathcal{O}(1)$ against user input. Whether the user inputs **car** or **albhfsdkjabhwekljrhakejr**, the runtime and memory costs are on average the same.
- Linear time and space costs $\mathcal{O}(n)$ against the word list. The server would proportionally require more processing time, memory, and disk space as this list grows.

Thus, the choice of a lookup table is adequate, from an algorithmic point of view.

Implementation Performance Benchmarking

Algorithmic complexity informs only about *cost growth* against inputs, but does not inform about the *cost values*. For this, implementation benchmarks for resource costs are added. The benchmarking computer's relevant hardware information is tabulated below:

PROCESSOR INFO	MEMORY INFO	STORAGE INFO
Intel® Core™ i3-8130U CPU **Base speed:** 2.21 GHz **Sockets:** 1 **Cores:** 2 **Logical processors:** 4 **Virtualization:** Enabled **L1 cache:** 128 KB **L2 cache:** 512 KB **L3 cache:** 4.0 MB	**Memory:** 8.0 GB **Speed:** 2400 MHz **Slots used:** 1 of 2 **Form factor:** SODIMM	*WDC PC SN520 SDAPNUW-256G-1006* **Type:** SSD **Read speed:** 164 KB/s **Write speed:** 94.0 KB/s **Average response time:** 0.6 ms

The benchmark can be performed with the command:
$ /usr/bin/time --verbose python3 ./solver.py input

Note, this is different than the **bash** default **time** utility. It comes preinstalled on Ubuntu 20.04, but must be explicitly referred to using the path **/usr/bin/time** to distinguish it from the default.

For example, **$ /usr/bin/time --verbose ./solver.py akjjasdfkjhaer** yields the following results (less relevant details have been omitted, run the command if interested):

```
        User time (seconds): 0.18
        System time (seconds): 0.05
```

```
        Percent of CPU this job got: 99%
        Elapsed (wall clock) time (h:mm:ss or m:ss): 0:00.24
        Maximum resident set size (kbytes): 59276
```

Costs for this run on the input **akjjasdfkjhaer**: 0.18 seconds and 59,276 KB of memory.

Similarly, **$ /usr/bin/time --verbose ./solver.py dog** yields the following results (again, less relevant details have been omitted):

```
        User time (seconds): 0.20
        System time (seconds): 0.02
        Percent of CPU this job got: 99%
        Elapsed (wall clock) time (h:mm:ss or m:ss): 0:00.23
        Maximum resident set size (kbytes): 59404
```

Costs for this run on the input **dog**: 0.20 seconds and 59,404 KB of memory.

After running this benchmark 20 times (twice, for 10 different inputs of different sizes), the following is observed:

Average size of the lookup file on disk (this does not change, in fact): **4.57 MB**

***Typical run*:** (lookup dictionaries previously generated, no extra resources spent generation)

- Average memory usage (resident set size): **59 MB**
- Approximate runtime (user time): **0.2 seconds**

***Initial run*:** (lookup dictionaries unavailable, extra resources spent generating them)

- Average memory usage (resident set size): **67 MB**
- Average runtime (user time): **0.6 seconds**

Obviously, on a system with more/less powerful hardware, these results will vary. The performance for casual use of the program is adequate. However, for high performance use (such as on an optimized web service), further improvements can be made (suggested below).

Potential Improvements

Time, memory, and storage costs can be further reduced by attempting the following ideas:

- Using **numpy** or other high-performance libraries for containers and their processing.
  (I did not want to use modules outside the Python 3 Standard Library, so I had to make sacrifices)
- Researching and implementing more efficient read/write procedures.
  (In effect, this is a large contributor in terms of cost)
- Finding an adequate storage format (*e.g.*, JSON, Pickle's serialization format) for a good tradeoff between storage size/compression and retrieval time/speed.

Static Analysis & Type Annotations

Type annotations are used in the entirety of the program. In Python 3.9, type annotations of containers and their elements can be added without importing modules. Since I used Python 3.8, (and for the added benefit of better compatibility with older versions of Python), the module **`typing`** has been used for annotating containers. Performing static analysis using **`mypy`** (v0.961):

> Command used: **`$ mypy solver.py`**
> Output: **`Success: no issues found in 1 source file`**

Mathematical Comment

The functions **`generate_lookup`** and **`get_results`** are designed with intrinsic optimizations. For example, words form the word list are associated to their sorted strings. For instance, **`"state"`** and **`"taste"`** have the same key **`"aestt"`**. This means all anagrams are related by having the same key, which is their common sorted strings. By doing this, there are far less keys than there are words in the word list. Effectively, this yields large cost reductions and thus performance boosts for obtaining the search results. The vectorization occurs over these sorted string keys as well, meaning that anagram words are mapped to the same vector (of letter counts).

Mathematically speaking, the solution exploits the non-injectivity of the $\mathrm{Words} \to \mathrm{Key}$ map (since there are several word anagrams for the same sorted key). This comes with the drastic set cardinality reduction from the domain to the codomain: $\mathrm{card}(\mathrm{Words}) \gg \mathrm{card}(\mathrm{Keys})$.

Abandoned Ideas

Improving performance using better combinatorics: **failed**

> I experimented with improving performance using specially-designed alphabetical permutations, rather than searching through all the keys in the aforementioned **`for`** loop at line 94. After trying several permutations patterns using the **`itertools`** module and deriving their complexities, I decided to abandon the idea. The complexity is high due to combinatorial explosions. Factorial complexity with input size is a common scenario.

Handling of user input: **skipped**

- Providing the user with additional help and error messages.
- Skipping/exiting useless runs when the input contains numbers, whitespace, or symbols.
- I elected to skip this feature as it's trivial to implement and not central to the challenge.

Preparing a virtual environment using **`venv`** and exporting a **`requirements.txt`**: **skipped**

- It's a good practice, but since I used modules exclusively from the Python Standard Library, the benefits of using a virtual environment are slim.

Writing unit tests using the Standard Library module **`unittest`**: **skipped**

- It's a great practice, but the application and the program are small and thus can be checked manually with little inconvenience.
- Although, for unit tests, it would have been interesting to test against special cases, *e.g.*:

> Multiple letter appearances such as **`e`** in **`fever`**
> Consecutive letter appearances such as **`e`** in **`free`**
> Among others (*e.g.*, large words, nonsensical words, maybe palindromes, …)