TEST-ANAGRAMS

Mina Youssef

August 27, 2018

INTRODUCTION

This document reports on Test-Anagrams task by Mina Youssef for Bank of America Merrill Lynch (Dublin) Python Software Engineer position

```
IDE: PyCharm Community Edition 2017.2.1 / Jupyter Notebook
Python: 3.6.5
```

METHODOLOGY

Two approaches have been implemented, Simple and Fast Retrieval

1. Simple Approach

Execution Complexity: O(n) / Space Complexity: O(1)

In simple approach, for a give word w to find all its anagrams, we compute frequency table and iterate on all words calculate frequency table and compare, if matched then we find an anagram otherwise it is not.

```
W = meeting
frequency_table
m | 1
e | 2
t | 1
i | 1
n | 1
g | 1
```

Python class: SimpleSolution @ ..\solutions.py

2. Fast Retrieval Approach

Execution Complexity: O(1) / Space Complexity: O(n^2)

In fast retrieval approach, procedure is as following:

- Build an indexer hash table for all words in words.txt
- For a given word:
    - Calculate hash value (h)
    - Calculate frequency table (ft)
    - Retrieve all anagrams using (h) and (ft)

For hashing method used, I've created a hash function that would hash a given word to a weighted sum of prime numbers. This is done by mapping alphabets to numbers of following criteria

- Hash values are primes
- Delta between two consecutive numbers are unique

Appendix contains the code snippet that generate the mapping table

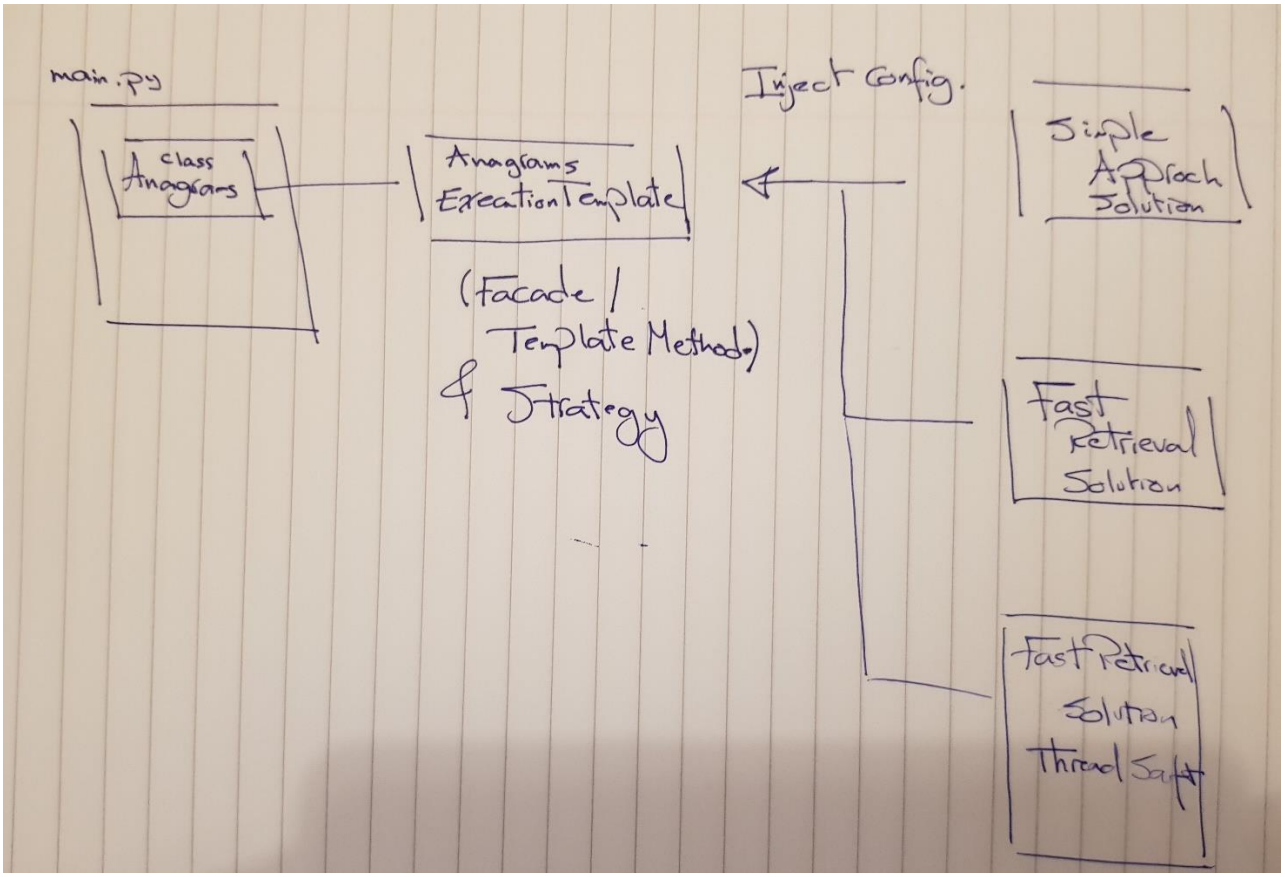
Python class: FastRetrievalSolution @ ..\solutions.py

HIGH-LEVEL DESIGN

As in the below block diagram, Anagrams class (main.py) will be use `AnagramsExecutionTemplate` instance as a Façade to switch between different solutions.

`AnagramsExecutionTemplate` implement both Template Method in `find_anagrams`() method and strategy pattern in `set_config`()

Each solution approach, has to implement two components Indexer and Finder, from the name. indexer would provide an indexing service while Finder will be performing searching/retrieval



THREADSAFE IMPLEMENTATION

ThreadSafe implementation is a wrapper for FastRetrievalSolution that provide a thread lock before entering the body of public API

We use RLock to prevent locking on same-thread invocations.

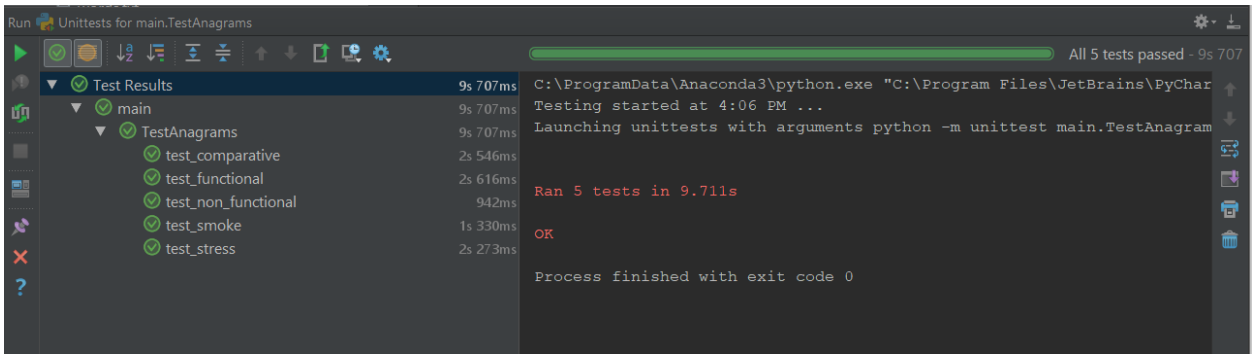
As a result of this design choice we adopted Proxy Design Pattern
https://en.wikipedia.org/wiki/Proxy_pattern

Python class: FastRetrievalFinderThreadSafe @ ..\solutions.py

UNIT TESTING

For unit testing I have create 5 categories:

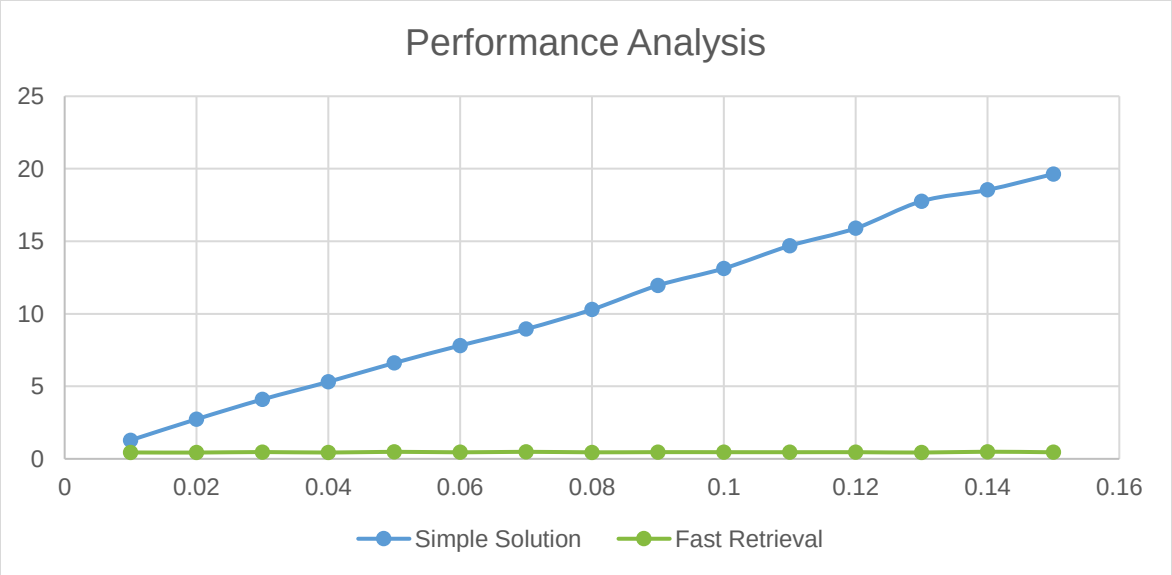
1. Smoke
2. Functional
3. Non-Functional
4. Stress
5. Comparative

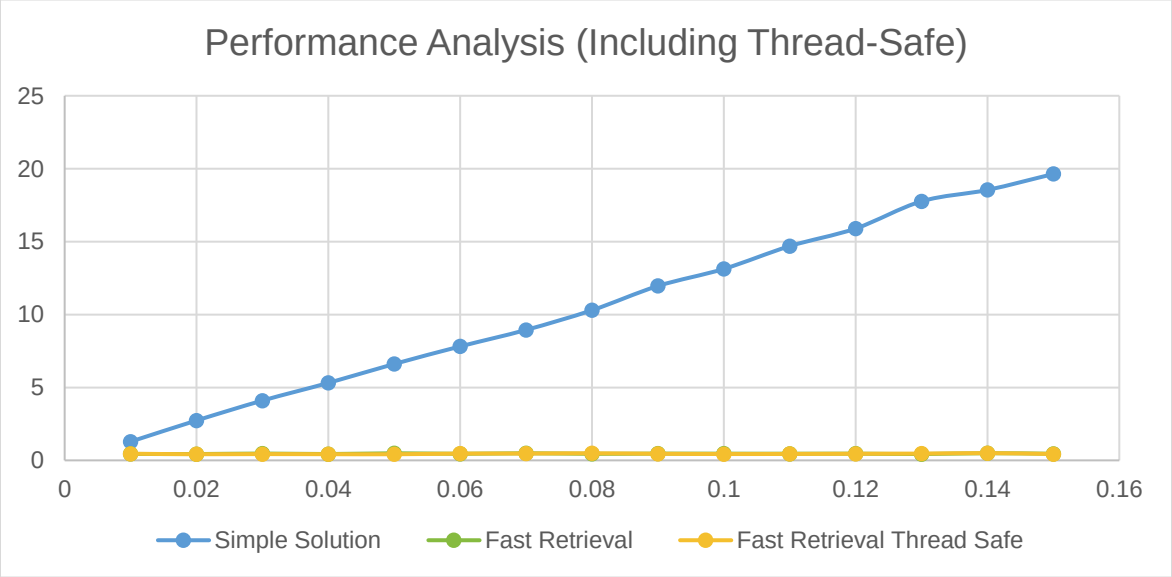


PERFORMANCE

I measured the performance of execution for both main approaches on 1% to 15% sample of all anagrams, with the following result, as we can see it confirm the algorithmic claim of simple solution execute in linear time O(n) while fast retrieval is constant O(1) (approximately)

With ThreadSafe/Non-ThreadSafe almost identical with neglectable thread overhead





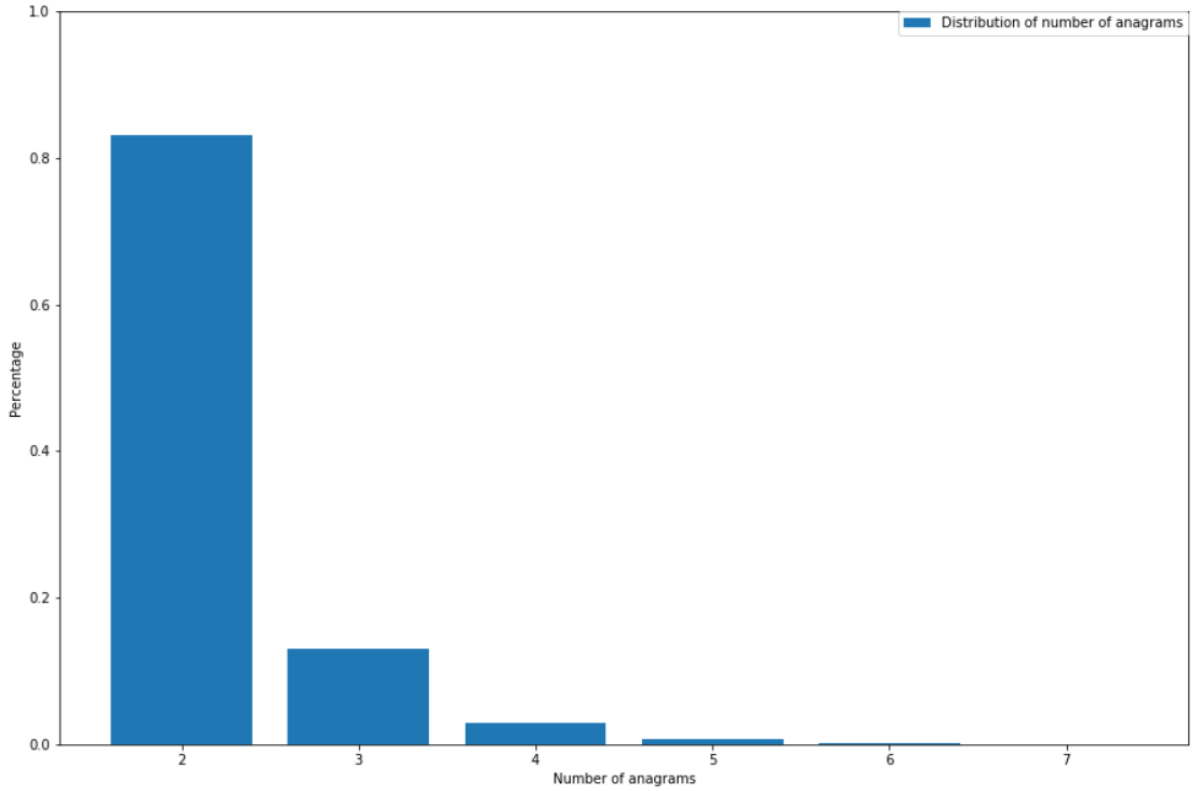
Appendix contains the numerical tabular format

ANALYTICS

I performed analytics on the resultant anagrams data set, included in Jupyter notebook, to get an idea of dataset and design the unit test accordingly

For example, I could get the distribution of number of anagrams, so I know there are from 2 to 7 anagrams sizes and based on that I have created a Functional test that cover all possible sizes

Rest of analytics finding would be located in the notebook



Python code: ..\analytics.ipynb

APPENDIX

Part of Fast Retrieval solution is to hash a given word, and one way is to calculated alphabetical weighted sum of a given word using prime number that satisfy the following two conditions:

Hash values are primes

Delta between two consecutive numbers are unique

The following auxiliary code snippet has been developed to generate such sequence to be further used in **FastRetrievalIndexer** class

```
def isPrime(n):
    if n <= 1:
        return False
    for i in range(2, n):
```

```
        if n % i == 0:
            return False
    return True


def last(l):
    if len(l) == 0:
        return None
    return l[len(l) - 1]


def get_deltas(number_list):
    deltas_list = []
    for idx in range(len(number_list) - 1):
        deltas_list.append(number_list[idx + 1] -
number_list[i])
    return deltas_list


primes = [2, 3, 5]
upto = 30
i = 4
while len(primes) < upto:
    last_prime = last(primes)
    next_prime = last_prime + 1
    deltas = get_deltas(primes)

    delta = next_prime - last_prime
    while delta in deltas:
        while not isPrime(next_prime):
            next_prime += 1
        delta = next_prime - last_prime
        next_prime += 1
    primes.append(next_prime - 1)

print(primes)
# Primes = [2, 3, 5, 11, 19, 23, 37, 47, 59, 79, 97, 113, 137,
163, 191, 223, 257, 293, 331, 353, 383, 431, 487, 541, 587, 631,
673, 733, 773, 823]

print(deltas)
# Deltas = [1, 2, 6, 8, 4, 14, 10, 12, 20, 18, 16, 24, 26, 28,
32, 34, 36, 38, 22, 30, 48, 56, 54, 46, 44, 42, 60, 40]
```

Numerical Performance result

Sample Size	Simple Solution (In Seconds)	Fast Retrieval (In Seconds)	Fast Retrieval Thread Safe (In Seconds)
0.01	1.283	0.441	0.455
0.02	2.735	0.434	0.423
0.03	4.099	0.469	0.432
0.04	5.31	0.435	0.425
0.05	6.61	0.486	0.42
0.06	7.813	0.451	0.464
0.07	8.946	0.486	0.468
0.08	10.295	0.45	0.487
0.09	11.957	0.464	0.452
0.1	13.129	0.46	0.435
0.11	14.691	0.457	0.44
0.12	15.899	0.46	0.454
0.13	17.764	0.436	0.464
0.14	18.545	0.491	0.498
0.15	19.637	0.453	0.436