Assignment n°2: Searching Algorithm

Every code part is written in Python.

Firstly, I developed the 2 asked functions:

- LinearCount
- BinaryCount

The first function use a linear algorithm in order to found the number of occurrence of the searched file, making a loop to iterate over a list of words.

When the word is found, a counter is incremented and we resume until we arrive at the end of the list.

```
from typing import List

def LinearCount(wordList: List[str], searchedWord: str) -> int:
    word_occurrence = int(0)

    for word in wordList:
        if word.casefold() == searchedWord.casefold():
            word_occurrence += 1
    print(f"Occurence of word '{searchedWord}': {word_occurrence}")
    return word_occurrence
```

The second function I developed use a Binary Tree and a Node class.

The Binary Tree has some utils functions used in the binary tree building, such as insertion or as searching.

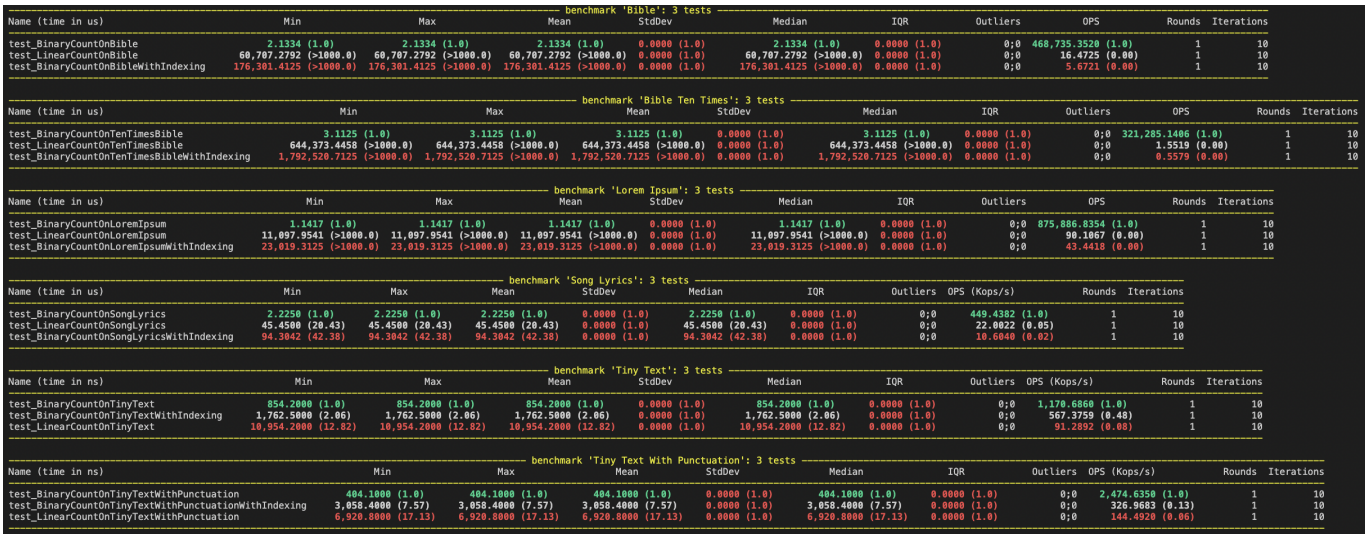
```
from enum import IntEnum

class AddOnSide(IntEnum):
    UNKNOWN = -1
    LEFT = 0
    RIGHT = 1

class Node:
    def __init__(self, word):
        self.occurrence = 1
        self.leftTree = None
        self.rightTree = None
        self.word = word

class BTree:
    def __init__(self):
        self.root = None

    def isEmpty(self) -> bool:
```

```
        return self.root is None

    def getOccurrenceOfWord(self, word) -> int:
        if self.isEmpty():
            return 0
        currentNode = self.root

        while currentNode is not None:
            if word.casefold() == currentNode.word:
                return currentNode.occurrence
            elif word.casefold() < currentNode.word:
                currentNode = currentNode.leftTree
            else:
                currentNode = currentNode.rightTree
        return 0

    def insertInTree(self, word) -> None:
        sideToAdd = AddOnSide.UNKNOWN
        if self.isEmpty():
            self.root = Node(word.casefold())
            return
        parentNode = None
        currentNode = self.root

        while currentNode is not None:
            if word.casefold() < currentNode.word:
                parentNode = currentNode
                currentNode = currentNode.leftTree
                sideToAdd = AddOnSide.LEFT
            elif word.casefold() > currentNode.word:
                parentNode = currentNode
                currentNode = currentNode.rightTree
                sideToAdd = AddOnSide.RIGHT
            elif word.casefold() == currentNode.word:
                currentNode.occurrence += 1
                return
        if sideToAdd == AddOnSide.LEFT:
            parentNode.leftTree = Node(word.casefold())
        elif sideToAdd == AddOnSide.RIGHT:
            parentNode.rightTree = Node(word.casefold())

def BinaryCount(bTree: BTree, searchedWord: str, wordList: List[str] = [])
-> int:
    if bTree.isEmpty():
        for word in wordList:
            bTree.insertInTree(word)

    occurrenceOfWord = bTree.getOccurrenceOfWord(searchedWord)
    print(f"Occurence of word '{searchedWord}': {occurrenceOfWord}")
    return occurrenceOfWord
```

Question a

In order to compare both functions for various document length and structure, I made some unit tests. Theses unit tests use a Pytest Fixture called Pytest-Benchmark.

The Pytest-Benchmark makes a benchmark for the function used. On 10 iteration for each test, I was able to arrived at this result:



benchmark 'Bible': 3 tests

Name (time in us)	Min	Max	Mean	StdDev	Median	IQR	Outliers	OPS	Rounds	Iterations
test_BinaryCountOnBible	2.1334 (1.0)	2.1334 (1.0)	2.1334 (1.0)	0.0000 (1.0)	2.1334 (1.0)	0.0000 (1.0)	0;0	468,735.3520 (1.0)	1	10
test_LinearCountOnBible	60,707.2792 (>1000.0)	60,707.2792 (>1000.0)	60,707.2792 (>1000.0)	0.0000 (1.0)	60,707.2792 (>1000.0)	0.0000 (1.0)	0;0	16.4725 (0.00)	1	10
test_BinaryCountOnBibleWithIndexing	176,301.4125 (>1000.0)	176,301.4125 (>1000.0)	176,301.4125 (>1000.0)	0.0000 (1.0)	176,301.4125 (>1000.0)	0.0000 (1.0)	0;0	5.6721 (0.00)	1	10

benchmark 'Bible Ten Times': 3 tests

Name (time in us)	Min	Max	Mean	StdDev	Median	IQR	Outliers	OPS	Rounds	Iterations
test_BinaryCountOnTenTimesBible	3.1125 (1.0)	3.1125 (1.0)	3.1125 (1.0)	0.0000 (1.0)	3.1125 (1.0)	0.0000 (1.0)	0;0	321,285.1406 (1.0)	1	10
test_LinearCountOnTenTimesBible	644,373.4458 (>1000.0)	644,373.4458 (>1000.0)	644,373.4458 (>1000.0)	0.0000 (1.0)	644,373.4458 (>1000.0)	0.0000 (1.0)	0;0	1.5519 (0.00)	1	10
test_BinaryCountOnTenTimesBibleWithIndexing	1,792,520.7125 (>1000.0)	1,792,520.7125 (>1000.0)	1,792,520.7125 (>1000.0)	0.0000 (1.0)	1,792,520.7125 (>1000.0)	0.0000 (1.0)	0;0	0.5579 (0.00)	1	10

benchmark 'Lorem Ipsum': 3 tests

Name (time in us)	Min	Max	Mean	StdDev	Median	IQR	Outliers	OPS	Rounds	Iterations
test_BinaryCountOnLoremIpsum	1.1417 (1.0)	1.1417 (1.0)	1.1417 (1.0)	0.0000 (1.0)	1.1417 (1.0)	0.0000 (1.0)	0;0	875,886.8354 (1.0)	1	10
test_LinearCountOnLoremIpsum	11,097.9541 (>1000.0)	11,097.9541 (>1000.0)	11,097.9541 (>1000.0)	0.0000 (1.0)	11,097.9541 (>1000.0)	0.0000 (1.0)	0;0	90.1067 (0.00)	1	10
test_BinaryCountOnLoremIpsumWithIndexing	23,019.3125 (>1000.0)	23,019.3125 (>1000.0)	23,019.3125 (>1000.0)	0.0000 (1.0)	23,019.3125 (>1000.0)	0.0000 (1.0)	0;0	43.4418 (0.00)	1	10

benchmark 'Song Lyrics': 3 tests

Name (time in us)	Min	Max	Mean	StdDev	Median	IQR	Outliers	OPS (Kops/s)	Rounds	Iterations
test_BinaryCountOnSongLyrics	2.2250 (1.0)	2.2250 (1.0)	2.2250 (1.0)	0.0000 (1.0)	2.2250 (1.0)	0.0000 (1.0)	0;0	449.4382 (1.0)	1	10
test_LinearCountOnSongLyrics	45.4500 (20.43)	45.4500 (20.43)	45.4500 (20.43)	0.0000 (1.0)	45.4500 (20.43)	0.0000 (1.0)	0;0	22.0022 (0.05)	1	10
test_BinaryCountOnSongLyricsWithIndexing	94.3042 (42.38)	94.3042 (42.38)	94.3042 (42.38)	0.0000 (1.0)	94.3042 (42.38)	0.0000 (1.0)	0;0	10.6040 (0.02)	1	10

benchmark 'Tiny Text': 3 tests

Name (time in ns)	Min	Max	Mean	StdDev	Median	IQR	Outliers	OPS (Kops/s)	Rounds	Iterations
test_BinaryCountOnTinyText	854.2000 (1.0)	854.2000 (1.0)	854.2000 (1.0)	0.0000 (1.0)	854.2000 (1.0)	0.0000 (1.0)	0;0	1,170.6860 (1.0)	1	10
test_BinaryCountOnTinyTextWithIndexing	1,762.5000 (2.06)	1,762.5000 (2.06)	1,762.5000 (2.06)	0.0000 (1.0)	1,762.5000 (2.06)	0.0000 (1.0)	0;0	567.3759 (0.48)	1	10
test_LinearCountOnTinyText	10,954.2000 (12.82)	10,954.2000 (12.82)	10,954.2000 (12.82)	0.0000 (1.0)	10,954.2000 (12.82)	0.0000 (1.0)	0;0	91.2892 (0.08)	1	10

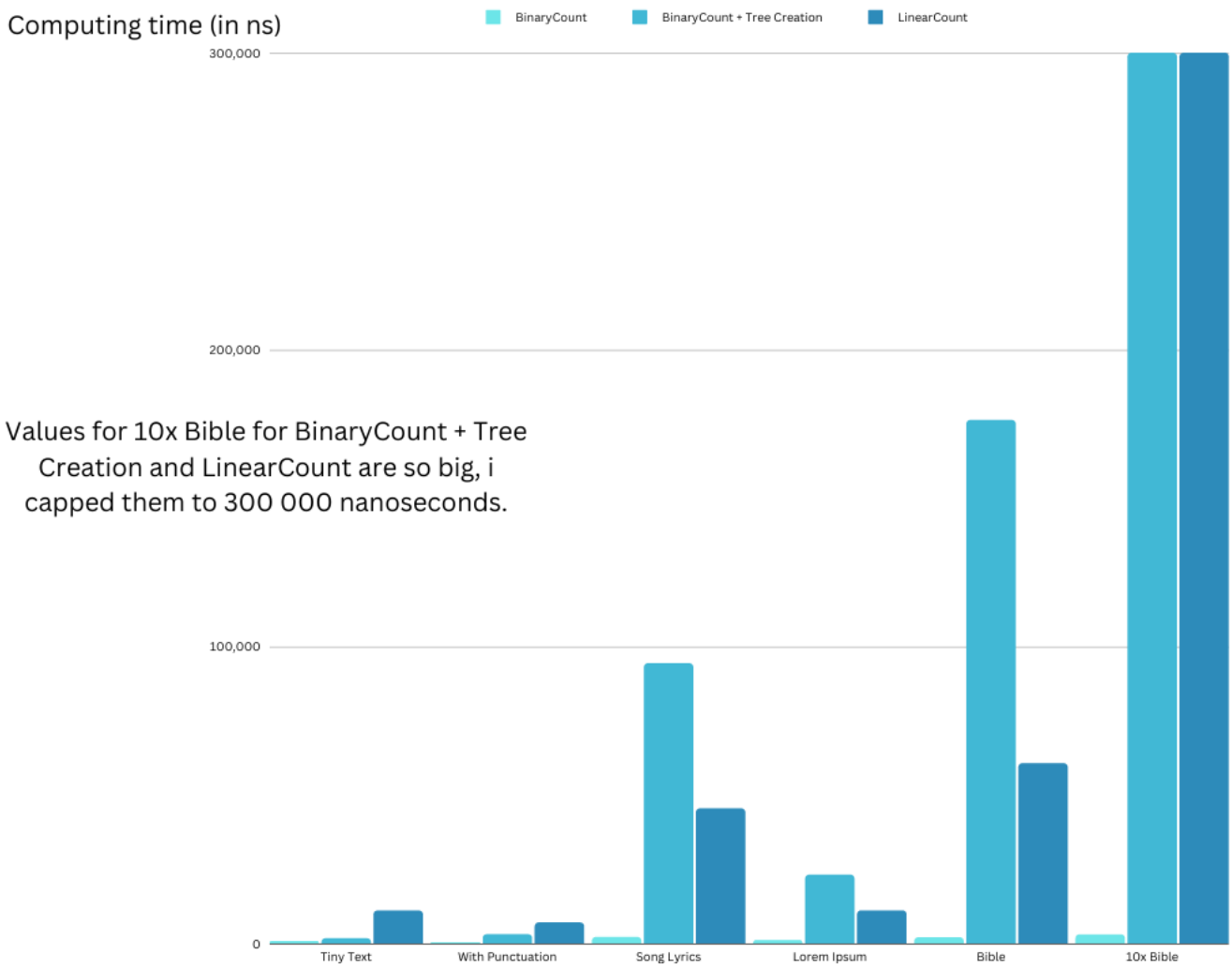
benchmark 'Tiny Text With Punctuation': 3 tests

Name (time in ns)	Min	Max	Mean	StdDev	Median	IQR	Outliers	OPS (Kops/s)	Rounds	Iterations
test_BinaryCountOnTinyTextWithPunctuation	404.1000 (1.0)	404.1000 (1.0)	404.1000 (1.0)	0.0000 (1.0)	404.1000 (1.0)	0.0000 (1.0)	0;0	2,474.6350 (1.0)	1	10
test_BinaryCountOnTinyTextWithPunctuationWithIndexing	3,058.4000 (7.57)	3,058.4000 (7.57)	3,058.4000 (7.57)	0.0000 (1.0)	3,058.4000 (7.57)	0.0000 (1.0)	0;0	326.9683 (0.13)	1	10
test_LinearCountOnTinyTextWithPunctuation	6,920.8000 (17.13)	6,920.8000 (17.13)	6,920.8000 (17.13)	0.0000 (1.0)	6,920.8000 (17.13)	0.0000 (1.0)	0;0	144.4920 (0.06)	1	10

The results shows 3 types of benchmark:

- Linear Search
- Binary Search with Tree Creation Time
- Binary Search without Tree Creation Time

I used some text with incremented size (Phrase, Football Commentary, Lyrics of French Music, Lorem Ipsum of 2Mo, Bible).



- Real values for '10x Bible' are available in the benchmark screenshot (in microseconds).
- For each different size, the time taken by each function is different. The larger the file, the longer it takes to search.
- For each test, we can see that the Binary Tree Search is faster than the Linear Search.

We also can see that the creation of the tree can is also a parameter to think about.

In some tests on tiny file, the creation of the binary tree and the search using a Binary Search Tree is faster than the linear search.

With larger files, the creation and the search using a Binary Search Tree takes too much time.

Question b

Concerning the maximum handled file size, the highest file size I made was 40 times the Bible in a file without crashing (1,74 Go).

Being developed with Python, I think the function I made don't have any file size limit or the file size limit would be really long to find.

The reason is the memory management of Python making any program almost impossible to make crash.

Therefore, both of my functions don't use recursion so they can't crash with recursion limit either.

The only thing that can make this functions crash would be adding a '/dev/zero' value to the end of a line in a file making the program running out of memory because the line doesn't end.

Question c

The diversity of a document vocabulary has an influence on the search performance.

Using the binary search tree, we can ask ourselves "How can the diversity influence the search performance?".

The more different words we have, the longer the binary tree will be and the longer it will take to go through the binary tree.

If we have words that come up frequently, we will have a number of nodes that is less than the number of words in the text. Therefore we will take less time to go through the tree.