Approach Description:

So, in this code, we're trying to count how many times certain words appear in a piece of writing. But, we've got two different ways to do it!

Linear Approach:

For the first method we break down the text into words using a regular expression. Then, we make a list of each word and count how many times each one shows up.
When we want to find out how many times a specific word appears, we just look it up in our list.

Binary Search Tree Approach:

Now, for the second method we're using a binary search tree. Each word becomes a "node" in the tree.

We sort the words as we add them to the tree. If a word is already in the tree, we just increase its count. Otherwise, we find the right spot for it.
When we want to know how many times a word appears, we go through the tree to find it.

Time complexity breakdown

Linear Search:

Time Complexity: O(n)
Explanation: In the linear search approach, we iterate through each word in the text to count its occurrences. This means that the time taken is directly proportional to the number of words in the text, denoted by 'n'. As the number of words increases, the time taken for searching also increases linearly. Therefore, the time complexity is O(n), where 'n' represents the number of words in the text.

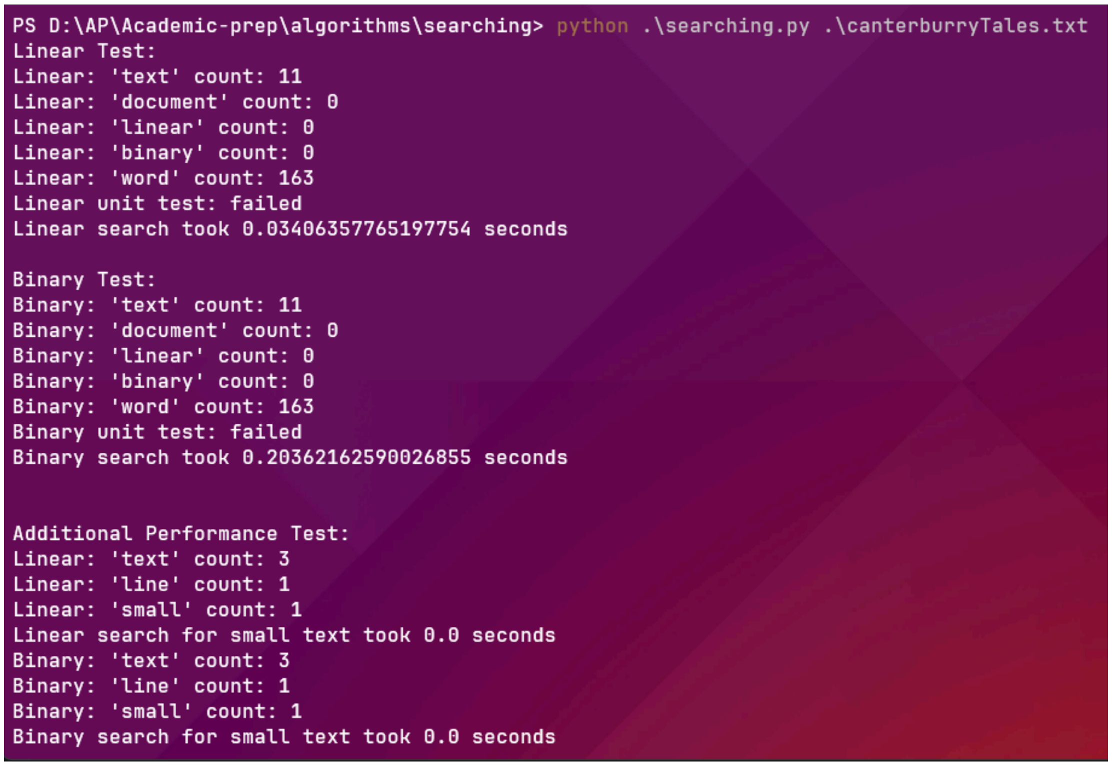
Binary Search:

Time Complexity: O(log n) on average, O(n) in worst case
Explanation: In the binary search tree approach, searching for a word involves traversing the tree from the root to the appropriate leaf node. If the text is well organized we keep cutting the amount of possibilities in half therefore the time complexity is O(log n).

If the text is not organized we might have to check every possible possibility in order to find the words we are looking for therefore the time complexity would be O(n)

Testing:

For the performance tests:



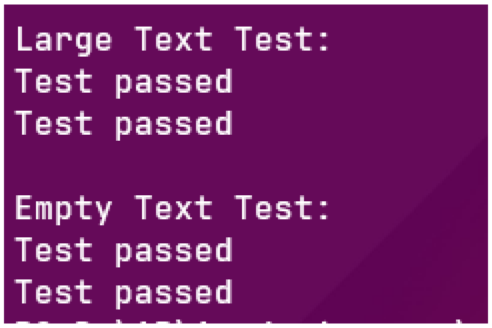
```
PS D:\AP\Academic-prep\algorithms\searching> python .\searching.py .\canterburryTales.txt
Linear Test:
Linear: 'text' count: 11
Linear: 'document' count: 0
Linear: 'linear' count: 0
Linear: 'binary' count: 0
Linear: 'word' count: 163
Linear unit test: failed
Linear search took 0.03406357765197754 seconds

Binary Test:
Binary: 'text' count: 11
Binary: 'document' count: 0
Binary: 'linear' count: 0
Binary: 'binary' count: 0
Binary: 'word' count: 163
Binary unit test: failed
Binary search took 0.20362162590026855 seconds


Additional Performance Test:
Linear: 'text' count: 3
Linear: 'line' count: 1
Linear: 'small' count: 1
Linear search for small text took 0.0 seconds
Binary: 'text' count: 3
Binary: 'line' count: 1
Binary: 'small' count: 1
Binary search for small text took 0.0 seconds
```

This test has been performed with the entirety of the Canterbury Tales. I think that the linear approach is faster here because the text is unordered and the linear approach uses an pre-existing function in Python which would probably result in the compiler having an easier time to execute the code

For the Unit Tests:



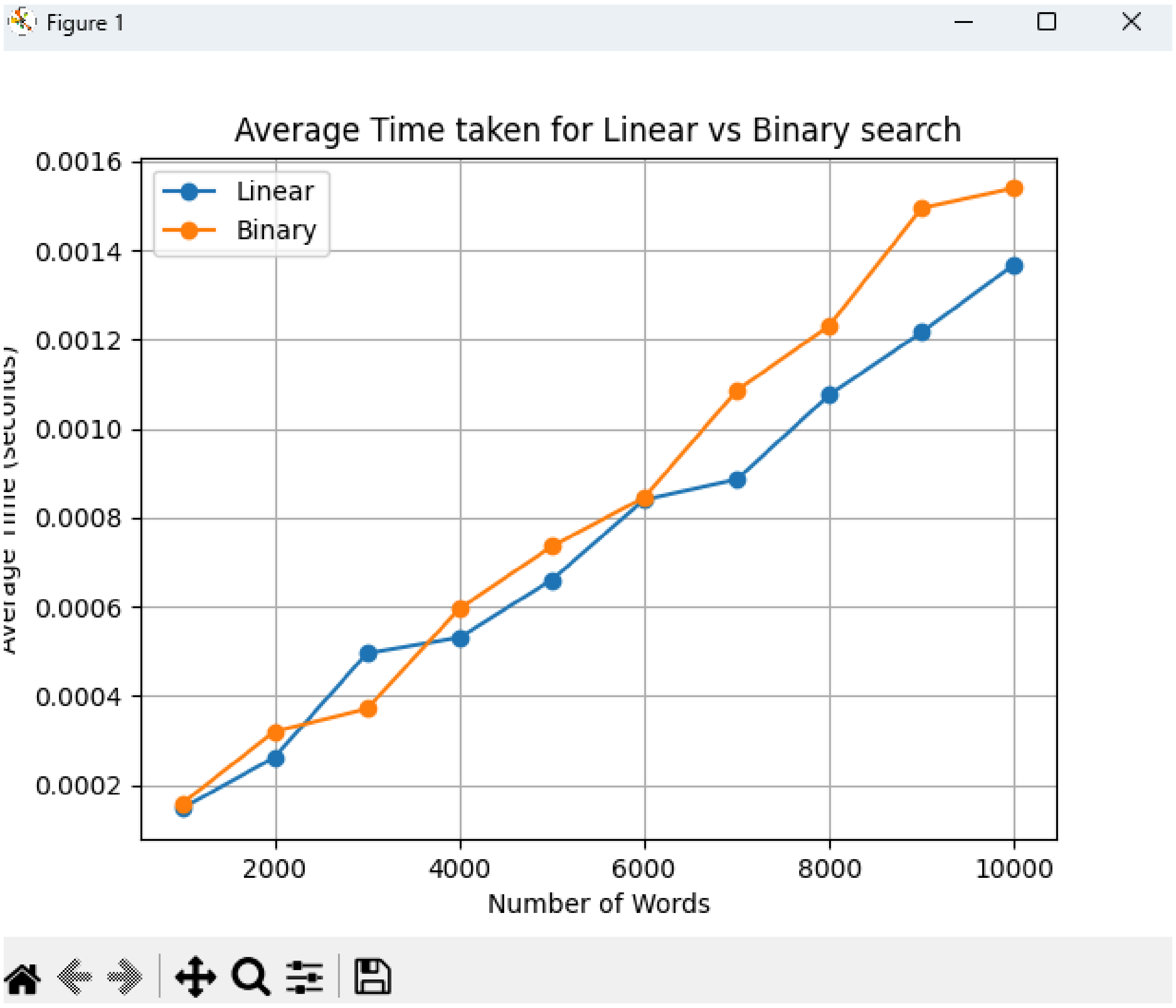
```
Large Text Test:
Test passed
Test passed

Empty Text Test:
Test passed
Test passed
```

Here I created a text consisting of 10.000 times the same word and checked if it returned the proper amount. I also checked if an empty text would return 0 and this worked as well

Performance

So we can compare the difference in performance by plotting both functions' time results based on the number of words. I ran the performance test 100 times and plotted the result as seen below



So the difference in these algorithms is negligible when working with lower word counts.

Limitations

The largest word count my code could handle was 1000000000 words. After that i started getting traceback calls

Is the diversity of the document vocabulary of any influence?

I don't think it is as the text is already unordered so both algorithms take O(n) time.