Searching Algorithms

Julio Rodriguez-Zambrano

CSCI 3412 - Algorithms

Part 1 - Problem Description

For this assignment, we are given a text file named "wordlist.txt" that contains the complete works of Shakespeare in the form of a single word, punctuation, or blank lines per line. We will need to create a concordance of the complete works of Shakespeare using the given .txt file. For this program, we will also need to print the number of unique words found, the first ten (10) words, and the last ten (10) words, to verify that the algorithm is correct. There are some rules we need to follow like a string is considered to be a word if the first character is alphabetic, although characters like hyphens and quotes may occur within a word and convert each word to lowercase to ensure that differences in case don't lead to different words, we also don't need to add the punctuation to the concordance. To complete this assignment, we are tasked to use 3 different searching algorithms for 3 different scenarios, an unsorted sequence, a sorted sequence and a hash table. We also need to perform a static analysis of the three algorithms based on the pseudo-code and also perform an analysis on the performance of the algorithms based on the execution of the code. For the unsorted sequence I will use a linear search, for the sorted sequence I will use a binary search, and for the final I will use a hash table.

Part 2 - Algorithm Design

Linear Search for a Unsorted Sequence Pseudo-code:

```
Procedure LinearSearchConcordance(words):
    Initialize an empty dictionary concordance
    Initialize variables comparisons_linear and assignments_linear to zero

    For each word in the words list do:
        found = False
        For each key in concordance do:
            Increment comparisons_linear by 1
            If key is equal to the current word then:
                Increment the count of the word in concordance
                Increment assignments_linear by 1
                Set found to True
                Break out of the inner loop
    If found is False then:
        Add the current word to concordance with a count of 1
        Increment assignments_linear by 1

    Return concordance, comparisons_linear, and assignments_linear
```

Asymptotic Growth for a Linear Search of an Unsorted Sequence Pseudo-code:

In the linear search algorithm used for constructing the concordance, The worst-case scenario would occur when there are no repeated words in the unsorted sequence. In this case, for each word in the unsorted sequence, the algorithm would have to iterate through the entire concordance dictionary to check if the word already exists. This results in O(n^2) comparisons where n is the number of words in the unsorted sequence. The expected-case scenario would depend on the distribution of words in the unsorted sequence. If there are some repeated words, the algorithm will find them more quickly, leading to fewer iterations through the concordance dictionary. The expected case time complexity would be better than the worst case but may still be somewhat close to O(n^2) if there are few repeated words. The best-case scenario would occur when all words in the unsorted sequence are the same, i.e., there's only one unique word. In this case, the algorithm would iterate through the unsorted sequence only once to build the concordance dictionary. The best-case time complexity would be O(n), where n is the number of words in the unsorted sequence.

Binary Search for a Sorted Sequence Pseudo-Code:

```
Define a function binary_search_concordance(word, sorted_sequence, concordance):
    Initialize left to 0
    Initialize right to the length of sorted_sequence - 1
    Initialize found to False

    While left is less than or equal to right and not found:
        Calculate mid as the floor division of (left + right) by 2
        If the word at mid in sorted_sequence is equal to the given word:
            If the word is already in concordance:
                Increment the count of the word in concordance
            Else:
                Add the word to concordance with count 1
            Set found to True
        If the word at mid in sorted_sequence is less than the given word:
            Update left to mid + 1
        Otherwise:
            Update right to mid - 1

    If found is False:
        Add the word to concordance with count 1

 Initialize an empty dictionary concordance

 For each word in sorted_sequence:
     Call binary_search_concordance(word, sorted_sequence, concordance)
```

Asymptotic Growth for a Binary Search of an Sorted Sequence Pseudo-Code:

In the binary search algortihm used for constructing the concordance, the worst-case scenario occurs when each word in the sorted sequence is unique, and the binary search has to traverse the entire sequence to find each word. In this scenario, for each word, the binary search would perform O(log n) comparisons, where n is the number of words in the sorted sequence. Since there are n words, the total number of comparisons would be O(n log n). The expected-case depends on the distribution of words and the efficiency of the binary search algorithm. If there are some repeated words in the sorted sequence, the binary search would find them more quickly, leading to fewer comparisons. In this case, the number of comparisons would be less than O(n log n), but still somewhat close to it. The best-case occurs when each word in the sorted sequence is repeated multiple times. In this case, the binary search would find each word quickly with only a few comparisons. The best-case time complexity would be O(n), where n is the number of words in the sorted sequence.

Hash Table Pseudo-code:

```
function hash_function(word, size):
    hash_object = hash(word)
    hash_value = convert_to_integer(hash_object) % size
    return hash_value

function insert_into_hash_table(table, word):
    index = hash_function(word, size_of_table)
    for item in table[index]:
        if item[0] == word:
            item[1] += 1
            return
    table[index].append([word, 1])

 function get_count_from_hash_table(table, word):
     index = hash_function(word, size_of_table)
     for item in table[index]:
         if item[0] == word:
             return item[1]
     return 0

 function get_unique_words_from_hash_table(table):
     unique_words = set()
     for bucket in table:
         for item in bucket:
             unique_words.add(item[0])
     return unique_words

 Create a hash table as a list of lists
 table_size = 27886  # You can adjust the size according to your data
 hash_table = [[] for _ in range(table_size)]

 Populate the hash table with words from the file
 with open('wordlist.txt', 'r') as file:
     for line in file:
         word = line.strip().lower()
         if word and word[0].isalpha():
             insert_into_hash_table(hash_table, word)
```

Asymptotic Growth for Hash Table Pseudo-Code:

In using a hash table to create a concordance, the worst-case scenario occurs when the function doesn't distribute the values evenly accross the hash table, resulting in hash collision. In this case, the complexity of inserting a word into a hash table or searching for a word would become O(n), where n is the total number of words.The expected-case scenario occurs when the hash function produces uniformly distributed hash values across the hash table.With a good hash function and an evenly distributed set of words, the complexity of inserting a word or searching for a word would be O(1) on average, as each bucket would contain a small number of elements. The best-case scenario occurs when the hash function perfectly distributes the values such that each word maps to a unique index in the hash table meaning no collision. In this case, the complexity of inserting a word or searching for a word would be O(1), as there is only one item in each bucket, and no need to iterate through a list of collisions.

Part 3 - Implementation

In [12]:

```python
# This section will follow the given rules for the concordance and create a list from those rules
words = []
with open('wordlist.txt', 'r') as file:
    for line in file:
        word = line.strip().lower()
        if word and word[0].isalpha():
            words.append(word)


# This line create the unsorted sequence
unsorted_sequence = words


# This initializes an empty dictionary to store the concordance
concordance = {}
# These two lines initialize variables to track the number of comparisons and assignments
comparisons_linear = 0
assignments_linear = 0

# This for loop will iterate through each word in the unsorted sequence
for word in unsorted_sequence:
    # Flag to indicate if the word is found in the concordance
    found = False
    # This loop will iterate through each key in the concordance
    for key in concordance:
        # Increment comparison count
        comparisons_linear += 1
        # This will check if the current key matches the word
        if key == word:
            # If so, increment the count for that word
            concordance[key] += 1
            # Increment assignment count
            assignments_linear += 1
            # Set found flag to True and exit loop
            found = True
            break
    # If the word was not found in the concordance
    if not found:
        # Add the word to the concordance with a count of 1
        concordance[word] = 1
        # Increment assignment count
        assignments_linear += 1

# Print the number of unique words
print("Number of unique words =", len(concordance))

# Print the first ten words alphabetically and their counts
print("First ten words alphabetically and their counts:")
first_ten_sorted = sorted(concordance.items())[:10]
for word, count in first_ten_sorted:
    print(f'"{word}" : {count}')

# Print the last ten words alphabetically and their counts
print("\nLast ten words alphabetically and their counts:")
last_ten_sorted = sorted(concordance.items())[-10:]
for word, count in last_ten_sorted:
    print(f'"{word}" : {count}')


# This section will print the performance metrics for linear search on unsorted sequence
print("\nPerformance Metrics for Linear Search on Unsorted Sequence:")
print("Number of comparisons:", comparisons_linear)
print("Number of assignments:", assignments_linear)
```

```
Number of unique words = 27886
First ten words alphabetically and their counts:
"a" : 13679
"a'" : 120
"a's" : 1
"a-bat-fowling" : 1
"a-bed" : 12
"a-birding" : 4
"a-bleeding" : 2
"a-breeding" : 1
"a-brewing" : 1
"a-broach" : 1

Last ten words alphabetically and their counts:
"zenelophon" : 1
"zenith" : 1
"zephyrs" : 1
"zo" : 1
"zodiac" : 1
"zodiacs" : 1
"zone" : 1
"zounds" : 1
"zur" : 2
"zwaggered" : 1

Performance Metrics for Linear Search on Unsorted Sequence:
Number of comparisons: 1645031853
Number of assignments: 807861
```

In [14]:

```python
# This section will follow the given rules for the concordance and create a list from those rules
words = []
with open('wordlist.txt', 'r') as file:
    for line in file:
        word = line.strip().lower()
        if word and word[0].isalpha():
            words.append(word)

# This line will create the Sorted Sequence
sorted_sequence = sorted(words)

# This will initialize an empty dictionary to store the concordance
concordance = {}
# These two lines will initialize variables to track the number of comparisons and assignments
comparisons_binary = 0
assignments_binary = 0

# Define a binary search function
def binary_search(word, sorted_sequence):
    # This will initialize pointers for binary search
    left = 0
    right = len(sorted_sequence) - 1
    # This will perform a binary search
    while left <= right:
        # Increment comparison count
        global comparisons_binary
        comparisons_binary += 1
        # This will calculate the middle index
        mid = (left + right) // 2
        # Compare the word with the word at the middle index
        if sorted_sequence[mid] == word:
            # If the word is found, return the index
            return mid
        elif sorted_sequence[mid] < word:
            # If the word is greater than the word at the middle index, update the left pointer
            left = mid + 1
        else:
            # If the word is less than the word at the middle index, update the right pointer
            right = mid - 1
    # If the word is not found, return -1
    return -1

# This will iterate through each word in the sorted sequence
for word in sorted_sequence:
    # This will perform a binary search to find the word in the sorted sequence
    index = binary_search(word, sorted_sequence)
    # If the word is found
    if index != -1:
        # This will increment the count of the word in the concordance dictionary
        if word in concordance:
            concordance[word] += 1
        else:
            concordance[word] = 1
        # This will increment the assignments count
        assignments_binary += 1

# Print out the unique number of words
print("Number of unique words =", len(concordance))

# Print the first ten words alphabetically and their counts
print("First ten words alphabetically and their counts:")
first_ten_sorted = sorted(concordance.items())[:10]
for word, count in first_ten_sorted:
    print(f'"{word}" : {count}')

# Print the last ten words alphabetically and their counts
print("\nLast ten words alphabetically and their counts:")
last_ten_sorted = sorted(concordance.items())[-10:]
for word, count in last_ten_sorted:
    print(f'"{word}" : {count}')

# This section will print performance metrics for binary search on sorted sequence
print("\nPerformance Metrics for Binary Search on Sorted Sequence:")
print("Number of comparisons:", comparisons_binary)
print("Number of assignments:", assignments_binary)
```

```
Number of unique words = 27886
First ten words alphabetically and their counts:
"a" : 13679
"a'" : 120
"a's" : 1
"a-bat-fowling" : 1
"a-bed" : 12
"a-birding" : 4
"a-bleeding" : 2
"a-breeding" : 1
"a-brewing" : 1
"a-broach" : 1

Last ten words alphabetically and their counts:
"zenelophon" : 1
"zenith" : 1
"zephyrs" : 1
"zo" : 1
"zodiac" : 1
"zodiacs" : 1
"zone" : 1
"zounds" : 1
"zur" : 2
"zwaggered" : 1

Performance Metrics for Binary Search on Sorted Sequence:
Number of comparisons: 7144746
Number of assignments: 807861
```

In [17]:

```python
import hashlib

# Global counters for assignments and comparisons
assignments = 0
comparisons = 0

def hash_function(word, size):
    global assignments, comparisons
    # A simple hash function using hashlib
    hash_object = hashlib.md5(word.encode())
    hash_value = int(hash_object.hexdigest(), 16) % size
    assignments += 2  # Assignment for hash_object and hash_value
    return hash_value

def insert_into_hash_table(table, word):
    global assignments, comparisons
    index = hash_function(word, len(table))
    for item in table[index]:
        comparisons += 1  # Comparison for item[0] == word
        if item[0] == word:
            item[1] += 1
            assignments += 1  # Assignment for incrementing count
            return
    table[index].append([word, 1])
    assignments += 1  # Assignment for appending new word

def get_count_from_hash_table(table, word):
    global comparisons
    index = hash_function(word, len(table))
    for item in table[index]:
        comparisons += 1  # Comparison for item[0] == word
        if item[0] == word:
            return item[1]
    return 0

def get_unique_words_from_hash_table(table):
    global assignments
    unique_words = set()
    for bucket in table:
        for item in bucket:
            unique_words.add(item[0])
            assignments += 1  # Assignment for adding word to set
    return unique_words

# Create a hash table as a list of lists
table_size = 27886  # Number of unique words
hash_table = [[] for _ in range(table_size)]

# Populate the hash table with words from the file
with open('wordlist.txt', 'r') as file:
    for line in file:
        word = line.strip().lower()
        if word and word[0].isalpha():
            insert_into_hash_table(hash_table, word)

# Get the unique words and sort them
unique_words = sorted(get_unique_words_from_hash_table(hash_table))

# Print the number of unique words found
print("Number of unique words =", len(unique_words))

# Print the first ten words
print("First ten words alphabetically and their counts:")
for i in range(10):
    word = unique_words[i]
    count = get_count_from_hash_table(hash_table, word)
    print(f'"{word}" : {count}')

# Print the last ten words
print("\nLast ten words alphabetically and their counts:")
for i in range(len(unique_words) - 10, len(unique_words)):
    word = unique_words[i]
    count = get_count_from_hash_table(hash_table, word)
    print(f'"{word}" : {count}')

# Print the number of assignments and comparisons
print("\nPerformance Metrics for Hash table implementation:")
print("Number of comparisons:", comparisons)
print("Number of assignments:", assignments)
```

```
Number of unique words = 27886
First ten words alphabetically and their counts:
"a" : 13679
"a'" : 120
"a's" : 1
"a-bat-fowling" : 1
"a-bed" : 12
"a-birding" : 4
"a-bleeding" : 2
"a-breeding" : 1
"a-brewing" : 1
"a-broach" : 1

Last ten words alphabetically and their counts:
"zenelophon" : 1
"zenith" : 1
"zephyrs" : 1
"zo" : 1
"zodiac" : 1
"zodiacs" : 1
"zone" : 1
"zounds" : 1
"zur" : 2
"zwaggered" : 1

Performance Metrics for Hash table implementation:
Number of comparisons: 836421
Number of assignments: 2451509
```

Part 4 - Analysis I really didn't have any expectations before implementing the algorithms beside knowing that a linear search algorithm is slower than a binary search algorithm, which in turn is slower than a hash table, so I kinda expected that the linear search algorithm will compare more and assign more than the binary search and vice versa between the binary search algorithm and the hash table implementation. Post implementation of the algorithms and with the results, it shows the same results I predicted, for the linear search algorithm I got the number of comparisons: 1645031853 and the number of assignments: 807861 which is understandable. For the binary search algorithm I got the results of number of comparisons: 7144746 and the number of assignments: 807861 and for the hash table I got the number of comparisons: 836421 and the number of assignments: 2451509. I expected these results when it came to comparisons but for the assignments I got an equal amount between the linear and binary search but I got more for the hash table, maybe I messed up somewhere to get these results.

Part 5 - Conclusion

For me the most difficult part of this assignment was creating a hash table since I don't really have much experience with using the hashlib or hash tables in general, so I think thats why my assignments came out really high compared to both the linear and binary search algorithms, but besides that the project wasn't to hard and like I said in the analysis section, I already knew the general results from the algorithms and wasn't suprised by the results. It was also intersting to implement these algortihms and actually prove the time complexities.

In [ ]: