CSCI 3412 Algorithms

Homework 2

Matt Sullivan

Part 1: Program Description

In Problem Set 2, each student will analyze and implement three (3) different search algorithms.
There will be static analysis of the algorithms – based on the written pseudo-code – as well as analysis of the performance of the algorithms – based on the execution of the code.
I will give you an input file, wordlist.txt, that contains the complete works of Shakespeare with a single word or punctuation per line as well as blank lines where they appear in the text. 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. [This isn't perfect, but it is simple to implement and will ensure everyone uses the same rules and gets the same counts.] Convert each word to lowercase to ensure that differences in case don't lead to different words. It is only the words we are interested in – that is, don't add punctuation to the concordance.
Here is a sample from the beginning and end of the file.

```
A
MIDSUMMER-NIGHT'S
DREAM
Now
,
fair
Hippolyta
,
our
…
state
,
That
like
events
may
ne'er
it
ruinate
.
```

Each student will create a concordance of the complete works of Shakespeare using the given word list. A concordance is a sorted list of the words with a count of the number of occurrences of each word. For this program, you 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.
For this data set and using the rules above, the first ten (10) and last ten (10) words alphabetically and their counts are:
Alphabetic
Number of unique words = 27886
"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
…
"zenelophon" : 1
"zenith" : 1
"zephyrs" : 1
"zo" : 1
"zodiac" : 1
"zodiacs" . 1
"zone" : 1
"zounds" : 1
"zur" : 2
"zwaggered" : 1

The Search Algorithms

Each student will select and implement search algorithms to create the concordance based on the following data structures.

1. Unsorted sequence (vector or list)
2. Sorted sequence (vector or list)
3. Hash table

For the first two – unsorted and sorted sequence, you may use either a vector or list based implementation. Also, the sorted sequence naturally maintain the concordance in sorted order. For the other two, you will have to sort the concordance following its construction.

Note that it is the searching algorithm used in building the concordance whose analysis we are interested in – not the sorting phase (for the unsorted and hash table algorithms).

Part 2: Algorithm Design

For all of my searches, I used python to remove all duplicates from the overall array and make a second array masterList, which showed that there are 27886 unique words out of the 807861 words in Shakespeare. This allowed me to increase the efficiency of my searches and eliminate concerns about double dipping (where a word to be found appears multiple times in the search, and is thereby counted on both ends) I then sorted this masterList using sort, and stepped through the masterlist in order to create sorted concordances by default. I also defined a custom class named Entry, containing a string Name and an int Number. This allowed me to store the name and number together in the concordance array, by making it an array of Entries. NOTE: I also tested the masterList unsorted and it had the same results. I just had to add the < operator to my Entry class, which allowed me to sort the concordance after creation. I left it as it is because it's easier to debug a program if I can count on at least one set of inputs beign sorted. The code appears both ways as I wanted to test each search function. They all perform correctly in both instances.

For my pseudo-code, I didn't find any pseudo-code specifically online, but I did adapt my work from both my code in CSCI 1410 and 2312.

Please see pseudo-code and static analysis on the next page.

Unsorted Search:

```
for word in masterList:
        for entry in wordArray
                if word ==entry
                        wordCount++
                        compare++
                else
                        compare++
concordance.append(entry(word,wordCount)
```

The search and construction of this concordance will be the size of masterlist * the size of the wordArray, and dependent on both. The maximum size is if both are the same size, leading to a O(n^2) All of the appends to the concordance are equal to the size of the masterList.

Sorted Search:

```
wordCount =0
middle = begin+end/2
if begin>end
        return none
if the middle value > searchTerm
        compare++
        return binarySearch(begin, middle)
if the middle value < searchTerm
        compare++
        return binarySearch(middle, end)
else
        step through array left and right until not searchTerm
                wordCount++
        return wordCount,compare
for item in masterList
        append(Entry(item,binarySearch)
        Assign++
```

Since the wordArray was sorted prior to the search, each search is completed in O(log n) comparisons, where n is the size of the wordArray. The total number of operations should be m(log n) where m is the size of the masterList. Again, the number of appends to the concordance are equal to size of masterList.

Hash Search:

```
for item in masterList
        index = hash(item)
        for contents in index
                if contents == search
                        concordance.append(contents)
                        append++
                        compare++
                else
                        compare++
```

Since the hash is created before the search, it is simple to see that it has O(1) search speed for each

individual search, giving the total operations of n where n is the size of the masterList. There will be slightly more comparisons than the size of the masterList, due to collisions in the hash table. These can be reduced at the expense of memory by increasing the size of the hash table. Again, the number of appends to the concordance are equal to size of masterList.

Part 3: Implementation

Unsorted Search

```
class Entry:                    ## Custom class to create concordance
    def __init__(self, word, number):
        self.word=word
        self.number=number
    def __repr__(self):     # used for debugging
        return (self.word + ": " + str(self.number))
    def __str__(self):      #used for debugging
        return (self.word + ": " + str(self.number))
    def __lt__(self,other): #used to sort
        return self.word<other.word
wordArray=[]
concordance=[]
compareNum=0
assignNum=0
myFile = open("wordlist.txt", "r")
for line in myFile:
    if line[0].isalpha():
        wordArray.append(line.strip('\n').lower())
masterList=list(set(wordArray))     #removes doubles from wordArray, stores in masterList
print("Number of words total: " + str(len(wordArray)))
print("Number of unique words" + str(len(masterList)))
for word in masterList:
    wordCount=0
    for item in wordArray: # Searching unsorted list
        if item == word:
            wordCount+=1
            compareNum +=1
        else:
            compareNum+=1
    concordance.append(Entry(word,wordCount))   #appends new Entry to concordance
    assignNum+=1
concordance.sort()      # sorts concordance using defined <
for entry in concordance:       #Prints only the first and last 10
    if entry in concordance[:10] or entry in concordance[-10:]:
        print (entry)
print ("Number of comparisons: " + str(compareNum))
print ("Number of assignments: " + str(assignNum))
```

Sorted Search:

```
totalCompare=0
totalAssign=0
class Entry:        #same class to store Entries in concordance
    def __init__(self, word, number):
        self.word=word
        self.number=number
    def __repr__(self):
        return (self.word + ": " + str(self.number))
    def __str__(self):
        return (self.word + ": " + str(self.number))
    def __lt__(self,other):
        return self.word<other.word
def binarySearch(array, search, compareNum, begin, end):
    wordCount=0
    middle = int((begin+end)/2)
    if begin>end:   #Terminates if word not found
        return None
    elif array[middle] > search:    #recursively calls search on lower half
        compareNum+=1
        return binarySearch(array,search, compareNum, begin, middle-1)
    elif array[middle] < search:    #recusively calls search on upper half
        compareNum+=1
        return binarySearch(array,search, compareNum, middle+1, end)
    else:       # if the search term is found
        compareNum+=1
        wordCount+=1
        left = middle-1
        right = middle+1
        while (left>=0) and (array[left]==search):  #steps through array backwards until word is not found
or beginning is reached
            left-=1
            wordCount+=1
        while (right < len(array)-1) and (array[right]==search):#steps through array until word is not
found or end is reached
            right+=1
            wordCount+=1
        return wordCount, compareNum  #returns both the word count and the number of comparisons
wordArray=[]
concordance=[]
myFile = open("wordlist.txt", "r")  #reads into wordArray from file
for line in myFile:
    if line[0].isalpha():
        wordArray.append(line.strip('\n').lower())
masterList=list(set(wordArray))     #removes doubles from wordArray and stores in masterList
#masterList.sort()
print("Number of words total: " + str(len(wordArray)))
print("Number of unique words: " + str(len(masterList)))
```

```
wordArray.sort() #sorts wordArray
for item in masterList:
    concordance.append(Entry(item,str(binarySearch(wordArray,item,0,0,len(wordArray))[0])))
#appends each item from masterArray
    totalAssign+=1
    totalCompare+= binarySearch(wordArray,item,0,0,len(wordArray))[1] #increments totalCompare by
number of operations during search
concordance.sort()      #sorts concordance
for entry in concordance:       #prints out first and last 10 entries
    if entry in concordance[:10] or entry in concordance[-10:]:
        print(entry)
print("Total number of comparisons: " + str(totalCompare))
print("Total number of assignments: " + str(totalAssign))
```

Hash Table Search:

```
class Entry:
    def __init__(self, word,number):
        self.word=word
        self.number=number
    def __repr__(self):
        return (self.word + ": " + str(self.number))
    def __str__(self):
        return (self.word + ": " + str(self.number))
    def __lt__(self,other):
        return self.word<other.word
hashTable=[[] for _ in range(100000)]   #creats list of empty lists
wordArray=[]    #array to store words read from file
concordance=[]  #array to store Entrys made of words from file
numCompare=0
numAssign=0
myFile = open("wordlist.txt", "r")      #reads from file
for line in myFile:
    if line[0].isalpha():
        wordArray.append(line.strip('\n').lower())
#print (wordArray)
masterList=list(set(wordArray)) #removes duplicates from wordArray
masterList.sort()
wordsInHash=0
print("Number of words total: " + str(len(wordArray)))
print("Number of unique words: " + str(len(masterList)))
for word in wordArray:  #Creates hash table using words in wordArray
    sum=0
    inTable=False
    sum = hash(word)%100000     #uses python hash function and limits it to size of array
    for contents in hashTable[sum]:     #steps through elements in the list at that address
        if word == contents.word:       #if the word is already there
            inTable=True
            contents.number+=1          #iterates word number
```

```
            break
    if inTable==False:                  # if it reaches the end and the word is not present
        hashTable[sum].append(Entry(word,1))    #Adds new entry and sets the number to one
        wordsInHash+=1          #iterates the number of words in the hash
for item in masterList:     #steps through master list
    sum=0
    sum=hash(item)%100000
    for contents in hashTable[sum]:         #steps through the array at the hashed address
        if (item == contents.word):         #if the word is in the array
            concordance.append(contents)    #append the entry to the concordance
            numAssign+=1
            numCompare+=1
            break
        else:
            numCompare+=1
#concordance.sort       ## works, but sorting masterlist first aids in debugging
for entry in concordance:
    if entry in concordance[:10] or entry in concordance[-10:]:
        print (entry)
print("Number of comparisons: " + str(numCompare))
print("Number of assignments: " + str(numAssign))
```

Part 4: Analysis

For the unsorted search. We, basically stepped through the entire array of words every time, and just counted the number of times the word appeared. Thus the basic O was O(n). This was multiplied by the number of searches performed. In our case, we had an n of 807861 (total list of words in file) an m of 27886 (the list of unique words in the file) and our program returns a comparison number of 22,528,011,846, which is m*n, as we predicted. The number of assignments made to build the concordance was flat at m, or 27,886.

For the sorted search, we used a binary search through the sorted array for each number. We then totaled the number of comparisons made and got 476,681, which is roughly m(log n). The number of comparisons is increased by roughly n if you include the steps needed to count all of the sorted members of a particular word. In this case the number of operations is 1,256,656. The base O value of this algorithm is O (log n). Again, the number of assignments made to build the concordance was based directly on the amount of items searched for.

For the hash search, we used a hash table. For each member in the wordArray, we assigned a hash value using the built in python hash function. Then we stepped through the array at that hash value, and if there was already an Entry matching the item, we incremented the number value of that Entry. Otherwise, we added a new Entry to the array. At 100,000 spaces, most of the lists at the hash values had 2 collisions, which meant that during the search, the program only had to step through arrays of size of about 2. In this case, the number of comparisons made counted in at 31777, compared to the total number of search items 27886. The extra 3891 operations are the result of the collisions, but I found by trial that increasing or shrinking the size of the hash table caused more to build than was saved, and this is a happy middle ground in terms of performance. The big O value for the searching of a single object is clearly O(1), and this was multiplied by the number of searches performed.

Part 5: Conclusion
As we can see, each of the three searches acted as we predicted.
The unsorted search returned with an O (n).
The sorted search returned with an O(log n).
The hash search returned with an O(1).

Clearly, from a strictly searching point of view, hash tables are the quickest of the 3 methods. But they require more memory and require time to construct the table. Likewise, depending on the sort method, it will take time and memory to sort the initial input. However, the decrease in counts more than makes up for these two when compared to the unsorted search.

The chart below is on the logarithmic scale, as the sorted and hash comparisons would be too small to see on a normal scale.