**Lab 2 – Password Analysis Report**

For our second lab, we were instructed to read a password file containing 10 million combinations of usernames and passwords, count the number of times that a password appears, add the passwords to both a linked list and a Python dictionary list, and finally obtain the 20 most used passwords in a given list by means of bubble and merge sort. However, due to the size of the text file being used, smaller password files were created in order to truly determine the program’s robustness, time/memory complexity, and special cases.

The way the solution was approached in this problem was by first implementing a built-in, Python file reader and saving the “read” contents of the file into a variable simply called “file.” Once the file was read and stored, a splitting and partitioning method was used to separate the contents of every line into three separate sections, as each line contained usernames and not only passwords. With the partition method, truples, or sub-lists, of each line were created, which allowed the isolation of passwords to be a relatively simple process.

Once the passwords were isolated, as per Solution A of the lab, an empty Linked List with nodes containing not only the password string but also the integer “count” for reach item in the list was created. A simple while loop was then created with two conditionals within it that check the current list for the presence of the password, and the “count” was the only element updated if the password was found in the list. If the password was not found in the list, it was simply inserted, and the count was also updated, along with the “size” value of the list (used for merge sort access later). This insertion method runs at a O(n) time complexity, where n is the number of items in the given text file being used.

Once the linked list was populated without duplicates, a bubble sort was performed to determine the top 20 passwords of any given list. The bubble sort I implemented was very similar to the one using arrays, which contains two nested for-loops. Each adjacent item is simply compared, and descending order was achieved not by the re-linking of nodes within the list, but by swapping the data of each node (i.e. the password string and count integer). The bubble sort itself runs at a O(n2) running time, since it consists of two, nested for-loops. Finally, to print the top 20, a simple method that prints the first 20 elements of the sorted list was created and called. This runs at a O(1), since printing a set number of values never takes more than constant time.

The dictionary portion of this lab proved to be a little confusing, as the exact procedure outlined in the lab instructions was rather vague. Two different methods were implemented in order to make use of the dictionary, Python lists. The first one simply uses the file variable and does what the previous linked list populator did, that is, read the file, check for duplicates, and insert at the hash value/key provided by the password string itself. If the password already existed on the list, the count was updated, but the password would not be added. This is direct indexing, and there were no nodes at each “bucket” of the dictionary. This method runs at an O(n), as the insertion itself is O(1), however, the size of the file being read truly determines the running time.

The second implementation of the dictionary was using the already sorted Linked List. After sorting the linked list with bubble sort, each Node of the list was inserted at the respective hash key created by the password string itself. Since the list was already filtered for duplicates, there was no duplicates in the “Nodes” list and accessing the actual password and count value for each node at each Hash Key was very simple. This method runs at a O(n), since it simply adds the Nodes to the “buckets” based on the size, n, of the list.

The final, and most challenging, portion of this lab was the merge sort of the linked list after it was already populated. This method works by determining the linked list sized, using an iterator to determine the mid-point of the linked list, and then “de-links” the two halves, and continues to separate the list into halves recursively until the size is “None.” Once the list has been reduced to several halves, and ultimately into single nodes, they are merged back together in descending order, with the greatest being at the beginning of the list. This method has a time complexity of O(n\*logn), as the work is recursively being divided by 2, however, the memory complexity is O(2n). This is due to the fact that for every split in the list, a temporary list is created until they are all fully merged. Due to the fact that Python only allocated a certain amount of memory in their virtual environment to avoid stack overflow, this method tends to fail with larger lists. The threshold appears to be greater than 2000 passwords, but less than 5000. No modifications to the allotted memory for recursion in Python were attempted (articles online advised against it).

**Time Complexity/Results**

Since the given time complexity of any algorithm is based on the fastest growing operation, the time complexity of this entire program would be O(n2). This is due to the bubble sort possessing the two nested for-loops. The following charts and screen shots illustrate this n2 behavior:



Chart of all the total number of comparisons for each method implemented. Clearly, bubble-sort requires the most number of comparisons/operations.



Screen shot of operation count with an input size of 2000.



Screen shot of operation count with an input size of 5000.



Screen shot of operation count with in input size of 10000.



Screen shot of operation count with an input size of 15000.



Screen shot of operation count with an input size if 20000

**Conclusion**

This lab really demonstrated how certain data structures are better suited for some applications but not for others. For instance, the dictionary’s ability to insert, search, and essentially manage data at a constant rate (since it is a hash table) makes it extremely quick and efficient. In addition, merge sorting and bubble sorting seem to be a better approach when it comes to sorting arrays, but not necessarily linked lists. Regardless, it was definitely a great refresher on how linked lists can be used.

**Appendix (Source Code)**

#!/usr/bin/env python3  
# -\*- coding: utf-8 -\*-  
*"""  
File: Main.py  
Name: Angel Villalpando  
Date: 10/17/2018  
Course: CS 2302 - Data Structures  
  
"""*linkedListCounter = 0  
dictionaryCounter = 0  
nodeDictionaryCounter = 0;  
bubbleSortCounter = 0  
mergeSortCounter = 0  
  
class Node(object): # basic Node class, as per lab-description  
 def \_\_init\_\_(self, password):  
 self.password = password  
 self.count = 1  
 self.next = None  
  
class LinkedList(object): # linked list class  
 def \_\_init\_\_(self):  
 self.head = None  
 self.size = 0  
  
 def get\_size(self):  
 return self.size  
  
 def insert(self, data): # this method inserts into the linked list, but only updates the 'count' if duplicate  
 newNode = Node(data)  
 global linkedListCounter  
  
 if self.head == None:  
 self.head = newNode  
 self.size += 1  
 return  
  
 curr = self.head  
  
 while curr is not None: # checks duplicate, only updates the count  
 if curr.password == newNode.password:  
 curr.count += 1  
 linkedListCounter += 1  
 break  
  
 elif curr.next is None:  
 curr.next = newNode  
 self.size += 1  
 linkedListCounter += 1  
 break  
  
 curr = curr.next  
  
 def print\_list(self): # this is the Linked List print method  
 printval = self.head  
 while printval is not None:  
 print(printval.password, printval.count)  
 printval = printval.next  
  
 def print\_top\_20(self): # This is also a linked list method, but only for the top 20 passwords used  
 curr = self.head  
 print("The top 20 passwords are: ")  
 print("-------------------------")  
 for i in range(20):  
 print("Password: ", curr.password)  
 print("Count: ", curr.count, "\n")  
 curr = curr.next  
  
 def bubble\_sort(self): # this is a bubble sort method. The information in the Nodes is swapped, not the Nodes  
 global bubbleSortCounter  
 if self.size > 1:  
 for i in range(self.size-1):  
 bubbleSortCounter += 1  
 current = self.head  
 for j in range(self.size-1-i):  
 bubbleSortCounter += 1  
 if current.count < current.next.count:  
 bubbleSortCounter += 1  
 tempCount = current.count  
 tempPassword = current.password  
 current.count = current.next.count  
 current.password = current.next.password  
 current.next.count = tempCount  
 current.next.password = tempPassword  
 current = current.next  
  
  
def merge\_sort(list): # merge sort method for list sorting  
  
 curr1 = list # temporary iterator to obtain the midpoint of the list  
 mid = size\_update(list) // 2 # determines mid point for iteration  
  
 if list == None or list.next == None:  
 return list  
  
 while mid-1 > 0:  
 curr1 = curr1.next # gets to mid point  
 mid = mid - 1  
  
 curr2 = curr1.next  
 curr1.next = None  
 curr1 = list  
  
 left = merge\_sort(curr1) # creates left half of list  
 right = merge\_sort(curr2) # creates right half of list  
  
 return merge(left, right) # merges already sorted halves  
  
  
def merge(left, right):  
 global mergeSortCounter  
  
 if left == None and right == None:  
 return None  
 if left == None:  
 return right  
 if right == None:  
 return left  
  
 if left.count < right.count: # merges the parameter "halves" in descending order  
 mergeSortCounter += 1  
 final = right  
 final.next = merge(left, right.next)  
 else:  
 mergeSortCounter += 1  
 final = left  
 final.next = merge(left.next, right)  
  
 return final  
  
  
def size\_update(list): # size updater is called on every recursive of merge sort and is updated  
 size = 0  
 curr = list  
 while curr is not None:  
 size = size + 1  
 curr = curr.next  
  
 return size  
  
def print\_node\_dictionary(dict, node): # This prints the dictionary when each index is a linked list "head"  
 curr = node.head  
 while curr != None:  
 print("Password: ", dict[curr.password].password)  
 print("Count: ", dict[curr.password].count, "\n")  
 curr = curr.next  
  
def print\_20\_dict(dictionary, node): # This prints the top 20 from linked list dictionary  
 curr = node.head  
 for k in range(20):  
 print("Password: ", dictionary[curr.password].password)  
 print("Count: ", dictionary[curr.password].count, "\n")  
 curr = curr.next  
  
def dictionary\_populator(dictionary, node): # this method populates the linked list dictionary, during file read  
 global nodeDictionaryCounter  
 current = node.head  
 while current != None:  
 dictionary[current.password] = current  
 nodeDictionaryCounter += 1  
 current = current.next  
  
def non\_sort\_dictionary\_populator(dictionary, item): # This method populates the dictionary straight from the file  
 global dictionaryCounter  
 if item in dictionary: # checks for duplicates  
 dictionary[item] = dictionary[item] + 1  
 dictionaryCounter += 1  
 else:  
 dictionary[item] = 1  
 dictionaryCounter += 1  
  
def print\_dictionary(dict): # This prints the dictionary contents when the values at index are NOT Linked Lists  
 for item in dict:  
 print("Password: ", item)  
 print("Count: ", dict[item], "\n")  
  
def main():  
  
 file = open("passwordTester75K.txt", "r", encoding="ISO-8859-1") # encoding used to correct error at file read  
  
 bubblePasswordList = LinkedList() # list to be sorted by Bubble Sort  
 mergePasswordList = LinkedList() # list to be sorted by the Merge sort  
 nonSortDict = {} # dictionary with NO LINKED LIST indices  
 sortedDict = {} # Linked List dictionary  
  
 for line in file: ## This populates all of the lists and dictionaries as the file is being read  
 justPassword = line.strip('\n').partition('\t')  
 bubblePasswordList.insert(justPassword[2])  
 mergePasswordList.insert(justPassword[2])  
 non\_sort\_dictionary\_populator(nonSortDict, justPassword[2])  
  
 file.close()  
  
 print("\n\*\*\*\*\* Using Bubble Sort \*\*\*\*\*\*\*")  
 bubblePasswordList.bubble\_sort()  
 bubblePasswordList.print\_top\_20()  
  
 #print("\*\*\*\*\*\* Using Merge Sort \*\*\*\*\*\*\*")  
 #merge\_sort(mergePasswordList.head)  
 #mergePasswordList.print\_top\_20()  
  
 print("\nPre-Sorted Dictionary: ")  
 print("----------------------")  
 print\_dictionary(nonSortDict)  
  
 print("\nPost-Sorted Dictionary")  
 print("----------------------")  
 dictionary\_populator(sortedDict, bubblePasswordList)  
 print\_node\_dictionary(sortedDict, bubblePasswordList)  
  
 print("The number of comparisons for Linked List insertion (w/ duplicate check) is: ", linkedListCounter)  
 print("The number of comparisons for Node Dictionary population is: ", nodeDictionaryCounter)  
 print("The number of comparisons for Dictionary Insertion is: ", dictionaryCounter)  
 print("The number of comparisons for Bubble Sort is: ", bubbleSortCounter)  
 print("The number of comparisons for Merge Sort is: ", mergeSortCounter)  
  
  
  
  
main()

**Certification**

“I certify that this project is entirely my own work. I wrote, debugged, and tested the code being presented, performed the experiments, and wrote the report. I also certify that I did not share my code or report or provided inappropriate assistance to any students in the class.”

Angel Villalpando

10/20/2018