**Edgar Escobedo**

**CS 2302**

**Diego Aguirre**

**Lab 2 Report**

**CS 2302 Lab 2 Report**

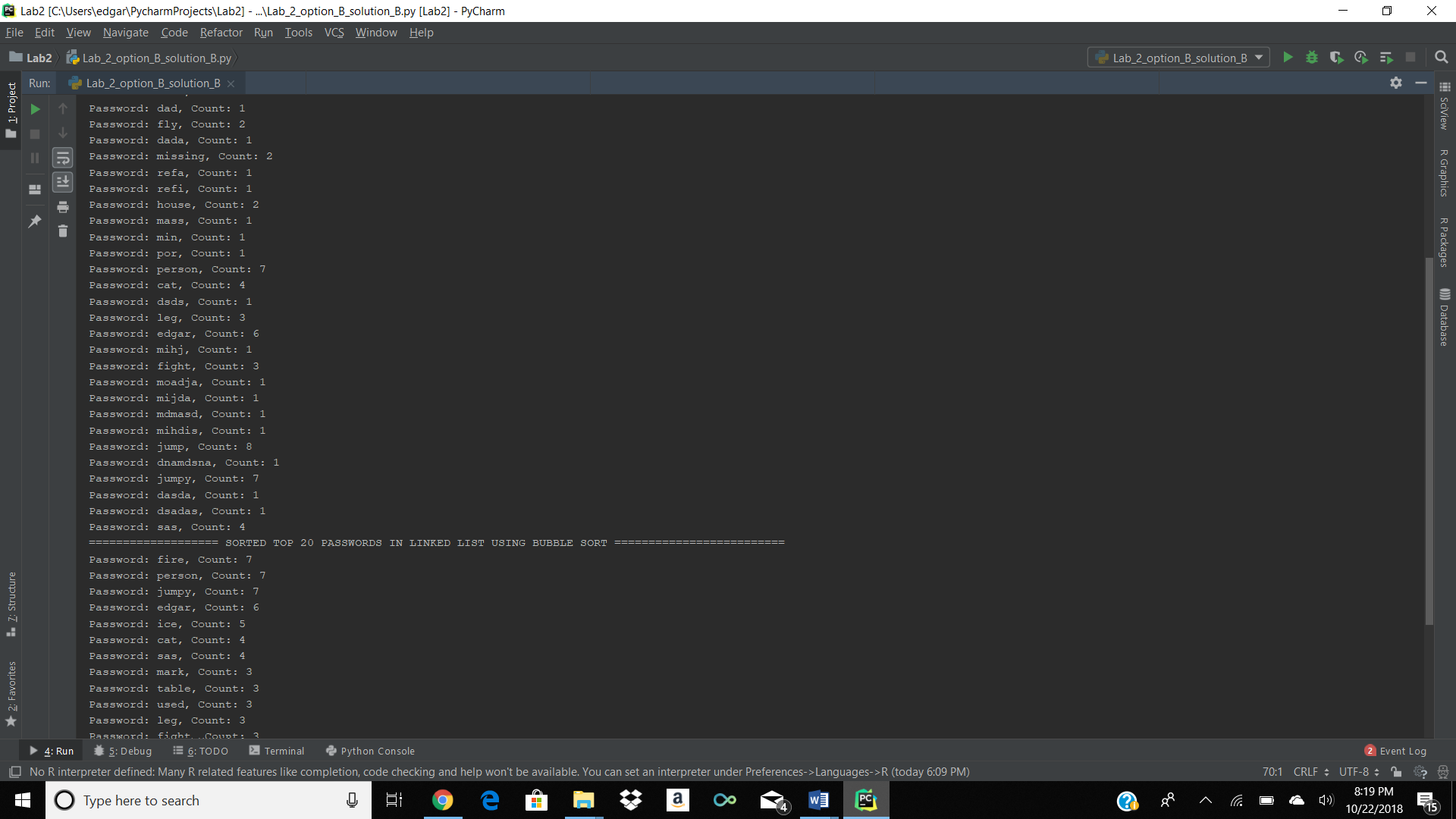
For this lab 2, we were asked to create a program which would perform previously known sorting methods, but now with the use of a linked list. To perform these methods, given that the nodes had to have a counter to sort them by the amount of times that they appear in the text, we created the linked list in two different ways. The first was to count the number of times that the password appear, by going through the whole linked list and checking if that word already appeared, if it did, we only had to update the counter. The second way was to create a dictionary, in which we would we would use the password as key and the information as the amount of times that they appeared. Lastly as mentioned before we had to use the sorting algorithms, which were the bubble sort and the merge sort to sort the linked list, using again the counter inside of the nodes to order them in ascending order.

To solve this problem, I divided the program into different tasks, starting by making sure to read only the passwords inside the file, not the usernames. To do so, I read the file, stored the whole file into lines inside a regular list, in each line I would have the username and password. Then by using the split method already installed in python, I would again divide each line now into a list with 2 inputs, the username and password. Finally, I would check if the length was greater than one (since some passwords were null) inside a loop, and if it was, I would send that word either to the add to linked list method (solution A) or to the dictionary (solution B).

Having the passwords by themselves, the next step was to do the first solution A and B, to start I did solution A. For this solution, I created a method inside the linked list class which would create a node with the password I gave it at that moment, set the counter to 1 and the next to None. Then using a loop, it would iterate through the whole linked list in which I was trying to add the node, if that node wasn’t in the list, I would add it to the end, if I found a match, I would update the counter of that node inside the linked list to plus 1. Then for solution B, having already created an empty dictionary, I would look in the dictionary by the key (which is the password) every time I tried to add a password, if that password was already in the dictionary, I would update the information of the item plus 1, if not, again I would add it to the dictionary.

Lastly, I had to implement the part 2 solution A and B, which consisted of the sorting the linked list using bubble sort and selection sort, again I started with solution A. To do so, I would have a while loop, which was going to iterate until “swap” (my Boolean variable which would tell me if any swaps were made) did not have any swaps. Then the next while loop would have a previous node, and a current node (current was a “next” ahead of previous), which would iterate while current was not none. If the counter of the current node was greater than the counter of the previous node, I would change the information of the previous and current node, such information being the counter and the actual password.

The last solution B was the merge sort, I decided to divide that structure into two methods, merge sort and merge list. First the merge sort method would divide the by half out initial list, until it reached the base case of having lists of size 1, to do so, I created a method inside the node class, which would iterate through the whole length of the node and would return in that way the size of the node. Then using this, I would iterate starting by the head of the linked list until the length of that node, divided by 2 and minus 1. This would leave that node exactly in the middle, after that I would set the “right” side as the next value after the midpoint in another new node linked list, then set the “next” of the midpoint to None in order to cut that part of the list by half, and set the left side as the head of the initial node (now with half the size). After this the second method, merge list, would come in, we compare the two given lists and if either one of them are empty we just return the other one, if both contain an element we compare and keep the greatest one, then we repeat this using the next element of the one we kept and compare as parameters for the same method again, attaching the result of that recursive call to the next of the greatest value now stored in the sorted list. At the end both methods would give use the sorted linked list, next I will talk about the big O running time with my initial experiments I performed to check the program. To start there is a display of one of my text examples which sorted the text as desired.



This next table will show the running time for the 5 different inputs, solution A\_A is the bubble sort using the iterative method of solution A, A\_B is the merge sort using again the iterative method of solution A. Next Solution B\_A times are given using the bubble sort but this time with the dictionary of solution A, lastly, using the dictionary for B\_B we sort the linked list using merge sort.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Solution A\_A |  |  |  |  |  |
| Input | 100 | 200 | 500 | 1000 | 2000 |
| Time | 0.004983425 | 0.009974479 | 0.030915737 | 0.131649494 | 0.52509593 |
| Solution A\_B |  |  |  |  |  |
| Input | 100 | 200 | 500 | 1000 | 2000 |
| Time | 0.000963211 | 0.002223405 | 0.005952835 | 0.01196909 | 0.029922246 |
| Solution B\_A |  |  |  |  |  |
| Input | 100 | 200 | 500 | 1000 | 2000 |
| Time | 0.004983187 | 0.008010387 | 0.03693294 | 0.116693258 | 0.546584606 |
| Solution B\_B |  |  |  |  |  |
| Input | 100 | 200 | 500 | 1000 | 2000 |
| Time | 0.00099659 | 0.00543723 | 0.008978367 | 0.011453152 | 0.024931192 |

In conclusion after running all the experiments, the hypothesis of the running times results true. As learned from previous classes, but finally seen with my own program and experiments, the running time for bubble sort is O(n^2) and merge sort is O(n(log n)). Giving this, I realize that is true that merge sort is in fact faster than bubble sort, but also noticing that the memory space used for the method is very large for the multiple creation of smaller lists. I learned a great number of things during this lab, from a deeply understanding of the bubble sort and merge sort now in a harder implementation (which is with the linked lists) to how to get certain inputs from a text and the usage of a dictionary.

Appendix:

|  |
| --- |
|  |
|  |
|  | #Edgar Escobedo |
|  | #80502432 |
|  | #Lab 2 option B, Solution A |
|  | #\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* |
|  |  |
|  | #This lab is solution A, which uses the "iterative" traversal, meaning that every time a new node is introduced, the program |
|  | #will traverse the whole linked list in order to check if this is a new node and added, if it's already in the list |
|  | #it will simply increase the counter inside the node |
|  |  |
|  |  |
|  | #Regular node class and constructor which stores the next and counter |
|  | class Node(object): |
|  | password = "" |
|  | count = -1 |
|  | next = None |
|  |  |
|  | def \_\_init\_\_(self, password, count, next): |
|  | self.password = password |
|  | self.count = count |
|  | self.next = next |
|  |  |
|  |  |
|  | #Method to get the size of the node by traversing it |
|  | def get\_length(self): |
|  | curr = self |
|  | counter = 0 |
|  | while curr != None: |
|  | counter +=1 |
|  | curr = curr.next |
|  | return counter |
|  |  |
|  |  |
|  | #Linked list, when is first created it has size 0, here we have the get length method |
|  | #The most important part is my add\_to\_list method which will first create the node with the information that it's given, if the list |
|  | #is empty it will insert the node as head. If is is not empty it will traverse the whole list to check if that password "node" is already |
|  | #in it, if it is it will only update the counter, if it is not, it will add to the end of the list |
|  |  |
|  | class Linked\_List: |
|  | def \_\_init\_\_(self): |
|  | self.size = 0 |
|  | self.head = None |
|  |  |
|  | def length(self): |
|  | return self.size |
|  |  |
|  | def add\_to\_list(self, password): |
|  | added\_node = Node(password, 1, None) |
|  | if self.head == None: |
|  | self.head = added\_node |
|  | self.size = self.size + 1 |
|  | else: |
|  | curr = self.head |
|  | while (curr != None): |
|  | if str(added\_node.password) == str(curr.password): |
|  | curr.count += 1 |
|  | return |
|  | elif curr.next == None: |
|  | curr.next = added\_node |
|  | self.size = self.size + 1 |
|  | return |
|  | curr = curr.next |
|  |  |
|  | def list\_read(self): |
|  | if (self.head == None): |
|  | return |
|  | curr = self.head |
|  | while curr != None: |
|  | print("Password: "+curr.password + ", Count: "+str(curr.count)) |
|  | curr = curr.next |
|  |  |
|  |  |
|  | def list\_20\_top(self): |
|  | if self.head == None: |
|  | return |
|  | curr = self.head |
|  | counter = 0 |
|  | while curr != None: |
|  | if counter == 20: |
|  | return |
|  | else: |
|  | print("Password: " + curr.password + ", Count: " + str(curr.count)) |
|  | curr = curr.next |
|  | counter +=1 |
|  |  |
|  |  |
|  |  |
|  | #READING FILE |
|  | #creation of a list with all the usernames and passwords from the file |
|  | my\_file = open("10-million-combos.txt", "r") |
|  | passwords\_list = my\_file.readlines() |
|  |  |
|  | #Separating passwords |
|  | #In this method we will use the previously created list with usernames and passwords to store it in the linked list, we will do so, |
|  | #by iterating each "line" in our list with usernames and passwords, we will separate them by means of the split method, check if |
|  | #the length is greater than 1 (because in some cases we have a null password), if it is, we will send the password as the information of |
|  | #the node and add it to our linked list |
|  | counter = 0 |
|  | linked\_list\_passwords = Linked\_List() |
|  | while (counter != len(passwords\_list)): |
|  | list\_x = passwords\_list[counter] |
|  | wrds = list\_x.split() |
|  | if len(wrds) > 1: |
|  | linked\_list\_passwords.add\_to\_list((wrds[1])) |
|  | counter = counter + 1 |
|  |  |
|  |  |
|  |  |
|  | #My bubble sort method has 4 cases, which are the simple cases, checks if the list is empty or if it's length 1, just returning the list |
|  | #If the length is 2, it will swap the nodes if needed. If the list if any longer it will enter my list method, which will check if any swaps |
|  | #were as the base cases, if no swaps were made it means that is sorted and it will end, if not it will go throught the length of |
|  | #the list and swap the information inside of the nodes if the curr (node one step ahead) is greater than previous. |
|  | def bubble\_sort(ls): |
|  | if ls.size == 0: |
|  | return None |
|  | if ls.size == 1: |
|  | return ls |
|  | if ls.size == 2: |
|  | if ls.head.count < ls.head.next.count: |
|  | tmp = ls.head |
|  | ls.head = ls.head.next |
|  | ls.head.next = tmp |
|  | return ls |
|  | previous = ls.head |
|  | swapped = True |
|  | while swapped: |
|  | swapped = False |
|  | previous = ls.head |
|  | curr = ls.head.next |
|  | while curr != None: |
|  | if previous.count < curr.count: |
|  | tmp\_count = previous.count |
|  | tmp\_password = previous.password |
|  | previous.count = curr.count |
|  | previous.password = curr.password |
|  | curr.count = tmp\_count |
|  | curr.password = tmp\_password |
|  | swapped = True |
|  | curr = curr.next |
|  | previous = previous.next |
|  |  |
|  |  |
|  | #The merge sort method works by recursively "cutting" the linked list by half, we will do so by means of traversing through |
|  | #our list to find the right place to partition it. Once the list is divided into lists of one value or node, we will use |
|  | #the second part which is the merge\_lists, which will check the value inside the node and compare it against each other, to place |
|  | #the biggest values in the right position and in this way sort it. |
|  | def merge\_sort(list): |
|  | tmp = list |
|  | if list is None or list.next is None: |
|  | return list |
|  | length = get\_length(list) |
|  | for i in range(int(length / 2) - 1): |
|  | tmp = tmp.next |
|  | lefty = tmp.next |
|  | tmp.next = None |
|  | righty = list |
|  | left = merge\_sort(lefty) |
|  | right = merge\_sort(righty) |
|  | return merge\_lists(left, right) |
|  |  |
|  |  |
|  | def merge\_lists(lefty, righty): |
|  |  |
|  | if lefty == None: |
|  | return righty |
|  | if righty == None: |
|  | return lefty |
|  | if lefty.count > righty.count: |
|  | final\_list = lefty |
|  | final\_list.next = merge\_lists(lefty.next, righty) |
|  | else: |
|  | final\_list = righty |
|  | final\_list.next = merge\_lists(lefty, righty.next) |
|  | return final\_list |
|  |  |
|  |  |
|  | second\_list = Linked\_List() |
|  | second\_list = linked\_list\_passwords |
|  | print("====================================== UNSORTED LINKED LIST ==========================================") |
|  | #Sorting the linked list |
|  | Linked\_List.list\_read(linked\_list\_passwords) |
|  | bubble\_sort(linked\_list\_passwords) |
|  | print("================== SORTED TOP 20 PASSWORDS IN LINKED LIST USING BUBBLE SORT =========================") |
|  | Linked\_List.list\_20\_top(linked\_list\_passwords) |
|  | merge\_sort(second\_list.head) |
|  | print("") |
|  | print("================== SORTED TOP 20 PASSWORDS IN LINKED LIST USING MERGE SORT =========================") |
|  | Linked\_List.list\_20\_top(second\_list) |
|  |  |

I, Edgar Escobedo, 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 student in class.

<https://github.com/ejescobedo/LAB2>