Stephanie Galvan

Date: September 27, 2019

Lab #2 – Option B

CS 2302 – Datat Structures MW 1:30PM – 2:50PM

Instructor: Diego Aguirre

TA: Gerardo Barraza

Fall 2019

Lab 2 – Option B

Introduction

Lab 2 explores the use of linked lists and the effectiveness of merge sort and bubble sort. Given a file with 10 million passwords, store all of the passwords in both a linked list and a dictionary, and compare the runtime of the aforementioned sorting algorithms. A node class has been provided which includes a password, a count (to keep track of duplicates), and a pointer to the next node. The provided file is 10 million lines long separated with new spaces, but it is expected that additional test cases will be implemented to ensure the program runs smoothly. The following lab will be divided in two parts: Part A which deals with storing the data in linked lists and dictionaries, and Part B, which deals with bubble and merge sort for only the acquired linked lists.

Proposed Solution Design and Implementation

Part A: Linked Lists & Dictionaries

Linked List:

Since the node class was already provided, a LinkedList class was created which contains a head, a tail, and length. Additionally, the linked list class contains two functions that print the content of the linked list, the first function *print_llist()* prints all contents and *print_20()* prints the first 20 elements of a linked list. In order to store the data into a linked list, the function *get_llist()* will be called and it will take a filename as a parameter. First, the function will define an empty linked list and then the function will open the file given and traverse through each line. Each line will be converted into a node and call the *check_duplicate()* function to determine whether the current node is a duplicate. If there is a duplicate, it will increase the count of the node on the list; else, it will append the current node to the list. Finally, a linked list will be returned and the result will be printed.

Dictionary:

Similar to the linked list function, the *get_dict()* takes a filename as a parameter and traverses through the file and adds the current line to the dictionary; it will increase the key if a duplicate is found. Also, the dictionary gets printed by using pprint instead of the regular print (this was used to avoid creating an additional method).

Part B: Bubble Sort & Merge Sort

Next, the linked list needed to be sorted and the program implements both bubble sort and merge sort. Bubble sort receives a linked list as a parameter and it checks the elements pair by pair. Additionally, it contains a filed called *is_sorted* that will prevent the linked list from being checked when it is already sorted. On the other hand, merge sort is divided in three different methods: *merge_sort(), splitting(),* and *merging()*. Inside the *merge_sort ()* method all the main functions happen. First, this method takes a node as a parameter (initially it is the head), then it calls the *splitting()* method in order to divide the linked list in half. Once the linked list is fully divided, the *merging()* method is called in which everything gets merged together.

Test cases will be discussed in the section below.

Experimental Results

Some of the test cases used will be explained below. Besides running the normal behavior of the code, seven additional test cases were developed to test the program:

1. 20unique.txt: This test case contains 500 unique passwords and the expected output is a linked list of length 500 with a count of 1.

2. alreadySorted.txt: The text file contains an already sorted document that is expected to run more efficiently with bubble sort than merge sort.

3. ascendingOrder.txt: All the passwords count are sorted in ascending order, but they should be in descending order.

4. empty.txt: An empty file. Program should handle this test smoothly.

5. sameAmountOfDuplicates.txt: Different passwords but thy all contain the same count.

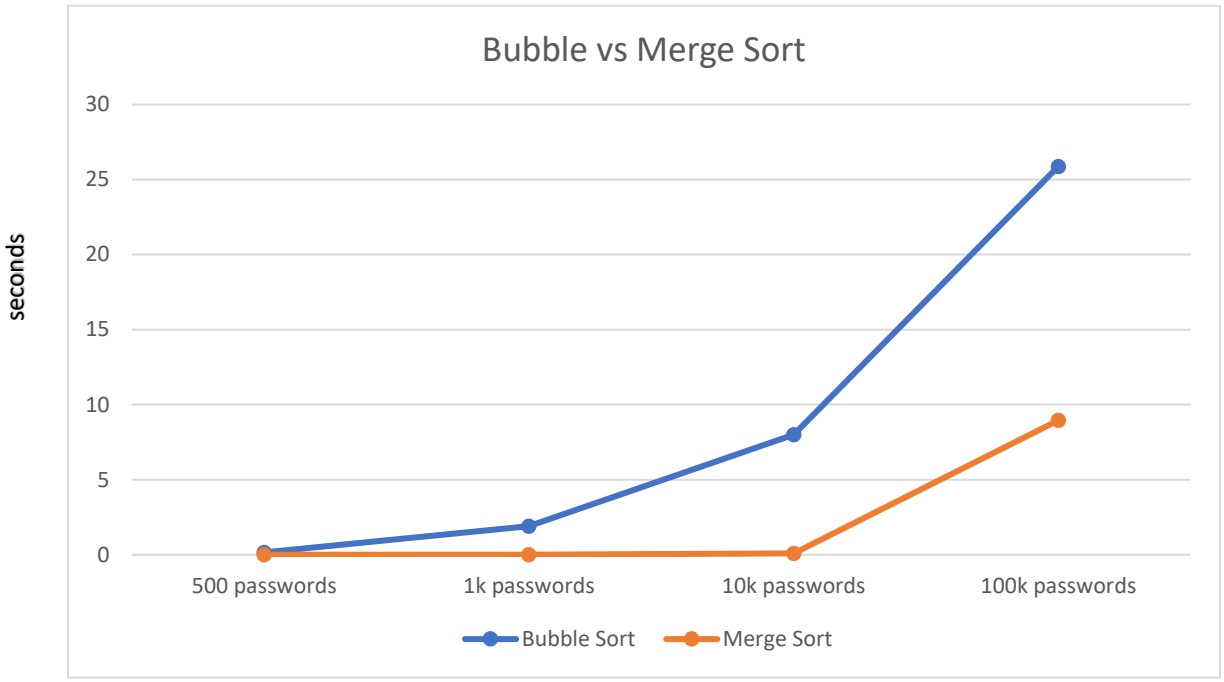
6. samePassword.txt: This test case contains the same password and the expected output is linked list of length 1 with a count of 500.

7. smallTest.txt: A smaller random test case to ensure all functions run smoothly.

For the normal behavior of the code, the test run for 10 million passwords was tested once but the size and run time was immensely large. In order to compare the run time for bubble sort and merge sort, the aforementioned test cases are included in the table below. Also, test cases A, B, C, and D refer to the normal behavior of the code when 500, 1k, 10k, and 100k passwords are passed as a parameter.

	Bubble Sort	Merge Sort
empty	0s	0s
same password	0s	0s
unique passwords	0.04146742s	0.00409436s
same amount of duplicates	0s	0s
already sorted	0s	0.00100278s
ascending order	0s	0s
small test	0.0s	0.00102686s
[A] 500 passwords	0.15784s	0.0017384s
[B] 1k passwords	1.89956s	0.00427899s
[C] 10k passwords	7.99351s	0.982431s
[D] 100k passwords	25.87641s	8.954323s

The graph below only compares the test cases A, B, C, and D. As demonstrated, the bubble sort reflects a complexity of O(n^2) while merge sort reflects a complexity of O(n log n).



***important note: the seconds refers to only sorting without printing the results. Printing the results leads to longer runtime.

Conclusion

Overall, merge sort proved to be the most effective algorithm for sorting compared to bubble sort; however, if the file given was already sorted, then bubble sort would run faster since merge sort would continue to divide the data while bubble sort contains a variable that checks whether the given file has already been sorted or not. Regarding linked lists and dictionaries, dictionaries appear to be a more straightforward solution; however, linked lists give the flexibility of being heavily user-defined and linked lists can also be sorted while dictionaries do not. Finally, the lab underscored the understanding of the time complexity of algorithms in order to avoid running a program that runs for several hours.

Appendix

Source code found at https://github.com/stephypy/CS2302_P2OB/blob/master/lab2.py

Code originally written in py charm.

Imported libraries:

```
11  import pprint
12  import time
```

Time used to measure runtime and ppring used for dictionaries.

Given node class:

```
15  class Node(object):
16      password = ""
17      count = -1
18      next = None
19
20      def __init__(self, password, count, next):
21          self.password = password
22          self.count = count
23          self.next = next
```

Linked List class:

```
26  class LinkedList:
27      def __init__(self):
28          self.head = None  # First node of linked list
29          self.tail = None  # Last node of linked list
30          self.length = 0
```

Print Linked List:

```
32      # Print the entire linked list
33      def print_llist(self):
34          temp = self.head
35          num = 0
36          while temp:
37              if num == 0:
38                  print('Head', num)
39              else:
40                  print('Node ', num)
41              print(' Password: ', temp.password, 'Count: ', temp.count, '\n')
42              temp = temp.next
43              num += 1
```

Print first 20 elements on linked lists:

```
45      # Print only the first 20 nodes
46      def print_20(self):
47          temp = self.head
48          num = 0
49          while num < 20:
50              if num == 0:
51                  print('Head', num)
52              else:
53                  print('Node ', num)
54              print(' Password: ', temp.password, 'Count: ', temp.count, '\n')
55              temp = temp.next
56              num += 1
```

```
109 # descending order bubble sort
110 def bubble_sort(unsorted):
111     # After the first iteration, the smallest count will be at the end
112     # This is the last element we will check; it will get smaller
113     last_element = None
114     is_sorted = True
115
116     # We will start with the maximum being the last node up until we reach a maximum of the first node
117     while last_element is not unsorted.head:
118         prev = None  # Keep variable for the previous node; prev will be initially zero since head has not previous
119         first_node = unsorted.head  # Starting point
120
121         # This is where we will be swapping nodes
122         while first_node.next is not last_element:
123             # Set the second node to be the node after the first one
124             second_node = first_node.next
125
126             # We will sort in descending order (5,4,3,2,1 ....)
127             if first_node.count < second_node.count:
128                 is_sorted = False  # Since a swap happened, the linked list is still not sorted
129                 # Swapping:
130                 first_node.next = second_node.next
131                 second_node.next = first_node
132                 # After switching nodes, remember to connect previous node to the newly switched node
133                 if prev is not None:
```

```
132                 # After switching nodes, remember to connect previous node to the newly switched node
133                 if prev is not None:
134                     # Update the prev node (it's the equal to the second node because a swap happened)
135                     prev.next = second_node
136                 else:
137                     # If a prev hadnt been set yet, then update head every time head is switched
138                     unsorted.head = second_node
139
140                 # Update the first node and second note for the next iteration
141                 first_node, second_node = second_node, first_node
142
143                 # Update tail
144                 if second_node.next is None:
145                     unsorted.tail = second_node
146
147             # updating prev and first node (next iteration)
148             prev = first_node
149             first_node = first_node.next
150
151         # Updating the last element
152         last_element = first_node
153         # This will prevent iterating several times if linked list is already sorted
154         if is_sorted:
155             break
156     return unsorted
```