**Introduction**: For this Lab, I need to be able to use the data from two text files that holds the amount of user’s data. I have to insert these into a linked list and then sort them with the merge and bubble sort.

**Proposed** **Solution**: For bubble sort the possible solutions to make this algorithm possible was to be able to find a way to compare two values and make the program to compare two values and decide if they need to be switched if one of the values that comes before them in order is greater than the value next to it. However, this program needs to use linked lists instead of a python native list due to accomplish the requirement of sorting a linked list with bubble sort.

For this Merge sort, I knew that I first needed to split my list by half which can be considered as the left and right list, I have to do this in order to follow the rules of merge sort. However, after splitting my list, I would need to keep splitting my list until it reaches the limit of being unable to split my list anymore. After the entire list becomes split, then I would have to compare my two values that are adjacent to each other and if they are not in order then I would switch the positions they are in. Even if they switch or not I would have to merge the two values into a list, when we return that list then I’m going to have another list that either went through the same process or not, but I would have to compare the values of my two merged lists and put them in order and then merge it into another sorted list. The comparisons and merging would only stop after the entire list is sorted and when we only have one merged sorted list remain.

**Implementation**: When creating the bubble sorting algorithm that will sort a linked list, I first needed to apply the linked list algorithms that were given in class in order to be able to use linked list and its functions. The first thing that I had to do was to be able to keep traversing through the linked list until every value in the list is sorted. In order to do that I needed to make a while loop that contains a value named “change” that is declared true; the value “change” would traverse the list until the value becomes false which means that no more changes in the sorting algorithm is possible. The next part of the program would be to follow a basic rule which is to never lose the head of the linked list, which is done by storing my head in a temporary variable which keeps my original list from being changed. After finding a way to not lose my head, I would then need to declare my “change” value as false which comes into play later on in the code. After that I need to traverse my linked list until the next value in my temporary list is None. After finding a way to traverse my temporary list, I had to make my algorithm to compare my current item in the list to the one next to it and if the current item was larger than I have to switch my values position in the list.

When finding out that if my current item is larger than the one next to it then I had to create a code that switches the position only between those two items. The way that the items where switched was to store my current item into another temporary variable to store the value for the meantime. Next, I needed to affect my temporary linked list by declaring the current item to the one that was being compared to it in order to make the changed item into my current item. Now we need to use the current item that was stored in my temporary variable and assign it to the position that is next to out new current item which will now change the temporary list by switching both items. After switching then we declare the value “change” as true to show that there was a switch done and next, we need to traverse to the next item on the list where we continue from the current item we had. In order to create my merge sort algorithm then I would need to have 3 method, one to pass the list and call the functions to split and merge the sorted list. The second method will be to divide the list by half until no more splits are possible. The final method is to compare the values in the list and then merge them in to a new sorted list.

To begin with the first method, I had to make a base case to make sure that if the list is empty or if it only has 1 variable then the method would return the list we have since there is no need to sort it. If that base condition doesn’t apply to the list, then we continue to create a variable called middle which would call the method split and splits the list. Since the middle variable is going into the method split, I would then have to create the split method before anything else in the merge sort algorithm. In my split method I created two variables one called fast which is storing the next node of the original list and the other variable is called slow which holds the list. The purpose of these two variables is that they will actually be our pointers that will get the middle position of the lists. Once the two variables have been declared I would need the pointers to go through a while loop which will allow the fast pointer to move two positions ahead and move the slow pointer by one. In the while loop I had to make the fast pointer iterate through the list until it reaches the end of the linked list. Inside the while loop the fast pointer will point at the next node in its list and then there will be an if statement that if fast is not None then the next following actions will be taken. In the if statement, slow is now pointing at the next node in its own linked list and fast points at the next node in the next node again. Until our fast variable reaches the end of the linked list, we will be able to exit the list and return the slow variable from where it last pointed.

Back to the first method, after declaring the variable middle to the list we returned from the split method I had to make another variable called next Middle which will point at the value after the middle which will be used to get the right side of the list. After declaring the previous variable, I had to assign the node next to my middle value as None, so it gets rid of its remaining list. Now I created two variables that are called left and right which the purpose of them is to get the two split lists and pass them through the method that we’re in recursively. Since the lists that we have are being passed recursively then it would go to through the same process that I have explained and will continue to split those lists until it’s not possible anymore. However, I declared a value that is called sorted List which calls the method where it compares my values, sorts them in order and merges them by inserting them in a brand-new sorted list.

In my sorting method I have to create an empty variable called Result which its purpose is to store the new sorted linked list and return it at the end. After that, I have to make base cases which will check if my left list is empty then it returns the right side of the list only and vice versa for when the right side is empty. After checking that then I had to compare my left sorted linked list with the right sorted linked list and it was compared as if the value of the left side is less than or equal to the value at the right linked list the result will hold the item of the left list. Then the result value needs to store its next node and in order to do that it has to call the sorting method recursively and pass the next node of the left side and pass the right side of the list, but without making any changes to it. However, if the left link list item is greater than the value in the right link list then the Result will store the right link list item in its current Node and then call the sorting method recursively and pass an unaffected left link list and passes the next position of the right link list. This process will be passed until no more merging is possible and will return a sorted merge sort linked list .

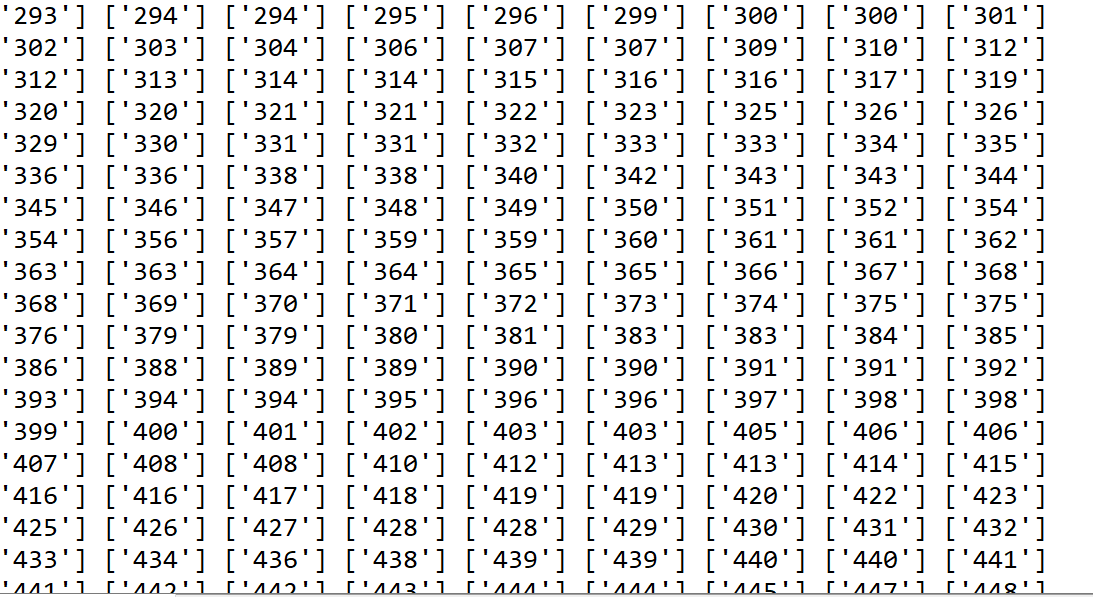
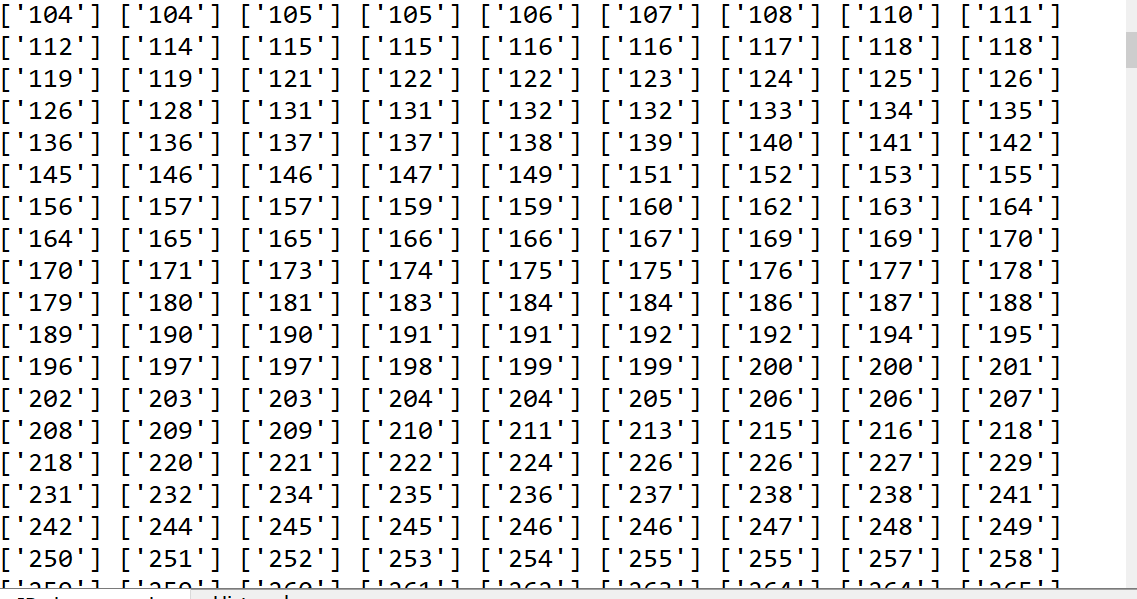
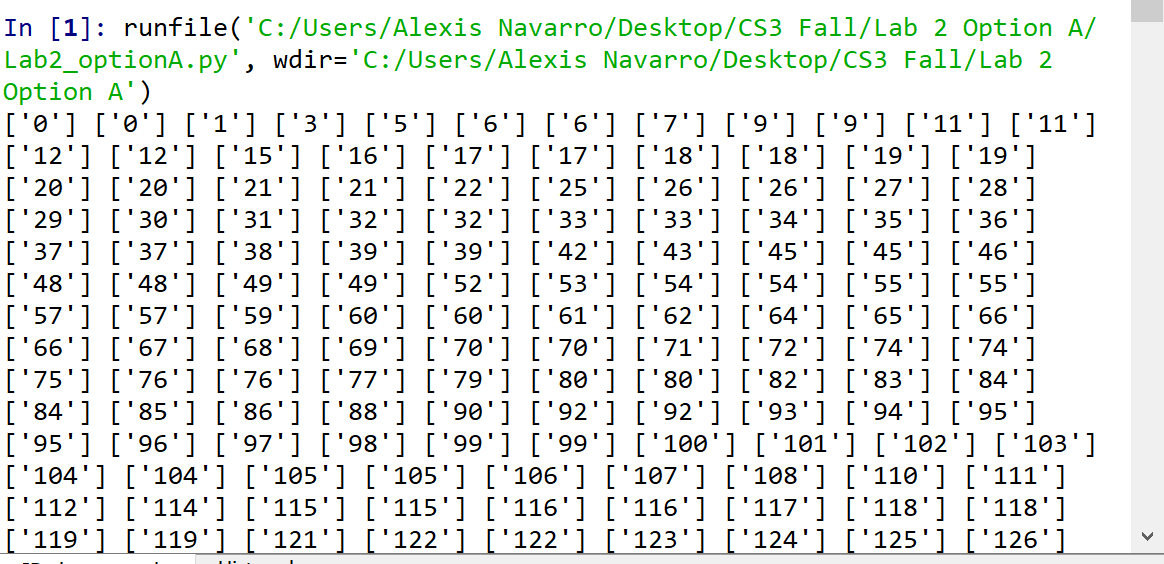
**Errors:** Had encountered many errors when using sorting such as maximum recursion depth reached. Merge sort not sorting even when it runs and compiles.

**Conclusions:**

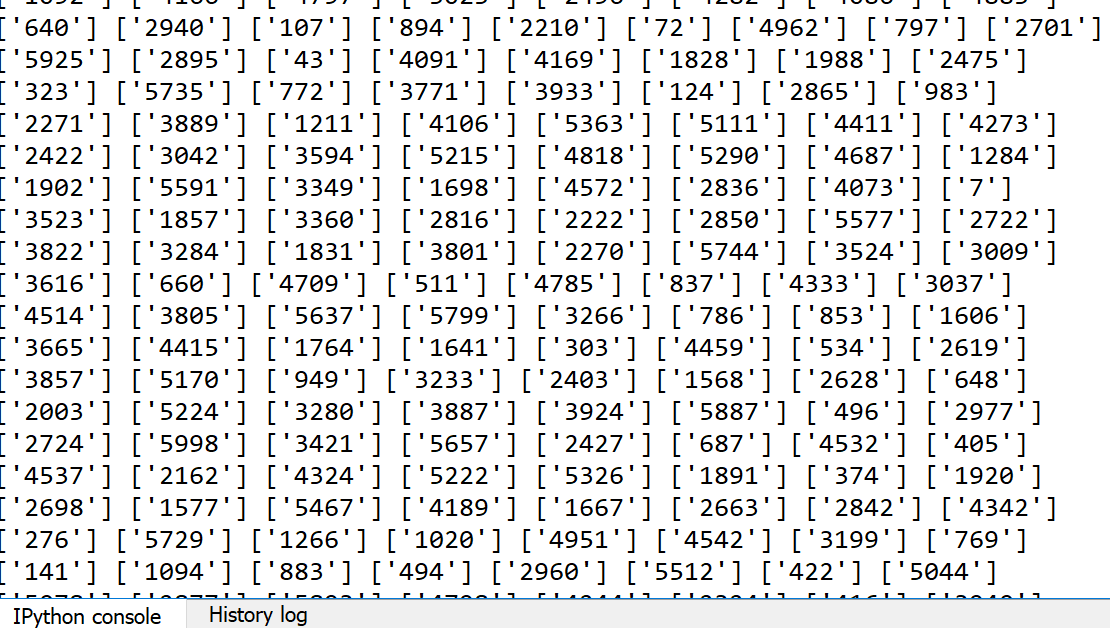
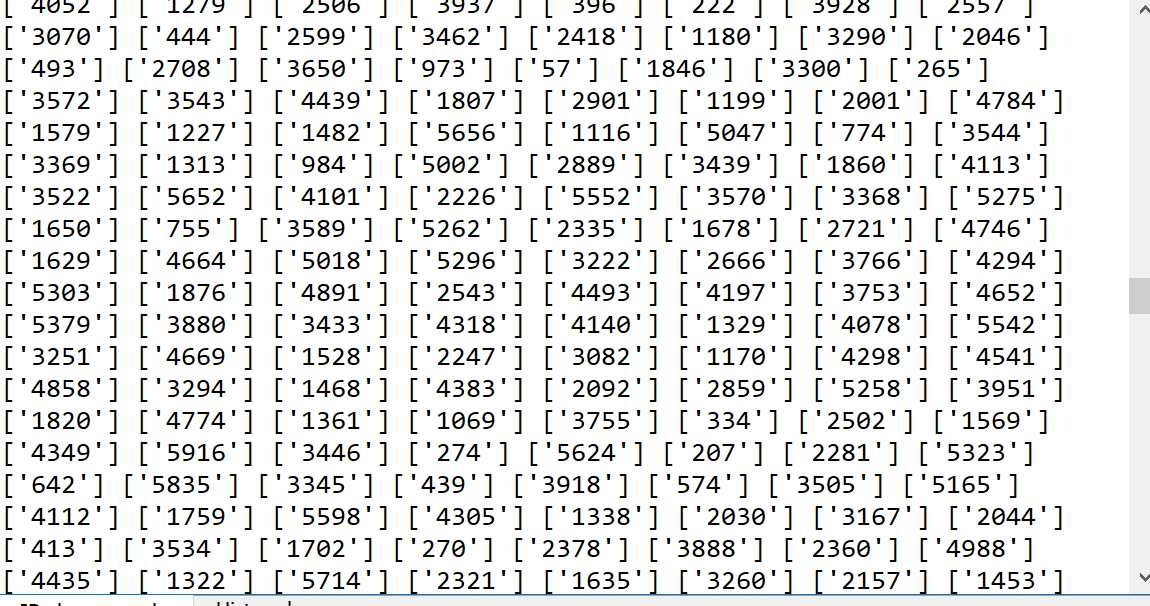
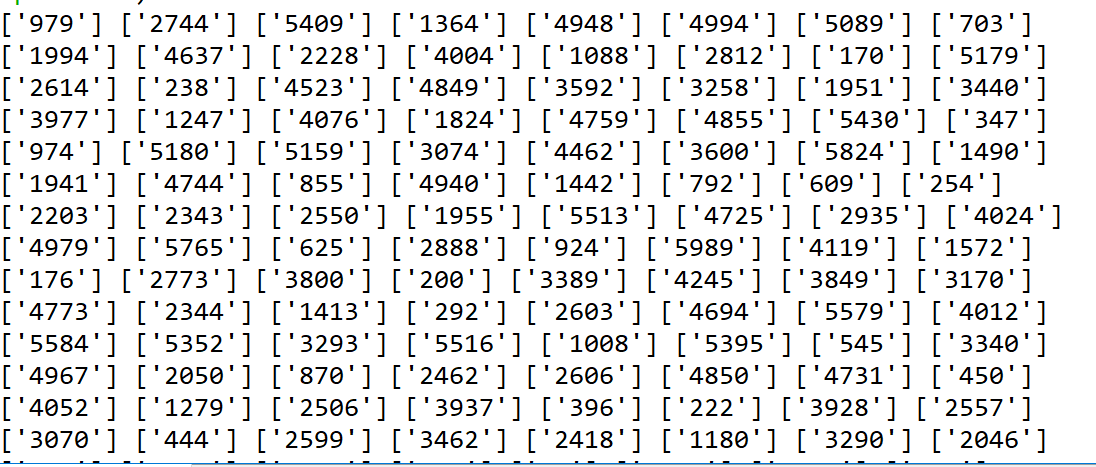
**Bubble sort:**

Big O:

O(n^2)



**Merge sort:**

****

BigO:

O(n log(n))

Academic dishonesty includes but is not limited to cheating, plagiarism and collusion. Cheating may involve copying from or providing information to another student, possessing unauthorized materials during a test, or falsifying data (for example program outputs) in laboratory reports. Plagiarism occurs when someone represents the work or ideas of another person as his/her own. Collusion involves collaborating with another person to commit an academically dishonest act.

Professors are required to - and will - report academic dishonesty and any other violation of the Standards of Conduct to the Dean of Students.

**Results**:

# -\*- coding: utf-8 -\*-

"""

Created on Wed Sep 18 19:30:50 2019

@author: Alexis Navarro

CS2302 - Lab 2 Option A

MW 1:30-2:50 PM

Professor: Diego Aguirre

Purpose: To create a linked lists with data from a text file and understand sorting with the

"""

class Node(object):

def \_\_init\_\_(self, item, next=None):

self.item = item

self.next = next

class List(object):

def \_\_init\_\_(self):

self.head=None

self.tail=None

def is\_empty(L):

return L.head==None

def Append(L,x):

if is\_empty(L):

L.head = Node(x)

L.tail = L.head

else:

L.tail.next = Node(x)

L.tail = L.tail.next

def Print(L):

temp = L.head

while temp is not None:

print(temp.item, end=' ')

temp = temp.next

print()

#-------------------------------------------------------------------------------------------------------------------------------------------------

'''

def compare(employees):

duplicate\_list = List()

temp1=employees

while temp1 != None:

temp2 = temp1.next

while temp2 != None:

if temp1.item == temp2.item:

duplicate\_list.append(temp1.item)

temp2= temp2.next

temp1= temp1.next

return employees

'''

'''

def compare(L):

dupe\_list = List()

if L.head.item == L.next.item:

dupe\_list.append(L.head.item)

else:

compare(L.next)

return dupe\_list

'''

#working bubble sort

def bubble\_sort(L):

change\_items=True #set this to true

while change\_items:

t = L.head #make a temp to store the head

change\_items=False

while t.next is not None:

if int(t.item[0]) > int(t.next.item[0]):

temp = t.item

t.item = t.next.item

t.next.item = temp

change\_items = True

t = t.next

#-------------------------------------------------------------------------------------------------------------------------------------------------

'''

#merge sort that did not work due the error maximum depth reached

def merge\_sort(L):

if L == None or L.next == None:

return L

middle = split(L) #calls the split method to split the linked list by the middle

next\_middle = middle.next

middle.next = None

left = merge\_sort(L)

right = merge\_sort(next\_middle)

sorted\_list = sorted\_merge(left, right)

return sorted\_list

def split(L):

fast = L.next #fast and slow refer to the pointer ( slow will always be 1 item behind the fast pointer)

slow = L #stores the entire list/ start of the list

while fast != None: # fast will move up the linked list by two

fast = fast.next

if fast != None:

slow = slow.next # moves slow by 1 till it gets to the middle postion

fast = fast.next

return slow

def sorted\_merge(left\_side,right\_side):

Result=None#List()

if left\_side is None:

return right\_side

if right\_side is None:

return left\_side

if left\_side <= right\_side:

Result = int(left\_side.item[0])

Result.next = sorted\_merge(left\_side.next, right\_side)

else:

Result = right\_side

Result.next = sorted\_merge(left\_side, right\_side.next)

return Result

'''

#Merge sort that does not sort correctly

def Merge(L):

if is\_empty:

return None

else:

Split(L)

print(L)

#split method that splits the linked list in the middle

def Split(L):

temp=L.head

middle=len(L)//2

left=L[middle:]

right=L[:middle]

i=0#tracking for L1

j=0#tracking for L2

k=0#tracking for entire L

while temp is not None:

while i<len(left) and j<len(right):#if both split lists are less than its length

if left[i] < right[j]:#if its value is bigger, then will make compare/sort

L[k]=right[j]

i+=1

else:

L[k]=right[j]

j+=1

k+=1

while i<len(left):#if L1 index less than length

L[k]=left[i]

i+=1

k+=1

while j<len(right):#if L2 index less than length

L[k]=right[i]

j+=1

k+=1

return L

#-------------------------------------------------------------------------------------------------------------------------------------------------

def main():

#Normal way to append and read the file

'''

file\_info=[]

file = open('activision.txt','r')

line = file.readline()

while line!='':

line = line.strip()

parts=line.split(' ')

file\_info.append(parts)

line=file.readline()

print(file\_info)

'''

#Append with linked list

file\_info=List()

with open("activision.txt","r") as i:

for line in i:

line = line.strip()

parts=line.split(' ')

Append(file\_info,parts)

#Print(file\_info)

with open("vivendi.txt","r") as j:

for line in j:

line = line.strip()

parts=line.split(' ')

Append(file\_info,parts)

#print("complete: ")

#Print(file\_info) #check to see if the items from txt file are in the linked list

#compare(file\_info)

bubble\_sort(file\_info)

#Merge(file\_info)

Print(file\_info)

main()