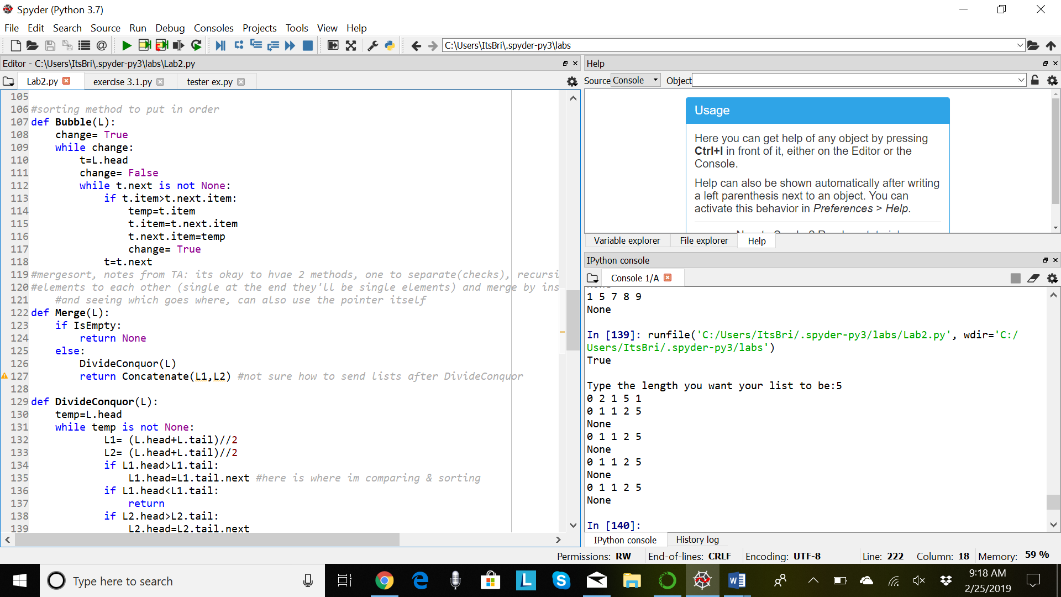
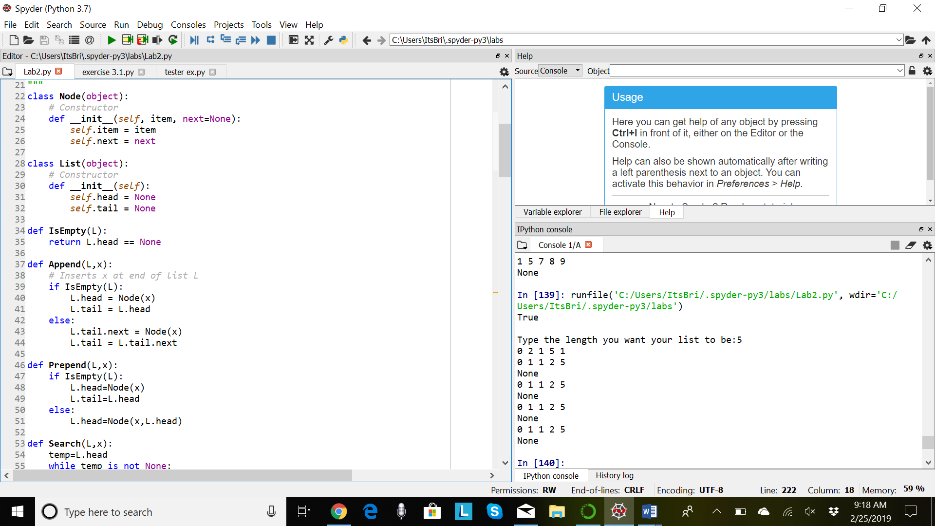
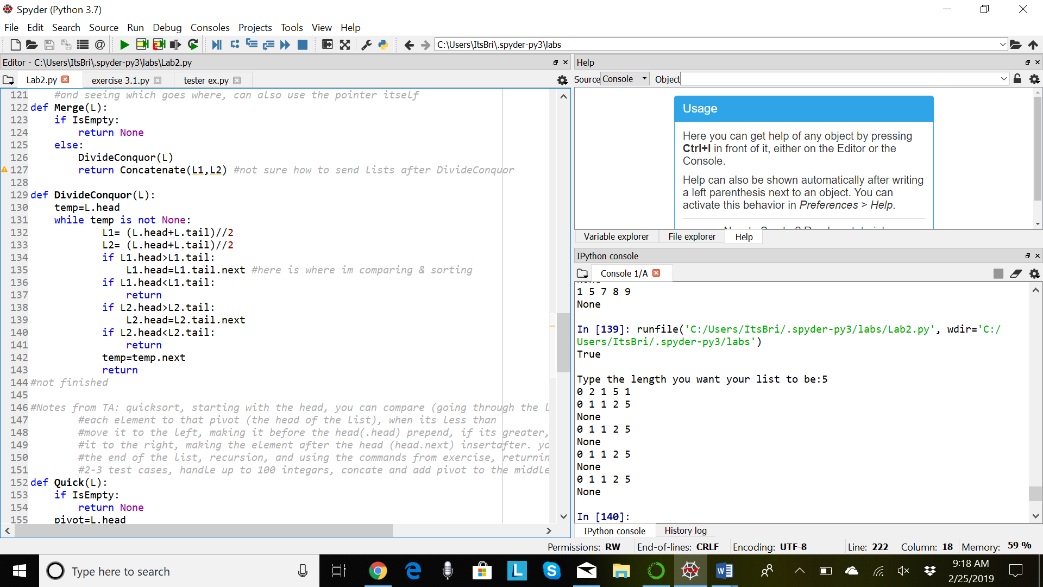
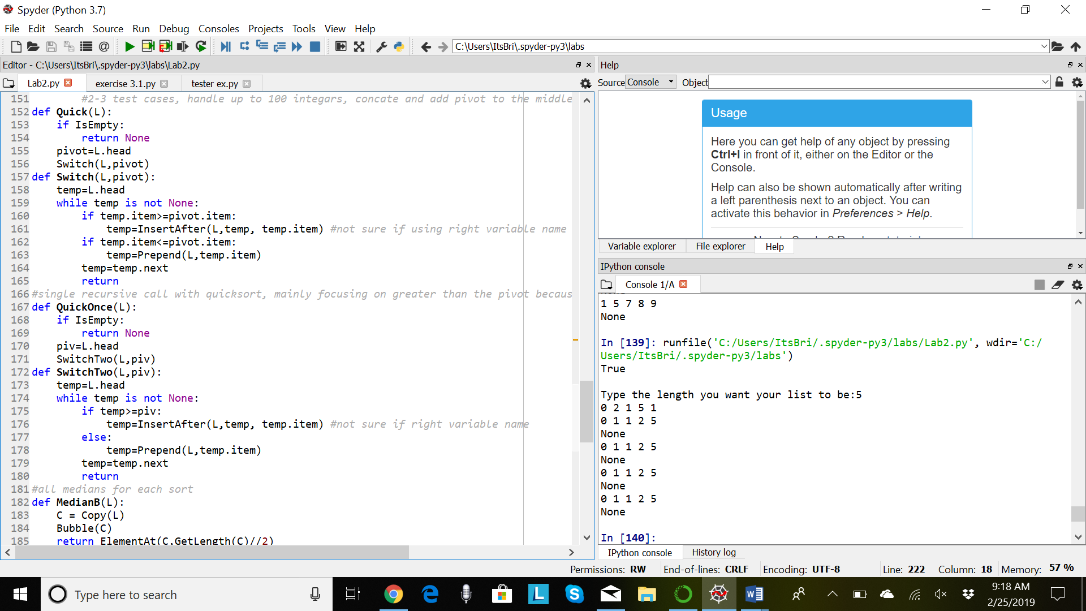
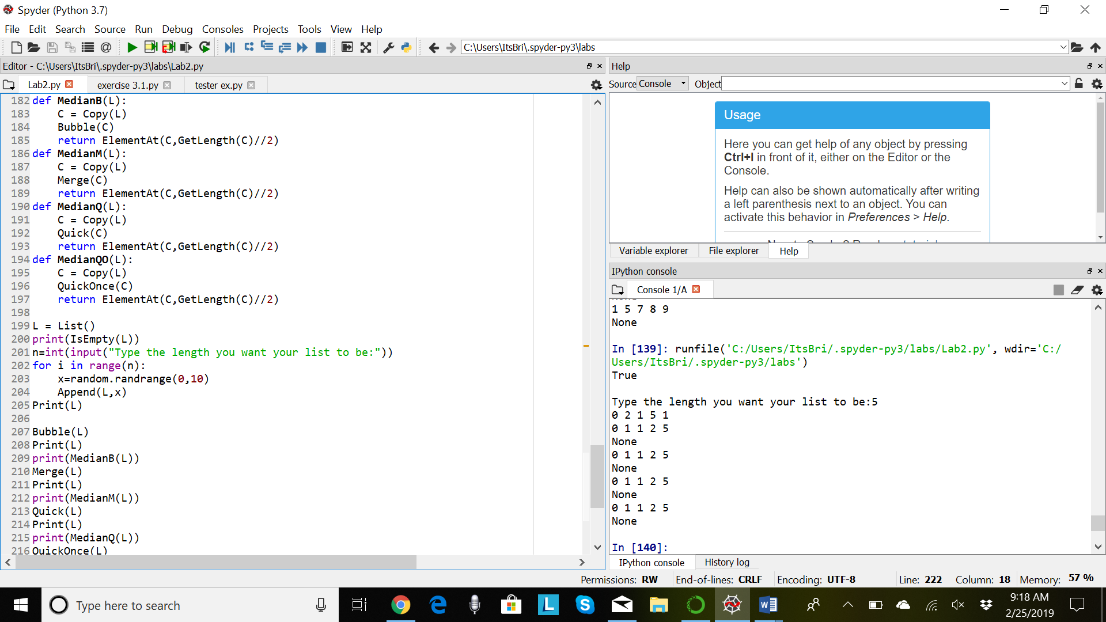
**Lab 2 Report: Sorting Linked Lists and Medians**

Starting out my program, I used almost all code given from our exercises during class, and fellow classmates help we did together during class on Prepend, InsertAfter, etc. This helped keep my code neatly together while not having such long methods for my sorting algorithms while calling these other methods. Also, to note that I have both my constructors for my object of Lists and Nodes. This screenshot pertains my first method of Bubble Sorting. This sorting algorithm wasn’t hard to figure out since Dr. Fuentes was kind enough to show us this code during class. It cleared up many questions I had on how to start this lab and help me lead to my next sorting algorithms.My next sorting algorithm of Merge sort was a bit difficult to figure out. During the week, I visited Ms. Dita which helped cleared up many doubts I had about how to start coding this algorithm. For this sort, I had to make a separate method to help me not just organize but separate where I am dividing and comparing my list elements, then putting them back together in the original method to return the completed, sorted list.These last two methods were a challenge. They both were of Quick Sorting, but the second method named QuickOnce needed only one return call. These algorithms also required a separate method with them because besides the choosing of the pivot, I had to separate the list’s elements into left or right of my pivot and then return the list after each element has been put in its position. This, as my other methods needed, were the exercise methods we had in class that help kept my code clean and efficient. Unfortunately, I was not able to correctly find the medians of each.I put my list random generator at the end of my code since it was the first method I created, needed it first, and I also needed it out of the way after I got it figured out since I wouldn’t have to look at it unless there were errors leading from it. Here, is where I added four separate methods of Medians for each sorting algorithms to find each of their medians they produced. I started with one median method, see if it fitted for each algorithm, then coded four of them for each. My code complied, after many hours of errors, and kept returning None instead of an element from that list. There was a point in which I was able to produce a median number by simply grabbing the length, dividing my 2, then returning the element item in that position but each algorithm would’ve had the same number. It wouldn’t have mattered. By changing my algorithm specifically to how the lab intended its use, my program was not efficient. The sorting algorithms worked, but the medians weren’t returned correctly.

|  |  |  |  |
| --- | --- | --- | --- |
| Bubble Sorting | Merge Sorting | Quick Sorting | QuickOnce Sorting |
| O(n) | O(log n) | O(n) | O(n) |

My running times for each of my algorithms are shown above. Bubble sorting has n for Big O since when going through its sorting, its taking each element and switches or compares each. In Merge, this is different since its dividing the list into separate list of singly element lists, then combining them in the end after being compared to each other. This is Log n since we are dividing. The same explanation for Bubble sorting goes for Quick sorts. They go through each element and place the element in their corresponding position.

Lastly, I wasn’t able to add the count for how many comparisons each method provides to sort their list since besides running out of time, I had forgotten to add count+=1 in each method to count the number they had. I did, however, have different lists for test cases, but the user can input whichever length of list they want and get a randomly generated list with numbers ranging 1-10 for the value of each element. The user can run their own tests through it.

Appendix:

import random

class Node(object):

# Constructor

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

self.item = item

self.next = next

class List(object):

# Constructor

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

self.head = None

self.tail = None

def IsEmpty(L):

return L.head == None

def Append(L,x):

# Inserts x at end of list L

if IsEmpty(L):

L.head = Node(x)

L.tail = L.head

else:

L.tail.next = Node(x)

L.tail = L.tail.next

def Prepend(L,x):

if IsEmpty(L):

L.head=Node(x)

L.tail=L.head

else:

L.head=Node(x,L.head)

def Search(L,x):

temp=L.head

while temp is not None:

if temp.item==x:

return temp

temp=temp.next

return None

def InsertAfter(L,x,item): #correct method

s=Search(L,x)

if s is None:

Append(L,x)

else:

s.next=Node(item,s.next)

def Concatenate(L1,L2):

#appends list 1 to 2

if IsEmpty(L1):

L1.head=L2.head

L1.tail=L2.tail

else:

if IsEmpty(L2):

L1.tail=L2.head

else:

L1.tail.next=L2.head

def Print(L):

temp = L.head

while temp is not None:

print(temp.item, end=' ')

temp = temp.next

print() # New line

def GetLength(L):

temp=L.head

count =0

while temp is not None:

count+=1

temp=temp.next

return count

def Copy(L):

temp=L.head

new=List()

while temp is not None:

Append(new,temp.item)

temp=temp.next

return new

def ElementAt(L,x):

Search(L,x)

return

#sorting method to put in order

def Bubble(L):

change= True

while change:

t=L.head

change= False

while t.next is not None:

if t.item>t.next.item:

temp=t.item

t.item=t.next.item

t.next.item=temp

change= True

t=t.next

#mergesort, notes from TA: its okay to hvae 2 methods, one to separate(checks), recursion check(comparing each

#elements to each other (single at the end they'll be single elements) and merge by insertafter after comparing

#and seeing which goes where, can also use the pointer itself

def Merge(L):

if IsEmpty:

return None

else:

DivideConquor(L)

return Concatenate(L1,L2) #not sure how to send lists after DivideConquor

def DivideConquor(L):

temp=L.head

while temp is not None:

L1= (L.head+L.tail)//2

L2= (L.head+L.tail)//2

if L1.head>L1.tail:

L1.head=L1.tail.next #here is where im comparing & sorting

if L1.head<L1.tail:

return

if L2.head>L2.tail:

L2.head=L2.tail.next

if L2.head<L2.tail:

return

temp=temp.next

return

#not finished

#Notes from TA: quicksort, starting with the head, you can compare (going through the list), pivot is head

#each element to that pivot (the head of the list), when its less than

#move it to the left, making it before the head(.head) prepend, if its greater, then move

#it to the right, making the element after the head (head.next) insertafter. you keep going until

#the end of the list, recursion, and using the commands from exercise, returning median too

#2-3 test cases, handle up to 100 integars, concate and add pivot to the middle

def Quick(L):

if IsEmpty:

return None

pivot=L.head

Switch(L,pivot)

def Switch(L,pivot):

temp=L.head

while temp is not None:

if temp.item>=pivot.item:

temp=InsertAfter(L,temp, temp.item) #not sure if using right variable name

if temp.item<=pivot.item:

temp=Prepend(L,temp.item)

temp=temp.next

return

#single recursive call with quicksort, mainly focusing on greater than the pivot because the less than will stay the same

def QuickOnce(L):

if IsEmpty:

return None

piv=L.head

SwitchTwo(L,piv)

def SwitchTwo(L,piv):

temp=L.head

while temp is not None:

if temp>=piv:

temp=InsertAfter(L,temp, temp.item) #not sure if right variable name

else:

temp=Prepend(L,temp.item)

temp=temp.next

return

#all medians for each sort

def MedianB(L):

C = Copy(L)

Bubble(C)

return ElementAt(C,GetLength(C)//2)

def MedianM(L):

C = Copy(L)

Merge(C)

return ElementAt(C,GetLength(C)//2)

def MedianQ(L):

C = Copy(L)

Quick(C)

return ElementAt(C,GetLength(C)//2)

def MedianQO(L):

C = Copy(L)

QuickOnce(C)

return ElementAt(C,GetLength(C)//2)

L = List()

print(IsEmpty(L))

n=int(input("Type the length you want your list to be:"))

for i in range(n):

x=random.randrange(0,10)

Append(L,x)

Print(L)

Bubble(L)

Print(L)

print(MedianB(L))

Merge(L)

Print(L)

print(MedianM(L))

Quick(L)

Print(L)

print(MedianQ(L))

QuickOnce(L)

Print(L)

print(MedianQO(L))

"""

#Test Cases

L = List()

print(IsEmpty(L))

for i in range(5):

Append(L,i)

Print(L)

L=List()

L=[]

L=[1,7,2,2,5]

L=[4]

L=[10,8,2,4,1]

"""