Jon Munoz

Lab 2

CS2302

Professor: Olac Fuentes

**INTRODUCTION**

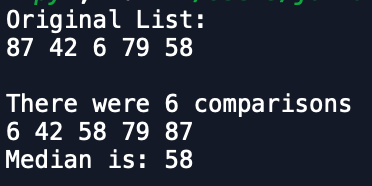
For this lab we were assigned the task of making four different sorting algorithms: bubble sort, merge sort, quick sort, and a modified version of quick sort that only made one recursive call instead of the normal two. Once we made the sorting algorithms, we were then tasked with counting how many comparisons were made in each method for a given linked list and return the middle value of the sorted list. Along with these methods we had to make a method that would create a list of random integers in the range that the user input.

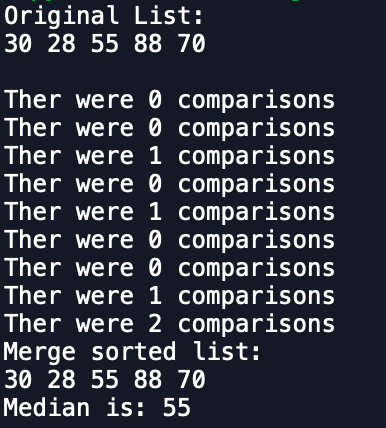
**PROPOSED SOLUTION DESIGN AND IMPLEMENTATION**

For these different algorithms I had to try implementing different designs. For bubble sort it was quite simple since it was not done recursively. For that algorithm I made a value to keep track of if a change was made and made two while loops, one keeping track of the change value and one keeping track of if the passed list at a current node was valid. In the second while loop there was also an if statement for if there was a change that needed to be done it would make the change and then have the change value set to true. This went on for the entire list until the change value was false. For merge sort what I attempted to do was have a while loop to put half of the elements from the original and once I had those two split up list I would then pass those list into another method called merge that would order the two passed list and then return it at the end of the method. This to me in theory made sense but my code did not work out as I planned. The final output was an “ordered” list in a way since it ordered the original two list which was not the desired outcome. I will demonstrate the output in the next paragraph. For quicksort I attempted to have the head item of the list be the pivot and then compare each element of the original list to the pivot and based on if they were greater to or less than the pivot then id either out them in a list called lessList or moreList. After I was done creating these sub lists I would then recursively call the quicksort method with each of them. After all the recursive calls were completed, I assumed that they would be ordered and then I would concatenate them, but this did not work for me and I did not get the desired outcome. The modified quicksort I could not implement into code since I could not complete the base version of quicksort, and how could I make a modified version if I couldn’t even get the base version, but with the modified version of quicksort you were only supposed to recursively call the list with the median of the sorted list. The method that made the list was quite simple. All I did was make a for loop that was in the range that the user inputted (for example a value of 4 being passed would create a list of length 4). The median methods would just return the values of the sorted lists at the “index” of the length of each list divided by 2.

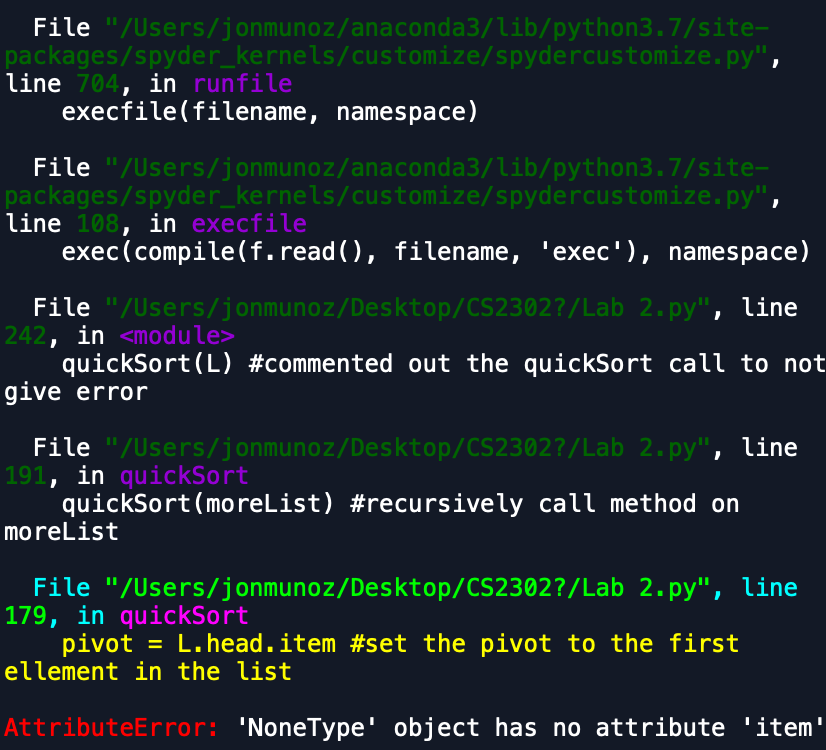
**EXPERIMENTAL RESULTS**

My results for my different sorting methods were not what I was trying to get. For bubble sort the method worked perfectly which is the only one that worked. For my test I made a list of length 5 with values between the range of 1 and 100 and this was the outcome:



As you can see the method sorted the original list correctly and also correctly returned the middle value and number of comparisons. For merge sort my method did not work very well. I tried the same set of parameters for my merge sort method and this was the result:

The list was sorted with the first half of the list being 30, 28 and the second half being 55,88,70 and the merge method sorted them according to those two list. So since 30 < 50 the new list contained 30 and then it compared the next two values 28 < 55 so the list then contained 30,28 and it then added the remaining list to the sorted list hence the final list of 30,28,55,88,70. This however is incorrect. For quick sort I do not know why but I kept getting the same error with every call I made to it. The error it gave to me was:



I did not know why this was the case because once there was a None value the while loop should not have executed and it should have returned to the previous recursive calls. This error I could not figure out so the method is incomplete and not functioning.

**CONCLUSION**

At the end of this assignment I understood the algorithms and steps necessary for the different sorting methods but not necessarily how to code them. It also taught me the way that the different sorting methods do their sorting which I did not completely understand at the beginning of the assignment.

**APPENDIX**

**SOURCE CODE**

#Jon Munoz

#CS2302 Data Structures

#Lab 2

#Instructor:Olac Fuentes

#TA:Anindita Nath

#Last Modified 2/22/19

#incomplete

import random

#Node Functions

class Node(object):

# Constructor

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

self.item = item

self.next = next

def PrintNodes(N):

if N != None:

print(N.item, end=' ')

PrintNodes(N.next)

def PrintNodesReverse(N):

if N != None:

PrintNodesReverse(N.next)

print(N.item, end=' ')

#List Functions

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 Print(L):

# Prints list L's items in order using a loop

temp = L.head

while temp is not None:

print(temp.item, end=' ')

temp = temp.next

print() # New line

def Remove(L,x):

# Removes x from list L

# It does nothing if x is not in L

if L.head==None:

return

if L.head.item == x:

if L.head == L.tail: # x is the only element in list

L.head = None

L.tail = None

else:

L.head = L.head.next

else:

# Find x

temp = L.head

while temp.next != None and temp.next.item !=x:

temp = temp.next

if temp.next != None: # x was found

if temp.next == L.tail: # x is the last node

L.tail = temp

L.tail.next = None

else:

temp.next = temp.next.next

#BubbleSort method

def BubbleSort(L):

change = True #change keeps track if there was a change or not

counter = 0 #counter is used to keep track of how many comaprisons there were

while change: #while loop to keep the code going while change is True

t = L.head

change = False #set change to false so once the second while loop finishes change is no longer true

while t.next is not None: #while loop too keep going while t is not empty/none

if t.item > t.next.item: #check to see if adjacent values are in order

counter += 1 #iterate counter once since a comparison is being done

temp = t.item #create a temp in order to to hold the value of t.item and not lose it

t.item = t.next.item

t.next.item = temp #replace t.next.item with the original value of t.item

change = True #a change was made so change the value of change to true

t = t.next #iterate through t

return counter

#method makes a copy of a given list in order to make changes without changing original list

def copy(L):

temp = L.head #

newList = Node(0) #creates a new empty list

newList.head = temp

while temp is not None: #while loop poulates the newList (AKA the copy)

temp = temp.next

newList.next = temp

return newList #return the copy

#gets the length of a given list

def getLength(L):

temp = L.head

count = 0

while temp is not None:

temp = temp.next

count += 1

return count

#this method merges the 2 passed list

def merge(L1, L2):

newList = List() #new list is the result of merging the list

temp1 = L1.head #"copy" of L1

temp2 = L2.head #"copy" of L2

counter = 0 #counter for comparisons

while temp1 is not None and temp2 is not None:

#below are different conditions on where and when to add to the newList

if temp1.item < temp2.item: #if temp1.item is less than temp2.item then we want to add temp1.item to the newList

counter += 1

Append(newList, temp1.item)

temp1 = temp1.next #iterate through temp1

else: #if temp2.item is less than temp1.item then we want to add temp2.item to the newList

counter += 1

Append(newList, temp2.item)

temp2 = temp2.next #iterate through temp2

if temp1 is None: #if temp1 is empty then you want to add whatever is left in temp2 into the newList

while temp2 is not None:

Append(newList, temp2.item)

temp2 = temp2.next

if temp2 is None: #if temp2 is empty then you want to add whatever is left in temp2 into the newList

while temp1 is not None:

Append(newList, temp1.item)

temp1 = temp1.next

if temp1 is None and temp2 is None:

print("Ther were", counter, "comparisons") #print the number of comparisons

return newList

#mergeSort method

def mergeSort(L):

C = copy(L) #create a copy of the passed list to modify

halfPoint = getLength(C)//2 #get the middle index and assign it to halfPoint

leftList = List() #leftList will be half of the original list

sortedList = List() #sortedList will be the

if getLength(L) > 1:

for i in range(halfPoint): #use this loop to go halfway through the original loop

Append(leftList, C.head.item) #add the first half of the original list into leftList

C.head = C.head.next #iterate through C

mergeSort(C) #recursively call mergeSort on whats left of C

mergeSort(leftList) #recursively call mergeSort on what is in leftList

sortedList = merge(C, leftList) #merge C and leftList

return sortedList #return the sorted list, however with my code the list does not get sorted, the logic makes sense to me but does not work

#method gives you the item at a desired element

def ElementAt(L, index):

temp = L.head

#loop iterates to the desired position

for i in range(index):

temp = temp.next

return temp.item

#puts two list together

def concatenate(L1,L2):

#if L1 is empty then you want to change its pointers to L2's pointers

if IsEmpty(L1):

L1.head = L2.head

L1.tail = L2.tail

else: #change L1's pointers to the end and start of the second list

L1.tail.next = L2.head

L1.tail = L2.tail

#quickSort Method

def quickSort(L):

pivot = L.head.item #set the pivot to the first ellement in the list

temp = L.head #make a "copy" in order to modify

lessList = List() #list contains elements less than the pivot

moreList = List() #list contains elements more than the pivot

sortedList = List() #list should contain the sorted elements

while temp is not None: #iteratre through temp until its empty

if temp.item > pivot:

Append(moreList, temp.item) #add the item that is greater than the pivot to moreList

temp = temp.next #iterate

else:

Append(lessList, temp.item) #add the item that is less than the pivot to lessList

temp = temp.next #iterate

quickSort(moreList) #recursively call method on moreList

quickSort(lessList) #recursively call method on lessList

sortedList = concatenate(moreList, lessList) #set sortedList to the concatination of more and less list

return sortedList #return the sorted list but in my code I get an error saying None has no item

#get the median value of the bubble sorted list

def Median(L):

C = copy(L)

print("There were", BubbleSort(C), "comparisons")

Print(C)

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

#get the median of the merge sorted list

def Median2(L):

C = copy(L)

S = mergeSort(C)

print("Merge sorted list: ")

Print(S)

return ElementAt(S, getLength(C)//2)

def Median3(L):

C = copy(L)

S = quickSort(C)

print("Quick sorted list: ")

Print(S)

return ElementAt(S, getLength(C)//2)

L = List()

#method creates a linked list with randomly generated integers with n length

def makeList(L, length):

for i in range(length):

t = random.randint(1,100)

Append(L,t)

return L

makeList(L, 5)

print("Original List: ")

Print(L)

print()

#print("Median is:", Median2(L))

print()

#print("Bubble sorted list: ")

#print("This sorting had", BubbleSort(L), "comparisons")

#Print(L)

quickSort(L) #commented out the quickSort call to not give error

“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 provide inappropriate assistance to any student in the class.”

