Marco Valerio

Lab Report 2

CS 2302

Olac Fuentes

02/26/2019

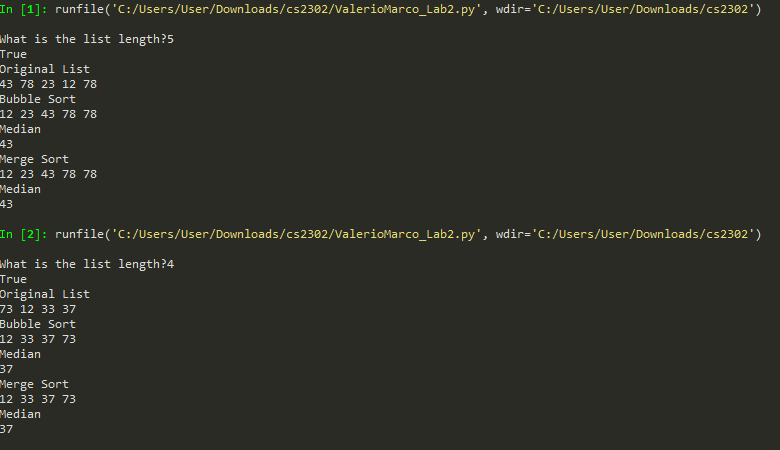
***Introduction:*** The goal for lab 2 was for the student to implement several algorithms to find the median of a List of size ‘n’ containing randomly generated numbers using four different sorting methods. The methods included bubble, quick, modified quick and merge sort; each method should return the element in the middle.

***Proposed Solution design and Implementation:***

* Creating the List
  + In order to create a list of size ‘n’ with randomly generated number I used a modified version of the “Append()” method called “AppendRandom()” that would insert numbers from 0-100 into the empty list. The size of the list was determined by the user’s input then it would run the “AppendRandom()” ‘n’ times.
* Bubble Sort, O(n^2)
  + The purpose of bubble sort is for all the larger values to rise to the top like big bubbles in water, hence the name “bubble sort”. The method would take the new list and create a temporary list. While the list is not empty, it checks the item in the current spot to the next, and if the current is larger, they swap values. There is a Boolean variable that checks if the sorting is done.
* Quick Sort
  + I was not able to get this or the modified Quicksort method to work. I attempted this method by creating a temporary list and then splitting the list a pivot point which would be the head and create two list one with values less than the pivot and the other with values greater than the pivot. Then the method would recursively call itself with the two new list to sort those as well. After the two list have been sorted, they would be combined with the pivot being at the end of the first list.
* Merge Sort, O(nlog\_2 n)
  + The merge sort function was split into three different methods, “mergeSort()” which created two new lists left and right, “separate()” would then determine which values would go into the appropriate list, and “merge()” which actually puts the whole list together by comparing the left side to the right side and ordering them from least to greatest.

***Experimental Results:***

Since each method works of the same randomly generated list, I tested the list odd numbers and even numbers to make sure the “Median()” method worked. The output for odd length was as expected, the even length list would return only one of the two desired outputs. The two functions I was able to complete were functioning properly.



Runtimes: I used these values to give a variety of runs

***Conclusion:***  After completing what I could from this project, my understanding of runtimes and Linked List in python has increased slightly. There are still areas where I struggle, for example the implementation of quick sort would give in error saying that the “Node” had no “item” when I was creating the temp lists. I believe there is an error in the math or the recursion because I had a similar issue with the “getCount()” method. In the process of interpreting the results I have also bettered my understanding of recurrence relations.

***Source Code:***

"""

Lab 2

@author: Marco Valerio

"""

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)

#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 AppendRandom(L,x):

# Inserts x at end of list L

if IsEmpty(L):

L.head = Node(x+random.randint(1,101))

L.tail = L.head

else:

L.tail.next = Node(x+random.randint(1,101))

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 PrintRec(L):

# Prints list L's items in order using recursion

PrintNodes(L.head)

print()

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

#Merge Sort

"""

Merge sort takes the list seperates it into two individual that also get sorted then combines the two sorted lists

"""

def mergeSort(head):

if head is None or head.next is None:

return head

left,right = seperate(head)

left = mergeSort(left)

right = mergeSort(right)

head = merge(left, right)

return head

"""

Seperate method serves to determine which elements go into which list 'left' or 'right'

"""

def seperate(L):

temp1 = L

temp2 = L

if temp2:

temp2 = temp2.next

while temp2:

temp2 = temp2.next

if temp2:

temp2 = temp2.next

temp1 = temp1.next

temp = temp1.next

temp1.next = None

return L, temp

"""

This is were the two list are compared and the elements are added in proper order

"""

def merge(left, right):

temp = None

if left is None:

return right

if right is None:

return left

if left.item <= right.item:

temp = left

temp.next = merge(left.next, right)

else:

temp = right

temp.next = merge(left, right.next)

return temp

"""

Bubble sort compares the list so that the bigger numbers rise to the top by checking each element against each other.

Has a boolean variable to check if the whole list has been sorted, example provided by Olac Fuentes

"""

#Bubble Sort

def bubblesort(L):

done = True

while done:

done = False

temp = L.head

while temp.next is not None:

if temp.item > temp.next.item:

hold = temp.item

temp.item = temp.next.item

temp.next.item = hold

done = True

temp = temp.next

"""

Quick sort should have split the list at a pivot point then created two list one less than and one greater than the pivot,

after that it would combine both list.

This method gave me an error with the pivot variable but I don't believe that is the true issue

"""

#Quick Sort

def quickSort(head):

temp = head

if temp is not None:

pivot = temp.item

L1 = List()

L2 = List()

temp=temp.next

for i in range(getCount(head)):

if temp.item < pivot:

Append(L1,temp.item)

else:

Append(L2,temp.item)

quickSort(L1)

quickSort(L2)

Append(L1,pivot)

return L1 + L2

def getCount(head):

temp = head

count = 0

while (temp):

count += 1

temp = temp.next

return count

def ElementAt(L,x):

bingo = 0

for i in range(x):

L=L.next

bingo = L.item

return bingo

"""

Median takes the list and finds the element of the sorted list by going through half the length of the list

"""

def Median(head):

temp = head

print(ElementAt(temp,(n//2)))

n = int(input("What is the list length?"))

L = List()

print(IsEmpty(L))

for i in range(n):

AppendRandom(L,i)

print("Original List")

Print(L)

print("Bubble Sort")

bubblesort(L)

Print(L)

print("Median")

Median(L.head)

print("Merge Sort")

mergeSort(L.head)

Print(L)

print("Median")

Median(L.head)

"""

I certify that this project is entirely my own work. I wote, debugged, and tested the code being presented, preformed 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 the

class.

"""