**Name:** Sai Rohit Kalyan Gandham

**StudentID:** 1002070724

**EmailID:** sxg0724@mavs.uta.edu

Q1)

**Code:**

import numpy

import time

import matplotlib.pyplot as plt



x\_points = []

y\_insertion = []

y\_mergesort = []

lengthOfList = 3 #inital length

while(True):

#Random List Generation

x\_points.append(lengthOfList)

mainList=list(numpy.random.randint(lengthOfList\*\*2, size=(lengthOfList)))

randomList=mainList

#Inserction Sort

start\_time = time.time()

for i in range(1,lengthOfList):

index = randomList[i]

j = i-1

while j >=0 and index < randomList[j] :

randomList[j+1] = randomList[j]

j -= 1

randomList[j+1] = index

insertion\_sort\_time = time.time()-start\_time

y\_insertion.append(insertion\_sort\_time)



#Merge Sort

def mergeSort(List):

if len(List) > 1:

mid = len(List)//2

Left = List[:mid]

Right = List[mid:]

mergeSort(Left)

mergeSort(Right)

i = j = k = 0

while i < len(Left) and j < len(Right):

if Left[i] < Right[j]:

List[k] = Left[i]

i += 1

else:

List[k] = Right[j]

j += 1

k += 1

while i < len(Left):

List[k] = Left[i]

i += 1

k += 1

while j < len(Right):

List[k] = Right[j]

j += 1

k += 1

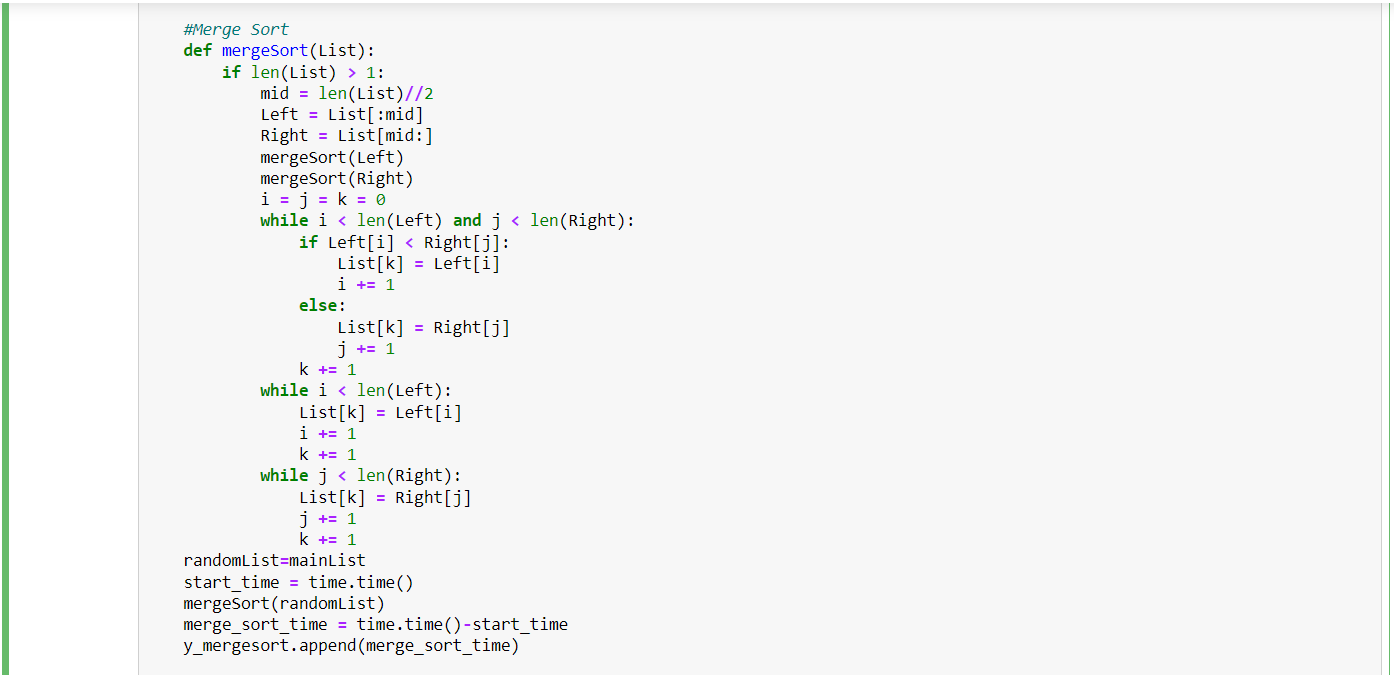
randomList=mainList

start\_time = time.time()

mergeSort(randomList)

merge\_sort\_time = time.time()-start\_time

y\_mergesort.append(merge\_sort\_time)



#Comparing Time

if(round(insertion\_sort\_time,2)>round(merge\_sort\_time,2)):

print("insertion sort time:",insertion\_sort\_time)

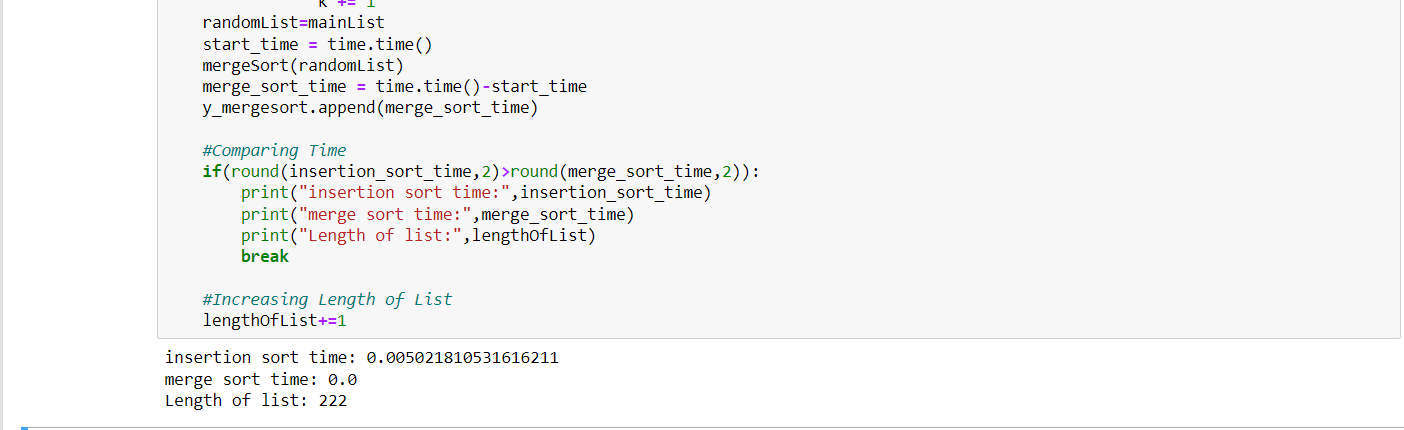
print("merge sort time:",merge\_sort\_time)

print("Length of list:",lengthOfList)

break

#Increasing Length of List

lengthOfList+=1



#plot

plt.rcParams["figure.figsize"] = (15,10)

plt.plot(x\_points, y\_insertion, 'g', label='Insertion Sort')

plt.plot(x\_points, y\_mergesort, 'b', label='Merge Sort')

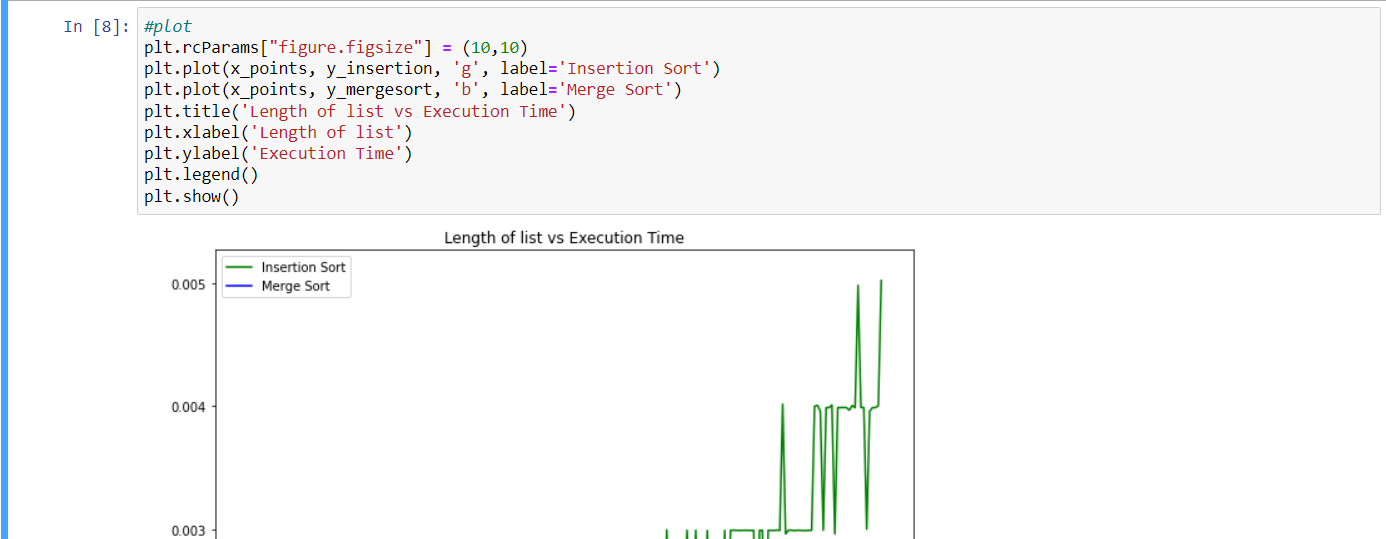
plt.title('Length of list vs Execution Time')

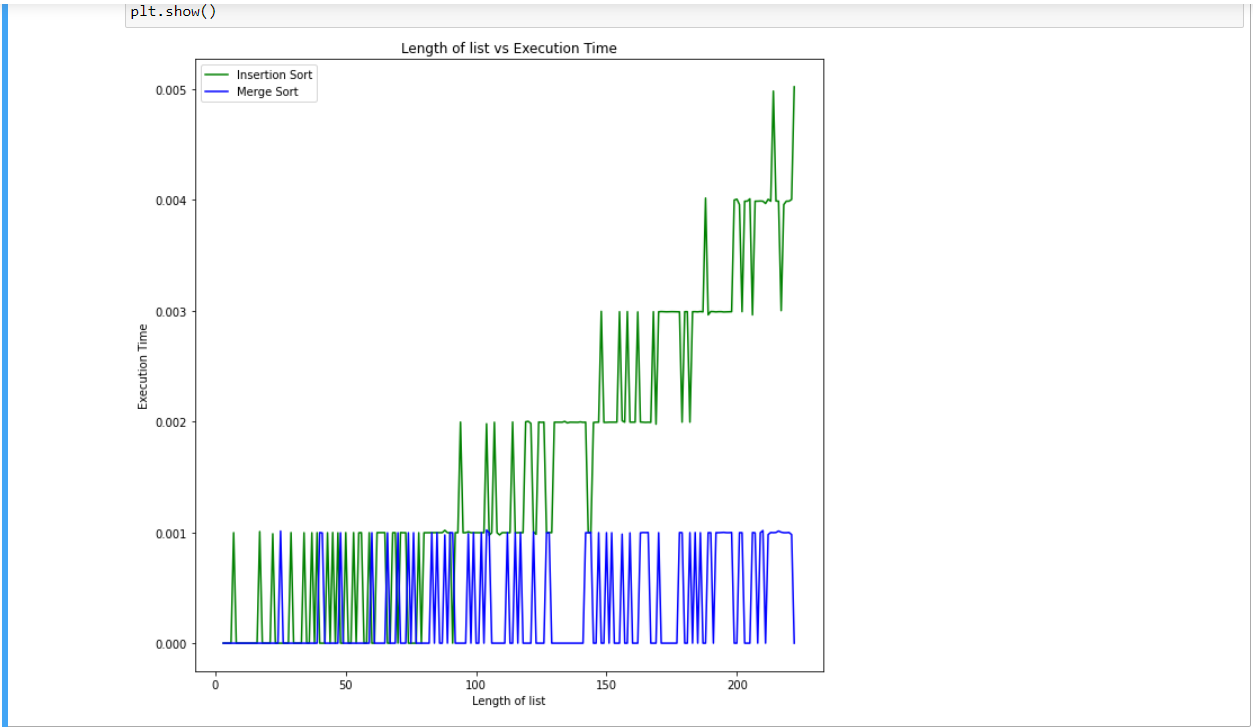
plt.xlabel('Length of list')

plt.ylabel('Execution Time')

plt.legend()

plt.show()





Asymptotic Analysis: is basically saying about the sutions in diving into algorithms we keep continuous test on time, space, memory and the stack over flow of the input size taken,we only calculate the actual execution time of code performance.

Hence we prove that when the length/Size of the input increases mergsort performance better than insertion sort.

Q2)

**Code:**

K = 5

#defining the insearation Sort

def insertionSortarrayListlgo(arrayList,farwardPointer,backwardPointer):

for i in rarrayListnge(farwardPointer,backwardPointer):

temfarwardPointerVarrayListl = arrayList[i + 1]

j = i + 1

while (j > farwardPointer and arrayList[j - 1] > temfarwardPointerVarrayListl):

arrayList[j] = arrayList[j - 1]

j-=1

arrayList[j] = temfarwardPointerVarrayListl

temfarwardPointer = arrayList[farwardPointer:backwardPointer +1]

print(temfarwardPointer)

#defining the merge Sort

def merge(arrayList,farwardPointer,backwardPointer,r):

leftIndexPrefix = backwardPointer - farwardPointer + 1

rightIndexPrefix = r - backwardPointer

LarrayList = arrayList[farwardPointer : backwardPointer +1]

RarrayList = arrayList[backwardPointer+1 : r +1]

rightIndex = 0

leftIndex = 0

for i in range(farwardPointer, r - farwardPointer + 1):

if(rightIndex == rightIndexPrefix):

arrayList[i] = LarrayList[leftIndex]

leftIndex+=1

elif(leftIndex == leftIndexPrefix):

arrayList[i] = RarrayList[rightIndex]

rightIndex+=1

elif(RarrayList[rightIndex] > LarrayList[leftIndex]):

arrayList[i] = LarrayList[leftIndex]

leftIndex+=1

else:

arrayList[i] = RarrayList[rightIndex]

rightIndex+=1

# Defining and calling of the merge insertion sort or also forn-jhonson sort

def sort(arrayList,farwardPointer,r):

if(r - farwardPointer > K):

backwardPointer = int((farwardPointer + r) / 2)

sort(arrayList, farwardPointer, backwardPointer)

sort(arrayList, backwardPointer + 1, r)

merge(arrayList, farwardPointer, backwardPointer, r)

else:

insertionSort(arrayList,farwardPointer,r)

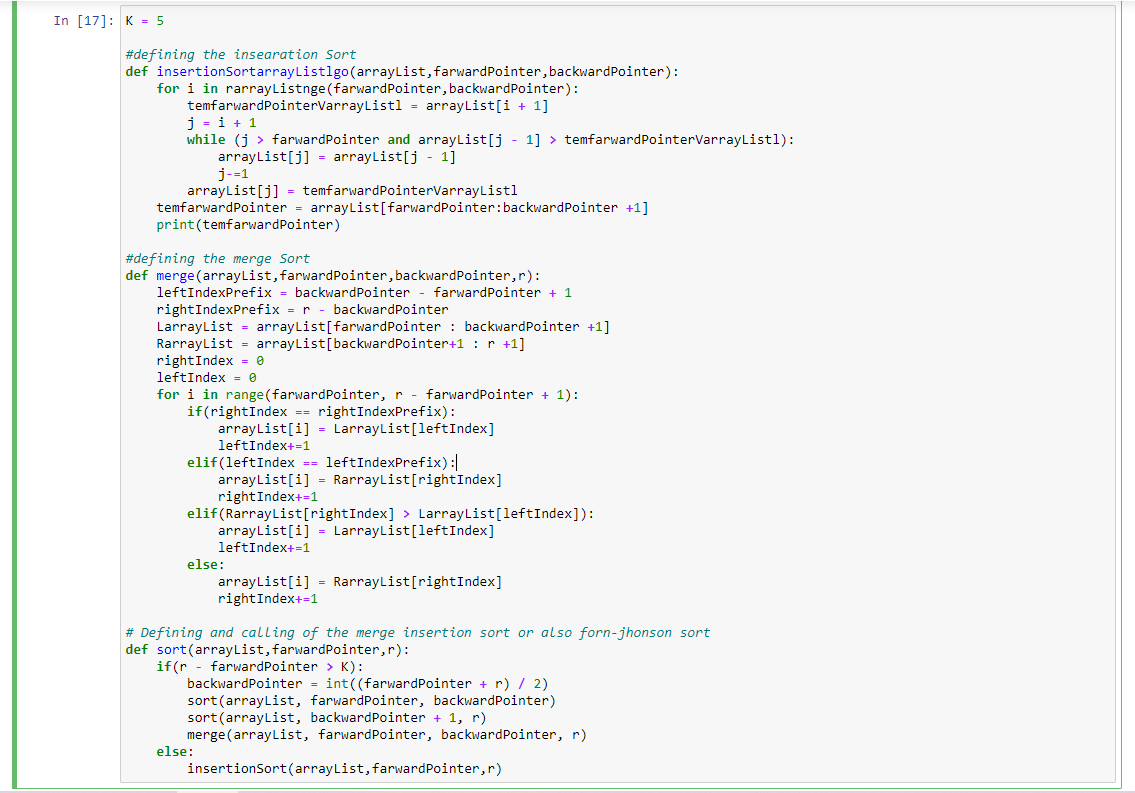
#Create a sample arrayList

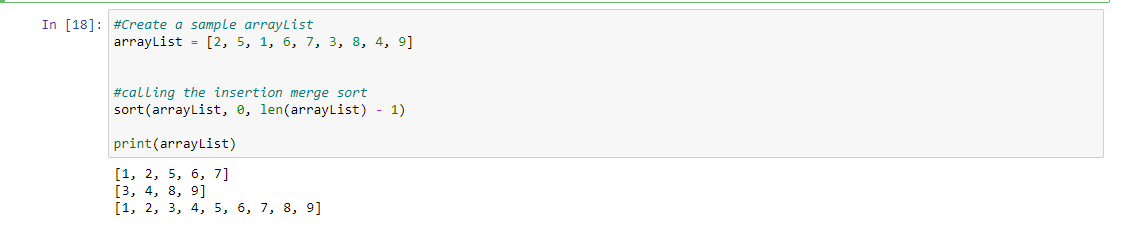
arrayList = [2, 5, 1, 6, 7, 3, 8, 4, 9]

#calling the insertion merge sort

sort(arrayList, 0, len(arrayList) - 1)

print(arrayList)





a. The worst-case time of the sort of k sublists are T(n) = (n/k) \* bigO(k2) = bigO(nk).

because the sublist depends upon the number of merge and insertion sort functions calling.

b. The height of the sublists of the given tree will be depends on the k and length of the list (n)

so if there is n/k sublists the length/height of the log(n/k) and it is under merge sort

so the complexity of the merge is bigO(n) so the final worst-case senario is bigO(nlog(n/k)).

c. If you close look at the code for problem 2

for i in range(farwardPointer, r - farwardPointer + 1):

if(rightIndex == rightIndexPrefix):

arrayList[i] = LarrayList[leftIndex]

leftIndex+=1

elif(leftIndex == leftIndexPrefix):

arrayList[i] = RarrayList[rightIndex]

rightIndex+=1

elif(RarrayList[rightIndex] > LarrayList[leftIndex]):

arrayList[i] = LarrayList[leftIndex]

leftIndex+=1

else:

arrayList[i] = RarrayList[rightIndex]

rightIndex+=1

The time complexity of sorting is bigO(nk+ nlog(n/k)) that is approxmatly equal to the bigO(nlog(n))

so when the K value increses the faster the logic will be sorting. so to keep bigO(nk + nlog(n/k)) ==

bigO(nk + nlog(n) - nlog(k)) must be equal to bigO(nlog(n/k)) we should let the K grow quciker than the log(n)

otherwise nk term complxity will run less effecitive k <= big)(log(n)). so the highest value of the k = log(n)

d. Given

insertion sort = c1og2

merge sort = c2nlog(n)

Step1: comaparision of the c1k2 <= c2klog(k)

Step2: comaparision of the k <= c2klog(k)

Case 1:

k = 0

Step1: c1 \* 0 = c2 \* 0 \* log(0) ==> 0 == 0

Step2: 0 <= c2 \* 0 \* 0 ==> 0 == 0

case 2:

k = 1

Step1: c1 \* 1 = c2 \* 1 \* log(1) ==> c1 < c2

c1 = c2

Step2: 1 = c2 \* 1 \* log(1) ==> 1 < c2

Hence from both the equitation we put values for K, c1, c2 and get the respected other resulted.