Q1) [60 points] Implement the insertion sort and merge sort algorithms with any programming language you choose and run them with the same input number list. Generate the list elements with a random function and increase the list size incrementally until you find the execution time of your merge sort program is consistently shorter. Plot the two curves in a figure (execution time vs input list size) about the two programs. Using the example to discuss why the asymptotic analysis is meaningful. Attach program codes in your submission.

**Answer:-**

\* Spyder version: 5.1.5 None

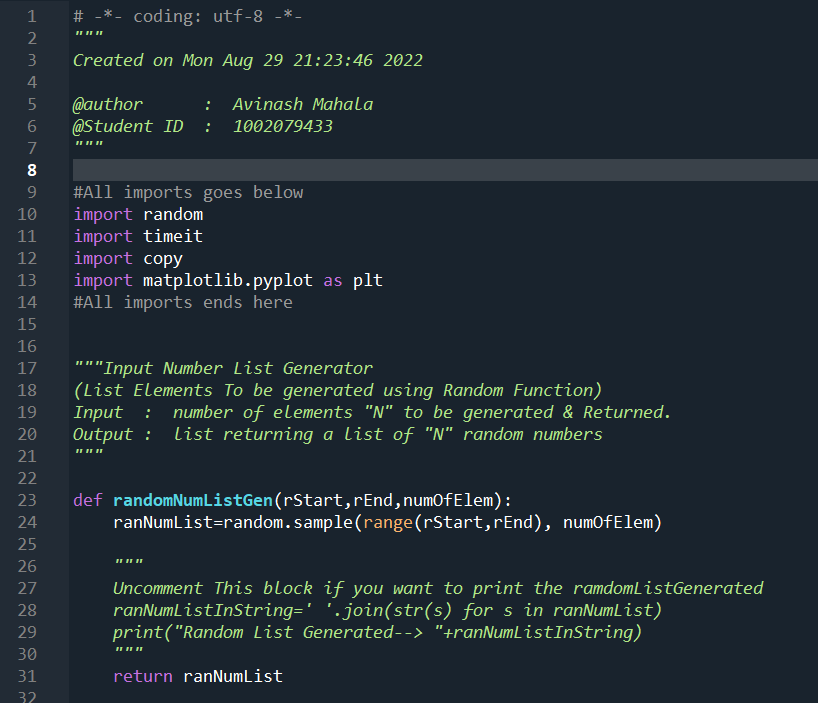
\* Python version: 3.9.12 64-bit

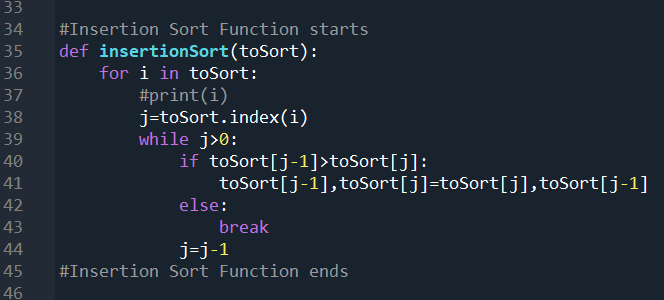
\* Qt version: 5.9.7

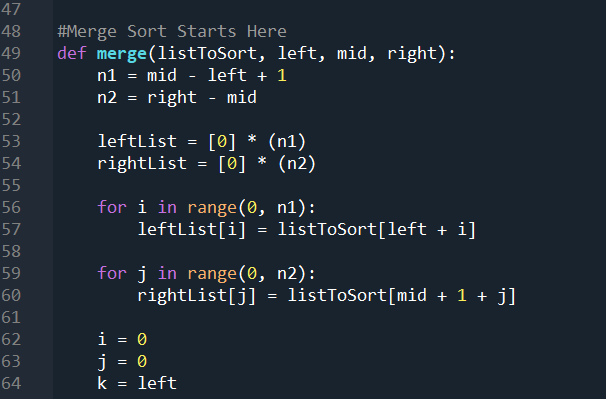
\* PyQt5 version: 5.9.2

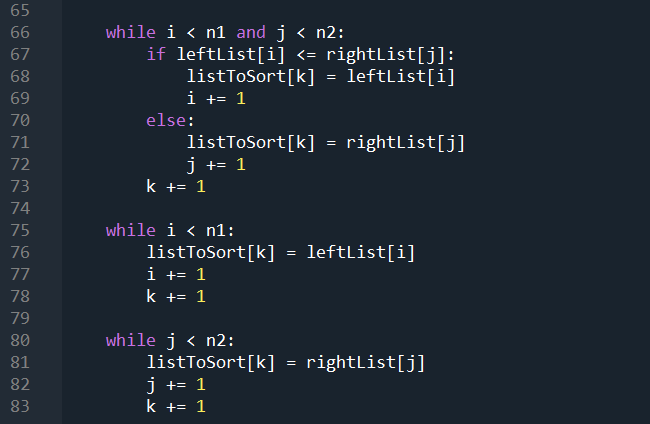
\* Operating System: Windows 10

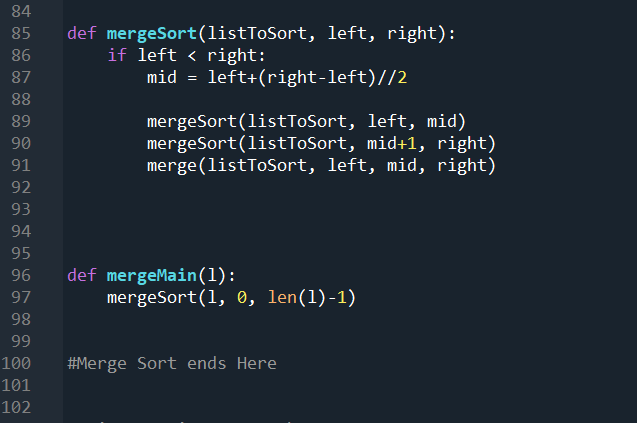
Code: - The Code was written and developed in Spyder by Anaconda IDE. Please make sure all the modules are successfully imported before running the below code.

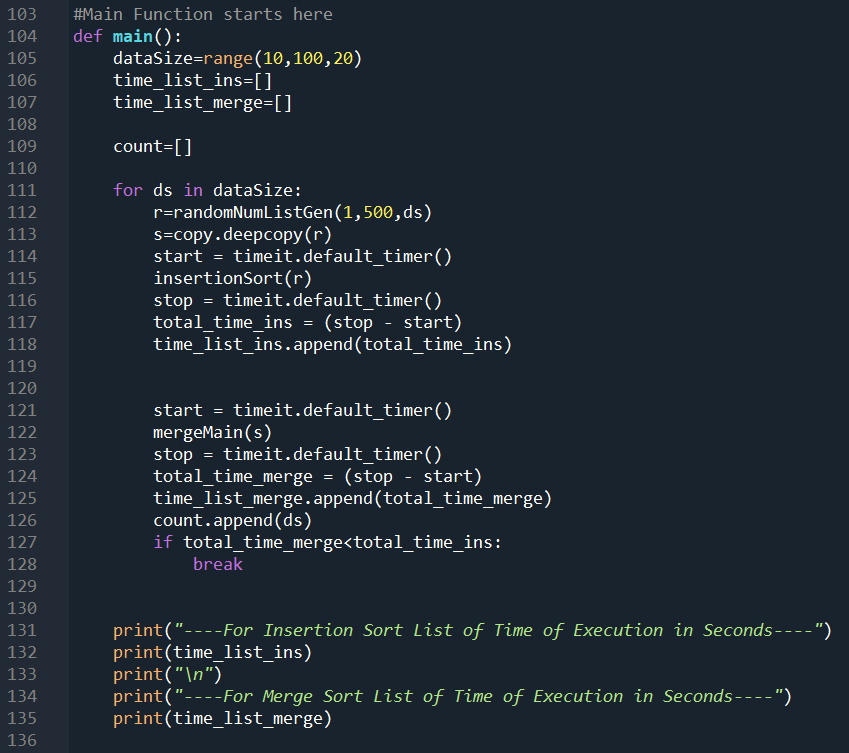














**Code With Line Numbers:-[Copy The below Code Without Line Numbers in Order to test in an IDE]**

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

002: """

003: Created on Mon Aug 29 21:23:46 2022

004:

005: @author : Avinash Mahala

006: @Student ID : 1002079433

007: """

008:

009: #All imports goes below

010: import random

011: import timeit

012: import copy

013: import matplotlib.pyplot as plt

014: #All imports ends here

015:

016:

017: """Input Number List Generator

018: (List Elements To be generated using Random Function)

019: Input : number of elements "N" to be generated & Returned.

020: Output : list returning a list of "N" random numbers

021: """

022:

023: def randomNumListGen(rStart,rEnd,numOfElem):

024: ranNumList=random.sample(range(rStart,rEnd), numOfElem)

025:

026: """

027: Uncomment This block if you want to print the ramdomListGenerated

028: ranNumListInString=' '.join(str(s) for s in ranNumList)

029: print("Random List Generated--> "+ranNumListInString)

030: """

031: return ranNumList

032:

033:

034: #Insertion Sort Function starts

035: def insertionSort(toSort):

036: for i in toSort:

037: #print(i)

038: j=toSort.index(i)

039: while j>0:

040: if toSort[j-1]>toSort[j]:

041: toSort[j-1],toSort[j]=toSort[j],toSort[j-1]

042: else:

043: break

044: j=j-1

045: #Insertion Sort Function ends

046:

047:

048: #Merge Sort Starts Here

049: def merge(listToSort, left, mid, right):

050: n1 = mid - left + 1

051: n2 = right - mid

052:

053: leftList = [0] \* (n1)

054: rightList = [0] \* (n2)

055:

056: for i in range(0, n1):

057: leftList[i] = listToSort[left + i]

058:

059: for j in range(0, n2):

060: rightList[j] = listToSort[mid + 1 + j]

061:

062: i = 0

063: j = 0

064: k = left

065:

066: while i < n1 and j < n2:

067: if leftList[i] <= rightList[j]:

068: listToSort[k] = leftList[i]

069: i += 1

070: else:

071: listToSort[k] = rightList[j]

072: j += 1

073: k += 1

074:

075: while i < n1:

076: listToSort[k] = leftList[i]

077: i += 1

078: k += 1

079:

080: while j < n2:

081: listToSort[k] = rightList[j]

082: j += 1

083: k += 1

084:

085: def mergeSort(listToSort, left, right):

086: if left < right:

087: mid = left+(right-left)//2

088:

089: mergeSort(listToSort, left, mid)

090: mergeSort(listToSort, mid+1, right)

091: merge(listToSort, left, mid, right)

092:

093:

094:

095:

096: def mergeMain(l):

097: mergeSort(l, 0, len(l)-1)

098:

099:

100: #Merge Sort ends Here

101:

102:

103: #Main Function starts here

104: def main():

105: dataSize=range(10,100,20)

106: time\_list\_ins=[]

107: time\_list\_merge=[]

108:

109: count=[]

110:

111: for ds in dataSize:

112: r=randomNumListGen(1,500,ds)

113: s=copy.deepcopy(r)

114: start = timeit.default\_timer()

115: insertionSort(r)

116: stop = timeit.default\_timer()

117: total\_time\_ins = (stop - start)

118: time\_list\_ins.append(total\_time\_ins)

119:

120:

121: start = timeit.default\_timer()

122: mergeMain(s)

123: stop = timeit.default\_timer()

124: total\_time\_merge = (stop - start)

125: time\_list\_merge.append(total\_time\_merge)

126: count.append(ds)

127: if total\_time\_merge<total\_time\_ins:

128: break

129:

130:

131: print("----For Insertion Sort List of Time of Execution in Seconds----")

132: print(time\_list\_ins)

133: print("\n")

134: print("----For Merge Sort List of Time of Execution in Seconds----")

135: print(time\_list\_merge)

136:

137: x1 = count

138: y1 = time\_list\_ins

139: plt.plot(x1, y1, label = "Insertion Sort")

140:

141: x2 = count

142: y2 = time\_list\_merge

143: plt.plot(x2, y2, label = "Merge Sort")

144: plt.xlabel('Input List Size')

145: plt.ylabel('Execution Time [ In Seconds ]')

146: plt.title('Insertion Sort Vs Merge Sort')

147:

148: plt.legend()

149: plt.show()

150: #Main Function ends here

151:

152:

153:

154: if \_\_name\_\_ == "\_\_main\_\_":

155: main()

156:

157:

158:

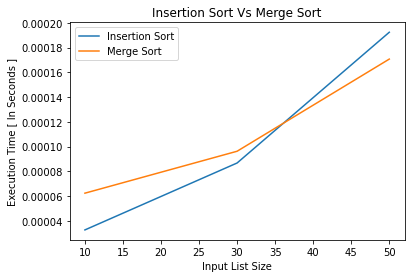
159:

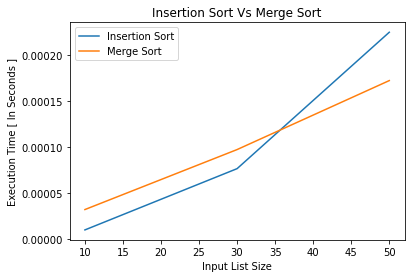
160:

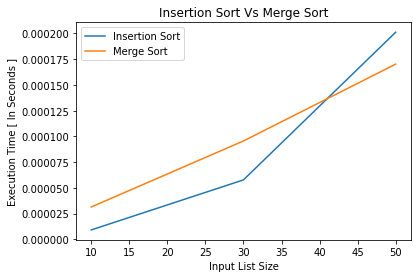
161:

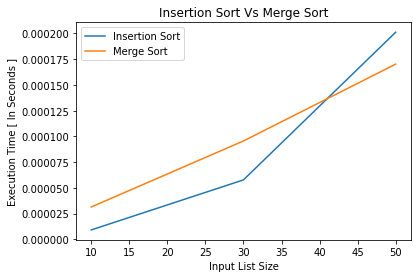
**Multiple Graphs When Executed at Different Times:-**

Here While executing the code at different time instances, I found that the graphs produced were not constant or same all the time, but there were similarities in terms of best case, average case and worst case time scenarios. Here every time the code ran, the run time performance was dependent on multiple external factors apart from input set of numbers. Hence, asymptotic analysis is meaningful in order to conclude the mathematically bounded run time performance. Asymptotic analysis is input bound. It means the algorithms run time complexity depends on only the input to the algorithm. All other factors are considered to be constant. Hence if there is no input to the algorithm, it is established to work in a constant time.





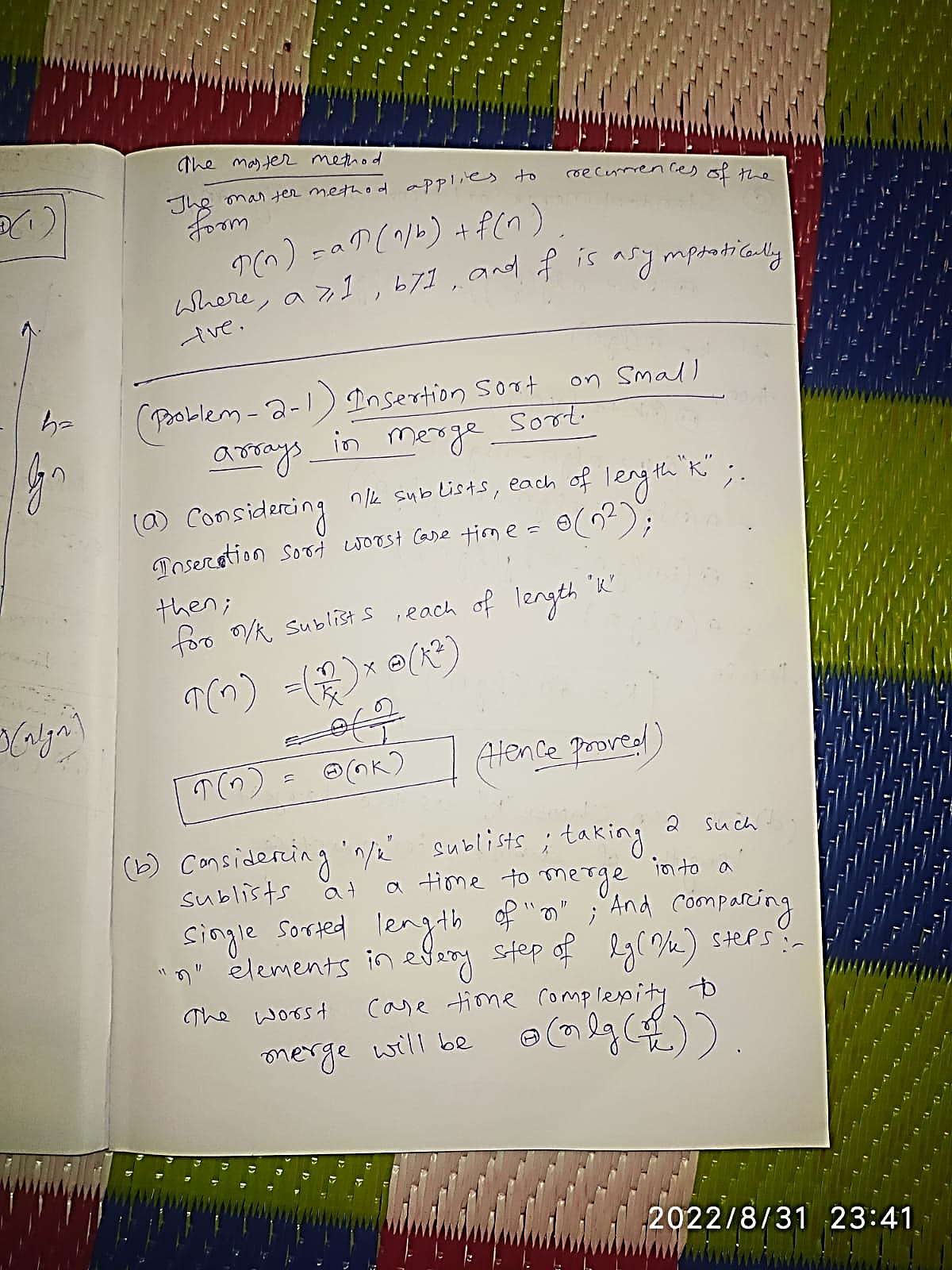


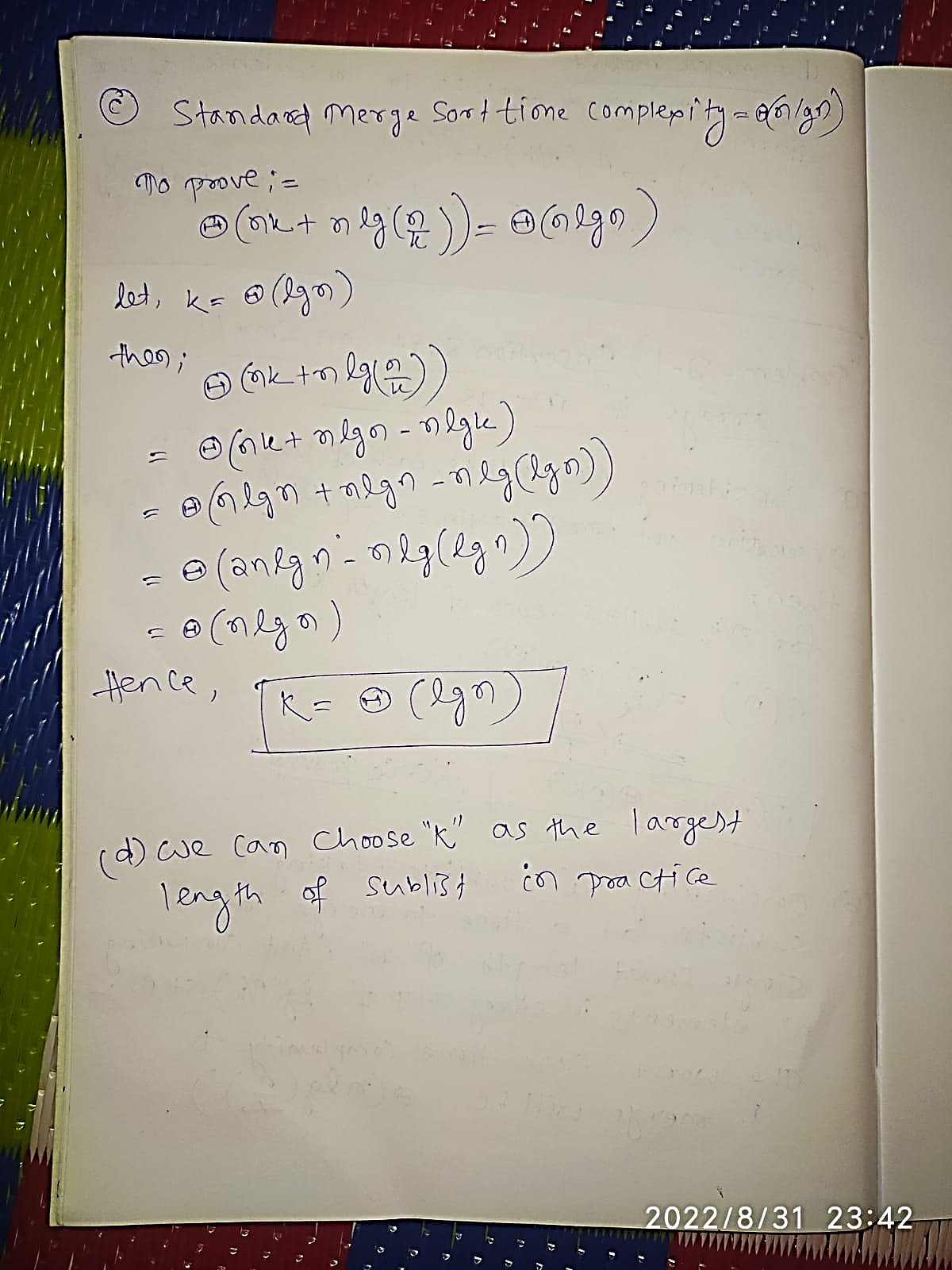
s

**Q2) [40 points] Problem 2-1 on Page 39 of the CLRS textbook. (“2-1 Insertion sort on small arrays in merge sort”)**

2-1 Insertion sort on small arrays in merge sortAlthough merge sort runs in ‚.n lg n/ worst-case time and insertion sort runs  
in ‚.n2/ worst-case time, the constant factors in insertion sort can make it faster  
in practice for small problem sizes on many machines. Thus, it makes sense to  
coarsen the leaves of the recursion by using insertion sort within merge sort when subproblems become sufficiently small. Consider a modification to merge sort in  
which n=k sublists of length k are sorted using insertion sort and then merged  
using the standard merging mechanism, where k is a value to be determined.  
***a.*** Show that insertion sort can sort the n=k sublists, each of length k, in ‚.nk/  
worst-case time.  
***b.*** Show how to merge the sublists in ‚.n lg.n=k// worst-case time.  
***c.*** Given that the modified algorithm runs in ‚.nk C n lg.n=k// worst-case time,  
what is the largest value of k as a function of n for which the modified algorithm  
has the same running time as standard merge sort, in terms of ‚-notation?  
***d.*** How should we choose k in practice?

**Answer:-**





--------------------------------End of HomeWork-1---------------------------------