**SVKM’s NMIMS**

**Mukesh Patel School of Technology Management & Engineering**

**Computer Engineering Department**

Program: B.Tech. Sem V

**Course: Design and Analysis of Algorithms**

w.e.f. 1st Jul 2020

**Faculty:** Prof. Abhay Kolhe.

LAB Manual 4

PART B

|  |  |
| --- | --- |
| Roll No: B032 | Name: Naman Garg |
| Class: CS B | Batch: B2 |
| Date of Experiment: 28/07/20 | Date of Submission: 07/08/20 |
| Grade: | Time of Submission: |
| Date of Grading: |  |

**B.1 Software Code written by student:**

#Name: Naman Garg

#Roll No: B032

#Aim: Implementation of a divide & conquer algorithm - Merge Sort

# function to sort the 2 subarrays and merge them

def merge(m, n, r):

    global l, copy1, copy2, c, total\_c

    c = 0

    i, j, k = 0, 0, m

    # Subarray 1 - left of the mid element

    copy1 = l[m:n+1]

    # Subarray 2 - right of the mid element

    copy2 = l[n+1:r+1]

    # Till all elements of a subarray are compared

    while(i < len(copy1) and j < len(copy2)):

        c += 1

        # smaller element between the corresponding 2 elements of the 2 subarrays is chosen

        if(copy1[i] <= copy2[j]):

            l[k] = copy1[i]

            i, k = i + 1, k + 1

        elif(copy1[i] > copy2[j]):

            l[k] = copy2[j]

            j, k = j + 1, k + 1

    # When 1 subarray gets exhausted of elements

    else:

        #remaining elements will be filled in the sorted array as it is

        while(i < len(copy1)):

            l[k] = copy1[i]

            i, k = i+1, k+1

        while(j < len(copy2)):

            l[k] = copy2[j]

            j, k = j+1, k+1

    total\_c += c

# Repeatedly divides the array into subarrays to sends to merge

def mergesort(m, r):

    global l, copy1, copy2

    mid = (m+r)//2                                    # mid -> the middle index

    if(m < r):

        print(

            f'Division : Array from index {m} to {r} {l[m:r+1]} breaks into Array 1: from {m} to {mid} {l[m:mid+1]} and Array 2: from {mid+1} to {r}          {l[mid+1:r+1]}')

        # recursive call for left

        mergesort(m, mid)

        # recursive call for right

        mergesort(mid+1, r)

        print(

            f'Merging : Lower bound index ={m} Middle index ={mid} Upper bound index ={r} ->', end=' ')

        merge(m, mid, r)                              # sort and merge

        print(f'{copy1} and {copy2} on sorting and merging ->{l[m:r+1]} -> comparisons ={c}')

# Input

l = list(map(int, input().split()))

# Initialize the global variables

copy1, copy2 = [], []

c, total\_c = 0, 0

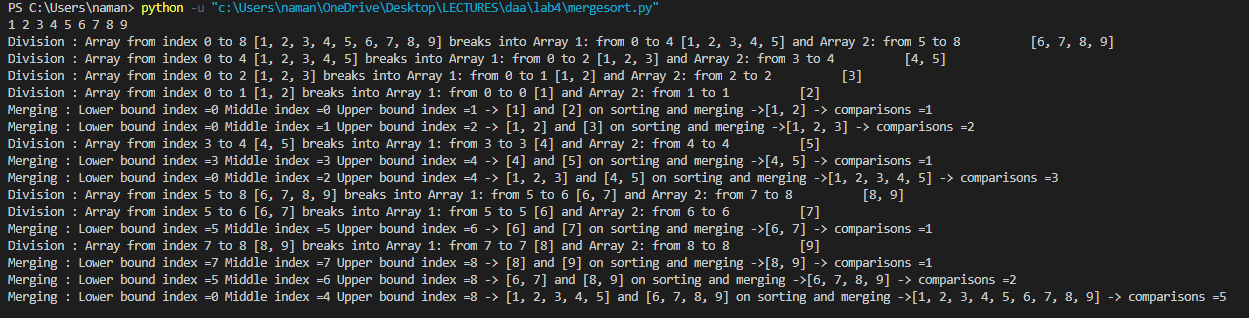
mergesort(0, len(l)-1)

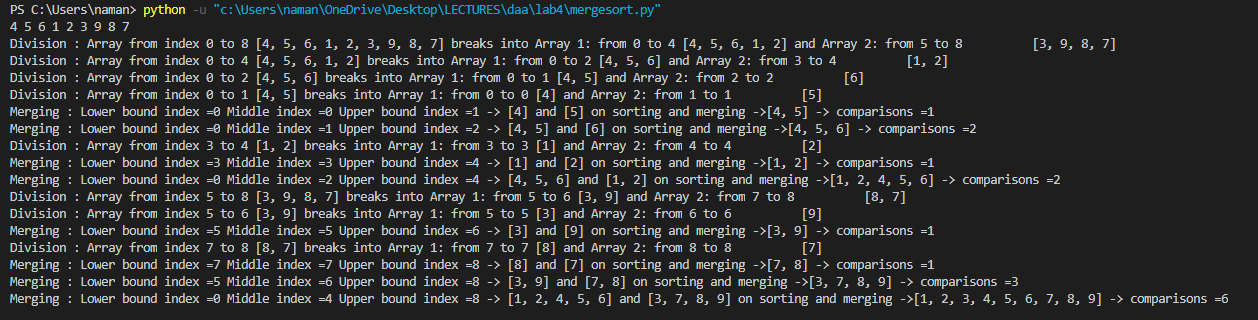
print('\n')

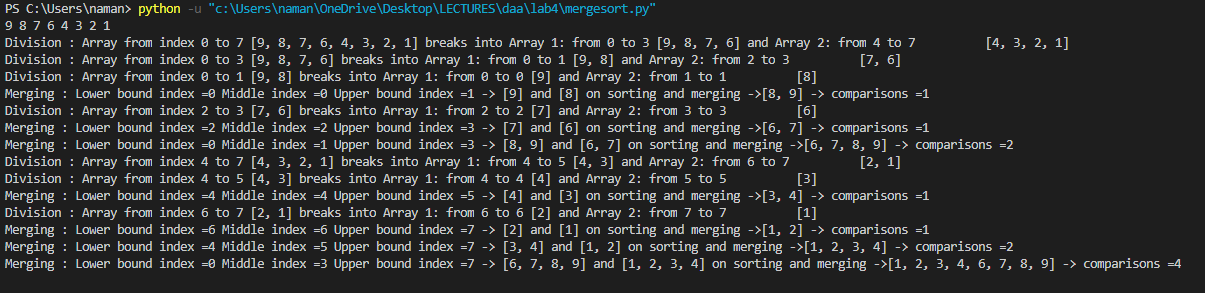
print(f'Final sorted list ->{l} -> Total comparisons ={total\_c}')

**B.2 Input and Output:**

1. **For Sorted Array**

****

1. **For Unsorted Array**
2. **For Reverse Sorted Array**

****

**B.3 Observations and learning:**

I observed that merge sort is all useful sorting technique when there are size complications and is used in production in many applications however most production applications use a version of merge sort which is more optimized by statistics

**B.4 Conclusion:**

1. Designed and developed the merge sorting technique
2. Identified the applications of merge sort
3. Analyzed the time complexity of merge sort

**B.5 Question of Curiosity**

Q.1 Identify the applications of Merge Sort.

What sort is more applicable than quicksort in many applications because it does not modify the original array and it does not reshuffle the elements therefore if it fails or if it stops in the middle of execution We don't lose data whereas quick short we would have to abandon.

Some of its real life applications are:

* Useful for sorting linked lists in O(n Log n) time
* Used in external sorting

Q.2 Analyze the algorithm for Merge sort – Best case, Worst Case and Average case.

What should always divides the array in halfs therefore it's time complexity is always N log N

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*