**KATHMANDU UNIVERSITY**

**Department of Computer Science and Engineering**

**Dhulikhel, Kavre**

****

**Lab Sheet 2**

**Algorithm and Complexity**

**[Course Code: COMP 314]**

**Submitted by:**

Sajag Silwal (Roll no: 48)

**Submitted to:**

Dr. Rajani Chulyadyo

Department of Computer Science and Engineering

**Submission Date:** 24-03-2022

# Implementation of Merge Sort and Insertion Sort

# Merge Sort

Merge Sort is quite similar to the Quicksort algorithm as it is based on the “Divide and Conquer” algorithm. It fop-down recursion and divides the array into two halves, calls itself for the two halves, and merges the sorted array.

Its simulation is given below:

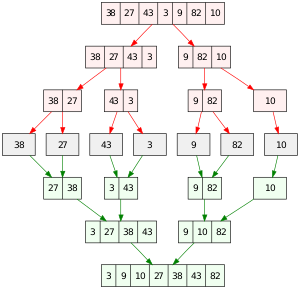


Figure : Simulation of Merge Sort

# Source Code:

The Source Code for Merge Sort Implementation is given below:

def MergeSort(array, left, right):

    if left >= right:

        return

    middle = (left + right)//2

    MergeSort(array, left, middle)

    MergeSort(array, middle + 1, right)

    Merge(array, left, right, middle)

def Merge(array, left, right, middle):

    leftCopy = array[left:middle + 1]

    rightCopy = array[middle+1:right+1]

    leftCopyIndex = 0

    rightCopyIndex = 0

    sortedIndex = left

    while leftCopyIndex < len(leftCopy) and rightCopyIndex < len(rightCopy):

        if leftCopy[leftCopyIndex] <= rightCopy[rightCopyIndex]:

            array[sortedIndex] = leftCopy[leftCopyIndex]

            leftCopyIndex = leftCopyIndex + 1

        else:

            array[sortedIndex] = rightCopy[rightCopyIndex]

            rightCopyIndex = rightCopyIndex + 1

        sortedIndex = sortedIndex + 1

    while leftCopyIndex < len(leftCopy):

        array[sortedIndex] = leftCopy[leftCopyIndex]

        leftCopyIndex = leftCopyIndex + 1

        sortedIndex = sortedIndex + 1

    while rightCopyIndex < len(rightCopy):

        array[sortedIndex] = rightCopy[rightCopyIndex]

        rightCopyIndex = rightCopyIndex + 1

        sortedIndex = sortedIndex + 1

## Output:

## Test Case for Merge Sort

The test case for the merge sort is given below:

from MergeSort import MergeSort

import unittest

input1  =  [1,2,3,4,5,6,7,8,9,10]

output1 =  [1,2,3,4,5,6,7,8,9,10]

input2 = [9,1,21,5,6,8,109,4,20,50]

output2  =  [1,4,5,6,8,9,20,21,50,109]

input3  =  [10,9,8,7,6,5,4,3,2,1]

output3  = [1,2,3,4,5,6,7,8,9,10]

r1=len(input1)

r2=len(input2)

r3=len(input3)

class MergeSortCase(unittest.TestCase):

    def test\_MergeSort(self):

        MergeSort(input1,0,r1)

        MergeSort(input2,0,r2)

        MergeSort(input3,0,r3)

        self.assertEqual(input1,output1)

        self.assertEqual(input2,output2)

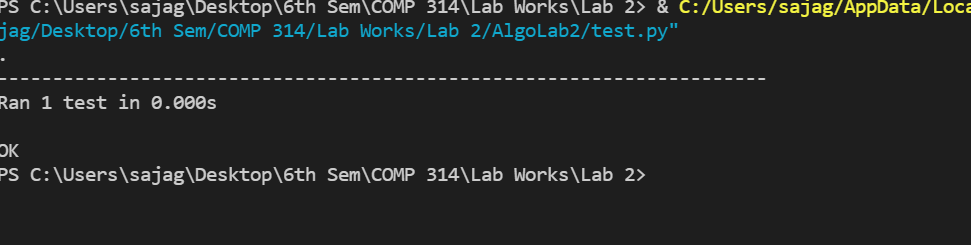
        self.assertEqual(input3,output3)

if  \_\_name\_\_=="\_\_main\_\_":

    unittest.main()

## Output:

The output for the test case is given below:



# Insertion Sort

Insertion sort is a simple sorting algorithm that is similar to the way playing card is sorted in hands. In this method, an array is divided into two parts: Sorted and Unsorted. It is sorted gradually from left to right. The simulation of insertion sort is given below:

Figure : Simulation of Insertion Sort

## Source Code:

The source code for the Insertion Sort is :

def InsertionSort(A):

    n  = len(A)

    for j  in range (1,n):

        key =A[j]

        i = j-1

        while i>=0  and A[i]>key:

            A[i+1] = A[i]

            i  =  i-1

        A[i+1] =  key

array= [10,9,1,8,3,2,4,5,90,5]

array= [10,9,1,8,3,2,4,5,90,5]

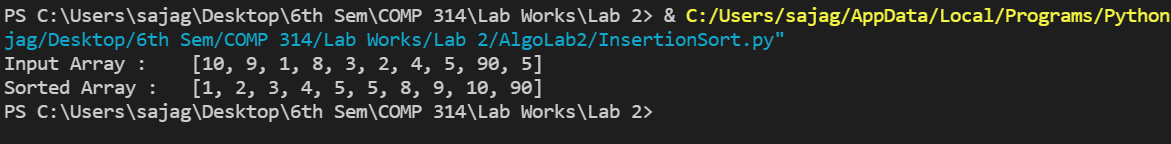
print(f"Input Array : \t {array}")

InsertionSort(array)

print(f"Sorted Array : \t {array}")

## Output:

The output of the source code above is:



# Test Case for Merge Sort

The test case for insertion sort is given below:

from InsertionSort import InsertionSort

import unittest

input1  =  [1,2,3,4,5,6,7,8,9,10]

output1 =  [1,2,3,4,5,6,7,8,9,10]

input2 = [9,1,21,5,6,8,109,4,20,50]

output2  =  [1,4,5,6,8,9,20,21,50,109]

input3  =  [10,9,8,7,6,5,4,3,2,1]

output3  = [1,2,3,4,5,6,7,8,9,10]

class   InsertionTestCase(unittest.TestCase):

    def test\_insertionSort(self):

        InsertionSort(input1)

        InsertionSort(input2)

        InsertionSort(input3)

        self.assertEqual(input1,output1)

        self.assertEqual(input2,output2)

        self.assertEqual(input3,output3)

if  \_\_name\_\_=="\_\_main\_\_":

    unittest.main()

# Time complexity (Merge Sort and Insertion Sort)

The time complexity for merge sort and insertion sort where the sample size is 100 is given below:

## Source Code:

import time

import random

from matplotlib import pyplot as plt

from MergeSort import MergeSort

from InsertionSort import InsertionSort

def generate\_random\_list(size):

    return [random.choice(range(size)) for i in range(size)]

def calculate\_time(func):

    """Decorator function for calculating time"""

    def inner(\*args, \*\*kwargs):

        tic = time.time\_ns()

        func(\*args, \*\*kwargs)

        toc = time.time\_ns()

        return toc - tic

    return inner

@calculate\_time

def check\_time\_insertion\_sort(arr):

    return InsertionSort(arr)

@calculate\_time

def check\_time\_merge\_sort(arr):

    r = len(arr)

    return MergeSort(arr,0,r)

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

    samples = [generate\_random\_list(i) for i in range(0, 1000, 10)]

    sample\_sizes = []

    insertion\_sort\_times = []

    merge\_sort\_times = []

    for sample in samples:

        sample\_sizes.append(len(sample))

        insertion\_sort\_times.append(check\_time\_insertion\_sort(sample))

        merge\_sort\_times.append(check\_time\_merge\_sort(sample))

    # Plotting

    plt.figure(figsize=(10, 6))

    plt.xlabel("Sample Size (n)")

    plt.ylabel("Time Elasped (ns)")

    plt.title("Time Complexity: Insertion sort vs Merge Sort")

    plt.plot(sample\_sizes, insertion\_sort\_times, ",-", label="Insertion Sort")

    plt.plot(sample\_sizes, merge\_sort\_times, ",-", label="Merge Sort")

    plt.legend()

    plt.show()

    print("done")

Output:   
The graph for the time complexity is given as:

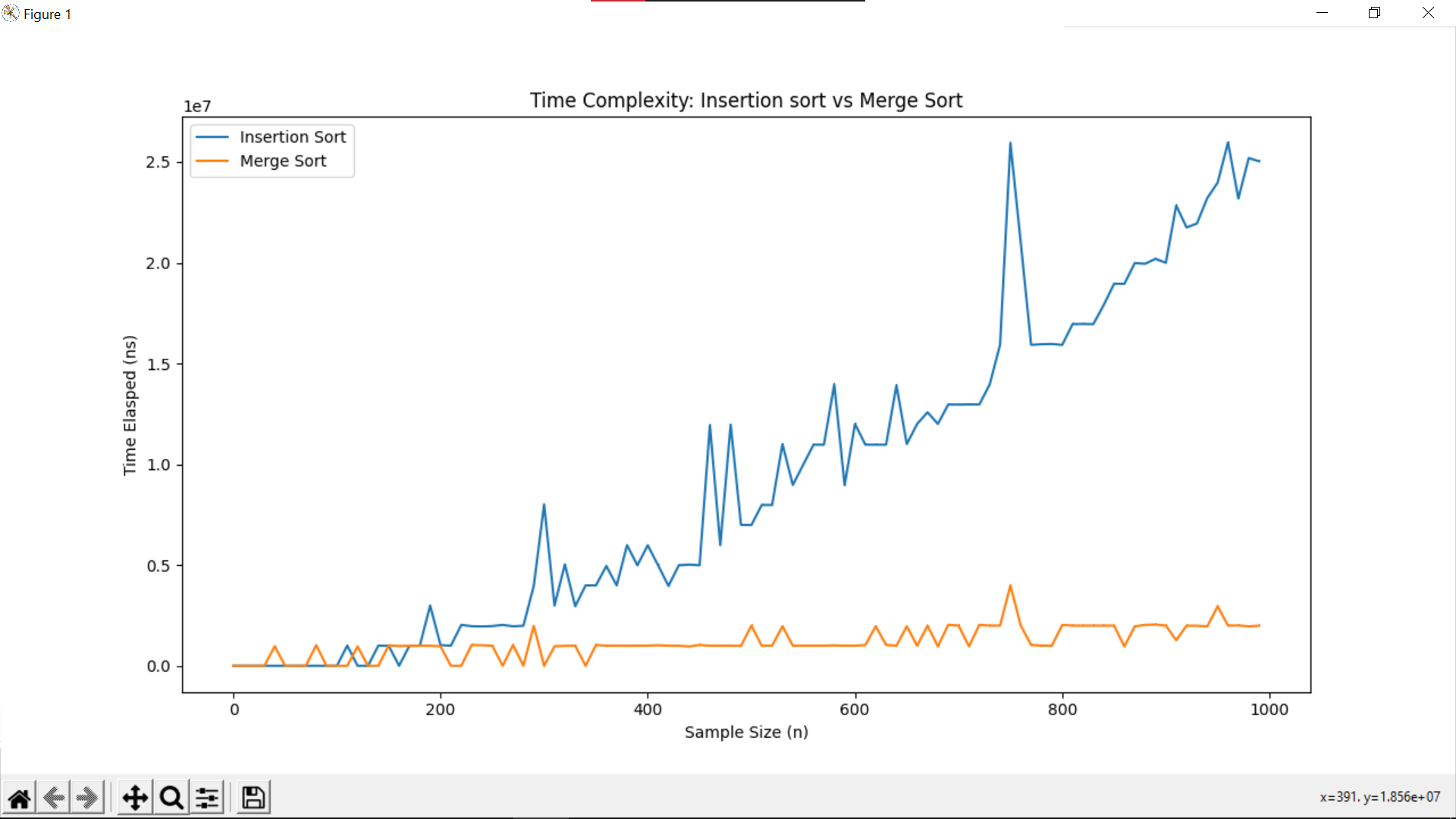


Figure : Time Complexity for Merge Sort and Insertion Sort