# **Sorting Algorithms**

**Merge Sort**

Merge Sort is a kind of divide and conquer algorithm. The concept of Divide and Conquer involves three steps:

* Divide the problem into multiple small problems.
* Conquer the subproblems by solving them. The idea is to break down the problem into atomic subproblems, where they are actually solved.
* Combine the solutions of the subproblems to find the solution of the actual problem.

In Merge Sort, the given unsorted array with n elements, is divided into n subarrays, each having one element, because a single element is always sorted in itself. Then, it repeatedly merges these subarrays, to produce new sorted subarrays, and in the end, one complete sorted array is produced.

**Algorithm:**

MERGE-SORT (A, p, r)

if p < r

q = 𝖫 (p + r) / 2 **⅃**

MERGE-SORT (A, p, q)

MERGE-SORT (A, q+1, r)

MERGE (A, p, q, r)

MERGE (A, p, q, r)

n1 = q – p + 1

n2 = q – p + 1

let L[1…n1 + 1] and R[1…n2 + 1]

for i = 1 to n1

L[i] = A[p + i - 1]

for j = 1 to n2

R[j] = A[q + j]

L[n1 + 1] = ∞

L[n2 + 1] = ∞

i = 1

j = 1

for k = p to r

if L[i] ≤ R[j]

A[k] = L[i]

i = i + 1

else A[k] = R[j]

j = j + 1

**Code (merge.py)**

def mergeSort(data, p, r):

    if p < r:

        q = int((p+r-1)/2)

        mergeSort(data, p, q)

        mergeSort(data, q + 1, r)

        merge(data, p, q, r)

def merge(data, p, q, r):

    n1 = q - p + 1

    n2 = r - q

    L = [0] \* (n1)

    R = [0] \* (n2)

    for i in range(n1):

        L[i] = data[p + i]

    for j in range(n2):

        R[j] = data[q + j + 1]

    i = 0

    j = 0

    k = p

    while (i < n1 and j < n2):

        if (L[i] <= R[j]):

            data[k] = L[i]

            i += 1

        else:

            data[k] = R[j]

            j += 1

        k += 1

    while (i < n1):

        data[k] = L[i]

        i += 1

        k += 1

    while (j < n2):

        data[k] = R[j]

        j += 1

        k += 1

**Insertion Sort**

Insertion sort is a sorting algorithm in which the elements are transferred one at a time to the right position. In other words, an insertion sort helps in building the final sorted list, one item at a time, with the movement of higher-ranked elements.

In an insertion sort, the first element in the array is considered as sorted, even if it is an unsorted array. In an insertion sort, each element in the array is checked with the previous elements, resulting in a growing sorted output list. With each iteration, the sorting algorithm removes one element at a time and finds the appropriate location within the sorted array and inserts it there. The iteration continues until the whole list is sorted.

**Algorithm:**

INSERTION-SORT (A)

for j = 2 to A.length

key = A[j]

// Insert A[j] into the sorted sequence A[1…j-1]

i = j – 1

while i > 0 and A[i] > key

A[i+1] = A[i]

i = i – 1

A[i+1] = key

**Code (insert.py)**

def insertionSort (data):

    length = len(data)

    for i in range(1, length):

        key = data[i]

        j = i - 1

        while ( (j+1) > 0 and data[j] > key):

            data[j+1] = data[j]

            j = j - 1

        data[j+1] = key

**Test Case**

Both the sorting algorithms were tested using unittest test framework to validate that each unit of the software performs as designed.

**Code (test\_sorting.py)**

import unittest

from insert import insertionSort

from merge import mergeSort

class SortingTestCase(unittest.TestCase):

    def test\_insertion\_sort(self):

        input = [3, 1, 9, 4, 2, 7, 6, 10, 8, 5]

        output = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

        insertionSort(input)

        self.assertListEqual(input, output)

    def test\_merge\_sort(self):

        input = [3, 1, 9, 4, 2, 7, 6, 10, 8, 5]

        output = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

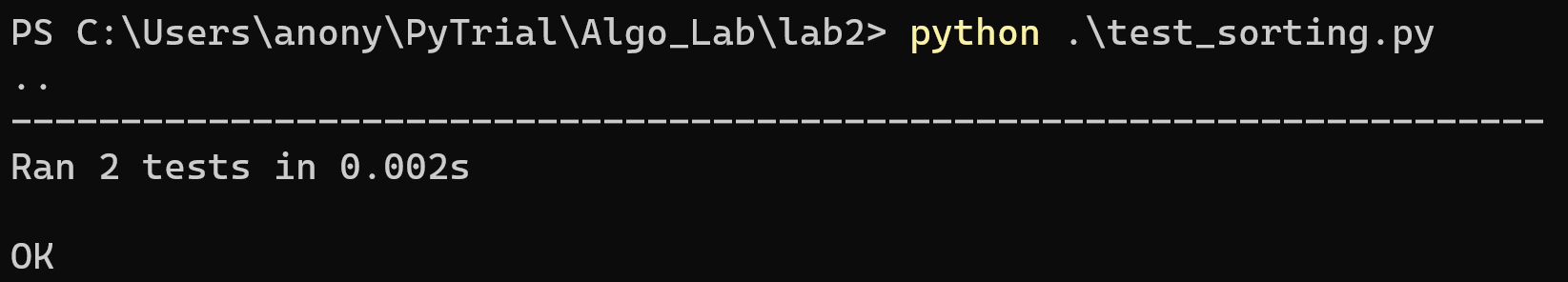
        mergeSort(input, 0, len(input)-1)

        self.assertListEqual(input, output)

if \_\_name\_\_ == "\_\_main\_\_":

    unittest.main()

**Output:**

****

**Main Program (main.py)**

from random import sample

from time import time

from insert import insertionSort

from merge import mergeSort

import matplotlib.pyplot as plt

def runInsert(n):

    data = sample(range(n+1), n)

    start\_time = time()\*1000

    insertionSort(data)

    end\_time = time()\*1000

    time\_taken\_insertion = end\_time - start\_time

    print(f"\nTime taken for insertion sort of {n} data = {time\_taken\_insertion} ms")

    return time\_taken\_insertion

def runMerge(n):

    data = sample(range(n+1), n)

    start\_time = time()\*1000

    mergeSort(data, 0, len(data) - 1)

    end\_time = time()\*1000

    time\_taken\_merge = end\_time - start\_time

    print(f"Time taken for merge sort of {n} data = {time\_taken\_merge} ms")

    return time\_taken\_merge

if \_\_name\_\_ == "\_\_main\_\_":

    inpSize = []

    execTimeInsert = []

    execTimeMerge = []

    for i in range(0, 25001, 5000):

        inpSize.append(i)

        execTimeInsert.append(runInsert(i))

        execTimeMerge.append(runMerge(i))

    plt.xlabel("Input Size")

    plt.ylabel("Execution Time")

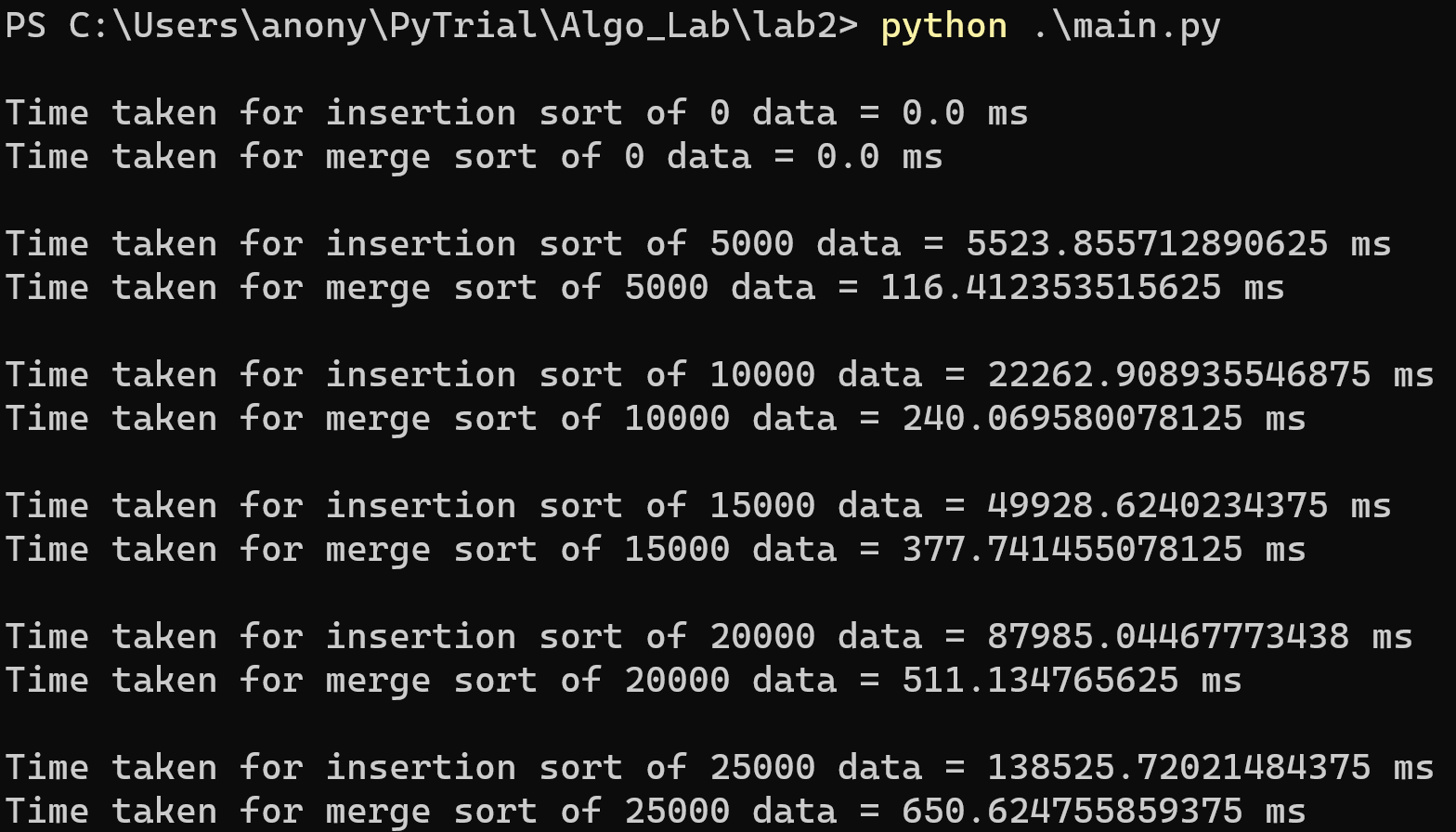
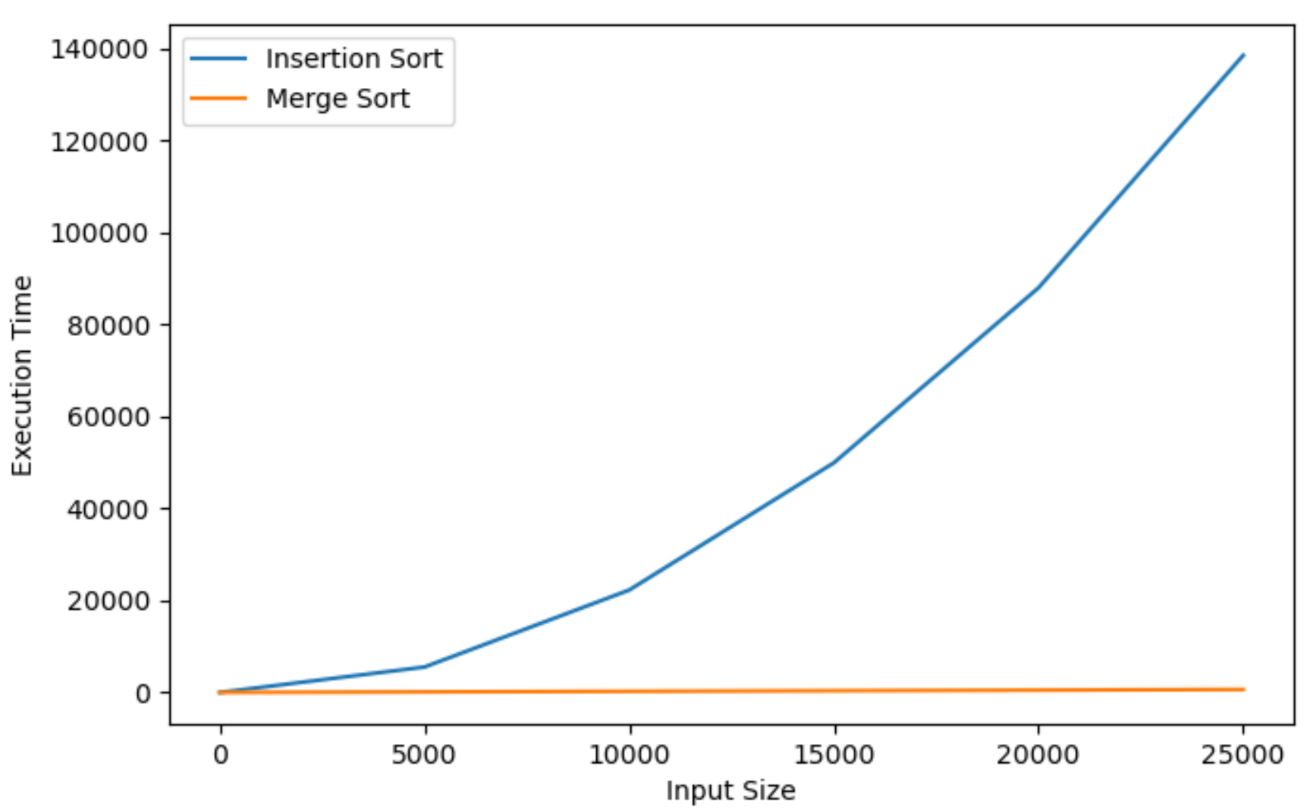
    plt.plot(inpSize, execTimeInsert, label="Insertion Sort")

    plt.plot(inpSize, execTimeMerge, label="Merge Sort")

    plt.legend()

    plt.show()

**Output:**

****

*Fig: Input Size vs. Execution Time*

**Observation:**

From the above program and the obtained graph, we can clearly see that merge sort is much more efficient and saves time compared to insertion sort. As the number of data to be sorted increases, the insertion sort takes quite a lot of time, whereas the merge sort performs it in a short amount of time, as we can see from the output of the main program.

The merge sort takes O (n log n) time in all cases, i.e., best, worst and average cases. Whereas, the insertion sort takes O (n) time in best case, and O(n2) time in worst and average cases.