Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

A lab Report 2

On

"Algorithm and Complexity"

[Course Code: COMP 314]

Submitted By:

Sabil Shrestha (51)

Submitted To:

Rajni Mam

Submission Date: 8th November, 2020

Sorting Algorithms

Merge Sort

Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being O (n log n), it is one of the most respected algorithms. Merge sort first divides the array into equal halves and then combines them in a sorted manner 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.

Merge sort keeps on dividing the list into equal halves until it can no more be divided. By definition, if it is only one element in the list, it is sorted. Then, merge sort combines the smaller sorted lists keeping the new list sorted too.

Algorithm:

MERGE-SORT (A, p, r)

```
if p < r
        q = ⌊ (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

Source Code

```
# Merge sort code
def mergeSort(A, p, r):          #merge sort
    if p < r:
        q = int((p+r-1)/2)       # partition between p-r

        mergeSort(A, p, q)       # the part from p to q is here
        mergeSort(A, q + 1, r)  # the part from after q that is q
+1 to r is here
        merge(A, p, q, r)

def merge(A, p, q, r):
    n1 = q - p + 1      #as A starts from 0 so +1 is added
    n2 = r - q

    L = [0] * (n1)      # making L and R initialize
    R = [0] * (n2)

    for i in range(n1):
        L[i] = A[p + i]
    for j in range(n2):
        R[j] = A[q + j + 1]

    i = 0
    j = 0
    k = p

    while (i < n1 and j < n2):
        if (L[i] <= R[j]):      #if left side one is smaller then
 right the final array
            A[k] = L[i]         #taken first
            i += 1
        else:
            A[k] = R[j]         # if right is smaller then it is
taken first
```

```
            j += 1
        k += 1

    while (i < n1):
        A[k] = L[i]
        i += 1
        k += 1

    while (j < n2):
        A[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. It is the sorting mechanism where the sorted array is built having one item at a time. The array elements are compared with each other sequentially and then arranged simultaneously in some particular order. The analogy can be understood from the style we arrange a deck of cards. This sort works on the principle of inserting an element at a particular position, hence the name Insertion Sort.

The first step involves the comparison of the element in question with its adjacent element. And if at every comparison reveals that the element in question can be inserted at a particular position, then space is created for it by shifting the other elements one position to the right and inserting the element at the suitable position. The above procedure is repeated until all the element in the array is at their apt position.

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
```

Source code

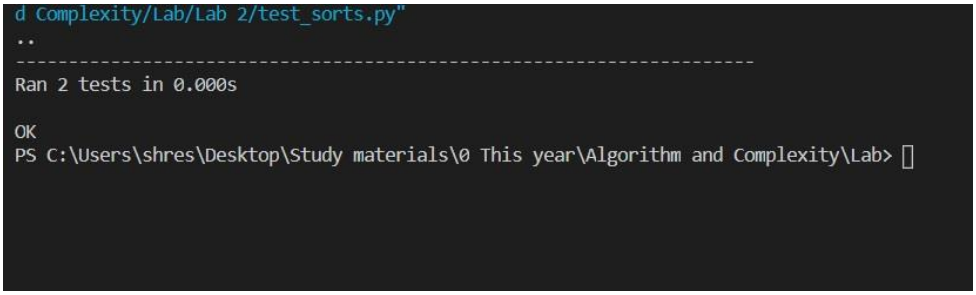
```python
def insertionSort (data):
    length = len(data)

    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

The test of the algorithm was done using python unittest. The test was successfully done. Entering wrong value in output made test fail which is exactly what we wanted. The below test worked well, which is also shown in the picture below the code.

```
import unittest
from insertion_sort import insertionSort
from merge_sort import mergeSort

class SortingTestCase(unittest.TestCase):
    def test_insertion_sort(self):
        input = [30, 10, 20, 5, 25, 15,35,45,40]
        output = [5,10,15,20,25,30,35,40,45]

        insertionSort(input)

        self.assertListEqual(input, output)

    def test_merge_sort(self):
        input = [30, 10, 20, 5, 25, 15,35,45,40]
        output = [5,10,15,20,25,30,35,40,45]

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

        self.assertListEqual(input, output)

if __name__ == "__main__":
    unittest.main()
```

Output



Main Program

```
from random import sample
from time import time
from insertion_sort import insertionSort
from merge_sort 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_t
aken_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_m
erge} 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:

The output was successfully compiled and using matplotlib the following graph was determined and plotted in graph.

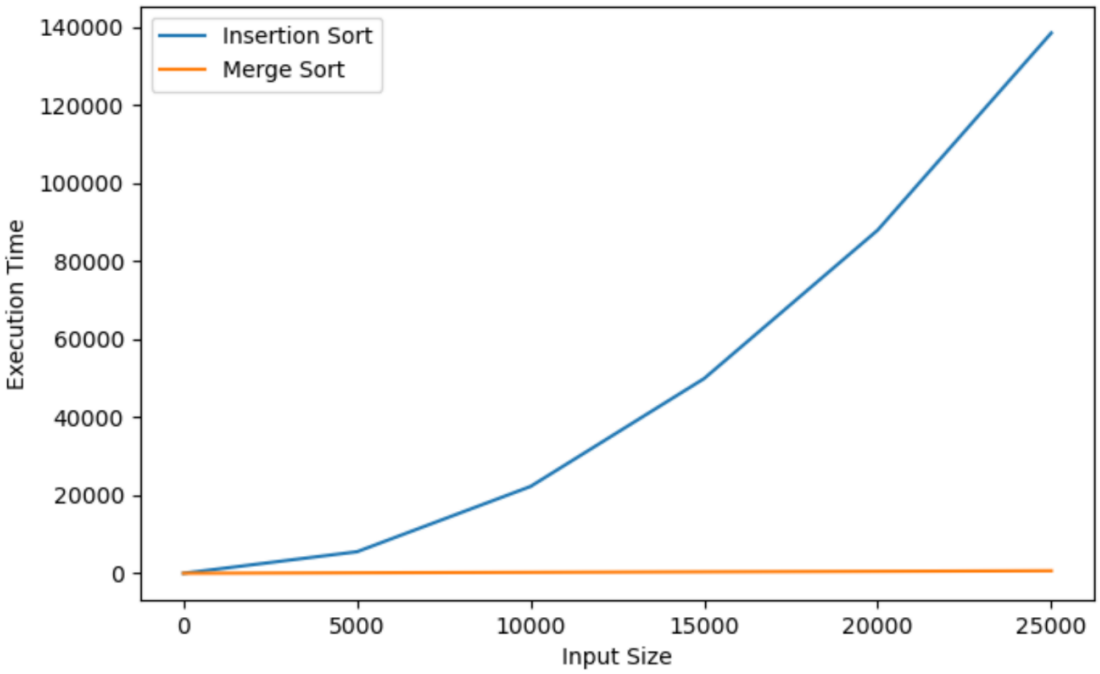


Fig: Input Size vs. Execution Time

Observation

The merge sort algorithm took shorter time compared with insertion sort. The merge sort takes O (nlogn) time complexity in both best and worst cases and that is why it is uniform compared to insertion sort which can be seen in graph by flat line. The insertion sort takes o(n) time complexity in best case scenario when the data is already arranged, and in takes O(n^2) complexity in worst and average cases. The time complexity is given in graph which is very different in nature to merge sort complexity for 25000 input size. As the data increases, the insertion sort takes longer time compared to merge sort.