Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

LAB Work 2

Course Code: COMP 314

Submitted By

Name: Amar Kumar Mandal

Roll No: 58

Level: 3rd year/ 2nd Semester

Group: CE

Submitted To

Name: Rajani Chulyadyo, PhD

DoCSE

Department of Computer Science and Engineering

Submission Date: March 26th, 2022

1. Implementation, testing and performance measurement of sorting algorithms

A. Implement the following sorting algorithms:
   a. Insertion sort
   b. Merge sort

Python Implementation of Insertion sort

Insertion sort is a simple sorting algorithm that works similar to the way we sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

insertion_sort.py > ...

```python
def insertionSort(theSeq):
    n = len(theSeq)

    for i in range(1, n):

        value = theSeq[i]

        pos = i

        while pos > 0 and value < theSeq[pos - 1]:
            theSeq[pos] = theSeq[pos - 1]
            pos -= 1

        theSeq[pos] = value


    return theSeq

```

Here, *line 1* defines the **insertionSort Func** which takes the list to be sorted. While loop at *line 10* compares each element from the unsorted element virtual list with every element of the sorted list at the left side. Finally, after the end of the for loop we can obtain the sorted list.

Python Implementation of Merge Sort

Merge Sort is a Divide and Conquer algorithm. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves. The merge() function is used for merging two halves. Below is the python implementation for details

```
def recMergeSort(theSeq, first, last, tmpArray):

    if first == last:
        #single element list if sorted
        return;
    else:
        mid = (first + last) // 2

        recMergeSort(theSeq, first, mid, tmpArray)
        recMergeSort(theSeq, mid + 1, last, tmpArray)

        return mergeVirtualSeq (theSeq, first, mid + 1, last + 1, tmpArray)
```

Fig: Recursive merge sort algorithm

```
18  def mergeVirtualSeq(theSeq, left, right, end, tmpArray):
19      a = left
20      b = right
21
22      m = 0
23
24      while a < right and b < end:
25          if theSeq[a] < theSeq[b]:
26              tmpArray[m] = theSeq[a]
27              a += 1
28
29          else:
30              tmpArray[m] = theSeq[b]
31              b += 1
32
33          m += 1
34
35
36      while a < right:
37          tmpArray[m] = theSeq[a]
38          a += 1
39          m += 1
40
41      while b < end:
42          tmpArray[m] = theSeq[b]
43          b +=1
44          m += 1
45
46      for i in range (end - left):
47          theSeq[i + left] = tmpArray[i]
48
49      return theSeq
50
```

Fig: Merge Function

```
51
52  def mergeSort(theSeq):
53      n = len(theSeq)
54
55      tmpArray = [0] * n
56
57      return recMergeSort(theSeq, 0, n-1, tmpArray)
58
59
60
```

Fig: Wrapper Function

2. Write some test cases to test your program.

Test Case for Sort

```
 9  class TestSortCase(unittest.TestCase):
10      def __init__(self, *args, **kwargs):
11          super(TestSortCase, self).__init__(*args, **kwargs)
12          self.sorted_value = [1, 2, 3, 4, 5]
13
14      def test_insertion_sort_reverse(self):
15          reverse_array = [5, 4, 3, 2, 1]
16          self.assertListEqual(insertionSort(reverse_array), self.sorted_value)
17
18      def test_insertion_sort_sorted(self):
19          self.assertListEqual(insertionSort(self.sorted_value), self.sorted_value)
20
21
22      def test_insertion_sort_random(self):
23          random_value = [1, 2, 3, 4, 5]
24          random.shuffle(random_value)
25          self.assertListEqual(insertionSort(random_value), self.sorted_value)
26
27      def test_merge_sort_reverse(self):
28          reverse_array = [5, 4, 3, 2, 1]
29          self.assertListEqual(mergeSort(reverse_array), self.sorted_value)
30
31      def test_merge_sort_sorted(self):
32          self.assertListEqual(mergeSort(self.sorted_value), self.sorted_value)
33
34      def test_merge_sort_random(self):
35          random_value = [1, 2, 3, 4, 5]
36          random.shuffle(random_value)
37          self.assertListEqual(mergeSort(random_value), self.sorted_value)
38
```

Fig: Test case for merge sort and insertion sort

Output of the test

```
(dsa-lab) amar@white-shadow:$ python test_sort.py
......
----------------------------------------------------------------------
Ran 6 tests in 0.001s

OK
(dsa-lab) amar@white-shadow:$
```

Figure: Test Case Output

We can observe that the program passes all the test cases defined under **TestSortCase** class. Therefore, we can conclude that the program is working as expected.

3. Generate some random inputs for your program and apply both insertion sort and merge sort algorithms to sort the generated sequence of data. Record the execution times of both algorithms for inputs of different sizes. Plot an input-size vs execution-time graph.

Graph Plot For Insertion Sort

→ Input sizes ranges from **100 to 2000** at an interval of **10**

Output

Fig (a): Time complexity of an Insertion Sort algorithm

Output (Best case Insertion sort)

Figure (b): Best Case Time complexity of Insertion Sort algorithm

Graph Plot For Merge Sort

→ Input sizes range from **100 - 10000** at an interval of **100**

Output

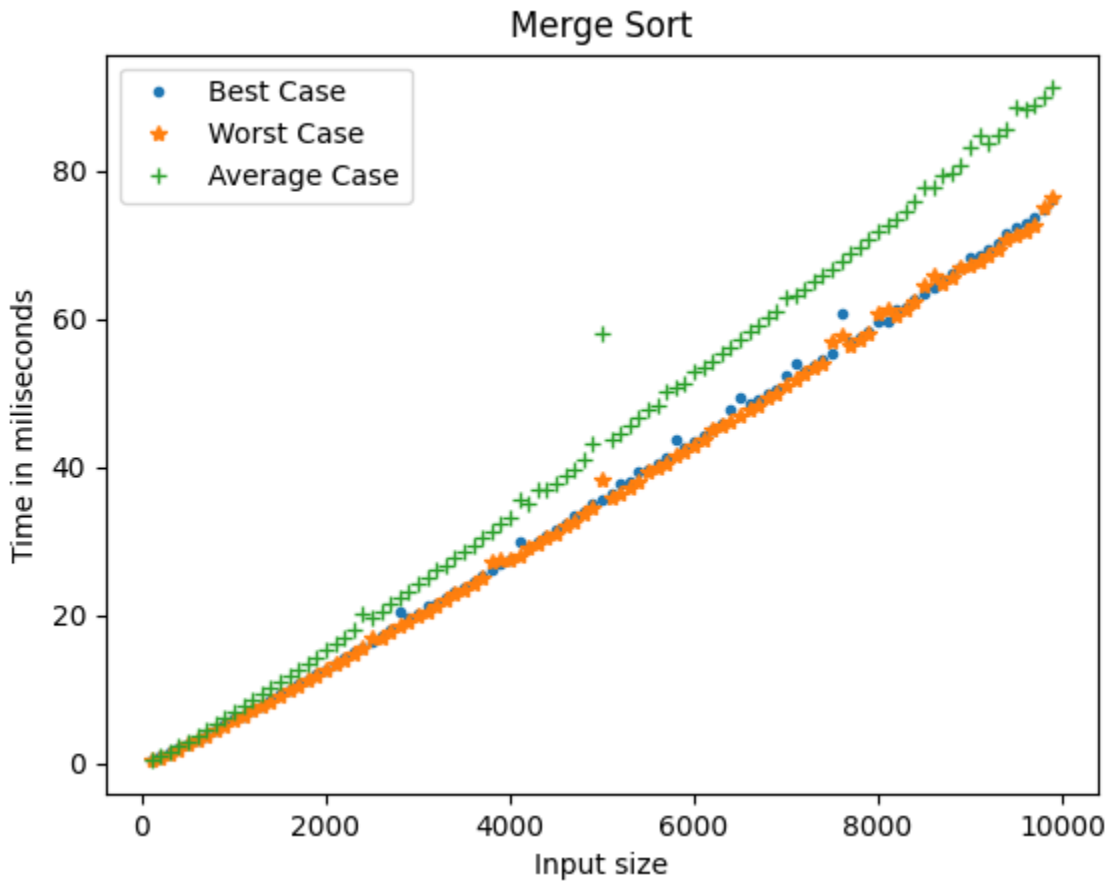

Figure (c): Time complexity of Merge Sort algorithm

4. Explain your observations

→ Figure (a) shows the worst case and average case time complexity of an **insertion sort** algorithm as input size increases. As we know that the time complexity of an **insertion sort** algorithm in the best case scenario is **O(n)**, the same can be seen on the Figure(b). As the size of input increases the running time increases in the linear fashion as indicated by the blue line. Similarly, for the **worst case** and **average case** the time complexity of an **insertion sort** algorithm is **O(n^2),** that is why as the size of input increases the running time increases in a non-linear fashion i.e quadratic manner.

→ Figure (c) shows the best case, worst case and average case time complexity of the **merge sort** algorithm as the input size increases. The worst case, best case and average case time complexity of merge sort algorithms is **O(n* logn)**. So, the curve that is obtained after plotting the input size vs time value is the combination of linear and logarithmic graphs as depicted in the figure above.