Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

A Lab Report

On

"Sorting"

[Course Code: COMP 314]

Submitted by:

Ankit Khatiwada (24)

Submitted to:

Dr. Rajani Chulyadyo

Department of Computer Science and Engineering

20th March, 2022

Table of Contents

Objective

The task for the second lab was to write a program to sort the data in the list. For this, we used two sorting algorithms: insertion sort and merge sort and compared the time complexities of the two algorithms.

Procedure

At first, code for both insertion and merge sort was written. This was then tested using python’s unit test library. Tests ran successfully and the algorithms were validated. For merge sort we tested the merge algorithm also. Then moving towards to test the performance of both algorithms. For both sorts, 10 to 1000 data points with an increment of 10 each step were generated and simulated for two cases: Best Case and Worst-Case for insertion sort and only best case for merge sort since we cannot predict what the worst case for merge sort be. For Best Case of insertion sort, we gave the algorithm to sort the already sorted list. For the worst case, we gave it the reverse sorted list.

For merge sort Best Case, the already sorted list or reverse sorted list can be given. In the lab already sorted list was given. We cannot predict the data for the worst case of merge sort algorithm since the worst case is when the merge algorithm is worst i.e. all the elements in the array must shift their position.

The snapshot of the algorithms are as listed below:

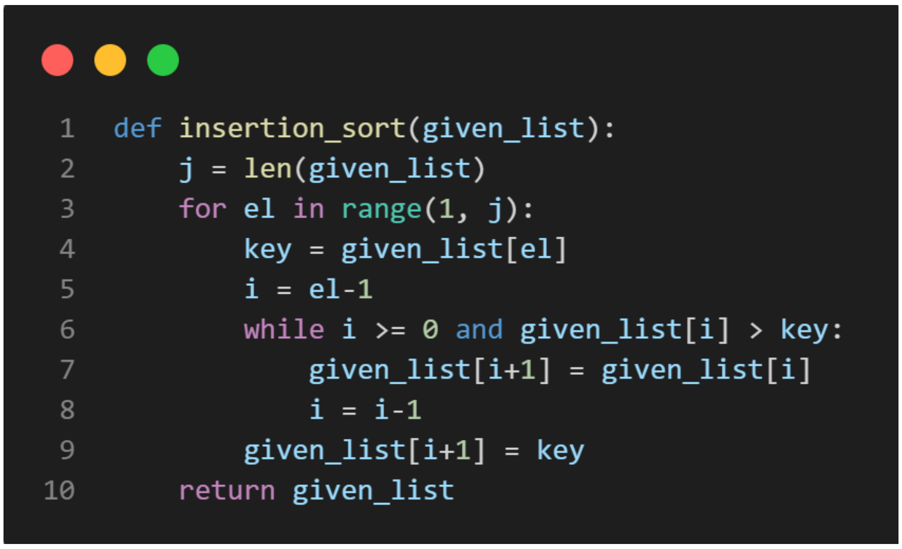

```python
def insertion_sort(given_list):
    j = len(given_list)
    for el in range(1, j):
        key = given_list[el]
        i = el-1
        while i >= 0 and given_list[i] > key:
            given_list[i+1] = given_list[i]
            i = i-1
        given_list[i+1] = key
    return given_list
```

Fig 1: Algorithm for Insertion Sort

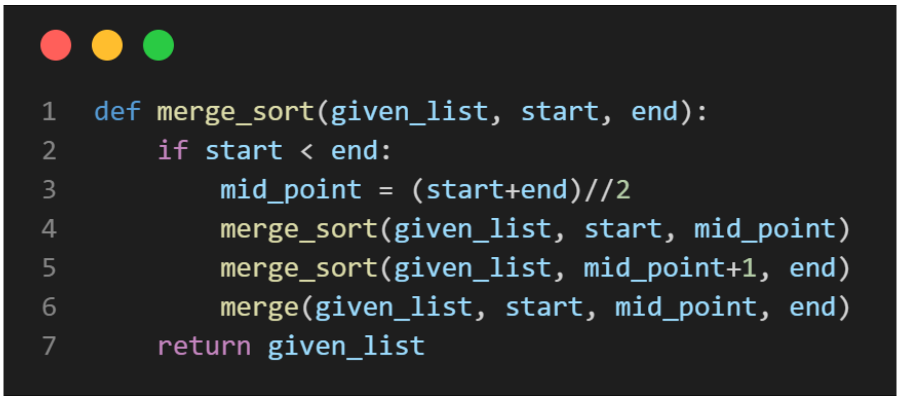
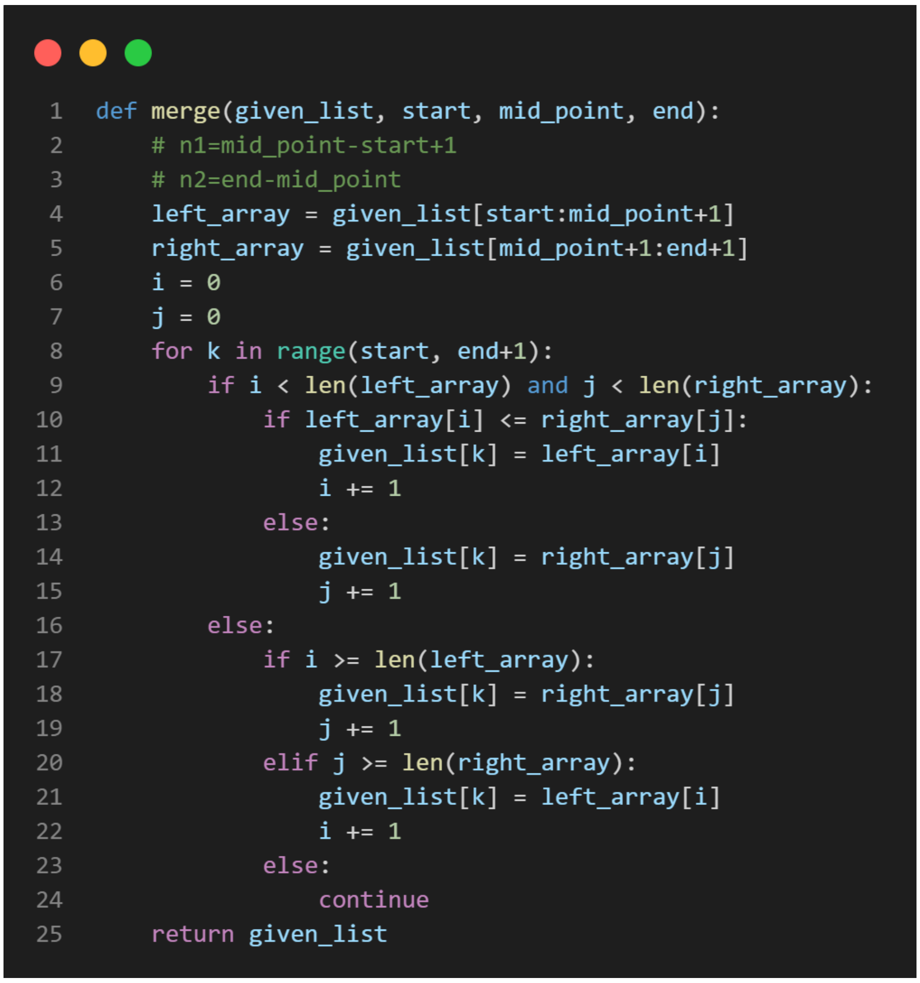

```python
def merge_sort(given_list, start, end):
    if start < end:
        mid_point = (start+end)//2
        merge_sort(given_list, start, mid_point)
        merge_sort(given_list, mid_point+1, end)
        merge(given_list, start, mid_point, end)
    return given_list
```

Fig 2: Algorithm for Merge Sort

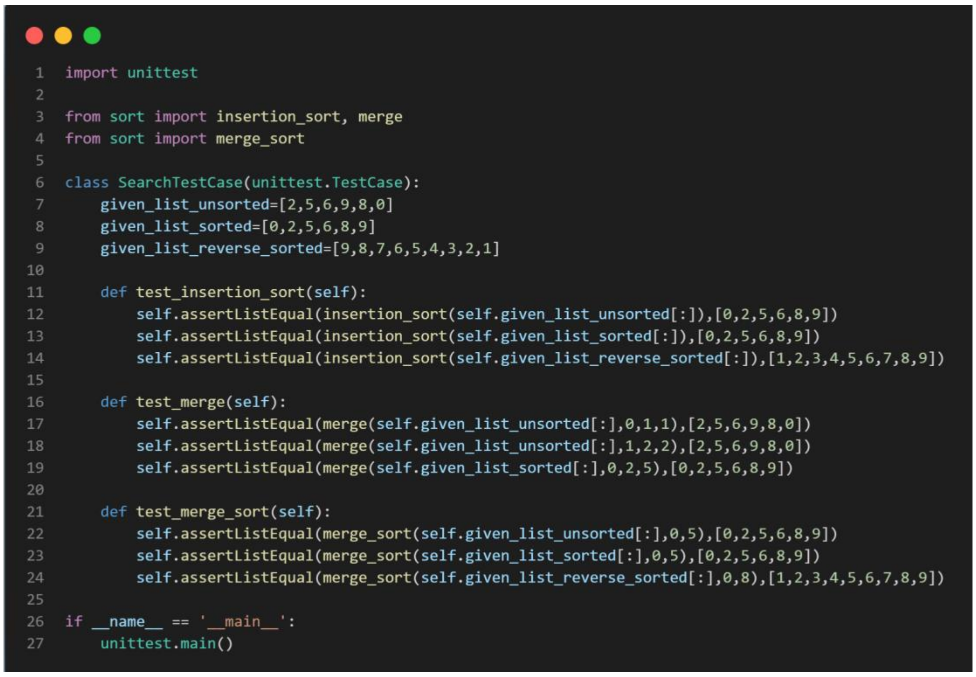


```python
def merge(given_list, start, mid_point, end):
    # n1=mid_point-start+1
    # n2=end-mid_point
    left_array = given_list[start:mid_point+1]
    right_array = given_list[mid_point+1:end+1]
    i = 0
    j = 0
    for k in range(start, end+1):
        if i < len(left_array) and j < len(right_array):
            if left_array[i] <= right_array[j]:
                given_list[k] = left_array[i]
                i += 1
            else:
                given_list[k] = right_array[j]
                j += 1
        else:
            if i >= len(left_array):
                given_list[k] = right_array[j]
                j += 1
            elif j >= len(right_array):
                given_list[k] = left_array[i]
                i += 1
            else:
                continue
    return given_list
```

Fig 3: Algorithm for Merge

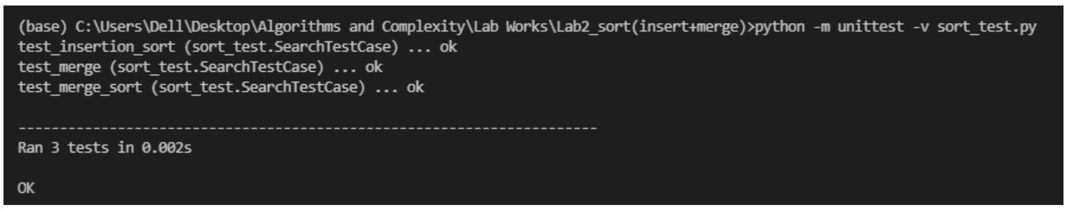
Test case for algorithms

The unittest standard library of the Python programming language was used for performing unit tests on the search algorithms as shown in the figure below. A random self generated list was taken and tests were performed to sort the data in the list. Test methods for both the algorithms were passed and the test was validated. The test for merge algorithm was also validated:



```python
import unittest

from sort import insertion_sort, merge
from sort import merge_sort

class SearchTestCase(unittest.TestCase):
    given_list_unsorted=[2,5,6,9,8,0]
    given_list_sorted=[0,2,5,6,8,9]
    given_list_reverse_sorted=[9,8,7,6,5,4,3,2,1]

    def test_insertion_sort(self):
        self.assertListEqual(insertion_sort(self.given_list_unsorted[:]),[0,2,5,6,8,9])
        self.assertListEqual(insertion_sort(self.given_list_sorted[:]),[0,2,5,6,8,9])
        self.assertListEqual(insertion_sort(self.given_list_reverse_sorted[:]),[1,2,3,4,5,6,7,8,9])

    def test_merge(self):
        self.assertListEqual(merge(self.given_list_unsorted[:],0,1,1),[2,5,6,9,8,0])
        self.assertListEqual(merge(self.given_list_unsorted[:],1,2,2),[2,5,6,9,8,0])
        self.assertListEqual(merge(self.given_list_sorted[:],0,2,5),[0,2,5,6,8,9])

    def test_merge_sort(self):
        self.assertListEqual(merge_sort(self.given_list_unsorted[:],0,5),[0,2,5,6,8,9])
        self.assertListEqual(merge_sort(self.given_list_sorted[:],0,5),[0,2,5,6,8,9])
        self.assertListEqual(merge_sort(self.given_list_reverse_sorted[:],0,8),[1,2,3,4,5,6,7,8,9])

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

Fig 4: Test cases for Insertion sort and Merge Sort

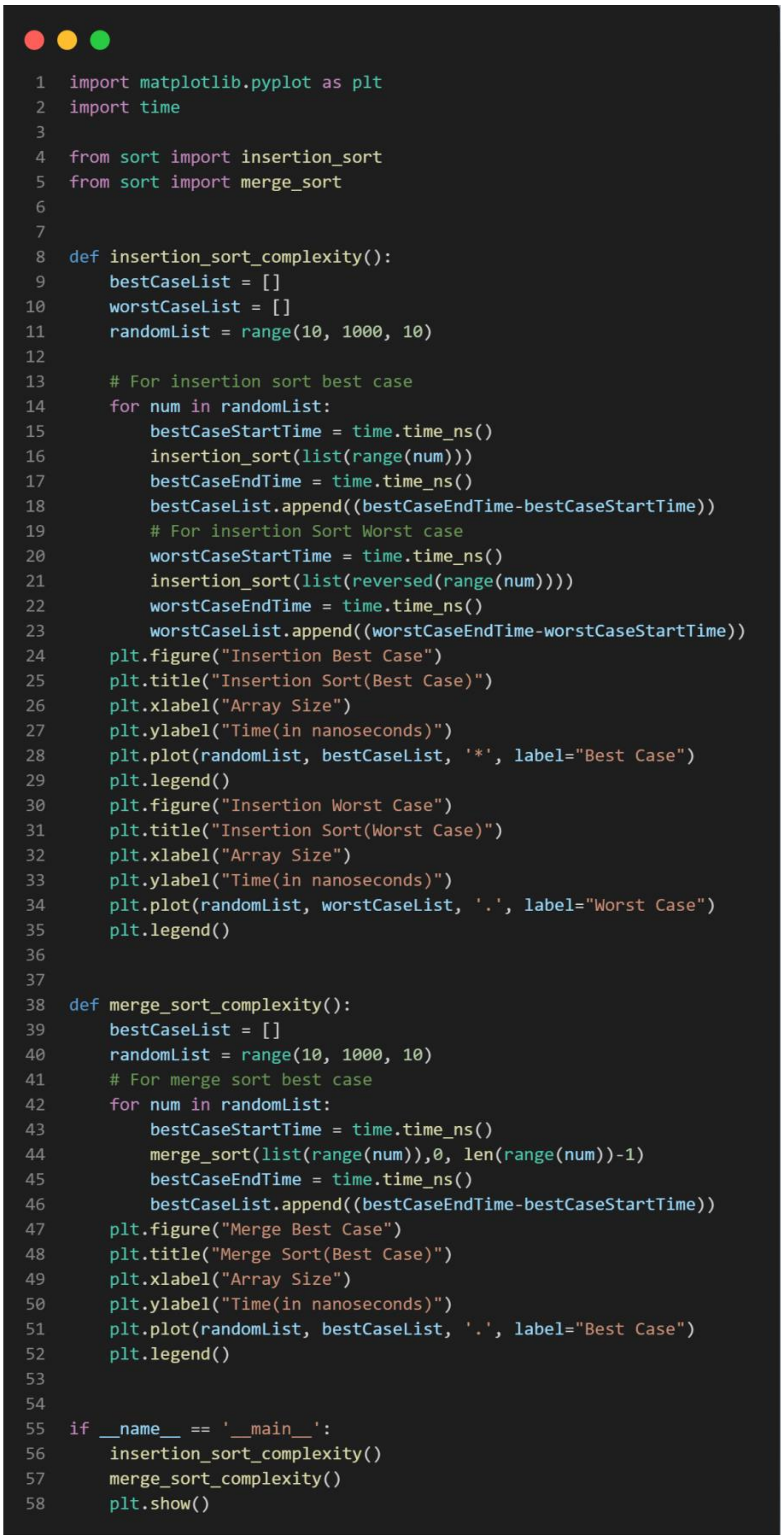


```
(base) C:\Users\Dell\Desktop\Algorithms and Complexity\Lab Works\Lab2_sort(insert+merge)>python -m unittest -v sort_test.py
test_insertion_sort (sort_test.SearchTestCase) ... ok
test_merge (sort_test.SearchTestCase) ... ok
test_merge_sort (sort_test.SearchTestCase) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.002s

OK
```

Fig 5: Result of unit test

Plotting the running time of algorithms vs number of data

Arrays with varying sizes were used to perform both insertion and merge sort for each array. Arrays of length ranging from 10 to 1000 were taken at an interval of 10 for both sorts.



```python
import matplotlib.pyplot as plt
import time

from sort import insertion_sort
from sort import merge_sort


def insertion_sort_complexity():
    bestCaseList = []
    worstCaseList = []
    randomList = range(10, 1000, 10)

    # For insertion sort best case
    for num in randomList:
        bestCaseStartTime = time.time_ns()
        insertion_sort(list(range(num)))
        bestCaseEndTime = time.time_ns()
        bestCaseList.append((bestCaseEndTime-bestCaseStartTime))
        # For insertion Sort Worst case
        worstCaseStartTime = time.time_ns()
        insertion_sort(list(reversed(range(num))))
        worstCaseEndTime = time.time_ns()
        worstCaseList.append((worstCaseEndTime-worstCaseStartTime))
    plt.figure("Insertion Best Case")
    plt.title("Insertion Sort(Best Case)")
    plt.xlabel("Array Size")
    plt.ylabel("Time(in nanoseconds)")
    plt.plot(randomList, bestCaseList, '*', label="Best Case")
    plt.legend()
    plt.figure("Insertion Worst Case")
    plt.title("Insertion Sort(Worst Case)")
    plt.xlabel("Array Size")
    plt.ylabel("Time(in nanoseconds)")
    plt.plot(randomList, worstCaseList, '.', label="Worst Case")
    plt.legend()


def merge_sort_complexity():
    bestCaseList = []
    randomList = range(10, 1000, 10)
    # For merge sort best case
    for num in randomList:
        bestCaseStartTime = time.time_ns()
        merge_sort(list(range(num)),0, len(range(num))-1)
        bestCaseEndTime = time.time_ns()
        bestCaseList.append((bestCaseEndTime-bestCaseStartTime))
    plt.figure("Merge Best Case")
    plt.title("Merge Sort(Best Case)")
    plt.xlabel("Array Size")
    plt.ylabel("Time(in nanoseconds)")
    plt.plot(randomList, bestCaseList, '.', label="Best Case")
    plt.legend()


if __name__ == '__main__':
    insertion_sort_complexity()
    merge_sort_complexity()
    plt.show()
```

Fig 6: Code for plotting the best and worst case time complexities insertion sort and best case of merge sort

The graph for time complexities was then plotted form the matplotlib package of Python as shown in the figure below

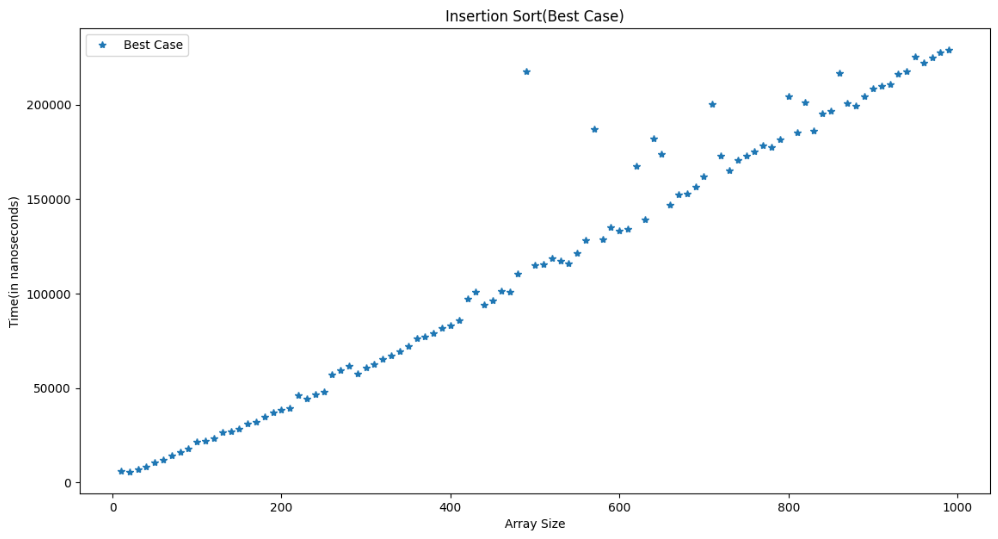


Fig 7: Running time graph of best case of Insertion Sort [O(n)]

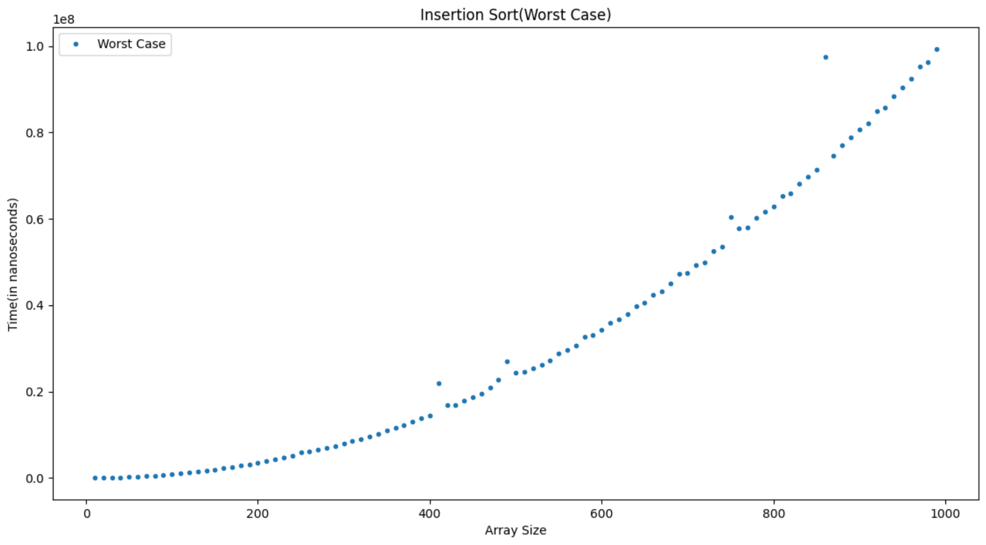


Fig 8: Running time graph of worst case of Insertion Sort [O(n^2)]

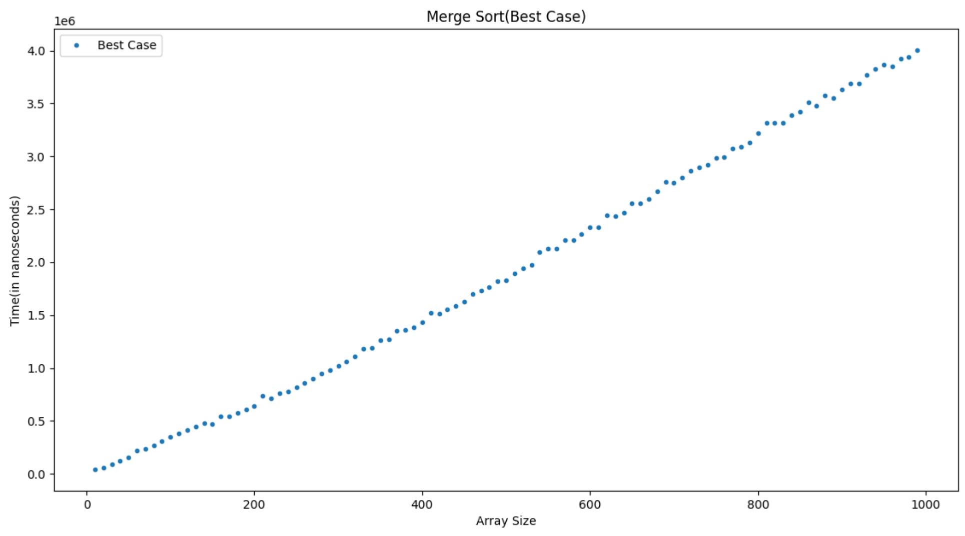


Fig 9: Running time graph of best case of Merge Sort [O(nlogn)]

Conclusion

From the above graphs we can see that, the best case for insertion sort is when the list is already sorted for which we get the complexity of O(n) i.e. time grows linearly with data. Similarly, the worst case is when the given list is reverse sorted due to which the complexity becomes O(n^2).

The best case complexity of merge is when the list is already sorted or reverse sorted and the complexity is O(nlogn). The worst case complexity cannot be exactly determined since we cannot predict the list for which every element moves during the merge operation.

Appendix

Source Code: https://github.com/ch-ankit/algo-and-complexity