Name: Vishal Ramane
Roll.No: 2019430010

Experiment No.1

Aim:
Sorting a very large number of elements using various sorting methods and file operations, and analyzing time complexities of these sorting algorithms.

Problem Statement:
Analyzing and comparing the time complexities of classical sorting algorithms (Quick sort, Insertion sort, Heap Sort and Merge sort) against a randomly generated, very large data-set (4.5 lac) of positive integer elements, with help of file handling operations.

Objective:
- Random generation of 4.5 lac elements.
- Implementation of File Handling concepts in Python3/Java/C++ programming language and Storing these elements into file.
- Implementation of Quick Sort, Insertion Sort, Heap Sort and Merge Sort in Python3/Java/C++ programming language and sorting the elements in file.
- Comparing and Analyzing Time Complexity of Quick Sort, Insertion Sort, Heap Sort and Merge Sort.

Methodology:
I. 4.5 Lac elements (positive-integers) are randomly generated and are written into a file.
II. The data in the file is then equally divided into 6 files each containing 75,000 (75K) elements.
III. Each of these 6 file is sorted using the sorting algorithm, thus creating a set of 6-sorted files.
IV. A pair of these sorted file is then merged creating a new file of 150K elements, which is again sorted and stored in the file.
V. Step IV is repeated 3 Times creating a new set of 3 files each containing 150K elements.
VI. A pair of these file (i.e, 150K) is then merged, elements are sorted and stored into a new file. (now containing 300K elements).
VII. Lastly we have 2 files containing 300K, 150K elements respectively, same process as mentioned in step IV is repeated, giving a single file containing 450K sorted elements.

Steps III to VII are repeated for each of 4 sorting algorithms and results are stored externally for comparison.

Name: Vishal Ramane
Roll.No: 2019430010

Experiment No.1

Implementation:

- The Experiment is implemented using ***Python3*** programming language (version ***3.7.3***) on Windows-10 platform.
- Libraries and Modules Used:
  a. ***matplotlib.pyplot*** : python library for MATLAB like graph plotting
  b. ***random*** : provides methods for random variable generation.
  c. ***time :*** a stub file for generating time stamps.
  d. ***os*** : os module is used to invoke
- Source code is kept modular with each sorting algorithm kept in separate module.viz, { *app.py, quickSort.py ,mergeSort.py,insertionSort.py,heapSort.py*}
- A function named ***initialize*** is defined for generating random elements and storing them in file.
- Function named ***distributeIntoFiles*** divides the file containing 4.5 lac elements into 6 files, each of which contains 75000 elements. *{file1,file2,....file6}*
- Each module of sorting algorithm has a ***main*** function which acts as a calling function to the sorting procedure and also does file handling. This ***main*** function uses ***os*** module to delete files from current directory after there use and ***time*** module to generate timestamp readings for analysis.

Observations:

- Initially for a file of 75000 unsorted elements **Quick sort** operates in **Best Case O(n.log(n))** , where as thereafter as we merge sorted files and re-sort them, the quick sort by it's nature operates in **Worst Case O(n^2)**, which is supplemented by it's complex partitioning process and recursive nature, thus exhibiting **worst performance of all sorting algorithms.**
- **Insertion sort** as compared to **Heap sort** and **Merge sort** shows worst performance as it has running time complexity of **O(n^2)** as compared to **O(n. log(n))** of aforementioned two sorts.
- **Merge sort** and **Heap sort** shows **almost equal** performance, with merge sort performing slightly better than heap sort due to it's simpler implementation.

Name: Vishal Ramane
Roll.No: 2019430010

Experiment No.1

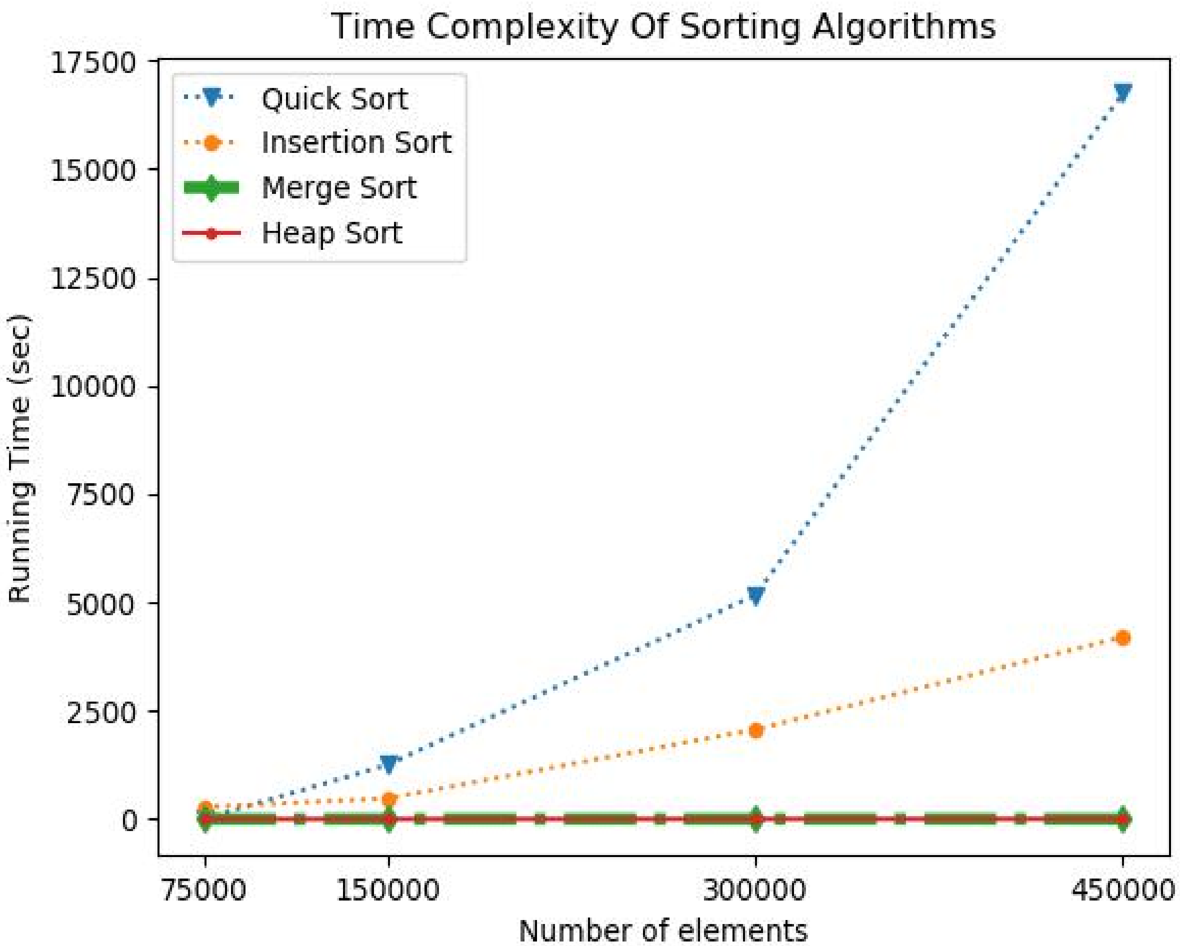
Results:

Quick Sort							
File-Gen	**File-Read**	**75K**	**150K**	**300K**	**450K**	**File-Write**	**Total Time**
1.746	0.1252	0.3150	1251.5	5166.1	16741.6	0.4856	25628.3

Insertion Sort							
File-Gen	**File-Read**	**75K**	**150K**	**300K**	**450K**	**File-Write**	**Total Time**
1.681	0.1406	276.90	477.14	2067.7	4202.5	0.8606	9167.6

Heap Sort							
File-Gen	**File-Read**	**75K**	**150K**	**300K**	**450K**	**File-Write**	**Total Time**
1.729	0.1326	0.6652	1.235	2.671	4.254	0.5226	15.292

Merge Sort							
File-Gen	**File-Read**	**75K**	**150K**	**300K**	**450K**	**File-Write**	**Total Time**
1.618	0.1262	0.7471	0.9213	1.7997	3.0163	0.5027	12.636



Name: Vishal Ramane
Roll.No: 2019430010

Experiment No.1

Conclusions:

Quick sort operates in **Best Case O(n.log(n))** initially , where as thereafter operates in **Worst Case O(n^2)**, which is supplemented by it's complex partitioning process and recursive nature, thus exhibiting **worst performance** out of all sorting algorithms. **Insertion sort** as compared to **Heap sort** and **Merge sort** shows poor performance, **Merge sort** and **Heap sort** exhibit **almost equal** running time, with merge sort performing slightly better than heap sort due to it's simpler implementation.