Simulation of sorting Techniques Documentation

Group Members:

- Deeksha Rao Gorige (**1970751**)
- Sumitha Ravindran (**1978232**)

The five sorting techniques which we have implemented are-

- Bubble Sort
- Insertion Sort
- Selection Sort
- Merge Sort
- Quick Sort

Implementation: All the sorting Algorithms are implemented in JAVA and for plotting the graphs we have implemented the code in PYTHON.

Description of all the Sorting Algorithms-

Bubble Sort -

Bubble sort which is also referred as "Sinking Sort" is one of the simple algorithms to implement. In this algorithm we repeatedly step through the list that is to be sorted, compare each of the adjacent items and swap them if they are in the wrong order.

Algorithm Implementation –

```
BubbleSort (int array [])
    int n = array. length;
   for (int i =0; i<n-1; i++) //run from first cell to last cell
       for (int j = 0; j < n-i-1; j++) //run from first cell to "last cell -iteration"
            if (array[j] > array[j+1]) {
                swap(array[j], array[j+1])
```

Time Complexity –

-Worst and Average Time Complexity for bubble sort – O (n * n). This is when the array is completely in a reverse order.
-Best case Time complexity for bubble sort – O (n). This is when the array is already sorted.

Space Complexity –

For bubble sort we are not using any extra space for sorting. It's an in-place sort. Therefore, the space complexity for bubble sort is O (1).

Usage:
When to use –

- It can be used when the input is already sorted.
- When space is a major concern then this sort can be used.
- When you need an easy implementation to sort elements this type of sorting can be used.

When not to use –

- When time is a concern, we shouldn't use bubble sort as it takes much time.

Insertion Sort –

Insertion Sort, in this type of sorting we basically divide the given array into two parts. That is sorted and unsorted. Then from the unsorted array we pick the first element and find its right postion in the sorted array. This is repeated till the unsorted array is not empty.

Algorithm Implementation –

```
InsertionSort (int array []):
    Loop: j = 1 to n
        Current_value = array[i],
        i = j,
        while (array[i-1] > Current_value && i>0):
            array[i] = array[i-1]
            i- -
        array[i] = Current_value
```

Time Complexity –

-Worst and Average Time Complexity for Insertion sort – O (n * n). This is when the array is completely in a reverse order.
- Time complexity for Insertion sort is fast at its best case. This is when the array is already sorted.

Space Complexity –

For Insertion sort we are not using any extra space for sorting. It's an in-place sort. Therefore, the space complexity for Insertion sort is O (1).

Usage:
When to use –

- When space is a major concern then this sort can be used.
- When you need an easy implementation to sort elements this type of sorting can be used.
- It is best when we have continuous inflow of numbers and we want to keep the list shorted.

When not to use –

- When time is a concern, we shouldn't use Insertion sort as it takes much time.

Selection Sort –

The Selection Sort algorithm is based on the idea of finding the minimum or the maximum element in an unsorted array and then putting it in its correct position in the sorted array

Algorithm Implementation –

```
SelectionSort (int array []):
    Loop: j = 0 to n-1
        int min_val = j;
        loop: i = j+1 to n-1
            if (array[i] < array[min_val])
                min_val = i
       swap(array[j], array[min_val])
```

Time Complexity –

-Worst and Average Time Complexity for Selection sort – O (n * n) and thus making it inefficient on larger lists.
- Time complexity for Selection sort is much worse than similar insertion sort.

Space Complexity –

For Selection sort we are not using any extra space for sorting. It's an in-place sort. Therefore, the space complexity for Selection sort is O (1).

Usage:
When to use –

- When space is a major concern then this sort can be used.
- When you need an easy implementation to sort elements this type of sorting can be used.

When not to use –

- When time is a concern, we shouldn't use Selection sort as it takes much time.

Merge Sort –

Merge Sort is a kind of "Divide and Conquer" technique. Here it divides the input array into two halves, keeps breaking these two halves recursively until the two halves become too small to be broken further. Then the broken parts are merged together to get the final output.

Algorithm Implementation –

```
MergeSort (array, left, right):
    if (right > 1):
        middle = (left + right)/2
        MergeSort (array, left, middle)
        MergeSort (array, middle+1, right)
        Merge (array, left, middle, right)

Merge (array, p, middle, right):
      Create tmp arrays A and B and then copy array, p, middle into A and array, middle+1, right into B
       i = 0, j = 0;
       loop: k = p to right
            if A[i] < B[j]
                array[k] = A[i];
                i++
            else
               array[k] = B[j];
              j++
```

Time Complexity –

-Merge Sort is a recursive algorithm and its time complexity can be expressed in a recurrence relation. After solving the recurrence relation using master theorem or any other method, we get the complexity as Θ(n log n).
-Merge sort is Θ(n log n) in all the cases i.e.. best, worst and average case.

Space Complexity –

-The space complexity for Merge sort will be O(n) as we are creating two temporary arrays in the merge function.

Usage:
When to use –

- When we need a stable sort, we can make use of Merge sort.
- When we want less time for the execution, we can make use of this.

When not to use –

- When space is a concern, we shouldn't use Merge sort as it takes much space.

Quick Sort –

Quick sort is a "Divide and Conquer" algorithm. At every step we find 'pivot' and then make sure all the smaller elements are left of pivot and all bigger elements are right to pivot. It does this recursively until the entire array is sorted.

Algorithm Implementation –

```
QuickSort (array, p, q)
    if (p < q):
       r = partition (array, p, q)
       QuickSort (array, p, r-1)
     QuickSort (array, r+1, p)

Partition (array, p, q) {
    Pivot = q;
    i = p-1
    for (j = p to q):
        if (array[i] < = array[pivot])
            first increment i
            then swap (array[i], array[j])
```

Time Complexity –

-Quick Sort is a recursive algorithm and its time complexity can be expressed in a recurrence relation. After solving the recurrence relation using master theorem or any other method, we get the complexity as Θ(n log n).
-Quick Sort at its worst case takes Θ(n * n)
-Quick Sort at its average case take O (n log n)

Space Complexity –

-The space complexity is O (n) as it makes use of external space during the recursive calls.

Usage:
When to use –

- When we want less time for the execution, we can make use of this.

When not to use –

- When space is a concern, we shouldn't use Quick sort as it takes much space.
- When we need a stable sort, we shouldn't use Quick Sort.

Description of Data Sets-

Synthetic Data Sets-

The Synthetic data sets were obtained using the concepts of **normal** and **exponential** distribution. A normal distribution is a distribution which is symmetric about the mean, showing that data near the mean is more occurring frequently than data far from the mean. Normal distribution is a Bell curve. An exponential Distribution is different from normal distribution. In exponential Distribution events occur continuously and independently at a constant rate.

Real Data Sets-

One of the real-world application data set that we used was "**Sales Record**" which consists Unit sold, Total Revenue, Total Cost, Total Profit etc. From this we have extracted the 'total profit' column. The other real-world data set which we have taken is "Consumer Complaints" which consist of Zip code, submitted status, Data sent, Company, Complain ID etc. From this we have extracted '**complain ID'** column.

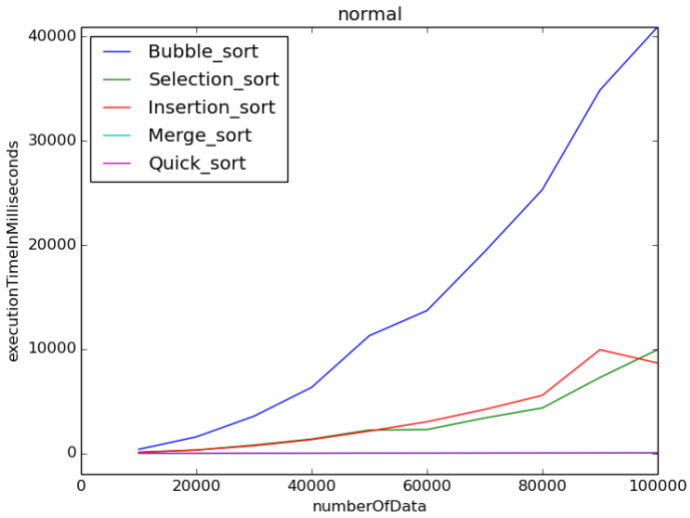
Measure of Sortedness-

The measure of sortedness that we are using for simulation of sorting techniques is '**INVERSION COUNT'**. Inversion count for an array indicates how far an array is sorted. If the array is already sorted, then the value of inversion count is '0' and if the entire array given is in reverse order, then the value of the inversion count is maximum. We have used Runtime class in Java which will tell us the memory usage for a particular process.

Performance Curves-

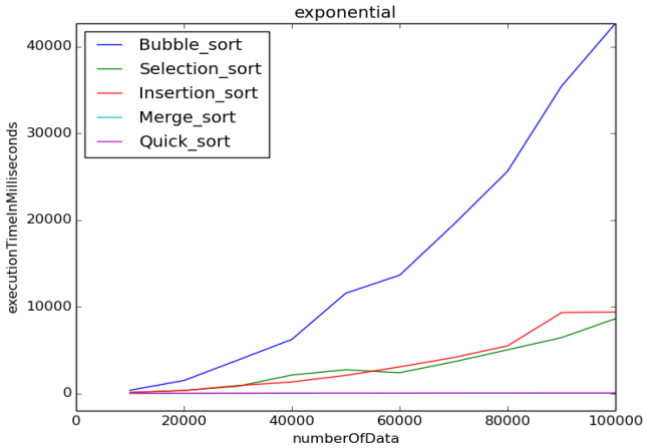
For plotting the performance curves, we have implemented the code in Python with the help of matplotlib. pyplot.

Graph 1- Normal distribution, Data Size v/s Run time



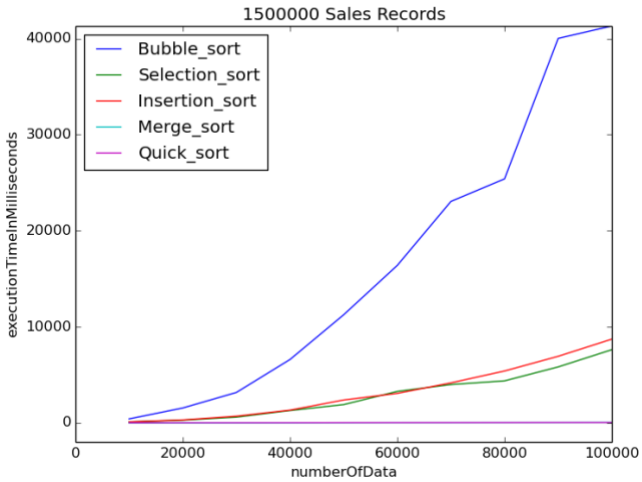
In this graph, the X-axis consists of the input size, and the Y-axis consists of the runtime of the different algorithms. Each sorting algorithm curve is represented in different colors. We have plotted the graph with normal distribution data set. So, from the graph we can say that Quick sort and Merge sort have the least running time. When observed the Selection Sort and Insertion Sort are almost overlapping each other. From the Bubble sort curve, we can see that it is taking more time when compared to other sorting algorithms. It also increases as the size of the data set increases.

Graph 2- Exponential distribution, Data Size v/s Run time



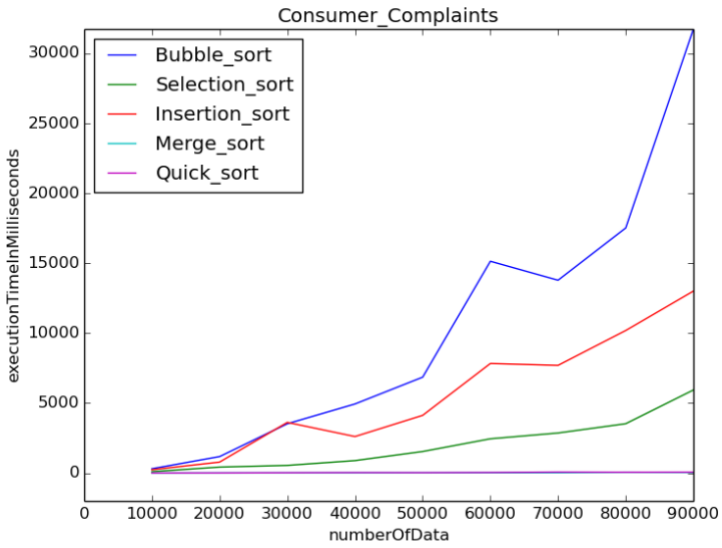
In this graph, the X-axis consists of the input size, and the Y-axis consists of the runtime of the different algorithms. Each sorting algorithm curve is represented in different colors. We have plotted the graph with Exponential distribution data set. So, from the graph we can say that Quick sort and Merge sort have the least running time. When observed the Selection Sort and Insertion Sort are almost overlapping each other. From the Bubble sort curve, we can see that it is taking more time when compared to other sorting algorithms. It also increases as the size of the data set increases.

Graph 3-Real Data#1, Data Size v/s Run time



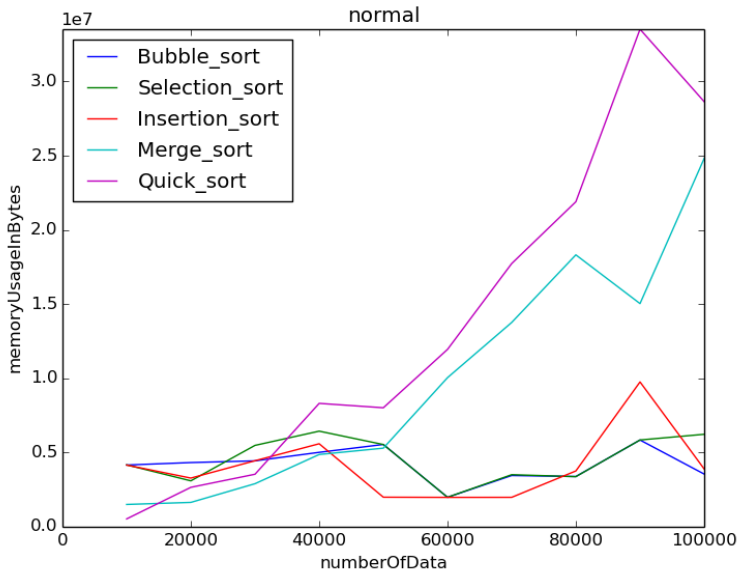
In this graph, the X-axis consists of the input size, and the Y-axis consists of the runtime of the different algorithms. Each sorting algorithm curve is represented in different colors. We have plotted the graph with Real data set. So, from the graph we can say that Quick sort and Merge sort have the least running time. When observed the Selection Sort and Insertion Sort are almost overlapping each other. From the Bubble sort curve, we can see that it is taking more time when compared to other sorting algorithms. It also increases as the size of the data set increases.

Graph 4-Real Data#2, Data Size v/s Run time



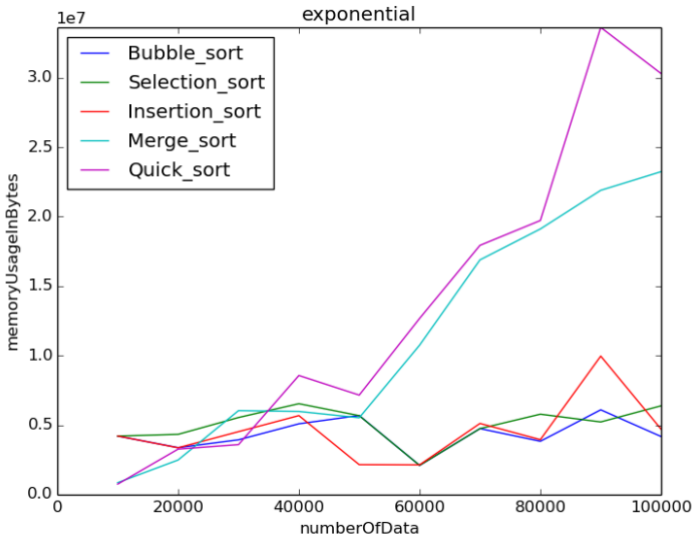
In this graph, the X-axis consists of the input size, and the Y-axis consists of the runtime of the different algorithms. Each sorting algorithm curve is represented in different colors. We have plotted the graph with Real data set. So, from the graph we can say that Quick sort and Merge sort have the least running time. When observed the Selection Sort and Insertion Sort are almost overlapping each other. From the Bubble sort curve, we can see that it is taking more time when compared to other sorting algorithms. It also increases as the size of the data set increases.

Graph 5- Normal distribution, Data Size v/s Memory Usage



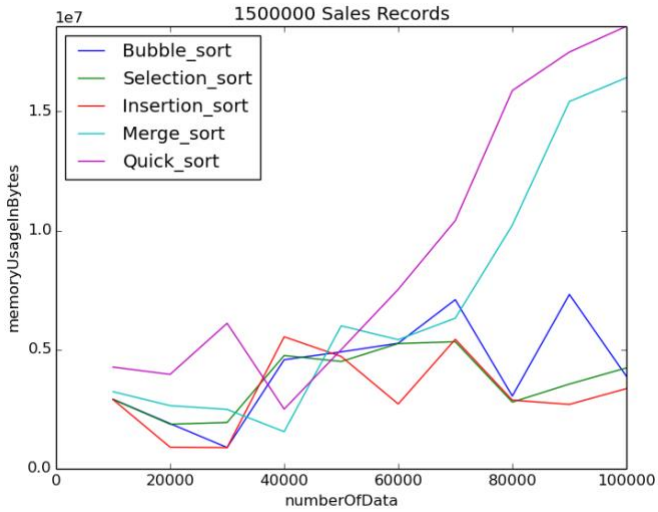
In this graph, we can see that Quick sort is using most memory compared to other sorting algorithms. It is followed by merge sort which also uses the same divide and conquer methodology. We can see that all the remaining algorithms (Insertion, Selection, Bubble) use significantly less memory compared to quick sort and merge sort.

Graph 6- Exponential distribution, Data Size v/s Memory Usage



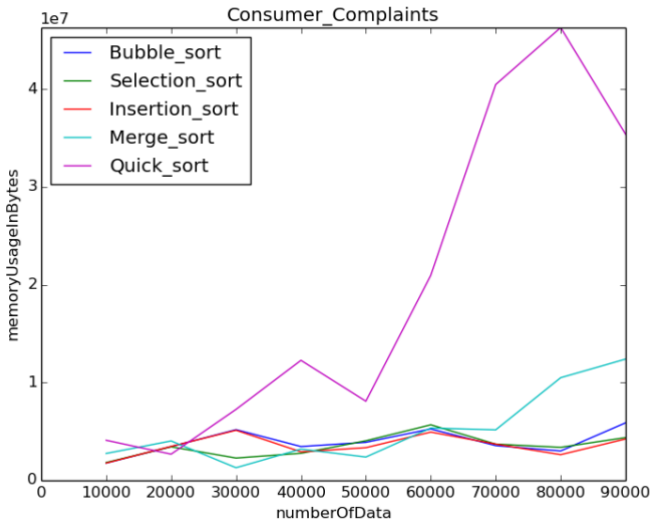
In this graph, we can see that Quick sort is using most memory compared to other sorting algorithms. It is followed by merge sort which also uses the same divide and conquer methodology. We can see that all the remaining algorithms (Insertion, Selection, Bubble) use significantly less memory compared to quick sort and merge sort.

Graph 7-Real Data#1, Data Size v/s Memory Usage



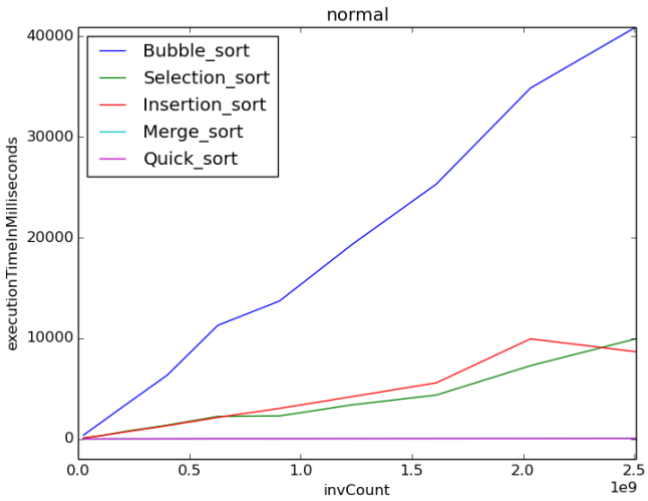
In this graph, we can see that Quick sort is using most memory compared to other sorting algorithms. It is followed by merge sort which also uses the same divide and conquer methodology. We can see that all the remaining algorithms (Insertion, Selection, Bubble) use significantly less memory compared to quick sort and merge sort.

Graph 8-Real Data#2, Data Size v/s Memory Usage



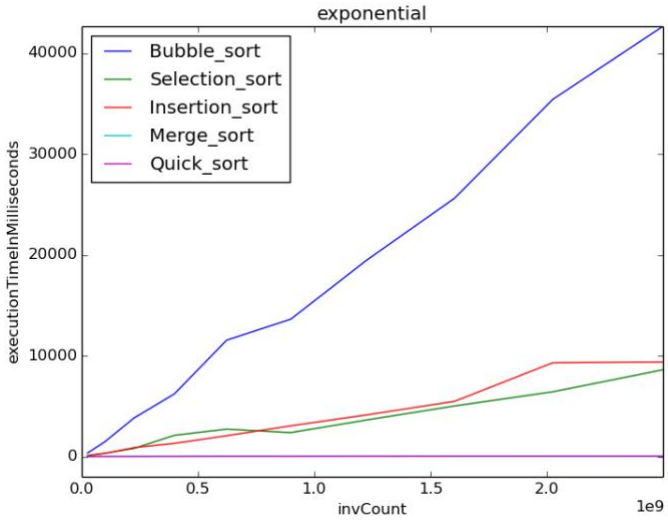
In this graph, we can see that Quick sort is using most memory compared to other sorting algorithms. It is followed by merge sort which also uses the same divide and conquer methodology. We can see that all the remaining algorithms (Insertion, Selection, Bubble) use significantly less memory compared to quick sort and merge sort.

Graph 9- Normal distribution, Inversion Count v/s Run time



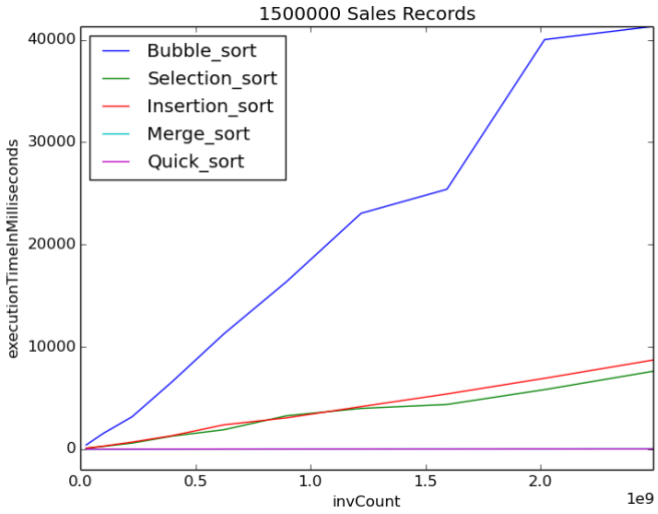
In this graph, as the inversion count increases i.e. the algorithm has to swap more entries which increases the execution time. Again, we can see that Bubble sort takes the maximum time when compared to all the other algorithms and relatively Merge sort, Quick sort take merely a fraction of the time.

Graph 10- Exponential distribution, Inversion Count v/s Run time



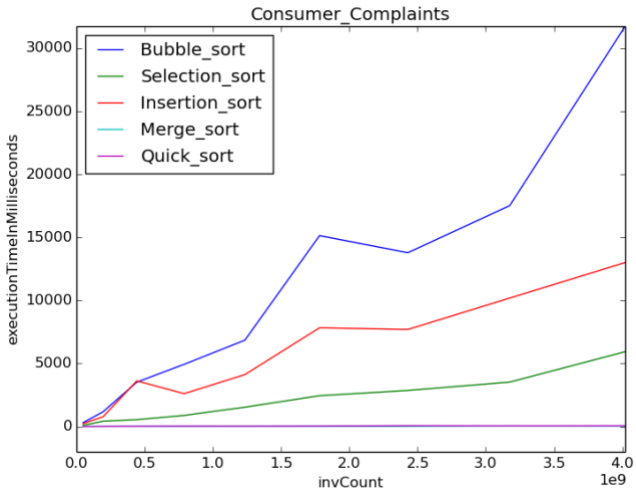
In this graph, as the inversion count increases i.e. the algorithm has to swap more entries which increases the execution time. Again, we can see that Bubble sort takes the maximum time when compared to all the other algorithms and relatively Merge sort, Quick sort take merely a fraction of the time.

Graph 11-Real Data#1, Inversion Count v/s Run time



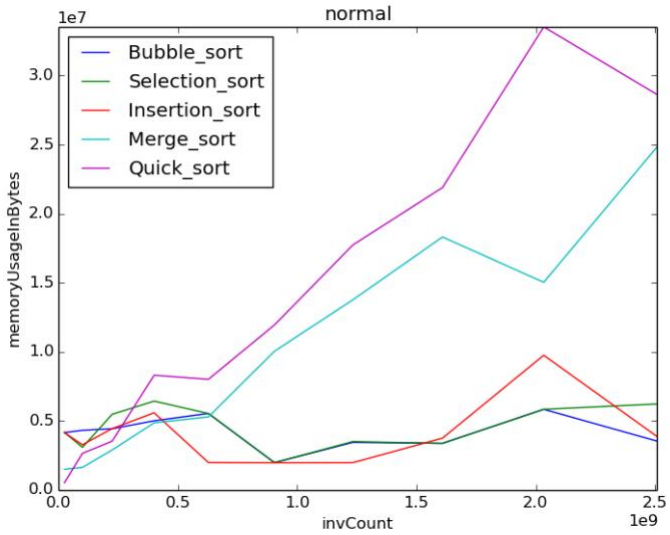
In this graph, as the inversion count increases i.e. the algorithm has to swap more entries which increases the execution time. Again, we can see that Bubble sort takes the maximum time when compared to all the other algorithms and relatively Merge sort, Quick sort take merely a fraction of the time.

Graph 12-Real Data#2, Inversion Count v/s Run time



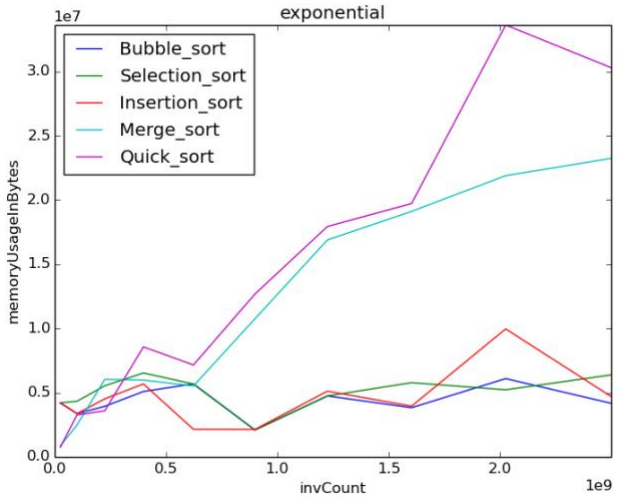
In this graph, as the inversion count increases i.e. the algorithm has to swap more entries which increases the execution time. Again, we can see that Bubble sort takes the maximum time when compared to all the other algorithms and relatively Merge sort, Quick sort take merely a fraction of the time.

Graph 13- Normal distribution, Inversion Count v/s Memory Usage



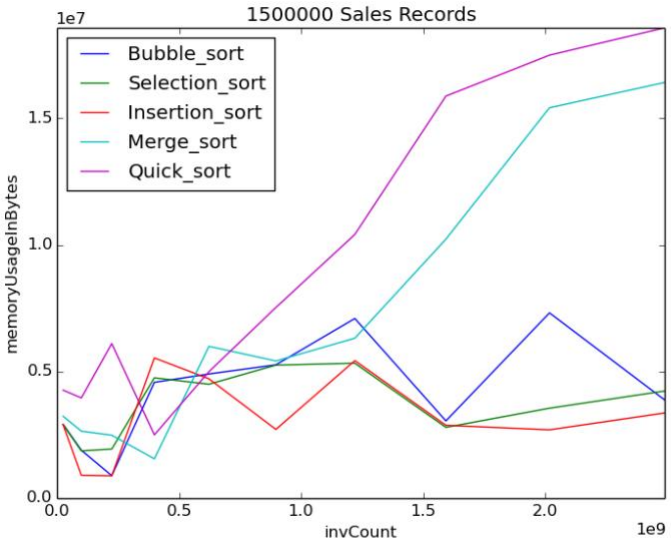
In this graph, though we see that the graph is not smooth enough, the general trend is that the memory consumption of merge sort and quick sort is relatively high as inversion count increases. All the other algorithms which sort in place use almost the same amount of memory which is significantly less than quick sort and merge sort.

Graph 14- Exponential distribution, Inversion Count v/s Memory Usage



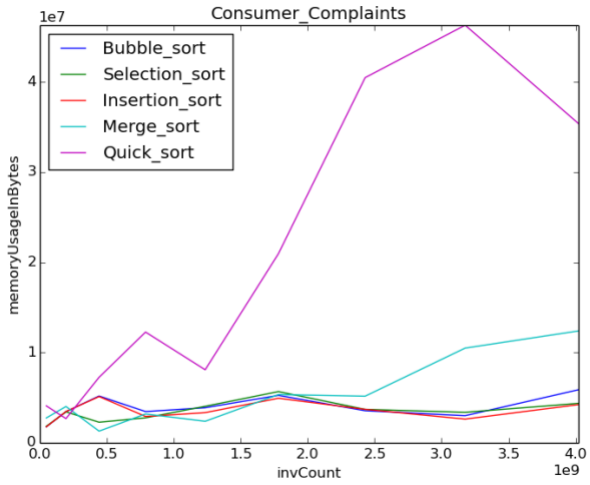
In this graph, though we see that the graph is not smooth enough, the general trend is that the memory consumption of merge sort and quick sort is relatively high as inversion count increases. All the other algorithms which sort in place use almost the same amount of memory which is significantly less than quick sort and merge sort.

Graph 15- Real Data#1, Inversion Count v/s Memory Usage



In this graph, though we see that the graph is not smooth enough, the general trend is that the memory consumption of merge sort and quick sort is relatively high as inversion count increases. All the other algorithms which sort in place use almost the same amount of memory which is significantly less than quick sort and merge sort.

Graph 16- Real Data#2, Inversion Count v/s Memory Usage



In this graph, though we see that the graph is not smooth enough, the general trend is that the memory consumption of merge sort and quick sort is relatively high as inversion count increases. All the other algorithms which sort in place use almost the same amount of memory which is significantly less than quick sort and merge sort.

Comparing the graphs:

Name of the Algorithm	Time Complexity / Space Complexity
Bubble Sort	O (n^2) / O (1)
Selection Sort	O (n^2) / O (1)
Insertion Sort	O (n^2) / O (1)
Merge Sort	O (n log n) / O (n)
Quick Sort	O (n log n)/ O (n)

From the Simulations, we have observed that the run time for Insertion sort, Selection Sort and Bubble Sort was much higher than Quick sort and Merge sort which represents the respective running times. Similarly, we also observe that the memory usage is higher for Quick sort and Merge sort when compared to other studied algorithms again representing the space complexity for the algorithms.