CSE 5311-004
Project Report

Yash Suraj Shah
Uta Id-1001858299

Project 1 **(Sorting Algorithms)**
Implement and compare the following sorting algorithm :

- Mergesort
- Heapsort
- Quicksort (Regular quick sort* and quick sort using 3 medians)
- Insertion sort
- Selection sort
- Bubble sort

* For regular quick sort you can decide between choosing first, last or a random element as pivot. But you need to include both regular and 3 medians as separate algorithms.

The table below shows the time complexity for the different sorting algorithms.

Algorithm	Best Case	Average Case	Worst Case
Mergesort	Ω(n log(n))	θ(n log(n))	O(n log(n))
Heapsort	Ω(n log(n))	θ(n log(n))	O(n log(n))
Quicksort	Ω(n log(n))	θ(n log(n))	O(n^2)
Insertion Sort	Ω(n)	θ (n^2)	O(n^2)
Selection Sort	Ω (n^2)	θ (n^2)	O(n^2)
Bubble Sort	Ω(n)	θ (n^2)	O(n^2)

---

1. **Merge Sort** –

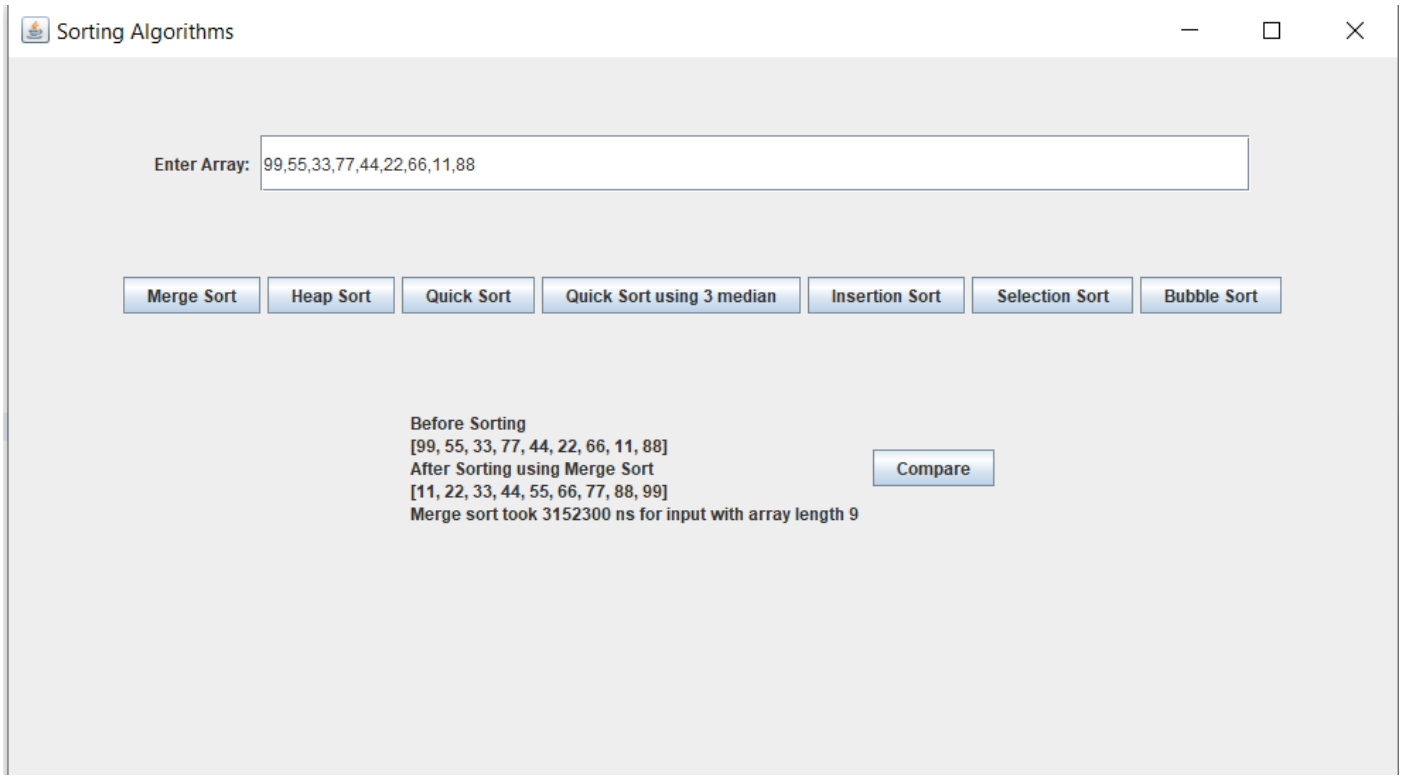
Code –

MainClass.java | GUI.java | MergeSort.java | HeapSort.java | QuickSort.java | InsertionSort.java | SelectionSort.java | BubbleSort.java | QuickSort3way.java

```
1  package sorting;
2
3  public class MergeSort {
4
5      void merge(int arr[], int left, int mid, int right)
6          {
7              int no1 = mid - left + 1;
8              int no2 = right - mid;
9
10             int leftarr[] = new int[no1];
11             int rightarr[] = new int[no2];
12
13             for (int i = 0; i < no1; ++i)
14                 leftarr[i] = arr[left + i];
15             for (int j = 0; j < no2; ++j)
16                 rightarr[j] = arr[mid + 1 + j];
17
18
19             int i = 0, j = 0;
20
21
22             int k = left;
23             while (i < no1 && j < no2) {
24                 if (leftarr[i] <= rightarr[j]) {
25                     arr[k] = leftarr[i];
26                     i++;
27                 }
28                 else {
29                     arr[k] = rightarr[j];
30                     j++;
31                 }
32                 k++;
33             }
34
35
36             while (i < no1) {
37                 arr[k] = leftarr[i];
38                 i++;
```

```
35
36              while (i < no1) {
37                  arr[k] = leftarr[i];
38                  i++;
39                  k++;
40              }
41
42
43              while (j < no2) {
44                  arr[k] = rightarr[j];
45                  j++;
46                  k++;
47              }
48          }
49
50          void sort(int arr[], int left, int right)
51          {
52              if (left < right) {
53
54                  int mid =left+ (right-left)/2;
55
56
57                  sort(arr, left, mid);
58                  sort(arr, mid + 1, right);
59
60
61                  merge(arr, left, mid, right);
62              }
63          }
64  }
65
```

Output –



Merge Sort is Divide and Conquer Algorithm. It divides the array into halves recursively and then merges the sorted halves.

There are 2 functions in this algorithm –

1. merge()- It is used to merge the sorted arrays.
2. Sort() – It checks the array inputs, divide the array recursively and then calls merge() function.

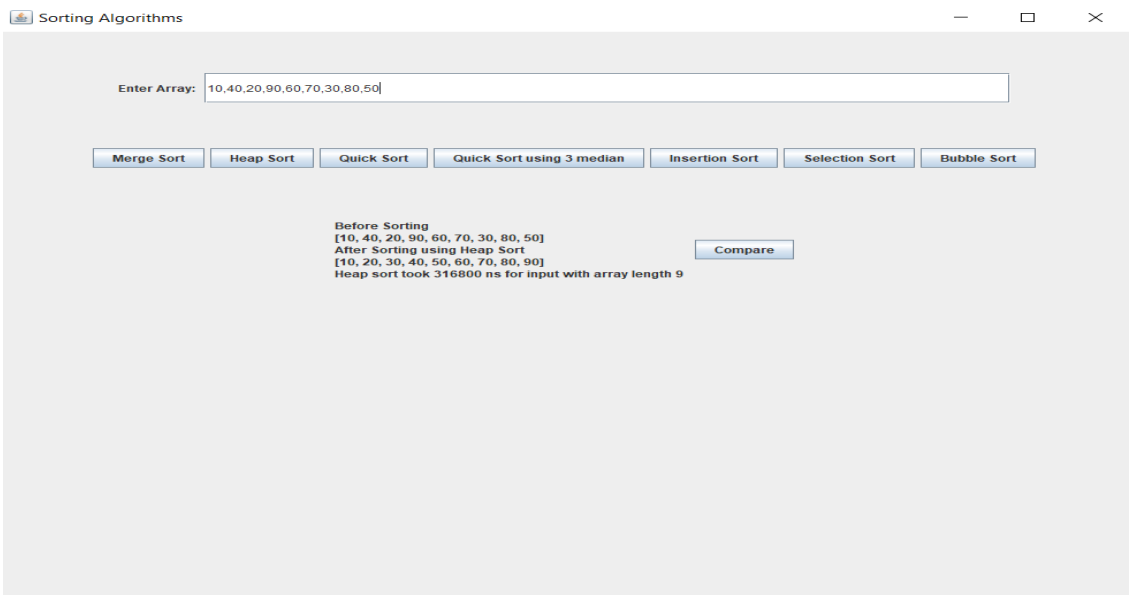
	Best Case	Average Case	Worst Case
Mergesort	Ω(n log(n))	θ(n log(n))	O(n log(n))

2. **Heapsort –**
Code –

MainClass.java | GUI.java | MergeSort.java | *HeapSort.java | QuickSort.java | InsertionSort.java | SelectionSort.java | BubbleSort.java | QuickSort3way.java

```java
package sorting;

public class HeapSort {
    public void sort(int arr[])
    {
        int length = arr.length;

        for (int i = length / 2 - 1; i >= 0; i--)
            heapify(arr, length, i);

        for (int i = length - 1; i > 0; i--) {
            int temp = arr[0];
            arr[0] = arr[i];
            arr[i] = temp;

            heapify(arr, i, 0);
        }
    }
    void heapify(int arr[], int n, int i)
    {
        int largest = i;
        int left = 2 * i + 1;
        int right = 2 * i + 2;

        if (left < n && arr[left] > arr[largest])
            largest = left;

        if (right < n && arr[right] > arr[largest])
            largest = right;

        if (largest != i) {
            int temp = arr[i];
            arr[i] = arr[largest];
            arr[largest] = temp;
            heapify(arr, n, largest);
        }
    }
}
```

Output –



Heap Sort builds a binary heap data structure where minimum element is placed at the root.
There are 2 functions in this algorithm –
1. heapify() – It is used to arrange the tree into a min heap after insertion of an element.
2. sort()- It is used to add new element and calls heapify() function.

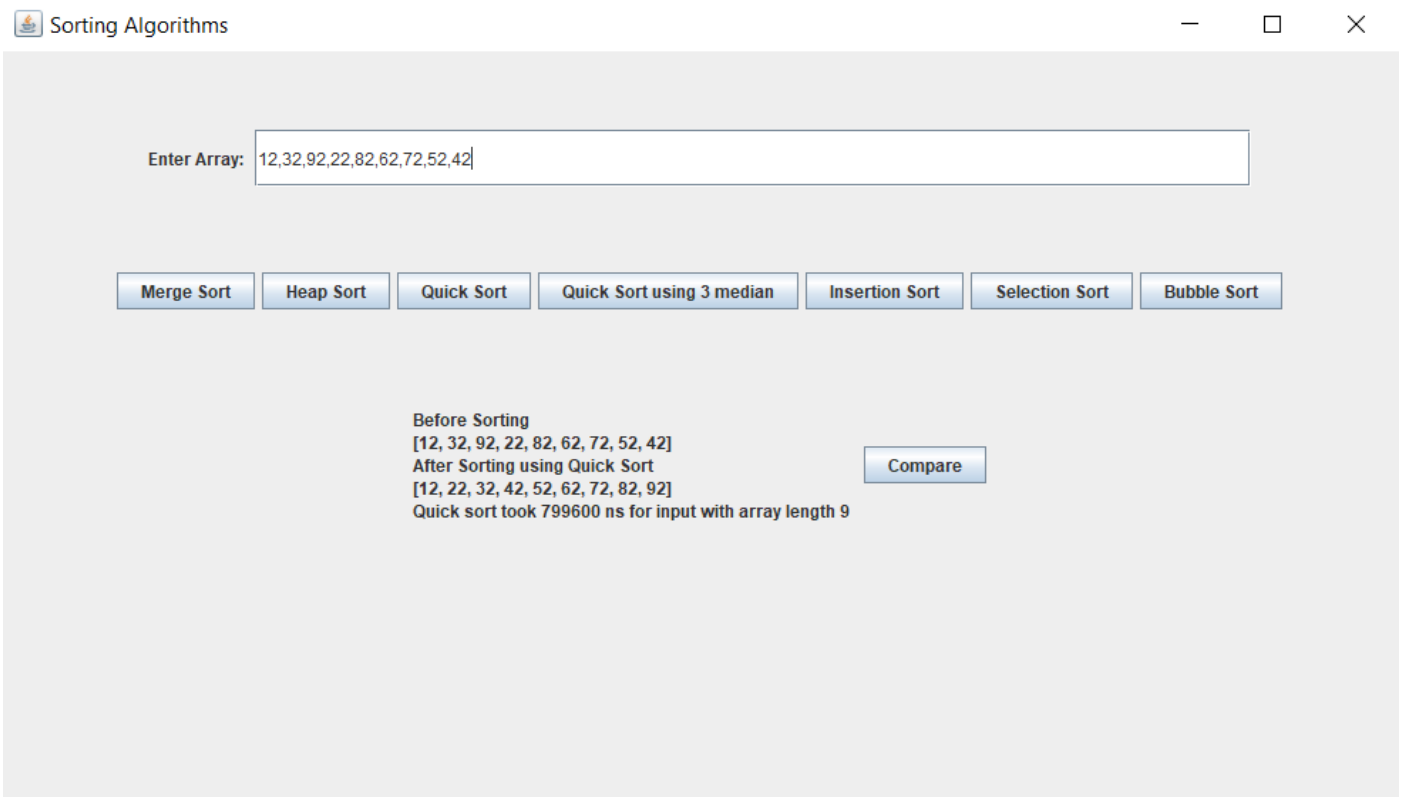
	Best Case	Average Case	Worst Case
Heapsort	Ω(n log(n))	θ(n log(n))	O(n log(n))

3. **Quicksort –**
Code –

MainClass.java | GUI.java | MergeSort.java | HeapSort.java | QuickSort.java | InsertionSort.java | SelectionSort.java | BubbleSort.java | QuickSort3way.java

```java
package sorting;

public class QuickSort {

    static void swap(int[] arr, int i, int j)
    {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }
    static int partition(int[] arr, int left, int right)
    {

        int pivot = arr[right];

        int i = (left - 1);

        for(int j = left; j <= right - 1; j++)
        {

            if (arr[j] < pivot)
            {
                i++;
                swap(arr, i, j);
            }
        }
        swap(arr, i + 1, right);
        return (i + 1);
    }
    void sort(int[] arr, int left, int right)
    {
        if (left < right)
        {  int pivot = partition(arr, left, right);
            sort(arr, left, pivot - 1);
            sort(arr, pivot + 1, right);
        }
    }
}
```

Output –



Quick Sort is Divide and Conquer Algorithm. It picks an element as pivot and divides the array according to pivot. There are 2 functions in this algorithm –
1. partition() – It takes last element as pivot and place smaller elements to left and greater elements to right.
2. sort() – Its calls partition() and then recursively calls itself until the array is sorted.

	Best Case	Average Case	Worst Case
Quicksort	Ω(n log(n))	θ(n log(n))	O(n^2)

4. **Quicksort using 3 median –**

Code –

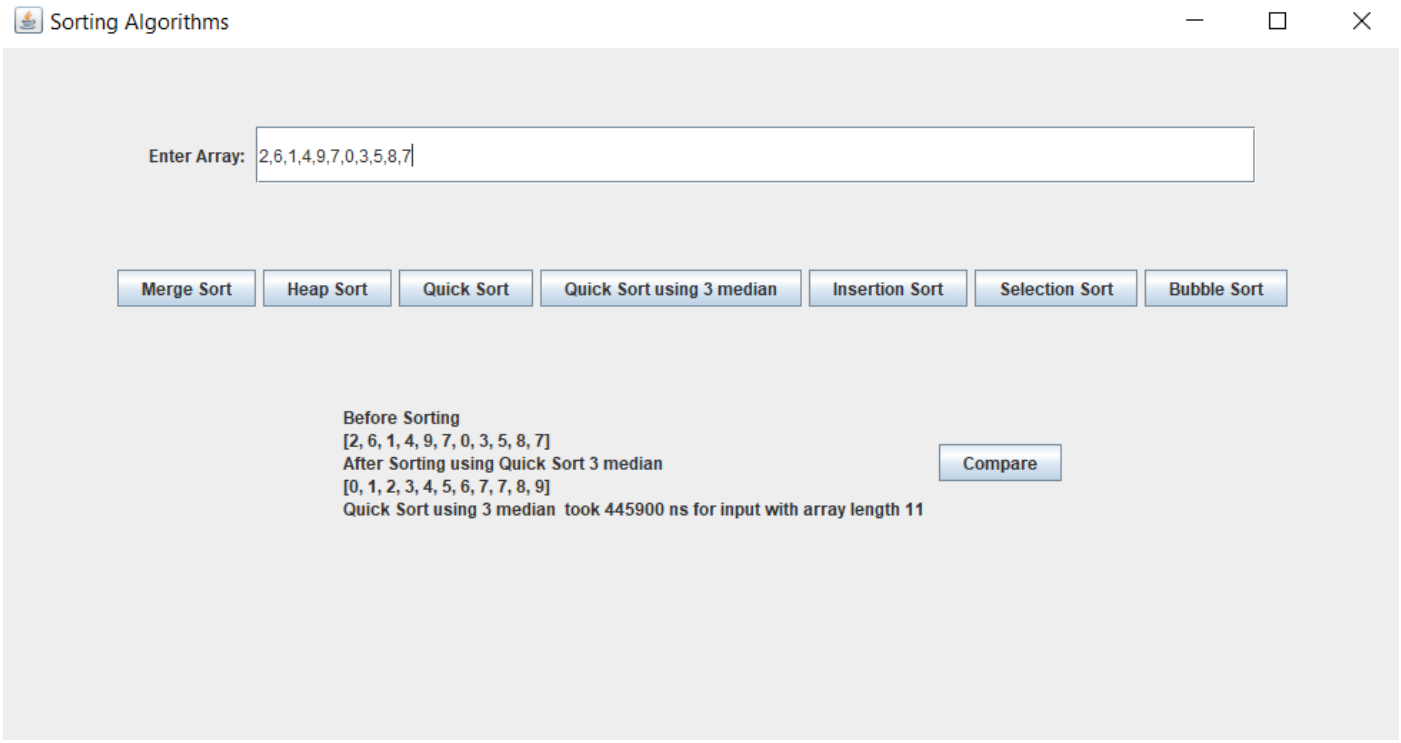
MainClass.java | GUI.java | MergeSort.java | HeapSort.java | QuickSort.java | QuickSort3way.java | InsertionSort.java | SelectionSort.java | BubbleSort.java

```
1  package sorting;
2
3  public class QuickSort3way {
4
5      public  void sort(int[] arr, int left, int right) {
6          int size = right - left + 1;
7          if (size <= 3)
8              manualSort(arr, left, right);
9          else {
10             double median = medianOf3(arr, left, right);
11             int partition = partitionIt(arr, left, right, median);
12             sort(arr, left, partition - 1);
13             sort(arr, partition + 1, right);
14         }
15     }
16
17     public  int medianOf3(int[] arr, int left, int right) {
18         int center = (left + right) / 2;
19
20         if (arr[left] > arr[center])
21             swap(arr, left, center);
22
23         if (arr[left] > arr[right])
24             swap(arr, left, right);
25
26         if (arr[center] > arr[right])
27             swap(arr, center, right);
28
29         swap(arr, center, right - 1);
30         return arr[right - 1];
31     }
32     public  void swap(int[] arr, int x, int y) {
33         int temp = arr[x];
34         arr[x] = arr[y];
35         arr[y] = temp;
36     }
37
38     public  int partitionIt(int[] intArray, int left, int right, double pivot) {
```

MainClass.java | GUI.java | MergeSort.java | HeapSort.java | QuickSort.java | QuickSort3way.java | InsertionSort.java | SelectionSort.java | BubbleSort.java

```
36     }
37
38     public  int partitionIt(int[] intArray, int left, int right, double pivot) {
39         int leftPtr = left;
40         int rightPtr = right - 1;
41         while (true) {
42             while (intArray[++leftPtr] < pivot)
43                 ;
44             while (intArray[--rightPtr] > pivot)
45                 ;
46             if (leftPtr >= rightPtr)
47                 break;
48             else
49                 swap(intArray, leftPtr, rightPtr);
50         }
51         swap(intArray, leftPtr, right - 1);
52         return leftPtr;
53     }
54
55     public  void manualSort(int[] intArray, int left, int right) {
56         int size = right - left + 1;
57         if (size <= 1)
58             return;
59         if (size == 2) {
60             if (intArray[left] > intArray[right])
61                 swap(intArray, left, right);
62             return;
63         } else {
64             if (intArray[left] > intArray[right - 1])
65                 swap(intArray, left, right - 1);
66             if (intArray[left] > intArray[right])
67                 swap(intArray, left, right);
68             if (intArray[right - 1] > intArray[right])
69                 swap(intArray, right - 1, right);
70         }
71     }
72 }
73
```

Output –



Quick Sort using 3 median is Divide and Conquer Algorithm. It picks 3 element as pivot and divides the array according to all occurrences of their median.
There are 4 functions in this algorithm –
1. medianOf3()- It returns the median of the array.
2. partionIt() – This will arrange the smaller elements to left and larger to right of pivot.
3. manualSort() – This will sort the partioned array.
4. sort() – This will call the functions if the size of array is greater than 3 and recursively call itself.

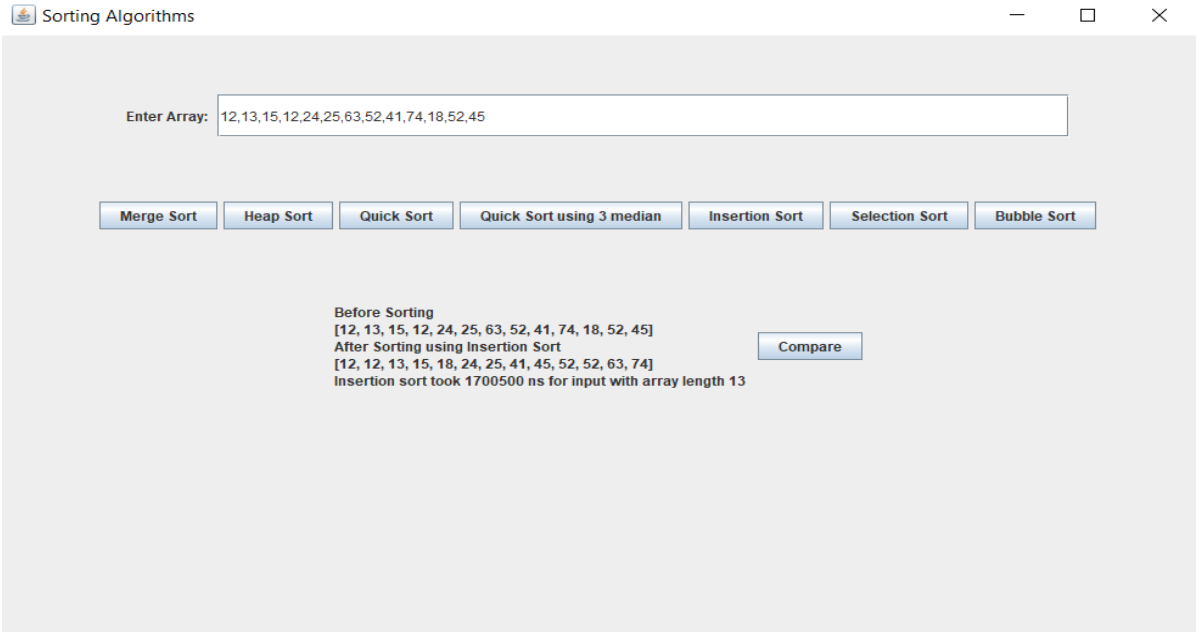
5. **Insertion sort -**

Code –

MainClass.java | GUI.java | MergeSort.java | HeapSort.java | QuickSort.java | QuickSort3way.java | InsertionSort.java | SelectionSort.java | BubbleSort.java

```java
package sorting;

public class InsertionSort {
    void sort(int arr[])
    {
        int length = arr.length;
        for (int i = 1; i < length; ++i) {
            int temp = arr[i];
            int j = i - 1;

            while (j >= 0 && arr[j] > temp) {
                arr[j + 1] = arr[j];
                j = j - 1;
            }
            arr[j + 1] = temp;
        }
    }
}
```

Output –



In insertion sort we virtually split the array in ordered and unordered arrays and place the elements from the unordered array to the correct position in ordered array.

There is just one function sort() that compare the elements to the previous elements and then insert into the correct position if smaller.

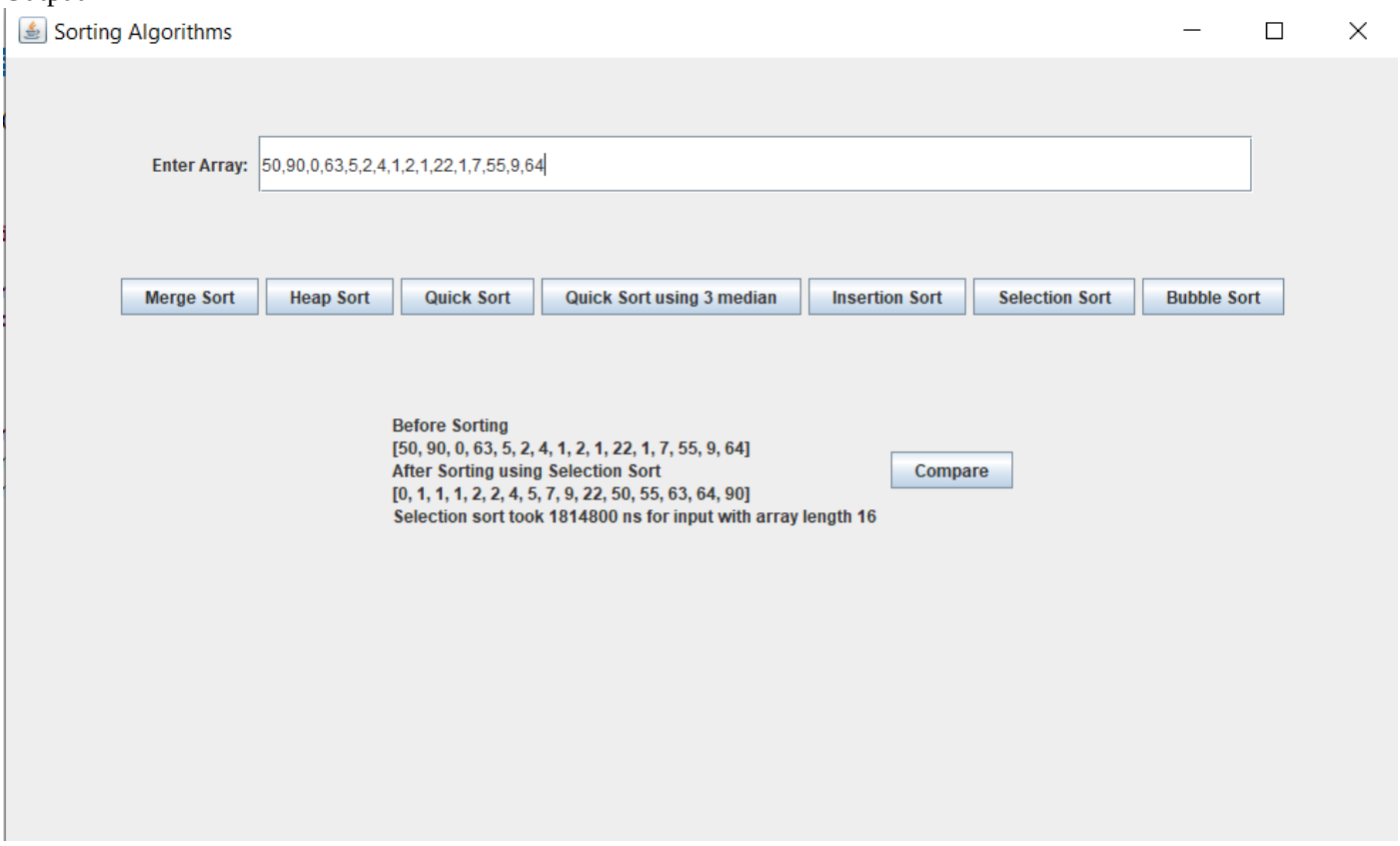
	Best Case	Average Case	Worst Case
Insertion Sort	Ω(n)	θ (n^2)	O(n^2)

6. **Selection sort -**
Code –

MainClass.java | GUI.java | MergeSort.java | HeapSort.java | QuickSort.java | QuickSort3way.java | InsertionSort.java | SelectionSort.java | BubbleSort.java

```java
package sorting;

public class SelectionSort {

    void sort(int arr[])
    {
        int n = arr.length;

        for (int i = 0; i < n-1; i++)
        {
            int min = i;
            for (int j = i+1; j < n; j++)
                if (arr[j] < arr[min])
                    min = j;

            int temp = arr[min];
            arr[min] = arr[i];
            arr[i] = temp;
        }
    }

}
```

Output –



Selection sort repeatedly finds the minimum element from unsorted part and putting it in the beginning.
This algorithm has only 1 function sort() that moves the boundary of unsorted part and swaps the min element of the unsorted part to the first position.

	Best Case	Average Case	Worst Case
Selection Sort	$\Omega(n^2)$	$\theta(n^2)$	$O(n^2)$

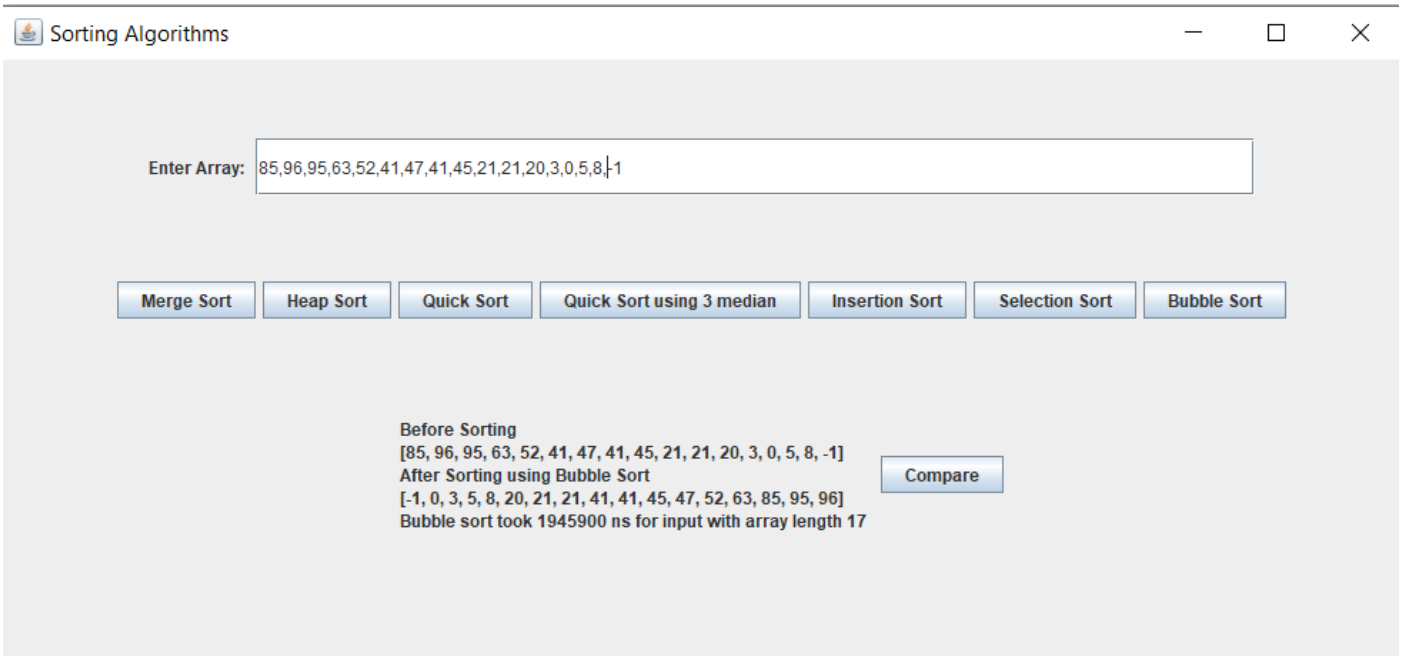
7. **Bubble Sort –**

Code –

MainClass.java | GUI.java | MergeSort.java | HeapSort.java | QuickSort.java | QuickSort3way.java | InsertionSort.java | SelectionSort.java | BubbleSort.java

```java
package sorting;

public class BubbleSort {
    void sort(int arr[])
    {
        int n = arr.length;
        for (int i = 0; i < n-1; i++)
            for (int j = 0; j < n-i-1; j++)
                if (arr[j] > arr[j+1])
                {
                    int temp = arr[j];
                    arr[j] = arr[j+1];
                    arr[j+1] = temp;
                }
    }
}

```

Output –



Bubble Sort repeatedly swaps the neighbouring elements if they are in wrong order.
This algorithm has only one function that compares the elements with neighbouring element and swaps if the second element is smaller.

	Best Case	Average Case	Worst Case
Bubble Sort	$\Omega(n)$	$\theta\ (n^2)$	$O(n^2)$

8. **Comparison** –
Code –

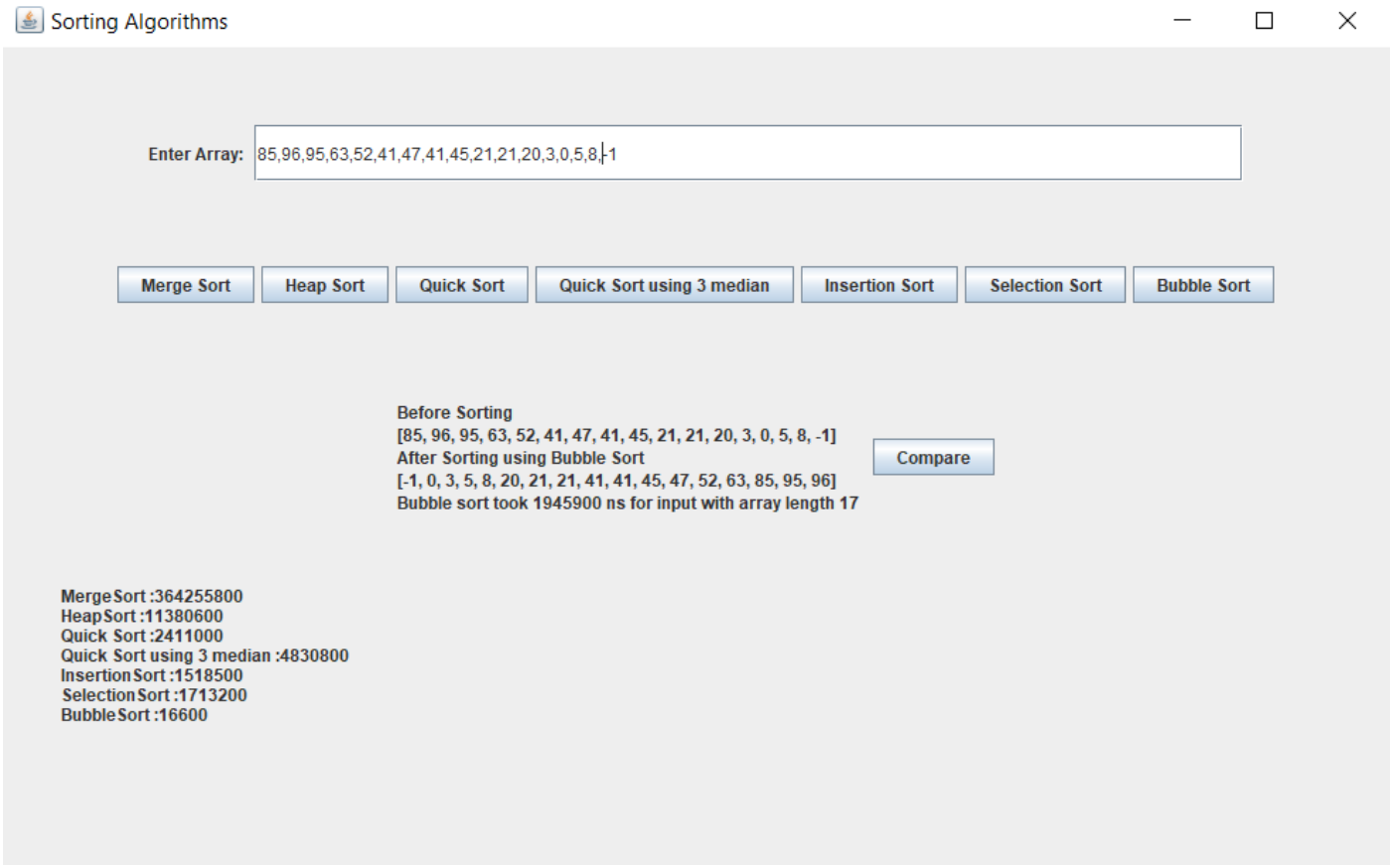
MainClass.java | GUI.java | MergeSort.java | HeapSort.java | QuickSort.java | QuickSort3way.java | InsertionSort.java | SelectionSort.java | BubbleSort.java

```
142
143                 case 8 :
144                     // Merge Sort
145                     startTime = System.nanoTime();
146                     MergeSort ms1 = new MergeSort();
147                     ms1.sort(arr,0,arr.length-1);
148                     endTime = System.nanoTime();
149                     time = endTime - startTime;
150                     System.out.println("MergeSort :"+ time);
151
152                     // Heap Sort
153                     startTime = System.nanoTime();
154                     HeapSort hs1 = new HeapSort();
155                     hs1.sort(arr);
156                     endTime = System.nanoTime();
157                     time = endTime - startTime;
158                     System.out.println("HeapSort :"+ time);
159
160                     // Quick Sort
161                     startTime = System.nanoTime();
162                     QuickSort qs1 = new QuickSort();
163                     qs1.sort(arr,0,arr.length-1);
164                     endTime = System.nanoTime();
165                     time = endTime - startTime;
166                     System.out.println("Quick Sort :"+ time);
167
168                     // Quick Sort using 3 median
169                     startTime = System.nanoTime();
170                     QuickSort3way qs3_1 = new QuickSort3way();
171                     qs3_1.sort(arr,0,arr.length-1);
172                     endTime = System.nanoTime();
173                     time = endTime - startTime;
174                     System.out.println("Quick Sort using 3 median :"+ time);
175
```

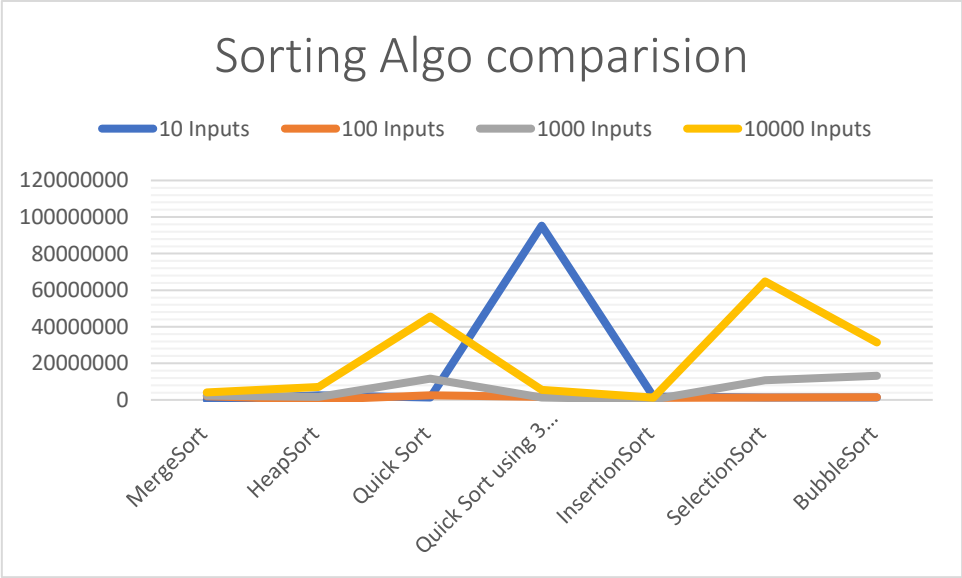
MainClass.java | GUI.java | MergeSort.java | HeapSort.java | QuickSort.java | QuickSort3way.java | InsertionSort.java | SelectionSort.java | BubbleSort.java

```
175
176                     // Insertion Sort
177                     startTime = System.nanoTime();
178                     InsertionSort is1 = new InsertionSort();
179                     is1.sort(arr);
180                     endTime = System.nanoTime();
181                     time = endTime - startTime;
182                     System.out.println("InsertionSort :"+ time);
183
184                     // Selection Sort
185                     startTime = System.nanoTime();
186                     SelectionSort ss1 = new SelectionSort();
187                     ss1.sort(arr);
188                     endTime = System.nanoTime();
189                     time = endTime - startTime;
190                     System.out.println("SelectionSort :"+ time);
191
192                     // Bubble Sort
193                     startTime = System.nanoTime();
194                     BubbleSort bs1 = new BubbleSort();
195                     bs1.sort(arr);
196                     endTime = System.nanoTime();
197                     time = endTime - startTime;
198                     System.out.println("BubbleSort :"+ time);
199                     System.out.println();
200                     break;
201
```

Output –



Here running time of the sorts are displayed in order to compare the complexity.



9. **Main Class** –

- This Class is the main class which has the driver code for all the algorithms mentioned above.
- It also has the code to take the input from the user, i.e., the array size, array elements, the algorithms he wish to use and the comparison option.
- It has code to create object of the Sorting classes and call their function when specific input is given and to display the output.
- Moreover it shows the time taken by the algorithm to execute.

10. **GUI Class** –

- This class contains all the functionality of the main class along with the code to GUI.
- Unlike Main Class this class provides with an User Interface for execution.