Design and Analysis of Algorithms – Assignment 2

Team Members:

Pragathi Thammaneni (16230695)

Sridevi Mallipudi(16251191)

Assignment -2

Insertion Sort:

Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort.

The array is searched sequentially, and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst-case complexity are of $O(n^2)$, where n is the number of items.

Worst Case: $O(n^2)$

Best Case: $O(n)$

Average Case: $O(n^2)$

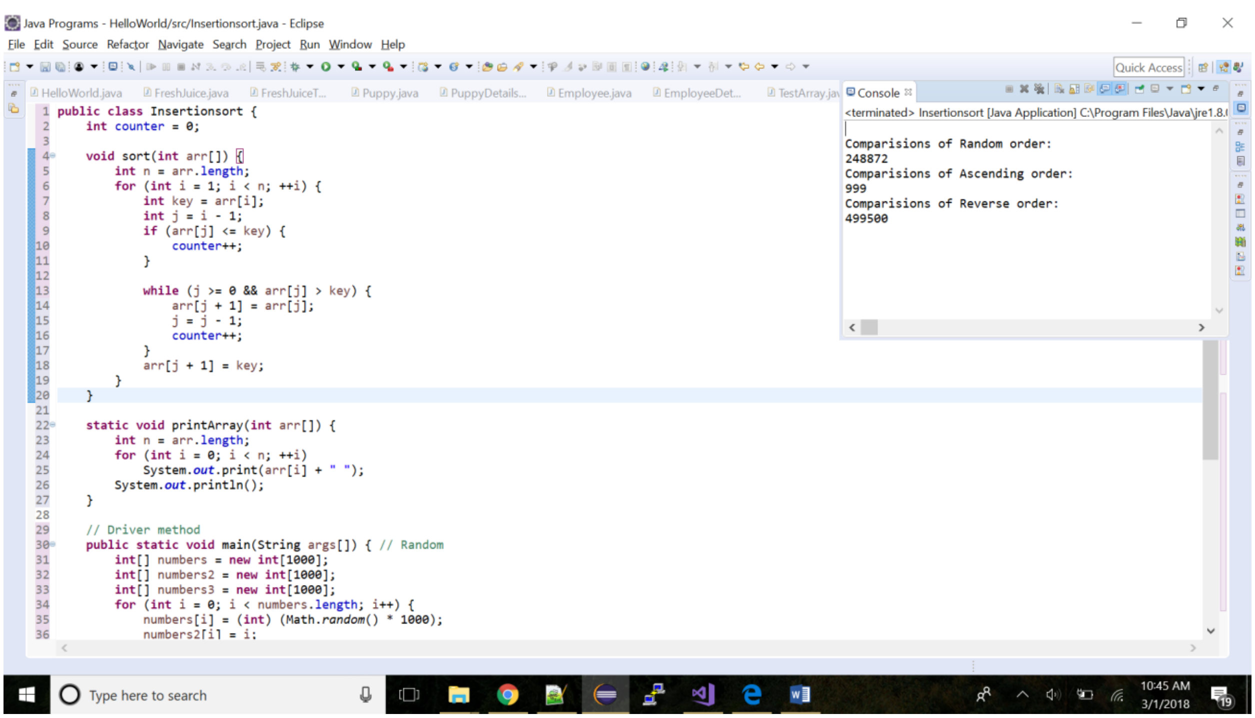
Programmatically we considered three type arrays of size 1000, an array that is sorted in ascending order **-Sorted Array**, a **Reversed array** and a **Random array**.

Source code for the Insertion Sort:

```
public class Insertionsort {
	int counter = 0;

	void sort(int arr[]) {
		int n = arr.length;
		for (int i = 1; i < n; ++i) {
			int key = arr[i];
			int j = i - 1;
			if (arr[j] <= key) {
				counter++;
			}

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

	static void printArray(int arr[]) {
		int n = arr.length;
		for (int i = 0; i < n; ++i)
			System.out.print(arr[i] + " ");
		System.out.println();
	}
```

```java
	// Driver method
	public static void main(String args[]) { // Random
		int[] numbers = new int[1000];
		int[] numbers2 = new int[1000];
		int[] numbers3 = new int[1000];
		for (int i = 0; i < numbers.length; i++) {
			numbers[i] = (int) (Math.random() * 1000);
			numbers2[i] = i;
			numbers3[i] = 1000 - i;
		}
		Insertionsort ob = new Insertionsort();
		Insertionsort ob2 = new Insertionsort();
		Insertionsort ob3 = new Insertionsort();
		ob.sort(numbers);
		ob2.sort(numbers2);
		ob3.sort(numbers3);
		printArray(numbers);
		System.out.println("Comparisions of Random order: ");
		System.out.println(ob.counter);
		System.out.println("Comparisions of Ascending order: ");
		System.out.println(ob2.counter);
		System.out.println("Comparisions of Reverse order: ");
		System.out.println(ob3.counter);
	}
}
```

Output Screenshot:



Merge Sort:

Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being O(n log n).

Merge sort first divides the array into equal halves and then combines them in a sorted manner.

Merge sort keeps on dividing the list into equal halves until it can no more be divided. If it is only one element in the list, it is sorted. Then, merge sort combines the smaller sorted lists keeping the new list sorted too.

```
Step 1 – if it is only one element in the list it is already sorted, return.
Step 2 – divide the list recursively into two halves until it can no more be divided.
Step 3 – merge the smaller lists into new list in sorted order
```

Worst Case: O(n log n)

Best Case: O(n log n)

Average Case: O(n log n)

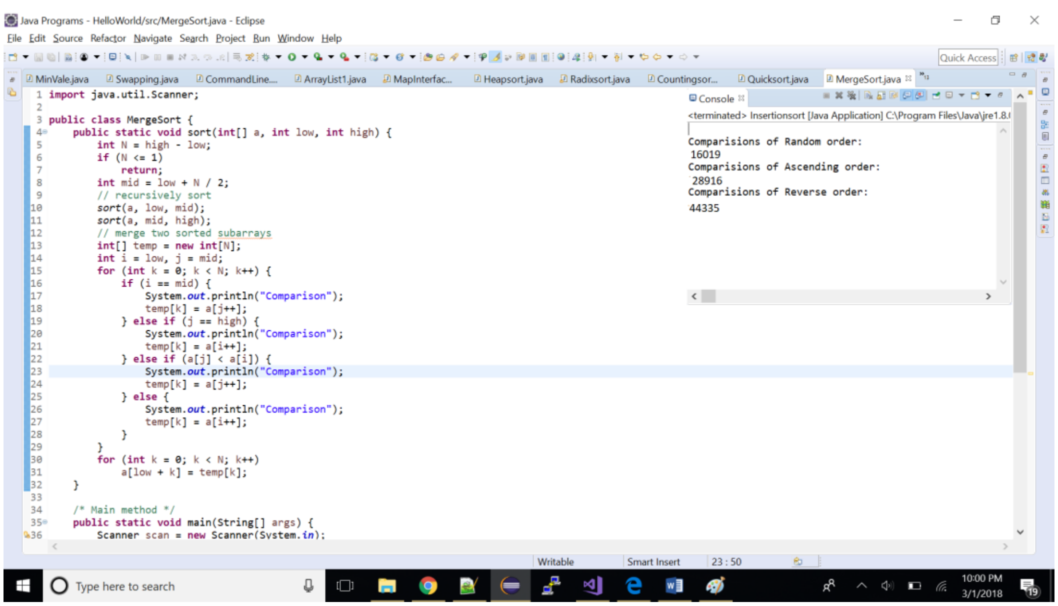
Programmatically we considered three type arrays of size 1000, an array that is sorted in ascending order **-Sorted Array**, a **Reversed array** and a **Random array**.

Source code for the Merge Sort:

```java
import java.util.Scanner;

public class MergeSort {
	public static void sort(int[] a, int low, int high) {
		int N = high - low;
		if (N <= 1)
			return;
		int mid = low + N / 2;
		// recursively sort
		sort(a, low, mid);
		sort(a, mid, high);
		// merge two sorted subarrays
		int[] temp = new int[N];
		int i = low, j = mid;
		for (int k = 0; k < N; k++) {
			if (i == mid) {
				System.out.println("Comparison");
				temp[k] = a[j++];
			} else if (j == high) {
				System.out.println("Comparison");
				temp[k] = a[i++];
			} else if (a[j] < a[i]) {
				System.out.println("Comparison");
				temp[k] = a[j++];
```

```java
                } else {
                    System.out.println("Comparison");
                    temp[k] = a[i++];
                }
            }
            for (int k = 0; k < N; k++)
                a[low + k] = temp[k];
        }

        /* Main method */
        public static void main(String[] args) {
            Scanner scan = new Scanner(System.in);
            System.out.println("Merge Sort Test\n");
            int n, i;
            /* Accept number of elements */
            System.out.println("Enter number of integer elements");
            n = scan.nextInt();
            /* Create array of n elements */
            int arr[] = new int[n];
            /* Accept elements */
            System.out.println("\nEnter " + n + " integer elements");
            for (i = 0; i < n; i++)
                arr[i] = scan.nextInt();
            /* Call method sort */
            sort(arr, 0, n);
            /* Print sorted Array */
            System.out.println("\nElements after sorting ");
            for (i = 0; i < n; i++)
                System.out.print(arr[i] + " ");
            System.out.println();
        }
}
```

Output Screenshot:



Heap Sort:

Heapsort is a comparison-based sorting algorithm to create a sorted array (or list), and is part of the selection sort family

Heap Sort is one of the best sorting methods being in-place and with no quadratic worst-case scenarios. Heap sort algorithm is divided into two basic parts: Creating a Heap of the unsorted list. Then a sorted array is created by repeatedly removing the largest/smallest element from the heap and inserting it into the array. The heap is reconstructed after each removal.

Worst Case: O(n log n)

Best Case: O(n log n)

Average Case: O(n log n)

Programmatically we considered three type arrays of size 1000, an array that is sorted in ascending order **-Sorted Array**, a **Reversed array** and a **Random array**.

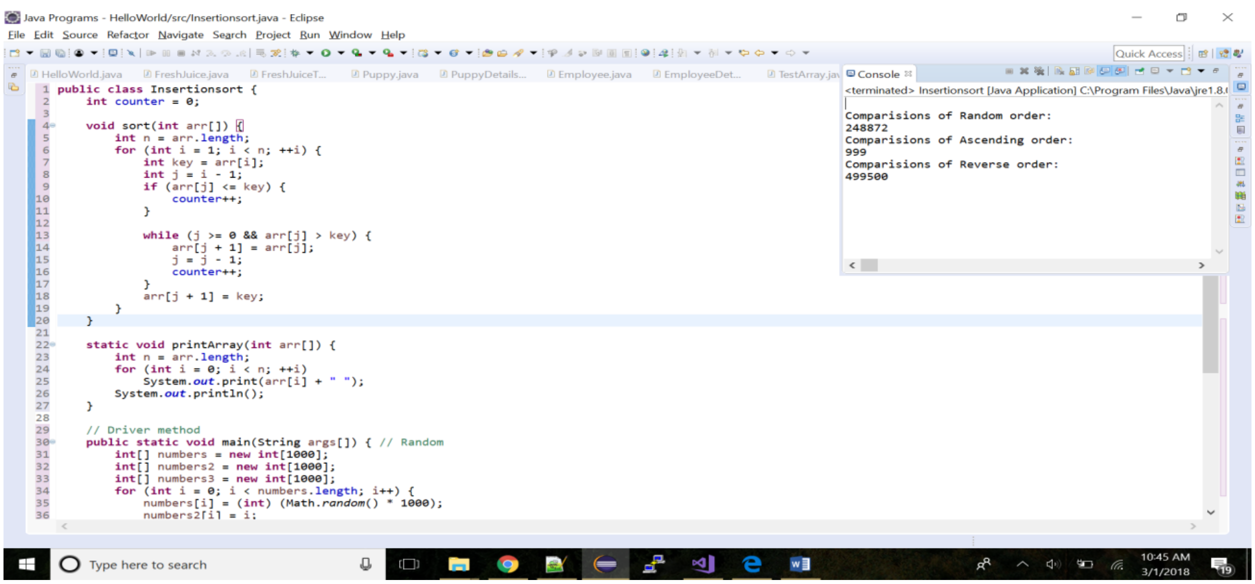
Source code for the Heap Sort:

```
public class Heapsort {
static int count = 0;
public void sort (int arr[])
{
 int n = arr.length;
 // Build heap (rearrange array)
 for (int i = n / 2 - 1; i >= 0; i--)
 count = count + heapify(arr, n, i);
 // One by one extract an element from heap
 for (int i=n-1; i>=0; i--)
 {
 // Move current root to end
 int temp = arr[0];
 arr[0] = arr[i];
 arr[i] = temp;
 // call max heapify on the reduced heap
 count = count + heapify(arr, i, 0);
 }
 }
 // To heapify a subtree rooted with node i which is
 // an index in arr[]. n is size of heap
int heapify(int arr[], int n, int i)
{
int count = 0;
int largest = i; // Initialize largest as root
int l = 2*i + 1; // left = 2*i + 1
int r = 2*i + 2; // right = 2*i + 2
// If left child is larger than root
if (l < n && arr[l] > arr[largest]){
```

```
 largest = l;
 count++;
}
 // If right child is larger than largest so far
if (r < n && arr[r] > arr[largest]){
 largest = r;
 count++;
}
 // If largest is not root
 if (largest != i)
{
 int swap = arr[i];
 arr[i] = arr[largest];
 arr[largest] = swap;
 // Recursively heapify the affected sub-tree
 heapify(arr, n, largest);
}
 return count;
}
 /* A utility function to print array of size n */
static void printArray(int arr[])
{
 int n = arr.length;
 for (int i=0; i<n; ++i)
 System.out.print(arr[i]+" ");
 System.out.println();
}
 // Driver program
public static void main(String args[])
{
 int[] arr1 = new int[1000];
 int[] arr2 = new int[1000];
 int[] arr3 = new int[1000];
int i = 0;
while(i<1000){
 arr1[i] = i+1;
 i++;
}
i = 1000;
int j = 0;
while(i>0){
arr2[j] = i;
i--;
j++;
}
i=0;
while(i<1000){
arr3[i] = 0 + (int)(Math.random() * 1111);
i++;
}
Heapsort ob = new Heapsort();
ob.sort(arr1);
System.out.println("Reverse array is");
printArray(arr1);
System.out.println("Comparisions of Reverse order : "+count);
```

```java
count = 0;
ob = new Heapsort();
ob.sort(arr2);
System.out.println("Sorted array is");
printArray(arr2);
System.out.println("Comparisions of ascending order: "+count);
count = 0;
ob = new Heapsort();
ob.sort(arr3);
System.out.println("Random array is");
printArray(arr3);
System.out.println("Comparisions of Random order: "+count);
}
}
```

Output Screenshot:



Quick Sort:

Quicksort (sometimes called partition-exchange sort) is an efficient sorting algorithm, serving as a systematic method for placing the elements of an array in order. Quicksort can operate in-place on an array, requiring small additional amounts of memory to perform the sorting. It is very similar to selection sort, except that it does not always choose worst-case partition.

Worst Case: O(n2)

Best Case: O(nlogn)

Average Case: O(nlogn)

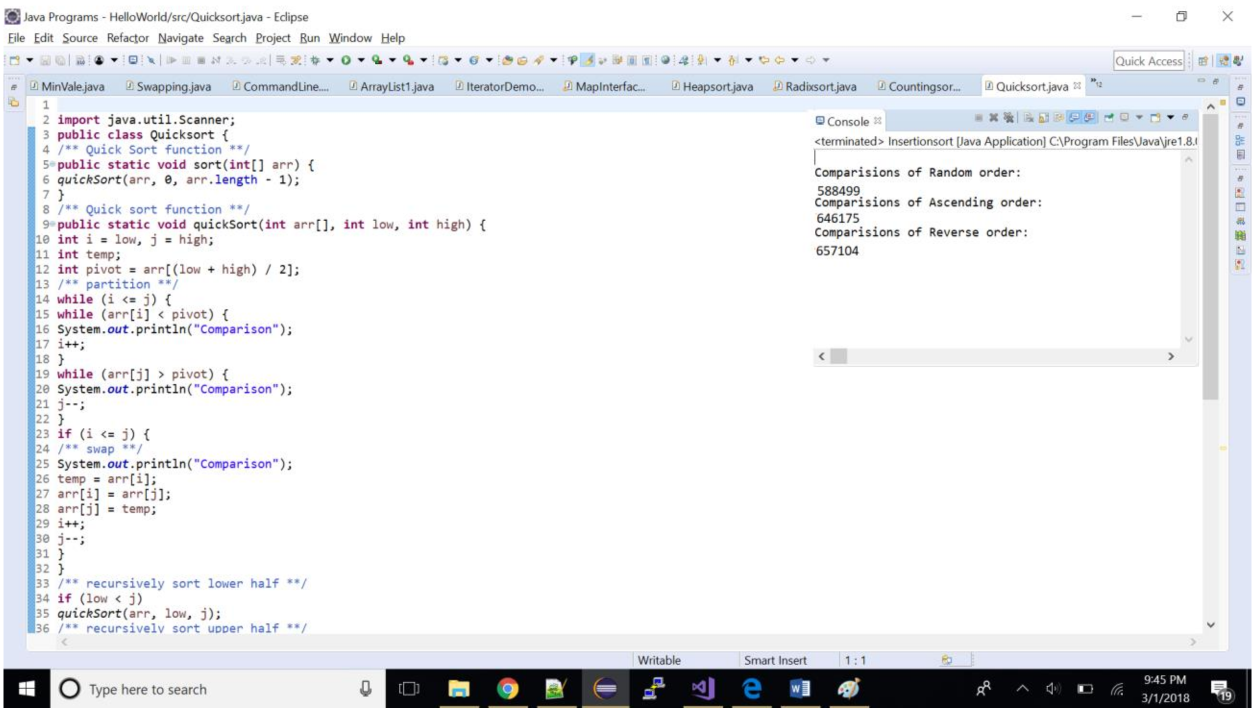
Programmatically we considered three type arrays of size 1000, an array that is sorted in ascending order **-Sorted Array**, a **Reversed array** and a **Random array**.

Source code for the Quick Sort:

```java
import java.util.Scanner;
public class Quicksort {
/** Quick Sort function **/
public static void sort(int[] arr) {
quickSort(arr, 0, arr.length - 1);
}
/** Quick sort function **/
public static void quickSort(int arr[], int low, int high) {
int i = low, j = high;
int temp;
int pivot = arr[(low + high) / 2];
/** partition **/
while (i <= j) {
while (arr[i] < pivot) {
System.out.println("Comparison");
i++;
}
while (arr[j] > pivot) {
System.out.println("Comparison");
j--;
}
if (i <= j) {
/** swap **/
System.out.println("Comparison");
temp = arr[i];
arr[i] = arr[j];
arr[j] = temp;
i++;
j--;
}
}
/** recursively sort lower half **/
if (low < j)
quickSort(arr, low, j);
/** recursively sort upper half **/
if (i < high)
quickSort(arr, i, high);
}
/** Main method **/
public static void main(String[] args) {
Scanner scan = new Scanner(System.in);
System.out.println("Quick Sort Test\n");
int n, i;
/** Accept number of elements **/
System.out.println("Enter number of integer elements");
n = scan.nextInt();
/** Create array of n elements **/
int arr[] = new int[n];
/** Accept elements **/
System.out.println("\nEnter " + n + " integer elements");
for (i = 0; i < n; i++)
arr[i] = scan.nextInt();
/** Call method sort **/
sort(arr);
```

```
/** Print sorted Array **/
System.out.println("\nElements after sorting ");
for (i = 0; i < n; i++)
System.out.print(arr[i] + " ");
System.out.println();
}
}
```

Output Screenshot:



Counting Sort:

Counting sort is an efficient algorithm for sorting an array of elements that each have a nonnegative integer key, for example, an array, sometimes called a list, of positive integers could have keys that are just the value of the integer as the key, or a list of words could have keys assigned to them by some scheme mapping the alphabet to integers (to sort in alphabetical order, for instance). Unlike other sorting algorithms, such as mergesort, counting sort is an **integer sorting** algorithm, not a comparison-based algorithm. While any comparison based sorting algorithm requires comparisons, counting sort has a running time of when the length of the input list is not much smaller than the largest key value, , in the list. Counting sort can be used as a subroutine for other, more powerful, sorting algorithms such as radix sort.

Worst Case: O(n+k)

Best Case: O(n+k)

Average Case: O(n+k)

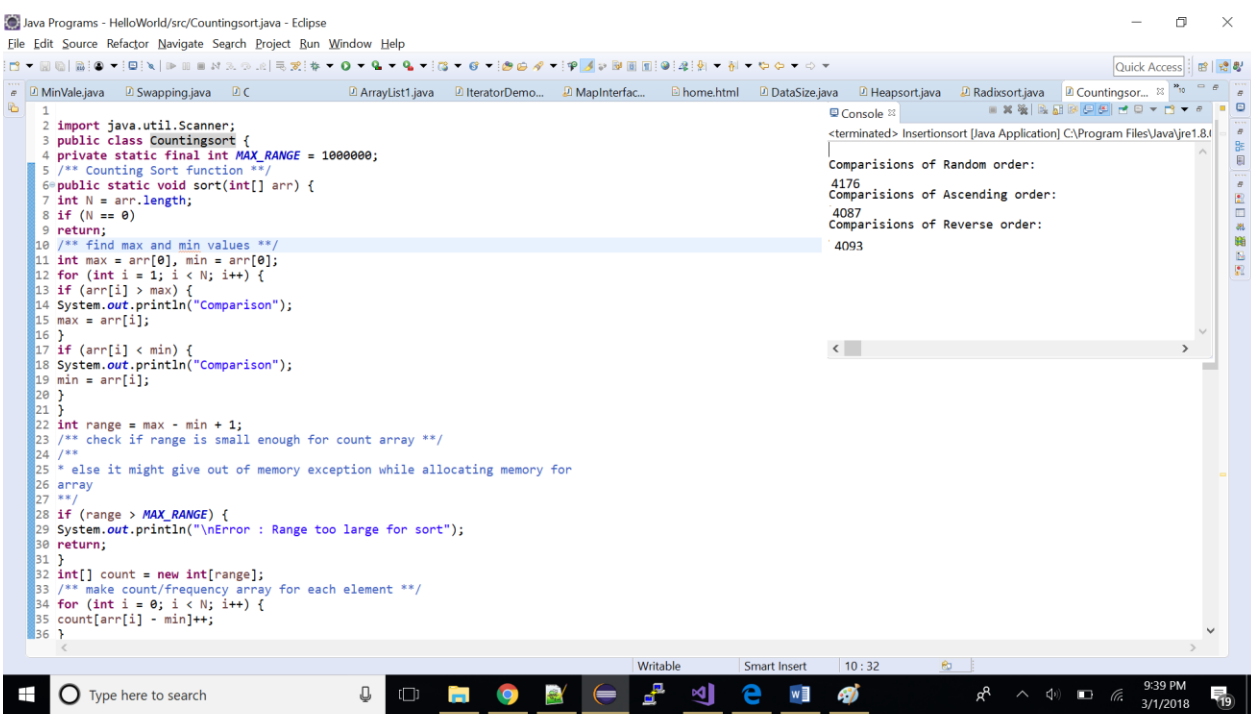
Programmatically we considered three type arrays of size 1000, an array that is sorted in ascending order -**Sorted Array**, a **Reversed array** and a **Random array**.

Source code for the Counting Sort:

```
import java.util.Scanner;
public class Countingsort {
private static final int MAX_RANGE = 1000000;
/** Counting Sort function **/
public static void sort(int[] arr) {
int N = arr.length;
if (N == 0)
return;
/** find max and min values **/
int max = arr[0], min = arr[0];
for (int i = 1; i < N; i++) {
if (arr[i] > max) {
System.out.println("Comparison");
max = arr[i];
}
if (arr[i] < min) {
System.out.println("Comparison");
min = arr[i];
}
}
int range = max - min + 1;
/** check if range is small enough for count array **/
/**
* else it might give out of memory exception while allocating memory for
array
**/
if (range > MAX_RANGE) {
System.out.println("\nError : Range too large for sort");
return;
}
int[] count = new int[range];
/** make count/frequency array for each element **/
for (int i = 0; i < N; i++) {
count[arr[i] - min]++;
}
/** modify count so that positions in final array is obtained **/
for (int i = 1; i < range; i++) {
count[i] += count[i - 1];
}
/** modify original array **/
int j = 0;
for (int i = 0; i < range; i++)
while (j < count[i]) {
System.out.println("Comparison");
arr[j++] = i + min;
```

```java
}
}
/** Main method **/
public static void main(String[] args) {
Scanner scan = new Scanner(System.in);
System.out.println("Counting Sort Test\n");
int n, i;
/** Accept number of elements **/
System.out.println("Enter number of integer elements");
n = scan.nextInt();
/** Create integer array on n elements **/
int arr[] = new int[n];
/** Accept elements **/
System.out.println("\nEnter " + n + " integer elements");
for (i = 0; i < n; i++)
arr[i] = scan.nextInt();
/** Call method sort **/
sort(arr);
/** Print sorted Array **/
System.out.println("\nElements after sorting ");
for (i = 0; i < n; i++)
System.out.print(arr[i] + " ");
System.out.println();
}
}
```

Output Screenshot:



Radix Sort:

Radix sort is an integer sorting algorithm that sorts data with integer keys by grouping the keys by individual digits that share the same significant position and value (place value). Radix sort uses counting sort as a subroutine to sort an array of numbers. Because integers can be used to represent strings (by hashing the strings to integers), radix sort works on data types other than just integers.

Worst Case: O(nk)

Best Case: Ω(nk)

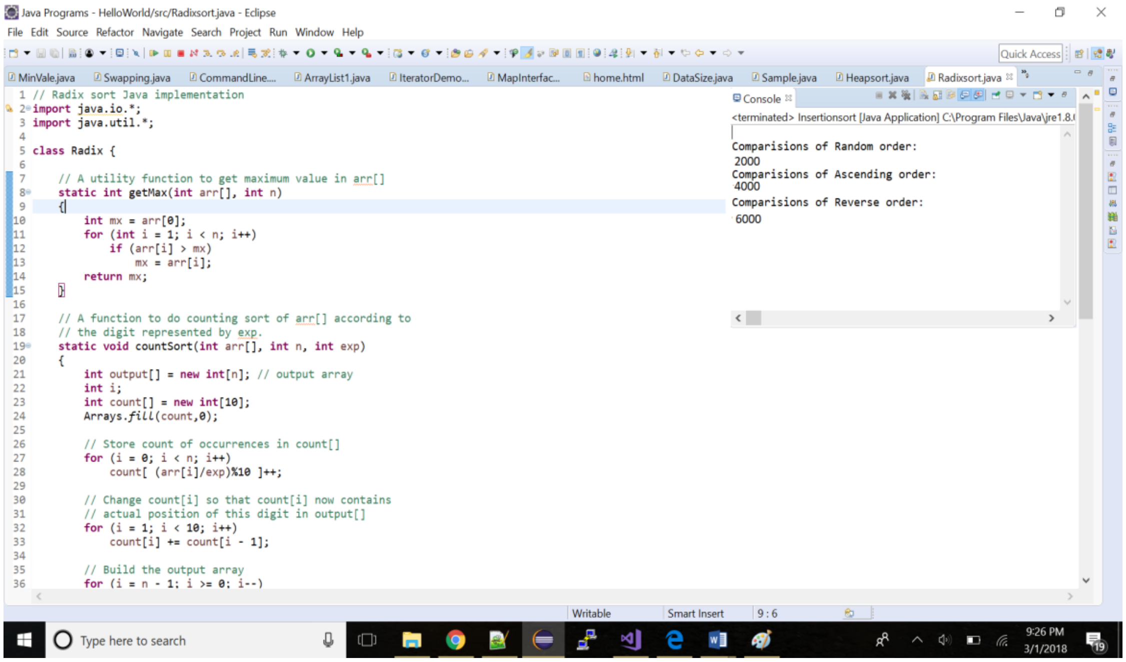
Programmatically we considered three type arrays of size 1000, an array that is sorted in ascending order **-Sorted Array**, a **Reversed array** and a **Random array**.

Source code for the Radix Sort:

```
// Radix sort Java implementation
import java.io.*;
import java.util.*;

class Radix {

    // A utility function to get maximum value in arr[]
    static int getMax(int arr[], int n)
    {
        int mx = arr[0];
        for (int i = 1; i < n; i++)
            if (arr[i] > mx)
                mx = arr[i];
        return mx;
    }

    // A function to do counting sort of arr[] according to
    // the digit represented by exp.
    static void countSort(int arr[], int n, int exp)
    {
        int output[] = new int[n]; // output array
        int i;
        int count[] = new int[10];
        Arrays.fill(count,0);

        // Store count of occurrences in count[]
        for (i = 0; i < n; i++)
            count[ (arr[i]/exp)%10 ]++;

        // Change count[i] so that count[i] now contains
        // actual position of this digit in output[]
        for (i = 1; i < 10; i++)
            count[i] += count[i - 1];

        // Build the output array
        for (i = n - 1; i >= 0; i--)
```

```
        {
            output[count[ (arr[i]/exp)%10 ] - 1] = arr[i];
            count[ (arr[i]/exp)%10 ]--;
        }

        // Copy the output array to arr[], so that arr[] now
        // contains sorted numbers according to curent digit
        for (i = 0; i < n; i++)
            arr[i] = output[i];
    }

    // The main function to that sorts arr[] of size n using
    // Radix Sort
    static void radixsort(int arr[], int n)
    {
        // Find the maximum number to know number of digits
        int m = getMax(arr, n);

        // Do counting sort for every digit. Note that instead
        // of passing digit number, exp is passed. exp is 10^i
        // where i is current digit number
        for (int exp = 1; m/exp > 0; exp *= 10)
            countSort(arr, n, exp);
    }

    // A utility function to print an array
    static void print(int arr[], int n)
    {
        for (int i=0; i<n; i++)
            System.out.print(arr[i]+" ");
    }


    /*Driver function to check for above function*/
    public static void main (String[] args)
    {
        int arr[] = {170, 45, 75, 90, 802, 24, 2, 66};
        int n = arr.length;
        radixsort(arr, n);
        print(arr, n);
    }
}
```

Output Screenshot:



The tabulated form of the number of comparisons is given below.

Algorithm	Random Array	Sorted Array	Reverse Array
Insertion Sort	248872	999	499500
Merge Sort	16019	28916	44335
Heap Sort	248872	999	499500
Quick Sort	588499	646175	657104
Counting Sort	4176	4087	4093
Radix sort	2000	4000	6000

These results approximately match the Big O Notation of the respective sorting algorithms.