1. Insertion Sort

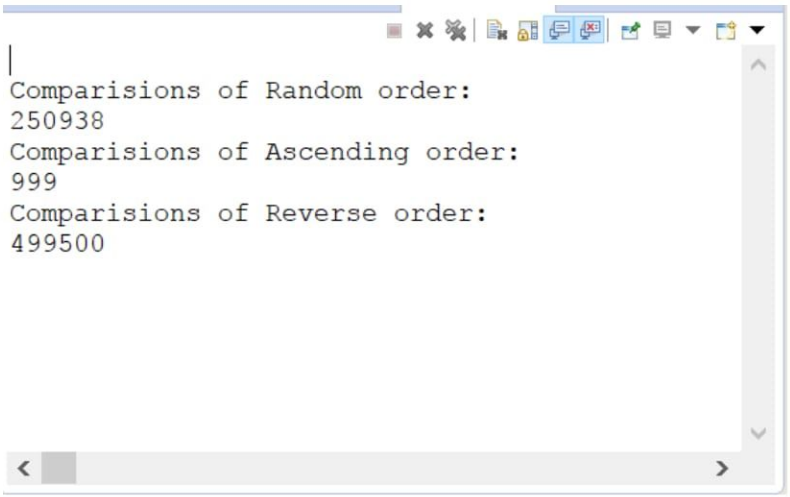
Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort.

Worst case performance : $O(n^2)$ comparisons, swaps
Best case performance : O(n) comparisons, O(1) swaps
Average case performance : $O(n^2)$ comparisons, swaps

```
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();
}
// Driver method
```

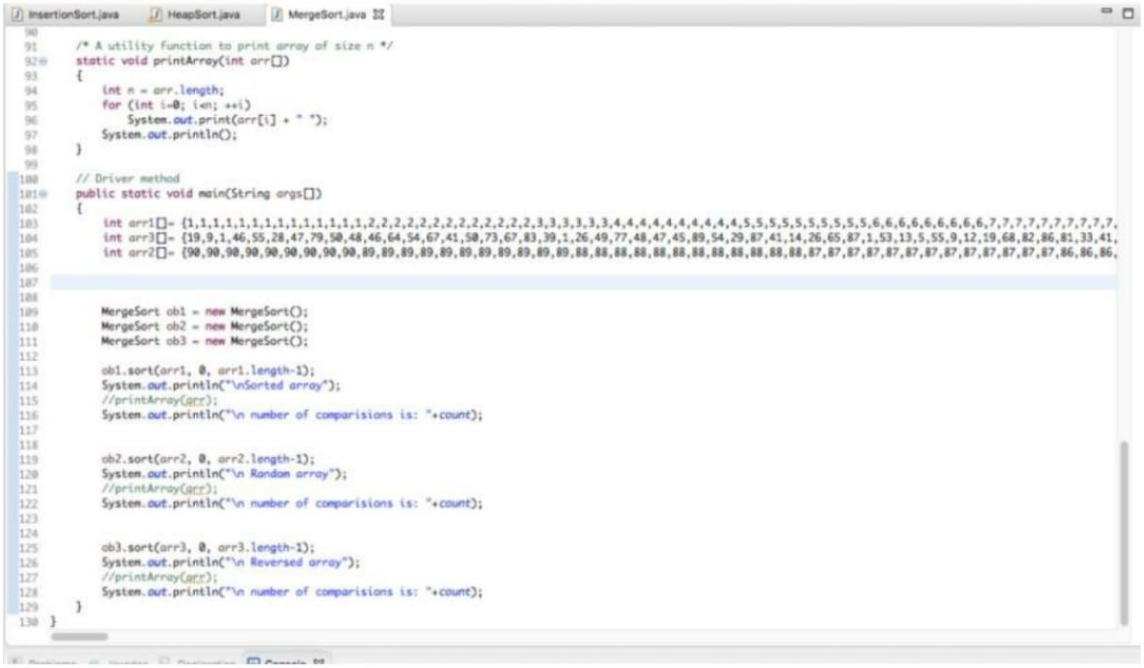
```
   public static void main(String args[])

   {  // 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);
   }

}
```



2. Merge Sort

The problem of sorting a list of numbers lends itself immediately to a divide-and-conquer strategy: split the list into two halves, recursively sort each half, and then merge the two sorted sub-lists.

Worst case performance : O(n log n)
Best case performance : O(n log n)
Average case performance : O(n log n)



Output Screen:

3.Heap Sort

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

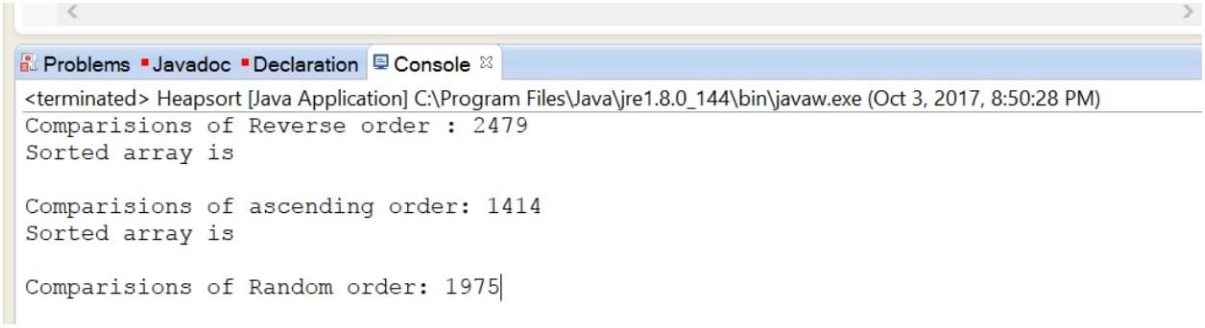
Worst case performance : O(n log n)
Best case performance : O(n log n)
Average case performance : O(n log n)

```
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[])
```

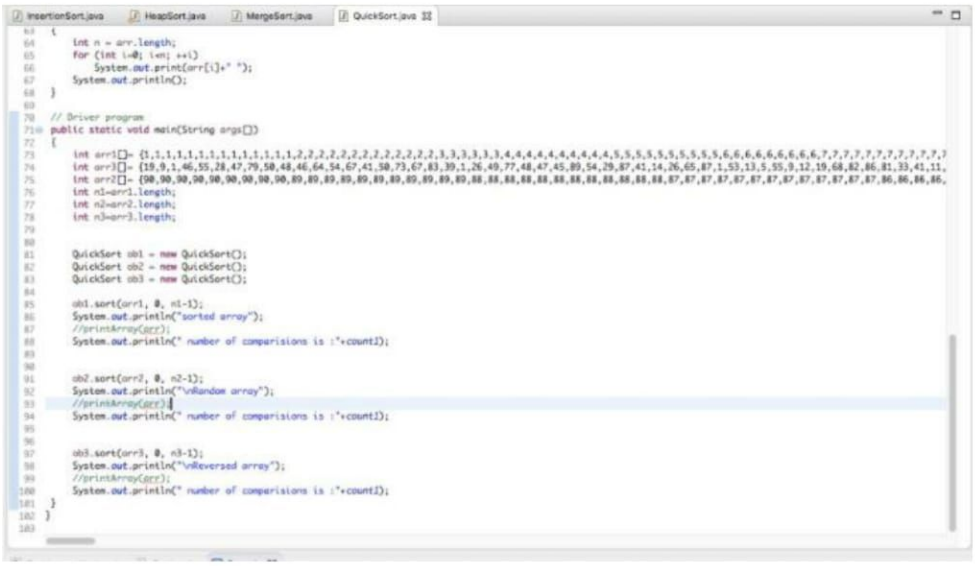
```
{
        int[] arr1 = new int[1000];
       int[] arr2 = new int[1000];
       int[] arr3 = new int[1000];
        int i = 0;
        while(i<1000){
                arr1[i] = i+1;
                i++;
        }
        i = 1000;
        int j = 0;
        while(i>0){
        arr2[j] = i;
        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("Sorted array is");
printArray(arr1);
System.out.println("Comparisions of Reverse order : "+count);
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("Sorted array is");
printArray(arr3);
System.out.println("Comparisions of Random order: "+count);
}
}
```

Output screens



4. Quick Sort

Randomized **Quick Sort** randomly selects a pivot element, after selecting pivot standard procedure is to be followed as quick sort.



Output Screens

```
<terminated> QuickSort [Java Application] /Library/Java/J
sorted array
 number of comparisions is :500499

Random array
 number of comparisions is :646175

Reversed array
 number of comparisions is :657184
```

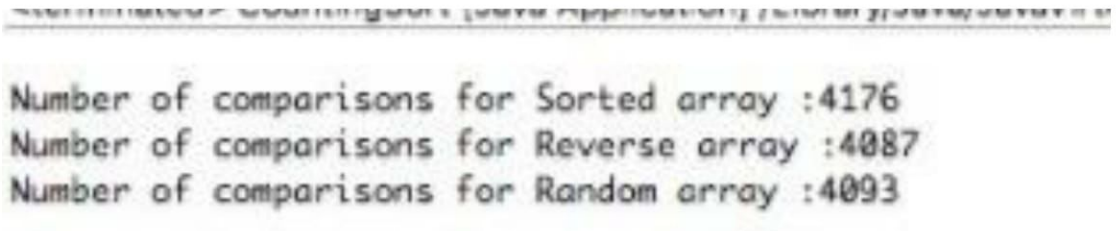
5. Counting Sort

Counting sort is a sorting technique based on keys between a specific range. It works by counting the number of objects having distinct key values (kind of hashing). Then doing some arithmetic to calculate the position of each object in the output sequence.

InsertionSort.java | HeapSort.java | MergeSort.java | QuickSort.java | CountingSort.java | RadixSort.java | Radix2.java

```
counter++;
counts[i] = counts[i] + counts[i-1];
}

/*
Sort the array right to the left
1) Look up in the array of occurences the last occurence of the given value
2) Place it into the sorted array
3) Decrement the index of the last occurence of the given value
4) Continue with the previous value of the input array (goto set1),
terminate if all values were already sorted
*/
for (int i = array.length - 1; i >= 0; i--) {
counter++;
aux[counts[array[i] - min]--] = array[i];
}
aux[1000] = counter;
return aux;
}

public static void main(String[] args) {
    int count1 = 0,count2 = 0,count3 = 0;
    |
     int arr1[]= {1,1,1,1,1,1,1,1,1,1,1,1,1,1,2,2,2,2,2,2,2,2,2,2,2,2,3,3,3,3,3,3,4,4,4,4,4,4,4,4,4,4,5,5,5,5,5,5,5,5,5,5,6,6,6,6,6,6,6,6,6,6,7,7,7,7,7,7,7,7,7,7,7,
     int arr3[]= {19,9,1,46,55,28,47,79,50,48,46,64,54,67,41,50,73,67,83,39,1,26,49,77,48,47,45,89,54,29,87,41,14,26,65,87,1,53,13,5,55,9,12,19,68,82,86,81,33,41,11,5
     int arr2[]= {90,90,90,90,90,90,90,90,90,89,89,89,89,89,89,89,89,89,89,88,88,88,88,88,88,88,88,88,88,88,88,87,87,87,87,87,87,87,87,87,87,87,87,86,86,86,86,8

        int [] sorted1 = sort(arr1,count1);
        count1 = sorted1[1000];
        int [] sorted2 = sort(arr2,count2);
        count2 = sorted2[1000];
        int [] sorted3 = sort(arr3,count3);
        count3 = sorted3[1000];

    System.out.println("");
    System.out.println("Number of comparisons for Sorted array :"+count1);
    System.out.println("Number of comparisons for Reverse array :"+count2);
    System.out.println("Number of comparisons for Random array :"+count3);
}
}
```

Output Screens



6. 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

```
import java.util.Arrays;


public class Radixsort {

static int counter=0;

static int arr[]=new int[1000];

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;

}
```

```
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);
for (i = 0; i < n; i++)
count[ (arr[i]/exp)%10 ]++;
for (i = 1; i < 10; i++)
count[i] += count[i - 1];
for (i = n - 1; i >= 0; i--)
{
output[count[ (arr[i]/exp)%10 ] - 1] = arr[i];
count[ (arr[i]/exp)%10 ]--;
counter++;
}

for (i = 0; i < n; i++)
arr[i] = output[i];
}
static void radixsort(int arr[], int n)
{
int m = getMax(arr, n);
for (int exp = 1; m/exp > 0; exp *= 10)
countSort(arr, n, exp);
counter++;
}
```

```
static void print(int arr[], int n)
{
for (int i=0; i<n; i++)
System.out.print(arr[i]+" ");
}
public static void main (String[] args)
{
        int k=0;
        while(k<1000){
                arr[k] =  0 + (int)(Math.random() * 123);
                k++;
        }
int n = arr.length;
Radixsort(arr, n);
System.out.println("No.of Computation (Randomized Array)=" +counter);
print(arr, n);
}}
```

Output Screens:

Comparing the number of comparisions :

	Random Array	Sorted Array	Reversed Array
Insertion sort	**250938**	**999**	**499500**
Merge sort	**28916**	**16019**	**44335**
Heap sort	**1975**	**1414**	**2479**
Counting sort	**4093**	**4176**	**4087**
Quick sort	**646175**	**500499**	**657104**
Radix Sort	**4000**	**2000**	**6000**

References:

http://www.sanfoundry.com/java-program-perform-counting-sort/

http://www.sanfoundry.com/java-program-perform-merge-sort/

http://www.sanfoundry.com/java-program-perform-quick-sort/

http://www.sanfoundry.com/java-program-perform-insertion-sort/

http://www.geeksforgeeks.org/merge-sort/

http://www.geeksforgeeks.org/radix-sort/

http://www.geeksforgeeks.org/heap-sort/

http://www.geeksforgeeks.org/insertion-sort/

http://www.geeksforgeeks.org/counting-sort/