Algorithms and Data Structures

Project 1: Comparison-based Sorting Algorithms

Project Developers:

Gokul Mani

Srinath Muralinathan

Introduction:

In this project, we have discussed and compared about 5 of the major sorting algorithms – Insertion Sort, Merge Sort, Heap Sort, In Place Quick Sort, Modified Quick Sort. We have also executed the code for all these sorting algorithms and have plotted the graph with respect to various data sets and time taken for each of the sort to happen. For this project we have included one normal case(Sorting random numbers) and two special cases - Sorting an already sorted array(Best Case), Sorting a reverse array(Worst Case).

Implementation and Observations:

Insertion Sort:

Insertion sort is one of the simpler sorting algorithms which is efficient for small and almost sorted data sets. The array elements are compared with each other sequentially and then arranged simultaneously in some particular order. The comparison is repeated until the input array is sorted. The algorithm implemented is an In-Place Algorithm. Hence, to summarize, Insertion sort is not efficient for large data sets and it is very efficient for smaller data sets. The time complexity of Insertion Sort is $O(n^2)$.

Merge Sort:

Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. Merge sort is efficient for small and large input data sets but it requires much more space when compared with other sorting algorithms. Time complexity of Merge Sort is O(nLogn) in all 3 cases (worst, average and best) as merge sort always divides the array into two halves and take linear time to merge two halves.

Heap Sort:

Heap sort is a comparison-based sorting technique based on Binary Heap data structure. Here, we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining element. The Heap sort algorithm is very efficient. While other sorting algorithms may grow exponentially slower as the number of items to sort increase, the time required to perform Heap sort increases logarithmically. This suggests that Heap sort is particularly suitable for sorting a huge list of items. Heap Sort's memory usage is minimal because apart from what is necessary to hold the initial list of items to be sorted, it needs no additional memory space to work. The overall time complexity of Heap Sort is O(nLogn).

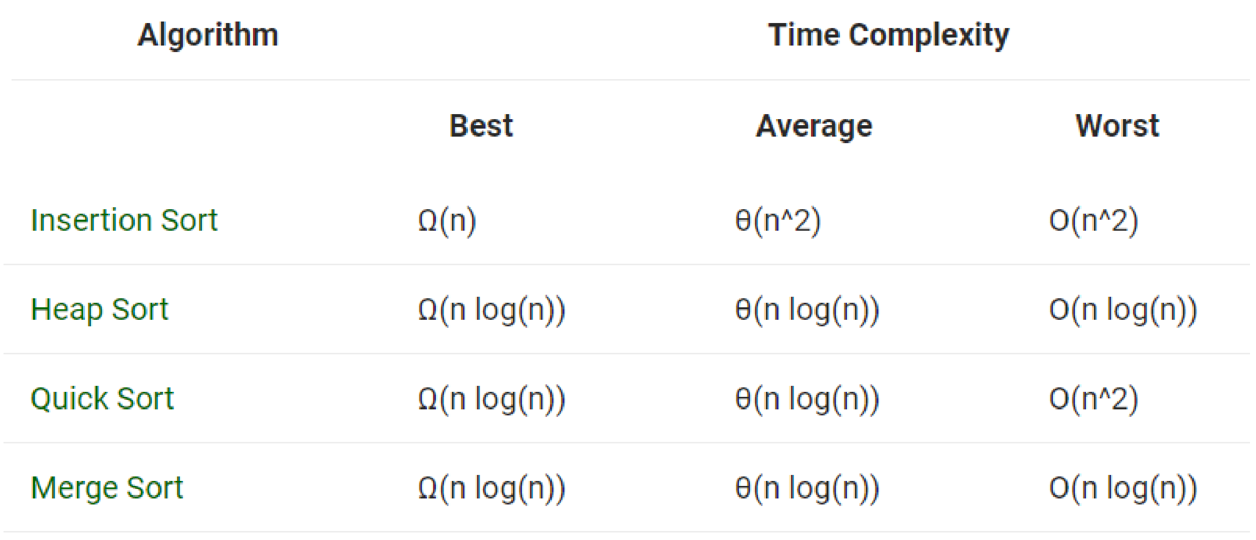
In Place Quick Sort:

Quick Sort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. Here, we have implemented in-place algorithm for quick sort. As it sorts in-place, no additional storage space is required. The quick sort is regarded as the best sorting algorithm. The worst-case time complexity of Quick Sort is O(n^2).

Modified Quick Sort – Median of 3:

This algorithm is similar to quicksort implementation. The pivot element is chosen by the median of three mechanism. That is, the pivot is the median of the first, last and middle element. The 3 elements are swapped such that the elements at positions first, middle and last are in ascending order. Then the partitioning is done recursively until the input array is sorted. The performance and efficiency of this sorting algorithm implementation is similar to merge sort and quicksort for small datasets. However, for large datasets, the implementation is observed to take more time for execution than the merge and quick sort, yet significantly lesser than the insertion sort implementation. The worst-case time complexity of Quick Sort is O(n^2).

Time Complexities of the sorting algorithms:



Algorithm	Time Complexity		
	Best	Average	Worst
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)
Heap Sort	Ω(n log(n))	θ(n log(n))	O(n log(n))
Quick Sort	Ω(n log(n))	θ(n log(n))	O(n^2)
Merge Sort	Ω(n log(n))	θ(n log(n))	O(n log(n))

Note: The time complexity of Modified quick sort and in-place quick sort are the same.

Comparison of Sorting Algorithms:

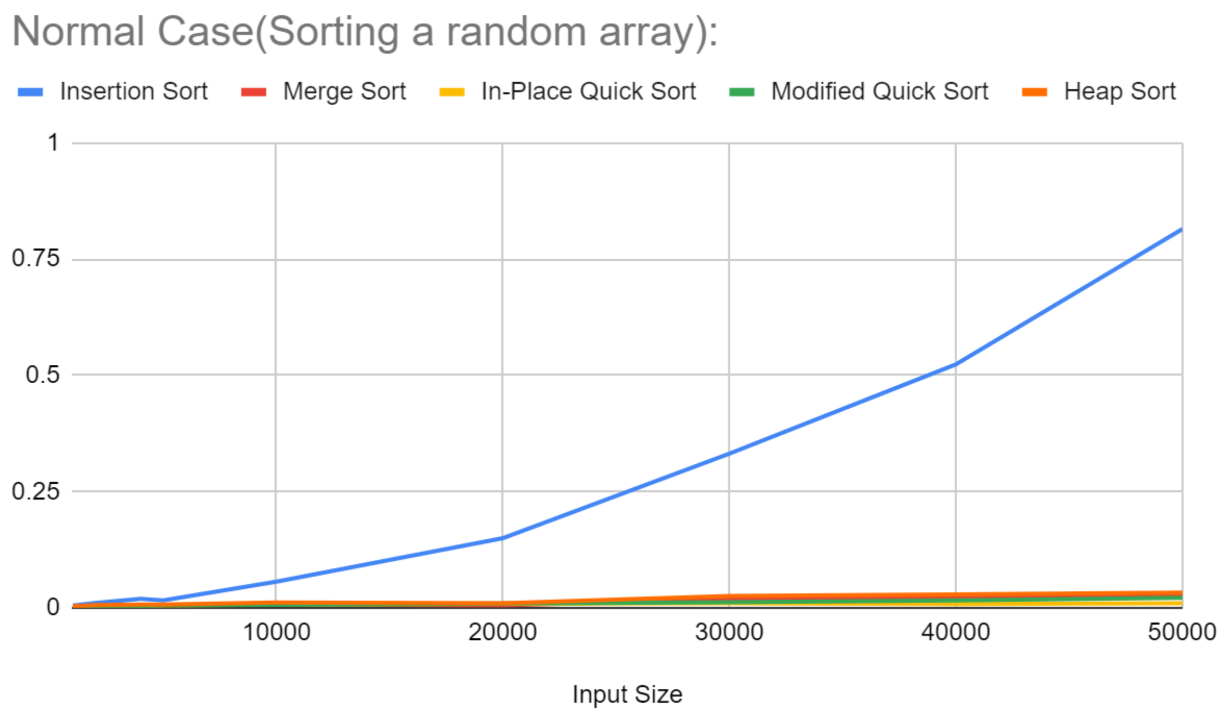
We have executed the sorting algorithms for various input data sets. We have executed all the data sets for 3 different cases – Normal Case(Sorting a random array), Best Case(Sorting an already sorted array), Worst case(Sorting an array that is in reverse order).

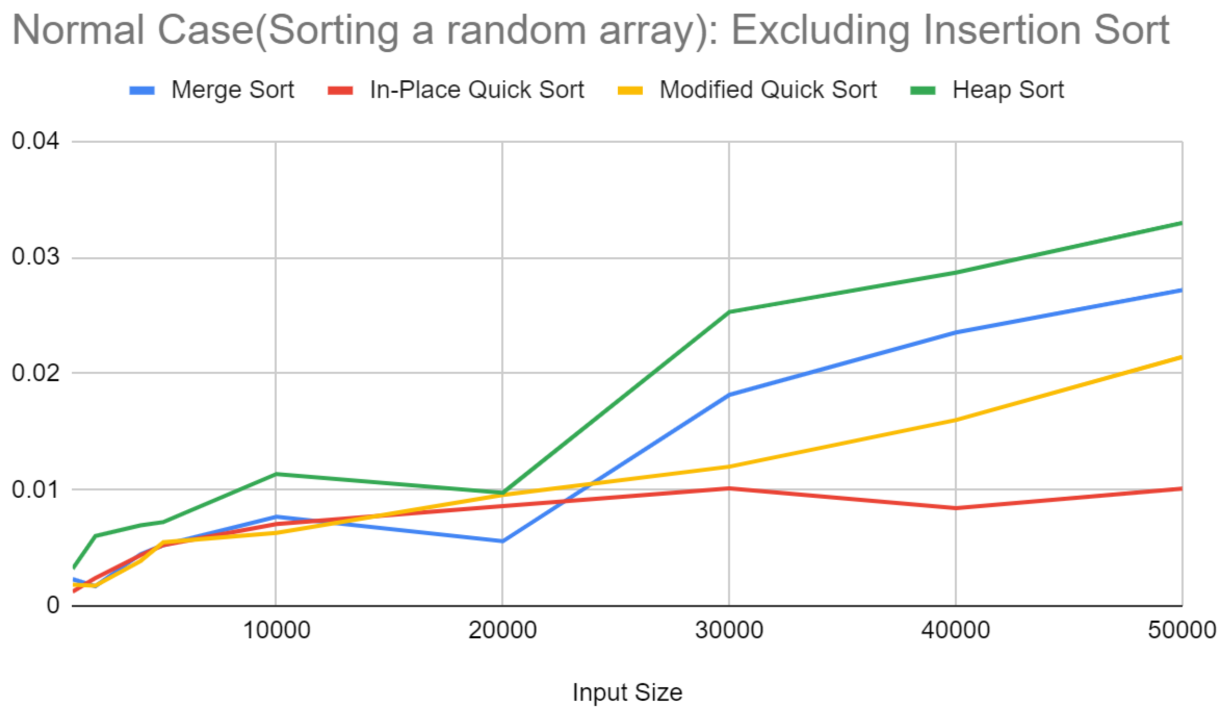
Note: All the time mentioned in the graph and tables are in seconds.

Normal Case(Sorting a random array):

By observing the table of results, we have plotted a graph for comparing the sorting algorithms for sorting a random array. From the graph, we can say that Insertion sort is efficient only for small input data sets and is very inefficient for large data sets. Merge sort, in-place quick sort, modified quick sort and heap sort algorithms takes much lesser time when compared to insertion sort for large data sets. In-Place Quick sort outperforms the rest of the sorting algorithm as observed from the graph.

Input Size	Insertion Sort	Merge Sort	In-Place Quick Sort	Modified Quick Sort	Heap Sort
1000	0.0049	0.0023	0.0012246	0.0018379	0.0032075
2000	0.0102	0.0017	0.0023913	0.0017313	0.0060129
4000	0.0194	0.0044547	0.0043599	0.0038606	0.0069298
5000	0.0158	0.0052325	0.0052377	0.005475	0.007218
10000	0.0563	0.0076761	0.0070453	0.0062819	0.0113479
20000	0.1498	0.00558	0.0085927	0.0095576	0.0097463
30000	0.3315	0.018181	0.0101265	0.0119872	0.0252997
40000	0.5238	0.0235437	0.0084252	0.0160009	0.0286868
50000	0.8148	0.0271964	0.0101057	0.0214346	0.0329715



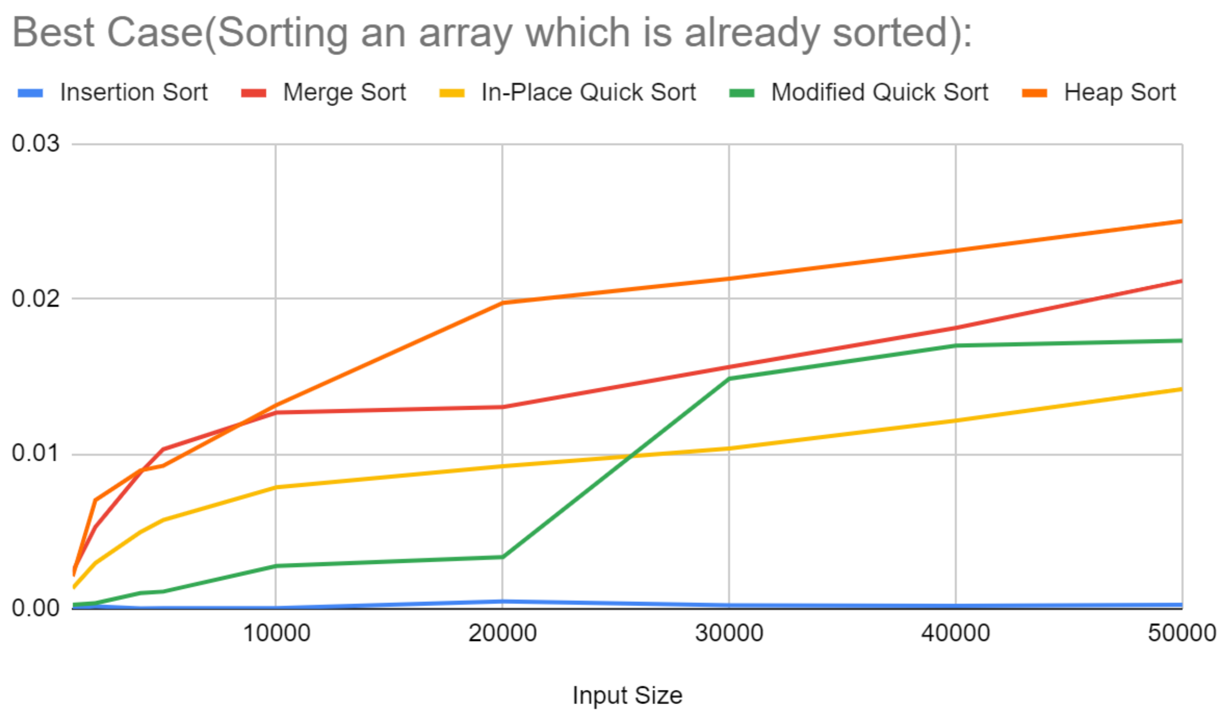


Note: As we cannot see the graph of other sorts clearly in the first graph, we have excluded insertion sort in the second graph.

Best Case(Sorting an array which is already sorted):

By observing the table of results, we have plotted a graph for comparing the sorting algorithms for sorting an already sorted array. From the graph, we can say that Insertion sort outperforms the rest of the sorts when the input is already sorted. This is because the comparison is done and no elements are swapped for insertion sort. We can also notice that apart from insertion sort, in-place quick sort outperforms modified quick sort, heap sort and merge sort.

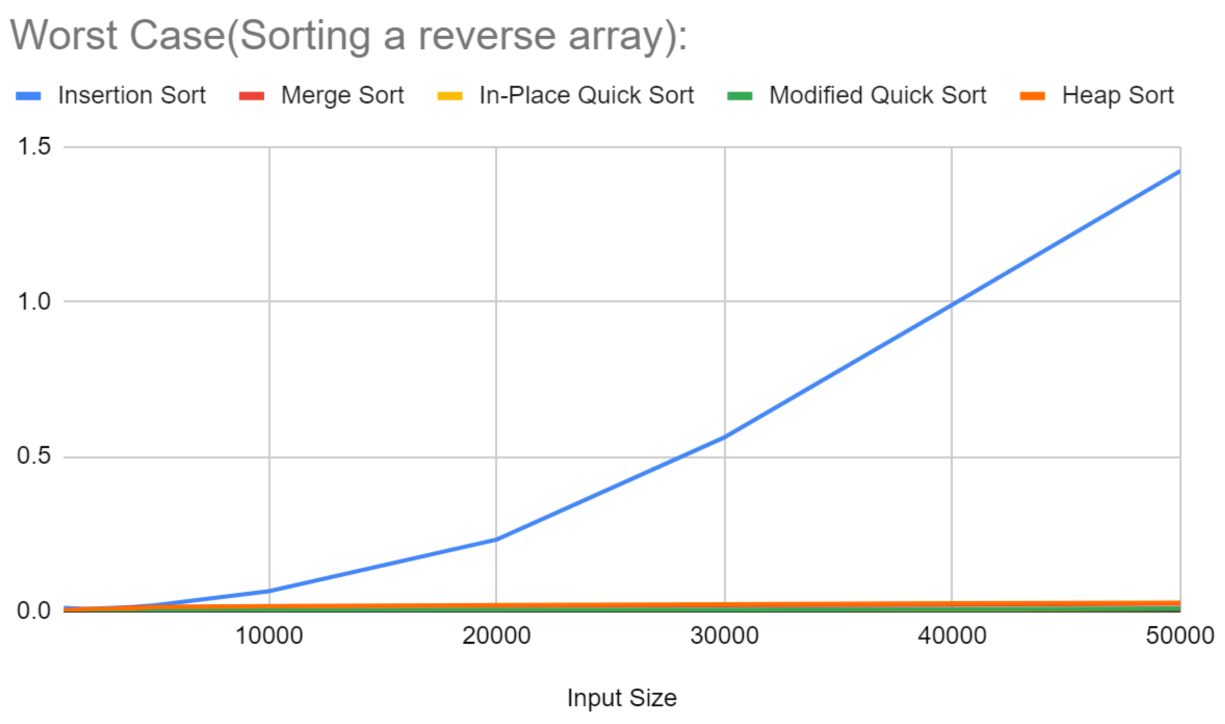
Input Size	Insertion Sort	Merge Sort	In-Place Quick Sort	Modified Quick Sort	Heap Sort
1000	0.000022	0.00239	0.0013247	2.63E-04	2.10E-03
2000	0.00016	0.005281	0.0029413	3.62E-04	7.01E-03
4000	0.000023	0.008812	0.0049568	0.0010254	8.92E-03
5000	0.000048	0.010287	0.0057327	0.0011105	9.23E-03
10000	0.0000517	0.012657	0.0078441	0.0027676	0.0131489
20000	0.000486	0.013013	0.0091967	0.0033385	1.97E-02
30000	0.000232	0.015598	0.0103495	0.0148498	2.13E-02
40000	0.0002004	0.018128	0.0121354	0.0169815	2.31E-02
50000	0.0002609	0.0211573	0.0141817	0.0172986	2.50E-02



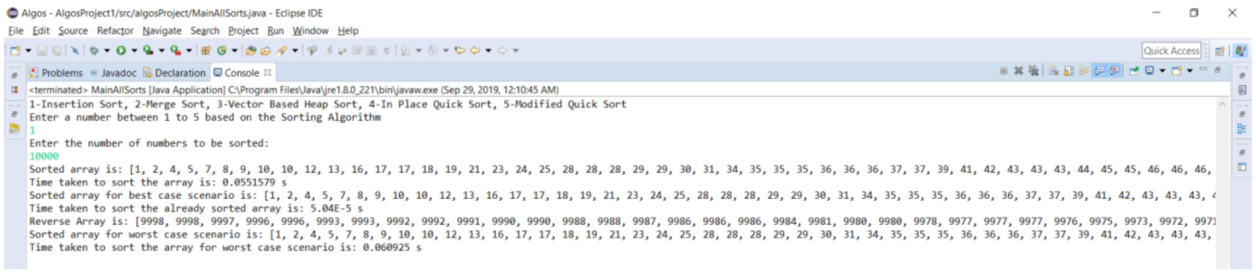
Worst Case(Sorting an array which is in reverse order):

By observing the table of results, we have plotted a graph for comparing the sorting algorithms for sorting an array that is in reverse order. From the graph, we can say that insertion sort takes a much longer time to execute than the rest of the sorting algorithms. This is because, when the array to be sorted in in reverse order, all the elements needs to be swapped and hence it will take a lot of time to execute. The other sorting algorithms does not differ significantly with respect to the order of the array.

Input Size	Insertion Sort	Merge Sort	In-Place Quick Sort	Modified Quick Sort	Heap Sort
1000	0.0111	0.002596	2.86E-04	4.27E-04	3.60E-03
2000	0.0079	0.002137	4.73E-04	4.48E-04	8.03E-03
4000	0.0139	0.013235	0.0013072	0.0019469	9.94E-03
5000	0.02	0.014006	0.0023522	0.0025634	1.36E-02
10000	0.065	0.017445	0.0032456	0.0034427	1.62E-02
20000	0.232	0.017185	0.0037228	0.0042883	2.07E-02
30000	0.562	0.019586	0.0041421	0.0053668	2.32E-02
40000	0.9903	0.021343	0.0060185	0.0072944	2.66E-02
50000	1.4222	0.0245073	0.0086344	0.0098622	2.80E-02



Sample Execution Result:


Algos - AlgosProject1/src/algosProject/MainAllSorts.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Problems Javadoc Declaration Console

```
<terminated> MainAllSorts [Java Application] C:\Program Files\Java\jre1.8.0_221\bin\javaw.exe (Sep 29, 2019, 12:10:45 AM)
1-Insertion Sort, 2-Merge Sort, 3-Vector Based Heap Sort, 4-In Place Quick Sort, 5-Modified Quick Sort
Enter a number between 1 to 5 based on the Sorting Algorithm
1
Enter the number of numbers to be sorted:
10000
Sorted array is: [1, 2, 4, 5, 7, 8, 9, 10, 10, 12, 13, 16, 17, 17, 18, 19, 21, 23, 24, 25, 28, 28, 28, 29, 29, 30, 31, 34, 35, 35, 35, 36, 36, 36, 37, 37, 39, 41, 42, 43, 43, 43, 44, 45, 45, 46, 46, 46,
Time taken to sort the array is: 0.0551579 s
Sorted array for best case scenario is: [1, 2, 4, 5, 7, 8, 9, 10, 10, 12, 13, 16, 17, 17, 18, 19, 21, 23, 24, 25, 28, 28, 28, 29, 29, 30, 31, 34, 35, 35, 35, 36, 36, 36, 37, 37, 39, 41, 42, 43, 43, 43, 4
Time taken to sort the already sorted array is: 5.04E-5 s
Reverse Array is: [9998, 9998, 9997, 9996, 9996, 9993, 9993, 9992, 9992, 9991, 9990, 9990, 9988, 9988, 9987, 9986, 9986, 9986, 9984, 9981, 9980, 9980, 9978, 9977, 9977, 9977, 9976, 9975, 9973, 9972, 9971
Sorted array for worst case scenario is: [1, 2, 4, 5, 7, 8, 9, 10, 10, 12, 13, 16, 17, 17, 18, 19, 21, 23, 24, 25, 28, 28, 28, 29, 29, 30, 31, 34, 35, 35, 35, 36, 36, 36, 37, 37, 39, 41, 42, 43, 43, 43,
Time taken to sort the array for worst case scenario is: 0.060925 s
```

Code:

MainAllSorts.java:

```java
package algosProject;

import java.util.Arrays;
import java.util.Random;
import java.util.Scanner;

public class MainAllSorts {

	public static void main(String[] args) {
		// TODO Auto-generated method stub
		System.out.println("1-Insertion Sort, 2-Merge Sort, 3-Heap Sort, 4-In Place Quick Sort, 5-Modified Quick Sort");
		System.out.println("Enter a number between 1 to 5 based on the Sorting Algorithm ");
		Scanner input=new Scanner(System.in);
		int sort=input.nextInt();
		switch (sort) {
      case 1:
         InsertionSort();
        break;
      case 2:
         MergeSort();
        break;
      case 3:
         HeapSort();
        break;
      case 4:
         InPlaceQuickSort();
        break;
      case 5:
         ModifiedQuickSort();
```

```
            break;

        default:
            System.out.println("Please enter a number between 1 to 5");
        }
             }
             static void InsertionSort(){
                        InsertionSort is=new InsertionSort();
                        ReverseArray ra=new ReverseArray();
                        PrintArray pa=new PrintArray();
                        System.out.println("Enter the number of numbers to be sorted: ");
                        Scanner input=new Scanner(System.in);
                        int n=input.nextInt();
                        long start = System.nanoTime();
                        int ar[]= new int[n];
        Random randomGenerator = new Random();
        for (int i = 0; i < ar.length; ++i){
            ar[i] = randomGenerator.nextInt(9999);
        }

                        is.insertionSort(ar);
                long end = System.nanoTime();

                        System.out.println("Sorted array is: "+Arrays.toString(ar));
                System.out.println("Time taken to sort the array is: "+(end - start)/1000000000.0 + "
s");

                start = System.nanoTime();
                is.insertionSort(ar);
                end = System.nanoTime();

                        System.out.println("Sorted array for best case scenario is: "+Arrays.toString(ar));
                System.out.println("Time taken to sort the already sorted array is: "+(end -
start)/1000000000.0 + " s");

                ra.Reverse(ar, 0,ar.length-1);
                        System.out.println("Reverse Array is: "+Arrays.toString(ar));
                start = System.nanoTime();
                is.insertionSort(ar);
                end = System.nanoTime();

                        System.out.println("Sorted array for worst case scenario is: "+Arrays.toString(ar));
                System.out.println("Time taken to sort the array for worst case scenario is: "+(end -
start)/1000000000.0 + " s");
```

```
            }
            static void MergeSort() {
                        MergeSort ms=new MergeSort();
                        ReverseArray ra=new ReverseArray();
                        System.out.println("Enter the number of numbers to be sorted: ");
                        Scanner input=new Scanner(System.in);
                        int n=input.nextInt();
                        long start = System.nanoTime();
                        int ar[]= new int[n];
        Random randomGenerator = new Random();
        for (int i = 0; i < ar.length; ++i){
           ar[i] = randomGenerator.nextInt(9999);
        }
        ms.sort(ar, 0, ar.length-1);
               long end = System.nanoTime();

                        System.out.println("Sorted array is: "+Arrays.toString(ar));

               System.out.println("Time taken to sort the array is: "+(end - start)/1000000000.0 + "
s");

               start = System.nanoTime();
        ms.sort(ar, 0, ar.length-1);
               end = System.nanoTime();

                        System.out.println("Sorted array for best case scenario is: "+Arrays.toString(ar));
               System.out.println("Time taken to sort the array for best case scenario is: "+(end -
start)/1000000000.0 + " s");
               ra.Reverse(ar, 0,ar.length-1);
                        System.out.println("Reverse Array is: "+Arrays.toString(ar));
               start = System.nanoTime();
        ms.sort(ar, 0, ar.length-1);
               end = System.nanoTime();

                        System.out.println("Sorted array for worst case scenario is: "+Arrays.toString(ar));
               System.out.println("Time taken to sort the array for worst case scenario is: "+(end -
start)/1000000000.0 + " s");

            }
            static void HeapSort() {
                        HeapSort hs=new HeapSort();
                        ReverseArray ra=new ReverseArray();
                        System.out.println("Enter the number of numbers to be sorted: ");
```

```
                  Scanner input=new Scanner(System.in);
                  int n=input.nextInt();
                  long start = System.nanoTime();
                  int ar[]= new int[n];
        Random randomGenerator = new Random();
        for (int i = 0; i < ar.length; ++i){
           ar[i] = randomGenerator.nextInt(9999);
        }

                  hs.sort(ar);
             long end = System.nanoTime();

                  System.out.println("Sorted array is: "+Arrays.toString(ar));


             System.out.println("Time taken to sort the array is: "+(end - start)/1000000000.0 + "
s");

             start = System.nanoTime();
             hs.sort(ar);
             end = System.nanoTime();
                  System.out.println("Sorted array for best case scenario is: "+Arrays.toString(ar));
             System.out.println("Time taken to sort the already sorted array is: "+(end -
start)/1000000000.0 + " s");

             ra.Reverse(ar, 0,ar.length-1);
                  System.out.println("Reverse Array is: "+Arrays.toString(ar));
             start = System.nanoTime();
             hs.sort(ar);
             end = System.nanoTime();

                  System.out.println("Sorted array for worst case scenario is: "+Arrays.toString(ar));
             System.out.println("Time taken to sort the array for worst case scenario is: "+(end -
start)/1000000000.0 + " s");

           }
           static void InPlaceQuickSort() {
                  InPlace_QuickSort qs=new InPlace_QuickSort();
                  ReverseArray ra=new ReverseArray();
                  System.out.println("Enter the number of numbers to be sorted: ");
                  Scanner input=new Scanner(System.in);
                  int n=input.nextInt();
                  long start = System.nanoTime();
                  int ar[]= new int[n];
        Random randomGenerator = new Random();
        for (int i = 0; i < ar.length; ++i){
          ar[i] = randomGenerator.nextInt(9999);
        }
        qs.sort(ar, 0, ar.length-1);
              long end = System.nanoTime();

                    System.out.println("Sorted array is: "+Arrays.toString(ar));

              System.out.println("Time taken to sort the array is: "+(end - start)/1000000000.0 + "
s");

              start = System.nanoTime();
        qs.sort(ar, 0, ar.length-1);
              end = System.nanoTime();

                    System.out.println("Sorted array for best case scenario is: "+Arrays.toString(ar));
              System.out.println("Time taken to sort the array for best case scenario is: "+(end -
start)/1000000000.0 + " s");
              ra.Reverse(ar, 0,ar.length-1);
                    System.out.println("Reverse Array is: "+Arrays.toString(ar));
              start = System.nanoTime();
        qs.sort(ar, 0, ar.length-1);
              end = System.nanoTime();

                    System.out.println("Sorted array for worst case scenario is: "+Arrays.toString(ar));
              System.out.println("Time taken to sort the array for worst case scenario is: "+(end -
start)/1000000000.0 + " s");

           }
           static void ModifiedQuickSort() {
                    ModifiedQuickSort mqs=new ModifiedQuickSort();
                    reverseComparable ra=new reverseComparable();
                    System.out.println("Enter the number of numbers to be sorted: ");
                    Scanner input=new Scanner(System.in);
                    int n=input.nextInt();
                    long start = System.nanoTime();
                    Comparable[] ar= new Comparable[n];
        Random randomGenerator = new Random();
        for (int i = 0; i < ar.length; ++i){
          ar[i] = randomGenerator.nextInt(9999);
        }
        mqs.quicksort(ar, 0, ar.length-1);
              long end = System.nanoTime();
```

```
                    System.out.println("Sorted array is: "+Arrays.toString(ar));

            System.out.println("Time taken to sort the array is: "+(end - start)/1000000000.0 + "
s");

            start = System.nanoTime();
            mqs.quicksort(ar, 0, ar.length-1);
            end = System.nanoTime();

                    System.out.println("Sorted array for best case scenario is: "+Arrays.toString(ar));
            System.out.println("Time taken to sort the array for best case scenario is: "+(end -
start)/1000000000.0 + " s");
            ra.Reverse(ar, 0,ar.length-1);
                    System.out.println("Reverse Array is: "+Arrays.toString(ar));
            start = System.nanoTime();
            mqs.quicksort(ar, 0, ar.length-1);
            end = System.nanoTime();

                    System.out.println("Sorted array for worst case scenario is: "+Arrays.toString(ar));
            System.out.println("Time taken to sort the array for worst case scenario is: "+(end -
start)/1000000000.0 + " s");

        }
}
```

ReverseArray.java:

```
package algosProject;

public class ReverseArray {
         static void Reverse(int arr[],
        int start, int end)
{
int temp;

while (start < end)
{
   temp = arr[start];
   arr[start] = arr[end];
   arr[end] = temp;
   start++;
   end--;
}
}
}
```

reverseComparable.java:

```
package algosProject;

public class reverseComparable{
            static void Reverse(Comparable arr[],
           int start, int end)
{
Comparable temp;

while (start < end)
{
   temp = arr[start];
   arr[start] = arr[end];
   arr[end] = temp;
   start++;
   end--;
}
}
}
```

InsertionSort.java:

```
package algosProject;

import java.util.Arrays;
import java.util.Collections;

public class InsertionSort {

            public InsertionSort(){

            }
            public void insertionSort(int ar[]) {

                        for(int i=1;i<ar.length;++i)
                        {
                                    int temp=ar[i];
                                    int j=i-1;
                                    while(j>=0&&ar[j]>temp) {
                                                ar[j+1]=ar[j];
                                                j=j-1;
                                    }
                                    ar[j+1]=temp;
```

```
                    }


          }

}
```

MergeSort.java:

```
package algosProject;

public class MergeSort {
          void merge(int arr[], int l, int m, int r)
   {
       int n1 = m - l + 1;
       int n2 = r - m;
       int L[] = new int [n1];
       int R[] = new int [n2];
       for (int i=0; i<n1; ++i)
          L[i] = arr[l + i];
       for (int j=0; j<n2; ++j)
          R[j] = arr[m + 1+ j];
       int i = 0, j = 0;
       int k = l;
       while (i < n1 && j < n2)
       {
          if (L[i] <= R[j])
          {
             arr[k] = L[i];
             i++;
          }
          else
          {
             arr[k] = R[j];
             j++;
          }
          k++;
       }

       while (i < n1)
       {
          arr[k] = L[i];
          i++;
          k++;
       }
```

```
        while (j < n2)
        {
            arr[k] = R[j];
            j++;
            k++;
        }
    }

    void sort(int arr[], int l, int r)
    {
        if (l < r)
        {
            int m = (l+r)/2;
            sort(arr, l, m);
            sort(arr , m+1, r);
            merge(arr, l, m, r);
        }
    }
}
```

HeapSort.java:

```
package algosProject;

public class HeapSort {
                public void sort(int ar[])
                {
                    int n = ar.length;
                    for (int i = n / 2 - 1; i >= 0; i--)
                        heapify(ar, n, i);

                    for (int i=n-1; i>=0; i--)
                    {
                        int temp = ar[0];
                        ar[0] = ar[i];
                        ar[i] = temp;

                        heapify(ar, i, 0);
                    }
                }

                void heapify(int ar[], int n, int i)
                {
                    int largest = i;
```

```
          int l = 2*i + 1;
          int r = 2*i + 2;

          if (l < n && ar[l] > ar[largest])
             largest = l;
          if (r < n && ar[r] > ar[largest])
             largest = r;
          if (largest != i)
          {
             int swap = ar[i];
             ar[i] = ar[largest];
             ar[largest] = swap;

             heapify(ar, n, largest);
          }
       }
}
```

InPlace_QuickSort.java:

```
package algosProject;

public class InPlace_QuickSort {
          int partition(int arr[], int low, int high)
   {
      int pivot = arr[high];
      int i = (low-1);
      for (int j=low; j<high; j++)
      {
         if (arr[j] < pivot)
         {
            i++;

            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
         }
      }
      int temp = arr[i+1];
      arr[i+1] = arr[high];
      arr[high] = temp;

      return i+1;
   }
```

```
    void sort(int arr[], int low, int high)
    {
        if (low < high)
        {
            int pi = partition(arr, low, high);
            sort(arr, low, pi-1);
            sort(arr, pi+1, high);
        }
    }
}
```

ModifiedQuickSort.java:

```
package algosProject;

public class ModifiedQuickSort {


    public static final int CUTOFF = 10;

    public static void quicksort( Comparable [ ] a, int low, int high ) {
        if( low + CUTOFF > high )
            insertionSort( a, low, high );
        else {
            // Sort low, middle, high
            int middle = ( low + high ) / 2;
            if( a[ middle ].compareTo( a[ low ] ) < 0 )
                swap( a, low, middle );
            if( a[ high ].compareTo( a[ low ] ) < 0 )
                swap( a, low, high );
            if( a[ high ].compareTo( a[ middle ] ) < 0 )
                swap( a, middle, high );

            swap( a, middle, high - 1 );
            Comparable pivot = a[ high - 1 ];

            int i, j;
            for( i = low, j = high - 1; ; ) {
                while( a[ ++i ].compareTo( pivot ) < 0 )
                    ;
                while( pivot.compareTo( a[ --j ] ) < 0 )
                    ;
                if( i >= j )
                    break;
                swap( a, i, j );
            }

            swap( a, i, high - 1 );

            quicksort( a, low, i - 1 );
            quicksort( a, i + 1, high );
        }
    }

    public static final void swap( Object [ ] a, int index1, int index2 ) {
        Object tmp = a[ index1 ];
        a[ index1 ] = a[ index2 ];
        a[ index2 ] = tmp;
    }

    private static void insertionSort( Comparable [ ] a, int low, int high ) {
        for( int p = low + 1; p <= high; p++ ) {
            Comparable tmp = a[ p ];
            int j;

            for( j = p; j > low && tmp.compareTo( a[ j - 1 ] ) < 0; j-- )
                a[ j ] = a[ j - 1 ];
            a[ j ] = tmp;
        }
    }
}
```