PROJECT 1

Comparison-based Sorting Algorithms

College of Computing and Informatics, University of North Carolina at Charlotte.

Algorithms and Data Structures

Dr. Dewan Tanvir Ahmed

ITCS 6114/8114 – SPRING 2024

Submitted By:

Nishant Acharekar – 801363902

Project 1: Comparison-based Sorting Algorithms

The major goal of this project is to construct comparison-based sorting algorithms and assess their efficiency over a variety of input sizes, including 1000, 2000, 3K, 5K, 10K, 20K, 40K, 50K, 60K, 80K, and 100K. We employed integer, long, array, and string data structures. This project covers the following sorting algorithms:

- Insertion sort
- Merge sort
- Heapsort [vector-based, and insert one item at a time]
- In-place quicksort (any random item or the first or the last item of your input can be pivoted).
- Modified Quicksort
    - Using median-of-three as a pivot.
    - For small sub-problems of size ≤15, using insertion sort.

The complexity study of various sorting algorithms is given below:

- **Insertion Sort:** Insertion Sort analyzes neighboring elements and swaps them if they are in the incorrect order. It begins with the smallest element and gradually sorts the array. This process will continue until all items have been sorted. The time complexity is $O(n^2)$, but the spatial complexity is O(1).
- **Merge Sort:** Merge sort uses the divide and conquer approach. It separates the list recursively into sub-lists, with each sub-list containing exactly one member. Merge sort efficiently sorts a list in O(n*log(n)) time. It is widely used to find inversions in a list and for external sorting. The spatial complexity is O(n).
- **Heap Sort:** Heap sort places elements in an array into a binary heap before repeatedly extracting the largest member and placing it into the sorted array. The time complexity is O(n*log(n)), whereas the space complexity is O(1).
- **In-place and Modified Quick Sort:** A pivot element is chosen, and other elements are arranged around it, with lower values on the left and equal or greater values on the right. Sorting is done recursively on the sub-arrays. In modified quicksort, the pivot element is taken from the array's median. Following sorting, the middle element is used to get the median. The time complexity remains O(n*log(n)), while the space complexity is O(n).

Sorting Algorithms Codes:

DriverMain.java code:

```
package algorithm;

import java.io.FileWriter;
import java.io.IOException;
import java.util.Arrays;
import java.util.Scanner;
public class DriverMain {
 public static void main(String[] args) {
  try {
  String charRand= "---------- The Following is Comparison-Based Sorting
Algorithm for Random Numbers ----------";
  String charSort= "---------- The Following is Comparison-Based Sorting
Algorithm for Sorted Numbers ----------";
  String charRevSort= "---------- The Following is Comparison-Based Sorting
Algorithm for Reversely Sorted Numbers ----------";
     Scanner in = new Scanner(System.in);

  FileWriter opFile = new FileWriter("myOutput.txt",true);

        System.out.println("Enter the no of input size to be sorted: ");
        int n = Integer.parseInt(in.nextLine());
        in.close();

        int randIpArray[]= AllSortingAlgorithm.getIntRandNos(n);
        int randIpArray1[]= Arrays.copyOf(randIpArray, randIpArray.length);
        int randIpArray2[]= Arrays.copyOf(randIpArray, randIpArray.length);

        opFile.write("\n The Ip Size are:" + n);
        opFile.flush();
        opFile.close();

        System.out.println( '\n' +" ------------- The Following is Comparison-
Based Sorting Algorithm for Random Numbers -------------" + '\t' + '\n');

        calTime.randSorted(randIpArray,charRand);

        System.out.println( '\n'  +"------------- The Following is Comparison-
Based Sorting Algorithm for Sorted Numbers -------------" + '\n');
  calTime.Sorting(randIpArray1,charSort);

        System.out.println( '\n'  + "------------- The Following is Comparison-
Based Sorting Algorithm for Reversely Sorted Numbers -------------" + '\n');
```

```
        calTime.revSorted(randIpArray2,charRevSort);
  }
   catch (IOException exception) {
        System.out.println("An Error occured, please check!!!");
        exception.printStackTrace();
      }

 }
}
```

AllSortingAlgorithm.java code:

```
package algorithm;

import java.util.Random;

public class AllSortingAlgorithm {

     private static final Random randNums = new Random();
     private static final int RANGE_OF_NUMS = 10000000;

        public static int[] getIntRandNos(int arrSize) {
      int[] array = new int[arrSize];
      for(int i = 0; i < array.length; i++) {
       array[i] = randNums.nextInt(RANGE_OF_NUMS);
      }
      return array;
     }
     //***** Insertion sort Code *****
        static void algoInsertionSort(int randIpArray[]) {
            int n = randIpArray.length;
            for (int i=1; i<n; ++i)
            {
                int key = randIpArray[i];
                int j = i-1;
                while (j>=0 && randIpArray[j] > key)
                {
                    randIpArray[j+1] = randIpArray[j];
                    j = j-1;
                }
                randIpArray[j+1] = key;
            }
        }
        //*****  Merge sort Code *****
        static void algoMergeSort(int left, int right,int arrayIP[])
```

```
        {
            if (left < right)
            {
                int middle = (left+right)/2;
                algoMergeSort(left, middle,arrayIP);
                algoMergeSort(middle+1, right,arrayIP);
                algoMerge(left, middle, right,arrayIP);

            }
        }
      private static void algoMerge(int left, int middle, int right,int
randIpArray[])
         {
             int m = middle - left + 1;
             int n = right - middle;
             int temp1[] = new int [m];
             int temp2[] = new int [n];
             for (int i=0; i<m; i++)
                 temp1[i] = randIpArray[left + i];
             for (int j=0; j<n; j++)
                 temp2[j] = randIpArray[middle + 1+ j];
             int i = 0, j = 0, key = left;
             while (i < m && j < n)
             {
                 if (temp1[i] <= temp2[j])
                 {
                     randIpArray[key] = temp1[i];
                     i++;
                 }
                 else
                 {
                     randIpArray[key] = temp2[j];
                     j++;
                 }
                 key++;
             }
             while (i < m)
             {
                 randIpArray[key] = temp1[i];
                 i++;
                 key++;
             }
             while (j < n)
             {
                 randIpArray[key] = temp2[j];
```

```
                j++;
                key++;
            }
        }
       public static void heapifyOp(int a, int b,int randIpArray[])
        {
            int big = b;
            int l = 2*b + 1;
            int r = 2*b + 2;

            if (l < a && randIpArray[l] > randIpArray[big])
                big = l;
            if (r < a && randIpArray[r] > randIpArray[big])
                big = r;
            if (big != b)
            {
                int swapVal = randIpArray[b];
                randIpArray[b] = randIpArray[big];
                randIpArray[big] = swapVal;

                heapifyOp( a, big,randIpArray);
            }
        }
        //*****  Heap sort Code *****
        public static void heapSortAlgo(int randIpArray[])
        {
            int n = randIpArray.length;
            for (int i = n / 2 - 1; i >= 0; i--) {
              heapifyOp( n, i,randIpArray);
            }
            for (int i=n-1; i>=0; i--)
            {
                int temp = randIpArray[0];
                randIpArray[0] = randIpArray[i];
                randIpArray[i] = temp;
                heapifyOp( i, 0,randIpArray);
            }
        }

        //***** In-Place quick sort Code *****
    static void algoInPlaceQuickSort( int leftint, int rightint,int
randIpArray[]) {
      if(leftint >= rightint)
       return;
      final int sintRandom = rightint - leftint + 1;
```

```
      int pivot = randNums.nextInt(intRandom) + leftint;
      int newPivot = algoInPlacePartition(leftint, rightint, pivot,randIpArray);

      algoInPlaceQuickSort(leftint, newPivot-1,randIpArray);
      algoInPlaceQuickSort(newPivot+1, rightint,randIpArray);
     }
     private static int algoInPlacePartition(int leftint, int rightint, int
pivot,int
    randIpArray[]) {
      int pivotTemp = randIpArray[pivot];
      swapVal( pivot, rightint,randIpArray);
      int temp = leftint;
      for(int i = leftint; i <= (rightint - 1); i++) {
              if(randIpArray[i] < pivotTemp) {
                  swapVal( i, temp,randIpArray);
                  temp++;
              }
          }
          swapVal( temp, rightint,randIpArray);
          return temp;
     }

        //*****  Modified Quick Sort Code *****
     static void algoModifiedQuickSort(int leftint, int rightint,int[] ipArr)
     {
      if(leftint + 15 <= rightint)
      {
       int pivot = getMedianVal(leftint, rightint,ipArr);
       int middle = getMiddleVal( leftint, rightint, pivot,ipArr);
       algoModifiedQuickSort(leftint, middle-1,ipArr );
       algoModifiedQuickSort(middle+1, rightint,ipArr );
      }
      else
      {
       algoInsertionSort(ipArr);
      }
     }
     private static int getMiddleVal( int leftint, int rightint, int pivot,int[]
    ipArr)
     {
      int i = leftint, j = rightint - 1;
      while(true)
      {
       while(ipArr[++i] < pivot);
       while(pivot < ipArr[--j]);
```

```
      if(i >= j)
       break;
      else
       swapVal(i, j,ipArr );
     }
     swapVal(i, rightint-1,ipArr);
     return i;
    }
    private static int getMedianVal(int leftint, int rightint,int[] ipArr ) {
     int center = (leftint+rightint)/2;
     if(ipArr[center] < ipArr[leftint])
      swapVal( center, leftint,ipArr);
     if(ipArr[rightint] < ipArr[leftint])
      swapVal( rightint, leftint,ipArr);
     if(ipArr[rightint] < ipArr[center])
      swapVal(rightint, center,ipArr );
     swapVal( center, rightint-1,ipArr);
     return ipArr[rightint-1];
    }
    private static void swapVal(int s, int t,int[] ipArr) {
     int temp = ipArr[s];
     ipArr[s] = ipArr[t];
     ipArr[t] = temp;
    }
   }
```

calTime.java code:

```
package algorithm;

import java.util.Arrays;
import java.io.FileWriter;
import java.io.IOException;
public class calTime {
 public static void Sorting(int[] randArray,String opHead) {
  Arrays.sort(randArray);
  calcTime(randArray,opHead);
 }
 public static void revSorted(int[] randArray,String opHead) {
  int[] revarraySortay= revSortedArr(randArray);
  calcTime(revarraySortay,opHead);
 }
 public static void randSorted(int[] randArray,String opHead) {
  int[] arr1=Arrays.copyOf(randArray, randArray.length);
  calcTime(arr1,opHead);
```

```
 }
  private static int[] revSortedArr(int randarr[]) {
   int arrIp[]= Arrays.copyOf(randarr,randarr.length);
   int length = arrIp.length;
         for (int i=1; i<length; i++)
         {
             int j = arrIp[i];
             int k = i-1;
             while (k>=0 && arrIp[k] < j)
             {
                 arrIp[k+1] = arrIp[k];
                 k = k-1;
             }
             arrIp[k+1] = j;
         }
   return arrIp;
  }
  static void calcTime(int arrayInt[],String a) {
   try {
  FileWriter opFile = new
FileWriter("C:\\Users\\firee\\OneDrive\\Desktop\\Project_1\\myOutput.txt",true);
  opFile.write('\n'+a+'\n');
  int arraySort[] = Arrays.copyOf(arrayInt, arrayInt.length);
  int arraySort1[] = Arrays.copyOf(arraySort, arraySort.length);
  int arraySort2[] = Arrays.copyOf(arraySort, arraySort.length);
  int arraySort3[] = Arrays.copyOf(arraySort, arraySort.length);
  int arraySort4[] = Arrays.copyOf(arraySort, arraySort.length);

  System.out.println("Array before the sorting algorithm:
"+Arrays.toString(arraySort));

  //Insertion Sort Code
        long insertSortTimeStart = System.nanoTime();
        AllSortingAlgorithm.algoInsertionSort(arraySort);
        long insertSortEndTime = System.nanoTime();
        System.out.println("------ Insertion Sort ------");
        System.out.println("The Execution Time for Insertion Sort
is  "+'\t'+(insertSortEndTime-insertSortTimeStart)+ " nanoseconds");
        opFile.write("\n------ Insertion Sort ------ \n");
        opFile.write("The Execution Time for Insertion Sort is
"+'\t'+(insertSortEndTime-insertSortTimeStart)+ " nanoseconds"+'\n');
        System.out.println("Ip Elements Sorted:
"+Arrays.toString(arraySort)+'\n');

   // Merge Sort Code
```

```
        long  mergeSortTimeStart = System.nanoTime();
        AllSortingAlgorithm.algoMergeSort(0, arraySort3.length-1,arraySort3);
        long mergeSortEndTime = System.nanoTime();
        System.out.println("------ Merge Sort ------");
        System.out.println("The Execution Time for Merge Sort is
"+'\t'+(mergeSortEndTime-mergeSortTimeStart)+ " nanoseconds");
        opFile.write("\n------ Merge Sort ------\n");
        opFile.write("\nThe Execution Time for Merge Sort is
"+'\t'+(mergeSortEndTime-mergeSortTimeStart)+ " nanoseconds"+'\n');
        System.out.println("Ip Elements Sorted:
"+Arrays.toString(arraySort3)+'\n');

    // Heap Sort Code
        long heapSortTimeStart = System.nanoTime();
        AllSortingAlgorithm.heapSortAlgo(arraySort4);
        long heapSortEndTime = System.nanoTime();
        System.out.println("------ Heap Sorting ------");
        System.out.println("The Execution Time for Heap Sort is
"+'\t'+(heapSortEndTime-heapSortTimeStart)+" nanoseconds");
        opFile.write("\n------ Heap Sorting ------\n");
        opFile.write("\n The Execution Time for Heap Sort
is"+'\t'+(heapSortEndTime-heapSortTimeStart)+ " nanoseconds"+'\n');
        System.out.println("Ip Elements Sorted:
"+Arrays.toString(arraySort4)+"\n");

    // In-Place QuickSorting Code
        long inplacequickSortTimeStart = System.nanoTime();
        AllSortingAlgorithm.algoInPlaceQuickSort(0, arraySort2.length-
1,arraySort2);
        long inplacequickSortEndTime = System.nanoTime();
        System.out.println("------ Inplace Quick Sorting ------");
        System.out.println(" The Execution Time for Inplace Quick is
"+'\t'+(inplacequickSortEndTime-inplacequickSortTimeStart)+ " nanoseconds");
        opFile.write("\n------ Inplace Quick Sorting ------\n");
        opFile.write("\nThe Execution Time for Inplace Quick is
"+'\t'+(inplacequickSortEndTime-inplacequickSortTimeStart)+ " nanoseconds"+'\n');
        System.out.println("Ip Elements Sorted:
"+Arrays.toString(arraySort2)+"\n");

    // Modified QuickSorting Code
        long modifiedquickSortTimeStart = System.nanoTime();
        AllSortingAlgorithm.algoModifiedQuickSort(0,arraySort1.length-
1,arraySort1);
        long modifiedquickSortEndTime = System.nanoTime();
        System.out.println("------ Modified Quick Sorting -------");
```

```
        System.out.println(" The Execution Time for Modified Quick Sort is
"+'\t'+(modifiedquickSortEndTime-modifiedquickSortTimeStart)+ " nanoseconds");
        System.out.println("Ip Elements Sorted: "+Arrays.toString(arraySort1));
        opFile.write("\n------ Modified Quick Sorting ------\n");
        opFile.write("\nThe Execution Time for Modified Quick Sort is
"+'\t'+(modifiedquickSortEndTime-modifiedquickSortTimeStart)+ "
nanoseconds"+'\n');
        opFile.close();
    }
    catch (IOException e) {
        System.out.println("An Error occured, please check!!!");
        e.printStackTrace();
      }
 }
}
```

Inputs:

Comparison-Based Sorting Algorithm for Random Numbers					
Input Size	**Insertion Sorting**	**Merge Sorting**	**Heap Sorting**	**In place quick Sorting**	**Modified quick Sorting**
1000	2149600	726600	324900	481700	603800
2000	3852600	795300	417700	776600	3435200
3000	5449000	873300	734200	801500	5228600
5000	9418400	1139400	1154100	1485600	11598600
10000	27388900	3119900	2306400	2402000	25703100
20000	92144700	3350900	2912700	3355100	73998000
40000	316097900	10529100	7530900	7825700	238747700
50000	497274800	8359200	8253500	5552900	447977300
60000	822387000	13939700	11015100	13993900	998789800
80000	1320857500	22197800	18082900	15198700	1062479900
100000	2206516900	21372800	12180500	14009900	1742835100

Table 1: Displays the input size and execution time (nanoseconds) for the Comparison-Based Sorting Algorithm for Random Numbers.

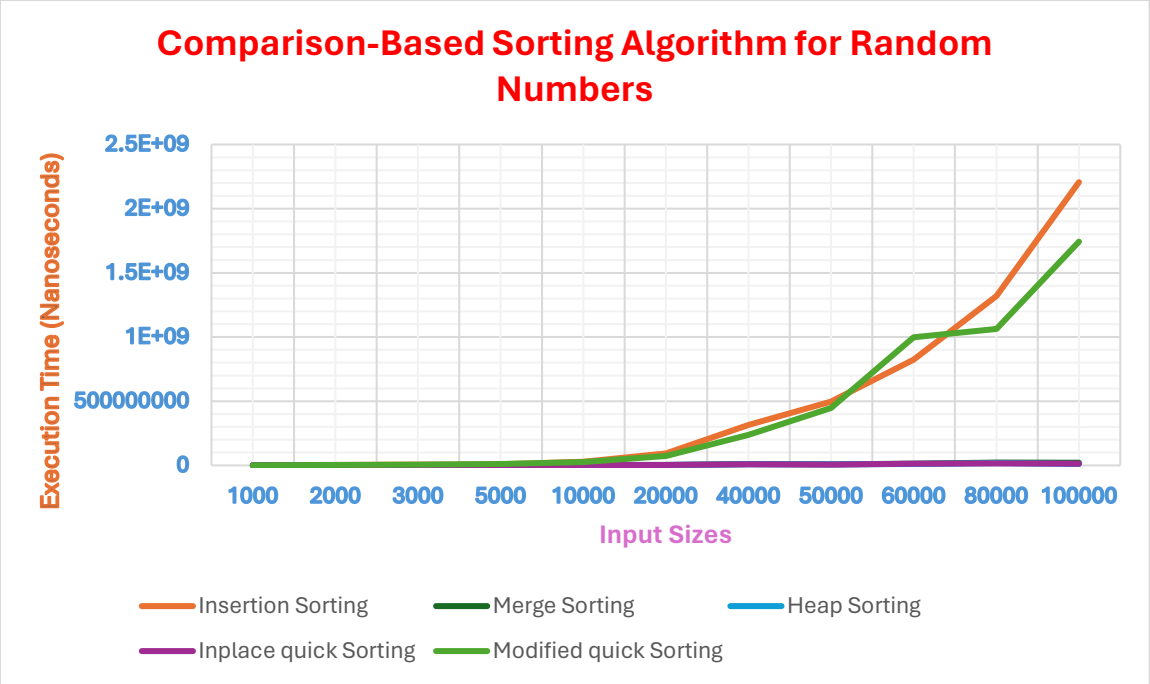

Figure 1: Depicts the Input Size versus Execution Time for the Comparison-Based Sorting Algorithm for Random Numbers.

Comparison-Based Sorting Algorithm for Sorted Numbers					
Input Size	**Insertion Sorting**	**Merge Sorting**	**Heap Sorting**	**In place quick Sorting**	**Modified quick Sorting**
1000	7600	147900	114500	201100	493800
2000	4200	314100	222400	265400	548500
3000	21100	191200	249800	142500	5521700
5000	10300	272300	400600	133300	2518700
10000	23100	744500	1140800	254400	16122400
20000	19300	1059300	1866500	789400	40563800
40000	68000	1200200	2609600	1072100	188651800
50000	46300	2708200	4327300	1628700	217625100
60000	124300	2470200	4101900	2340600	251752600
80000	75400	4039000	3936700	2185600	1028052300
100000	223700	7685600	6321300	3841000	1199335300

Table 2: Displays the input size and execution time (nanoseconds) for the Comparison-Based Sorting Algorithm for Sorted Numbers.

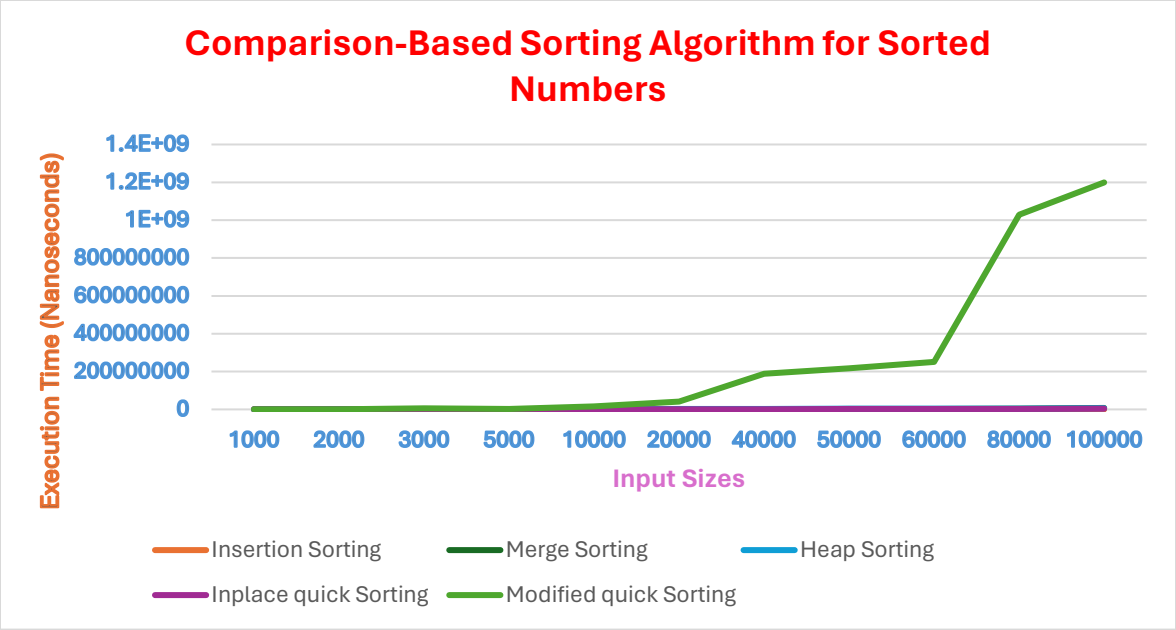

Figure 2: Depicts the Input Size versus Execution Time for the Comparison-Based Sorting Algorithm for Sorted Numbers.

Comparison-Based Sorting Algorithm for Reversely Sorted Numbers					
Input Size	**Insertion Sorting**	**Merge Sorting**	**Heap Sorting**	**In place quick Sorting**	**Modified quick Sorting**
1000	1949700	219000	131600	141400	552300
2000	695200	121400	172600	87200	292000
3000	1929000	167300	245900	131500	1739700
5000	3897100	251500	544000	160000	5420300
10000	15701100	527000	821500	317100	11515500
20000	58730500	622900	1570200	720200	46527000
40000	231774000	5058900	1991800	1290800	220831600
50000	349660800	1597100	4141000	1502300	222189200
60000	523094200	3456400	4040600	2490700	333477200
80000	923437700	3322500	3997700	2603500	758247600
100000	1311991500	6714900	6029500	4324300	1037215500

Table 3: Displays the input size and execution time (nanoseconds) for the Comparison-Based Sorting Algorithm for Reversely Sorted Numbers.

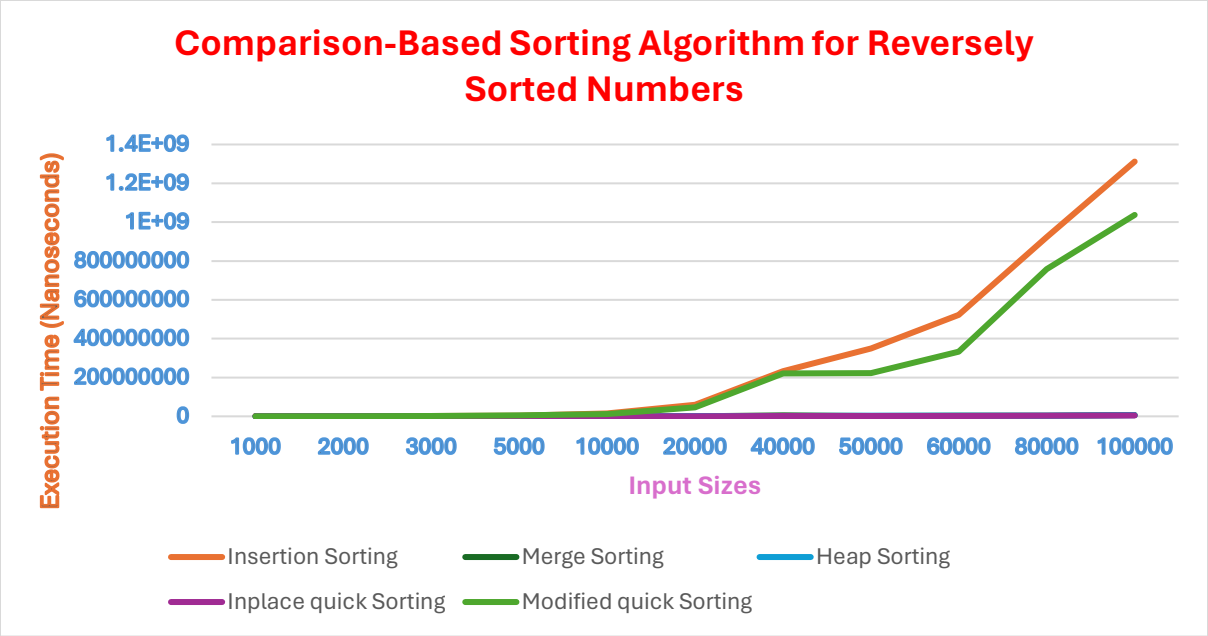

Figure 3: Depicts the Input Size versus Execution Time for the Comparison-Based Sorting Algorithm for Reversely Sorted Numbers.

Results:

Insertion sort is efficient when dealing with tiny pieces, demonstrating its usefulness in sorting this type of data. However, when we compare its performance to that of sorted and reversely sorted algorithms, it is clear that the reversely sorted algorithm has the greatest decrease in efficiency. This decline demonstrates the difficulties insertion sort confronts when dealing with data that is not naturally organized.

When we look at various sorting algorithms like merge sort, heap sort, and rapid sort, we see that their temporal complexity is similar, with all operating at O(nlogn). This constancy in time complexity shows that these sorting algorithms are efficient over a range of input sizes.

The graphs provide a visual representation of this data, providing for a better understanding of the performance differences between the sorting methods. They are useful tools for analyzing and comparing the efficiency of various algorithms under different conditions.