Name - Raveet Kumar
Roll No. - 1804310039
Branch - Computer Science and Engineering
College - BIET, Jhansi

Experiment - 01

Aim-

We have to implement 4 different sorting algorithms and study the time complexities for each of them and draw graphs of their best, worst and average case.

- *Bubble sort*
- *Insertion sort*
- *Selection sort*
- *Heap sort*

Tools & Language Used-

- ❑ Java - for coding the algorithm and calculating time
- ❑ Python - for plotting graphs using matplotlib module.

Code & Analysis-

1. Bubble sort:

- ❑ Bubble sort is an algorithm which is the simplest algorithm of all.
- ❑ In this algorithm every element is compared with the next element and if the two are not in the proper order they are swapped. In this way the largest element is at the end of the array. There are n comparisons required to push the largest element at the end of the array. These n comparisons have to be done n times for locating the current largest element and push to the end of the list.

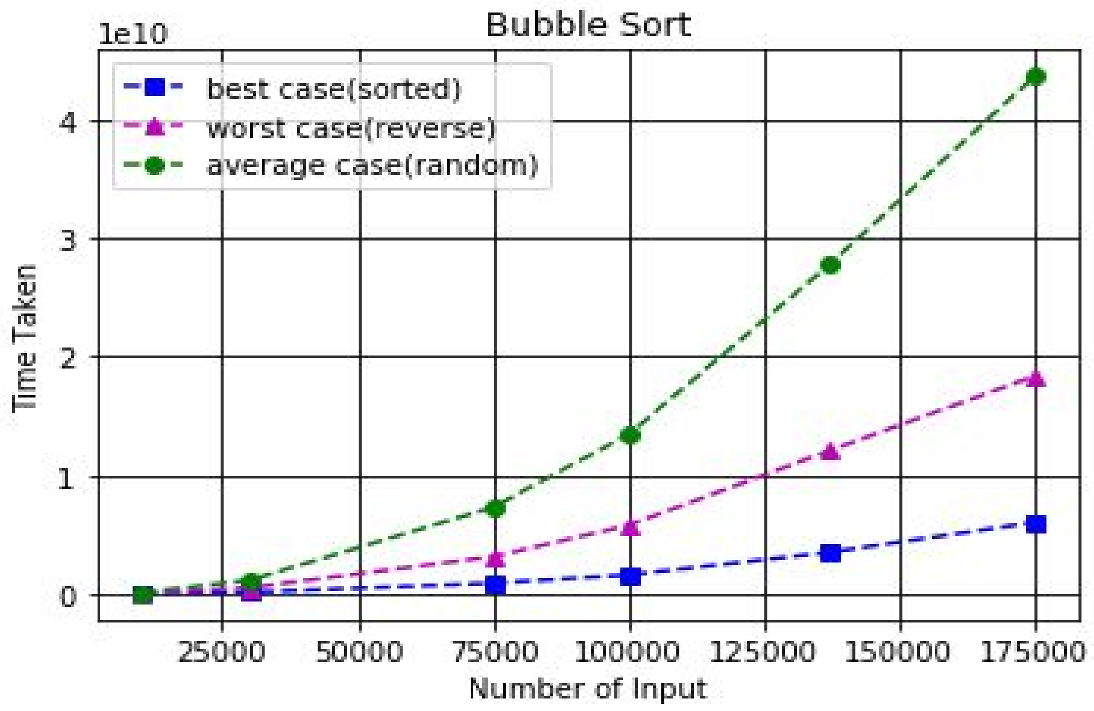
Code:

```
import java.util.*;
import java.io.*;

public class BubbleSort {
    public static void main(String[] args) throws IOException {

        File f=new File("D:\\java file handling\\bubblesort_analysis.txt");
        BufferedWriter bw=new BufferedWriter(new FileWriter(f,false),2);
        //10000,30000,75000,100000,137000,175000
        List<Integer>
TestCase=Arrays.asList(10000,30000,75000,100000,137000,175000);
        int[] best;
        int[] worst;
        int[] avg;


        int k=0;
        bw.write(" \tNumber_of_Input\tTime_Taken\n");

        while(k < TestCase.size()) {

              int arrSize=TestCase.get(k);
              best=new int[arrSize];
              worst=new int[arrSize];
              avg=new int[arrSize];

              Random rand=new Random();// To Generate Random Numbers...

//Filling Numbers in the range of (0, arrSize*10-1) in array of size arrSize
              for(int i=0;i<arrSize;i++) {
                    avg[i]=rand.nextInt(arrSize*10);
              }

              for(int i=0;i<arrSize;i++) {
                    best[i]=avg[i];
              }
// To make a sorted array... which we will use for best case..
              Arrays.sort(best);

// To make reverse order of the Best case ... to Check the worst case..
              for(int i=0;i<arrSize;i++) {
                    worst[i]=best[arrSize-1-i];
              }
```

```
            // For Best Case
            long initialTime=System.nanoTime();

            bubble_sort(best);

            long TimeTaken=System.nanoTime()-initialTime;
            bw.write(String.format("Best_Case\t%d\t%d\n",arrSize,TimeTaken));

            // For Worst Case
            initialTime=System.nanoTime();

            bubble_sort(worst);

            TimeTaken=System.nanoTime()-initialTime;
            bw.write(String.format("Worst_Case\t%d\t%d\n",arrSize,TimeTaken));

            // For Average Case
            initialTime=System.nanoTime();

            bubble_sort(avg);

            TimeTaken=System.nanoTime()-initialTime;
            bw.write(String.format("Avg_Case\t%d\t%d\n",arrSize,TimeTaken));

            // For Testing Only..
//          if(k==0) {
//                for(int i=0;i<arrSize;i++) {
//                      System.out.print(best[i] +" ");
//                }
//                System.out.println();
//                for(int i=0;i<arrSize;i++) {
//                      System.out.print(worst[i] +" ");
//                }
//                System.out.println();
//                for(int i=0;i<arrSize;i++) {
//                      System.out.print(avg[i] +" ");
//                }
//                System.out.println();
//          }
//
            k++;
            System.out.println("Success");
        }
        bw.close();
```

```
    }

    static void bubble_sort(int[] arr) {

        int temp;
        for (int i = 0; i < arr.length-1; i++) {
            for (int j = 0; j < arr.length-i-1; j++)
            if (arr[j] > arr[j+1])
            {
                    temp = arr[j];
                    arr[j] = arr[j+1];
                    arr[j+1] = temp;
            }
        }

    }

}
```

<u>Graph:</u>



Analysis:

- If the list is already sorted there are no swaps and the algorithm will run for only n times.So for the best case:O(n)
- If the list is sorted in the descending order i.e. it is sorted in the reverse order , it is the worst case and the time complexity is:O(n^2)
- The average time complexity for bubble sort is:O(n^2)

----

2. Insertion sort:

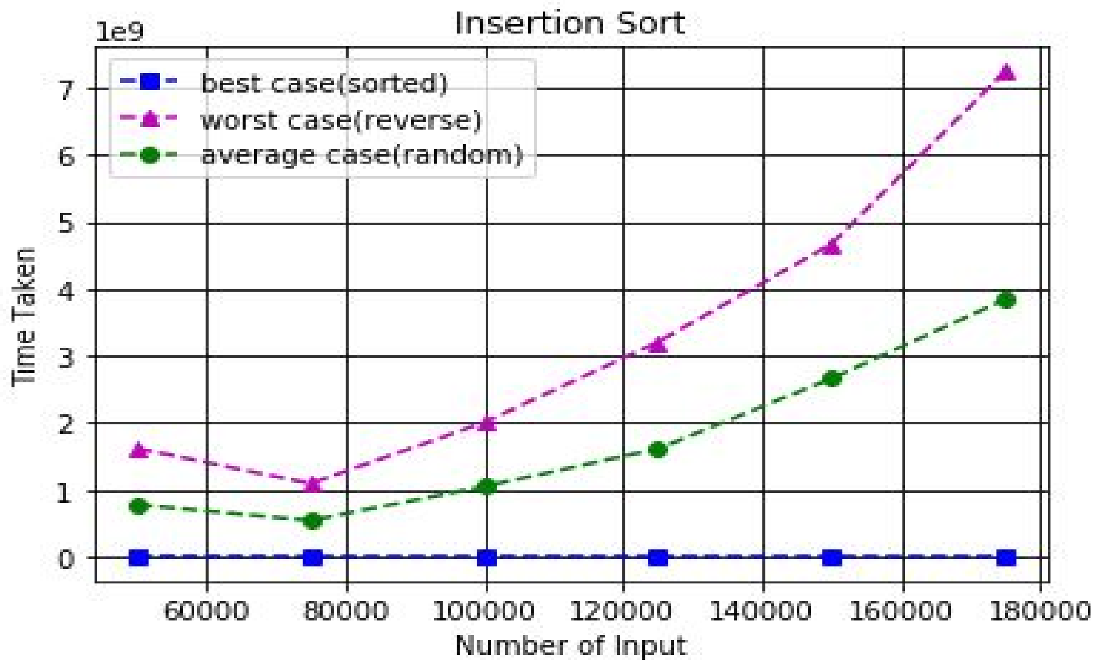
- ❑ This sort is more efficient than bubble sort and selection sort.
- ❑ In this algorithm we divide the entire array into two parts: the sorted array and the unsorted array. With every iteration we have to place the first element of the unsorted array in the correct position of the sorted array and increase the sorted list by one.

Code:

- *Main Code Will remain same ...only sorting function needs to be changed*

```
    static void insertion_sort(int[] arr) {

     int key;
     for(int i=1;i<arr.length;i++) {

           key=arr[i];
           int j=i-1;

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

           arr[j+1]=key;
     }
  }
```

Graph:



Analysis:

- When elements are sorted, there are no swaps and the correct position of the element in the sorted list is the current index itself. The time complexity is : O(n)
- When the elements are sorted in the reverse direction,there will always be swaps in proportion to the number of elements currently in the sorted list.The time complexity is: O(n^2).This is the worst case.
- The average case time complexity is:O(n^2)

----

3. Selection sort:

❑ In this sorting algorithm we divide the list into two parts: one is sorted and the other is unsorted.

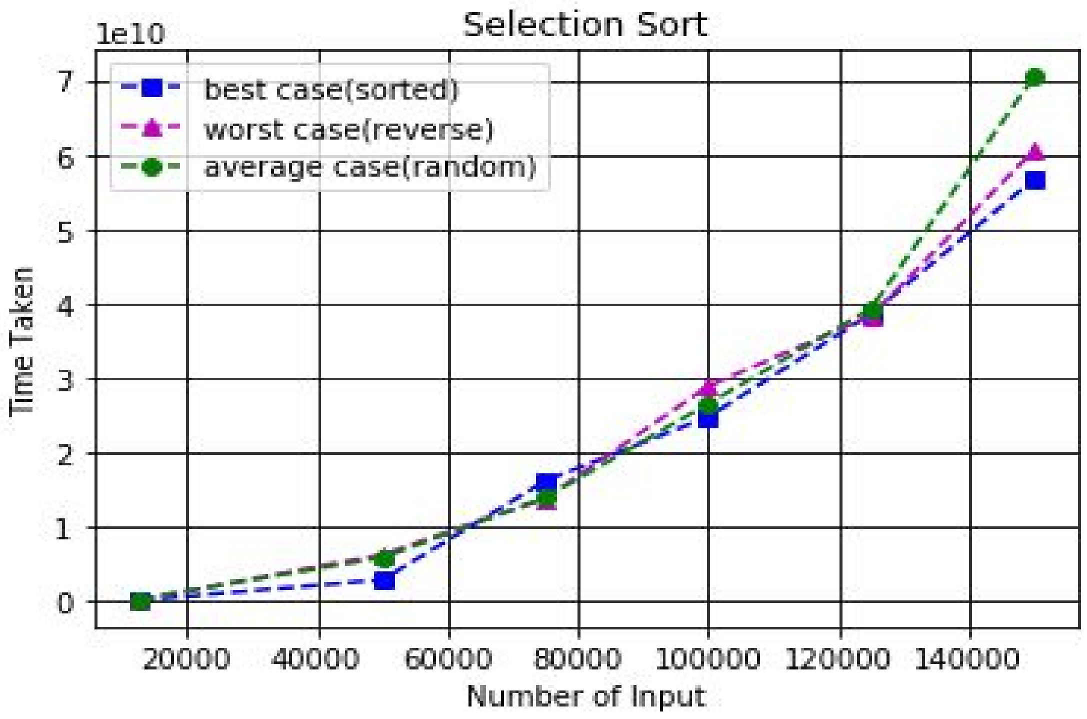
❑ Initially the entire array is an unsorted array. We find the minimum element in the array and place it in the current position of the unsorted array. Now we move the range of the unsorted array by one and then find next minimum element and place in the current first position of the unsorted list and continue the process till sorted array reaches till the end.

Code:

- *Main Code Will remain same ...only sorting function needs to be changed*

```
static void selection_sort(int[] arr) {
   int n = arr.length,temp;
   int min_index;

   for (int i = 0; i < n-1; i++)
   {
          min_index= i;

          for (int j = i+1; j < n; j++)
               if (arr[j] < arr[min_index])
                   min_index = j;

          temp = arr[min_index];
          arr[min_index] = arr[i];
          arr[i] = temp;
   }
}
```

Graph:



Analysis:

- *No matter how the data is arranged there would always be comparisons and swaps made and so the time complexity for best,average and worst case is : O(n^2)*

----

4. Heap sort:

❏ Heap sort is a comparison based sorting technique based on Binary Heap data structure.

❏ It is very fast as compared to bubble,selection or insertion sort.

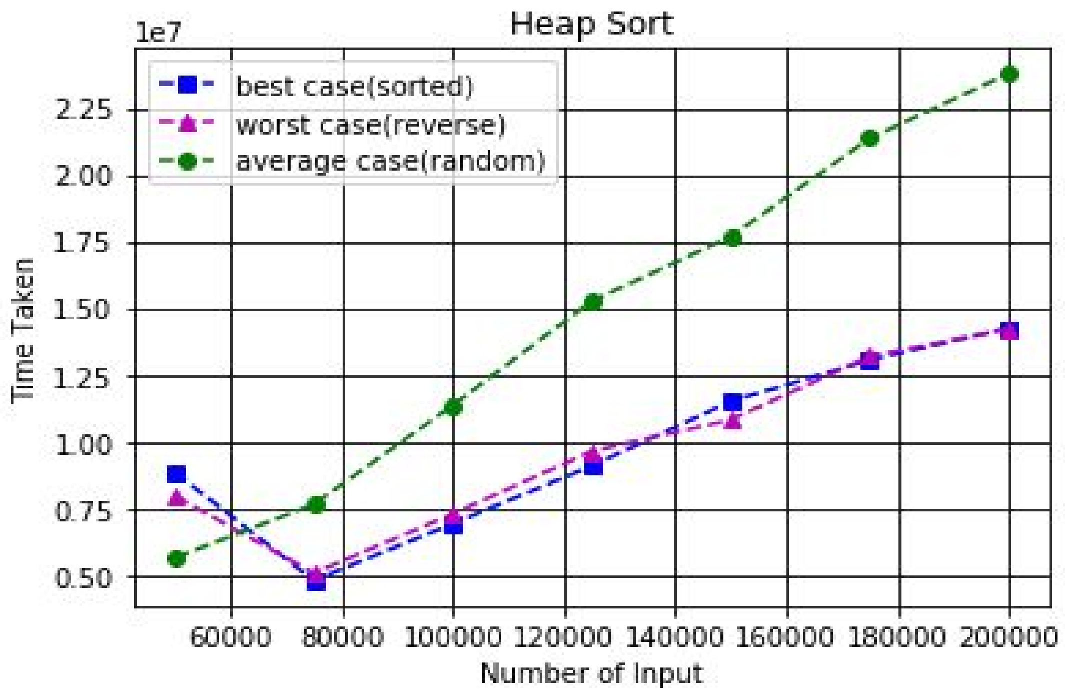
❏ It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for the remaining element.

Code:

- *Main Code Will remain same ...only sorting function needs to be changed*

```
public static void heap_sort(int arr[]){
       int n = arr.length;

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

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

              heapify(arr, i, 0);
       }
}

static void heapify(int arr[], int n, int i){
       int max= i;
       int l = 2*i + 1;
       int r = 2*i + 2;

       if (l < n && arr[l] > arr[max])
              max = l;

       if (r < n && arr[r] > arr[max])
              max = r;


       if (max != i)
       {
              int swap = arr[i];
              arr[i] = arr[max];
              arr[max] = swap;

              heapify(arr, n, max);
       }
}
```

Graph:



Analysis:

Time complexity of heapify is O(Logn). Time complexity of creating and Building Heap is O(n) and the overall time complexity of Heap Sort is O(nLogn).

Final Observation - Heap sort is very fast as Compared to bubble,insertion and selection sort.

Additional Information-

PYTHON CODE:

For plotting graph-

```python
import pandas as pd
import matplotlib.pyplot as mp

df=pd.read_csv("D:\\java file
handling\\insertion_sort_analysis.txt",sep="\t")

best=[]
worst=[]
avg=[]

for i in range(df.shape[0]):
    if i%3==0:
    best.append([df.iloc[i,1],df.iloc[i,2]])
    if i%3==1:
    worst.append([df.iloc[i,1],df.iloc[i,2]])
    if i%3==2:
    avg.append([df.iloc[i,1],df.iloc[i,2]])


mp.plot([i[0] for i in best],[i[1] for i in best],'bs--',label="best
case(sorted)")
mp.plot([i[0] for i in worst],[i[1] for i in
worst],'m^--',label="worst case(reverse)")
mp.plot([i[0] for i in avg],[i[1] for i in avg],'go--',label="average
case(random)")
mp.xlabel("Number of Input")
mp.ylabel("Time Taken")
mp.title("Insertion Sort")
mp.legend()
mp.grid(True,color='k')
```

SAMPLE FILE generated by Java Program:

Similarly for all sorting algorithms ,we can generate files for no. of input and time taken.

For Insertion sort:

```
      Number_of_Input    Time_Taken
 Best_Case    50000    2958500
 Worst_Case    50000    1617336900
 Avg_Case    50000    773831600
 Best_Case    75000    98700
 Worst_Case    75000    1090799600
 Avg_Case    75000    538286300
 Best_Case    100000    131500
 Worst_Case    100000    2001324200
 Avg_Case    100000    1046026100
 Best_Case    125000    173300
 Worst_Case    125000    3205471600
 Avg_Case    125000    1608988100
 Best_Case    150000    314600
 Worst_Case    150000    4670403100
 Avg_Case    150000    2661176400
 Best_Case    175000    293600
 Worst_Case    175000    7253538100
 Avg_Case    175000    3842901400
```