CHARLOTTE

Comparison based Sorting Algorithms

PROJECT REPORT

Submitted by:

Varad Deshpande 801243927

Table of Contents:

Problem Statement:

Given an array of random integers, analyze the different sorting algorithms for various input sizes. Observe and comment results and time complexity for each of the algorithms. Record the execution time for each algorithm and plot them all in a single graph to view consolidated results.

As part of the project, we will analyze the following algorithms: Insertion sort, Merge sort, Heap sort, Inplace Quick sort and Modified Quick sort.

We will analyze the behavior of these algorithms on following inputs:

1. Random sequence
2. Sorted sequence
3. Reverse sorted sequence

Algorithms and Analysis:

1. Insertion Sort:

Description:

- In Insertion sort, we virtually divide the array in two parts, the sorted array and the unsorted array.
- During each iteration, an element is taken from the unsorted array and placed into the sorted array. If the element is less than a few elements in the sorted array, they are moved by one place to the right to make room for the new integer.
- At the end of each iteration i, insertion sort ensures that we have a sorted array of ‘i’ integers along with an unsorted sequence.

Complexity Analysis:

- **Best case:** Insertion sort performs the best when the array is already sorted. Every element from the unsorted sequence will be placed at the end of the sorted sequence. Thus, the total time taken for sorting n integers would be **O(n)** in the best case.

- **Worst case:**The algorithm performs the worst for a sequence that is sorted in reverse order. In this case, every insertion in the sorted sequence with ‘i’ integers will be at the first position and the ‘i-1’ integers will have to be moved to make place for the new insertion. In the worst case, the time taken by the algorithm will be **$O(n^2)$**.
- **Average case:**On average, the time complexity for insertion sort is **$O(n^2)$**.
- Insertion sort performs well for smaller input sizes(<1K).

Data Structures Used:

- Array has been used in insertion sort to store integers. Sorting of data has been done on the same array.

Code:

```
public static void sort(int[] array, int min, int max){
   for(int i = min+1; i <= max; i++){
       int key = array[i];
       int j = i-1;
       while (j >= min && array[j] > key){
           array[j+1] = array[j];
           j--;
       }
       array[j+1] = key;
   }
}
```

2. Merge Sort:

Description:

- Merge sort is a divide and conquer algorithm.
- The algorithm divides the array into two halves recursively. This is the divide phase of the algorithm.

- Once we get to the smallest division possible, we combine the sub-arrays to get the solution.
- In the merge phase of the algorithm, we sort the elements from the two sub-arrays and combine them into one sorted array till we get the original sorted array.

Complexity Analysis:

- In merge sort, we divide the array into two halves recursively. The height **h** of a merge sort tree is **O(logn)**.
- At any given depth i, the total amount of work done is **O(n)**.
- Therefore, the total running time of the merge sort algorithm is **O(nlogn)**.
- Both the best and worst case time complexity of merge sort is **O(nlogn)**.
- This algorithm can be used to sort large datasets(>1M).

Data Structures Used:

- Integer arrays have been used in merge sort. The main array is recursively in left and right arrays and then combined to give the final sorted array.

Code:

```
public static void sort(int[] array, int min, int max){
   if(min < max){
        int mid = (min + max)/2;
        sort(array, min, mid);
        sort(array, mid+1, max);
        merge(array, min, mid, max);
   }
}
```

```java
private static void merge(int[] array, int min, int mid, int max){
    int[] leftArray = new int[mid - min + 1];
    int[] rightArray = new int[max - mid];
    int k = min;

    for(int i = 0; i < leftArray.length; i++){
        leftArray[i] = array[k++];
    }
    k = mid + 1;
    for(int i = 0; i < rightArray.length; i++){
        rightArray[i] = array[k++];
    }

    int i = 0;
    int j = 0;
    k = min;

    while(i < leftArray.length && j < rightArray.length){
        if(leftArray[i] < rightArray[j]){
            array[k++] = leftArray[i++];
        } else {
            array[k++] = rightArray[j++];
        }
    }
    while(i < leftArray.length)
        array[k++] = leftArray[i++];
    while(j < rightArray.length)
        array[k++] = rightArray[j++];
}
```

3. Heap Sort:

Description:

- Heap sort is based on heap to sort the sequence of data.
- Each element in the sequence is inserted in the min heap.
- As the heap used is min heap, we will have to satisfy the heap property, i.e, at every node in the heap, key(node) < key(children).

- After inserting all the elements in the heap from the sequence, we will remove the root, i.e., the minimum element and add it back in the sequence.
- Thus we get a sorted sequence of data using heap for sorting.

Complexity Analysis:

- Heap sort inserts and removes n items from the sequence to get the sorted list.
- Each insertion and deletion from the heap takes **O(logn)** time.
- Therefore the time taken for insertion and removal of ‘n’ elements from the heap will be **O(nlogn)**.
- Thus the time complexity of heap sort is **O(nlogn)**.

Data Structures Used:

- **Heap:** A min heap is used for sorting purposes in heap sort. We have implemented a vector-based heap with an array. As it is a min heap, it satisfies the property that at each node ‘k’, key(k) < key(children(k)).
    - In vector based implementation, for any item at index ‘i’, its left child is present at index ‘**2*i**’ and it right child will be present at ‘**(2*i)+1**’
    - When inserting an element in the heap with n elements, we will add the element at n+1 index and perform the heapify operation so that heap property is maintained(key of parent is less than children). We will perform swaps of the item in the heap if required to satisfy this property.
    - While removing an element, we will remove the minimum element which is present at the first index. We will then replace the first index with the last element and perform the heapify operation to maintain the heap property.
- Apart from heap, we use array to store the sequence of integers.

Code:

i) Heap:

```
public class Heap {
   int noOfElements;
   int[] array;

   public Heap(int[] array){
       this.array = new int[array.length + 1];
       this.noOfElements = 0;
   }

   public void insert(int element){
       noOfElements = ++noOfElements;
       array[noOfElements] = element;
       int i = noOfElements;
       while (i > 1 && array[i/2] > array[i]){
           int temp = array[i/2];
           array[i/2] = array[i];
           array[i] = temp;
           i = i/2;
       }
   }

   public int removeMin(){
       int temp = array[1];
       array[1] = array[noOfElements];
       noOfElements--;
       int i =1;
       while(i < noOfElements){
           if((2 * i) + 1 <= noOfElements){
                       if(array[i] <= array[2*i] && array[i] <=
array[(2*i)+1])
                   return temp;
               else{
                                   int min = Math.min(array[2*i],
array[(2*i)+1]);
                   int j = min == array[2*i]? (2*i) : (2*i)+1;
                   min = array[j];
                   array[j] = array[i];
```

```
                    array[i] = min;
                    i=j;
                }
            } else {
                if(2*i <= noOfElements){
                    if(array[i] > array[2*i]){
                        int min = array[i];
                        array[i] = array[2*i];
                        array[2*i] = min;
                    }
                }
                return temp;
            }
        }
        return temp;
    }
}
```

ii) Heap sort:

```
public static void sort(int[] array, int min, int max){
    heap = new Heap(array);
    for (int i = min; i <= max; i++) {
        heap.insert(array[i]);
    }
    for(int i = min; i <= max; i++) {
        array[i] = heap.removeMin();
    }
}
```

4. Inplace Quick Sort:

Description:

- Inplace quicksort uses the input array to do the sorting. Hence we require only one array in this algorithm to store a sequence of integers and hence the name.
- The pivot can be selected by any of these three methods:

    - Leftmost index
    - Rightmost index
    - Random index between 0 to n-1 (For size n)
- For getting the position of the pivot in the sequence, we perform an in -place partition with two pointers which are at min and max respectively.
- We in increment the leftmost pointer until we find an element which is greater than or equal to the pivot
- Similarly, we decrement the rightmost pointer until we find an element which is less than pivot.
- Once both the pointers stop, we perform a swap of the two elements shown by the pointer.
- We repeat this process till both pointers cross. We then swap the rightmost pointer with the pivot as it is the position of the pivot element.
- We recursively do this for the sub-array formed by the pivot till we get the minimum condition.
- Then we go on merging the two sub-arrays till we get the original array of size n.

Complexity Analysis:

- **Best Case:** The best case of the quicksort comes when the pivot divides the array into exactly two halves, i.e., during each split the array will be divided in half. In this case the depth of recursion will be **logn**. At each level of recursion the amount of work done is equal to the number of elements in the array, i.e., **n**. The best case complexity of inplace quicksort is thus **O(nlogn)**.
- **Worst Case:** The worst case of quicksort occurs when the pivot element divides the array of size **n** into one sub-array of size **n-1**,i.e.,the other sub-array is empty. As we dont recur the zero length part, the recurring on n-1 part requires recurring depth of n-1. The worst case time complexity comes out to be $\mathbf{O(n^2)}$.
- The worst case of quicksort occurs when the pivot is the minimum or the maximum element.

- In any average case, the time complexity of quicksort is **O(nlogn)**.

Data Structures Used:

- The input array is used in the algorithm to perform the sorting using divide and conquer paradigm.

Code:

```
public static void sort(int[] array, int min, int max){
   if(min < max){
       int pivotIndex = min;
       int pivot = array[pivotIndex];
          int index = inPlacePartition(array, pivot, min, max,
pivotIndex);
       sort(array, min, index-1);
       sort(array, index+1, max);
   }
}

private static int inPlacePartition(int[] array,int pivot, int
min, int max, int pivotIndex){
   int i = min+1;
   int j = max;
   while(i <= j){
       while(i<=j && array[i] <= pivot)
           i++;
       while(i<=j && array[j] > pivot)
           j--;
       if(i < j){
           int temp = array[i];
           array[i] = array[j];
           array[j] = temp;
       }
   }
   int temp = array[j];
   array[j] = pivot;
   array[pivotIndex] = temp;
   return j;
```

```
}
```

5. Modified Quick Sort:

Description:

- We make some modifications in the quicksort algorithm to make it perform better than quick sort.
- We select the pivot using the median of three method. The pivot is selected as follows:
    - We select the first element, last element and the middle element(first+last/2).
    - We swap and arrange these elements such that the rightmost element is the minimum of the three, leftmost is the maximum of the three and the middle element is the median of the three selected elements.
    - We select this middle element as the pivot. This pivot is moved to the second last element of the sequence, i.e., left of the highest element.
    - This pivot selection is recursively called.
- Also for arrays whose length is less than or equal to 10, we use insertion sort instead of quick sort. The reason for this is that insertion sort performs better on smaller arrays and hence the efficiency of the algorithm increases.

Complexity Analysis:

- **Best case:** The best case will occur when the array is divided into exactly two halves. The time complexity in this case will be **O(nlogn)**.
- **Worst case:** The worst case complexity of quicksort is **$O(n^2)$**.
- Due to the modifications implemented in quicksort, the algorithm tends to perform optimally most of the time with complexity of **O(nlogn)**.

Data Structures Used:

- In this algorithm, we use arrays to store the sequence of integers and perform sorting on the same.

Code:

```
public static void sort(int[] array, int min, int max){
   if(min + 10 <= max){
       int pivot = medianOfThree(array, min, max);
       int temp;
       int i = min+1;
       int j = max-2;
       while(i<=j){
           while(i<=j && array[i] <= pivot)
               i++;
           while(i<=j && array[j] > pivot)
               j--;
           if(i<j){
               temp = array[i];
               array[i] = array[j];
               array[j] = temp;
           }
       }
       temp = array[i];
       array[i] = array[max-1];
       array[max-1] = temp;
       sort(array, min, i-1);
       sort(array, i+1, max);
   }else{
       InsertionSort.sort(array, min, max);
   }
}

private static int medianOfThree(int[] array, int min, int max){
   int temp;
   int mid = (min + max) / 2;
   if (array[mid] < array[min]){
       temp = array[min];
       array[min] = array[mid];
```

```
        array[mid] = temp;
    }
    if (array[max] < array[min]){
        temp = array[min];
        array[min] = array[max];
        array[max] = temp;
    }
    if (array[max] < array[mid]){
        temp = array[mid];
        array[mid] = array[max];
        array[max] = temp;
    }
    temp = array[max-1];
    array[max-1] = array[mid];
    array[mid] = temp;
    return array[max-1];
}
```

Code Utilities:

RandomNumberGenerator:

- **generateRandomNumbers:** Function to generate random numbers till 100000
- **generateSortedRandomNumbers:** Function to generate random numbers in sorted manner
- **generateReverseSortedRandomNumbers:** Function to generate random array in reverse sorted order.

```
public class RandomNumberGenerator {
    public static int[] generateRandomNumbers(int size){
        return new Random().ints(0,
100000).limit(size).toArray();
    }

    public static int[] generateSortedRandomNumbers(int size){
        return new Random().ints(0,
100000).limit(size).sorted().toArray();
    }
```

```
    public static int[] generateReverseSortedRandomNumbers(int
size){
        Integer[] randomList = new Random().ints(0,
100000).limit(size).boxed().toArray(Integer[]::new);
        Arrays.sort(randomList, Collections.reverseOrder());
        return
Arrays.stream(randomList).mapToInt(Integer::intValue).toArray();
    }
}
```

LineChart:

- Java class used to generate line charts with a given input data set.

```
public LineChart(String applicationTitle , String chartTitle,
DefaultCategoryDataset dataset ) {
    super(applicationTitle);
    JFreeChart lineChart = ChartFactory.createLineChart(
            chartTitle,
            "Data Size","Time in milliseconds",
            dataset,
            PlotOrientation.VERTICAL,
            true,true,false);

    ChartPanel chartPanel = new ChartPanel( lineChart );
    chartPanel.setPreferredSize( new java.awt.Dimension( 720 ,
510 ) );
    setContentPane( chartPanel );
}
```

Results and Analysis:

Random Array:

Output:

```
java -Xmx512m -Xss512m -jar target/Project1-1.0-SNAPSHOT-jar-with-dependencies.jar
Please enter the Array type you want for the run:
1. Sorted Array
2. Reverse Sorted Array
3. Random Array
Enter your choice:
3
```

Size of data: 10000
Selected Random Array
Average Execution Times for the run:
Execution Time for Insertion sort: 24ms
Execution Time for Merge sort: 2ms
Execution Time for Heap sort: 2ms
Execution Time for In-place Quick sort: 1ms
Execution Time for Modified Quick sort: 1ms

Size of data: 20000
Selected Random Array
Average Execution Times for the run:
Execution Time for Insertion sort: 31ms
Execution Time for Merge sort: 2ms
Execution Time for Heap sort: 1ms
Execution Time for In-place Quick sort: 1ms
Execution Time for Modified Quick sort: 1ms

Size of data: 25000
Selected Random Array
Average Execution Times for the run:
Execution Time for Insertion sort: 49ms
Execution Time for Merge sort: 4ms
Execution Time for Heap sort: 2ms
Execution Time for In-place Quick sort: 1ms
Execution Time for Modified Quick sort: 1ms

Size of data: 50000
Selected Random Array
Average Execution Times for the run:
Execution Time for Insertion sort: 197ms
Execution Time for Merge sort: 7ms
Execution Time for Heap sort: 4ms
Execution Time for In-place Quick sort: 3ms
Execution Time for Modified Quick sort: 3ms

Size of data: 75000
Selected Random Array
Average Execution Times for the run:
Execution Time for Insertion sort: 446ms
Execution Time for Merge sort: 10ms
Execution Time for Heap sort: 7ms
Execution Time for In-place Quick sort: 5ms
Execution Time for Modified Quick sort: 5ms

Size of data: 100000
Selected Random Array
Average Execution Times for the run:
Execution Time for Insertion sort: 791ms
Execution Time for Merge sort: 14ms
Execution Time for Heap sort: 10ms
Execution Time for In-place Quick sort: 6ms
Execution Time for Modified Quick sort: 7ms

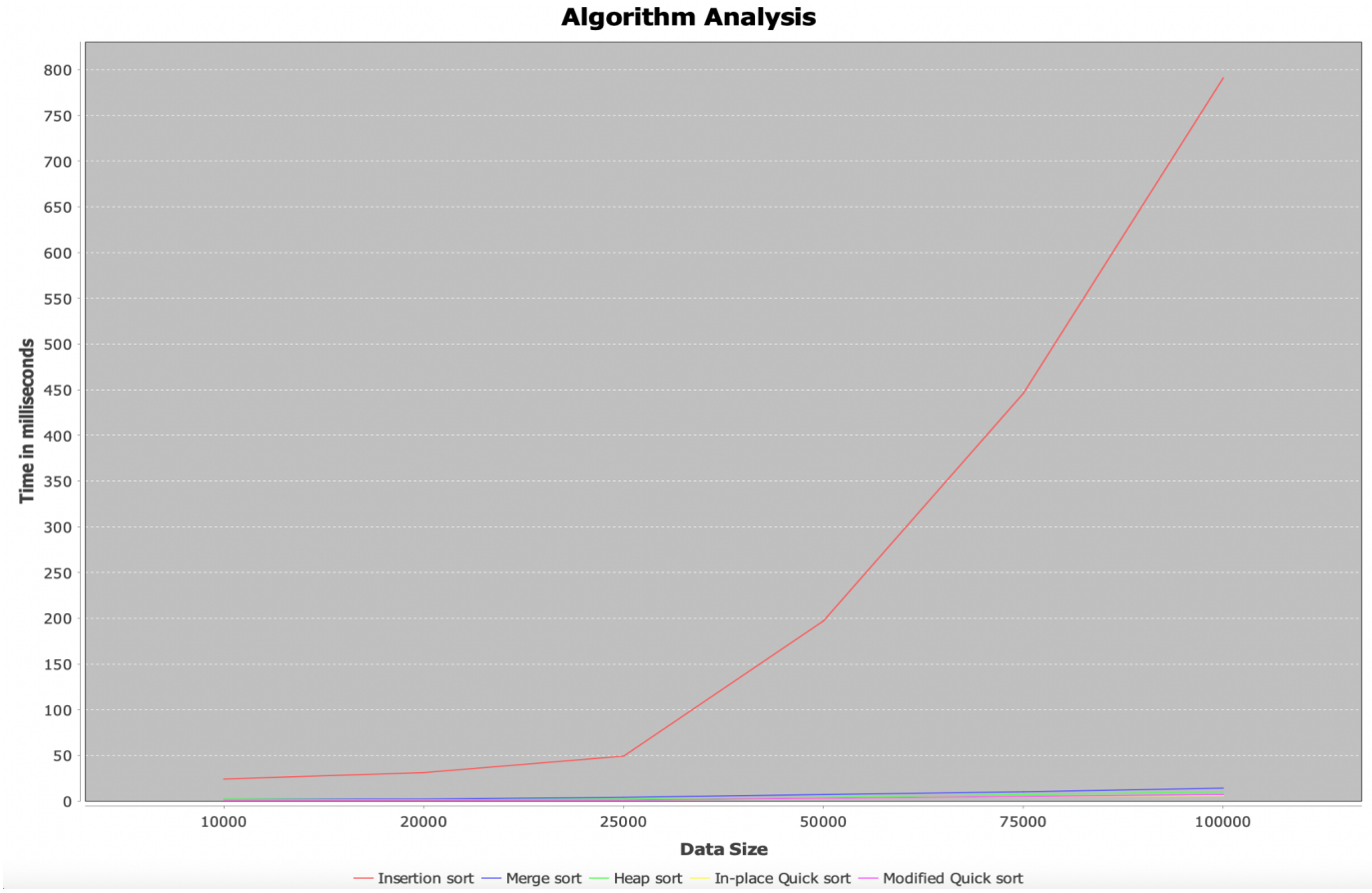
Output Times per input and algorithm in milliseconds:

Data size Algorithm	**10000**	**20000**	**25000**	**50000**	**75000**	**100000**
Insertion sort	24	31	49	197	446	791
Merge sort	2	2	4	7	10	14
Heap sort	2	1	2	4	7	10
Inplace Quick sort	1	1	1	3	5	6
Modified Quick sort	1	1	1	3	5	7

Analysis:

- With the time complexity of $O(n^2)$, we can clearly see the performance of insertion sort to be the worst for the selected data sizes.
- The remaining algorithms(Merge sort, heap sort, inplace quick sort, modified quick sort) perform quite efficiently with the time complexity of O(nlogn). Hence we do not see significant increase in the execution times of these four algorithms.

Observation Graph:



Sorted Array:

Output:

java -Xmx512m -Xss512m -jar target/Project1-1.0-SNAPSHOT-jar-with-dependencies.jar
Please enter the Array type you want for the run:
1. Sorted Array
2. Reverse Sorted Array
3. Random Array
Enter your choice:
1

Size of data: 10000
Selected Random Sorted Array
Average Execution Times for the run:
Execution Time for Insertion sort: 0ms
Execution Time for Merge sort: 1ms
Execution Time for Heap sort: 1ms
Execution Time for In-place Quick sort: 26ms

Execution Time for Modified Quick sort: 2ms

Size of data: 20000
Selected Random Sorted Array
Average Execution Times for the run:
Execution Time for Insertion sort: 0ms
Execution Time for Merge sort: 1ms
Execution Time for Heap sort: 2ms
Execution Time for In-place Quick sort: 55ms
Execution Time for Modified Quick sort: 0ms

Size of data: 25000
Selected Random Sorted Array
Average Execution Times for the run:
Execution Time for Insertion sort: 0ms
Execution Time for Merge sort: 1ms
Execution Time for Heap sort: 3ms
Execution Time for In-place Quick sort: 86ms
Execution Time for Modified Quick sort: 0ms

Size of data: 50000
Selected Random Sorted Array
Average Execution Times for the run:
Execution Time for Insertion sort: 0ms
Execution Time for Merge sort: 4ms
Execution Time for Heap sort: 3ms
Execution Time for In-place Quick sort: 383ms
Execution Time for Modified Quick sort: 1ms

Size of data: 75000
Selected Random Sorted Array
Average Execution Times for the run:
Execution Time for Insertion sort: 0ms
Execution Time for Merge sort: 5ms
Execution Time for Heap sort: 4ms
Execution Time for In-place Quick sort: 880ms
Execution Time for Modified Quick sort: 0ms

Size of data: 100000
Selected Random Sorted Array
Average Execution Times for the run:
Execution Time for Insertion sort: 0ms
Execution Time for Merge sort: 8ms
Execution Time for Heap sort: 6ms
Execution Time for In-place Quick sort: 1560ms
Execution Time for Modified Quick sort: 1ms

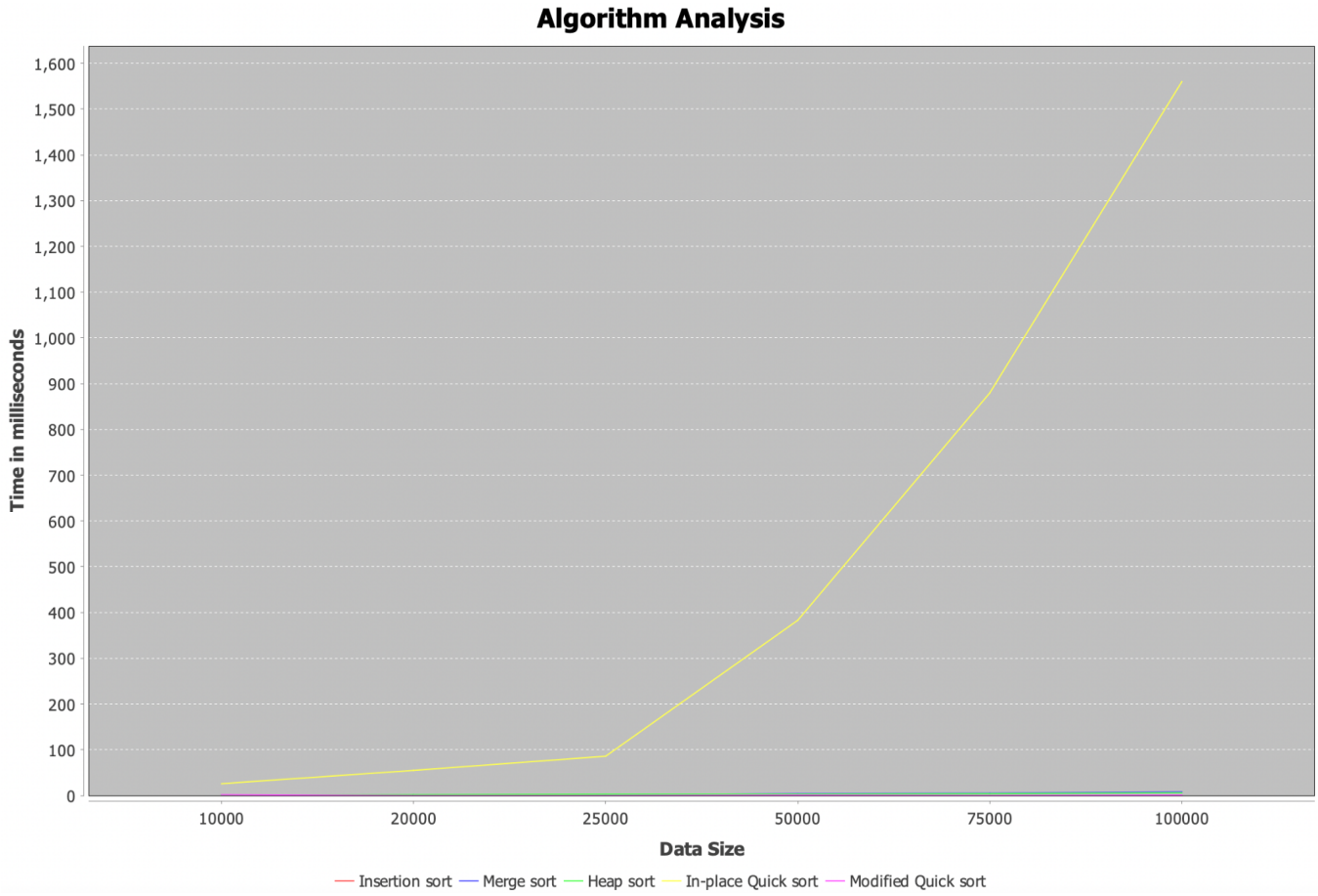
Output Times per input and algorithm in milliseconds:

Data size Algorithm	**10000**	**20000**	**25000**	**50000**	**75000**	**100000**
Insertion sort	0	0	0	0	0	0
Merge sort	1	1	1	4	5	8
Heap sort	1	2	3	3	4	6
Inplace Quick sort	26	55	86	383	880	1560
Modified Quick sort	2	0	0	1	0	1

Analysis:

- Sorted array is the best case scenario for insertion sort with time complexity of O(n). From the observation, we can see that the insertion sort is the best performing algorithm.
- As sorted sequences are the worst case for inplace quicksort as its time complexity becomes $O(n^2)$. (We have use leftmost element as the pivot in our implementation)
- Due to the median of three implementation and insertion sort invocation for small sequences, modified quicksort tends to perform like an average use case and avoids the worst case.
- The time complexity for merge sort and heap sort remain to be O(nlogn) and thus perform optimally even for sorted sequence.
- On larger inputs for sorted sequences, inplace quicksort can also give a stack overflow exception due to the number of recursion calls in the code. This depends on factors such as machine configuration as well as the JVM configuration for that machine.

Observation Graph:



Reverse Sorted Array:

Output:

java -Xmx512m -Xss512m -jar target/Project1-1.0-SNAPSHOT-jar-with-dependencies.jar
Please enter the Array type you want for the run:
1. Sorted Array
2. Reverse Sorted Array
3. Random Array
Enter your choice:
2

Size of data: 10000
Selected Random Reverse Sorted Array
Average Execution Times for the run:
Execution Time for Insertion sort: 39ms
Execution Time for Merge sort: 1ms

Execution Time for Heap sort: 1ms
Execution Time for In-place Quick sort: 23ms
Execution Time for Modified Quick sort: 2ms

Size of data: 20000
Selected Random Reverse Sorted Array
Average Execution Times for the run:
Execution Time for Insertion sort: 63ms
Execution Time for Merge sort: 1ms
Execution Time for Heap sort: 2ms
Execution Time for In-place Quick sort: 54ms
Execution Time for Modified Quick sort: 0ms

Size of data: 25000
Selected Random Reverse Sorted Array
Average Execution Times for the run:
Execution Time for Insertion sort: 99ms
Execution Time for Merge sort: 2ms
Execution Time for Heap sort: 2ms
Execution Time for In-place Quick sort: 85ms
Execution Time for Modified Quick sort: 0ms

Size of data: 50000
Selected Random Reverse Sorted Array
Average Execution Times for the run:
Execution Time for Insertion sort: 397ms
Execution Time for Merge sort: 3ms
Execution Time for Heap sort: 5ms
Execution Time for In-place Quick sort: 338ms
Execution Time for Modified Quick sort: 0ms

Size of data: 75000
Selected Random Reverse Sorted Array
Average Execution Times for the run:
Execution Time for Insertion sort: 887ms
Execution Time for Merge sort: 3ms
Execution Time for Heap sort: 7ms
Execution Time for In-place Quick sort: 761ms
Execution Time for Modified Quick sort: 1ms

Size of data: 100000
Selected Random Reverse Sorted Array
Average Execution Times for the run:
Execution Time for Insertion sort: 1580ms
Execution Time for Merge sort: 4ms
Execution Time for Heap sort: 28ms
Execution Time for In-place Quick sort: 1174ms

Execution Time for Modified Quick sort: 1ms

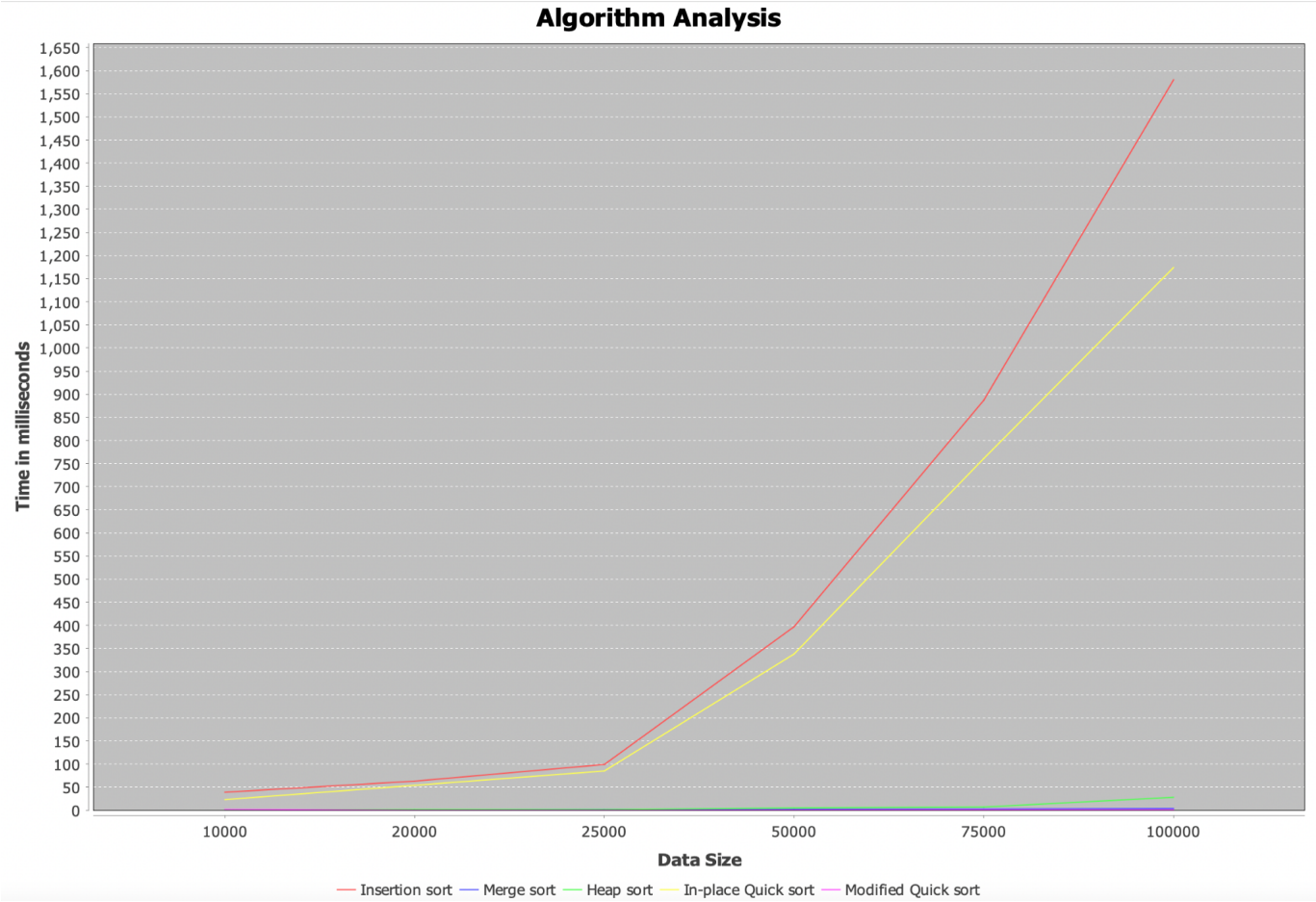
Output Times per input and algorithm in milliseconds:

Data size Algorithm	**10000**	**20000**	**25000**	**50000**	**75000**	**100000**
Insertion sort	39	63	99	397	887	1580
Merge sort	1	1	2	3	3	4
Heap sort	1	2	2	5	7	28
Inplace Quick sort	23	54	85	338	761	1174
Modified Quick sort	2	0	0	0	1	1

Analysis:

- Reverse sorted array is the worst case scenario for insertion sort with time complexity of $O(n^2)$. Thus the algorithm is the worst performing among the sorting algorithm for the reverse sorted array.
- The performance of inplace quicksort is also $O(n^2)$ as for any sequence n, one partition will have n-1 elements and the other partition will have no elements at all. This is also the worst case for quicksort
- Modified quicksort performs better than quicksort due to the modification which are in place which help in improved performance than the inplace quick sort
- Merge sort and heap sort perform with the time complexity of O(nlogn) and hence we do not see much difference in the execution times even for the high inputs.

Observation Graph:



Conclusion:

Thus we have successfully implemented the sorting algorithms and analyzed their time complexities and observed the same on sample data sets. We have also checked the performance of these algorithms on special cases such as sorted and reverse sorted inputs and analyzed the same as well.