```
package proj;



import java.util.Random;

public class project1 {

    static Heap heap = new Heap(10000);
    static int[] quick = new int[10000];
    static int[] insertion = new int[10000];
    static int[] merge = new int[10000];
    static int[] backup = new int[10000];
    static int[] helper = new int[merge.length]; //for the mergeSort
method

    //Some counters for time/space complexity analysis
    static long count=0; //Temporary arrays for mergesort
    static long swaps=0; //swaps for insertionSort
    static long swapsinMerge=0; //Swaps in mergesort method
    static long quicksortSwaps=0;

    public static void main(String[] args) {
        Random rand = new Random();
        for(int i=0; i<10000; i++) {
            quick[i]=rand.nextInt(1000000);
        }
        for(int i=0; i<10000; i++) {
            merge[i]=quick[i];
        }
        for(int i=0; i<10000; i++) {
            backup[i]=quick[i];
        }
        for(int i=0; i<10000; i++) {
            insertion[i]=quick[i];
        }
        for(int i=0; i<10000; i++) {
            heap.insert(quick[i], "max");
        }




        //MergeSort test time
        long time=0;
        swapsinMerge=0;
        for(int i=0; i<1000; i++) {
            long startTime = System.currentTimeMillis();
            MergeSort(merge, 0, merge.length-1);
            long endTime = System.currentTimeMillis();
            time= time+(endTime - startTime);
```

```
            for(int j=0; j<10000; j++) {   //Reset everytime we do a new
test
                merge[j]=backup[j];
            }
        }
        double avgTime= (double)time/1000;
        System.out.println("Time Taken for mergesort = " + (avgTime)+ "
miliseconds");
        System.out.println("Number of swaps = "+ (long)
(swapsinMerge/1000));
        System.out.println("Temporary arrays created = "+ (long)
(count/1000)+"\n");


        //QuickSort test time
        quicksortSwaps=0;
        time=0;
        for(int i=0; i<1000; i++) {
            long startTime = System.currentTimeMillis();
            QuickSort(quick, 0, quick.length-1);
            long endTime = System.currentTimeMillis();
            time= time+(endTime - startTime);
            for(int j=0; j<10000; j++) {  //Reset everytime we do a new
test
                quick[j]=backup[j];
            }
        }

        avgTime= (double)time/1000;
        System.out.println("Time Taken for quicksort = " + (avgTime)+ "
miliseconds");
        System.out.println("Number of swaps = "+ (long)
(quicksortSwaps/1000)+"\n");


        //HeapSort test time
                time=0;
                long heapSwaps=0;
                for(int i=0; i<1000; i++) {
                    long startTime = System.currentTimeMillis();
                    heap.sort("max");
                    long endTime = System.currentTimeMillis();
                    time= time+(endTime - startTime);
                    heapSwaps= heapSwaps+heap.swaps;
                    heap.deleteTree();//Reset everytime we do a new test
                    heap = new Heap(10000);
                    for(int j=0; j<10000; j++) {
                        heap.insert(backup[j], "max");
                    }
                }
                avgTime= (double)time/1000;
                System.out.println("Time Taken for HeapSort = " +
(avgTime)+ " miliseconds");
```

```
                System.out.println("Number of swaps = "+ (long)
(heapSwaps/1000)+"\n");



        //InsertionSort test time
        time=0;
        swaps=0;
        for(int i=0; i<1000; i++) {
            long startTime = System.currentTimeMillis();
            insertionSort(insertion);
            long endTime = System.currentTimeMillis();
            time= time+(endTime - startTime);

            for(int j=0; j<10000; j++) {  //Reset everytime we do a new
test
                insertion[j]=backup[j];
            }
        }
        avgTime= (double)time/1000;
        System.out.println("Time Taken for insertionSort = " + (avgTime)+
" miliseconds");
        System.out.println("Number of swaps = "+ (long)(swaps/1000)+"\n");



        //  QuickSort on a sorted array for one run

        int[] tempArr = new int[10000];
        for(int i=0; i<10000; i++) {
            tempArr[i]= new Random().nextInt(1000000);
        }

        //lets sort the array first
        QuickSort(tempArr, 0, tempArr.length-1);


        //sorted array
        long startTime2 = System.currentTimeMillis();
        QuickSort(tempArr, 0, tempArr.length-1);
        long endTime2 = System.currentTimeMillis();
        System.out.println("Time taken for QuickSort in sorted array = "+
(endTime2-startTime2)+ " milliseconds");

//      MergeSort on a sorted array for one run

            int[] tempArr2 = new int[10000];
            for(int i=0; i<10000; i++) {
                tempArr[i]= new Random().nextInt(1000000);
            }

            //lets sort the array first
            MergeSort(tempArr2, 0, tempArr2.length-1);
```

```
            //sorted array
            startTime2 = System.currentTimeMillis();
            MergeSort(tempArr2, 0, tempArr2.length-1);
            endTime2 = System.currentTimeMillis();
            System.out.println("Time taken for MergeSort in sorted array =
"+ (endTime2-startTime2)+ " milliseconds");

            //Part 2

            int[] smallArr;
            int[] backupArr;

            for(int i=16; i<=256; i=i*2) {
                smallArr= new int[i];
                backupArr= new int[i];
                heap = new Heap(i);
                for(int j=0; j<i; j++) {
                    smallArr[j]= new Random().nextInt(1000);
                    backupArr[j]=smallArr[j];
                    heap.insert(smallArr[j], "max");
                }
                System.out.println("Array of size "+ "["+i+"]:");
                System.out.println("--------------------------");
                //MergeSort
                time=0;
                swapsinMerge=0;
                for(int k=0; k<1000; k++) {
                    long startTime = System.nanoTime();
                    MergeSort(smallArr, 0, smallArr.length-1);
                    long endTime = System.nanoTime();
                    time= time+(endTime - startTime);

                    for(int j=0; j<i; j++) {  //Reset everytime we do a
new test
                        smallArr[j]=backupArr[j];
                    }
                }
                avgTime= (double)time/1000;
                System.out.println("Time Taken for MergeSort = " +
(avgTime)+ " nanoseconds");
                System.out.println("Number of swaps = "+ (long)
(swapsinMerge/1000)+"\n");

                //HeapSort
                time=0;
                heapSwaps=0;
                heap.swaps=0;
                for(int k=0; k<1000; k++) {
                    long startTime = System.nanoTime();
                    heap.sort("max");
                    long endTime = System.nanoTime();
                    time= time+(endTime - startTime);
                    heapSwaps= heapSwaps+heap.swaps;
```

```
                    heap.deleteTree();//Reset everytime we do a new test
                    heap = new Heap(i);
                    for(int j=0; j<i; j++) {
                        heap.insert(backupArr[j], "max");
                    }
                }
                avgTime= (double)time/1000;
                System.out.println("Time Taken for HeapSort = " +
(avgTime)+ " nanoseconds");
                System.out.println("Number of swaps = "+ (long)
(heapSwaps/1000)+"\n");
//
                //QuickSort
//              quicksortSwaps=0;
//              time=0;
//              for(int k=0; k<1000; k++) {
//                  long startTime = System.nanoTime();
//                  QuickSort(smallArr, 0, smallArr.length-1);
//                  long endTime = System.nanoTime();
//                  time= time+(endTime - startTime);
//                  for(int j=0; j<i; j++) {  //Reset everytime we do a
new test
//                      smallArr[j]=backupArr[j];
//                  }
//              }
//
//              avgTime= (double)time/1000;
//              System.out.println("Time Taken for quicksort = " +
(avgTime)+ " nanoseconds");
//              System.out.println("Number of swaps = "+ (long)
(quicksortSwaps/1000)+"\n");
//
                //insertionSort
                time=0;
                swaps=0;
                for(int k=0; k<1000; k++) {
                    long startTime = System.nanoTime();
                    insertionSort(smallArr);
                    long endTime = System.nanoTime();
                    time= time+(endTime - startTime);

                    for(int j=0; j<i; j++) {  //Reset everytime we do a
new test
                        smallArr[j]=backupArr[j];
                    }
                }
                avgTime= (double)time/1000;
                System.out.println("Time Taken for insertionSort = " +
(avgTime)+ " nanoseconds");
                System.out.println("Number of swaps = "+ (long)
(swaps/1000)+"\n");

            }
```

```
    }

      public static void insertionSort(int arr[]) {
            for(int i=1; i<arr.length; i++) {
                int j=i;
                while((j>0)&& (arr[j-1]>arr[j])) {
                    swaps++;
                    int temp= arr[j];
                    arr[j]=arr[j-1];
                    arr[j-1]=temp;
                    j--;
                }

            }
        }


    public static void MergeSort(int[] arr, int first, int last) {
        if(first<last) {
            int mid = (first+last)/2;
            MergeSort(arr, first, mid);
            MergeSort(arr, mid+1, last);
            Merge(arr, first, mid, last);
            }
    }



    public static void Merge(int[] merge, int first, int mid, int last) {
        count++;  //Counts the number of temporary arrays we make
        for(int i=first; i<=last; i++) {
            helper[i]=merge[i];
        }

        int i=first;
        int j= mid+1;
        int k=first;

        while(i<=mid && j<=last) {
            if(helper[i]<=helper[j]) {
                swapsinMerge++;
                merge[k]=helper[i];
                i++;
            }
            else {
                swapsinMerge++;
                merge[k]=helper[j];
                j++;
            }
            k++;
        }

        while(i<=mid) {
```

```
                swapsinMerge++;
                merge[k]=helper[i];
                i++;
                k++;
            }

            while(j<=last) {
                swapsinMerge++;
                merge[k]=helper[j];
                j++;
                k++;
            }
        }


        public static void QuickSort(int[] quick, int first, int last) {
            if(first<last) {
                int pivot = partition(quick,first, last);
                QuickSort(quick,first, pivot-1);
                QuickSort(quick,pivot+1, last);
            }
        }



          private static int partition(int[] quick,int first, int last) {
                int piv= quick[last];
                int i=first;
                int j= last-1;
                while(i<=j) {
                    while(quick[i]<piv && i<last) {
                        i++;
                    }
                    while(quick[j]>=piv && j>=first) {
                        j--;
                    }
                    if(i<j) {
                        quicksortSwaps++;
                        int temp= quick[i];
                        quick[i]=quick[j];
                        quick[j]= temp;
                    }
                }
                int temp= quick[i];
                quick[i]=piv;
                quick[last]= temp;
                return i;
        }

    }

    class Heap {
        int[] arr;
        int size;
```

```
    int swaps=0;
    public Heap(int size) {
        arr= new int[size+1];
        this.size=0;
    }

    public boolean isEmpty() {
        if(size==0) {
            return true;
        }
        else {
            return true;
        }
    }

    public String toString() {
        String ar= "";
        for(int i=1; i<=size; i++) {
            ar= ar+ arr[i]+" ";
        }
        return ar;
    }

    public void sort(String type) {
        for(int i=1; i<arr.length; i++) {
            delete(type);
        }
    }

    public void deleteTree() {
        arr= null;
        size=0;
    }
    public void heapify(int index, String type) {
        int parent = index/2;
        if(index==1) {
            return;
        }
        else {
            if(type =="max") {
                if(arr[parent]<arr[index]) {
                    int temp= arr[parent];
                    arr[parent]= arr[index];
                    arr[index]= temp;
                }
            }
            else if(type =="min") {
                if(arr[parent]>arr[index]) {
                    int temp= arr[parent];
                    arr[parent]= arr[index];
                    arr[index]= temp;
                }
            }
            heapify(parent, type);
```

```
        }
    }

    public void delete(String type) {
        int temp= arr[size];
        arr[size]= arr[1];
        arr[1]= temp;
        size--;
        heapifydown(1, type);
    }

    public void heapifydown(int index, String type) {
        int left= index*2;
        int right= (index*2)+1;
        int swap;
        if(size<left) {
            return;
        }
        if(type=="min") {
            if(size==left) {
                if(arr[left]<arr[index]) {
                    swaps++;
                    int temp= arr[left];
                    arr[left]= arr[index];
                    arr[index] = temp;
                }
                return;
            }
            else {
                if(arr[left]>arr[right]) {
                    swap= right;
                }
                else {
                    swap= left;
                }

                if(arr[index]>arr[swap]) {
                    swaps++;
                    int temp= arr[swap];
                    arr[swap]= arr[index];
                    arr[index] = temp;
                }
            }
            heapifydown(swap, type);
        }
        else if(type=="max") {
            if(size==left) {
                if(arr[left]>arr[index]) {
                    swaps++;
                    int temp= arr[left];
                    arr[left]= arr[index];
                    arr[index] = temp;
                }
                return;
```

```
            }
            else {
                if(arr[left]<arr[right]) {
                    swap= right;
                }
                else {
                    swap= left;
                }

                if(arr[index]<arr[swap]) {
                    swaps++;
                    int temp= arr[swap];
                    arr[swap]= arr[index];
                    arr[index] = temp;
                }
            }
            heapifydown(swap, type);
        }
    }

    public void insert(int val, String type) {
        size++;
        arr[size]=val;
        heapify(size, type);
    }
}
```