```
 * Zane Wonsey
package edu.emich.cosc;

import java.io.BufferedWriter;

public class Driver {

    private static int ascend[] = new int[10000];
    private static int descend[] = new int[10000];
    private static int random[] = new int[10000];

    public static void main(String[] args) {

        loadDataSet();
        sortDataSet();

    }

    public static void loadDataSet() {

        try {

            Scanner a = new Scanner(new File("./dataAscend.txt"));
            Scanner b = new Scanner(new File("./dataDescend.txt"));
            Scanner c = new Scanner(new File("./dataRandom.txt"));

            int count = 0;
            while (count < 10000) {
                ascend[count] = Integer.parseInt(a.nextLine());
                count++;
            }

            count = 0;
            while (count < 10000) {
                descend[count] = Integer.parseInt(b.nextLine());
                count++;
            }

            count = 0;
            while (count < 10000) {
                random[count] = Integer.parseInt(c.nextLine());
                count++;
```

```
            }

        } catch (FileNotFoundException e) {
            System.err.println("File was not Found");
        }

    }

    public static void sortDataSet() {
        merge();
        quick();
        heap();
    }

    private static void merge() {
        System.out.println("Ascending Merge sort");
        MergeSort a = new MergeSort(ascend);
        long startTime = System.nanoTime();
        int sortedMerge[] = a.mergeSort();
        long timeElapsed = System.nanoTime() - startTime;
        System.out.println("Time elapsed for this sort: " + timeElapsed + "
nanoseconds");
        output(sortedMerge);

        System.out.println("Descending Merge sort");
        MergeSort b = new MergeSort(descend);
        startTime = System.nanoTime();
        sortedMerge = b.mergeSort();
        timeElapsed = System.nanoTime() - startTime;
        System.out.println("Time elapsed for this sort: " + timeElapsed + "
nanoseconds");
        output(sortedMerge);

        System.out.println("Random Merge sort");
        MergeSort c = new MergeSort(random);
        startTime = System.nanoTime();
        sortedMerge = c.mergeSort();
        timeElapsed = System.nanoTime() - startTime;
        System.out.println("Time elapsed for this sort: " + timeElapsed + "
nanoseconds");
        output(sortedMerge);
    }
```

```
    private static void quick() {
        System.out.println("Ascending Quick sort");
        QuickSort a = new QuickSort(ascend);
        long startTime = System.nanoTime();
        int sortedQuick[] = a.quickSort();
        long timeElapsed = System.nanoTime() - startTime;
        System.out.println("Time elapsed for this sort: " + timeElapsed + "
nanoseconds");
        output(sortedQuick);

        System.out.println("Descending Quick sort");
        QuickSort b = new QuickSort(descend);
        startTime = System.nanoTime();
        sortedQuick = b.quickSort();
        timeElapsed = System.nanoTime() - startTime;
        System.out.println("Time elapsed for this sort: " + timeElapsed + "
nanoseconds");
        output(sortedQuick);

        System.out.println("Random Quick sort");
        QuickSort c = new QuickSort(random);
        startTime = System.nanoTime();
        sortedQuick = c.quickSort();
        timeElapsed = System.nanoTime() - startTime;
        System.out.println("Time elapsed for this sort: " + timeElapsed + "
nanoseconds");
        output(sortedQuick);
    }

    private static void heap() {
        System.out.println("Ascending Heap sort");
        HeapSort a = new HeapSort(ascend);
        long startTime = System.nanoTime();
        int sortedHeap[] = a.heapSort();
        long timeElapsed = System.nanoTime() - startTime;
        System.out.println("Time elapsed for this sort: " + timeElapsed + "
nanoseconds");
        output(sortedHeap);

        System.out.println("Descending Heap sort");
        HeapSort b = new HeapSort(descend);
```

```
        startTime = System.nanoTime();
        sortedHeap = b.heapSort();
        timeElapsed = System.nanoTime() - startTime;
        System.out.println("Time elapsed for this sort: " + timeElapsed + "
nanoseconds");
        output(sortedHeap);

        System.out.println("Random Heap sort");
        HeapSort c = new HeapSort(random);
        startTime = System.nanoTime();
        sortedHeap = c.heapSort();
        timeElapsed = System.nanoTime() - startTime;
        System.out.println("Time elapsed for this sort: " + timeElapsed + "
nanoseconds");
        output(sortedHeap);
    }

    public static void output(int[] sortedArray) {
        Scanner kb = new Scanner(System.in);
        String outputFile;

        System.out.println("Please enter the name of the file you would like
to use for output:");
        outputFile = kb.next();

        try {
            BufferedWriter outputWriter = null;
            outputWriter = new BufferedWriter(new FileWriter(outputFile));
            for (int i = 0; i < sortedArray.length; i++) {
                outputWriter.write(Integer.toString(sortedArray[i]));
                outputWriter.newLine();
            }
            outputWriter.flush();
            outputWriter.close();
        } catch (IOException e) {
            System.err.println("Could not output to file");
        }

    }

}
```

```
package edu.emich.cosc;

public class MergeSort {

    private int[] theArray;

    public MergeSort(int[] array) {
        theArray = array;
    }

    public int[] mergeSort() {
        int[] w = new int[10000];
        recMergeSort(w, 0, 9999);
        return theArray;
    }

    private void recMergeSort(int[] w, int LB, int UB) {
        if (LB == UB) {
            return;
        } else {
            int mid = (LB+UB)/2;
            recMergeSort(w, LB, mid);
            recMergeSort(w, mid+1, UB);
            merge(w, LB, mid+1, UB);
        }
    }

    private void merge(int[] w, int lowPtr, int highPtr, int UB) {
        int j = 0;
        int LB = lowPtr;
        int mid = highPtr-1;
        int n = UB-LB+1;

        while (lowPtr <= mid && highPtr <= UB) {
            if (theArray[lowPtr] < theArray[highPtr]) {
                w[j++] = theArray[lowPtr++];
            } else {
                w[j++] = theArray[highPtr++];
            }
        }

        while (lowPtr <= mid) {
```

```
            w[j++] = theArray[lowPtr++];
        }

        while (highPtr <= UB) {
            w[j++] = theArray[highPtr++];
        }

        for (j = 0; j < n; j++) {
            theArray[LB+j] = w[j];
        }
    }

}
```

```
package edu.emich.cosc;

public class QuickSort {

    private int theArray[];

    public QuickSort(int[] array) {
        theArray = array;
    }

    public int[] quickSort() {
        recQuickSort(theArray, 0, 9999);
        return theArray;
    }

    public int partition(int array[], int left, int right) {
        int i = left;
        int j = right;
        int temp;
        int pivot = array[(left + right) / 2];

        while (i <= j) {

            while (array[i] < pivot) {
                i++;
            }

            while (array[j] > pivot) {
                j--;
            }

            if (i <= j) {
                temp = array[i];
                array[i] = array[j];
                array[j] = temp;
                i++;
                j--;
            }

        }
        return i;
    }
```

```
    public void recQuickSort(int[] array, int left, int right) {

        int index = partition(array, left, right);

        if (left < index - 1) {
            recQuickSort(array, left, index - 1);
        }

        if (index < right) {
            recQuickSort(array, index, right);
        }

    }

}
```

```
package edu.emich.cosc;

public class HeapSort {
    private static int[] theArray;
    private static int n;
    private static int left;
    private static int right;
    private static int largest;

    public HeapSort(int[] array) {
        theArray = array;
    }

    public static void buildheap(int[] array) {
        n = array.length - 1;
        for (int i = n/2; i >= 0; i--) {
            maxheap(array, i);
        }
    }

    public static void maxheap(int[] a, int i) {
        left = 2*i;
        right = 2*i+1;
        if (left <= n && a[left] > a[i]) {
            largest=left;
        } else {
            largest=i;
        }

        if (right <= n && a[right] > a[largest]) {
            largest=right;
        }
        if (largest!=i) {
            exchange(i,largest);
            maxheap(a, largest);
        }
    }

    public static void exchange(int i, int j) {
        int t = theArray[i];
        theArray[i] = theArray[j];
        theArray[j] = t;
```

```
    }

    public int[] heapSort() {

        buildheap(theArray);

        for (int i = n; i > 0; i--){
            exchange(0, i);
            n = n - 1;
            maxheap(theArray, 0);
        }

        return theArray;
    }
}
```