```
import java.util.Comparator;
import java.util.LinkedList;
import java.util.Random;

/**
 * Your implementation of various sorting algorithms.
 *
 * @version 1.0
 */
public class Sorting {

    /**
     * Implement bubble sort.
     *
     * It should be:
     *  in-place
     *  stable
     *
     * Have a worst case running time of:
     *  O(n^2)
     *
     * And a best case running time of:
     *  O(n)
     *
     * Any duplicates in the array should be in the same relative position after
     * sorting as they were before sorting. (stable).
     *
     * See the PDF for more info on this sort.
     *
     * @throws IllegalArgumentException if the array or comparator is null
     * @param <T> data type to sort
     * @param arr the array that must be sorted after the method runs
     * @param comparator the Comparator used to compare the data in arr
     */
    public static <T> void bubbleSort(T[] arr, Comparator<T> comparator) {
        if (arr == null || comparator == null) {
            throw new IllegalArgumentException("Array or Comparator are null");
        }
        boolean noSwap = false;
        int lastIndex = arr.length - 1;
        while (!noSwap) {
            noSwap = true;
            for (int i = 0; i < lastIndex; i++) {
                if (comparator.compare(arr[i], arr[i + 1]) > 0) {
                    swap(arr, i, i + 1);
                    noSwap = false;
                }
            }
            lastIndex--;
        }

    }

    /**
     * Implement insertion sort.
     *
     * It should be:
     *  in-place
     *  stable
     *
     * Have a worst case running time of:
     *  O(n^2)
     *
     * And a best case running time of:
```

```
     *  O(n)
     *
     * Any duplicates in the array should be in the same relative position after
     * sorting as they were before sorting. (stable).
     *
     * See the PDF for more info on this sort.
     *
     * @throws IllegalArgumentException if the array or comparator is null
     * @param <T> data type to sort
     * @param arr the array that must be sorted after the method runs
     * @param comparator the Comparator used to compare the data in arr
     */
    public static <T> void insertionSort(T[] arr, Comparator<T> comparator) {
        if (arr == null || comparator == null) {
            throw new IllegalArgumentException("Array or Comparator are null");
        }
        for (int i = 1; i < arr.length; i++) {
            int h = i;
            while (h > 0 && comparator.compare(arr[h - 1], arr[h]) > 0) {
                swap(arr, h - 1, h);
                h--;
            }
        }
    }

    /**
     * Implement quick sort.
     *
     * Use the provided random object to select your pivots.
     * For example if you need a pivot between a (inclusive)
     * and b (exclusive) where b > a, use the following code:
     *
     * int pivotIndex = r.nextInt(b - a) + a;
     *
     * It should be:
     *  in-place
     *
     * Have a worst case running time of:
     *  O(n^2)
     *
     * And a best case running time of:
     *  O(n log n)
     *
     * Note that there may be duplicates in the array.
     *
     * Make sure you code the algorithm as you have been taught it in class.
     * There are several versions of this algorithm and you may not get full
     * credit if you do not use the one we have taught you!
     *
     * @throws IllegalArgumentException if the array or comparator or rand is
     * null
     * @param <T> data type to sort
     * @param arr the array that must be sorted after the method runs
     * @param comparator the Comparator used to compare the data in arr
     * @param rand the Random object used to select pivots
     */
    public static <T> void quickSort(T[] arr, Comparator<T> comparator,
                                     Random rand) {
        if (arr == null || comparator == null || rand == null) {
            throw new IllegalArgumentException("Either Arr, Comparator, "
                    + "or Rand is null");
        }

        quickSort(arr, comparator, rand, 0, arr.length - 1);
```

```
}

/**
 *
 * @param <T> data type to sort
 * @param arr the array that must be sorted after the method runs
 * @param comparator the Comparator used to compare the data in arr
 * @param rand the Random object used to select pivots
 * @param left starting index for array of objects of type T
 * @param right ending index for array of objects of type T
 */
private static <T> void quickSort(T[] arr, Comparator<T> comparator,
                                  Random rand, int left, int right) {

    if (left >= right) {
        return;
    }
    int pivotIndex = rand.nextInt(right - left + 1) + left;
    int i = left + 1;
    int j = right;
    swap(arr, left, pivotIndex);

    while (i <= j) {
        while (i <= j && comparator.compare(arr[i], arr[left]) <= 0) {
            i++;
        }

        while (i <= j && comparator.compare(arr[j], arr[left]) >= 0) {
            j--;
        }

        if (i < j) {
            swap(arr, i, j);
            j--;
            i++;

        }
    }

    swap(arr, left, j);

    if (left < j) {
        quickSort(arr, comparator, rand, left, j - 1);
    }

    if (right > i) {
        quickSort(arr, comparator, rand, i, right);
    }

}

/**
 * Implement merge sort.
 *
 * It should be:
 *  stable
 *
 * Have a worst case running time of:
 *  O(n log n)
 *
 * And a best case running time of:
 *  O(n log n)
 *
 * You can create more arrays to run mergesort, but at the end,
```

```
     * everything should be merged back into the original T[]
     * which was passed in.
     *
     * Any duplicates in the array should be in the same relative position after
     * sorting as they were before sorting.
     *
     * @throws IllegalArgumentException if the array or comparator is null
     * @param <T> data type to sort
     * @param arr the array to be sorted
     * @param comparator the Comparator used to compare the data in arr
     */
    public static <T> void mergeSort(T[] arr, Comparator<T> comparator) {
        if (arr == null || comparator == null) {
            throw new IllegalArgumentException("Array or Comparator are null");
        }

        if (arr.length == 1) {
            return;
        }

        //Mid point of array
        int midIndex = arr.length / 2;
        T[] leftSide = (T[]) new Object[midIndex];
        T[] rightSide = (T[]) new Object[arr.length - midIndex];

        //create left half array
        for (int i = 0; i < midIndex; i++) {
            leftSide[i] = arr[i];
        }

        //create right half array
        for (int i = midIndex; i < arr.length; i++) {
            rightSide[i - midIndex] = arr[i];
        }

        //Break up the array to subarrays with left and right.
        // Go down the left side first
        if (leftSide.length != 0) {
            mergeSort(leftSide, comparator);
        }
        if (rightSide.length != 0) {
            mergeSort(rightSide, comparator);
        }

        mergeBack(leftSide, rightSide, arr, comparator);

    }


    /**
     *
     * @param leftSide Left half of the array that needs to be sorted
     *                 by combining it with the rightSide array
     * @param rightSide  right half of the array that needs to be
     *                   sorted by combining it with the leftSide array
     * @param arr original array to edit that needs to be sorted
     * @param comparator the Comparator used to compare the data in arr
     * @param <T>  data type to sort
     */
    private static <T> void mergeBack(T[] leftSide, T[] rightSide,
                                      T[] arr, Comparator<T> comparator) {
        int left = 0; //Left half array starting index
        int right = 0; //Right half array starting index
        int k = 0; //Original Index of the array
```

```
        //Comparing each left and right array until one of them run
        // out of indices, then I put the rest of the other
        //array into the original starting at k
        while (left < leftSide.length && right < rightSide.length) {
            if (comparator.compare(leftSide[left], rightSide[right]) > 0) {
                arr[k++] = rightSide[right++];
            } else {
                arr[k++] = leftSide[left++];
            }
        }

        //Fill the remaining portion of the original array with left
        while (left < leftSide.length) {
            arr[k++] = leftSide[left++];
        }

        //fill the remaining portion of the original array with the right
        while (right < rightSide.length) {
            arr[k++] = rightSide[right++];
        }
    }

    /**
     * Implement LSD (least significant digit) radix sort.
     *
     * Remember you CANNOT convert the ints to strings at any point in your
     * code!
     *
     * It should be:
     *  stable
     *
     * Have a worst case running time of:
     *  O(kn)
     *
     * And a best case running time of:
     *  O(kn)
     *
     * Any duplicates in the array should be in the same relative position after
     * sorting as they were before sorting. (stable)
     *
     * Do NOT use {@code Math.pow()} in your sort. Instead, if you need to, use
     * the provided {@code pow()} method below.
     *
     * You may use {@code java.util.ArrayList} or {@code java.util.LinkedList}
     * if you wish, but it may only be used inside radix sort and any radix sort
     * helpers. Do NOT use these classes with other sorts.
     *
     * @throws IllegalArgumentException if the array is null
     * @param arr the array to be sorted
     * @return the sorted array
     */
    public static int[] lsdRadixSort(int[] arr) {
        if (arr == null) {
            throw new IllegalArgumentException("Array is null");
        }

        LinkedList<Integer>[] bucketsLikeMJ = new LinkedList[19];

        //Populate array with linkedlists
        for (int i = 0; i < bucketsLikeMJ.length; i++) {
            bucketsLikeMJ[i] = new LinkedList<Integer>();
        }
```

```
        int longestNumLength = 1;
        //Largest number in array
        int largestNum = 0;
        for (int i : arr) {
            if (Math.abs((largestNum / 10)) < (Math.abs(i) / 10)) {
                largestNum = i;
            }

        }

        //Dividing the largest number in the array by 10 until no
        // more digits left to get total number of places
        while (!(largestNum == 0)) {
            largestNum = largestNum / 10;
            longestNumLength++;
        }



        //Go thru all the digits in each number
        for (int i = 0; i < longestNumLength; i++) {
            int power = pow(10, i);
            //Sorting for each digit place into the correct bucket
            for (int j = 0; j < arr.length; j++) {
                int lsd = ((arr[j] / power) % 10) + 9;
                bucketsLikeMJ[lsd].add(arr[j]);
            }

            //Modifying the original array by going thru all of the buckets
            // and removing the elements in order
            int index = 0;
            for (int k = 0; k < bucketsLikeMJ.length; k++) {
                //Emptying one bucket at a time as if I'm Michael Jordan
                // in the 4th quarter of the finals
                while (!(bucketsLikeMJ[k].isEmpty())) {
                    arr[index] = bucketsLikeMJ[k].removeFirst();
                    index++;
                }
            }

        }

        return arr;


    }

    /**
     * Implement MSD (most significant digit) radix sort.
     *
     * Remember you CANNOT convert the ints to strings at any point in your
     * code!
     *
     * It should:
     *
     * Have a worst case running time of:
     *  O(kn)
     *
     * And a best case running time of:
     *  O(kn)
     *
     * Do NOT use {@code Math.pow()} in your sort. Instead, if you need to, use
```

```
     * the provided {@code pow()} method below.
     *
     * You may use {@code java.util.ArrayList} or {@code java.util.LinkedList}
     * if you wish, but it may only be used inside radix sort and any radix sort
     * helpers. Do NOT use these classes with other sorts.
     *
     * @throws IllegalArgumentException if the array is null
     * @param arr the array to be sorted
     * @return the sorted array
     */
    public static int[] msdRadixSort(int[] arr) {
        if (arr == null) {
            throw new IllegalArgumentException("Array is null");
        }

        LinkedList<Integer>[] bucketsLikeMJ = new LinkedList[19];

        //Populate array with linkedlists
        for (int i = 0; i < bucketsLikeMJ.length; i++) {
            bucketsLikeMJ[i] = new LinkedList<Integer>();
        }

        int[] temp = new int[arr.length];
        //Largest number in array
        int largestNum = 0;
        for (int i : arr) {
            if (Math.abs((largestNum / 10)) < (Math.abs(i)) / 10) {
                largestNum = i;
            }
        }

        //Dividing the largest number in the array by 10 until no
        // more digits left to get total number of places
        int longestNumLength = 1;
        while (!(largestNum / 10 == 0)) {
            largestNum = largestNum / 10;
            longestNumLength++;
        }


        int length = arr.length;
        int power = pow(10, longestNumLength);

        //Sort based on the MSD and place all ints in array into buckets
        for (int j = 0; j < length; j++) {
            int msd = ((arr[j] / power) % 10) + 9;
            bucketsLikeMJ[msd].add(arr[j]);
        }


        for (int bucket = 0; bucket < 19; bucket++) {
            if (bucketsLikeMJ[bucket].size() > 1 && longestNumLength > 1) {
                bucketsLikeMJ[bucket] = msdRadixSort(bucketsLikeMJ[bucket],
longestNumLength - 1);
            }
        }


        int index = 0;
        for (int bucket = 0; bucket < 19; bucket++) {
            while (!(bucketsLikeMJ[bucket].isEmpty())) {
                temp[index] = (bucketsLikeMJ[bucket].removeFirst());
                index++;
            }
```

```
        }

        return temp;
    }


    /**
     * MSD helper method to recursively sort array using Most Significant digit
     * @param arr Array to be sorted
     * @param i current digit place to sort on
     * @return
     */
    private static LinkedList<Integer> msdRadixSort(LinkedList<Integer> arr, int
i) {

        LinkedList<Integer>[] bucketsLikeMJ = new LinkedList[19];
        LinkedList<Integer> temp = new LinkedList<Integer>();

        //Populate array with linkedlists
        for (int k = 0; k < bucketsLikeMJ.length; k++) {
            bucketsLikeMJ[k] = new LinkedList<Integer>();
        }


        int length = arr.size();
        int power = pow(10, i);

        //Sort based on the MSD
        for (int j = 0; j < length; j++) {
            int num = arr.removeFirst();
            int msd = ((num / power) % 10) + 9;
            bucketsLikeMJ[msd].add(num);
        }

        for (int bucket = 0; bucket < 19; bucket++) {
            if (bucketsLikeMJ[bucket].size() > 1 && i > 0) {
                bucketsLikeMJ[bucket] = msdRadixSort(bucketsLikeMJ[bucket], i -
1);
            }

            while (!(bucketsLikeMJ[bucket].isEmpty())) {
                temp.add(bucketsLikeMJ[bucket].removeFirst());
            }
        }

        return temp;
    }




    /**
     * Calculate the result of a number raised to a power. Use this method in
     * your radix sorts instead of {@code Math.pow()}.
     *
     * DO NOT MODIFY THIS METHOD.
     *
     * @throws IllegalArgumentException if both {@code base} and {@code exp} are
     * 0
     * @throws IllegalArgumentException if {@code exp} is negative
     * @param base base of the number
     * @param exp power to raise the base to. Must be 0 or greater.
     * @return result of the base raised to that power
```

```
     */
    private static int pow(int base, int exp) {
        if (exp < 0) {
            throw new IllegalArgumentException("Exponent cannot be negative.");
        } else if (base == 0 && exp == 0) {
            throw new IllegalArgumentException(
                    "Both base and exponent cannot be 0.");
        } else if (exp == 0) {
            return 1;
        } else if (exp == 1) {
            return base;
        }
        int halfPow = pow(base, exp / 2);
        if (exp % 2 == 0) {
            return halfPow * halfPow;
        } else {
            return halfPow * halfPow * base;
        }
    }

    /**
     * Swapping method that takes in an array and
     * two indices, and swaps the indices
     * @param arr array to switch values
     * @param i first value index
     * @param j second value index
     * @param <T> data type to sort
     */
    private static <T> void swap(T[] arr, int i, int j) {
        T temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }


}
```