```
/* Ali Mojarrad
 * Comp282 Mon-Wed
 * Assignment 3
 * 04/21/2015
 * QuickSorts and HeapSorts */
import java.util.Random;

class ArraySorts {
    // book partition , random pivot
    public static void QuickSort1(int[] a, int n) {

        QuickSort1(a, 0, n - 1);
    }

    private static void QuickSort1(int[] a, int start, int end) {
        // easiest case 20
        while ((end - start) >= 20) {
            // use first element as division between small and big

            Random rand = new Random();
            int pivotIndex = start + rand.nextInt(end - start);
            // swap pivot with start
            swap(a, pivotIndex, start);
            int pivot = a[start];

            int partitionBook = partitionBook(a, start, end, pivot);

            // recursively sort the smalls and then the bigs
        if ((partitionBook - 1) - start < end - (partitionBook + 1)) {
                QuickSort1(a, start, partitionBook - 1);
                start = partitionBook + 1;

            } else {
                QuickSort1(a, partitionBook + 1, end);
                end = partitionBook - 1;
            }
        }
        // run easiest case
        insertion(a, end - start + 1);
    }

    // 2 ptr partition, random pivot
    public static void QuickSort2(int[] a, int n) {

        QuickSort2(a, 0, n - 1);

    }

    private static void QuickSort2(int[] a, int start, int end) {
        // easiest case 20
        if ((end - start + 1) < 20) {
            int num = end - start + 1;
            insertion(a, num);
        }

        while ((end - start + 1) >= 20) {
            // int pivot = a[end];
        Random rand = new Random();
        int pivotIndex = start + rand.nextInt(end - start + 1);
        swap(a, pivotIndex, end);
        int pivot = a[end];

        int partition = partition(a, start, end, pivot);

        if ((partition - 1) - start < end - (partition + 1)) {
            QuickSort2(a, start, partition - 1);
            start = partition + 1;

        } else {
            QuickSort2(a, partition + 1, end);
            end = partition - 1;
        }
    }

}

// book partition, pivot(a[lef] or a[start])
public static void QuickSort3(int[] a, int n) {

    QuickSort3(a, 0, n - 1);
}

private static void QuickSort3(int[] a, int start, int end) {
    // easiest case 20
    if ((end - start) <= 20) {
        int num = (end + 1) - start;
        insertion(a, num);
    } else {
        int pivot = a[start];
        int partitionBook = partitionBook(a, start, end, pivot);

        // recursively sort the smalls and then the bigs

    if ((partitionBook - 1) - start < end - (partitionBook + 1)) {
            QuickSort3(a, start, partitionBook - 1);
            start = partitionBook + 1;

        } else {
            QuickSort3(a, partitionBook + 1, end);
            end = partitionBook - 1;
        }
    }
}

// 2 ptr partition, random pivot
public static void QuickSort4(int[] a, int n) {

    QuickSort4(a, 0, n - 1);

}

private static void QuickSort4(int[] a, int start, int end) {
    // easiest case when array is done so do nothing
    if ((end - start + 1) <= 1) {
        return;
    }
    Random rand = new Random();
    int pivotIndex = start + rand.nextInt(end - start + 1);
    swap(a, pivotIndex, end);
    int pivot = a[end];
    int partition = partition(a, start, end, pivot);

    QuickSort4(a, start, partition - 1);

    QuickSort4(a, partition + 1, end);
}

// 2 ptr partition, random pivot
public static void QuickSort5(int[] a, int n) {
    QuickSort5(a, 0, n - 1);

}

private static void QuickSort5(int[] a, int start, int end) {
    // easiest case 500
    if ((end - start + 1) <= 500) {
        int num = end - start + 1;
        insertion(a, num);
    } else {
        Random rand = new Random();
        int pivotIndex = start + rand.nextInt(end - start + 1);
        swap(a, pivotIndex, end);
        int pivot = a[end];

        int partition = partition(a, start, end, pivot);

        if ((partition - 1) - start < end - (partition + 1)) {
            QuickSort5(a, start, partition - 1);
            start = partition + 1;

        } else {
            QuickSort5(a, partition + 1, end);
            end = partition - 1;
        }
    }
}

// random pivot, book partition
public static void QuickSort6(int[] a, int n) {

    QuickSort6(a, 0, n - 1);
}

private static void QuickSort6(int[] a, int start, int end) {
    // easiest case 1 - end of array - do nothing
    if ((end - start + 1) <= 1) {
        return;
    } else {

        // use first element as division between small and big
        Random rand = new Random();
        int pivotIndex = start + rand.nextInt(end - start);
        swap(a, pivotIndex, start);
        int pivot = a[start];

        int partitionBook = partitionBook(a, start, end, pivot);

        // recursively sort the smalls and then the bigs

    if ((partitionBook - 1) - start < end - (partitionBook + 1)) {
            QuickSort6(a, start, partitionBook - 1);
            start = partitionBook + 1;

        } else {
            QuickSort6(a, partitionBook + 1, end);
            end = partitionBook - 1;
        }
    }
}

// divide array into two part of small and big with pivot in the middle
//  2 pointers
private static int partition(int[] a, int start, int end, int pivot) {
    int startCursor = start;
    int endCursor = end;
    while (startCursor < endCursor) {
        while (a[++startCursor] < pivot)
            ;
        while (endCursor > 0 && a[--endCursor] > pivot)
            ;
        if (startCursor > endCursor) {
            break;
        } else {
            swap(a, startCursor, endCursor);
        }
    }
    swap(a, startCursor, end);
    return startCursor;

}

// divide array into two part of small and big with pivot in the middle
// 1 pointer
private static int partitionBook(int[] a, int start, int end, int pivot) {
    // the index of the last small element

    int lastSmall = start;

    for (int unknown = start + 1; unknown <= end; unknown++) {

        if (a[unknown] < pivot) {

            lastSmall++;

            swap(a, lastSmall, unknown);
        }
    }
    swap(a, lastSmall, start);
    return lastSmall;
}

private static void swap(int[] a, int start, int end) {
    int temp = a[start];
    a[start] = a[end];
    a[end] = temp;
}

public static void printArray(int[] a) {
    for (int i : a) {
        System.out.print(i + " ");
    }
}

private static int[] getArray() {
    int size = 10;
    int[] array = new int[size];
    int item = 0;
    for (int i = 0; i < size; i++) {
        item = (int) (Math.random() * 100);
        array[i] = item;
    }
    return array;
}

static int start(int iIndex) {
    return ((iIndex << 1) + 1);
}

static int end(int iIndex) {
    return ((iIndex << 1) + 2);
}

int Parent(int iIndex) {
    return ((iIndex - 1) >> 1);
}

static void Swap(int firstIndex, int secondIndex, int[] ipHeap) {
    int iTemp = ipHeap[firstIndex];
    ipHeap[firstIndex] = ipHeap[secondIndex];
    ipHeap[secondIndex] = iTemp;
}

static int SwapWithChild(int parent, int[] ipHeap, int iSize) {
    int startChild = start(parent);
    int endChild = end(parent);
    int iLargest = parent;
    if (endChild < iSize) {
        if (ipHeap[startChild] < ipHeap[endChild]) {
            iLargest = endChild;
        } else {
            iLargest = startChild;
        }
        if (ipHeap[parent] > ipHeap[iLargest]) {
            iLargest = parent;
        }
    } else if (startChild < iSize) {
        if (ipHeap[parent] < ipHeap[startChild]) {
            iLargest = startChild;
        }
    }
    if (ipHeap[parent] < ipHeap[iLargest]) {
        Swap(parent, iLargest, ipHeap);
    }
    return iLargest;
}

void RemoveRoot(int[] ipHeap, int iSize) {
    // Put the last element at the root
    ipHeap[0] = ipHeap[iSize - 1];
    --iSize;
    int iLasti = 0;
    int i = SwapWithChild(0, ipHeap, iSize);
    while (i != iLasti) {
        iLasti = i;
        i = SwapWithChild(i, ipHeap, iSize);
    }
}

int SwapWithParent(int i, int[] ipHeap) {
    if (i < 1) {
        return i;
    }
    int iParent = Parent(i);
    if (ipHeap[i] > ipHeap[iParent]) {
        Swap(i, iParent, ipHeap);
        return iParent;
    } else {
        return i;
    }
}

void AddElement(int iNewEntry, int[] ipHeap, int iSize) {
    ipHeap[iSize] = iNewEntry;
    int iLasti = iSize;
    int i = SwapWithParent(iLasti, ipHeap);
    while (iLasti != i) {
        iLasti = i;
        i = SwapWithParent(i, ipHeap);
    }
}

static void OutputArray(int[] ipArray, int iSize, int verticalBar) {
    for (int i = 0; i < iSize; ++i) {
        if (i == verticalBar) {
            System.out.print("|  ");
        }
        System.out.print(ipArray[i] + "  ");
    }
    System.out.println();
}
```

```
static void sortRoot(int[] ipHeap, int iSize) {

       // Swap the last element with the root
       Swap(0, iSize - 1, ipHeap);
       iSize--;
       int iLasti = 0;
       int i = SwapWithChild(0, ipHeap, iSize);
       while (i != iLasti) {
              iLasti = i;
              i = SwapWithChild(i, ipHeap, iSize);
       }
}

public static void HeapSort1(int[] a, int n) {
       int count = n;

       // first place a in max-heap order
       heapify(a, count);

       int end = count - 1;
       while (end > 0) {
              // swap the root(maximum value) of the heap with the
              // last element of the heap
              int tmp = a[end];
              a[end] = a[0];
              a[0] = tmp;
              // put the heap back in max-heap order
              trickleDown(a, 0, end - 1);
              // decrement the size of the heap so that the previous
              // max value will stay in its proper place
              end--;

       }
}

private static void heapify(int[] a, int count) {
       // find parent
       int start = (count - 2) / 2;
       // while there is a parent
       while (start >= 0) {
              // trickledown and decrement parent
              trickleDown(a, start, count - 1);
              start--;
       }
// after sifting down the root all nodes/elements are in heap order
}

private static void trickleDown(int[] a, int start, int end) {
       // lets start point be our root for subtrees
       int root = start;
       // as long as there is at least one child
       while ((root * 2 + 1) <= end) {
              // intialize child
              int child = root * 2 + 1;
              // if there is another child and is bigger than current one
              if (child + 1 <= end && a[child] < a[child + 1])
                     // point to the bigger one
```

```
                child = child + 1;
            // if root is smaller than biggest child
            if (a[root] < a[child]) {

                swap(a, root, child);
        root = child; // repeat to continue sifting down the child now
            } else
                return;
        }
    }
    //HeapSort2  Experimental  - Does not WORK
    public static void HeapSort2(int[] a, int n) {
        int count = n;

        heapifyTU(a, count);

        int end = count - 1;
        while (end > 0) {
            trickleUp(a, end - 1);
            end--;

        }
    }

    private static void heapifyTU(int[] a, int count) {
        // find parent
        int start = (count - 2) / 2;
        // while there is a parent
        while (start >= 0) {
            // trickleup and decrement parent
            trickleUp(a, start);
            start--;

// after sifting down the root all nodes/elements are in heap order
    }

    private static void trickleUp(int[] a, int start) {
        int parent = (start - 2) / 2;
        int bottom = a[start];

        while (start > 0 && a[parent] < bottom) {
            a[start] = a[parent];
            start = parent;
            parent = (parent - 1) / 2;
        }
        a[start] = bottom;
    }

    //Insertion Sort - runs insertion helper n times to sort all elements
    public static void insertion(int[] a, int n) {
        if (n <= 0)
            return;
        for (int i = 0; i < n; i++) {
            insertionHelper(a, i);
        }
    }
    private static void insertionHelper(int[] a, int pointer) {
        // verify pointer is not at the beginning of the array
        if (pointer <= 0)
            return;
// if pointer' value is smaller than the previous element, we need to
        // swap.
        while (pointer > 0 && a[pointer] < a[pointer - 1]) {
            Swap(pointer, pointer - 1, a);
            pointer--;

        }

    }

    public static String myName() {
        return "Ali Mojarrad";
    }

}
```