```java
/**
 * @author Bishoy A Abdelmalik
 * @class COMP 282 meeting at 2:00 PM
 * @Assigment Program #3
 * @DateTurnedIn Nov 20, 2019
 * @description this file includes ArraySorts which have all the code for
 * the diffrent sorting algorithms in addition
 * it incluedes the Pair class which was ued in some  of the sorts
 */
import java.util.Random;
class ArraySorts {
    /***********QuickSort1*********************/
    public static void QuickSort1(int[] a, int n, int cutoff) {
        Random r = new Random();
        //handle the situation if cutoff of 0 is passed in
        cutoff = (cutoff < 2) ? 2 : cutoff;
        QuickSort1(a, 0, n - 1, cutoff, r);
        insertionSort(a, n);
    }
    public static void AlmostQS1(int[] a, int n, int cutoff) {
        Random r = new Random();
        //handle the situation if cutoff of 0 is passed in
        cutoff = (cutoff < 2) ? 2 : cutoff;
        QuickSort1(a, 0, n - 1, cutoff, r);
        //insertionSort(a,n);
    }
    private static void QuickSort1(int[] a, int leftPointer, int rightPointer, int cutoff, Random r) {
        while (rightPointer - leftPointer >= (cutoff - 1)) {
            //choose random pivot in range
            int pivot = r.nextInt(rightPointer - leftPointer) + leftPointer;
            //partition
            pair partitionPoints = outsideIn(a, leftPointer, rightPointer, pivot);
            //recursively sort the smaller partition
            //and change the pointer to simulate the other
            if ((partitionPoints.getLeft() - leftPointer) < (rightPointer - partitionPoints.getRight())) {
                QuickSort1(a, leftPointer, partitionPoints.getLeft(), cutoff, r);
                leftPointer = partitionPoints.getRight();
            } else {
                QuickSort1(a, partitionPoints.getRight(), rightPointer, cutoff, r);
                rightPointer = partitionPoints.getLeft();
```

```
41              }
42          }
43      }
44      /***********End QuickSort1**********************/
45      /***********QuickSort2**********************/
46      public static void QuickSort2(int[] a, int n, int cutoff) {
47          Random r = new Random();
48          cutoff = (cutoff < 2) ? 2 : cutoff;
49          QuickSort2(a, 0, n - 1, cutoff, r);
50          insertionSort(a, n);
51      }
52      public static void AlmostQS2(int[] a, int n, int cutoff) {
53          Random r = new Random();
54          cutoff = (cutoff < 2) ? 2 : cutoff;
55          QuickSort2(a, 0, n - 1, cutoff, r);
56          //insertionSort(a,n);
57      }
58      private static void QuickSort2(int[] a, int leftPointer, int rightPointer, int cutoff, Random r) {
59          while (rightPointer - leftPointer >= (cutoff - 1)) {
60              //choose random pivot in range
61              int pivot = r.nextInt(rightPointer - leftPointer) + leftPointer;
62              //partition
63              int partitionPoints = leftToRight1Pivot(a, leftPointer, rightPointer, pivot);
64              //recursively sort the smaller partition
65              //and change the pointer to simulate the other
66              if (((partitionPoints - 1) - leftPointer) < (rightPointer - (partitionPoints + 1))) {
67                  QuickSort2(a, leftPointer, partitionPoints - 1, cutoff, r);
68                  leftPointer = partitionPoints + 1;
69              } else {
70                  QuickSort2(a, partitionPoints + 1, rightPointer, cutoff, r);
71                  rightPointer = partitionPoints - 1;
72              }
73
74          }
75
76      }
77      /***********End QuickSort2**********************/
78
79      /***********QuickSort3**********************/
80      public static void QuickSort3(int[] a, int n, int cutoff) {
```

```
81          Random r = new Random();
82          cutoff = (cutoff < 2) ? 2 : cutoff;
83          QuickSort3(a, 0, n - 1, cutoff, r);
84          insertionSort(a, n);
85
86      }
87      public static void AlmostQS3(int[] a, int n, int cutoff) {
88          Random r = new Random();
89          cutoff = (cutoff < 2) ? 2 : cutoff;
90          QuickSort3(a, 0, n - 1, cutoff, r);
91          //insertionSort(a,n);
92      }
93      private static void QuickSort3(int[] a, int leftPointer, int rightPointer, int cutoff, Random r) {
94          while (rightPointer - leftPointer >= (cutoff - 1) && leftPointer <= rightPointer) {
95
96              pair partitionPoints = leftToRight2Pivot(a, leftPointer, rightPointer, r);
97              // calculate the sizes of the partitions
98              int sizeOfLessThan = (partitionPoints.getLeft() - 1) - leftPointer;
99              int sizeOfGreaterThan = rightPointer - (partitionPoints.getRight() + 1);
100             int sizeOfMiddlePartition = (partitionPoints.getRight() - 1) - (partitionPoints.getLeft() + 1);
101             // sort the smallest 2 partitions recursively and simulate the third call with a loop
102             if (sizeOfGreaterThan > sizeOfLessThan && sizeOfGreaterThan > sizeOfMiddlePartition) {
103                 //less than
104                 QuickSort3(a, leftPointer, partitionPoints.getLeft() - 1, cutoff, r);
105                 //middle
106                 QuickSort3(a, partitionPoints.getLeft() + 1, partitionPoints.getRight() - 1, cutoff, r);
107                 leftPointer = partitionPoints.getRight() + 1;//greater than
108             } else if (sizeOfLessThan > sizeOfGreaterThan && sizeOfLessThan > sizeOfMiddlePartition) {
109                 //greater than
110                 QuickSort3(a, partitionPoints.getRight() + 1, rightPointer, cutoff, r);
111                 //middle
112                 QuickSort3(a, partitionPoints.getLeft() + 1, partitionPoints.getRight() - 1, cutoff, r);
113                 rightPointer = partitionPoints.getLeft() - 1;//less than
114             } else {
115                 //less than
116                 QuickSort3(a, leftPointer, partitionPoints.getLeft() - 1, cutoff, r);
117                 //greater than
118                 QuickSort3(a, partitionPoints.getRight() + 1, rightPointer, cutoff, r);
119                 rightPointer = partitionPoints.getRight() - 1; //middle
120                 leftPointer = partitionPoints.getLeft() + 1;
```

```
121             }
122         }
123 
124     }
125     /***********End QuickSort3**********************/
126     /***********QuickSort4**********************/
127     public static void QuickSort4(int[] a, int n, int cutoff) {
128         cutoff = (cutoff < 2) ? 2 : cutoff;
129         QuickSort4(a, 0, n - 1, cutoff);
130         insertionSort(a, n);
131     }
132 
133     private static void QuickSort4(int[] a, int leftPointer, int rightPointer, int cutoff) {
134         while (rightPointer - leftPointer >= (cutoff - 1)) {
135             int pivot = 0 + leftPointer;
136             pair partitionPoints = outsideIn(a, leftPointer, rightPointer, pivot);
137             if ((partitionPoints.getLeft() - leftPointer) < (rightPointer - partitionPoints.right)) {
138                 QuickSort4(a, leftPointer, partitionPoints.getLeft(), cutoff);
139                 leftPointer = partitionPoints.getRight();
140             } else {
141                 QuickSort4(a, partitionPoints.getRight(), rightPointer, cutoff);
142                 rightPointer = partitionPoints.getLeft();
143             }
144 
145         }
146 
147     }
148     /***********End QuickSort4**********************/
149     /***********QuickSort5**********************/
150     public static void QuickSort5(int[] a, int n, int cutoff) {
151         cutoff = (cutoff < 2) ? 2 : cutoff;
152         QuickSort5(a, 0, n - 1, cutoff);
153         insertionSort(a, n);
154     }
155 
156     private static void QuickSort5(int[] a, int leftPointer, int rightPointer, int cutoff) {
157         while (rightPointer - leftPointer >= (cutoff - 1)) {
158             int pivot = 0 + leftPointer;
159             int partitionPoints = leftToRight1Pivot(a, leftPointer, rightPointer, pivot);
160             if (((partitionPoints - 1) - leftPointer) < (rightPointer - (partitionPoints + 1))) {
```

```
161                 QuickSort5(a, leftPointer, partitionPoints - 1, cutoff);
162                 leftPointer = partitionPoints + 1;
163             } else {
164                 QuickSort5(a, partitionPoints + 1, rightPointer, cutoff);
165                 rightPointer = partitionPoints - 1;
166             }
167 
168         }
169 
170     }
171 
172     /***********End QuickSort5**********************/
173 
174 
175     private static int leftToRight1Pivot(int[] a, int lastSmall, int lastIndex, int pivot) {
176         int pivotValue = a[pivot];// save pivot value
177         // swap the pivot and the first element
178         swap(a, pivot, lastSmall);
179         pivot = lastSmall;
180         int firstUnknown = lastSmall + 1;
181         //a flag helps with big arrays
182         // if the array has a lot of values that are equal to the pivot
183         boolean pivotDuplicate = false;
184         while (firstUnknown <= lastIndex) {
185             // keep advancing the first unknown until
186             // it runs of the end of the array
187             // or
188             // it is smaller than the pivotValue in that case swap it
189             if (a[firstUnknown] > pivotValue) {
190                 firstUnknown++;
191             } else {
192                 boolean pivotAndFUEqual = a[firstUnknown] == pivotValue;
193                 if (pivotAndFUEqual) {
194                     pivotDuplicate = (!pivotDuplicate);
195                 }
196                 //sometimes swap and sometimes don't
197                 if (pivotAndFUEqual && pivotDuplicate) {
198                     firstUnknown++;
199                 } else {
200                     lastSmall++;
```

```
201                     swap(a, lastSmall, firstUnknown);
202                     firstUnknown++;
203                 }
204             }
205         }
206         swap(a, pivot, lastSmall);
207         return lastSmall;
208     }
209 
210     private static pair leftToRight2Pivot(int[] a, int firstIndex, int lastIndex, Random r) {
211         int lastSmall, firstBig, firstUnknown, smallPivot, smallPivotValue, bigPivot, bigPivotValue;
212         if ((lastIndex - firstIndex) < 2) {
213             //if array only has 2 elements decide which is bigger
214             //and swap if needed
215             if (a[firstIndex] >= a[lastIndex]) {
216                 swap(a, firstIndex, lastIndex);
217             }
218             lastSmall = firstIndex;
219             firstBig = lastIndex;
220         } else {
221             //choose the pivots
222             int pivot1 = r.nextInt(lastIndex - firstIndex) + (firstIndex);
223             swap(a, firstIndex, pivot1);
224             pivot1 = firstIndex;
225             int pivot2 = r.nextInt(lastIndex - (firstIndex + 1)) + (firstIndex + 1);
226             //decide which pivot value is bigger
227             // and put it in its right place
228             if (a[pivot1] > a[pivot2]) {
229                 bigPivot = pivot1;
230                 bigPivotValue = a[pivot1];
231                 smallPivot = pivot2;
232                 smallPivotValue = a[pivot2];
233             } else {
234                 bigPivot = pivot2;
235                 bigPivotValue = a[pivot2];
236                 smallPivot = pivot1;
237                 smallPivotValue = a[pivot1];
238             }
239             swap(a, firstIndex, smallPivot);
240             //handle the case where the big pivot
```

```
241             // comes to the first index
242             if (bigPivot == firstIndex) {
243                 bigPivot = smallPivot;
244             }
245             swap(a, lastIndex, bigPivot);
246             smallPivot = firstIndex;
247             bigPivot = lastIndex;
248             firstBig = bigPivot;
249             lastSmall = smallPivot;
250             firstUnknown = lastSmall + 1;
251             //handle duplicate values with pivot
252             // so we have unnecessary swaps
253             boolean pivotDuplicate = false;
254             while (firstUnknown < firstBig) {
255 
256                 if (a[firstUnknown] == smallPivotValue) {
257                     pivotDuplicate = (!pivotDuplicate);
258                 }
259 
260                 if (smallPivotValue < a[firstUnknown] && a[firstUnknown] <= bigPivotValue) {
261                     //the FU belong in the middle
262                     firstUnknown++;
263                 } else if (a[firstUnknown] > bigPivotValue) {
264                     //the FU belong in the big part
265                     firstBig--;
266                     swap(a, firstUnknown, firstBig);
267                 } else {
268                     //the FU belong in the small part
269                     // but to handle duplicate pivot values
270                     // we sometimes consider values = to pivot in middle
271                     if (a[firstUnknown] == smallPivotValue && pivotDuplicate) {
272                         firstUnknown++;
273                     } else {
274                         lastSmall++;
275                         swap(a, firstUnknown, lastSmall);
276                         firstUnknown++;
277                     }
278                 }
279 
280 
```

```
281             }
282             swap(a, smallPivot, lastSmall);
283             swap(a, bigPivot, firstBig);
284         }
285         return new pair(lastSmall, firstBig);
286     }
287 
288 
289     private static pair outsideIn(int[] a, int leftPointer, int rightPointer, int pivot) {
290         int pivotValue = a[pivot];// save the value of the pivot
291         while (leftPointer <= rightPointer) {//while the pointers didn't overlap
292             //find a value that needs swapping
293             while (leftPointer < rightPointer && a[leftPointer] < pivotValue) {
294                 leftPointer++;
295             }
296             //find a value that needs swapping
297             while (leftPointer < rightPointer && a[rightPointer] > pivotValue) {
298                 rightPointer--;
299             }
300             // if the pointers are not equal swap them
301             // if they are equal advance one of them so they cross
302             if (rightPointer != leftPointer) {
303                 swap(a, leftPointer, rightPointer);
304                 leftPointer++;
305                 rightPointer--;
306             } else {
307                 if (a[leftPointer] <= pivotValue) {
308                     leftPointer++;
309                 } else {//a[rightPointer] > pivotValue
310                     rightPointer--;
311                 }
312             }
313         }
314 
315         return new pair(rightPointer, leftPointer);
316     }
317 
318     /***********Heap Sort BU**********************/
319     public static void HeapSortBU(int[] a, int n) {
320         //Build heap
```

```
321         buildHeapBU(a, n);
322         // start from the end and
323         // swap the first and last elements
324         // then trickle down the first element into its right place
325         for (int i = n - 1; i >= 1; i--) {
326             swap(a, 0, i);
327             trickleDown(a, 0, i);
328         }
329     } // heapsort with linear buildheap
330 
331     private static void buildHeapBU(int[] a, int n) {
332         for (int i = n - 1; i >= 1; i--) {
333             if (a[i] <= a[(i - 1) / 2]) {
334                 // do nothing if its less than or equal
335             } else {
336                 // trickle up if the element is greater than its parent
337                 trickleUp(a, i, n);
338             }
339         }
340     }
341     /***********End Heap Sort BU**********************/
342     /***********Heap Sort TD**********************/
343     public static void HeapSortTD(int[] a, int n) {
344         //Build heap
345         buildHeapTD(a, n);
346         // start from the end and
347         // swap the first and last elements
348         // then trickle down the first element into its right place
349         for (int i = n - 1; i >= 1; i--) {
350             swap(a, 0, i);
351             trickleDown(a, 0, i);
352         }
353     } // heapsort with linear buildheap
354 
355     private static void buildHeapTD(int[] a, int n) {
356         for (int i = 0; i < n; i++) {
357             trickleDown(a, i, n);
358             //check if any of the current node parents
359             // need to be trickled down
360             for (int j = i; j > 0; j = (j - 1) / 2) {
```

```
361                 if (a[(j - 1) / 2] < a[j]) {
362                     trickleDown(a, (j - 1) / 2, n);
363                 }
364             }
365 
366         }
367     }
368 
369     /***********End Heap Sort TD**********************/
370     private static void trickleUp(int[] a, int i, int n) {
371         int save = a[i];
372         boolean exit = false;
373         while (((i - 1) / 2) >= 0 && !exit) {
374             if (save > a[(i - 1) / 2]) {
375                 a[i] = a[(i - 1) / 2];
376                 if (i > 0) {
377                     trickleDown(a, i, n);
378                 }
379                 if (i - 1 < 0) {//end of array
380                     exit = true;
381                 }
382                 i = (i - 1) / 2;
383 
384             } else {
385                 exit = true;
386             }
387         }
388         a[i] = save;
389     }
390 
391     private static void trickleDown(int[] a, int i, int n) {
392         int save = a[i];
393         boolean exit = false;
394         boolean secondChildExist = (2 * i + 2) < n;
395         boolean firstChildExist = (2 * i + 1) < n;
396 
397         while ((secondChildExist || firstChildExist) && !exit) {
398             if (secondChildExist) {
399                 if (a[2 * i + 2] > a[2 * i + 1] && save < a[2 * i + 2]) {
400                     a[i] = a[2 * i + 2];
```

```
401                     i = 2 * i + 2;
402                 } else if (save < a[2 * i + 1]) {
403                     a[i] = a[2 * i + 1];
404                     i = 2 * i + 1;
405                 } else {
406                     exit = true;
407                 }
408             } else if (firstChildExist && save < a[2 * i + 1]) {
409                 a[i] = a[2 * i + 1];
410                 i = 2 * i + 1;
411             } else {
412                 exit = true;
413 
414             }
415             firstChildExist = (2 * i + 1) < n;
416             secondChildExist = (2 * i + 2) < n;
417 
418         }
419         a[i] = save;
420     }
421 
422     /*
423       swap method perform basic swap that is needed any of the sorts
424       and doesn't swap if the 2 indices are the same
425      */
426     private static void swap(int[] a, int swap, int with) {
427         if (swap == with) {
428         } else {
429             int save = a[swap];
430             a[swap] = a[with];
431             a[with] = save;
432         }
433     }
434 
435     /**********insertionSort**********/
436     public static void insertionSort(int[] a, int n) {
437         //start from the second element in the array and insert every element
438         for (int i = 1; i <= n; i++) {
439             insert(a, i);
440         }
```

```java
    }// Insertion Sort

    private static void insert(int[] a, int n) {
        int i;
        int save = a[n - 1];
        //shift all the elements one spot
        for (i = n - 2; i >= 0 && save < a[i]; i--) {
            a[i + 1] = a[i];
        }
        a[i + 1] = save;
    }

    /***********End insertionSort**********************/
    public static String myName() {
        return "Bishoy Abdelamlik";
    }


}

// use this class to return two values in the outside-in and the 2-pivot // partition methods
class pair {
    public int left, right;

    public pair(int left, int right) {
        this.left = left;
        this.right = right;
    }

    // some getters below
    public int getLeft() {
        return left;
    }

    public int getRight() {
        return right;
    }
}
```