10/26/15 01:50:29 /home/15504319/DSA120/DSAAssignment/connorLib/Sorts.java

```
  1  /*****************************************************************************
  2  *  FILE: Sorts.java
  3  *  AUTHOR: Connor Beardsmore - 15504319
  4  *  UNIT: DSA120 Assignment S2- 2015
  5  *  PURPOSE: Implementation of The 5 Basic Sorting Algorithms
  6  *  LAST MOD: 05/10/15
  7  *  REQUIRES: NONE
  8  *****************************************************************************/
  9  package connorLib;
 10
 11  public class Sorts
 12  {
 13      //bubbleSort
 14      //IMPORT: array (int[])
 15      //PURPOSE: Implements Bubble Sort Sorting Algorithm
 16
 17      public static void bubbleSort(ISortable[] array)
 18      {
 19          int pass = 0;
 20          ISortable[] temp;
 21          boolean sorted;
 22
 23          do
 24          {
 25              //We assume its sorted, find out during loops
 26              sorted = true;
 27              //Up to -pass Optimization Applied. Don't check final element
 28              for ( int ii = 0; ii < ( array.length -1 - pass ); ii++ )
 29              {
 30                  if ( array[ii+1].lessThan( array[ii] ) )
 31                  {
 32                      //Swap 2 elements out of order
 33                      swap( array, ii, ii + 1 );
 34
 35                      //Still need to continue sorting
 36                      sorted = false;
 37                  }
 38              }
 39              //Each for loop is one pass of the array
 40              pass += 1;
 41
 42          //Stop sorting when sorted, ie. No Swaps Occur Optimization
 43          } while ( !sorted );
 44      }
 45  //-----------------------------------------------------------------------------------------------------------
 46      //selectionSort
 47      //IMPORT: array (int[])
 48      //PURPOSE: Implements Selection Sort Sorting Algorithm
 49
 50      public static void selectionSort(ISortable[] array)
 51      {
 52          int minIndex;
 53          int temp;
 54
 55          for ( int pass = 0; pass < array.length; pass++ )
 56          {
 57              minIndex = pass;
 58              //Ignore initial element, Already set to smallest
 59              for ( int jj = pass + 1; jj < array.length; jj++ )
 60              {
 61                  //Update Newly Found Minimum Index
 62                  if ( array[jj].lessThan( array[minIndex] ) )
 63                  {
 64                      minIndex = jj;
 65                  }
 66              }
 67
 68              //Do the actual swap
 69              swap( array, minIndex, pass );
 70          }
 71      }
 72  //-----------------------------------------------------------------------------------------------------------
 73      //insertionSort
 74      //IMPORT: array (int[])
 75      //PURPOSE: Implements Insertion Sort Sorting Algorithm
 76
 77      public static void insertionSort(ISortable[] array)
 78      {
 79          int ii = 0;
 80          ISortable temp;
 81          //Start Inserting at Element 1. 0 is already sorted
 82          for ( int pass = 1; pass < array.length; pass++ )
 83          {
 84              //Start from last item and go backwards
```

```
85              ii = pass;
86              temp = array[ii];
87
88              //Insert into sub-array to left of pass.
89              //Use > To keep the sort stable
90              while ( (ii > 0) && (temp.lessThan( array[ii - 1]) ) )
91              {
92                  //Shuffle until correct location
93                  array[ii] = array[ii - 1];
94                  ii--;
95              }
96
97              array[ii] = temp;
98          }
99      }
100 //-----------------------------------------------------------------------------------------------------
101     //mergeSortRecurse
102     //IMPORT: array (int[])
103     //PURPOSE: Kicks off mergeSortRecurve from given array. kick-starts
104
105     public static void mergeSort(ISortable[] array)
106     {
107         mergeSortRecurse( array, 0, array.length -1  );
108     }
109 //-----------------------------------------------------------------------------------------------------
110     //mergeSortRecurse
111     //IMPORT: array (int[]), leftIndex (int), rightIndex (int)
112     //PURPOSE: Performs the merge sort recursive calls
113
114     private static void mergeSortRecurse(ISortable[] array, int leftIndex, int rightIndex)
115     {
116         int midIndex;
117         if ( leftIndex < rightIndex )
118         {
119             midIndex = ( leftIndex + rightIndex ) / 2;
120
121             //Recursively: Sort Left and Right sides of sub-arrays
122             mergeSortRecurse(array, leftIndex, midIndex);
123             mergeSortRecurse(array, midIndex + 1, rightIndex);
124
125             //Merge left and right sub-arrays
126             merge(array, leftIndex, midIndex, rightIndex);
127         }
128     }
129 //-----------------------------------------------------------------------------------------------------
130     //merge
131     //IMPORT: array (int[]), leftIndex (int), midIndex (int) rightIndex (int)
132     //PURPOSE: Merge sub-arrays back together
133
134     private static void merge(ISortable[] array, int leftIndex, int midIndex, int rightIndex)
135     {
136         ISortable[] tempArray = new ISortable[rightIndex - leftIndex + 1];
137         //Index for front of left sub-array
138         int ii = leftIndex;
139         //Index for front of right sub-array
140         int jj = midIndex + 1;
141         //Index for next free element in tempArray
142         int kk = 0;
143
144         while ( (ii <= midIndex) && (jj <= rightIndex) )
145         {
146             //Take from left sub-array
147             if ( ( array[ii].lessThan(array[jj]) ) || ( array[ii].equals(array[jj]) ) )
148             {
149                 tempArray[kk] = array[ii];
150                 ii++;
151             }
152             //Take from right sub-array
153             else
154             {
155                 tempArray[kk] = array[jj];
156                 jj++;
157             }
158             kk++;
159         }
160
161         //Flush Remainder from left sub-array (midIndex Inclusively)
162         for ( ii = ii; ii <= midIndex; ii++ )
163         {
164             tempArray[kk] = array[ii];
165             kk++;
166         }
167         //Flush Remainder from right sub-array (rightIndex Inclusively)
168         for ( jj = jj; jj <= rightIndex; jj++ )
169         {
170             tempArray[kk] = array[jj];
171             kk++;
```

```
172         }
173
174         //Copy sorted tempArray back to actual array
175         for ( kk = leftIndex; kk <= rightIndex; kk++ )
176         {
177             //kk-leftIndex To align indexing to zero
178             array[kk] = tempArray[kk - leftIndex];
179         }
180     }
181 //-----------------------------------------------------------------------------------------------------
182     //quickSort
183     //IMPORT: array (int[])
184     //PURPOSE: Front end to kick off quick sort
185
186     public static void quickSort(ISortable[] array)
187     {
188         quickSortRecurse( array, 0, array.length - 1 );
189     }
190 //-----------------------------------------------------------------------------------------------------
191     //quickSortRecurse
192     //IMPORT: array (int[]), leftIndex (int), rightIndex (int)
193     //PURPOSE: Performs quick sort recursive calls
194
195     private static void quickSortRecurse(ISortable[] array, int leftIndex, int rightIndex)
196     {
197         int pivot, newPivot;
198
199         //Check that array is bigger than size one
200         if ( rightIndex > leftIndex )
201         {
202             //Pivot Selection Strategy - Use Middle For Simplicity
203             pivot = ( leftIndex + rightIndex ) / 2;
204             newPivot = doPartitioning( array, leftIndex, rightIndex, pivot );
205
206             //Recursively Sort Left Paritition, and Right Partition
207             quickSortRecurse( array, leftIndex, newPivot - 1 );
208             quickSortRecurse( array, newPivot + 1, rightIndex );
209         }
210     }
211 //-----------------------------------------------------------------------------------------------------
212     //doPartitioning
213     //IMPORT: array (int[]), leftIndex (int), rightIndex (int), pivot (int)
214     //EXPORT: newPivot (int)
215     //PURPOSE: Does The Actual Partioning based on the pivots. calcs new pivot.
216
217     private static int doPartitioning(ISortable[] array, int leftIndex, int rightIndex, int pivot)
218     {
219         int curIndex, temp, newPivot;
220         ISortable pivotVal;
221
222         //Swap pivotVal with right most element
223         pivotVal = array[pivot];
224         array[pivot] = array[rightIndex];
225         array[rightIndex] = pivotVal;
226
227         //Find all values smaller than pivot, transfer to left side of array
228         curIndex = leftIndex;
229
230         for ( int ii = leftIndex; ii < rightIndex; ii++ )
231         {
232             //Find next value to go on left side
233             if ( array[ii].lessThan(pivotVal) )
234             {
235                 //Push value onto left side of pivot
236                 swap( array, ii, curIndex );
237                 curIndex++;
238             }
239         }
240
241         //Put pivot in its rightful place
242         newPivot = curIndex;
243         array[rightIndex] = array[newPivot];
244         array[newPivot] = pivotVal;
245
246         return newPivot;
247     }
248 //-----------------------------------------------------------------------------------------------------
249     //swap
250     //IMPORT: array (int[]), source, dest
251     //PURPOSE: Swaps 2 items in an array, using temp variable
252     //ASSERTION: order of source/dest unimportant, same either way
253
254     private static void swap(ISortable[] array, int source, int dest)
255     {
256         ISortable temp = array[dest];
257         array[dest] = array[source];
258         array[source] = temp;
```

```
259       }
260
261   }
```