Homework 8: Sorting

Graded

Student
Vidit Dharmendra Pokharna

Total Points
97 / 100 pts

Autograder Score
97.0 / 100.0

Question 2

Feedback & Manual Grading

0 / 0 pts

✔ **+ 0 pts** Correct

[-2] kthSelect should be recursive

[-3] LsdRadix - make sure to have a specific edge case for whether one of the values is Integer.MIN_VALUE, because Math.abs(Integer.MIN_VALUE) = Integer.MIN_VALUE (due to the way overflow in Java works), in which case you won't count this as the max

Great work!! -Tomer

Autograder Results

Autograder Output

```
If you're seeing this message, everything compiled and ran properly!
-CS1332 TAs
```

Submitted Files

Sorting.java Download

```java
import java.util.Comparator;
import java.util.PriorityQueue;
import java.util.Queue;
import java.util.Random;
import java.util.List;
import java.util.LinkedList;

/**
 * Your implementation of various sorting algorithms.
 *
 * @author Vidit Pokharna
 * @version 1.0
 * @userid vpokharna3
 * @GTID 903772087
 *
 * Collaborators: LIST ALL COLLABORATORS YOU WORKED WITH HERE
 *
 * Resources: LIST ALL NON-COURSE RESOURCES YOU CONSULTED HERE
 */
public class Sorting {

    /**
     * Implement selection sort.
     *
     * It should be:
     * in-place
     * unstable
     * not adaptive
     *
     * Have a worst case running time of:
     * O(n^2)
     *
     * And a best case running time of:
     * O(n^2)
     *
     * @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
     * @throws java.lang.IllegalArgumentException if the array or comparator is
     *                                            null
     */
    public static <T> void selectionSort(T[] arr, Comparator<T> comparator) {
        if (arr == null || comparator == null) {
            throw new IllegalArgumentException("The array or comparator is null");
        }
        int length = arr.length;
```

```
47
48          for (int a = 0; a < length - 1; a++) {
49              int min = a;
50              for (int b = a + 1; b < length; b++) {
51                  if (comparator.compare(arr[b], arr[min]) < 0) {
52                      min = b;
53                  }
54              }
55              swap(min, a, arr);
56          }
57      }
58
59      /**
60       * Helper method to swap two elements
61       * @param <T> the type of data being swapped
62       * @param arr the array from which two elements will be swapped
63       * @param index1 the index of the first element to swap
64       * @param index2 the index of the second element to swap
65       */
66      private static <T> void swap(int index1, int index2, T[] arr) {
67          T temp = arr[index1];
68          arr[index1] = arr[index2];
69          arr[index2] = temp;
70      }
71
72      /**
73       * Implement cocktail sort.
74       *
75       * It should be:
76       * in-place
77       * stable
78       * adaptive
79       *
80       * Have a worst case running time of:
81       * O(n^2)
82       *
83       * And a best case running time of:
84       * O(n)
85       *
86       * NOTE: See pdf for last swapped optimization for cocktail sort. You
87       * MUST implement cocktail sort with this optimization
88       *
89       * @param <T>        data type to sort
90       * @param arr        the array that must be sorted after the method runs
91       * @param comparator the Comparator used to compare the data in arr
92       * @throws java.lang.IllegalArgumentException if the array or comparator is
93       *                                            null
94       */
95      public static <T> void cocktailSort(T[] arr, Comparator<T> comparator) {
```

```
96          if (arr == null || comparator == null) {
97              throw new IllegalArgumentException("The array or comparator is null");
98          }
99          int start = 0;
100         int end = arr.length - 1;
101
102         while (start < end) {
103             int swapped = start;
104             for (int a = start; a < end; a++) {
105                 if (comparator.compare(arr[a], arr[a + 1]) > 0) {
106                     swap(a, a + 1, arr);
107                     swapped = a;
108                 }
109             }
110             end = swapped;
111             for (int b = end; b > start; b--) {
112                 if (comparator.compare(arr[b], arr[b - 1]) < 0) {
113                     swap(b, b - 1, arr);
114                     swapped = b;
115                 }
116             }
117             start = swapped;
118         }
119     }
120
121     /**
122      * Implement merge sort.
123      *
124      * It should be:
125      * out-of-place
126      * stable
127      * not adaptive
128      *
129      * Have a worst case running time of:
130      * O(n log n)
131      *
132      * And a best case running time of:
133      * O(n log n)
134      *
135      * You can create more arrays to run merge sort, but at the end, everything
136      * should be merged back into the original T[] which was passed in.
137      *
138      * When splitting the array, if there is an odd number of elements, put the
139      * extra data on the right side.
140      *
141      * Hint: If two data are equal when merging, think about which subarray
142      * you should pull from first
143      *
144      * @param <T>       data type to sort
```

```
145     * @param arr        the array to be sorted
146     * @param comparator the Comparator used to compare the data in arr
147     * @throws java.lang.IllegalArgumentException if the array or comparator is
148     *                                  null
149     */
150    public static <T> void mergeSort(T[] arr, Comparator<T> comparator) {
151        if (arr == null || comparator == null) {
152            throw new IllegalArgumentException("The array or comparator is null");
153        }
154
155        int length = arr.length;
156
157        if (length <= 1) {
158            return;
159        }
160
161        int middleIndex = length / 2;
162        T[] leftArr = (T[]) new Object[middleIndex];
163        T[] rightArr = (T[]) new Object[length - middleIndex];
164        int leftLength = leftArr.length;
165        int rightLength = rightArr.length;
166
167
168        for (int a = 0; a < middleIndex; a++) {
169            leftArr[a] = arr[a];
170        }
171
172        for (int a = middleIndex; a < length; a++) {
173            rightArr[a - middleIndex] = arr[a];
174        }
175
176        mergeSort(leftArr, comparator);
177        mergeSort(rightArr, comparator);
178
179        int currIndex = 0;
180        int leftIndex = 0;
181        int rightIndex = 0;
182
183        while (leftIndex < leftLength && rightIndex < rightLength) {
184            if (comparator.compare(leftArr[leftIndex], rightArr[rightIndex]) <= 0) {
185                arr[currIndex] = leftArr[leftIndex];
186                leftIndex++;
187            } else {
188                arr[currIndex] = rightArr[rightIndex];
189                rightIndex++;
190            }
191            currIndex++;
192        }
193
```

```
194         while (leftIndex < leftLength) {
195             arr[currIndex] = leftArr[leftIndex];
196             currIndex++;
197             leftIndex++;
198         }
199 
200         while (rightIndex < rightLength) {
201             arr[currIndex] = rightArr[rightIndex];
202             currIndex++;
203             rightIndex++;
204         }
205     }
206 
207     /**
208      * Implement kth select.
209      *
210      * Use the provided random object to select your pivots. For example if you
211      * need a pivot between a (inclusive) and b (exclusive) where b > a, use
212      * the following code:
213      *
214      * int pivotIndex = rand.nextInt(b - a) + a;
215      *
216      * If your recursion uses an inclusive b instead of an exclusive one,
217      * the formula changes by adding 1 to the nextInt() call:
218      *
219      * int pivotIndex = rand.nextInt(b - a + 1) + a;
220      *
221      * It should be:
222      * in-place
223      *
224      * Have a worst case running time of:
225      * O(n^2)
226      *
227      * And a best case running time of:
228      * O(n)
229      *
230      * You may assume that the array doesn't contain any null elements.
231      *
232      * Make sure you code the algorithm as you have been taught it in class.
233      * There are several versions of this algorithm and you may not get full
234      * credit if you do not implement the one we have taught you!
235      *
236      * @param <T>       data type to sort
237      * @param k         the index to retrieve data from + 1 (due to
238      *                  0-indexing) if the array was sorted; the 'k' in "kth
239      *                  select"; e.g. if k == 1, return the smallest element
240      *                  in the array
241      * @param arr       the array that should be modified after the method
242      *                  is finished executing as needed
```

```
243     * @param comparator the Comparator used to compare the data in arr
244     * @param rand       the Random object used to select pivots
245     * @return the kth smallest element
246     * @throws java.lang.IllegalArgumentException if the array or comparator
247     *                                  or rand is null or k is not
248     *                                  in the range of 1 to arr
249     *                                  .length
250     */
251    public static <T> T kthSelect(int k, T[] arr, Comparator<T> comparator,
252                                  Random rand) {
253        if (arr == null || comparator == null || rand == null) {
254            throw new IllegalArgumentException("The array, comparator, or random object is null");
255        } else if (k < 1 || k > arr.length) {
256            throw new IllegalArgumentException("The value of k is invalid");
257        }
258        int left = 0;
259        int right = arr.length - 1;
260
261        while (true) {
262            int pivotIndex = rand.nextInt(right - left + 1) + left;
263            int j = quickSelect(arr, left, right, pivotIndex, comparator);
264            if (k - 1 == j) {
265                return arr[k - 1];
266            } else if (k - 1 < j) {
267                right = j - 1;
268            } else {
269                left = j + 1;
270            }
271        }
272    }
273
274    /**
275     * Helper method to use the quickselect algorithm for kselect
276     * @param <T> the type of data being swapped
277     * @param arr the array which the quickselect algorithm will utilize
278     * @param left the index of the leftmost element in the array
279     * @param right the index of the rightmost element in the array
280     * @param pivotIndex the index of pivot found in kselect
281     * @param comparator the object used to compare to elements
282     * @return the value of int j, which will be compared to k
283     */
284    private static <T> int quickSelect(T[] arr, int left, int right, int pivotIndex, Comparator<T> comparator)
    {
285        T pivotValue = arr[pivotIndex];
286        int i = left + 1;
287        int j = right;
288        swap(left, pivotIndex, arr);
289        while (i <= j) {
290            while (i <= j && comparator.compare(arr[i], pivotValue) <= 0) {
```

```
            i++;
        }
        while (i <= j && comparator.compare(arr[j], pivotValue) >= 0) {
            j--;
        }
        if (i <= j) {
            swap(i, j, arr);
            i++;
            j--;
        }
    }
    swap(left, j, arr);
    return j;
}

/**
 * Implement LSD (least significant digit) radix sort.
 *
 * 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 implement the one we have taught you!
 *
 * Remember you CANNOT convert the ints to strings at any point in your
 * code! Doing so may result in a 0 for the implementation.
 *
 * It should be:
 * out-of-place
 * stable
 * not adaptive
 *
 * Have a worst case running time of:
 * O(kn)
 *
 * And a best case running time of:
 * O(kn)
 *
 * You are allowed to make an initial O(n) passthrough of the array to
 * determine the number of iterations you need. The number of iterations
 * can be determined using the number with the largest magnitude.
 *
 * At no point should you find yourself needing a way to exponentiate a
 * number; any such method would be non-O(1). Think about how how you can
 * get each power of BASE naturally and efficiently as the algorithm
 * progresses through each digit.
 *
 * Refer to the PDF for more information on LSD Radix Sort.
 *
 * You may use ArrayList or LinkedList if you wish, but it may only be
 * used inside radix sort and any radix sort helpers. Do NOT use these
```

```
340     * classes with other sorts. However, be sure the List implementation you
341     * choose allows for stability while being as efficient as possible.
342     *
343     * Do NOT use anything from the Math class except Math.abs().
344     *
345     * @param arr the array to be sorted
346     * @throws java.lang.IllegalArgumentException if the array is null
347     */
348    public static void lsdRadixSort(int[] arr) {
349        if (arr == null) {
350            throw new IllegalArgumentException("The array is null");
351        }
352
353        LinkedList<Integer>[] buckets = new LinkedList[19];
354        for (int a = 0; a < buckets.length; a++) {
355            buckets[a] = new LinkedList<>();
356        }
357
358        int maxNumber = 0;
359        for (int a : arr) {
360            if (Math.abs(a) > maxNumber) {
361                maxNumber = Math.abs(a);
362            }
363        }
364        int iterations = 0;
365        while (maxNumber > 0) {
366            iterations++;
367            maxNumber /= 10;
368        }
369
370        int divide = 1;
371
372        for (int a = 0; a < iterations; a++) {
373            for (int b : arr) {
374                int ithDigit = ((int) ((Math.abs((long) b)) / divide)) % 10;
375                if (b < 0) {
376                    ithDigit *= -1;
377                }
378                buckets[9 + ithDigit].add(b);
379            }
380            int index = 0;
381            for (LinkedList<Integer> bucket : buckets) {
382                while (!bucket.isEmpty()) {
383                    arr[index] = bucket.removeFirst();
384                    index++;
385                }
386            }
387            divide *= 10;
388        }
```

```
389     }
390
391     /**
392      * Implement heap sort.
393      *
394      * It should be:
395      * out-of-place
396      * unstable
397      * not adaptive
398      *
399      * Have a worst case running time of:
400      * O(n log n)
401      *
402      * And a best case running time of:
403      * O(n log n)
404      *
405      * Use java.util.PriorityQueue as the heap. Note that in this
406      * PriorityQueue implementation, elements are removed from smallest
407      * element to largest element.
408      *
409      * Initialize the PriorityQueue using its build heap constructor (look at
410      * the different constructors of java.util.PriorityQueue).
411      *
412      * Return an int array with a capacity equal to the size of the list. The
413      * returned array should have the elements in the list in sorted order.
414      *
415      * @param data the data to sort
416      * @return the array with length equal to the size of the input list that
417      * holds the elements from the list is sorted order
418      * @throws java.lang.IllegalArgumentException if the data is null
419      */
420     public static int[] heapSort(List<Integer> data) {
421         if (data == null) {
422             throw new IllegalArgumentException("The list provided is null");
423         }
424         Queue<Integer> heap = new PriorityQueue<>(data);
425         int[] sortedList = new int[data.size()];
426
427         for (int a = 0; a < data.size(); a++) {
428             sortedList[a] = heap.remove();
429         }
430
431         return sortedList;
432     }
433 }
434
```