

Instructions:

Using Java, efficiently implement both mergeSort and quickSort. Do both of these using recursion. Complete a table to show the relative speeds of these algorithms on several kinds of input.

Terms:

Key comparison: a comparison involving one or more keys

Key movement: an assignment involving one or more keys (do not count the initial list set up)

mergeSort:

Use the algorithm from McConnell (which is also on the class slides). Make sure that your base case occurs at size 3 or less, since such a list can be sorted with 3 or fewer comparisons (Hint: bubbleSort or insertionSort) without requiring further recursion. Display the list being used at the beginning of each recursive call; display the entire list as each recursive call ends (use indent with arrow to distinguish). Note that the McConnell algorithm can be optimized slightly, and you are encouraged to incorporate optimizations.

Also, count the number of key comparisons and key movements in this algorithm with explicit counters in your code, e.g. when you do a key comparison you increment this counter. As well, time the algorithm (with `System.nanoTime()`). For the official times that you record in the table, make sure that you do not include any output operation (e.g. `System.out.print()`). You can make printing optional in your program, so that you are not printing when you are trying to get an accurate time measurement.

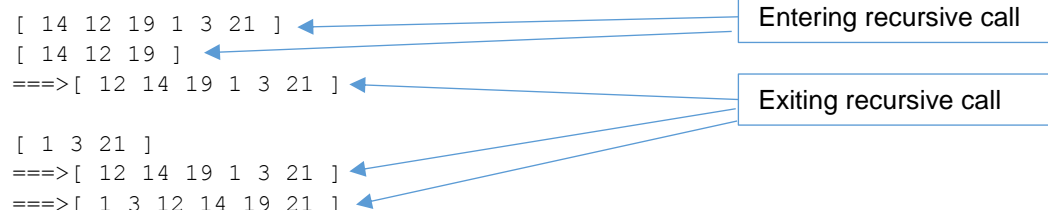
quickSort:

Implement an efficient in-place recursive quickSort, using median of 3 (first, middle, last). Make sure that your base case occurs at size 3 or less, since that can be sorted with 3 or fewer comparisons. Display the list being used at the beginning of each recursive call; display the entire list as each recursive call ends (use indent with arrow to distinguish).

Again, count the number of key comparisons and key movements in this algorithm, and time the algorithm. See notes above under mergeSort.

Sample Output from Program:

Running mergeSort on list: [14 12 19 1 3 21]



```
[ 14 12 19 1 3 21 ]  
[ 14 12 19 ]  
==>[ 12 14 19 1 3 21 ]  
  
[ 1 3 21 ]  
==>[ 12 14 19 1 3 21 ]  
==>[ 1 3 12 14 19 21 ]
```

Entering recursive call

Exiting recursive call

```
Finished sort:
[ 1 3 12 14 19 21 ]
```

```
=====
Running QuickSort on list:  [ 14 12 19 1 3 21 ]
```

```
[ 14 12 19 1 3 21 ]
[ 3 12 14 1 ]
[ 1 ]
==>[ 1 3 14 12 19 21 ]
```

Entering recursive call

```
[ 14 12 ]
==>[ 1 3 12 14 19 21 ]
==>[ 1 3 12 14 19 21 ]
```

Exiting recursive call

```
[ 21 ]
==>[ 1 3 12 14 19 21 ]
```

```
Finished sort:
[ 1 3 12 14 19 21 ]
```

```
mergeSort      key comparisons: 11
                key movements: 13
                time: 4909428 nanoseconds = 5 millis
quickSort      key comparisons: 14
                key movements: 18
                time: 3003870 nanoseconds = 3 millis
```

Timing Information
--depending on your
implementation
comparisons and
movements could
vary *slightly*

Table:

Fill in the following table, using the first four lines of `data.txt` as the test data. The first line of `data.txt` is the small list, the next line is the “all same” list, the third line is an in-order list, and the fourth line is a random big list. The remainder of the file contains further testing you can use to make sure that your methods work. Each line of the data file starts with a number indicating how many entries are on the line. When filling out the time, you should probably run the program a few times to make sure you are not recording an anomaly (you could record an average of 3, for example). Time should be recorded in an appropriate unit (e.g. nanoseconds, microseconds, milliseconds, seconds, or minutes)

		# Key Comparisons	# Key Movements	Time
mergeSort	Small list			
	All same			
	In order list			
	Random big list			

quickSort	Small list			
	All same			
	In order list			
	Random big list			
Conclusion:				

Examples of considerations in the conclusion: which sort is better?, does the time correspond with the # of key comparisons or with the # of key movements?, what other factors might affect time?, what did you learn?

Hand in:

1. a printed copy of your program (with documentation)
2. a printed copy of your testing results for only the first list in the data file (i.e. you are printing out one test case for each of mergeSort and quickSort)
3. the table you completed
4. submit your program on eClass

I will mark all of the following: Java programming style, documentation (including variable names, indentation, file headers, method headers, and inline comments), correctness, including testing results, complexity, and accurate completion of the table.