CSCI 241, Summer 2017, Filip Jagodzinski
Programming Assignment 1
Check Canvas for Due Date and Time

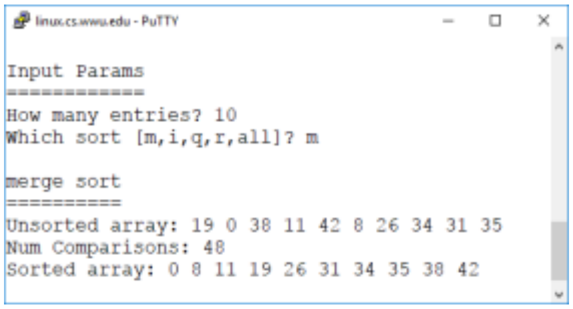
WESTERN WASHINGTON UNIVERSITY
Computer Science

For this first programming assignment, you will implement and compare quick sort, merge sort, insertion sort, and Least Significant Digit (LSD) radix sort by tallying the number of comparisons each of them performs when sorting an array of size specified by the user.

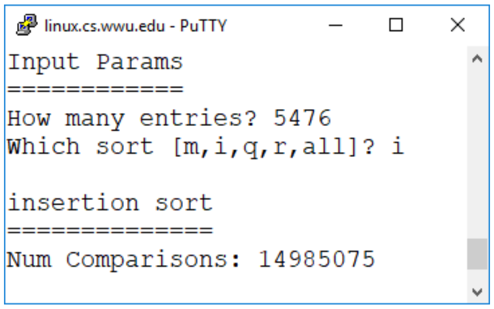
I. Program Behavior, User's perspective

- Prompts the user for the size of the array, *n*
- Prompts the user to specify if merge sort, quick sort, insertion sort, radix sort, or all of those 4 sorts, should be employed to sort the entries of the array of size *n*. The options should be one letter designed, ***m***, ***q***, ***i*** and ***r***
- If ***all*** sorts are specified, the input to each sort must be identical
- If *n* < 20, the pre-sorted and sorted array's contents are printed for each sort invoked
- If n ≥ 20, the pre-sorted and sorted array's contents are NOT printed for each sort invoked
- The count of comparisons performed for each sort invoked are output.

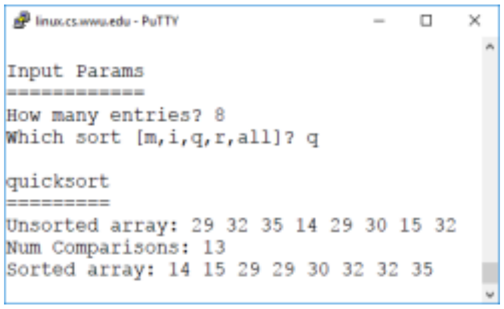
Four sample invocations on single-choice sorts



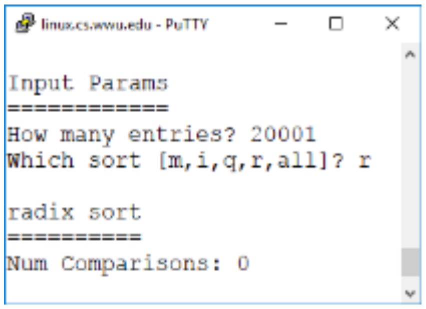
```
Input Params
============
How many entries? 10
Which sort [m,i,q,r,all]? m

merge sort
==========
Unsorted array: 19 0 38 11 42 8 26 34 31 35
Num Comparisons: 48
Sorted array: 0 8 11 19 26 31 34 35 38 42
```



```
Input Params
============
How many entries? 5476
Which sort [m,i,q,r,all]? i

insertion sort
==============
Num Comparisons: 14985075
```

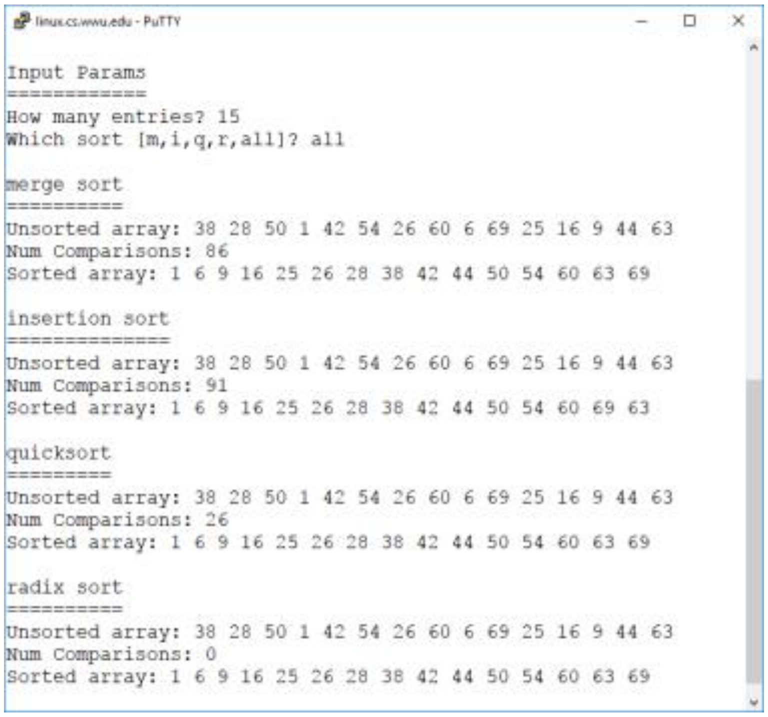


```
Input Params
============
How many entries? 8
Which sort [m,i,q,r,all]? q

quicksort
=========
Unsorted array: 29 32 35 14 29 30 15 32
Num Comparisons: 13
Sorted array: 14 15 29 29 30 32 32 35
```

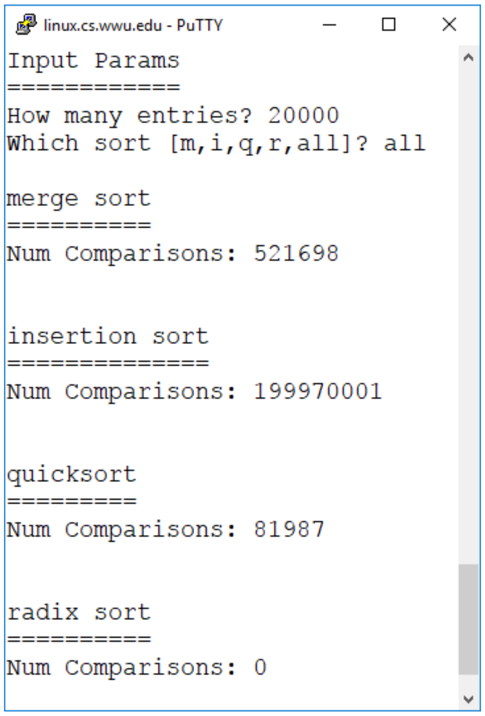


```
Input Params
============
How many entries? 20001
Which sort [m,i,q,r,all]? r

radix sort
==========
Num Comparisons: 0
```

Two sample invocations with the ***all*** option.



```
Input Params
============
How many entries? 15
Which sort [m,i,q,r,all]? all

merge sort
==========
Unsorted array: 38 28 50 1 42 54 26 60 6 69 25 16 9 44 63
Num Comparisons: 86
Sorted array: 1 6 9 16 25 26 28 38 42 44 50 54 60 63 69

insertion sort
==============
Unsorted array: 38 28 50 1 42 54 26 60 6 69 25 16 9 44 63
Num Comparisons: 91
Sorted array: 1 6 9 16 25 26 28 38 42 44 50 54 60 69 63

quicksort
=========
Unsorted array: 38 28 50 1 42 54 26 60 6 69 25 16 9 44 63
Num Comparisons: 26
Sorted array: 1 6 9 16 25 26 28 38 42 44 50 54 60 63 69

radix sort
==========
Unsorted array: 38 28 50 1 42 54 26 60 6 69 25 16 9 44 63
Num Comparisons: 0
Sorted array: 1 6 9 16 25 26 28 38 42 44 50 54 60 63 69
```



```
Input Params
============
How many entries? 20000
Which sort [m,i,q,r,all]? all

merge sort
==========
Num Comparisons: 521698


insertion sort
==============
Num Comparisons: 199970001


quicksort
=========
Num Comparisons: 81987


radix sort
==========
Num Comparisons: 0
```

II. Program Specifics, Back-end perspective

- Program file must be *SortCompare.java*.
- Error catching is NOT required. You do not have to catch if a user specifies a negative count of entries, or inputs a letter, or provides a sort option that is not ***m***, ***i***, ***q***, ***r*** nor ***all***.
- Your program must contain the following 4 functions (indicated EXACT names and parameters):

  ```
  mergeSort(int[] array)
  quickSort(int[] array)
  insertionSort(int[] array)
  radixSort(int[] array)
  ```

  Whether or not those functions have `void` return values or whether they return `int[]`, or whether they are `public` or `private`, is up to you.
- The `mergeSort` and `quickSort` implementations must be ***O(n log n)***. In other words, they must be the recursive versions.
- For each of the sorts that you implement, tally the count of comparisons that are done between entries of the array as it is sorted. For example, for quick sort, you'd tally the number of times that `array[j] < inputArr[j-1]` is checked. For quick sort, you'd tally the number of times that `A[j]` is compared to `x`, because `x` is referring to `A[r]`, etc.
- If you rely on source code from an external resource (the textbooks, the internet, etc.), you MUST cite that resource(s) in a comment portion at the top of your .java file. Under no circumstances is it allowed that you cut-and-paste entire blocks of code. Doing so will constitute a violation of the university plagiarism policy and will be grounds for an F in the course and reporting to the dean of students.

III. Hints and Notes

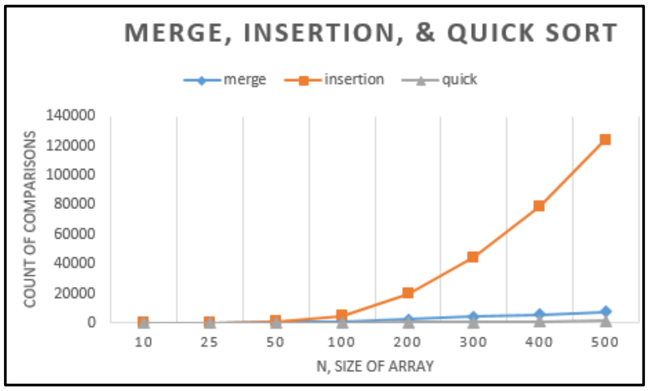
- Depending on how you implement each of the sorts, your count of comparisons, even for identical input arrays, might be slightly different than the counts your peers might be getting. However, the relative counts between insertion sort ***O(n^2)*** and quick sort ***O(n log n)***, for example, should differ greatly and clearly demonstrate relative run-times on input of large arrays.
- To ensure that the ***all*** option works as intended, you'll need to rely on a deep copy of the randomly generated array to make multiple identical copies.
- Start small. Write a stand-alone program that implements a function that performs quick sort on the array [3,6,2,4]. Once that is working, add the feature to prompt the user to specify the size of the array. When you've gotten that to work, do the same for the other sorts. Once done, then combine all of those separate functions in your multiple programs into a single java program, which also includes a switch/case statement for the sorts, including the all option.

IV. Plotting and performance assessment

- For the insertion, quick, and merge sorts, run each of them on successively larger arrays, and tally the count of comparisons made as a function of *n*, the array size. Vary *n* from 10 to 500. Create a table. For example:

***n*, array size**	**Merge sort**	**Insertion sort**	**Quick sort**
10	44	36	15
25	174	276	47
50	448	1176	104
100	1100	4851	237
200	2564	19701	520
500	4202	44551	815

- Using the software of your choice, create a plot of the values you have tallied. Be sure to label both the x and y axes, provide a title, and make sure that merge, insertion, and quick sort are all represented. A sample plot is shown on the right.



IV. Submission

Submit your .java file, as well as a PDF document with your tally counts and plot, via Canvas.

V. Rubric

Writeup	
Tally table of comparison counts for quicksort, merge sort, and insertion sort.	+5 points
Plot of tallied numbers, including axis labels, title.	+5 points

You earn points for implementing correctly the asked-for functions and overall program such that it works as intended (correctness) and so that the program runs in a reasonable amount of time (efficiency). Points are deducted for errors in commenting, style, etc. (clarity).

Code : Correctness	
Quick sort, merge sort, insertion sort, and LSD radix sort are coded with the requested function signatures (1 point each), and they correctly sort the integers in an input array (4 points each).	+20 points
Prompt user for number of integers desired (1 point), rely on random number generator to populate the array (3 points), and prompt for type of sort to run (***m***, ***i***, ***q***, and ***r***) as well as ***all*** (1 point).	+5 points
When the ***all*** option is specified, all four sorts are invoked, whose input is the same array.	+3 points
If *n<20*, pre-sorted and sorted array(s) are displayed, else the pre- and sorted arrays are not displayed.	+2 points
Each invocation of a sort correctly tallies the count of comparisons made.	+2 points

Code : Efficiency	
Recursive versions of quick sort and merge sort are coded. Each runs ***O(n log n)*** (3 points each).	+6 points
Insertion sort is coded in-place, and runs ***O(n^2).***	+2 points

Code : Clarity, deductions	
Inadequate commenting: your code should have a block comment at the top of the file that specifies the author, date, and program's purpose. Each function should be accompanied by a brief comment, and precede code sections in a function by a comment that specifies what that section does. Code sections are logical chunks of code, usually 5-10 lines. Your code should be compilable from the command line using *javac* and/or compiled using a standard IDE. If special libraries, or compile flags are needed, state that in the comment block at the top of the file.	-2 points
Inadequate variable and function names: variables should be descriptive (`a` is bad variable name, but `anArray` is fine); CS standard one-letter variables such as `i`, `j`, `k`, etc. for loop iterator variables are fine.	-2 points
Inadequate style: each function should be written concisely, and not be too lengthy (number of lines of code); if a function is 100+ lines, it should be broken up into separate functions. Alternatively, a function should not be so terse that it is cryptic.	-2 points