CS 211 – Data Structures
Comparing "Slow" and "Fast" Sort Algorithms

For this lab exercise, you MUST work in teams in the lab! The point here is to have teams discuss the lab and offer each other support in making sure the IDE of your choice is working. The teams you had working together last week for the Week 05 Lab are OK for this lab as well.

Lab Exercise

Today's lab is a continuation and extension of last week's lab comparing sorting algorithms.
The sorts involved are **Bubble Sort**, **Insertion Sort**, **Selection Sort**, **Merge Sort**, and **Quicksort**.

Last week, you added code for the **non-recursive** sorts (Bubble, Selection, Insertion) and counted the number of comparisons and moves made between array elements. This time, we're going to do the same for the **recursive** sorts (Merge, Quick) and we're going to add a new counter, counting the number of recursive function calls the recursive sorts do during their execution.

Global scope **long int** variables named **comps**, **moves**, and **calls** are already added to the code. They are global in scope so they can be used within different functions without needing to be passed as parameters. When a sort begins, all those variables are set to **0**, and are to be incremented whenever one of those actions occurs. Increment the appropriate variable whenever one of these actions occurs.

The supplied code has been modified from last week's lab, so download the supplied code! **All the work from last week's lab is already done on the non-recursive sorts** – you can examine it to see how to proceed with the new recursive sorts. But **please do leave the "slow" sorts code unchanged!**

Your first task is to add the code necessary to keep track of **comps**, **moves**, and **calls** in the **"fast" sorts**.

- Increment **comps** whenever any two array element values are compared to each other
  (This includes two values being compared in the extra space allocated for Merge Sort!)
- Whenever the **swap()** function is called, **moves** is already incremented by 3, do DON'T change the **comps** value when **swap()** is called!
- Increment **moves** whenever any array element is copied to or from another location in the data array or in temp storage.
- Increment **calls** whenever a function calls itself **recursively** in Merge Sort or Quicksort
- NOTE THAT in the helper function **findPivot** used by Quicksort, the correct places to increment **comps** is already in the code, but commented out. UNCOMMENT all the occurrences of **comps++** !

Of course, the non-recursive sorts have no recursive function **calls**, so those values will remain 0 in Bubble, Selection, and Insertion.

The **main.cpp** file has all the code needed to run the carious scenarios for the five sorting algorithms.

Your second task is to observe and interpret the stats of the counters and write up your team's findings in a report named **Week06Lab.pdf** (a PDF document).

You can address the following questions, just like you did last week with the "slow" sorts:

- Overall, do you think one of the sorts performs the best overall? If not one sort is the clear winner, then which sort would you **least** want to use overall?
- Which sort worked best in each scenario? Which sort did the worst?
- Did the "efficient" O(n log n) recursive sorts always work better? When did they do better? When did they do worse (if that happened at all)?
- Try changing the number of items in the array – this can be done by changing the **ARRAY_SIZE** variable in **main.cpp**. (I recommend also keeping the **MAX_VALUE** to be 10x the **ARRAY_SIZE**). How does making the array bigger/smaller affect the relative results of the sorts? How do they measure up with 10 items in the array? 100? 1000? 20,000? (**Warning**: setting ARRAY_SIZE to values larger than 20,000 means you can go get a cup of tea or maybe even lunch. And the folks who run CS50 might not like it if we hog all the CPU time!) Do the number of comps and/or moves in Merge sort and Quicksort seem to roughly correspond to an O(n log n) sort? Do the O(n log n) sorts get noticeably better relative to the O(n^2) sorts as the size of the array increases?
- Feel free to add your observations about these sorting algorithm comparisons.

Submit **both** your modified **fastSortCompare.cpp** file and your **Week06Lab.pdf** report file.