CCPS616 Lab 1 - Sorting and Problem Size

Preamble

In this lab you will write a C++ program that implements and tests three sorting algorithms. You must implement these yourself without the help of any external 3rd party libraries. Your lab should be completed in a single cpp file called **sort.cpp**, and should compile at the command line in **replit** very simply as follows:

```
g++ -o lab1 sort.cpp lab1.cpp
```

Where lab1.cpp is provided and contains a main() function that tests your code. Please do not submit labs that don't compile. I will not debug syntax errors when marking.

Lab Description

We've done a lot of hand-waving about O(n^2) sorting algorithms being faster than O(nlogn) sorting algorithms when the problem gets below a certain size. In this lab, we will put such claims to the test. You will implement **three** sorting algorithms:

First, implement standard, out-of-the-box **Selection sort** and **Merge sort**. Each should be in their own function(s). An unsorted array goes in and gets sorted. Sorting should be done on the *original* array (do not return a new one). Implementation beyond that is up to you.

Next, implement a modified Merge sort – ***MergeSel*** – that cuts to Selection sort when the sub-arrays get small enough. I.e., instead of recursively calling Merge sort all the way down to a subarray of a single element, call Selection sort instead when the subarrays are < **ax** elements. The ideal value for **ax** will be determined by you empirically.

You might wonder why we would use Merge sort and Selection sort, instead of Quicksort and Insertion sort. Merge and Selection sorts have the same best/average/worst case complexity (nlogn, n^2 respectively). We use them so our results will be as predictable as possible and not influenced by the initial "sortedness" of our input array.

Testing

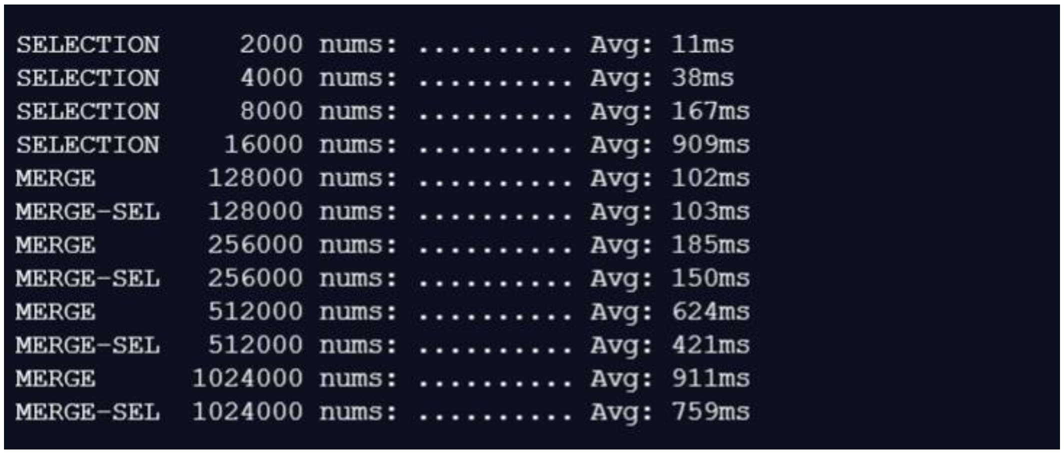
The **`main()`** function provided in lab1.cpp will generate integer arrays of increasing size containing **n** elements whose values are randomly distributed between **1** and **4n** and call your sorting algorithms on those arrays.

Check out the tester code in lab1.cpp, but **do not modify it**. Unless you think there's a bug in there, in which case please let me know. All you must do once your code is compiled using the command given above is run the executable. The tester does the rest.

When comparing Merge and MergeSel, we add the additional variable **ax** – the size at which MergeSel kicks to Selection sort. The values for **ax** are completely up to you, but I suggest starting at ax=3, 5, 7, 9, 11, etc. Your goal is to try and improve on Merge sort as much as possible. Have fun experimenting and follow the data!

Results

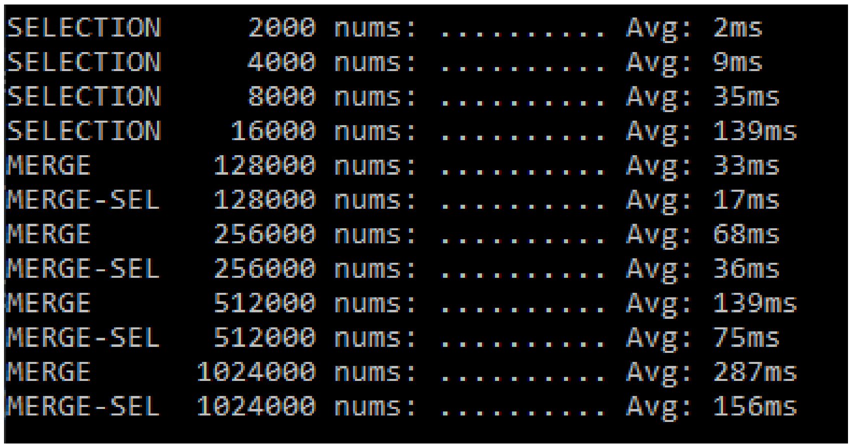
The tester will print your results, as well as the average time for each algorithm and each size. It will trip an assertion if your list is not properly sorted when the function returns. When I run the tester in replit on my own model solution, I get the following output:



```
SELECTION     2000 nums: .......... Avg: 11ms
SELECTION     4000 nums: .......... Avg: 38ms
SELECTION     8000 nums: .......... Avg: 167ms
SELECTION    16000 nums: .......... Avg: 909ms
MERGE       128000 nums: .......... Avg: 102ms
MERGE-SEL   128000 nums: .......... Avg: 103ms
MERGE       256000 nums: .......... Avg: 185ms
MERGE-SEL   256000 nums: .......... Avg: 150ms
MERGE       512000 nums: .......... Avg: 624ms
MERGE-SEL   512000 nums: .......... Avg: 421ms
MERGE      1024000 nums: .......... Avg: 911ms
MERGE-SEL  1024000 nums: .......... Avg: 759ms
```

Notes

`replit` can be very inconsistent when it comes to runtimes. Subsequent runtimes of the same sorting algorithm often vary by over 100%. I've tried to account for this in the tester by averaging over many trials, but you will likely still observe inconsistent results. Don't be afraid to rerun your code several times, or test locally on your own PC. When I test my code on my own PC, things are much more consistent (and MergeSel *dominates* Merge sort):



```
SELECTION     2000 nums: .......... Avg: 2ms
SELECTION     4000 nums: .......... Avg: 9ms
SELECTION     8000 nums: .......... Avg: 35ms
SELECTION    16000 nums: .......... Avg: 139ms
MERGE       128000 nums: .......... Avg: 33ms
MERGE-SEL   128000 nums: .......... Avg: 17ms
MERGE       256000 nums: .......... Avg: 68ms
MERGE-SEL   256000 nums: .......... Avg: 36ms
MERGE       512000 nums: .......... Avg: 139ms
MERGE-SEL   512000 nums: .......... Avg: 75ms
MERGE      1024000 nums: .......... Avg: 287ms
MERGE-SEL  1024000 nums: .......... Avg: 156ms
```

In addition, be sure to properly manage your memory. Memory leaks and inefficient use of very large arrays in C/C++ can also lead to runtime inconsistencies (or crashes).

Submission

Submit your source file (**sort.cpp**) on D2L. This lab must be submitted individually.