Jasque Saydyk

Professor Patrick Kelley

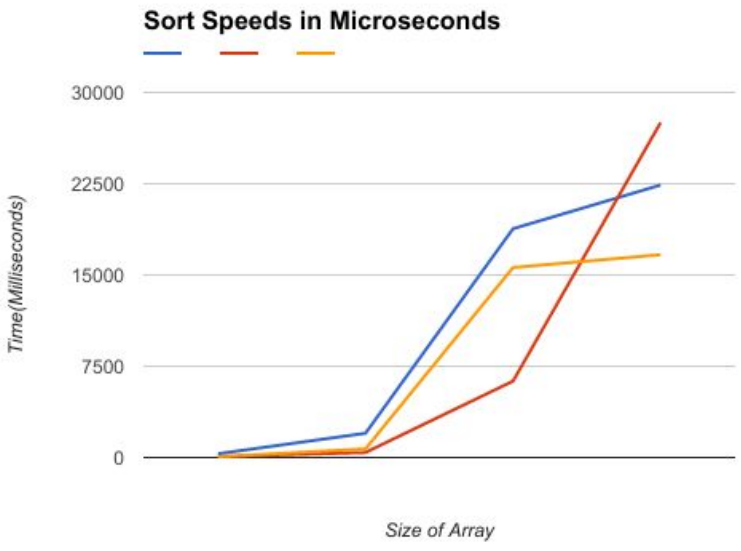
CS 249 Section 3140

1 March 2017

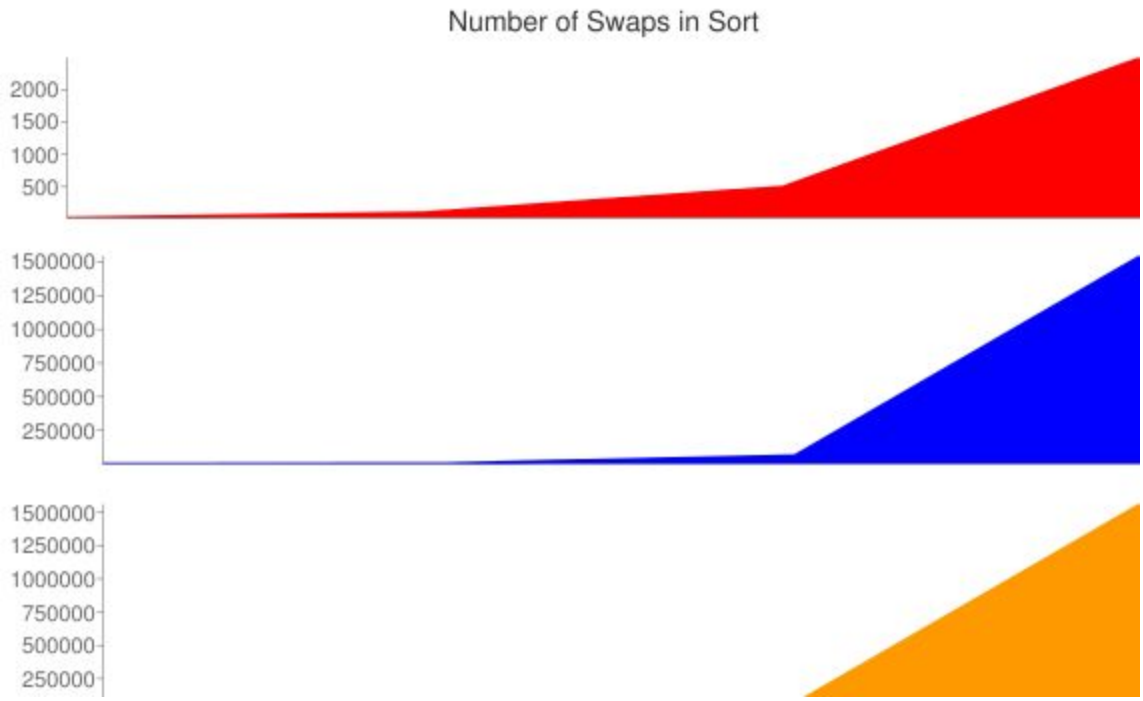
Project 1 - Simple Sorting

First this project, it was split into two primary sections, implementing the Bubble sort, Selection sort, and Insertion sort, then re-implementing those sorts with comparators. Beginning with the implementation of the basic sorts, there were no serious issues as they are relatively straightforward. Theoretically, these algorithms are $O(n^2)$, and when I timed my sorts, this seems to be the case. Below is a graph of the simple sort speeds in milliseconds, with each like being the average of five runs of each sort. The blue line is the Bubble sort, the orange line is the Insertion sort, and the red line is the Selection sort. To gather the time, I didn't use the Timer class as I could not figure out how to make it work. Instead, I used System.nanoTime() to get the time when I start the sort, then I get the time again when the sort ends, the subtract the two to get a close approximation as for how long the sort ran.

Thus we can see how these sorts practically compare to one another. All three of the sorts do generally tend towards being similar to a typically exponential curve, it is obvious that the orange line, the Insertion sort, is performing significantly better on large arrays it's fellow sorts. The Selection sort performs better than both sorts on array's less than 500 index's large, but sees a significant spike in computation time when it faces extremely large arrays. All around, the Bubble sort fared the worst, taking more time then every other sort at the tested array sizes.



As for the number of swaps each sort does, the red Selection sort does significantly the least, which is due to the nature of the algorithm searching for the lowest number, then making the swap. The Insertion and the Bubble sort both do a similar number of swaps, which exponentially increases with the size of the array.



If you would like to view the raw numbers, the screenshot to the right shows the typically console output for the tests on the sorts.

As for the Comparators, it took me a while to figure out how to implement them, as I thought way to hard about the problem. This is because I initially interpreted the answer to the problem as allowing the user to choose between different compareTo()s by overloading the compareTo. It took me a while to realize that the true answer was far more mundane, as the compareTo()s only had to compare the two given objects cascading down the specification.

Also the Artist comparator had me stumped for a while, but it dawned on me that the requirements state that this compareTo should sort the artist name alphabetically. So to any normal human, this means that A is greater than B, because alphabetically A is at the top, thus the greatest, and Z is at the bottom, the lowest. We sort from A to Z, not Z to A.

However, in Java, if you compare the char A and the char B, you'll find that B is greater, which also makes sense if you think of the letters as hexadecimals replacing the letters, this still makes sense, as A is 10 and B is 11, thus B is bigger.

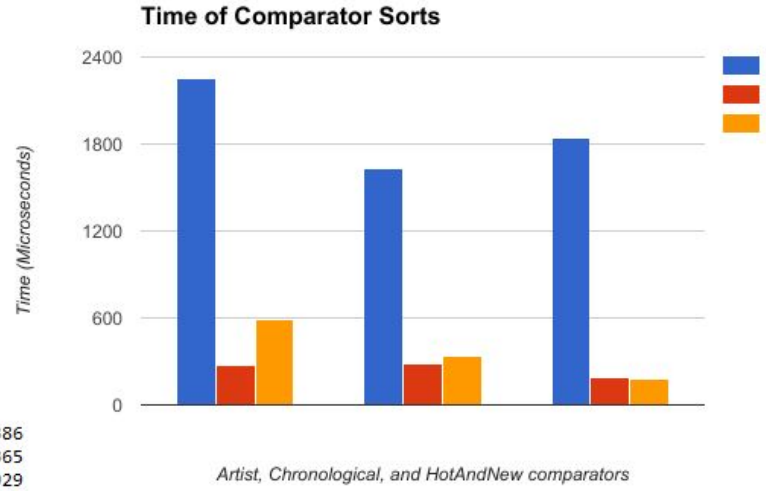
So what am I getting at here? Well, we have two different systems. To normal humans,if you ask, "Which is greater alphabetically, A or B", they would say A, and thus expect a compareTo function sorting alphabetically to return a number greater than 0, as A is greater. However, the programmer of the test for this class decided to go with the Java/char compare mindset, thus a return of less than 0.

For this assignment, beyond having to figure this out, the difference between the two doesn't mean much. However if an outside programmer were to use this, and got a return of less than 0 for the question of "Which is greater alphabetically, A or B", it may drive them a little bonkers and add bugs into code. Thus I believe this is a semantic bug that ought to be fixed, that in the ComparatorTests, flip the comp.compare(a,b)<0; to comp.compare(a,b)>0;

Aside from that, the rest of it was straight-forward. In retrospect, I should have spent more time developing more rigorous tests for the comparators to ensure that every part of them work as intended, and if any part of the program were to malfunction, it would be those parts.

Comparing the simple sorts with the comparators, we get slightly different results. The below Bar graph has three groups for the Artist, Chronological, and HotAndNew comparators, respectively. The color of the bars correspond with the different sorts, blue for Bubble sort, red for Selection sort, and orange for Insertion sort.

With this, we can see that the Bubble sort is by far the worst when using a comparator, as it takes the most time to run. As for the Selection and Insertion sorts, they are pretty close together in operating speed, with the Selection sort being ever so slightly faster than the Insertion sort, probably due to the small number of swaps it has to make, which can also be seen in the “Number of Swaps in Sort” chart, which follows the same form as the “Time of Comparator Sorts” chart.




```
Bubble Comparator Sort Test
artist        - Swaps: 461  Milliseconds: 2  Microseconds: 2386
chronological - Swaps: 484  Milliseconds: 2  Microseconds: 2365
hotAndNew     - Swaps: 408  Milliseconds: 1  Microseconds: 1929

Selection Comparator Sort Test
artist        - Swaps: 39  Milliseconds: 0  Microseconds: 272
chronological - Swaps: 38  Milliseconds: 0  Microseconds: 362
hotAndNew     - Swaps: 38  Milliseconds: 0  Microseconds: 472

Insertion Comparator Sort Test
artist        - Swaps: 461  Milliseconds: 1  Microseconds: 1219
chronological - Swaps: 484  Milliseconds: 0  Microseconds: 160
hotAndNew     - Swaps: 408  Milliseconds: 0  Microseconds: 184
```