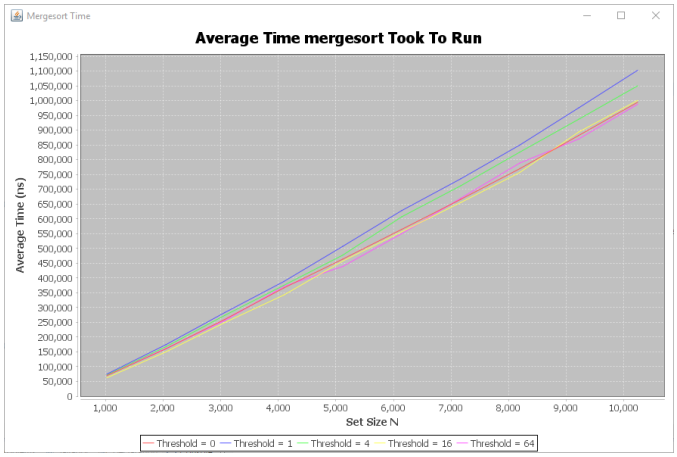
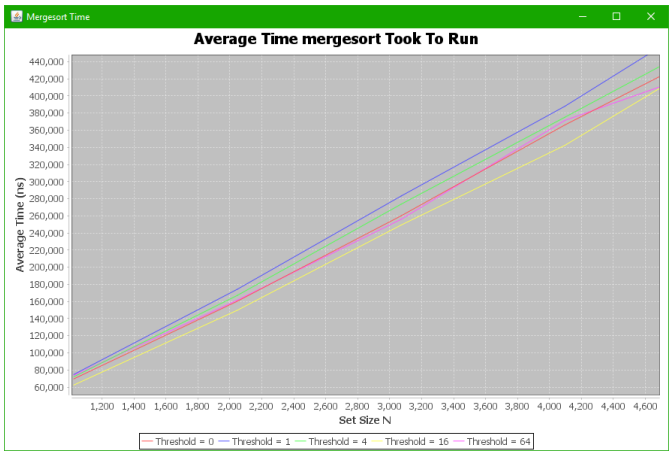
1. My partner was Ryan Bolander, and I submitted the source code.
2. If I'm able to, I would work with Ryan again.
3. Some of the pro's of pair programming was that when I make a mistake my partner catches it. Another pro was that both of us needed to work in the Cade lab, meaning we had less distractions. A way that I can be a better partner is to not take main control of the program, and to somehow be free up my time so that I can work in the afternoon like most everyone was looking for.
4. The five thresholds I chose were 0, 1, 4, 16, and 64. The full merge sort should be 0.

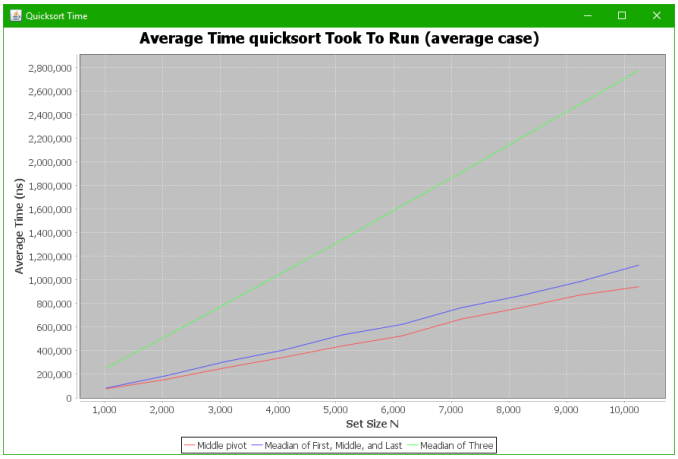


I wasn't quite happy with this graph on account of how close things were, but the only way I could get a better one is to drastically increase set sizes, and reduce iteration counts. That would leave me with a graph that is less accurate, slightly more prone to asymptotic behavior, and takes more than an hour to generate. As for what threshold value is best, when you reach large set sizes, of the five values I chose, it's a toss-up between 0, 16, and 64.



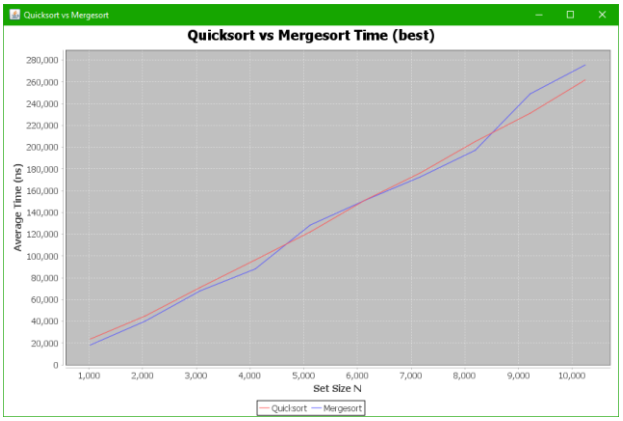
If I zoom in closer to the graph at the beginning you can see that a threshold of 16 is the better value, but this only firmly holds for small sizes. At the very least threshold 16 stays near the final three quickest functions on large set sizes. The red line shows a full merge sort and appears to closely follow the thresholds of 16 and 64. For that reason I will (not super confidently) choose threshold 16 as the best, but only marginally.

5. Of the three methods that I chose two of them were shown in class. The middle item. The median of the start middle and end of the list. “Median of Three” which is similar to the median of the start middle and end, but uses random indices to find the three elements.

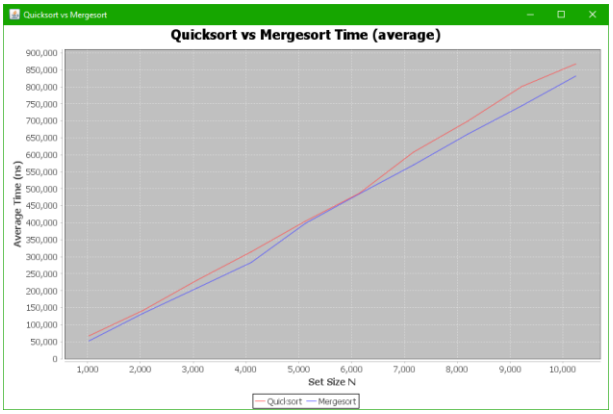


Of the average case scenarios the middle index was the best to take. Median of first, middle, and last is close.

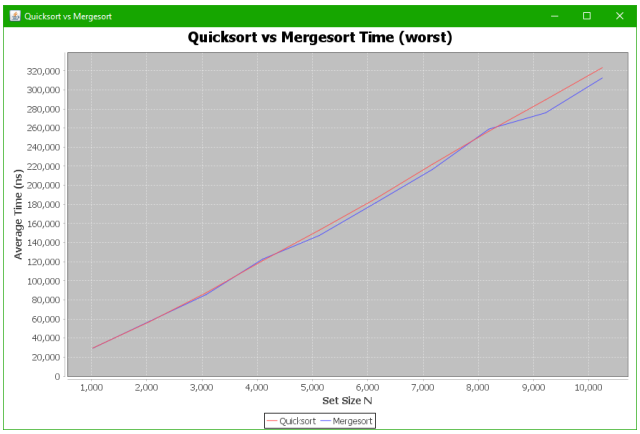
6. The best threshold for mergesort to switch to insertionsort was when the “sub-list’s” got to size 16 or less. The best pivot selection for quicksort was to just take the middle item.



They are relatively similar, which is to be expected. They are both O(Nlog(N)) functions. Mergesort takes a bit longer on larger set sizes due to allocation of a temporary array of size N.



Mergesort beats out quicksort on average case lists. This is probably because quicksort now has to do more computations.



Mergesort still beats out quicksort on worst case scenarios. It appears that mergesort is the better option, given that ram space isn't a problem as stated in class.

7. The running times of my methods don't exactly exhibit the growth rates I expected to see. I believe this is because of the relatively small set sizes I was using to avoid having to run timing code for more than an hour per graph. I expected both functions to perform as O(Nlog(N)) but they appear to run at O(N) which any graph would if zoomed in enough.
8. The time I spent programming was under 4 hours. The time I spent generating graphs for this paper was about 2 hours. Overall about 6 hours.

Fun fact – While I was waiting for graphs to be generated I would solve a Rubiks cube. I could solve it about 3 times per graph.