Christian Hansen
U0621884

Analysis Document

1. Who is your programming partner? Which of you submitted the source code of your program?

Jared Nielson. He submitted the code.

2. Evaluate your programming partner. Do you plan to work with this person again?

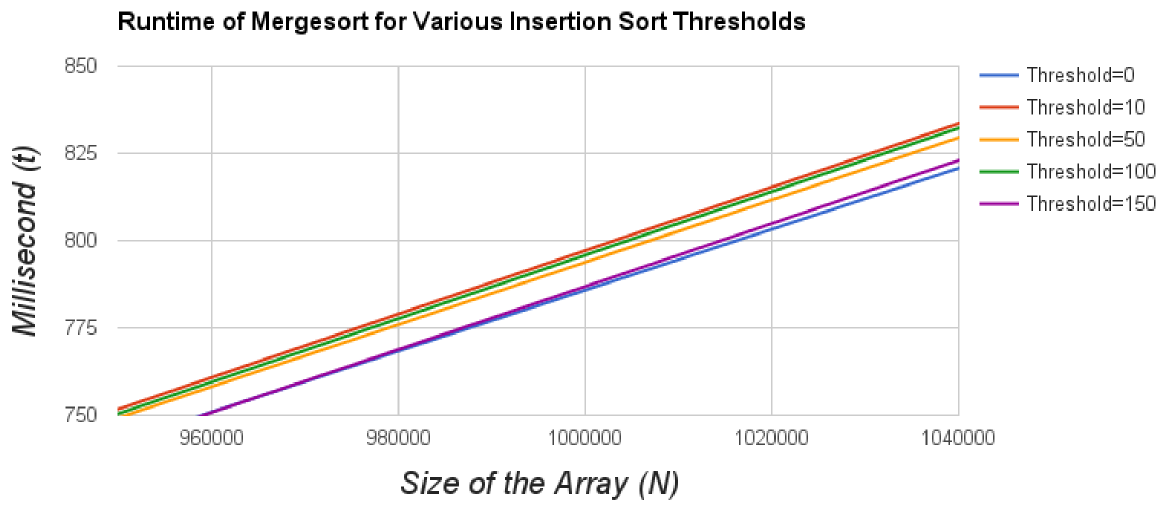
Jared is a smart guy. I have loved working with him these past three assignments. I would work with him again if I could, but it is my understanding that for the next assignment that I will have to switch partners. We have worked together for all three of these first partner assignments. I have learned a lot from him and we have been very successful.

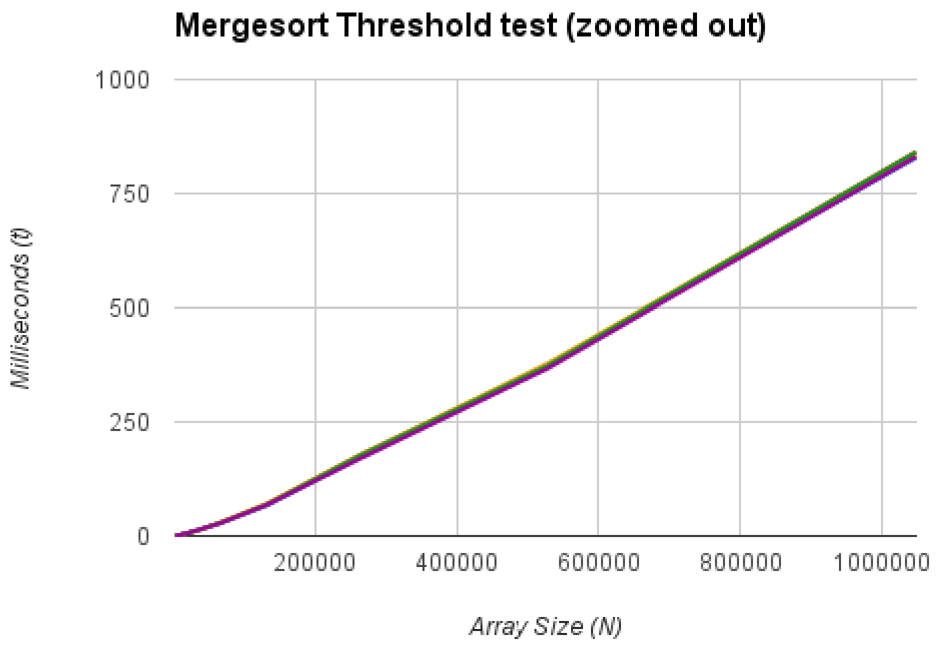
3. Evaluate the pros and cons of the pair programming you've done so far. What did you like, what didn't work out so well? You'll be asked to pair on three more of the remaining seven assignments. How can you be a better partner for those assignments?

Pair programming has gone well. There have been many things that I see as positives. Debugging is better with four eyes. Reasoning about problems can be better. Everyone approaches problems with a different perspective which can help with finding the right solutions. The main negative I feel is that when I do an assignment on my own, I know that I understand everything or at least it helps me to feel that I do. When pair programming, I worry about whether I have learned everything about the concept or is it just that together we can do it, but it is probably better to do some of these hard assignments pair programming than solo. I feel I should read in the book more and study the slides more. This will help me come into the assignments with a better understanding of the concepts and I will be better prepared for the assignment.

4. Mergesort Threshold Experiment: Determine the best threshold value for which mergesort switches over to insertion sort. Your list sizes should cover a range of input sizes to make meaningful plots, and should be large enough to capture accurate running times. To ensure a fair comparison, use the same set of permuted-order lists for each threshold value. Keep in mind that you can't resort the same ArrayList over and over, as the second time the order will have changed. Create an initial input and copy it to a temporary ArrayList for each test (but make sure you subtract the copy time from your timing results!). Use the timing techniques demonstrated in Lab 1 and be sure to choose a large enough value of timesToLoop to get a reasonable average of running times. Note that the best threshold value may be a constant value or a fraction of the list size.
Plot the running times of your threshold mergesort for five different threshold values on permuted-order lists (one line for each threshold value). In the five different threshold values, be sure to include the threshold value that simulates a full mergesort, i.e., never switching to insertion sort (and identify that line as such in your plot).

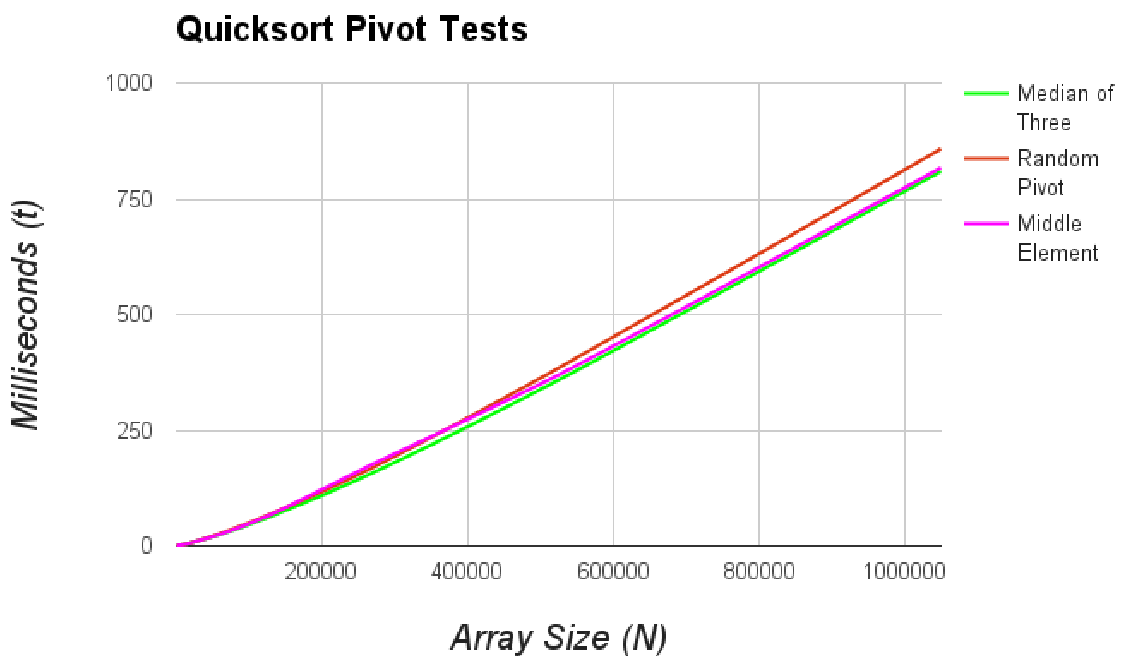
Comparing different running times for merge sort with varying thresholds shows that the outcomes are generally very similar, of course once the threshold gets unreasonably high, the runtime would begin to have a Big O notation of $O(n^2)$ as it would be insertion sort. I have a zoomed in view here of the different graphs otherwise it is difficult to even see a difference. Not using insertion sort at all performs the best at some sizes. When using insertion sort, a threshold of 10 performed the best at most sizes, though not all. This zoomed in part of the graph shows that a threshold of 150 performed better in a few cases. The zoomed out does not have a legend, but as we can see that looking at it zoomed out it is very difficult to see the separate lines.





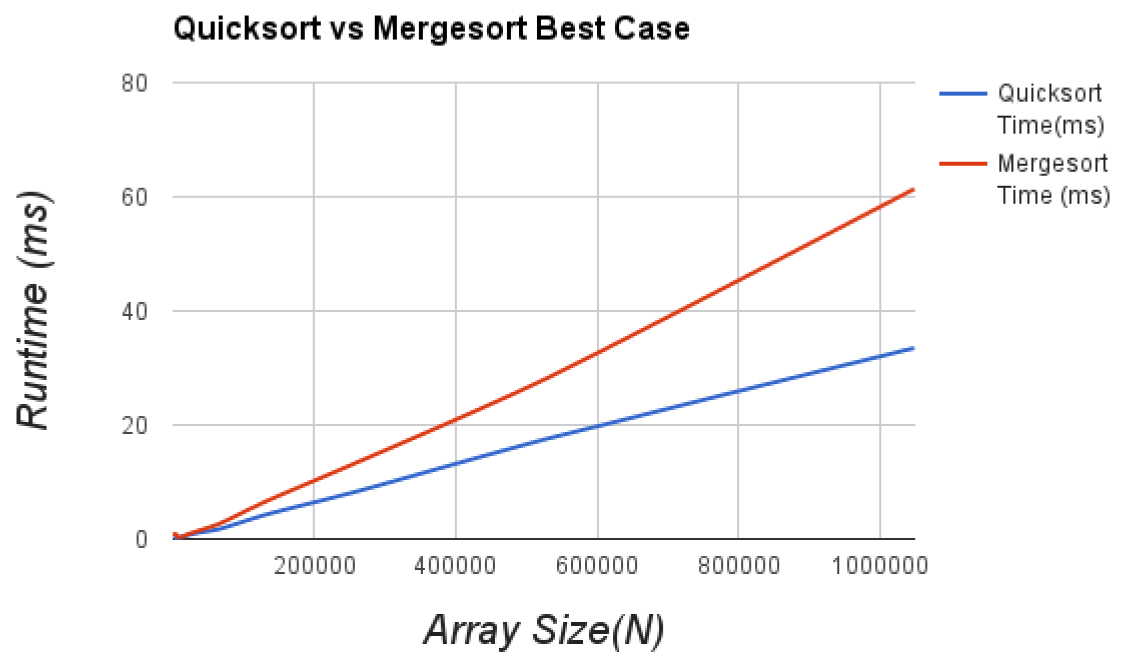
5. Quicksort Pivot Experiment: Determine the best pivot-choosing strategy for quicksot. (As in #3, use large list sizes, the same set of permuted-order lists for each strategy, and the timing techniques demonstrated in Lab 1.) Plot the running times of your quicksort for three different pivot-choosing strategies on permuted-order lists (one line for each strategy).

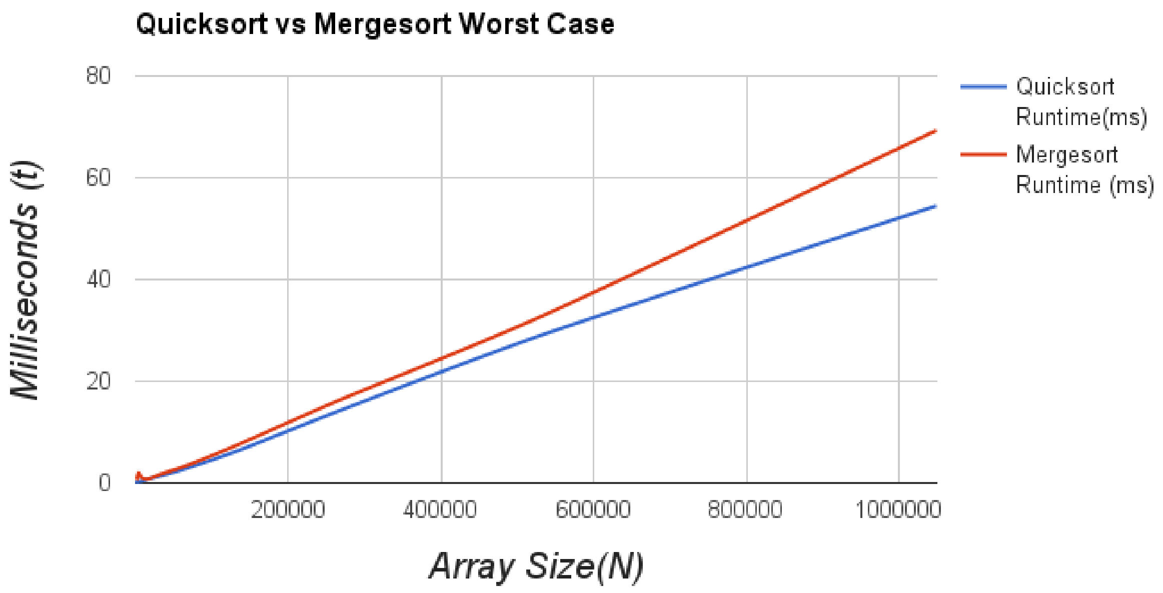
The pivot choosing strategy of Median of Three performed the best. It is just slightly lower than the middle element. This is what I expected to happen as this method analyzes three data points instead of just one. The random pivot performing the worst is not surprising as it could select the beginning element or the last. The middle element would be a good option for a list that is close to being sorted. In conclusion Median of Three is the best performing pivot chooser.

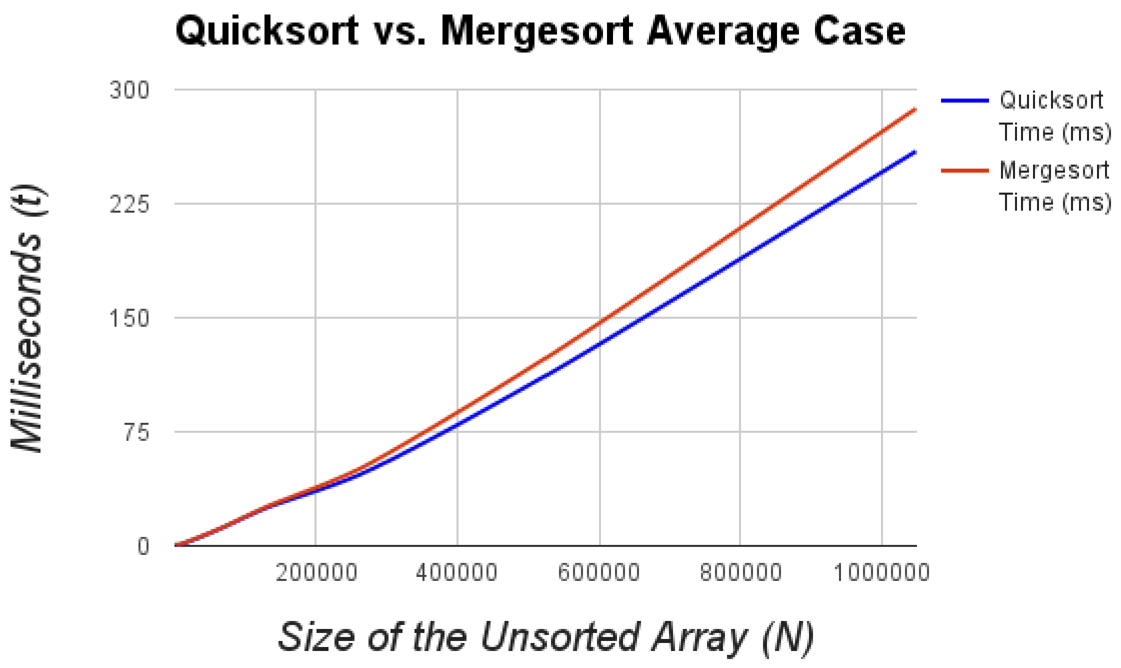


6. Mergesort vs. Quicksort Experiment: Determine the best sorting algorithm for each of the three categories of lists (best-, average-, and worst-case). For the mergesort, use the threshold value that you determined to be the best. For the quicksort, use the pivot-choosing strategy that you determined to be the best. Note that the best pivot strategy on permuted lists may lead to O(N^2) performance on best/worst case lists. If this is the case, use a different pivot for this part. As in #3, use large list sizes, the same list sizes for each category and sort, and the timing techniques demonstrated in Lab 1. Plot the running times of your sorts for the three categories of lists. You may plot all six lines at once or create three plots (one for each category of lists).

For this experiment of Mergesort and Quicksort, we chose a threshold of 10 for Mergesort and the Quicksort method of finding the pivot is the median of three method. Quick sort performed the best in all three scenarios. Quicksort performed significantly better in best and worst case scenarios. In the average case, it performed better but not as drastically. All three show Big O notation of O(NLog(N))


Quicksort vs Mergesort Best Case
Runtime (ms)
Array Size(N)
0
20
40
60
80
200000
400000
600000
800000
1000000
Quicksort Time(ms)
Mergesort Time (ms)


Quicksort vs Mergesort Worst Case
Milliseconds (t)
Array Size(N)
0
20
40
60
80
200000
400000
600000
800000
1000000
Quicksort Runtime(ms)
Mergesort Runtime (ms)



<u>7. Do the actual running times of your sorting methods exhibit the growth rates you expected to see? Why or why not? Please be thorough in this explanation.</u>

The methods had running times that I expected to see. Both sorting methods show Big O notation of O(NLog(N)), which was expected since both sorting methods are sub-quadratic. With the mergesort method, the splitting of the array down to 1 item chunks is the part of the code that produces the O(log(n)) behavior. Later the part in which the code then combines the arrays together is what produces the O(n) behavior. They are nested together so they create the Big O notation of O(NLog(N)). For quicksort, breaking down the array into sub-arrays accounts for the log(n) behavior. With quicksort when the items are then swapped to either side of the pivot, this produces the Big O notation of O(N). These two together produce the Big O notation for quicksort of O(NLog(N)). I feel the running times from our tests of both sorting methods match this predicted behavior.

<u>8. How many hours did you spend on this assignment?</u>

We spent 12 hours on this assignment.