Jana Klopsch
U0854469
CS 2420

Assignment 05 Analysis

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

My programming partner was Alex Henabray, and I submitted the source code.

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

I enjoyed working with Alex the past three assignments we have had. He was very easy to work with when writing and testing our programs. The only issue we really seemed to have was when one person understood how to implement a method, and the other did not. This only happened a few times, but when it did, it was difficult to move forward and continue working on the rest of the program until both people understood that method. Overall, he was great to work with, and I would consider working with him for the final project.

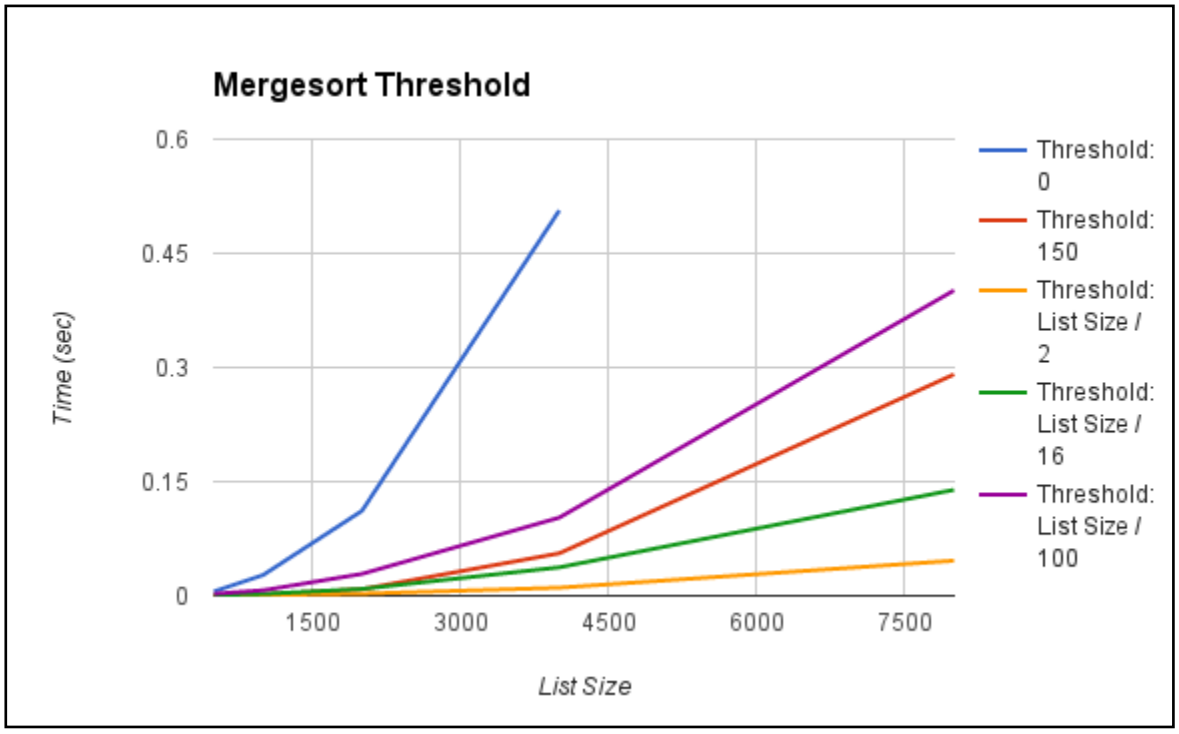
3. Evaluate the pros and cons of the pair programming you've done so far. What did you like, what didn't work out so well? You'll be asked to pair on three more of the remaining seven assignments. How can you be a better partner for those assignments?

For pair programming, I see many pros to it: learning different techniques from someone, helping someone understand a concept, learning a concept from someone else, ensuring that the program isn't overcomplicated, etc. For me, the biggest con has been scheduling conflicts. We both had different class schedules, so we could only go to very limited office hours, and it was difficult to find times where we could meet and get the assignments done. To be a better partner, I could read over the assignment more thoroughly before beginning, so that I am more prepared when starting the program.

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).

For timing mergesort with varying thresholds, the data resulted in the best threshold being at (0.5)*size, meaning the threshold was implemented at half of the input array's size.

Surprisingly, the threshold that performed worse than any other was when the threshold was equal to zero. With this threshold, insertion sort was never applied to the mergesort algorithm, as the subarrays were never as small, or smaller than zero. The threshold that performed the best, was when the threshold was set to be half of the input array's size. That meant insertion sort was applied almost immediately after the first call to mergesort. Because mergesort has the complexity O(N log N), and insertion sort has the complexity O(N^2), we expect mergesort to outperform insertion sort for large values of N. In looking at the data collected, it is possible that the data sizes are not large enough to see this behavior. Another possibility (that seems more likely) is that there are extra steps in the mergesort algorithm, that are adding to the time it takes to apply mergesort. This would most likely come from the operations performed on the array lists, and calling the array list add and set methods.



**Note: the time for size = 8000, threshold:0 is not included in the graph, so that the other threshold lines are still easy to read. This time value is listed in the table below.*

Mergesort Thresholds

List Size	Threshold: 0	Threshold: 150	Threshold: List Size / 2	Threshold: List Size /16	Threshold: List Size /100
500	5.64E-03	3.94E-04	2.40E-04	7.80E-04	0.002541014
1000	2.74E-02	1.54E-03	7.91E-04	2.42E-03	0.00721064
2000	0.111620645	0.008627657	2.72E-03	0.008982441	0.02873885
4000	0.506383433	0.055806123	0.010624946	0.037369853	0.102701152
8000	2.238911764	0.290914477	0.046142225	0.139216623	0.401274812

**Note: the values listed in the columns to the right of the list size columns are the time values collected in seconds, for each respective threshold.*

5. Quicksort Pivot Experiment: Determine the best pivot-choosing strategy for quicksort. (As in #4, 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).

Looking at the data collected, the best pivot choosing strategy for quicksort is median pivot. The median pivot was chosen by a method that took the first, last and middle objects in the array, sorted them, and then set the pivot to be the object that was placed in the middle of the three. Between the three different pivot choosing methods, this method would be expected to be the fastest, as the other two are selecting either the object at the center of the array (which could be anything, not necessarily an accurate median object to use), or just selecting an object in a random spot in the array (again, this object could be anything). So although the array isn't sorted, and in theory, the pivot could be chosen to be an extrema of the array, median pivot takes the median of three random objects in hopes of eliminating this from happening. For large list sizes, choosing the median pivot produced the fastest quicksort sorting times.

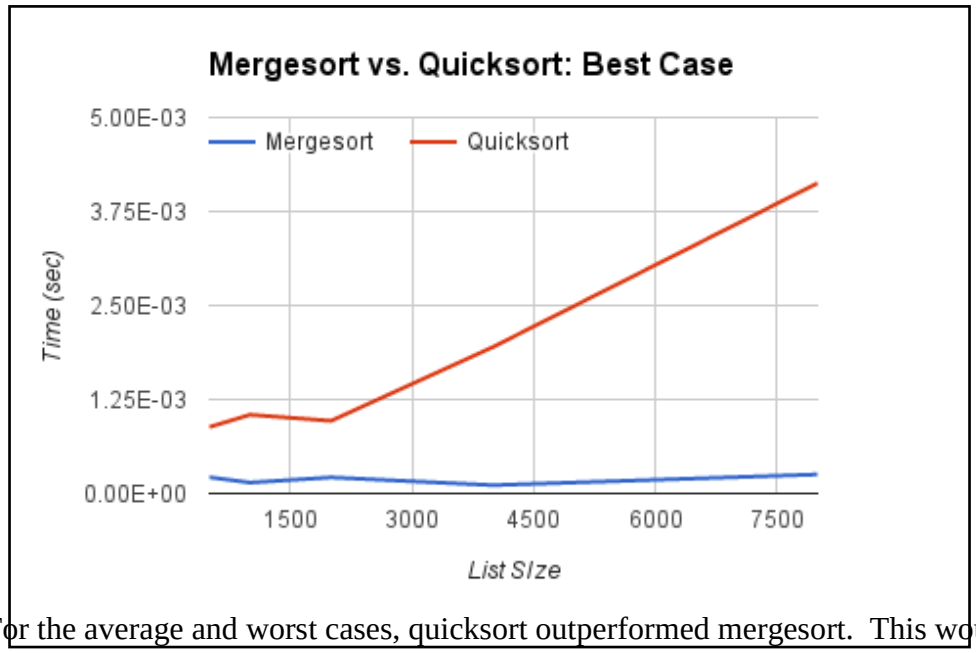
Quicksort Pivots

Quicksort Pivots			
List Size	Middle Pivot	Median Pivot	Random Pivot
2000	1.91E-03	3.43E-03	3.25E-03
4000	0.003551452	0.003660062	0.003907616
8000	0.007931522	0.00745591	0.008093295
16000	0.017631373	0.016049274	0.016776018
32000	0.036800116	0.032971515	0.035601327

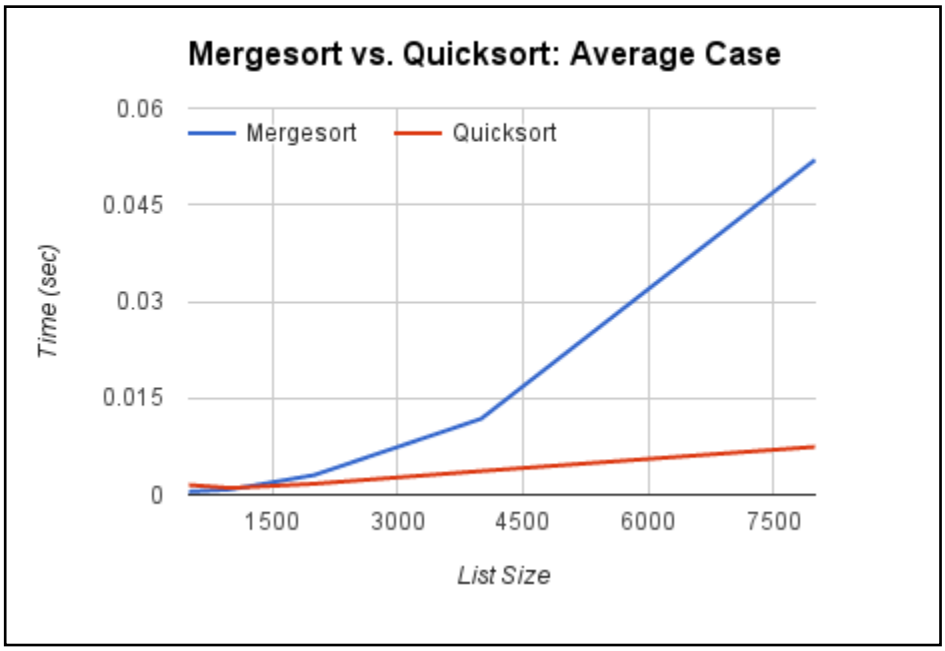
**Note: the values listed in the columns to the right of the list size columns are the time values collected in seconds, for each respective pivot.*

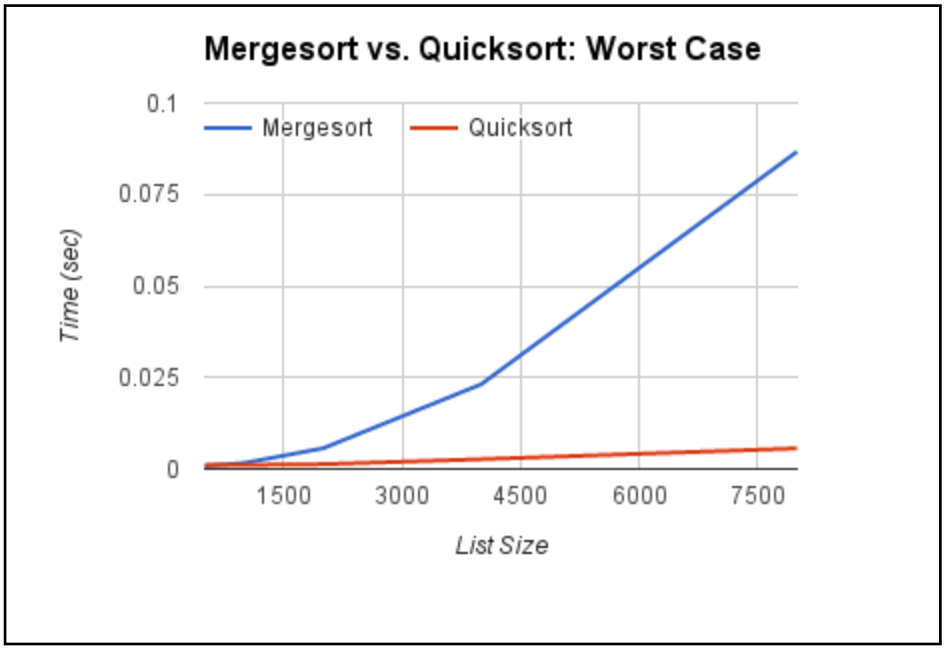
6. Mergesort vs. Quicksort Experiment: Determine the best sorting algorithm for each of the three categories of lists (best-, average-, and worst-case). For the mergesort, use the threshold value that you determined to be the best. For the quicksort, use the pivot-choosing strategy that you determined 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 #4, 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 the best case, mergesort outperformed quicksort. In looking at the behavior of each, mergesort seems to have more of a linear behavior, while quicksort seems to have quadratic, or subquadratic. Both of these behaviors make sense in this case, as the quicksort algorithm will have the same complexity for best case as it does for worst case. In mergesort, because the threshold chosen implements insertion sort so early on, it has the behavior of insertion sort in the best case, which is O(N).



For the average and worst cases, quicksort outperformed mergesort. This would be expected as mergesort applies insertion sort early in the algorithm, so we would expect to see more of the behavior of insertion sort which for the worst case is O(N^2). Quicksort is expected to have O(N log N) behavior, which although sub-quadratic, will outperform a quadratic function.





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.

The running times for quicksort exhibit the growth rates expected, however, the running times for mergesort did not exhibit the behavior expected. For quicksort, it was expected to run with a sub-quadratic growth rate, which it appeared to do in the experiments to select the best pivot, as well as the best, average, and worst case experiments. The quicksort algorithm uses recursion to sort the array with regard to the pivot, or partition. When the method partitions the array, it steps through the array at each item, which gives it linear behavior. However, because the array is smaller each time we partition it, it also has logarithmic behavior (due to the repeating halving principle). Together, this gives O(N log N) complexity.

Mergesort was expected to also have O(N log N) behavior. Looking at the experiments to determine the best threshold, as well as the experiments for best, average and worst case, mergesort appears to have quadratic behavior O(N^2). As stated before, I think this comes from errors within the merge algorithm, which in turn affect the implementation of insertion sort. Within the merge method, there are multiple ArrayList operations performed. These operations are time consuming, as they effect the entire ArrayList, which causes mergesort to not run as quickly as it should. This would account for the mergesort algorithm with threshold zero to be as slow as it was, because these errors were applied every time the array was merged. Thus, when the threshold was chosen so that insertion sort was applied very early in the algorithm, insertion sort was outperforming mergesort. Because this behavior was observed to be the fastest, it was chosen as the best threshold. Finally, in testing best, average and worst cases, mergesort was really showing insertion sort behavior. For the best case, insertion sort (within mergesort) can outperform quicksort, because it has the complexity O(N), rather than quicksort's complexity

O(N). However, for the average and worse cases, insertion sort (within mergesort) has complexity O(N^2), where quicksort has the better complexity O(N log N).

8. How many hours did you spend on this assignment?

10