Shahid Bilal Razzaq
U0996062
Partner: Nickolas Komarnitsky
Partner UID: u0717854
9/28/2016

Assignment 5
Analysis

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

My programming partner is Nickolas Komarnitsky. I will be the one submitting the code for this assignment.

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

Nick is a very talented programmer, and it was a unique experience working with him on this assignment. His natural ability to write near-perfect draft code was quite remarkable. He had a good attitude regarding the collaboration and was quick in responding to emails/text. When there were any concerns about the code, he was quick to perform a debug and isolate the problem before fixing it. His good nature and terrific programming skills make him a good pair programming partner.

Given the chance, I would work with him again, but we have already collaborated on two assignments, so future assignments will require a different partner.

3. Evaluate the pros and cons of the 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?

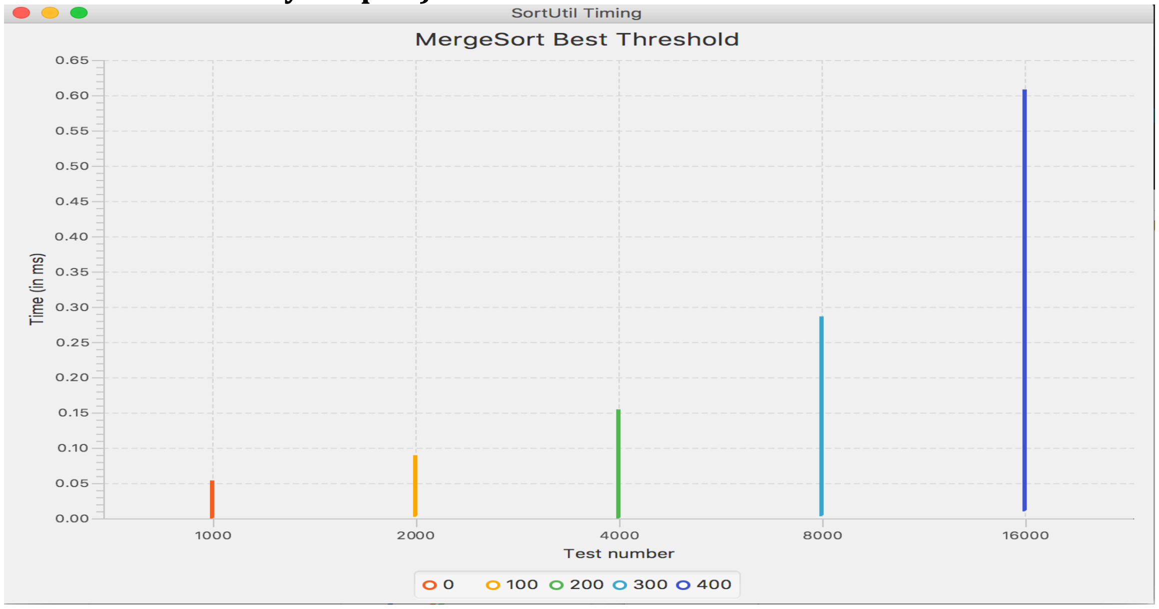
Pair programming can be challenging sometimes, especially when conflicts arise over the correct implementation of the code. Luckily this has not happened in my experience so far, but it is something that I notice with other paired partners. The most valuable part of paired programming, in my opinion, is the fact that you can collaborate and gain a second insight on your view of the coding logic. While I would consider this a pro, I know in many cases this can turn out to be a con as well, because different people may have a different approach to their code, and that can cause conflicts.

A more personal downside to paired programming is the fact that I like to have complete control over my code implementation, and with paired programming, that control needs to be shared, even though both partners are coding to achieve the same result.

Personally, I can become a better partner in paired assignments by reading ahead on the program specifications, getting obscure points clarified, and developing the logic needed before starting on the assignment with my partner. This will allow me to be bettered prepared, and spend my programming time more effectively.

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.

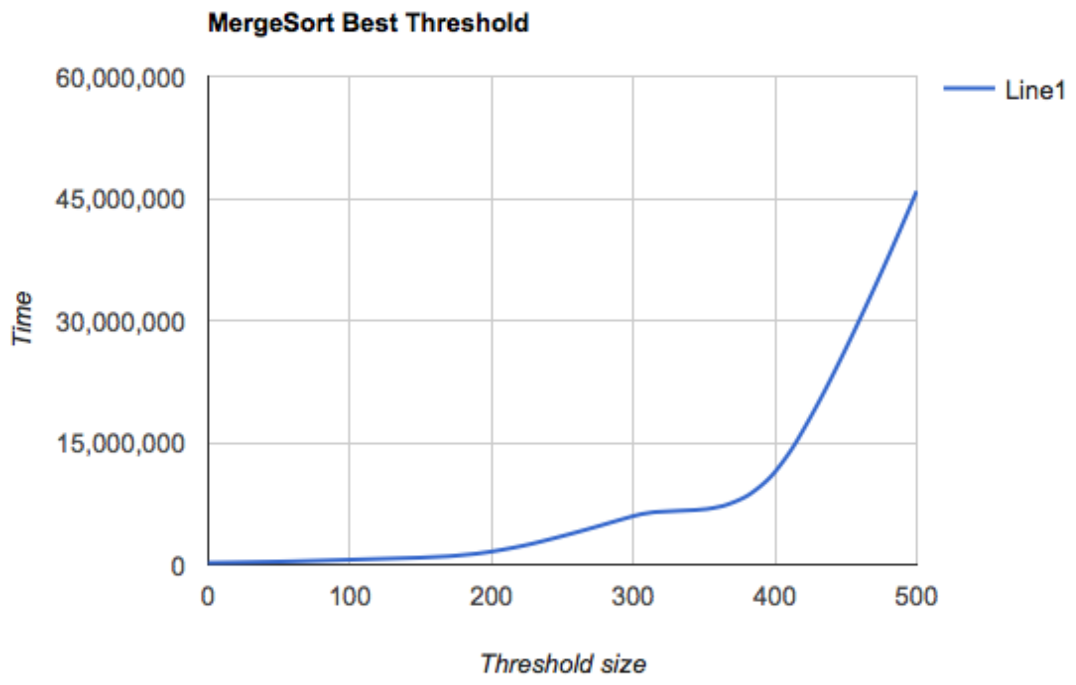
Plot the running times of your threshold mergesort for five different threshold values on permuted-order lists (one line for each threshold value). In the five different threshold values, be sure to include the threshold value that simulates a full mergesort, i.e., never switching to insertion sort (and identify that line as such in your plot)



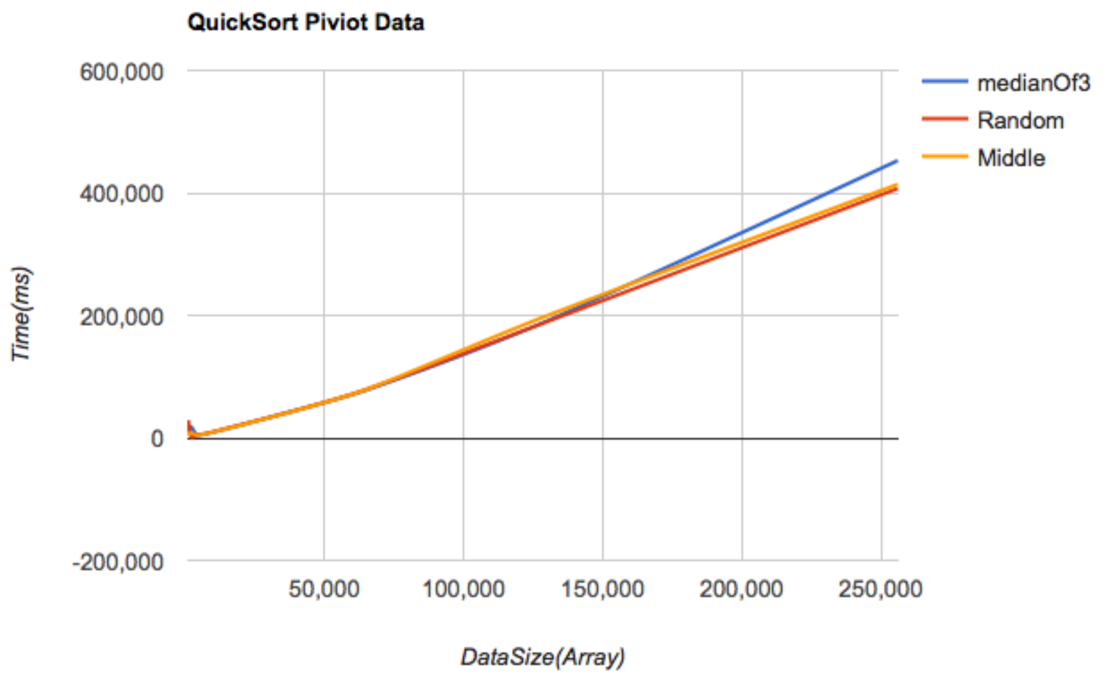
This Experiment was conducted to determine which mergeSort Threshold values would result in the best performance. The experiment was done on a large arraylist of integers covering best, average, and worst case list scenarios.

The experiment concluded that the threshold values close to 0 would provide with the fastest runtime performance of mergeSort because that is the value where insertion sort can be kicked in. The values where mergeSort is being fully utilized, and insertion sort is not kicking in, would be when the threshold value is set to around 400. This can be seen in the graph as this is the value which gives us the complexity of merge sort O(n log n), and not the complexity of insertion sort, which should be better.

Here is another graph that shows this relationship:



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).



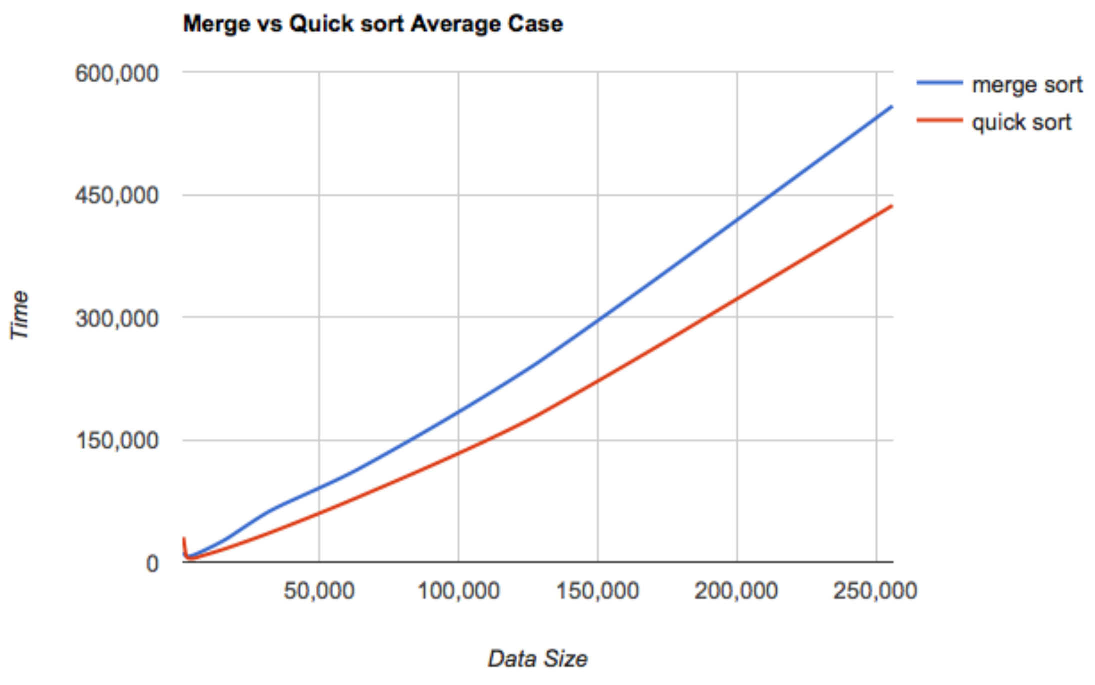
This experiment was conducted by using three different pivoting strategies for the quicksort algorithm. Quicksort, for its best and average cases, has a complexity of O(n log n), and O(n^2) for the worst case. This difference in complexity can be determined by which pivot value is used to do the sort, and from how the array list is originally "sorted" in its pre-quicksorted order. If the list is pre-sorted from greatest values to least values, and the pivot value is the very first value of the array, then this would be the worst case scenario leading to the unfavorable complexity of O(n^2). Whereas, the best-case scenario would have the pre-sorted list be in the desired sorted order.

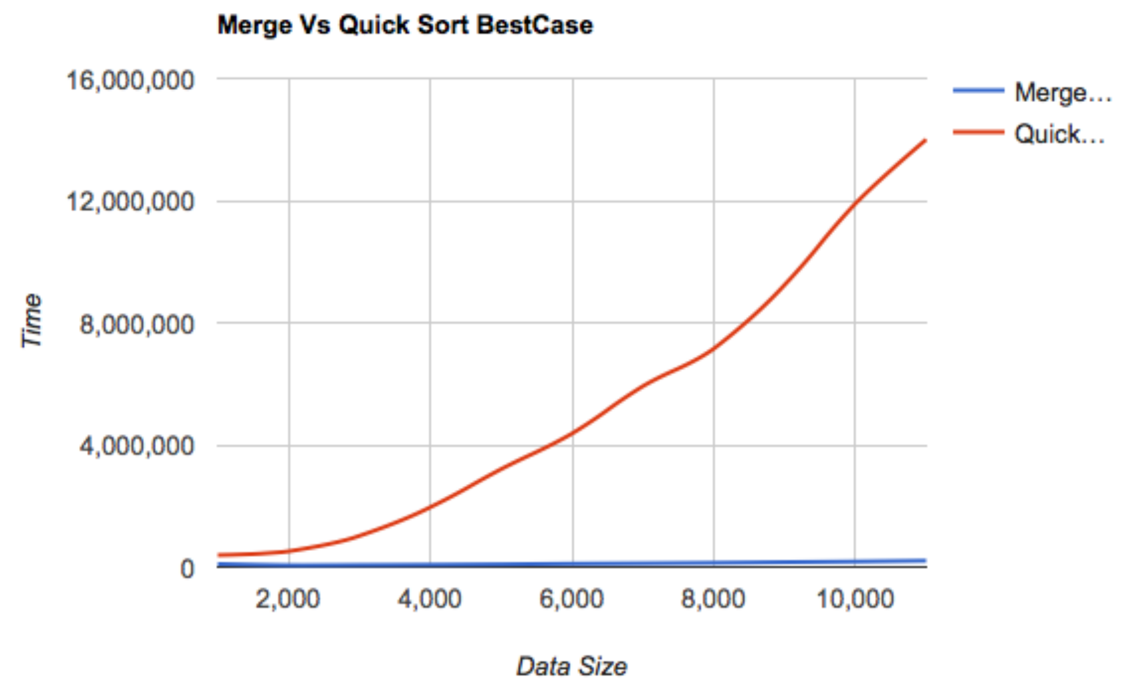
The pivoting strategies that we used are: "Median of Three", "Random Pivot", and "The Middle Value" as the pivot. "Median of Three" strategy will select the first, middle, and last values of the list and choose the median, which will then be used as the pivot for the sort. The "Random Pivot" strategy picks a pivot at random and "Middle Value" chooses the value in the middle of the list as the pivot. All three strategies are implemented on an average-case list, where the elements are randomized, and in random order.

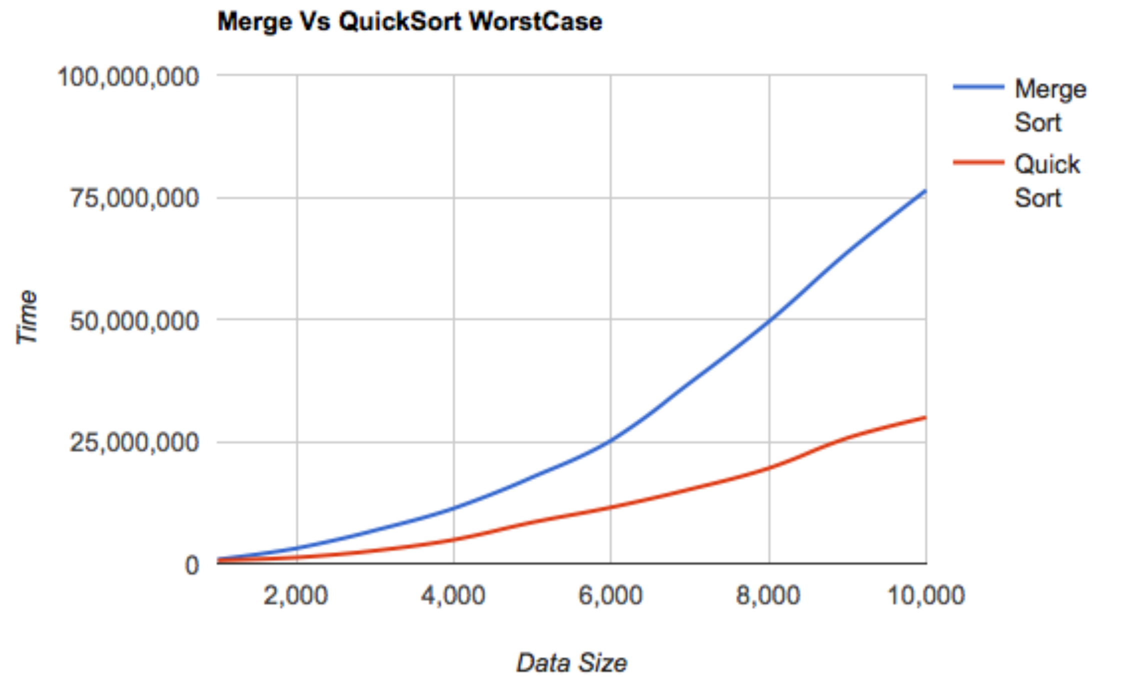
We expect the random and median of three pivot strategies to show us the best results in terms of performance, because always choosing just one value in the middle can result in unfavorable runtime performance.

After running the timing experiment, the results (as shown in the chart) translate to random pivot selection having the best runtime performance followed by middle with the median of three having the worst. Though the complexities are all no worse than O(n log n), there is a discernable difference between the three strategies. Random selection of the pivot gives us the best performance because of the type of list we used. If we had opted for a best case list and used a random pivot selection on it, the results would be unfavorable in the situations where the randomly selected pivot would be towards the end of the list.

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).




Merge Vs Quick Sort BestCase
16,000,000
12,000,000
8,000,000
4,000,000
0
Time
2,000
4,000
6,000
8,000
10,000
Data Size
Merge…
Quick…


Merge Vs QuickSort WorstCase
100,000,000
75,000,000
50,000,000
25,000,000
0
Time
2,000
4,000
6,000
8,000
10,000
Data Size
Merge Sort
Quick Sort

When comparing merge sort vs. quick sort with the average-case, worst-case, and best-case list scenarios, we see the strengths and weaknesses of each sorting algorithm. For this experiment we used the random pivot location for the quick sort as that was determined to be the fastest pivoting location, as well as a threshold value of 0 for the merge sort, as that was determined to be the fastest route.

For the Average case scenario, quicksort proved to be the faster algorithm, while for the best case scenario, merge sort proved to be more effective, mainly due to the fact that at the threshold value it would switch to insertion sort, which is a more effective sorting algorithm for best case scenarios. For the worst case scenario, quicksort is shown to be faster, exhibiting a O(n log n) behavior, while merge sort grows closer to O(n^2) complexity.

Merge Sort is more efficient for smaller array lists because its threshold value can switch it towards a faster insertion sort which is more beneficial in that case, whereas QuickSort is more efficient for larger data sets where a good pivoting value can make it relatively quick compared to merge sort where all the data needs to be resorted into a new array.

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.

In theory, we expect to see merge and quick sort algorithms to exhibit similar performance in the average and best case scenarios. The theoretical complexities of both algorithms in those cases should be O(n log n). The only place where they should differ is in the worst case scenario, where quick sort should approach O(n^2) complexity. As we can see through the graphs, the story is a little bit different, and this is due to the fact that even though we are using lists that are of average, best, and worst cases, we are using the most efficient pivoting and threshold values.

Due to these factors, the growth rate we see in our experiments do correlate with what we expect. We do see quick sort out performing merge-sort except where merge sort's threshold value allows it to exhibit insertion sort's complexity, due to the fact that it is going into insertion sort. Quick sort cannot compete with merge sort in this case, because our quick sort method is not transitioning to an insertion sort for the best case scenario.

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

18-20ish