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

Amir Mohsenian. I submitted the program.

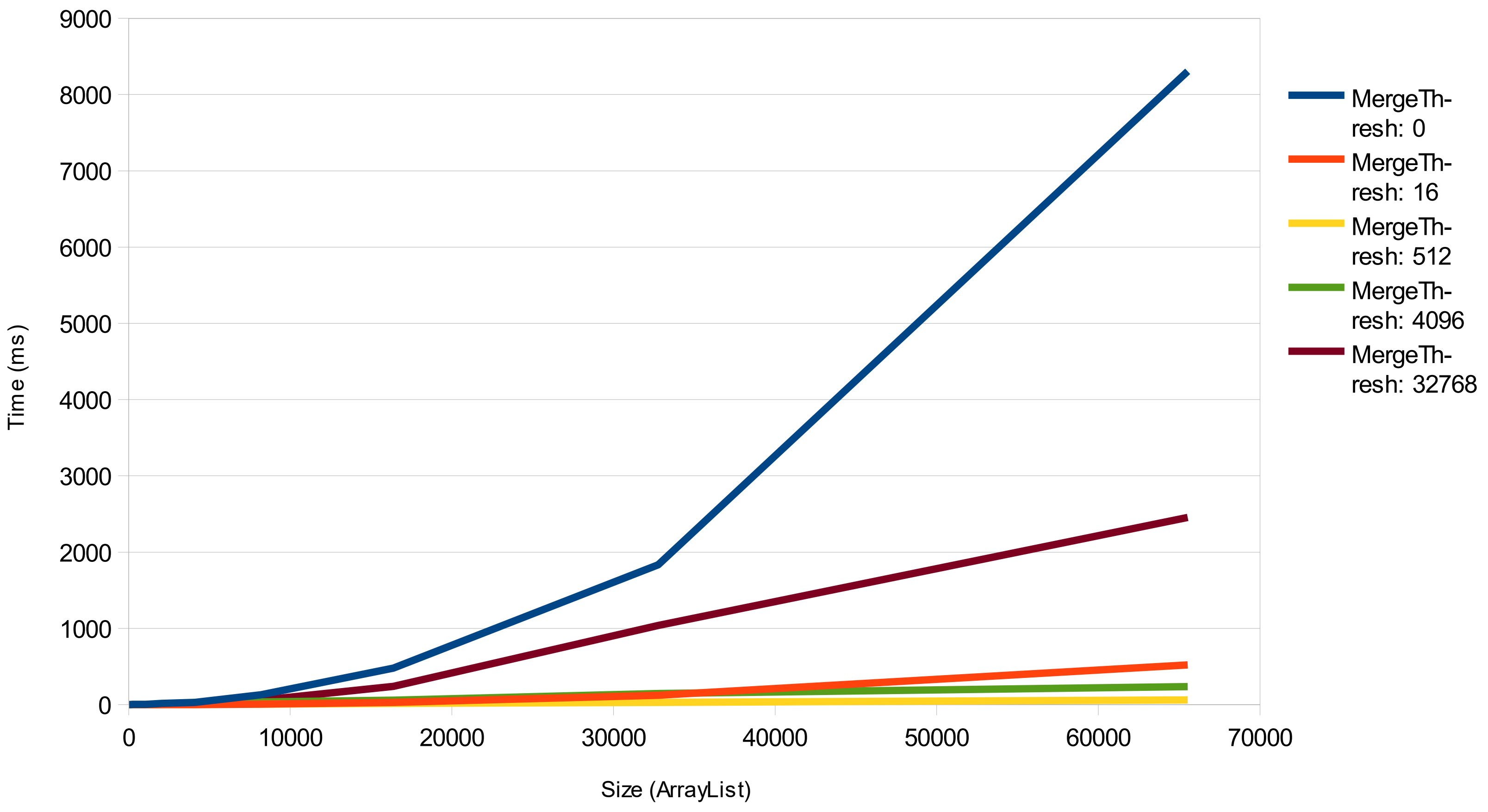
2. Evaluate your programming partner. Do you plan to work with this person again?

I feel as if Amir and I are both about on the same level as far as Java and knowledge about object oriented programming goes. I do plan to work with him again.

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?

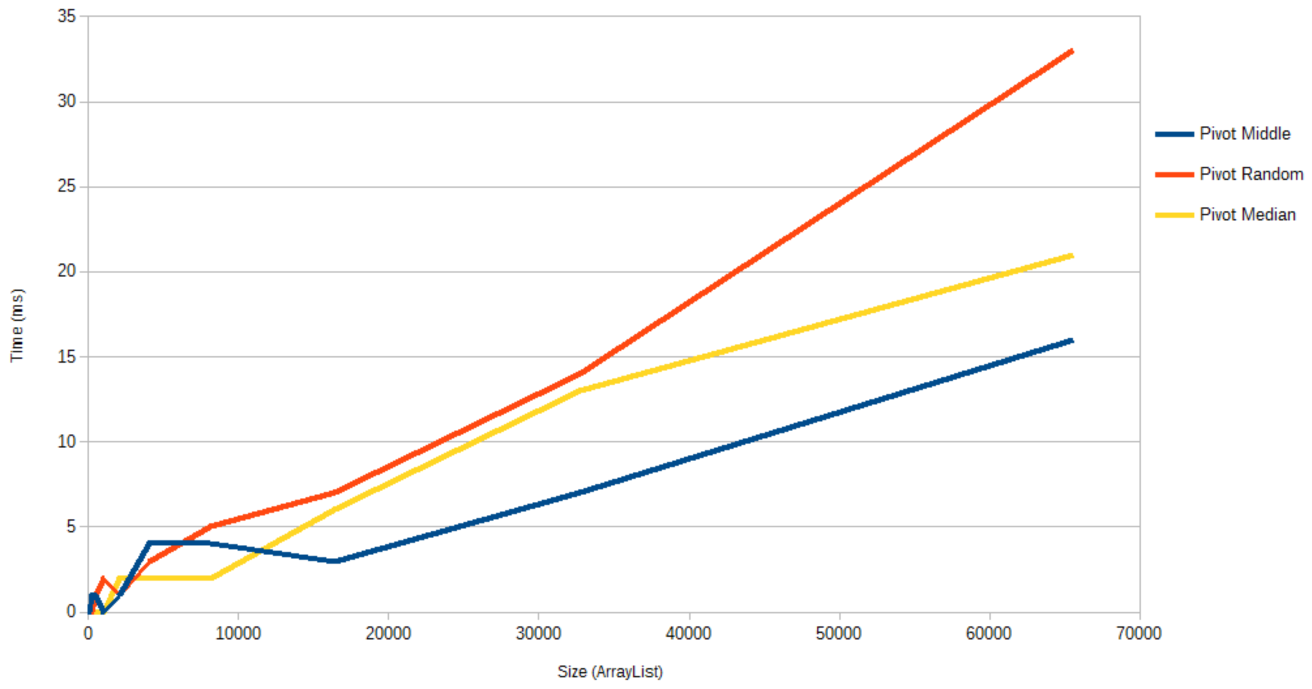
The good part of paired programming is having someone to bounce ideas off of. Also it is nice to share the work load. It can be hard to coordinate with the partner when your partner can't meet up in person. That's really the only con though. I think to be more prepared for the next paired assignment we can both come up with a schedule that keeps us on track to complete a certain section of the assignment by this time, and so forth. That would help us stay on track.

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).



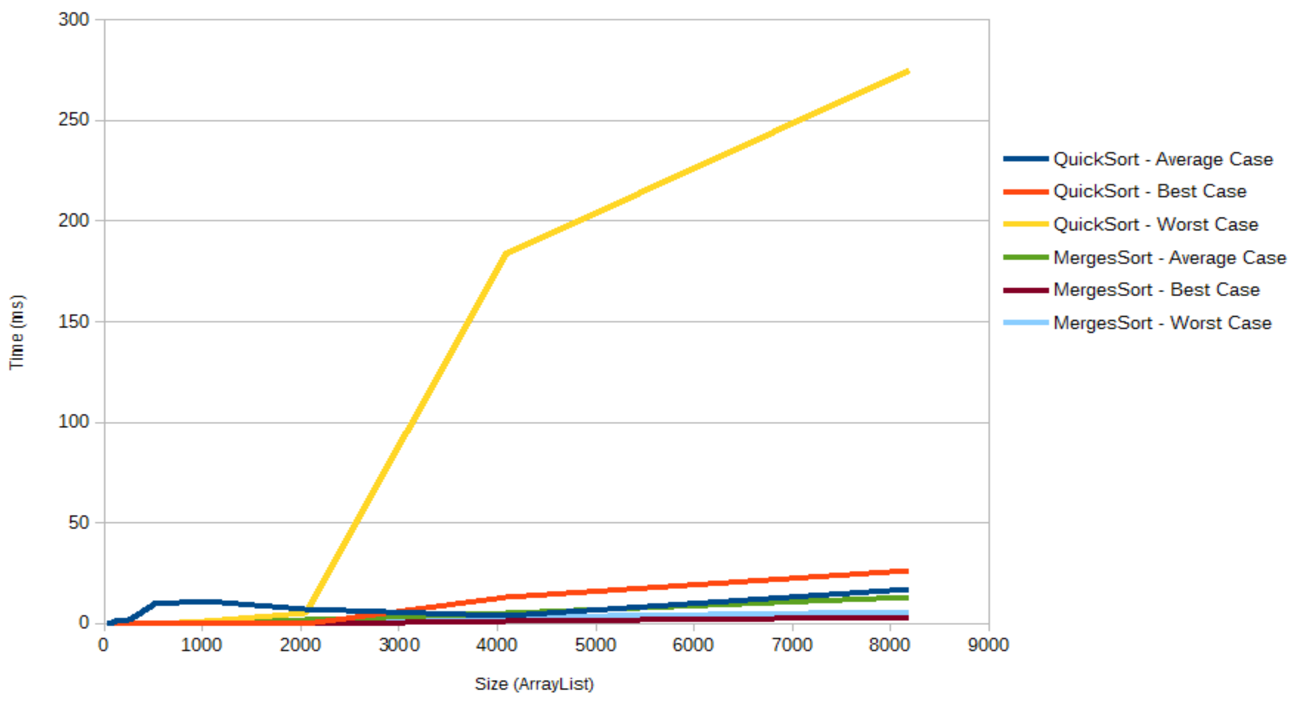
The above graph shows the time it took mergesort to complete using different thresholds for switching to insertion sort. The blue line shows the Big-O time for using mergesort without insertion sort, which was the slowest of them all. It's Big-O notation looks to be O(N log N). The other lines show O(N). Each timing test was completed with the same ArrayList of permuted order. A constant seed was used to ensure that generateAverageCase generated the same one each time. Thresholds of 0 to 32768 were used. 512 was the absolute fastest, and after that it began to get slower again.

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 generateAverageCase was used with the same seed for each experiment. The graph shows that using a random pivot gives a Big-O notation of O(N^2). Using a pivot that's in the middle and the median appear to have a O(N log N) notation. Middle and median perform better than the other depending on the circumstance, but in general middle is probably the best choice. This is because finding the median is a time intensive process that requires us to iterate through each element of the array. Finding the middle is a constant time because it only requires us to divide the array's size by two.

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).



The above graph shows that for quickSort the average and best cases are the same, which is O(N log N). It's worst case is O(N^2). MergeSort has O(N log N) for best, worst, and average case. This does not mean that mergeSort is superior to quickSort because mergeSort requires double the space of the array because it needs to make a copy of that array. When using very large arrays quickSort would be the better choice for this reason. If the size doesn't matter mergeSort would be the best because it matches quickSort in average and best cases, and it performs better in the worst case.

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.

Our growth rates partially exhibit what I expected. I expected mergeSort to have O(N log N) behavior in all cases because it will do the same thing no matter how sorted or unsorted the array is. QuickSort differs in that it's running time depends on the pivot and how unsorted the array is. I seem to be right in those two aspects. When it came to testing mergeSort for the most time efficient threshold for switching to insertionSort, I wasn't sure what to expect. It ended up being 512 for pretty much all the tested sizes of ArrayLists. This could be because 512 is not too large in relation to most of the ArrayList sizes, but also not small to the point where insertionSort's use would be meangingless. As for quickSort's most ideal pivot I was expected random to be on par with middle, and for median to take the longest. I was proven wrong by our data, as it showed a random pivot to be more time intensive than middle a lot of the time.

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

Around 10 - 12