When you are satisfied that your program is correct, write a detailed analysis document. The analysis document is 40% of your assignment 5 grade. Ensure that your analysis document addresses the following.

Note that if you use the same seed to a Java Random object, you will get the same sequence of random numbers (we will cover this more in the next lab). Use this fact to generate the same permuted list every time, after switching threshold values or pivot selection techniques in the experiments below. (i.e., re-seed the Random with the same seed)

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

Shahid Bilal, he submitted the code.

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

Shahid was a great partner. He was proactive and hard working. I was impressed with how fast he picked up github, as my previous partner couldn't use it. He did a lot of work on each assignment we worked on together, which was a nice change from my past experiences where the other person did nothing. Also, he is really good at doing the analysis side of things, a place where I could use a lot of work. If I could work with him again I would.

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?

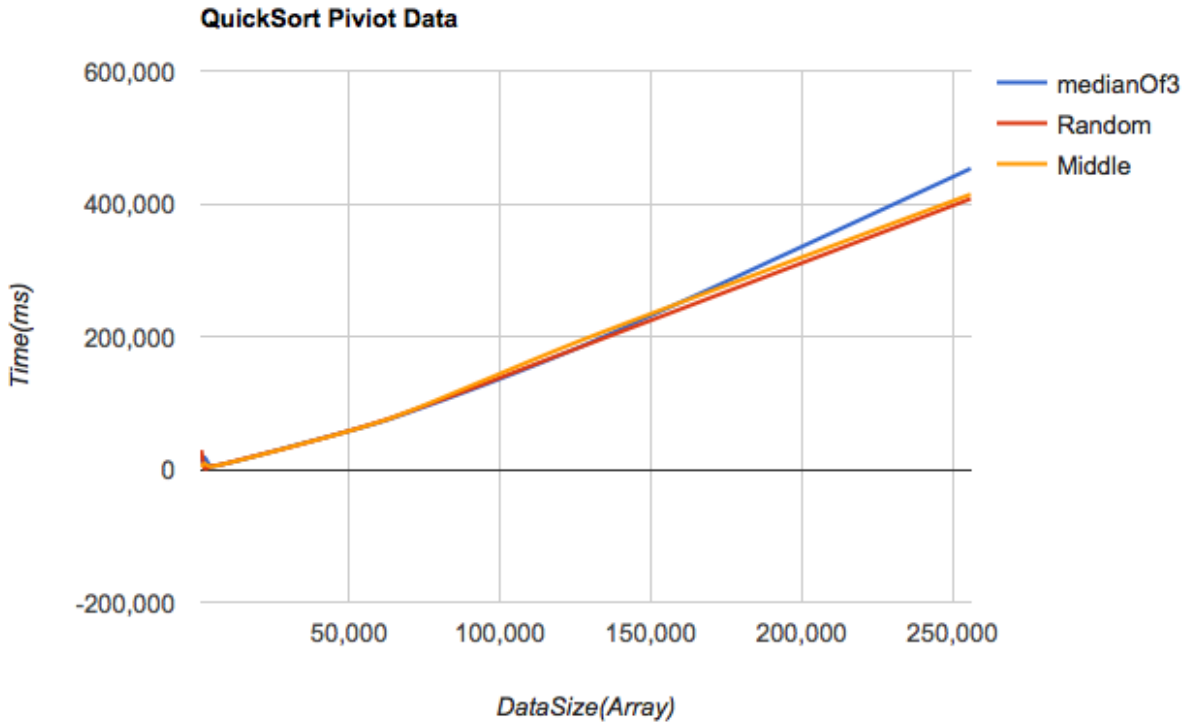
The pros of the pair programming we have done so far are a partner to back you up when you get something wrong. It is nice to have the constant backup on everything. I liked having someone to bounce ideas off of. It was hard to schedule times to meet together sometimes, but this was overcome using github and emailing. It made everything much easier. I can be a better partner by working on the program a little more often and communicating everything a little better.

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 run using only a few base thresholds. We started with a threshold of 0, no insertion sort run at all, which had a faster run time the more tests that were run. We simply increased by 100 from then on and found that the higher the threshold, the slow mergesort was. This led to the best threshold being 0, not using insertion sort at all. We used the moderate_word_list.txt as the arr from assignment 4 in the tests. The key is the threshold value that is being tested.

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).



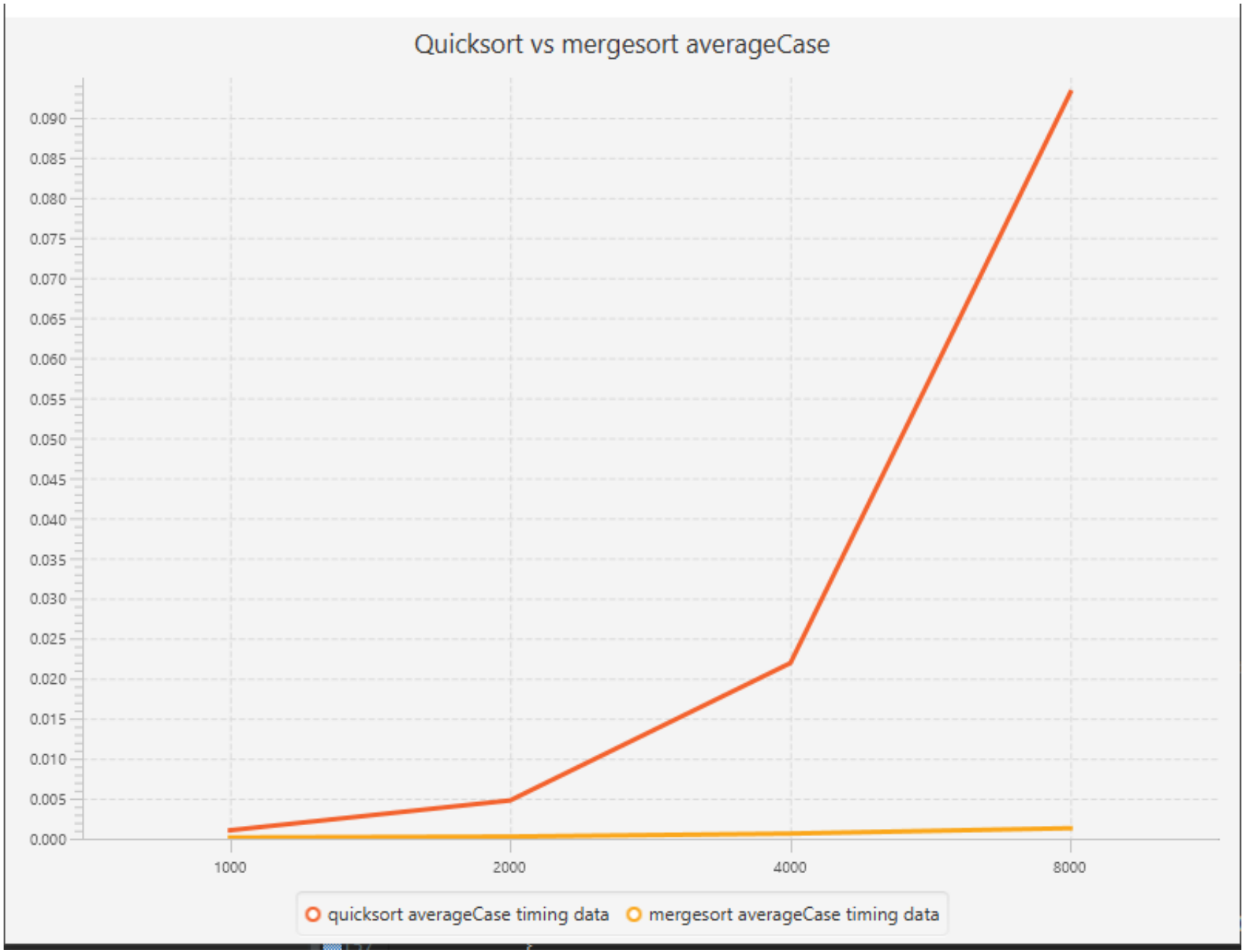
This experiment was conducted by using three different pivoting strategies for the quicksort algorithm. Quicksort, for its best and average cases, has a complexity of O(n log n), and O(n^2) for the worst case. This difference in complexity can be determined by which pivot value is used to do the sort, 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.

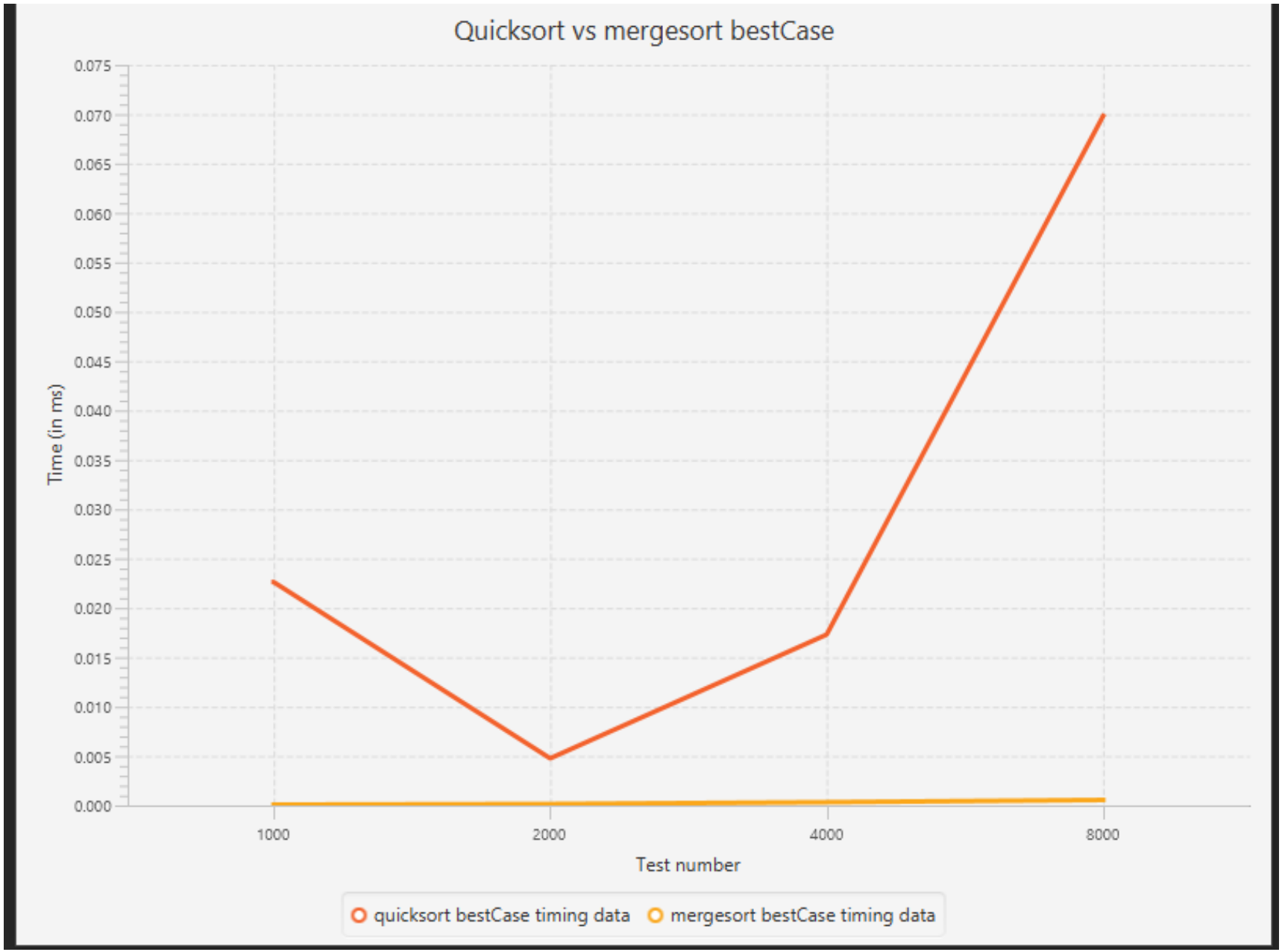
The pivoting strategies that we used are: “Median of Three”, “Random Pivot”, and “The Middle Value” as the pivot. “Median of Three” strategy will select the first, middle, and last values of the list and choose the median, which will then be used as the pivot for the sort. The “Random Pivot” strategy 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.

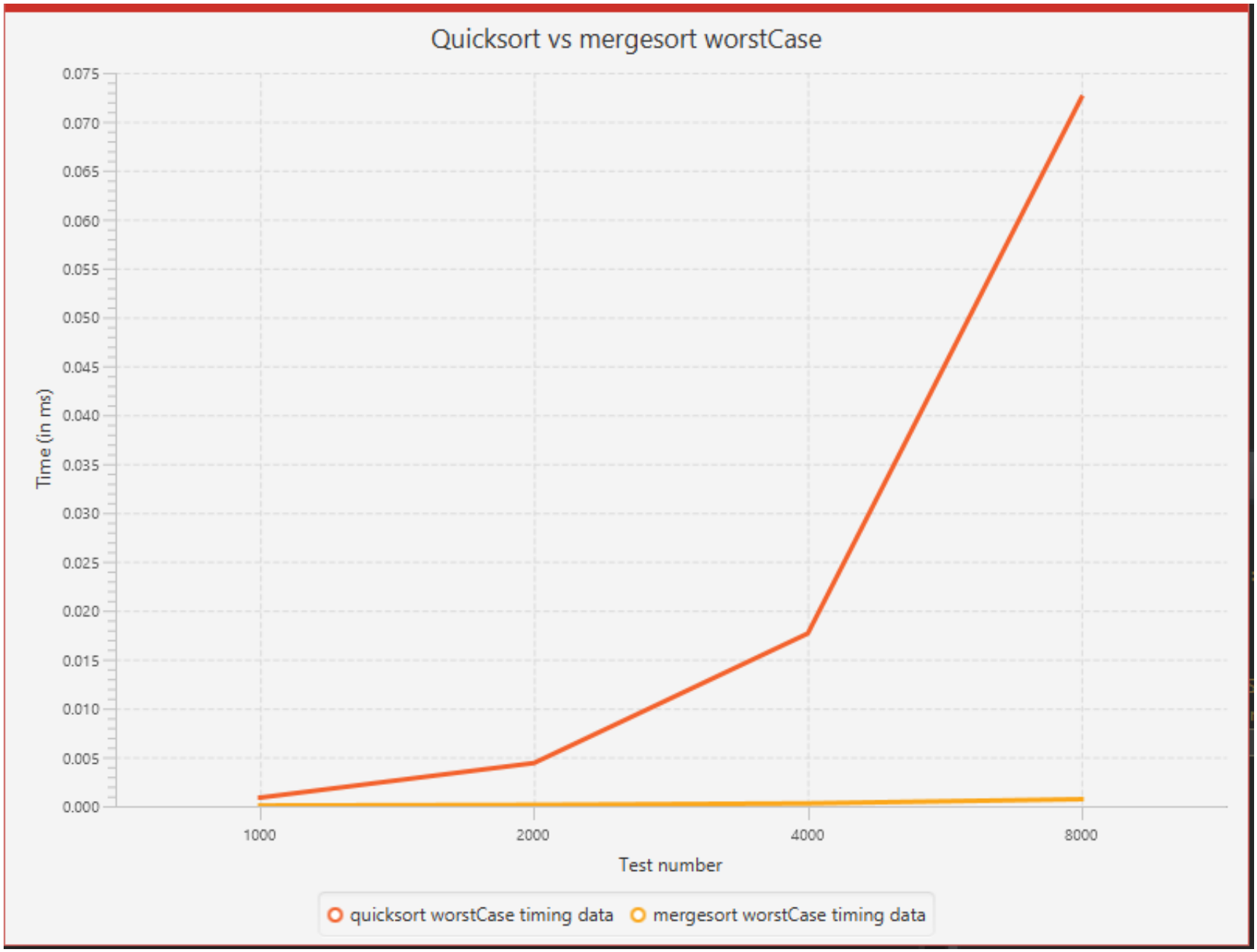
We expect the random and median of three pivot strategies to show us the best results in terms of performance, because always choosing just one value in the middle can result in unfavorable runtime performance.

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).


Quicksort vs mergesort averageCase
0.090
0.085
0.080
0.075
0.070
0.065
0.060
0.055
0.050
0.045
0.040
0.035
0.030
0.025
0.020
0.015
0.010
0.005
0.000
1000
2000
4000
8000
quicksort averageCase timing data
mergesort averageCase timing data


Quicksort vs mergesort bestCase
Time (in ms)
0.075
0.070
0.065
0.060
0.055
0.050
0.045
0.040
0.035
0.030
0.025
0.020
0.015
0.010
0.005
0.000
1000
2000
4000
8000
Test number
quicksort bestCase timing data
mergesort bestCase timing data


Quicksort vs mergesort worstCase
Time (in ms)
0.075
0.070
0.065
0.060
0.055
0.050
0.045
0.040
0.035
0.030
0.025
0.020
0.015
0.010
0.005
0.000
1000
2000
4000
8000
Test number
quicksort worstCase timing data
mergesort worstCase timing data

It would appear that mergesort is the best sorting algorithm, completing much faster than quicksort for smaller array sizes. However quicksort is completing much faster than mergesort when the array size is very large.

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.

Yes, we were expecting them to be O(n log n) because the entire input is run through (n) and the array is split log(n) times resulting in a O(n log n) complexity. Quicksort uses a pivot to create a similar effect, but it doesn't always half the array, it just chooses a point in the array and moves everything lower than it to the left and higher to the right. IN the best case it ends up being n log n because the array is split in half with only n groups to iterate through log n times. If it picks the highest value in the array then only one group is created and that gives you n groups to iterate through n times, resulting in the worst case being O(n^2) .

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

About 20

Programming partners are encouraged to collaborate on the answers to these questions. However, each partner must write and submit his/her own solutions.

Upload your solution (.pdf only) on the assignment 5 page by 11:59pm on September 28th.