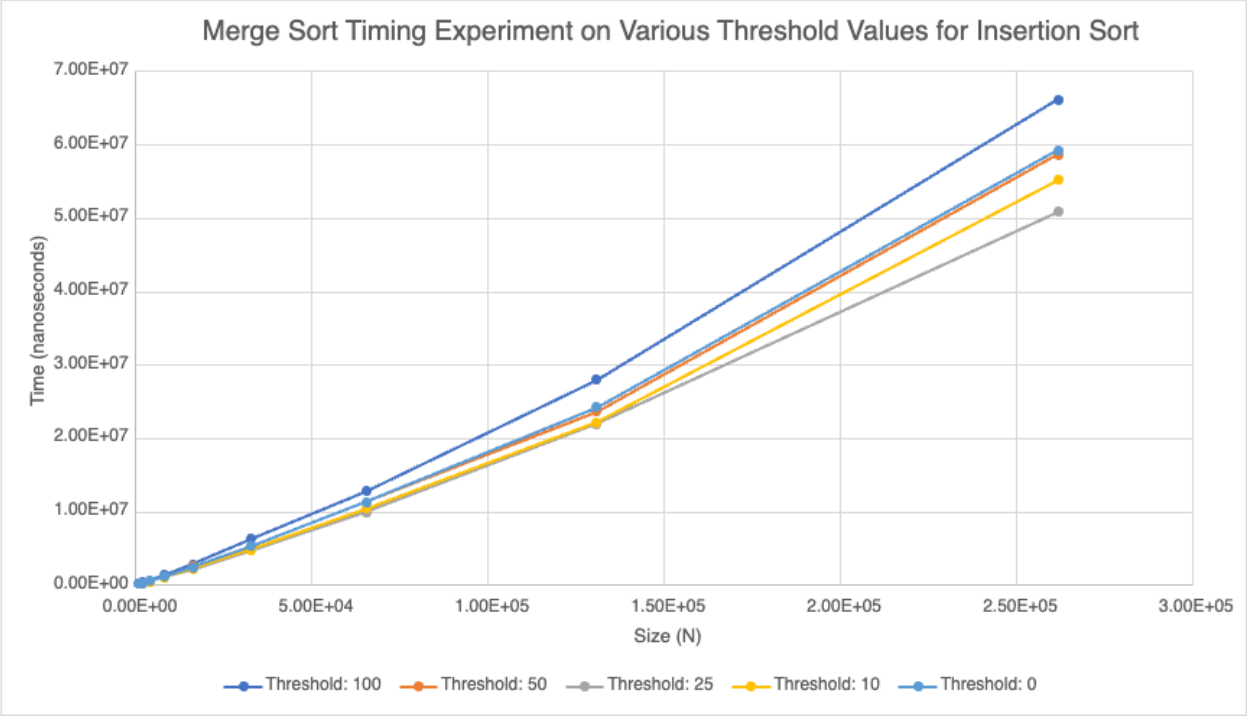
When you are satisfied that your program is correct, write a detailed analysis document. The analysis document is 40% of your assignment grade. Ensure that your analysis document addresses the following.

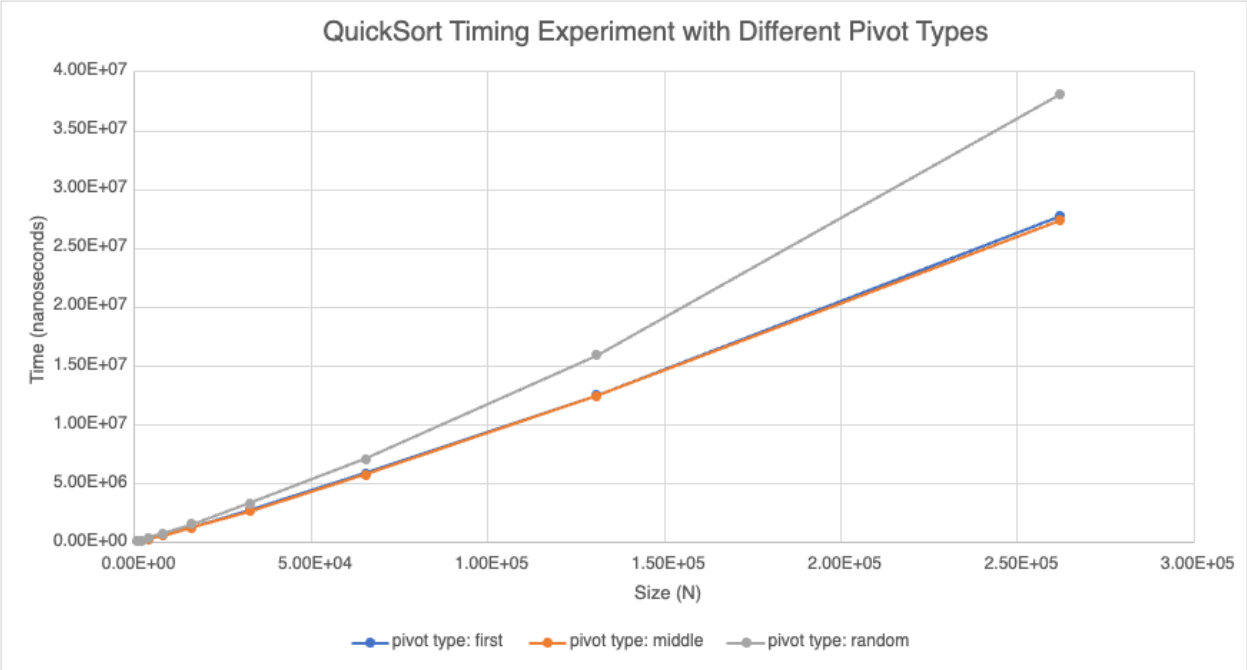
Note that if you use the same seed to a Java Random object, you will get the same sequence of random numbers (we will cover this more in a lab soon). Use this fact to generate the same permuted list every time, after switching threshold values or pivot selection techniques in the experiments below. (i.e.,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

1. Who are your team members?
    - Felix Ye

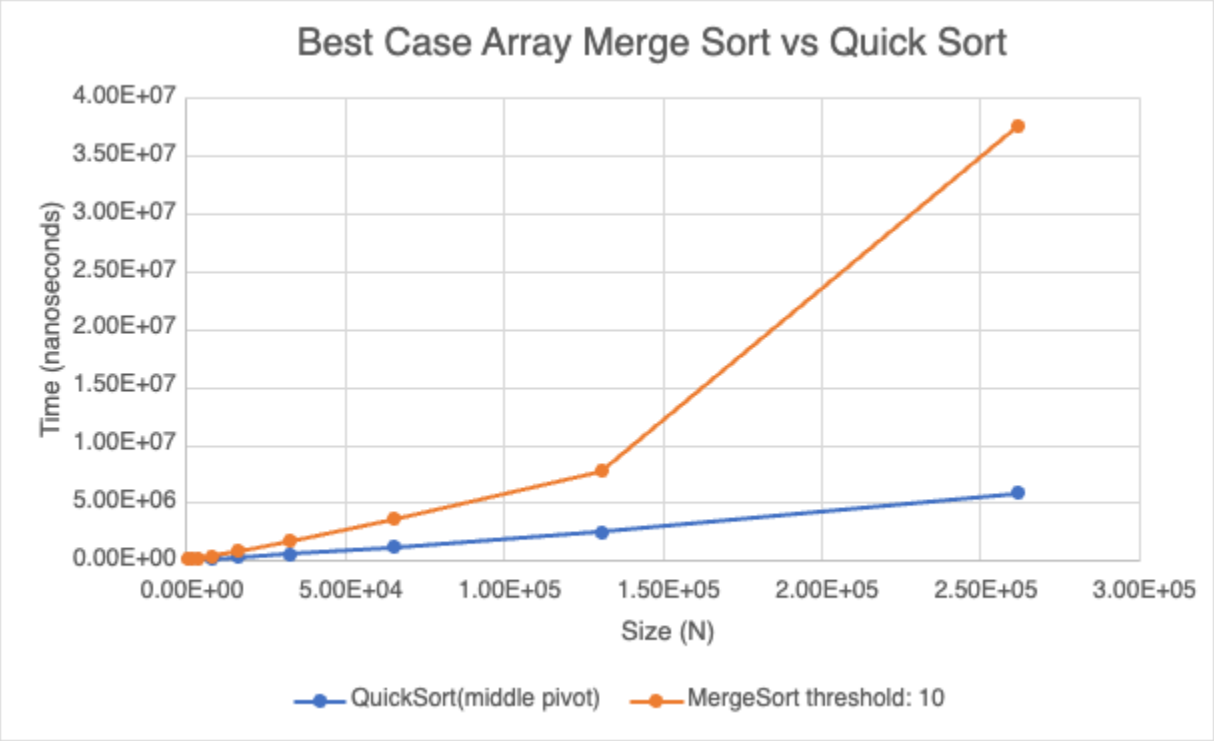
2. Mergesort Threshold Experiment: Determine the best threshold value for which mergesort switches over to insertion sort. Your list sizes should cover a range of input sizes to make meaningful plots, and should be large enough to capture accurate running times. To ensure a fair comparison, use the same 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 we already demonstrated, 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).

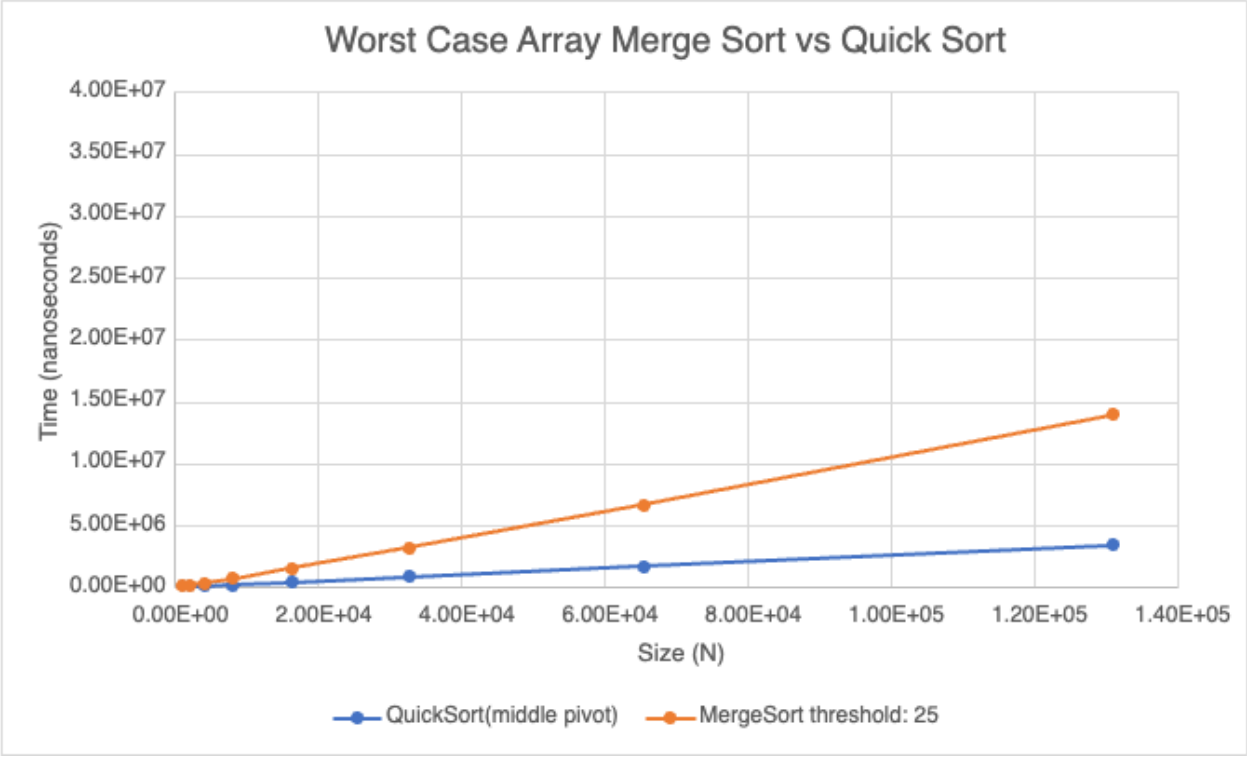


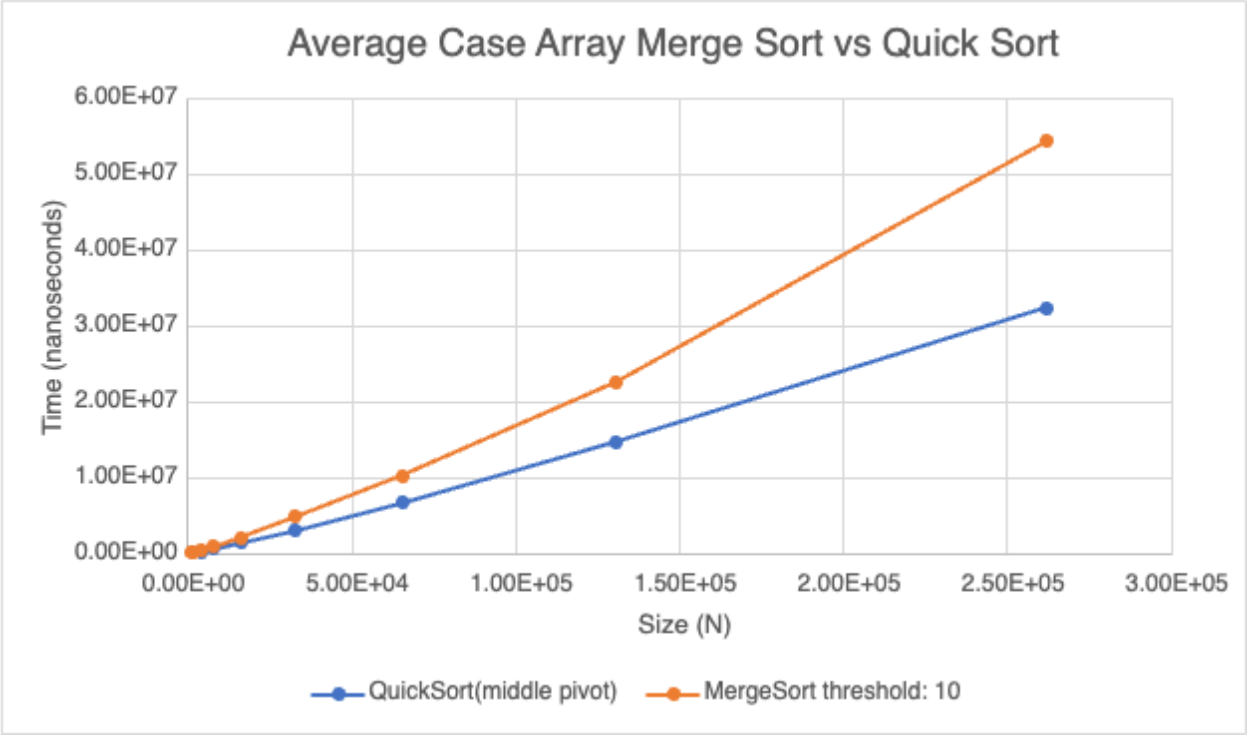
3. Quicksort Pivot Experiment: Determine the best pivot-choosing strategy for quicksot. (As in #2, use large list sizes, the same set of permuted-order lists for each strategy, and the timing techniques demonstrated before.) Plot the running times of your quicksort for three different pivot-choosing strategies on permuted-order lists (one line for each strategy).



4. 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 #2, use large list sizes, the same list sizes for each category and sort, and the timing techniques demonstrated before. 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).







5. 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.

Some of these graphs are what I expected to see and others are not. Overall they are all n log(n), which is what I expected, as any n^2 lines would indicate an error in my sorting methods. So the overall trend and growth rate is what I expected but the differences between Merge Sort and Quick Sort on my comparison graphs was not what I had initially expected.
For the Merge Sort pivot type experiment I was not surprised by my results/graph. When the threshold is set to a high value, this will increase the running time of Merge Sort as the time complexity of insertion sort can be as bad as O(n^2). Insertion sort also works better on smaller and mostly sorted arrays, so above a certain threshold, the number of swaps will increase which will slow down merge sort as a whole. The benefit of insertion sort is that it is a sort in place algorithm so it does not require an additional array to sort into, considering this is one of the constraints within Merge Sort, (reduced size capacity) this can be used to our advantage in reducing the number of times we need to access multiple arrays. In terms of the thresholds I tested, 25 was the most efficient point in which to switch to Insertion Sort. I tested 0, 10, 25, 50 and 100, so more values could have been tested in between to find the best threshold value for this particular random array (seeded with the same seed). This value of 25 is where the benefit of reduced size complexity from insertion sort is the greatest before the increased cost of time complexity from Insertion Sort becomes too costly.
For the Quick Sort pivot type experiment I was not surprised by my results/graph. When the array being sorted is permuted, the best pivot type comes down to luck. For “worst case” and “best case” arrays, the first, or last would be the worst pivot point, middle would be the best and random could be somewhere between the 2, but in the case of average arrays, it is highly dependant on the value of the chosen pivot. For all of these tests, the random was seeded with the same seed which results in the same series of “random” integers. So for this particular array list of integers, a pivot type of the first and middle values happened to perform quite similarly and the random pivot happened to perform worse, but this could be different and variable, particularly if seeding with a truly random seed (such as the time), so I was unsurprised by the results of the pivot experiment.
For my Quick Sort vs Merge Sort graphs I was initially surprised until I looked further into the results I saw. The time complexity of Merge Sort is O(n*log(n)) in best, average and worst case scenario. The time complexity of Quick Sort is n*log(n) for best and average but n^2 for worst case. This is not what is reflected in the graphs from the timing experiments I conducted, where I would expect to see Merge Sort performing better, particularly on larger data sets, instead, based on my graphs, Quick Sort performs better (meaning faster) than Merge Sort on best, average and worst case arrays. Initially I found this very surprising until I looked into it more and found that Quick Sort exhibits good cache locality and this makes it faster than Merge Sort. This is because Quick Sort is a sort in place algorithm which means that the elements are being sorted within the same array. Merge Sort on the other hand requires more space O(n) because it is NOT sorted in place, which means it requires an additional array for the sorting to occur. In some cases Merge Sort is set up in a way that a new array is created for each recursive call to MergeSort, I did not set up my algorithm this way, and instead passed in 1 array to place the sorted values into that is then copied back into the original passed in array list. Because there is only 1 array passed in, it does mean the values must be copied over within each call to merge, so this contributes to a small slow down in time efficiency as both of the arrays need to be

accessed for sorting to occur and this could be how the additional size complexity impacts the time in this case.

Team members are encouraged to collaborate on the answers to these questions and generate graphs together. However, each member must write and submit his/her own solutions.

Upload your solution (.pdf only) through Canvas.