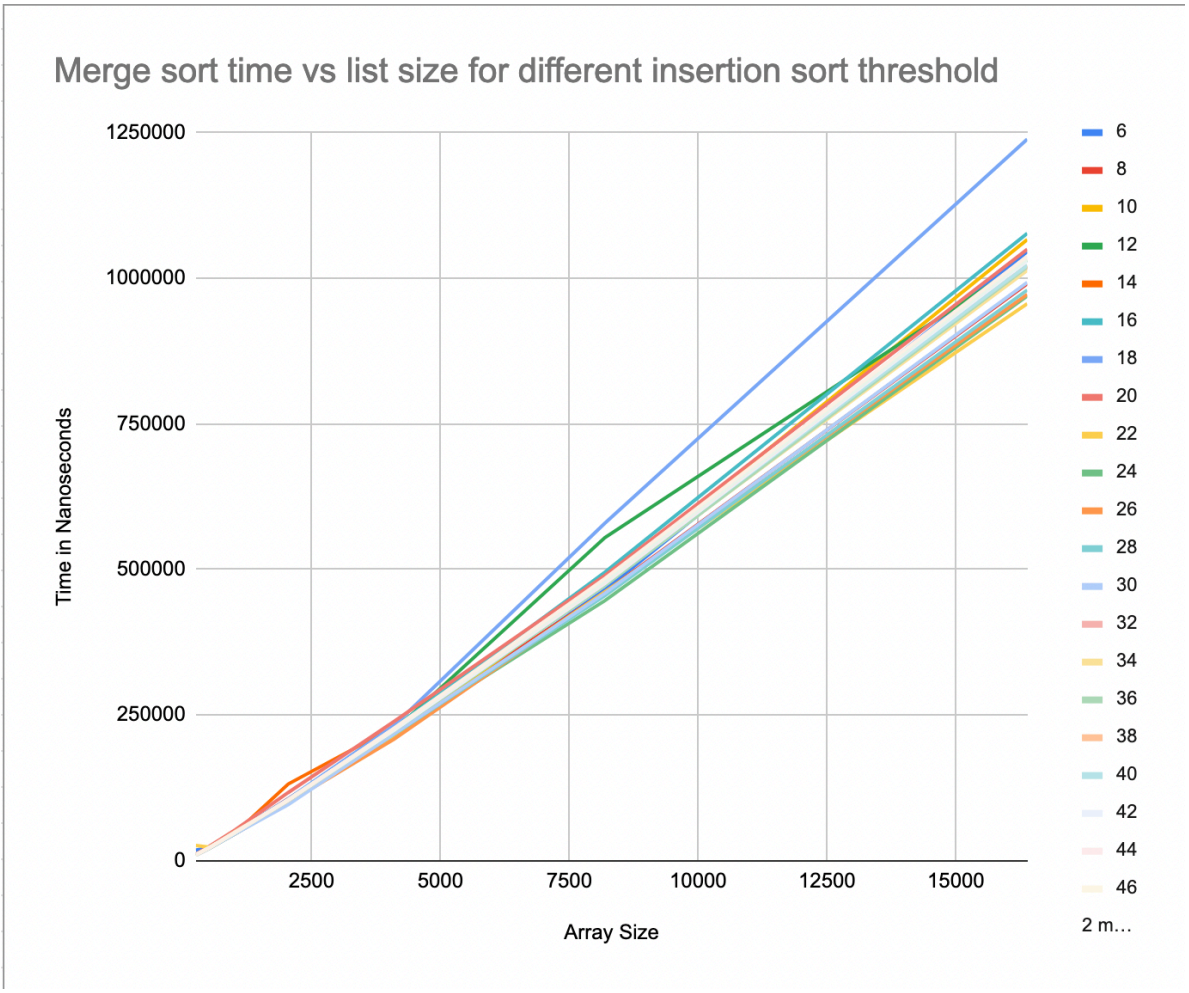
1. Who are your team members?

Randi Prince

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)

The best threshold we found was that of size 22 with the difference continuing to grow as list size increased

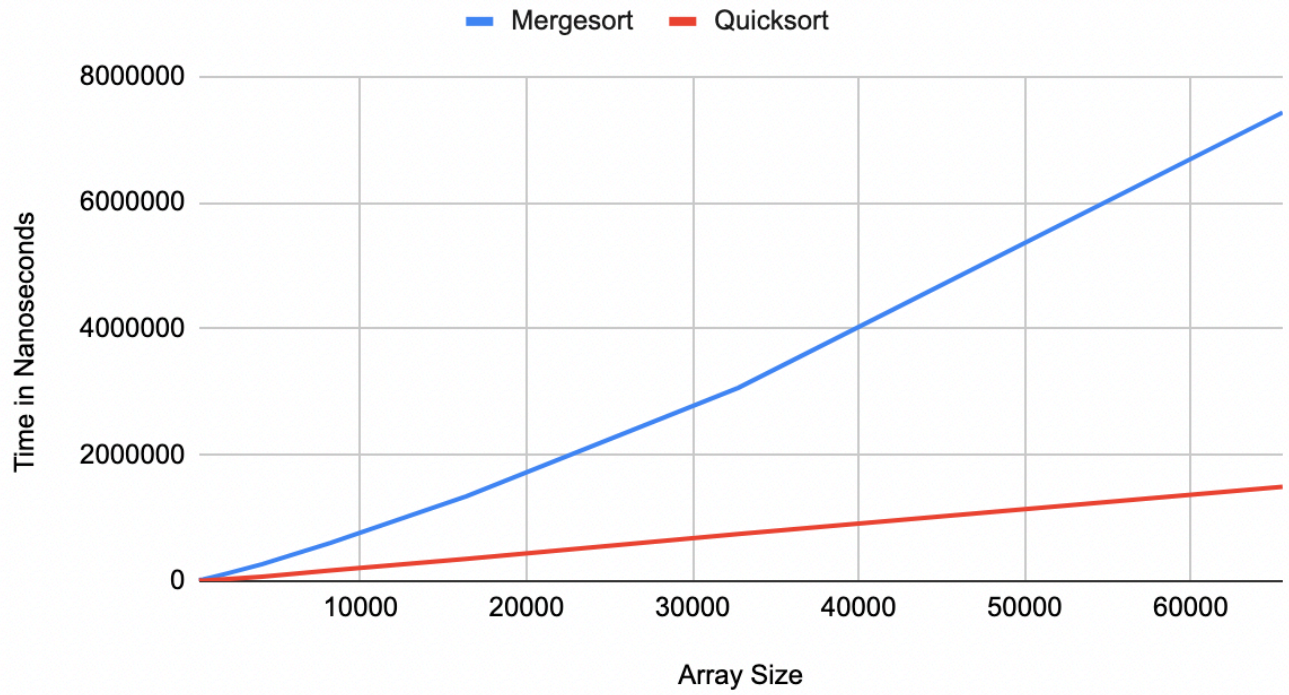


3. Quicksort Pivot Experiment: Determine the best pivot-choosing strategy for quicksot. (As in #2, use large list sizes, the same set of permuted-order lists for each strategy, and the timing techniques demonstrated before.) Plot the running times of your quicksort for three different pivot-choosing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

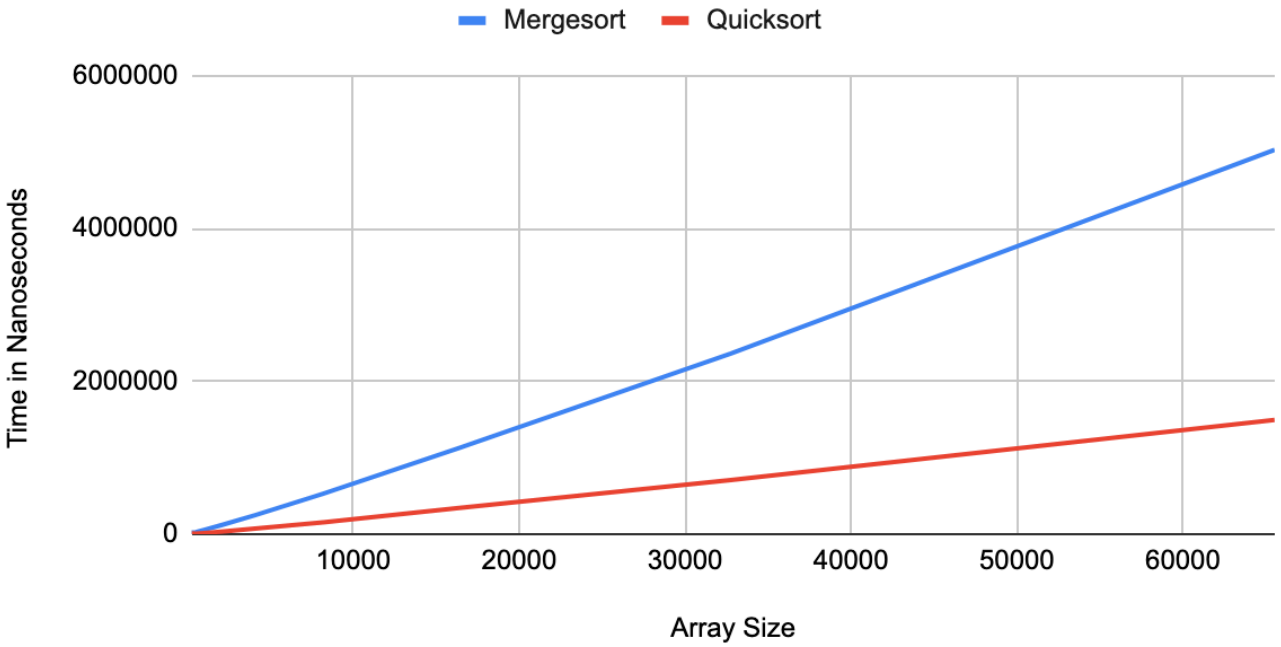
As seen above, choosing the center pivot had the best time for our strategies

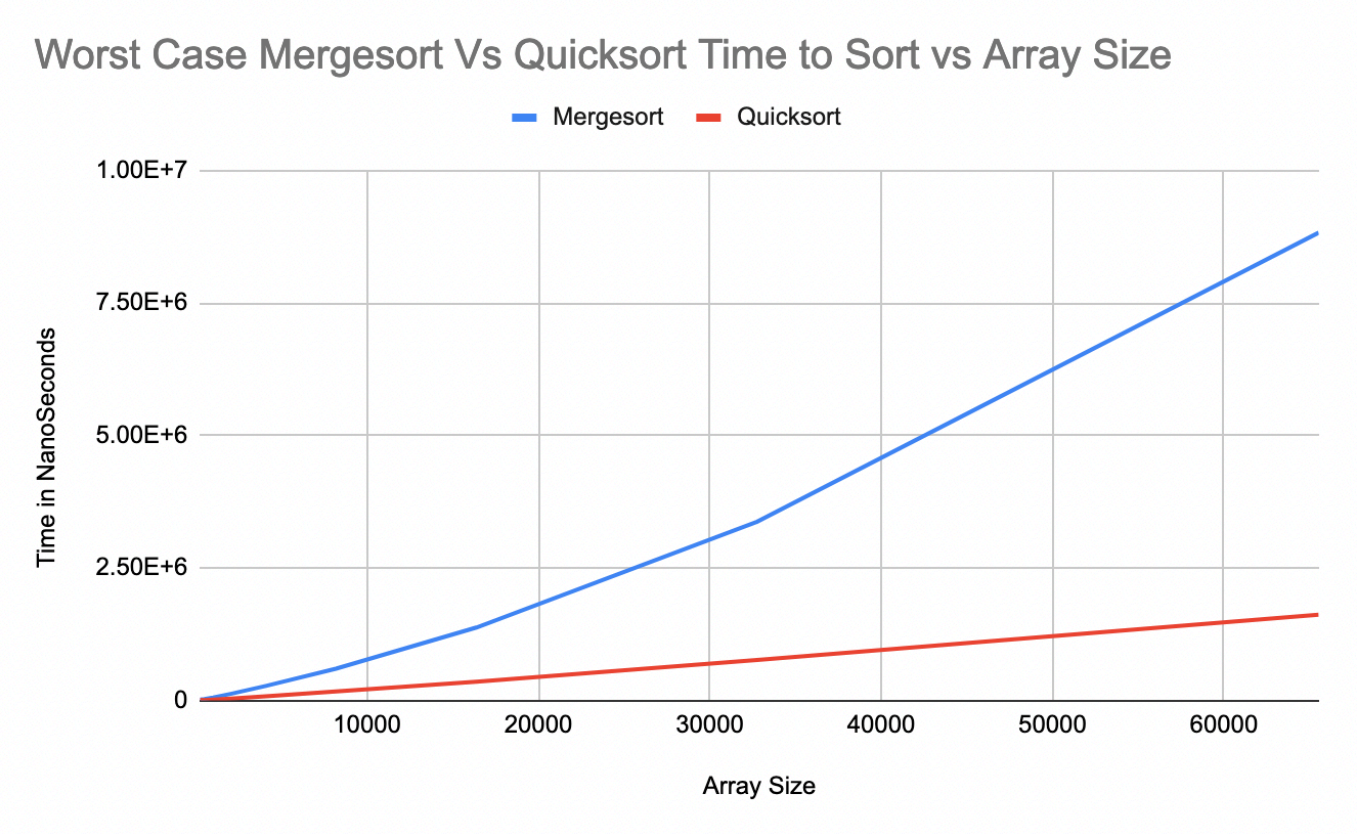
4. Mergesort vs. Quicksort Experiment: Determine the best sorting algorithm for each of the three categories of lists (best-, average-, and worst-case). For the mergesort, use the threshold value that you determined to be the best. For the quicksort, use the pivot-choosing strategy that you determined 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).

Best Case Mergesort Vs Quicksort Time to Sort vs Array Size



Average Case Mergesort Vs Quicksort Time to Sort vs Array Size





It seems like Quicksort is the faster choice for all three data types

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.

The best and average cases look like the expected time complexities of merge and quick sort, however the time complexity of the worst case seems strange.

For quick sorts worst case, the behavior is expected to be O(N^2) but quick sort seems to be quicker than merge sort.

The quick sorts quadratic time complexity seems to have been avoided because the worst case is avoided because I use the center point as the pivot instead of the first or last, which means that the worst case technically isn't the quick sorts worst case technically. So unless an array of very specific values were used to throw quick sort off, it seems like its faster than merge sort.

When considering that the O(N^2) complexity is avoided by the quick sort pivot strategy, the running time for these makes sense because the complexities are both about O(N * lg N) but the quick sort has much lower multiples