Assignment 5

I (Chase Yates) and Liam Astin worked on this assignment. I submitted the code to gradescope.

Part 1

> Briefly discuss the pair programming experience, including:
> About how many hours did you spend completing the programming and testing portion of this assignment?
> Evaluate your programming partner. Do you plan to work with this person again?

Pair programming was a similar experience to before. I was the driver this time and Liam was the navigator. It took us about 3 hours to complete the main portion of the assignment and a lot of debugging was needed and a lot of tests. My partner was fine, I think I will switch partners for after fall break though.

Part 2

> What are the expected growth rates of merge sort in the best, average, and worst cases? How does the ordering of the list to be sorted affect the running time of merge sort?

The expected growth rate of merge sort is O(NlogN) in all cases. Whether the inputted list is sorted or not doesn't matter, it still has to do the same number of comparisons and steps in the end.

Part 3

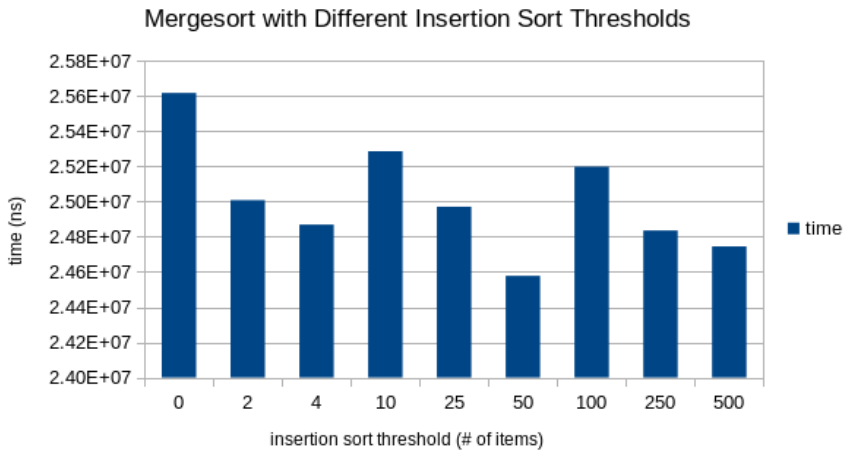
> Explain how you invoked insertion sort when the threshold for small sublists in merge sort was reached. In particular, what did you do to ensure that you are not invoking insertion sort for the entire list being sorted?

I checked the length of the passed `ArrayList` slice by subtracting the end index minus the start index plus 1 to find how many items were in the slice. If that value was less than the threshold, we ran insertion sort on the slice instead. This ensures it will not run on the entire array.

Part 4

> Merge sort threshold experiment: Determine the best threshold value for which merge sort switches over to insertion sort. (Use large list sizes, but not so large that you cannot collect the timing data in a reasonable period of time.) To ensure a fair comparison, use the same set of permuted lists for each threshold value.
> Note that the best threshold value may be a constant value or a fraction of the list size.
> Plot the running times of your merge sort for five different threshold values on permuted lists (one line for each threshold value). In the five different threshold values, be sure to include the threshold value that simulates a full merge sort (and identify that line as such in your plot).

For this experiment I used list sizes of 5000 as that was all my computer could do within a reasonable time. I used a bit more than five threshold sizes to get a larger sample. The following chart is what I found:



According to the chart, the most efficient threshold size ended up being 50 elements, at which mergesort would switch over to insertion sort. As expected, the mergesort time overhead caused it to take much longer with pure mergesort (0 threshold).

Part 5

> What are the expected growth rates of quicksort in the best, average, and worst cases? How does the ordering of the list to be sorted affect the running time of quicksort? How does the choice of pivot affect the running time of quicksort?

In the best case as well as the average, quicksort is O(NlogN). In the worst case, however, quicksort is O(N^2). The ordering of the list may not directly affect the sorting algorithm itself, however it will affect the choice of pivot. In a median of three case, for example, a sorted list would ensure a best case scenario for quicksort. The running time of quicksort is determined by how well the choice of pivot approximates the median of the data. In the worst case, the choice of pivot is an extreme of the data, and the algorithm must partition N times and partitioning would be O(N).

Part 6

> Explain the three strategies that you used to choose the pivot in your quicksort implementation. What is the Big-O behavior of each strategy? How does each pivot-selection strategy affect the overall Big-O behavior of quicksort?

The first strategy we implemented is to take the middle element of the list as the pivot. This is O(1). How it affects the overall order of quicksort is pretty much random on a permuted list, however it works well in ensuring O(NlogN) on an ascending or descending list.

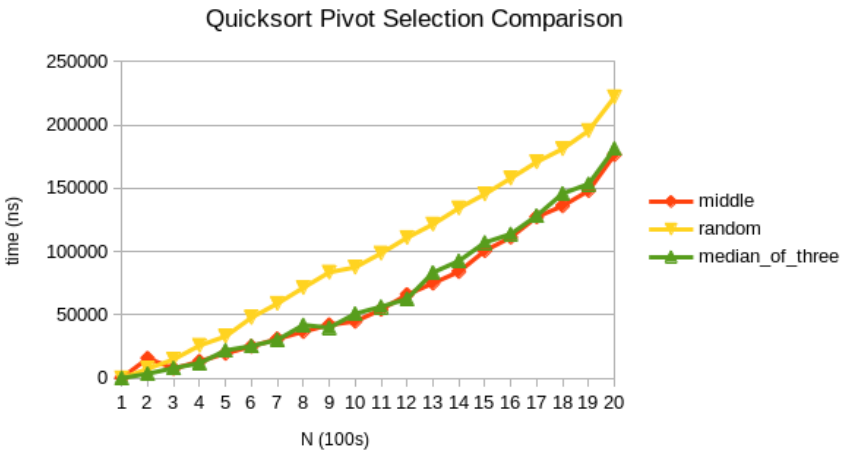
The second strategy we implemented is to take a random element of the list as a pivot. This is O(1), but its affect on the order of quicksort is random, however most of the time it will result in the average case of O(NlogN).

The third strategy we implemented is to take the median of the first, last, and middle element of the list. This is one of the most common choices for pivot selection as it is O(1), but makes the algorithm much more likely to choose a pivot that is fairly close to the median and much more representative of the entire array as it is taking a larger sample. This skews the algorithm more towards the best case of O(NlogN) on a permuted list, and confirms the best case for ascending and descending list. There is still a chance, however, that this choice of pivot can result in slower times like O(N^2), but this is very unlikely.

Part 7

> Quicksort pivot experiment: Determine the best pivot-selection strategy for quicksort. (As in #5, use large list sizes and the same set of permuted lists for each strategy.)
> Plot the running times of your quicksort for three different pivot-selection strategies on permuted lists (one line for each strategy).

For this experiment I ran 20 tests on all three pivot selection strategies on arraylists of size 0 to 1900, looping 1000 times on each sort to get a good average time. Here are the results:

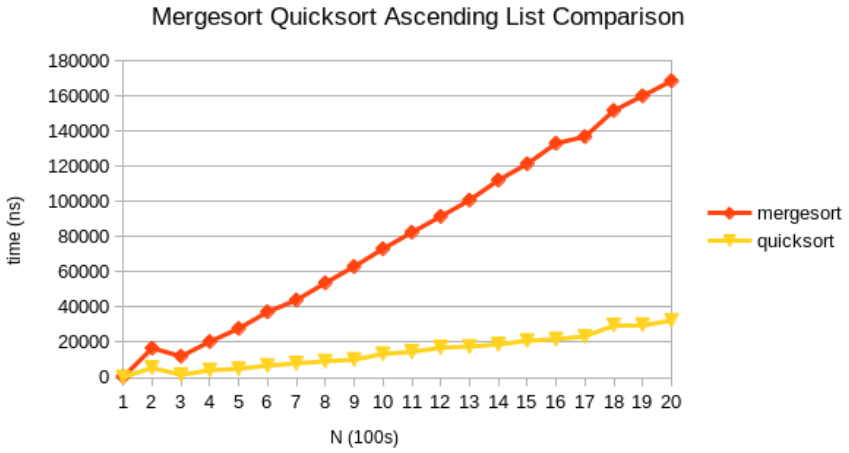


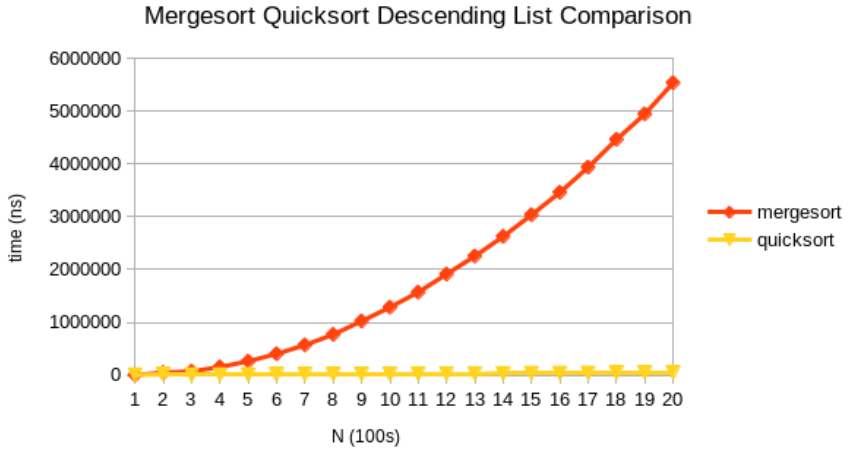
As you can see, the random selection strategy seems to have been the slowest, with median of three and middle following almost exactly the same trend. As median of three is more safe in edge cases, I think this is the best option for pivot selection.

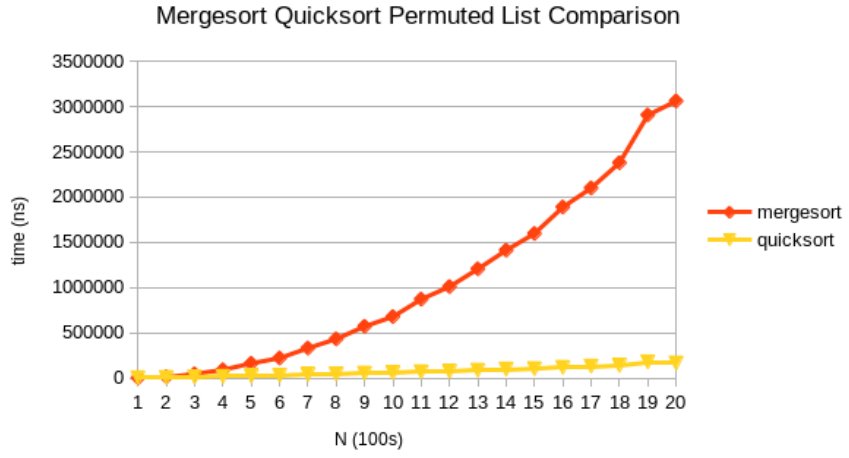
Part 8

> Merge sort vs. quicksort experiment: Determine the best sorting algorithm for each of the three categories of lists (ascending, permuted, and descending). For the merge sort, use the threshold value that you determined to be the best. For the quicksort, use the pivot-selection strategy that you determined to be the best. (As in #5, use large list sizes and the same list sizes for each category and sort.)
> 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).

For this experiment, I ran the same number of tests and list sizes as the previous experiment. The results I got were very shocking:







As you can see, quicksort simply outclassed mergesort in terms of speed on my machine. However, both algorithms seem to follow the trend of O(NlogN)

Part 9

> Do the actual running times of your sorting methods exhibit the expected growth rates? Why or why not?
> How did you determine the growth rate of the actual running times (e.g., trend of the plotted line, convergence of T(N)/F(N), or something else)?

The trends of the lines and the data I gathered seem to point to a roughly O(NlogN) performance for both algorithms. I am surprised, however at simply how much faster quicksort seems to work at these large problem sizes. I did use an insertion sort threshold on my quicksort algorithm as it was optional (and got rid of a few stack overflow headaches), which may have contributed to this trend as well.