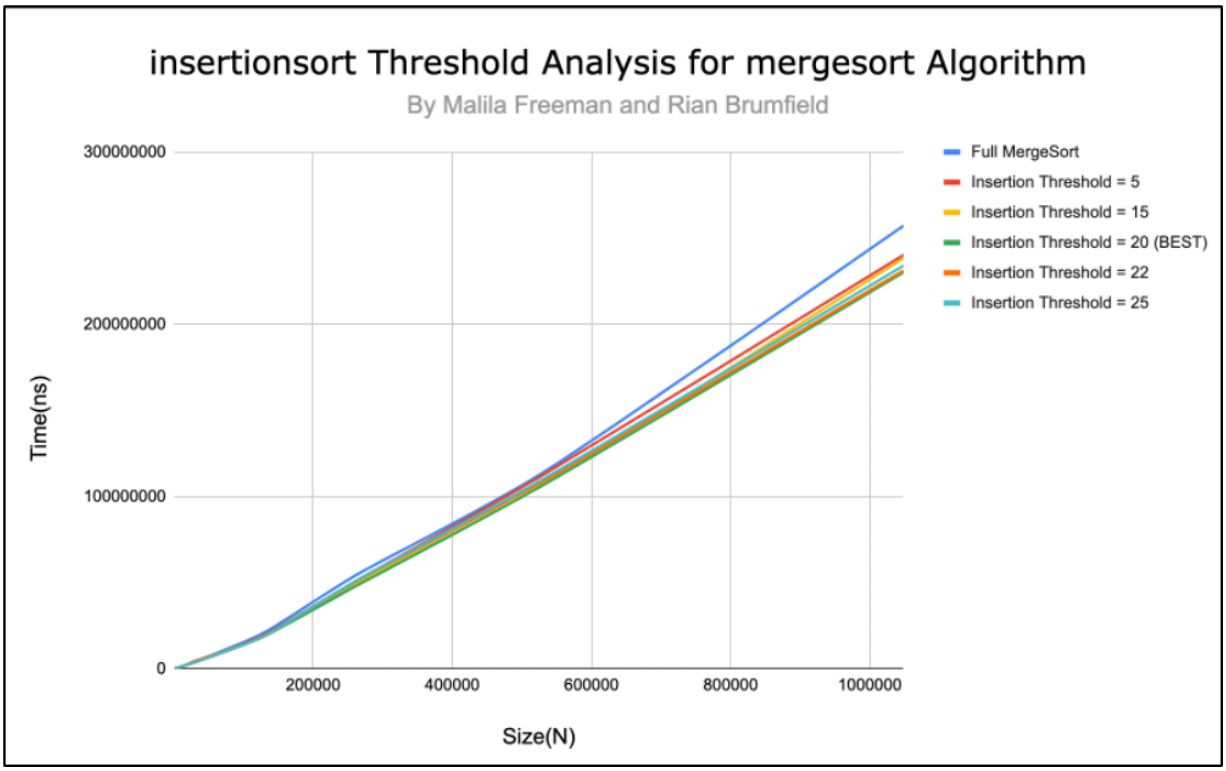
Rian Brumfield
November 23, 2021

Sorting Algorithms Analysis Document

1. Who are your team members?

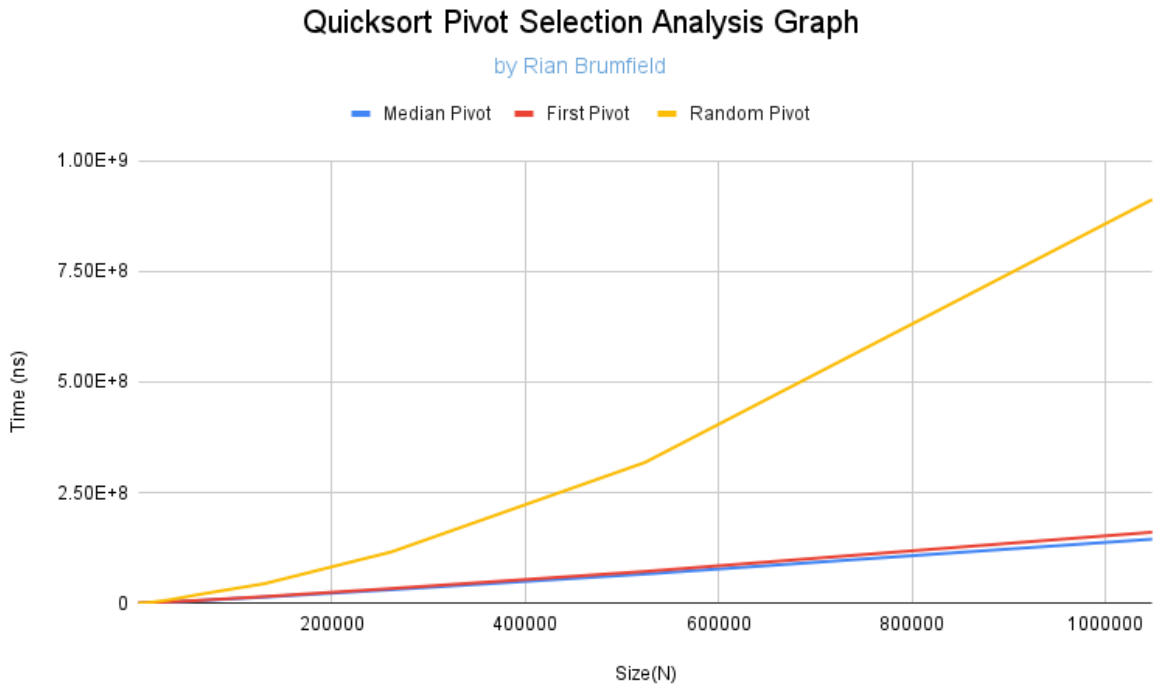
Malila Freeman

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).



In our insertionsort threshold experiments, we found that 20 appears to be the best threshold value for mergesort to switch over to insertionsort. We first started by plotting the running time for a full mergesort with no insertionsort used. We then plotted the running time for a mergesort with insertionsort thresholds of both 5 and 20, and saw that 20 was the more effective threshold of the two. Next, we plotted the running time for mergesort with an insertionsort threshold of 25 and saw that this performed worse than 20. We then plotted the running time for mergesort for an insertionsort threshold of 15, and saw that this also performed worse than 20. Lastly, we plotted the running time for a mergesort for an insertionsort threshold of 22, and saw that this also performed worse than 20, so we tentatively claimed that 20 was the best insertionsort threshold value for our mergesort algorithm. While this can be inferred by looking at the graph, we also verified our result by comparing the running time data for a threshold of 20 with the running time data for thresholds of 15, 22, and 25. For each value of size(N), we subtracted the running time data for thresholds of 15, 22, and 25 from the running time data for a threshold of 20. As can be seen in the rightmost 3 columns in the table above, the subtracted values were, in general, negative for each size(N), suggesting that running time values for a threshold of 20 were indeed smaller than the running times values for the other thresholds. This solidified the hypothesis we initially made after looking at the graph and lead us to conclude that 20 was the best threshold value for mergesort to switch over to insertionsort.

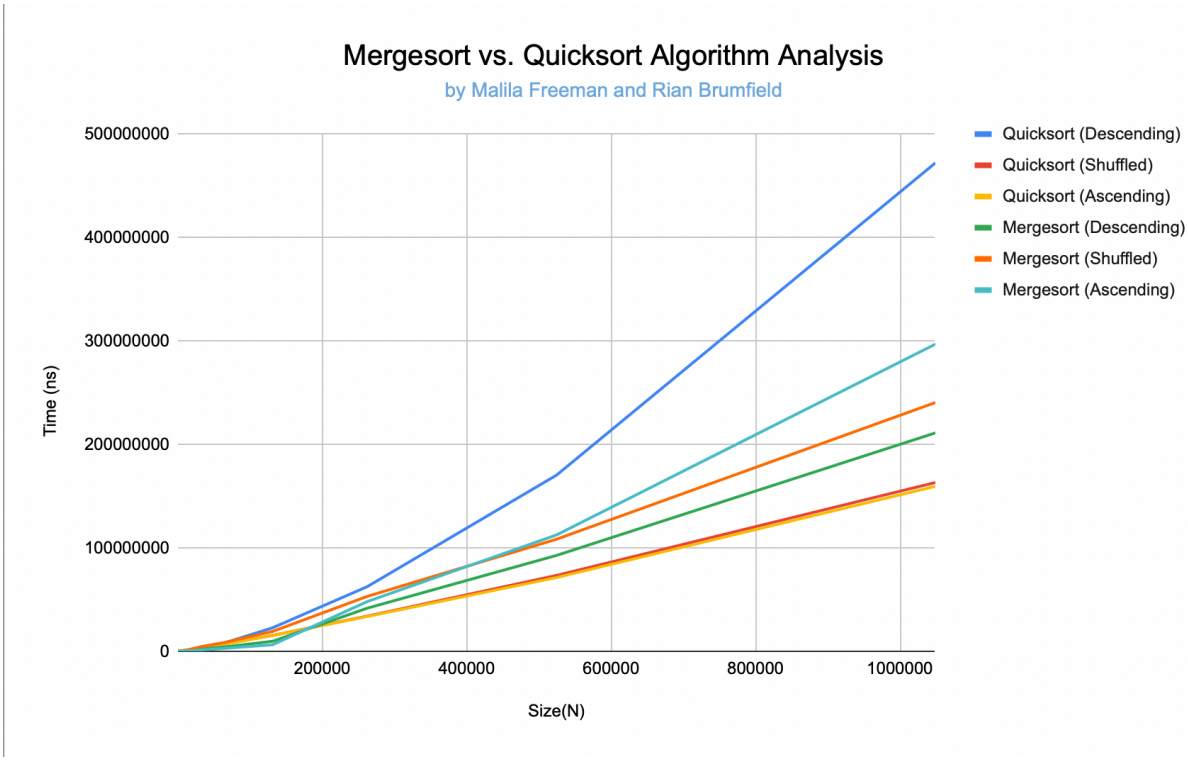
3. Quicksort Pivot Experiment: Determine the best pivot-choosing strategy for quicksort. (As in #2, use large list sizes, the same set of permuted-order lists for each strategy, and the timing techniques demonstrated before.) Plot the running times of your quicksort for three different pivot-choosing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



The most effective pivot selection is the Median Pivot selection. In this option, the median between the first, middle, and last elements is chosen as the pivot.
The second most effective pivot selection is always choosing the first element. This is only due to the fact that the ArrayList to be sorted was shuffled and in no sorted order. If the ArrayList were already sorted, this pivot selection would result in the worst case of Quicksort.
The first two pivot selections are very close in runtime, I assume because the least element is sometimes also the median.
The least effective pivot selection was the Random pivot, I think due to its inconsistency, sometimes choosing effective pivots and sometimes choosing really ineffective pivots. Also, the time it takes to reseed the random and get a random element drives the time cost up.

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. Forthe 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

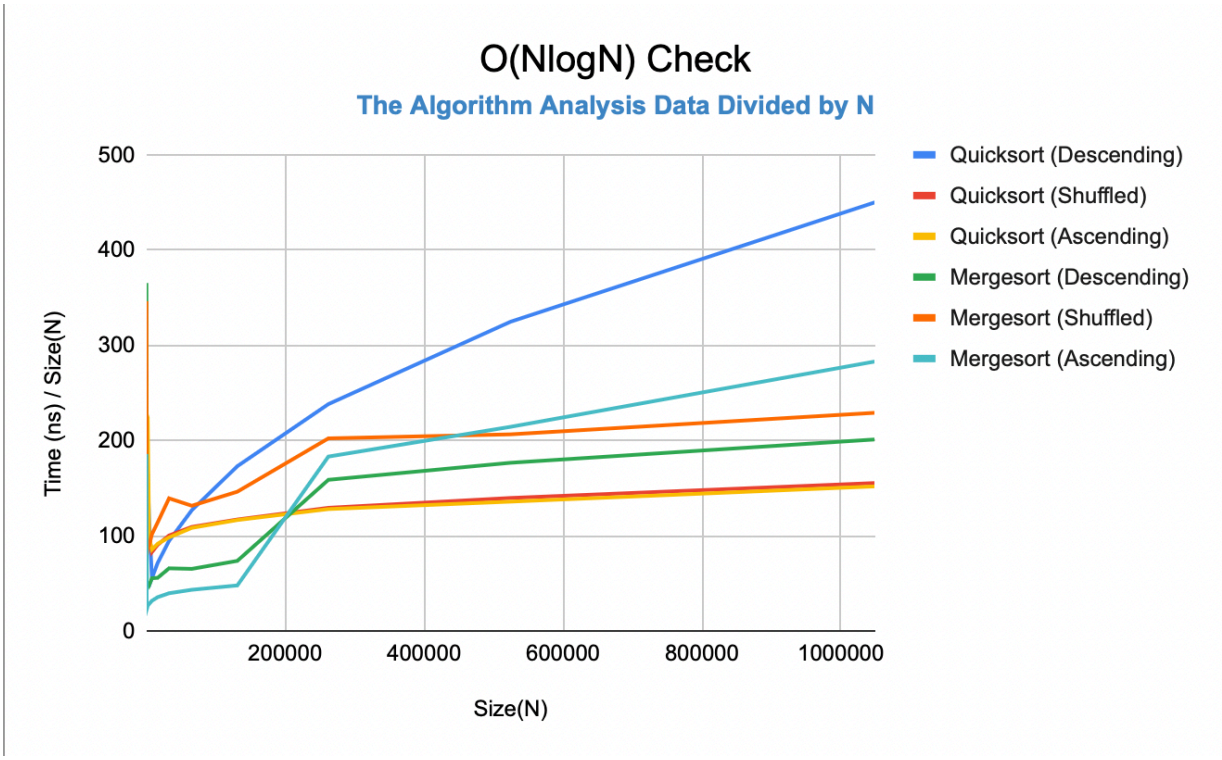
once or create three plots (one for each category of lists).



According to our timing experiments, Quicksort is the most effective sorting algorithm in the case where the ArrayList is already sorted, or if it is shuffled. However, the Mergesort algorithm runs the fastest on a descending order ArrayList. In order to guarantee fast running time with the Quicksort algorithm, the ArrayList should be shuffled before sorting.

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 average case for Mergesort and Quicksort are O(NlogN) and all timing experiments resulted in O(NlogN) behavior as anticipated. Because T = NlogN, it is true that T/N = logN. Therefore, as an extra measure to ensure the data is in fact O(NlogN), we created a graph of T/N data to see if the resulting lines are logarithmic. The resulting graph is provided below.



This graph increases our confidence that our data is correct.

The running times of Mergesort are mostly what I expected. An array's sortedness had little impact on the runtime of the algorithm because it only hastened the base case insertion sort by requiring less swaps. However, in all three cases, Mergesort will split the array the same amount of times. What I did not expect to see was the descending order ArrayList being sorted faster than the ascending order ArrayList. I would expect a sorted ArrayList to sort faster because it requires no swaps.

Similarly, Quicksort's run times were what I expected. The Ascended order ArrayList, represented by the YELLOW LINE, was sorted the fastest because the Median Pivot selection will always result in the true median in a sorted ArrayList. Therefore, the pivot selection is ideal and previous sortedness means no swaps will be required, causing sorting to be very fast. The Descending order ArrayList, represented by the BLUE LINE, will also choose the true median, the ideal pivot, but every single element must swap sides on the pivot, causing sorting to take much longer. Although the shuffled array is not guaranteed a perfect pivot selection, it will likely choose a fair pivot and require far fewer swaps than the descended order ArrayList. This is why we see the shuffled ArrayList, represented by the RED LINE, taking longer than an Ascending order ArrayList but much faster than a Descending order ArrayList.

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.