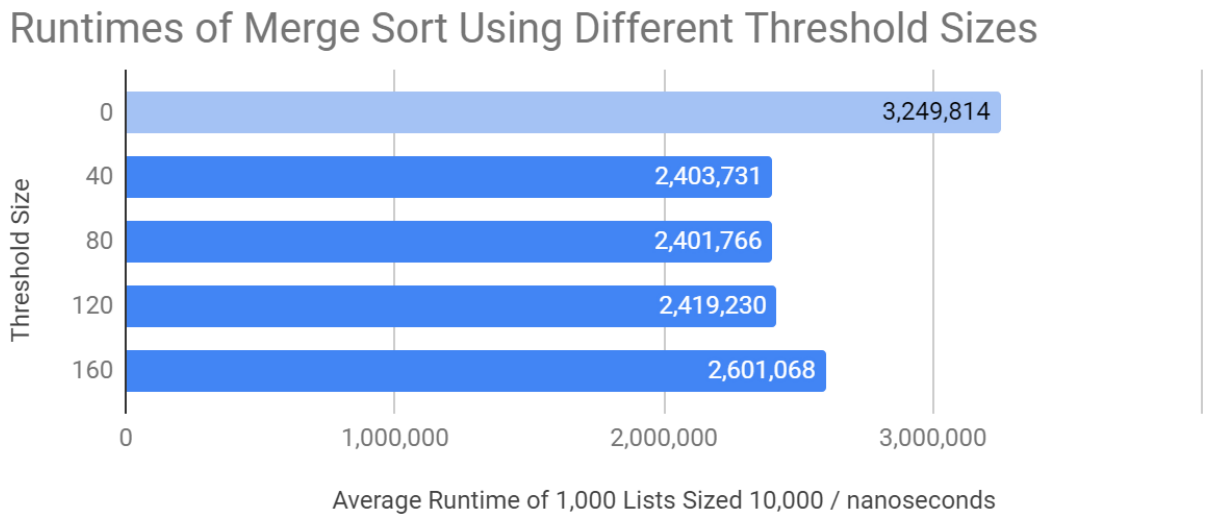
Assignment 5: ArrayList Sorter

Analysis Document by Qianlang Chen (u1172983)

As in all previous programming assignments, I had Brandon Walton as my partner in this assignment too. During the completion of the *ArrayListSorter* class, we encountered plenty of problems and errors in our design and codes, but since Brandon was so experienced and smart that we always quickly figured out what to change with our code, and this made us work together efficiently. I will definitely plan to work with him again in future assignments.

We implemented our sorting methods to have a Merge Sort and a Quicksort. We understood that the Merge Sort should always have an **NlgN** runtime behavior, regardless of the size of the list it sorts, due to its mechanism of sorting. Since Merge Sort is a recursive algorithm, it requires us to make a base case where it does not sort lists having less than 2 elements. We also realized that calling Merge Sort for small-sized lists are no more efficient than even an Insertion Sort, so we implemented a special Insertion Sort helper method that only sorts a specific range of a list. We made it to be invoked when the sub-list divided by the Merge Sort is less than some threshold and replaced the regular base case of the Merge Sort. The beginning and end indices of the sub-list are properly passed to the Insertion Sort method so that it only sorts that portion of the list without bothering the rest which would affect our runtime.

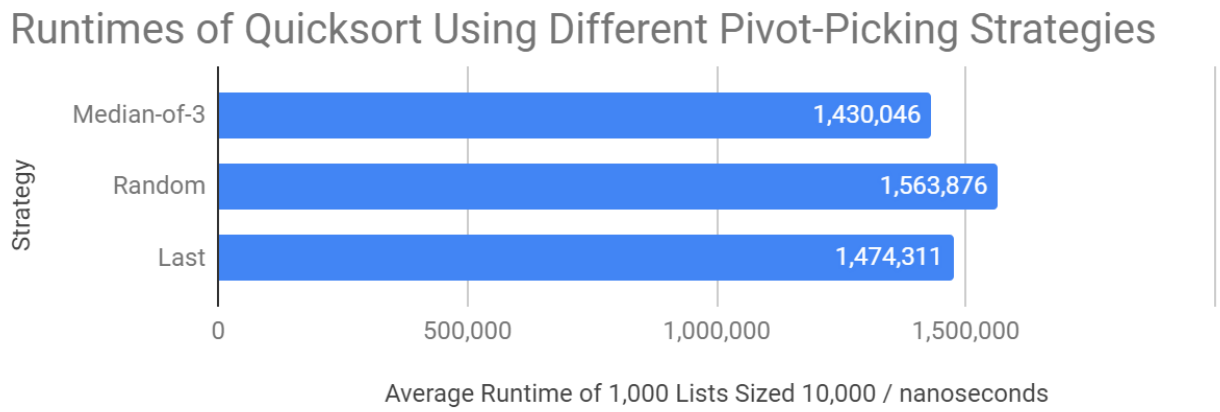
To figure out the best threshold of when to switch to Insertion Sort from Merge Sort, we did a timing experiment for different threshold sizes. Our results are in the chart on the next page, note that the threshold size of 0 indicates the runtime of using pure Merge Sort (without switching to Insertion Sort):



According to our results of experimenting the runtimes, a threshold size of 80 works the best. Therefore, we implemented it into our class so that our Merge Sort will switch to Insertion Sort when the sub-list to sort has 80 elements or less.

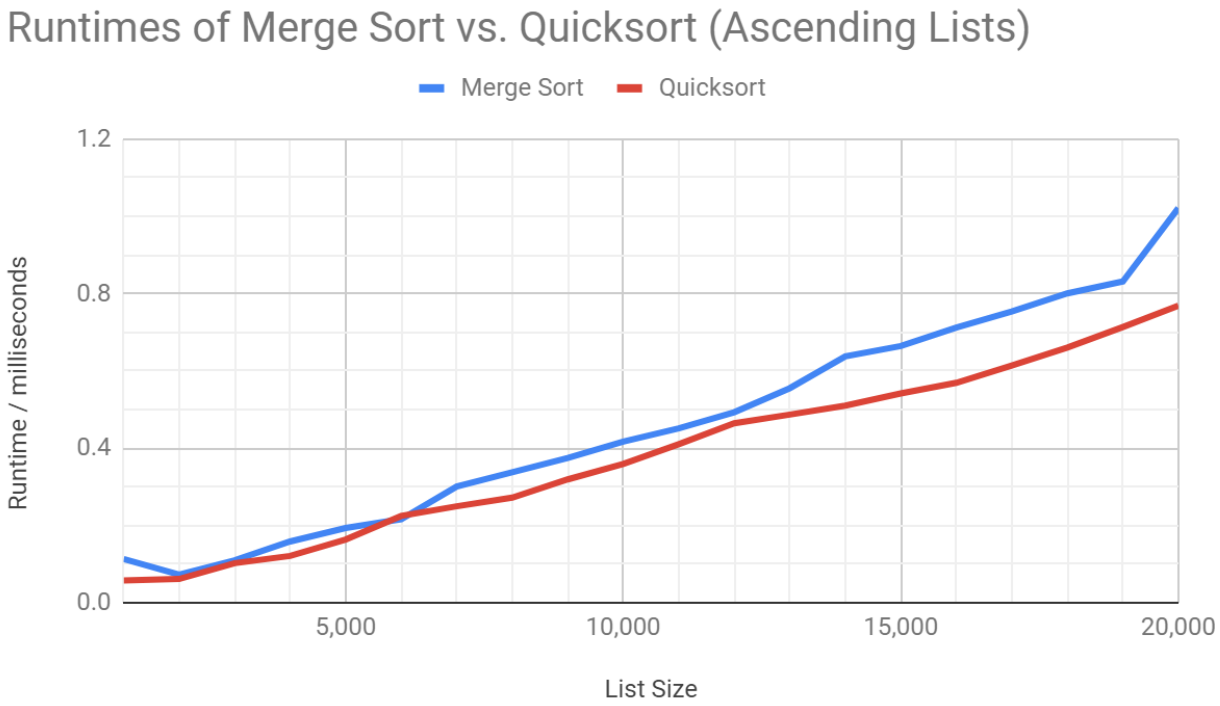
The Quicksort algorithm, on the other hand, depends on some luck of the quality of the pivot it chooses during its partitioning process. It has an **NlgN** runtime behavior in the best or average case scenario but has a horrible **quadratic** behavior in the worst case, where it always chooses the smallest or greatest elements in the list, causing the list to not be properly divided into halves. The order of the list to be sorted does affect the runtime of Quicksort, since there would be less number of elements needed to be swapped in an nearer-to-sorted list. However, this is not significant enough to change the overall runtime behavior of the algorithm.

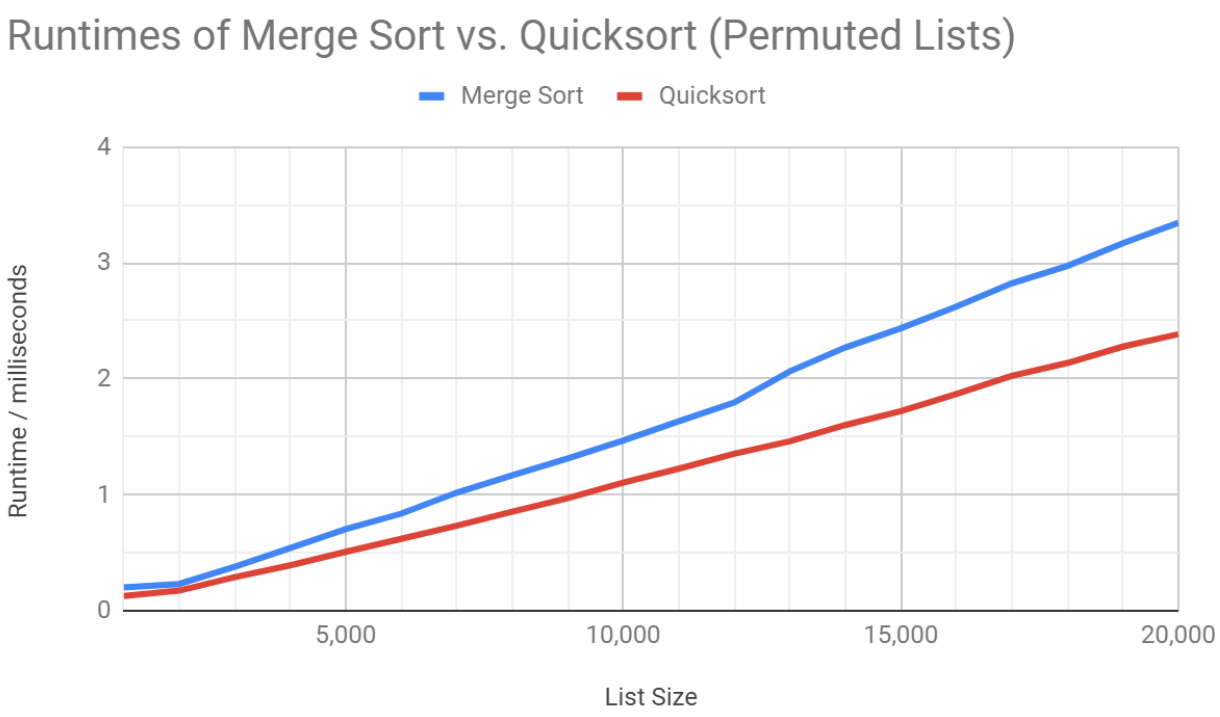
To experiment how different strategies would affect the runtime of our Quicksort, we implemented and used 3 different strategies to finding the pivot during the partitioning process: always choosing the last element, choosing a random element, and choosing the Median-of-3 (the median out of the first element, the middle element, and the last element in the list). We timed our Quicksort using different pivot-picking strategies, and here are the results:

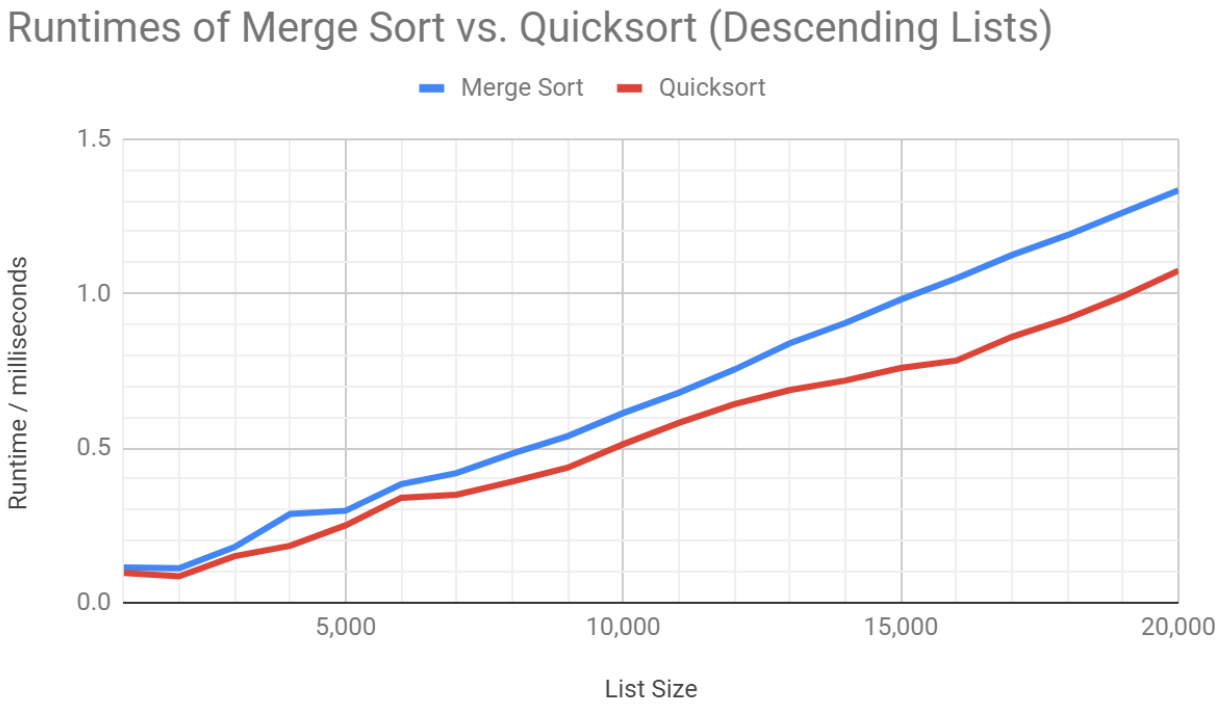


According to our results, always using the Median-of-3 as the pivot turns out to be the best strategy out of the three. This matches our prediction, and using the Median-of-3 method, in theory, can effectively reduce the chance of picking a bad pivot, as opposed to always using the first or last element as the pivot, in the case of the list is nearly or completely sorted either ascending or descending, which gives a horrible quadratic runtime behavior.

To find out which sorting algorithm is better out of the Merge Sort and the Quicksort in different situations (the lists are sorted ascending, descending, or randomly permuted), We timed each sorting method (with their best settings) in each situation. Here are our results:


Runtimes of Merge Sort vs. Quicksort (Ascending Lists)
Merge Sort
Quicksort
1.2
0.8
0.4
0.0
Runtime / milliseconds
5,000
10,000
15,000
20,000
List Size


Runtimes of Merge Sort vs. Quicksort (Permuted Lists)
Merge Sort
Quicksort
4
3
2
1
0
Runtime / milliseconds
5,000
10,000
15,000
20,000
List Size



Based on our results and the graphs, the Quicksort is better than the Merge Sort. We then carefully used the "Check analysis" technique and took the **T(N)/F(N)** method to confirm that their behaviors are all **NlgN**, and they match our predictions. The Merge Sort breaks the list in halves until it is completely in one-element pieces (even we used the Insertion Sort, its data size and behavior is insignificant) and uses a linear function to merge them back into one. The Quicksort's partition is linear, and it also breaks the list into halves. Therefore, both of these sorting algorithms have such behavior.