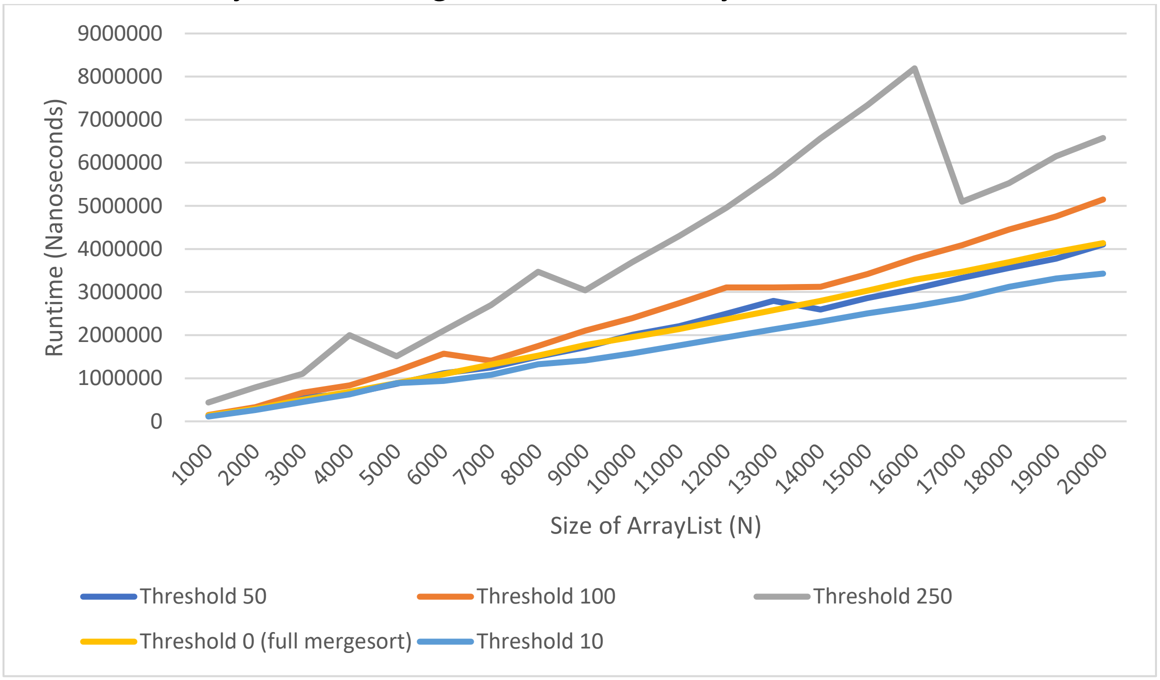
Assignment 5: ArrayListSorter Analysis Document

1. My partner and I spent about fifteen hours programming and testing this assignment. My partner and I worked well together, though there were a couple issues this assignment. We did struggle a bit more with communication, there were a few times where he started explaining code to me that I had just written, even though I didn't ask about it and clearly understood what I was doing in that section (considering I was the one that wrote that specific code). We also disagreed a few times on how to go about implementing the code, but that is normal and we were able to still compromise and work together to create a functional code. The second issue is that on Monday evening, he reached out to me and told me that our runtime complexity for mergesort was O(N^2). I told him that we needed to redo the code, because the assignment explicitly says that if we get that runtime complexity, we did it wrong and need to redo it. However, he told me he didn't have time to look at the code so I assumed that if I wanted to fix it I would have to do it alone. I was not alright with this, as I don't feel right about doing the work to fix the code and still giving him all the points even though he did not help fix the code. I messaged the instructors about it and they told me to mention it here. I completely understand time constraints, but as he reached out Monday evening, we still had plenty of time to look over it and I did not like that he didn't choose to make time to fix our code. When I let him know that I would be submitting on my own with the fix, he asked if we could call to sort things out, which frustrated me because he did not have the time to call to work out code until I had the correct code and was going to submit on my own. He did help me with a different error in our code, though, so I did appreciate that help. However, he did not discuss that error with me until long after I had told him about submitting our own work, and while it would have been ideal to catch those errors in testing, I also don't think it was good partnership for him to not mention the errors to me until after I had the answer to the other error. Apart from the errors, he contributed about equally to the assignment, so most of the assignment was done together. We used CodeTogether in Eclipse which made the process a lot easier, because we were able to scroll through the code at the same time so while we were discussing, we actually could see/analyze what the other person was talking about. I plan to work with my partner again, as I feel we work well together overall and usually put an equal amount of effort into the assignments. Though we did have some issues this assignment, it could be far worse and usually he is a good partner.
2. The expected growth rate of mergesort in the best case is O(NlogN), average case is O(NlogN), and worst case is O(NlogN). Mergesort is unique in that it always has a complexity of O(NlogN). The ordering of the list doesn't affect the running time of mergesort, because regardless of the way it's organized, the computer still has to make the exact same number of comparisons. There is no statement that checks if it is already organized or if it has become organized, so it must run all the way through the algorithm every time even if it's already sorted, because the computer doesn't know that it's sorted until the algorithm has completed.

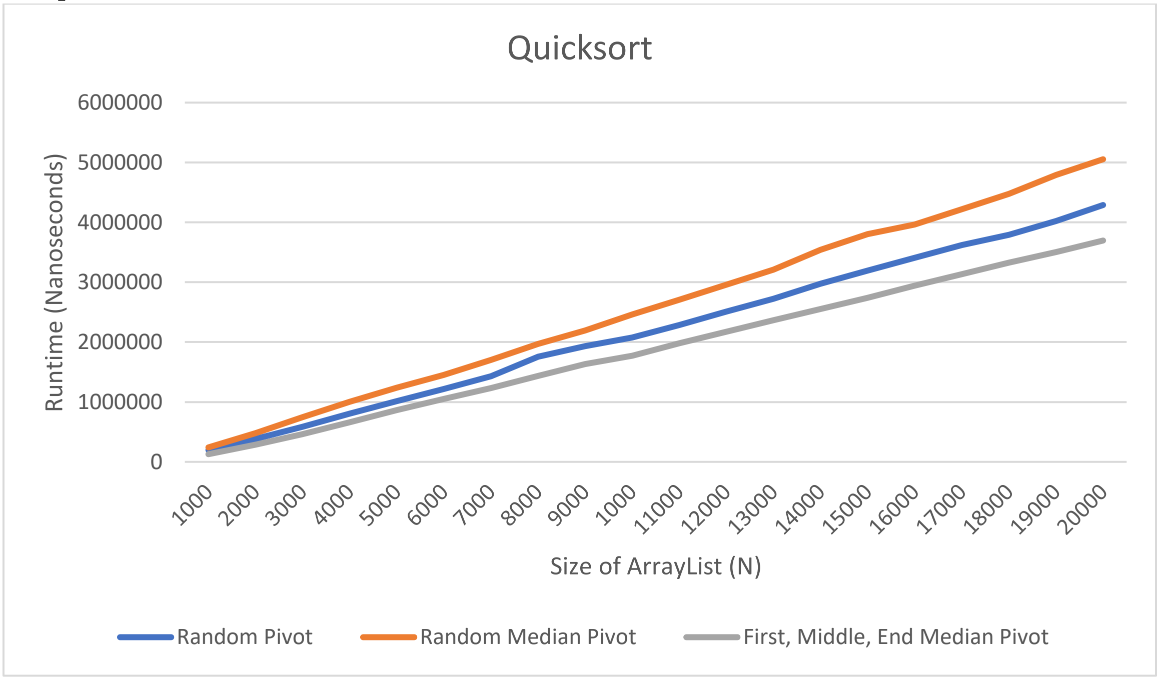
3. In the recursive method for mergesort, we began with an if statement that said if the start index subtracted from the end index is less than or equal to the threshold value, then call insertion sort on the array from start index to end index. The parameters for start index and end index for insertion sort ensure we don't call the insertion sort on the entire array. Instead, we do two for loops that start a variable i at startIndex + 1, and loop while it is less than the given end index. The second for loop starts a variable j at i-1, and loops while it is larger than the start index. This way, we are only looping over the specific section of the array we are looking at, not the entire array.



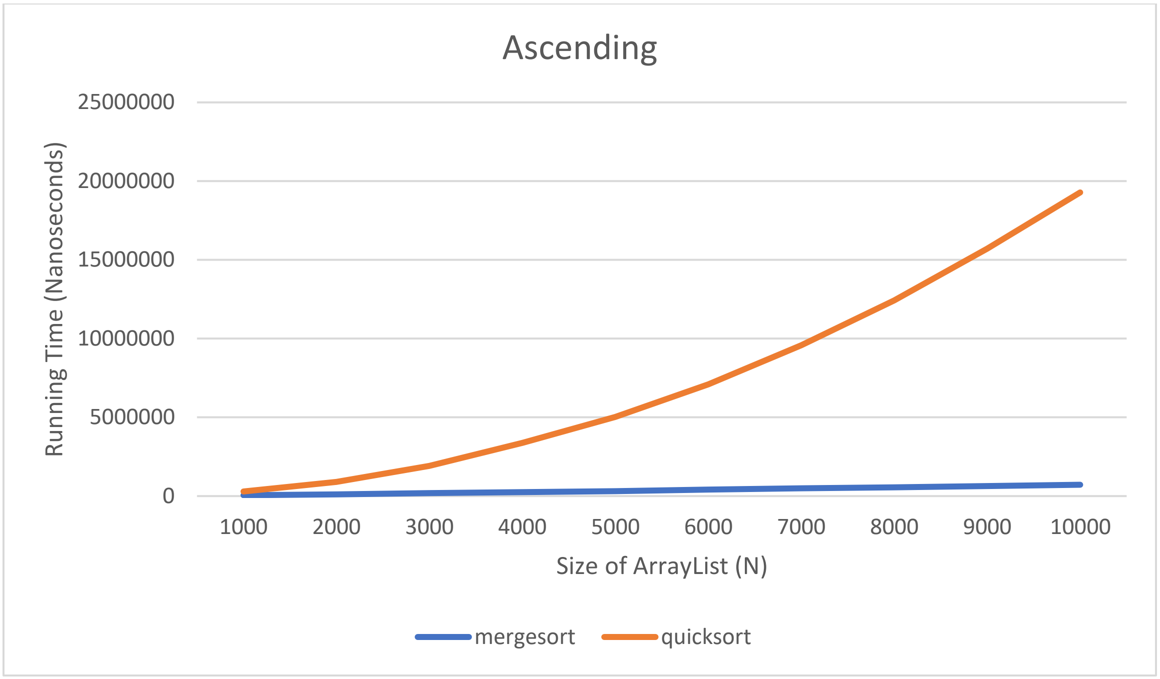
4. The best insertion sort threshold value from the list that I chose to experiment with was 10. It clearly ran faster than the rest of the threshold values, and it was the most consistent. In contrast, the threshold value of 250 was very inconsistent and took much longer than the other values.
5. The best and average cases for quicksort are O(NlogN). The worst case is O(N^2). The ordering of the array doesn't actually end up affecting the run time of the quicksort, because the only thing that really impacts it is the pivot. Similar to mergesort, the computer doesn't know whether or not the array is sorted until it reaches the very end, or the base case in our implementation. Thus, it will partition and sort all the way to the end regardless of whether or not it's sorted/what order it's sorted in. However, the pivot selection does impact the running time. If you choose a bad pivot, such as index 0 on a sorted list, you will end up with terrible partitions as there will be no values less than the value at index 0, and you will end up eventually going through the entire array. This is seen by the fact that the worst case complexity is O(N^2), because if you choose a bad pivot there is a possibility that you will end up looping over the entire array N times, resulting in the quadratic complexity. However, because the sorting of an array can affect whether or not your pivot is good, the sorting of the array (sort of) impacts the running

time, but very little and only through said pivot selection. A good pivot selection strategy will be effective regardless of the ordering of the array.

6. The first strategy we used for our quicksort implementation was to choose a random pivot index from the given set of indexes. The Big-O complexity for this strategy is O(1), because regardless of the list size, it just picks one random value between that given range. This doesn't really impact the overall Big-O behavior of quicksort, unless it chooses a poor pivot value, in which case it would increase the Big-O complexity to N^2. The second was to choose three random indexes from the arrayList, get the associated values, sort them using insertion sort, and then choose the median of the values. The Big-O complexity of this is either O(N) or O(N^2) because we call insertion sort. However, in terms of the overall complexity, this actually is also an almost constant complexity, because no matter the size of the list, we will always be choosing three values to find the median. Thus, it doesn't add too much to the overall complexity. The third was to choose the first, middle, and end values of the section of the arrayList, sort them using insertion sort, and then choose the median of those values. The Big-O complexity for this is similar to the previous method, either O(N) or O(N^2) because we call insertion sort. However, again, it doesn't greatly impact the overall complexity because we always randomly choose three values, no matter the size of the arrayList so it is more constant than anything. All three of the above strategies impact the overall complexity if they result in a bad pivot selection, which results in slower run times.

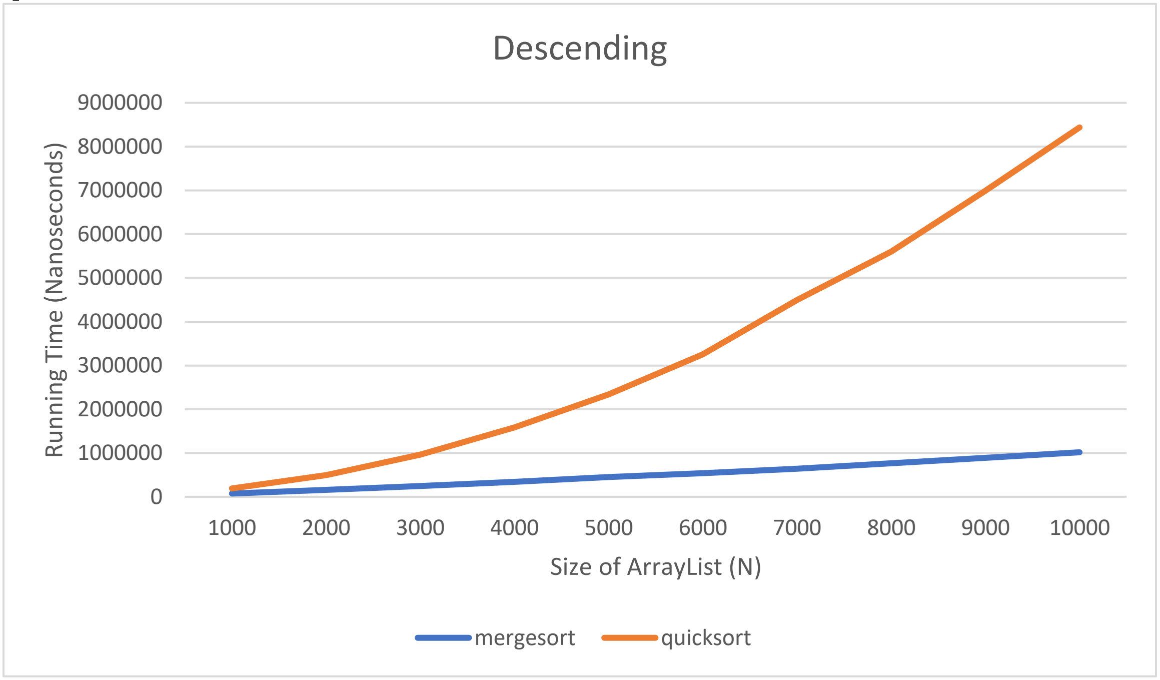


7. The best quicksort strategy out of the three I tested was clearly First, Middle, and End Median pivot selection strategy. It is clearly faster than the other two.

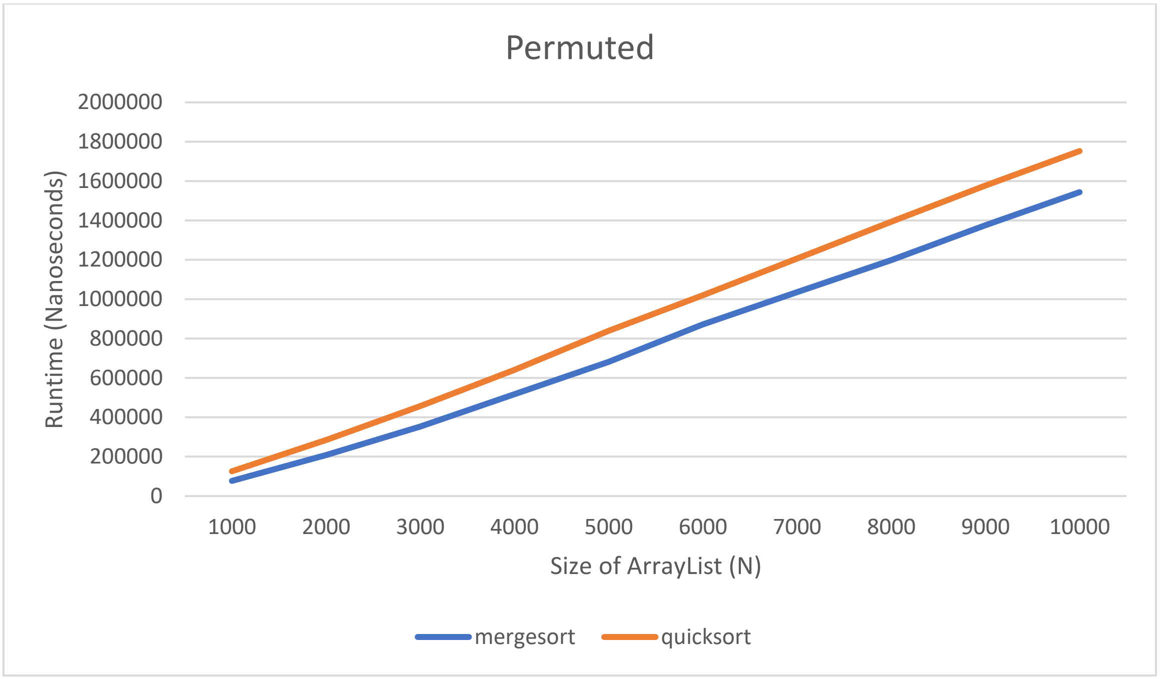


8.

For lists that are in ascending order, mergesort is clearly much more efficient than quicksort.



For a descending list, mergesort is faster than quicksort.



For a permuted list, quicksort is faster than mergesort but they're still fairly close in run times.

9. The running times of my sorting methods do actually exhibit the expected growth rates. This is clear from the graphs of the original run time plots of each algorithm. Both seem to be a graph of NlogN which is the expected average complexity for each sorting algorithm. Additionally, when I calculated the ratio between the expected growth rate and the actual growth rate, I got roughly the same proportion across each sorting plot. I determined the growth rate from the trend of the plotted line, the calculations of T(N)/F(N), as well as estimating the general growth rate from my code (looking at the code and determining what the Big-O complexity will be).