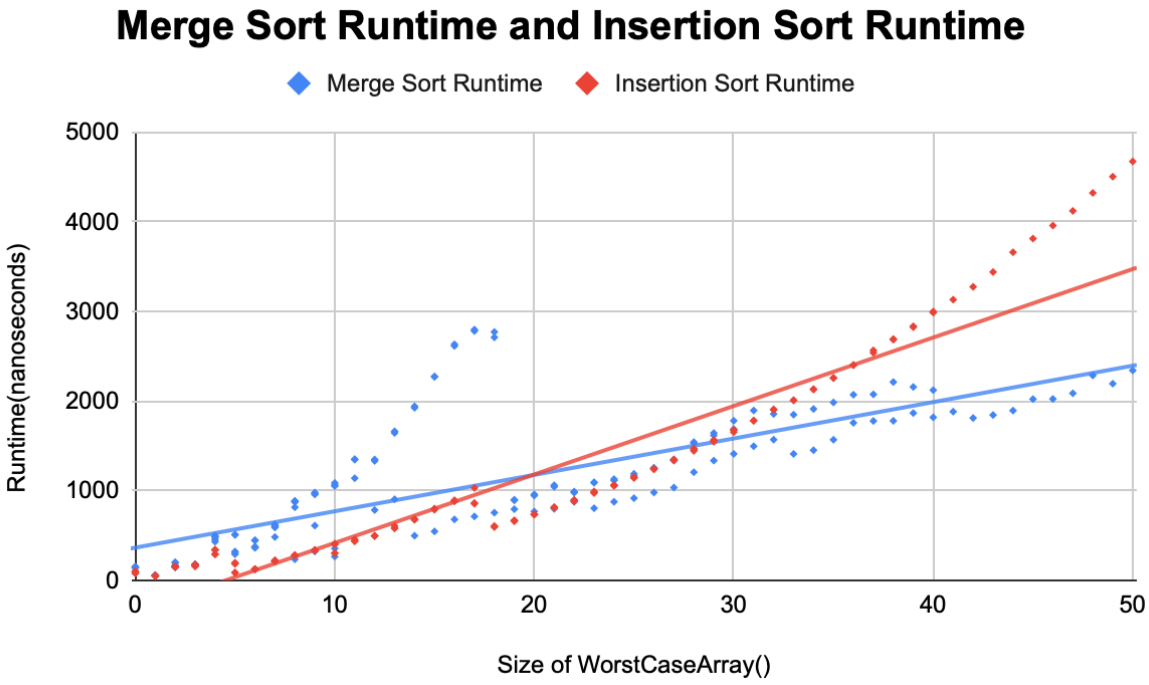
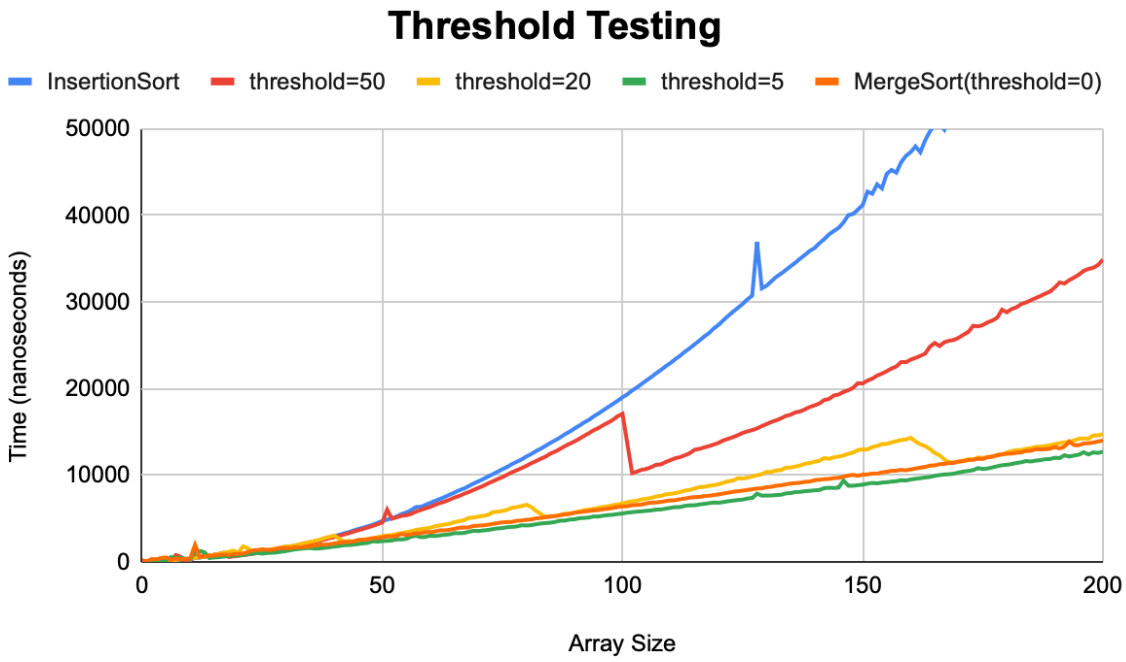
Assignment4: MergeSort and QuickSort
Samantha Pope's Analysis Document

Mergesort Threshold Experiment

For varying sizes of ArrayLists, I tested the generateWorstCase scenario. I tested arraylists from size 0 to size 50 against each other. I used 10,000 iterations to perform all of these tests. From the outcomes of these tests, the threshold value that would best suit my algorithm is around size 20. This is where the growth of insertion sort starts to become larger than the merge sort runtime.



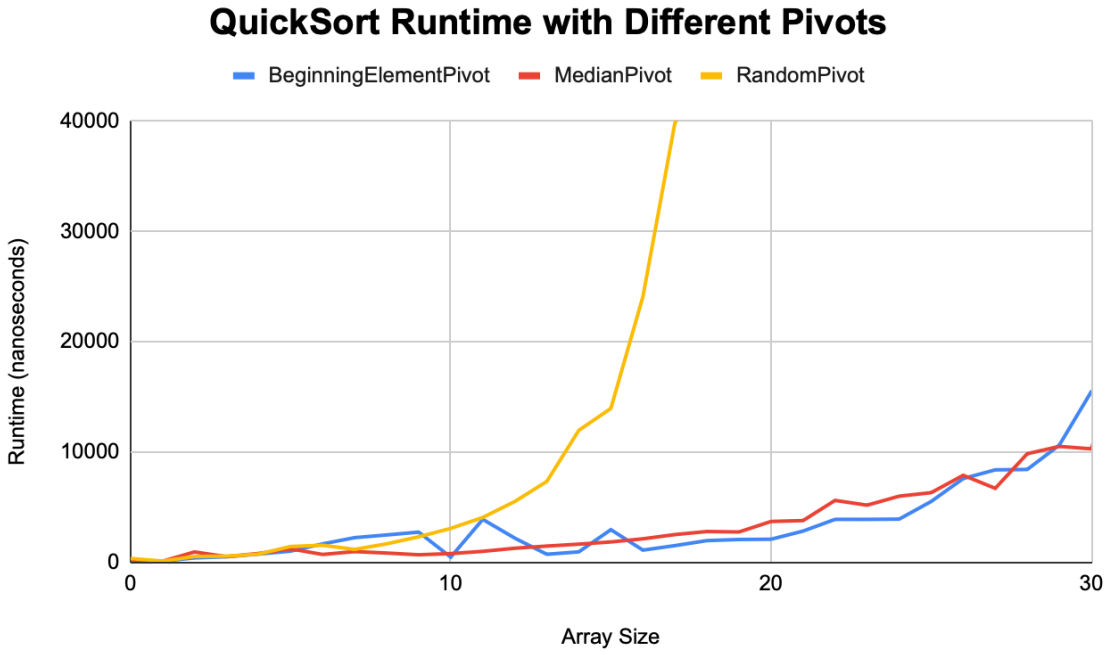
I tested five different sorting algorithms for arrays with sizes 0 to 200. The threshold was manipulated to three different variables: five, twenty and fifty. The threshold at fifty performed significantly worse than the smaller thresholds(see red line). The threshold at twenty, five and zero(only calling mergesort) performed similarly for cases of this size. The threshold of five performed the best in my analysis.


Threshold Testing
InsertionSort
threshold=50
threshold=20
threshold=5
MergeSort(threshold=0)
Time (nanoseconds)
50000
40000
30000
20000
10000
0
0
50
100
150
200
Array Size

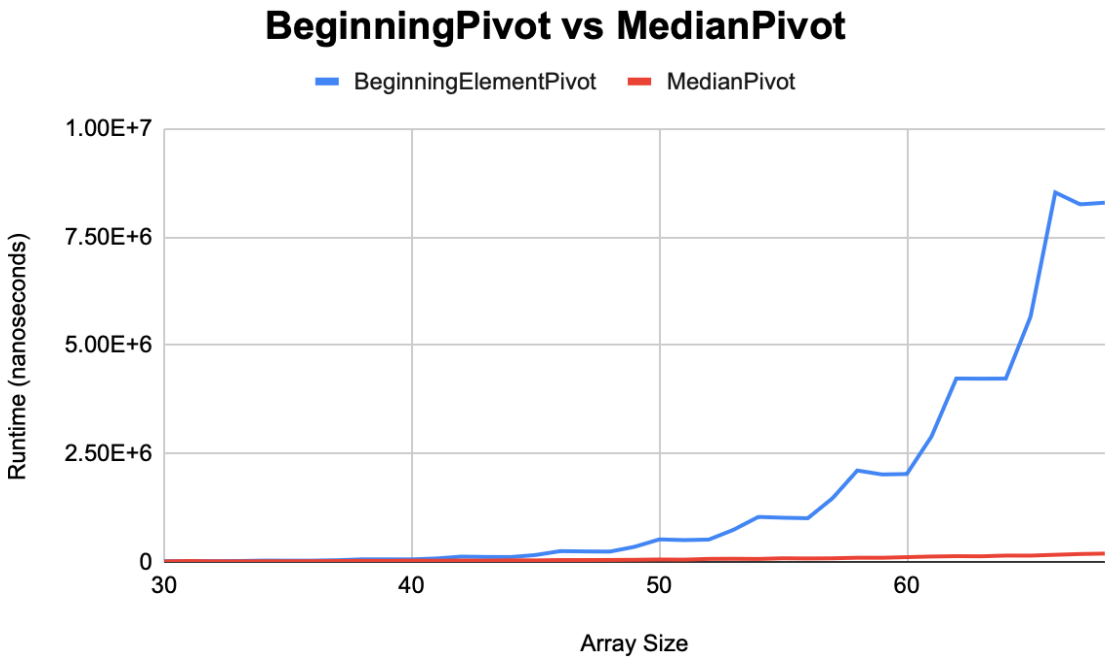
Quicksort Pivot Experiment

I tested three separate pivot experiments for the QuickSort method. My first method was choosing the first element of the array (shown in blue). The second method found an approximate median by taking the first, last and middle element of the array and finding the index of the median value (shown in red). The third method of pivot finding was a random pivot using a random generator (shown in yellow).

The first graph shows that the randomPivot method is extremely ineffective. I think that part of the reason why this takes so much more time to run is because of the cost of finding a random element inside of the quicksort function. I originally expected this random pivot to be better than the beginning element, which I expected to be the worst element.



Since this graph did not show the difference between the beginning and median pivot choices, I made another graph with array sizes going from 30 to 60 with just the beginning and median pivot choices. The below graph shows that the beginning element is significantly slower than the median element. The worst case of quicksort is O(N^2). The blue graph starts to show exponential growth. The best case of quicksort would be choosing the exact median of each array. The approximate median grows extremely slowly and follows a big O of O(log(N)).


BeginningPivot vs MedianPivot
BeginningElementPivot
MedianPivot
1.00E+7
7.50E+6
5.00E+6
2.50E+6
0
Runtime (nanoseconds)
30
40
50
60
Array Size

Mergesort vs. Quicksort Experiment

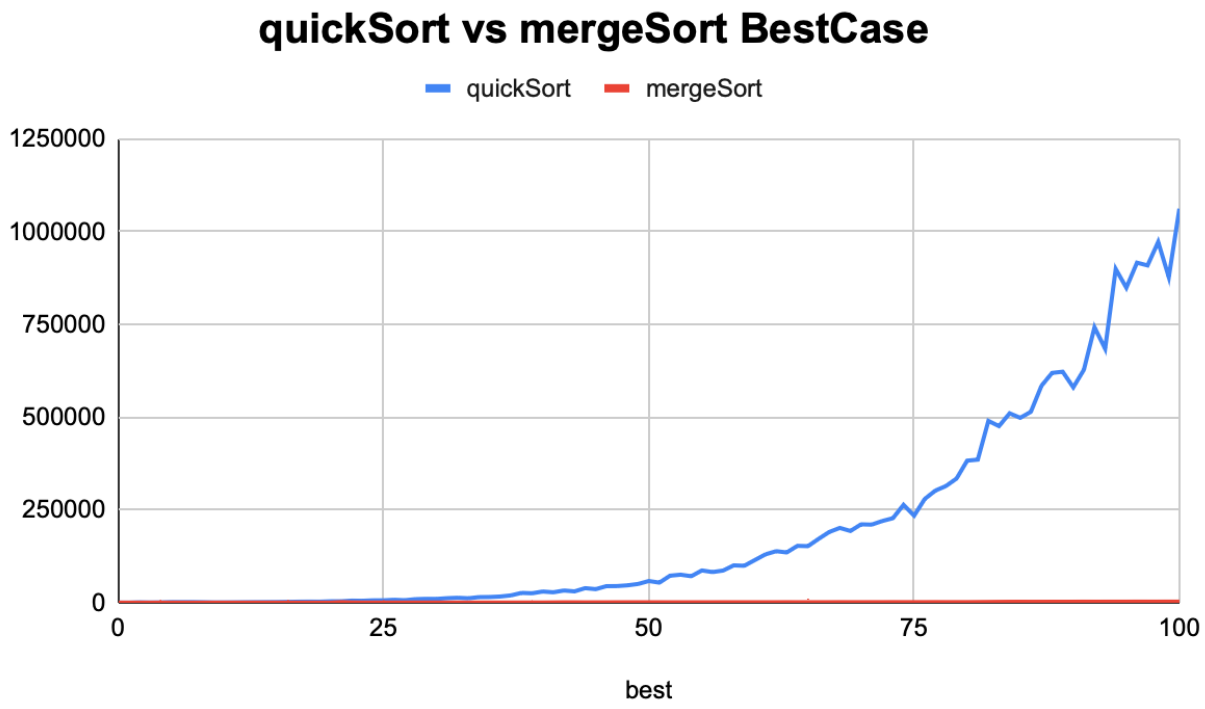
I used the median approximation for my pivot in all of the following quicksort vs mergesort experiments. I used a threshold of 20 in all of my mergesort functions, as that is where mergesort started to outperform insertion sort in my original analysis.

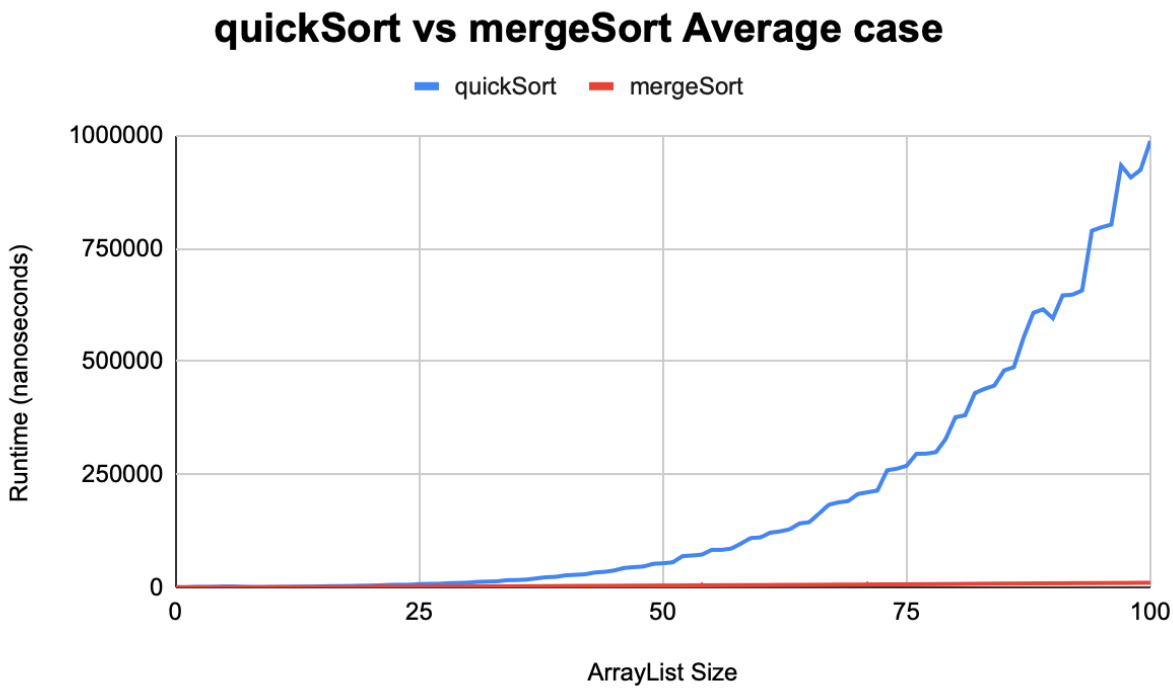
In all three of my tests, quicksort started to reach exponential growth at large numbers (>50). The quicksort method passed all of the tests that were written for it and was correctly sorting large and small arrays. However, the behavior of quicksort always showed O(N^2) for all three tests, using any pivot method that was written. According to all three of my cases that were tested, the runtime is significantly better for mergesort, which followed a logarithmic growth trend.

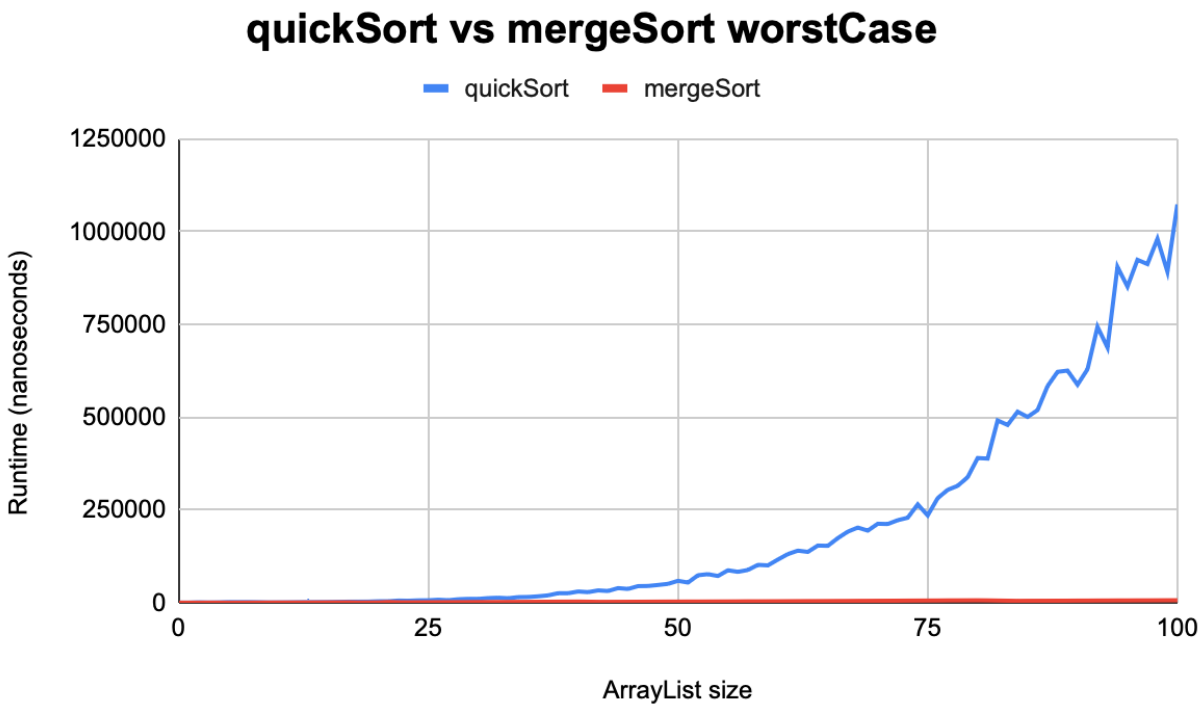
Do the actual running times of your sorting methods exhibit the growth rates you expected to see?

The growth rate for mergesort was what I expected. The graph followed a logarithmic growth trend. Mergesort is supposed to have a big O of O(Nlog(N)). This is due to the fact that mergesort works by continuously dividing N by two to get smaller and smaller pieces. These pieces are divided until they reach a one element and then are merged and sorted back together in the correct order. The “best case” scenario for mergesort is that you separate the array into two equal pieces, where one is larger than the median and one is smaller. The worst case is separating the array into two uneven pieces. An example of this is passing the minimum or maximum of the array. This would make the mergesort have to have N levels, each having a runtime of N. This leads to a bigO of O(N^2). We controlled for this by passing only the approximate median value. This successfully controlled for the worst case, as shown by all of the cases following a logarithmic growth rate.

The growth rate for quicksort was not at all what I expected. I found that the growth rate was O(N^2) for all three of my analyses. I expected O(log(N)). Like mergesort, quicksort touches each element in an array at each level (N runtime). The best case is having the least amount of levels, which leads to a runtime of O(Nlog(N)). The worst case is having N number of levels, which would lead to a runtime of O(N^2). If the pivot is between the 25th and 75th percentile, the runtime should be O(NlogN). This is why I was confident that our approximation of the median would make our returned index be within the middle 50%. I am not sure which part of the quicksort implementation that we used that is causing this exponential growth, but it is not at all what we had originally anticipated.


quickSort vs mergeSort BestCase
quickSort
mergeSort
1250000
1000000
750000
500000
250000
0
0
25
50
75
100
best


quickSort vs mergeSort Average case
quickSort
mergeSort
1000000
750000
500000
250000
0
Runtime (nanoseconds)
0
25
50
75
100
ArrayList Size


quickSort vs mergeSort worstCase
quickSort
mergeSort
1250000
1000000
750000
500000
250000
0
Runtime (nanoseconds)
0
25
50
75
100
ArrayList size