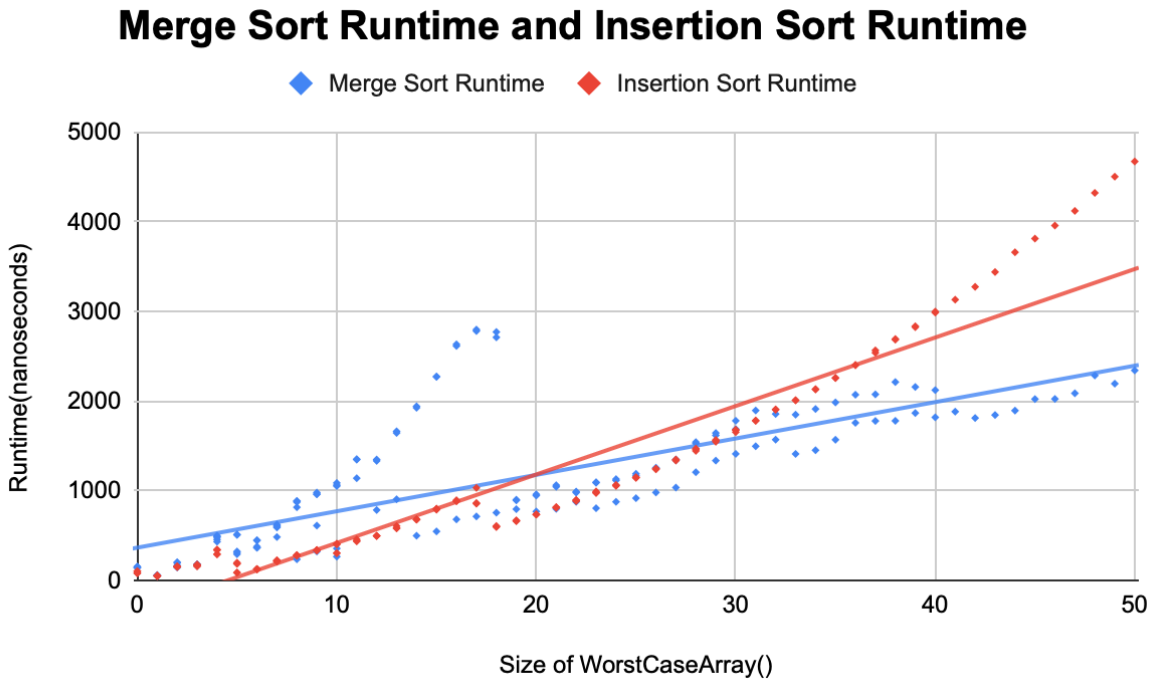
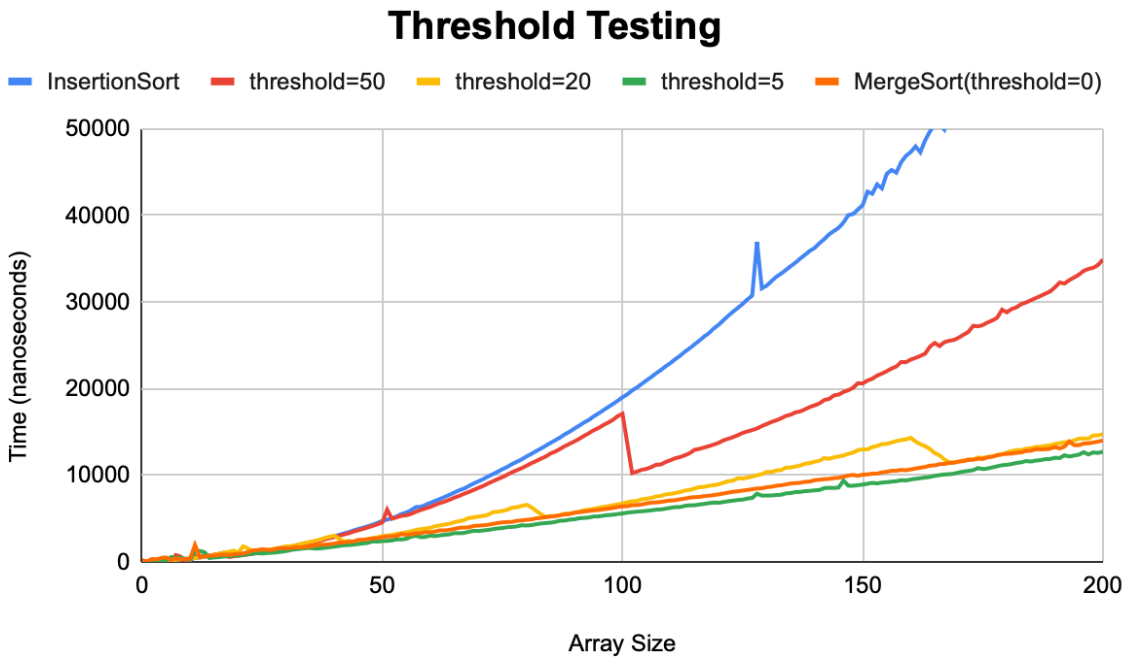
Assignment4: MergeSort and QuickSort
Samantha Pope's Analysis Document
Link to all of the original data:
https://docs.google.com/spreadsheets/d/1mEgu8CoXJv_5MrKIvUjLtSf4VvHjT2taD1ohOjXJguM/edit#gid=101277978

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.

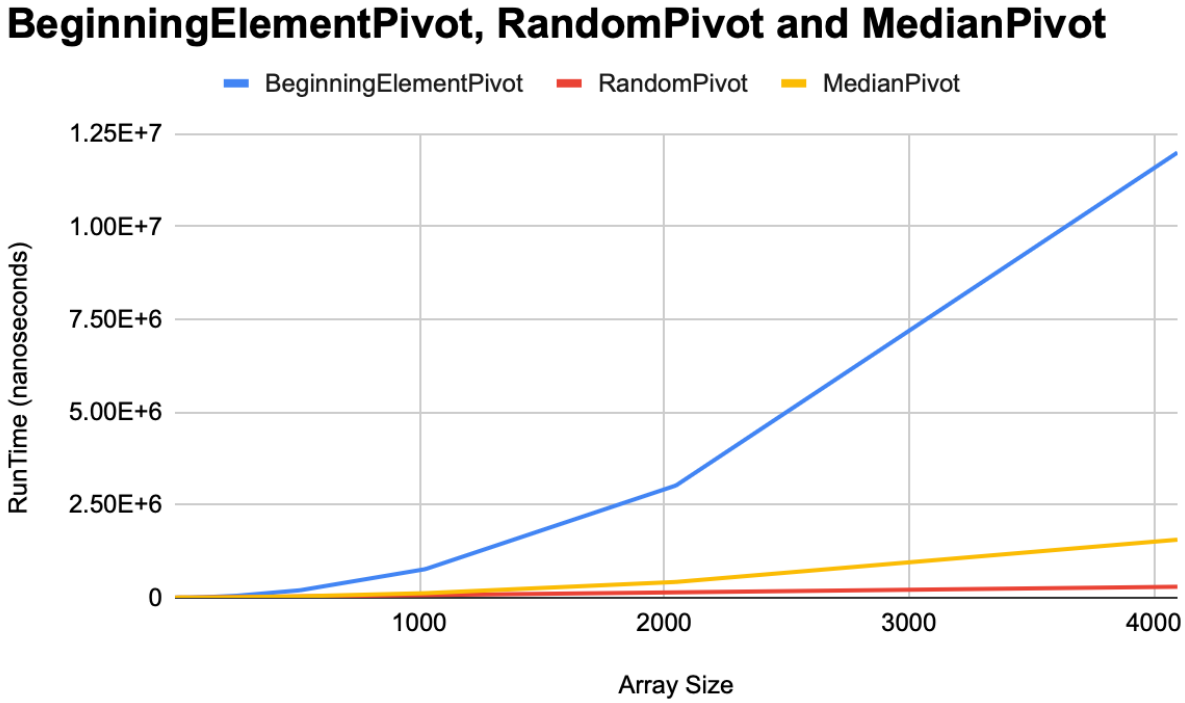

Threshold Testing
InsertionSort
threshold=50
threshold=20
threshold=5
MergeSort(threshold=0)
Time (nanoseconds)
50000
40000
30000
20000
10000
0
0
50
100
150
200
Array Size

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 yellow). The third method of pivot finding was a random pivot using a random generator (shown in red).

After we fixed our quicksort algorithm, I reran these tests to make sure that I was getting similar results. The error in our quicksort algorithm was in the recursive call to the "second" array that went from the pivot to the end element. I did not hypothesize that it would change, but I was wrong. My original analysis (prior to fixing the quicksort recursive call) is below in blue.

I found that the beginning element took significantly more time. This is expected, since it divides the groups into the most uneven pieces possible (1 in one "half", and N-1 in one "half"). The random pivot performed slightly better than the median pivot at smaller numbers, but as the size increased, the random pivot outperformed the median. I believe that this is due to us creating another array in the medianPivot method we used. I think the time to create a new smaller array, sort it, and pull out an element is taking significantly more time than the built in Random generator.



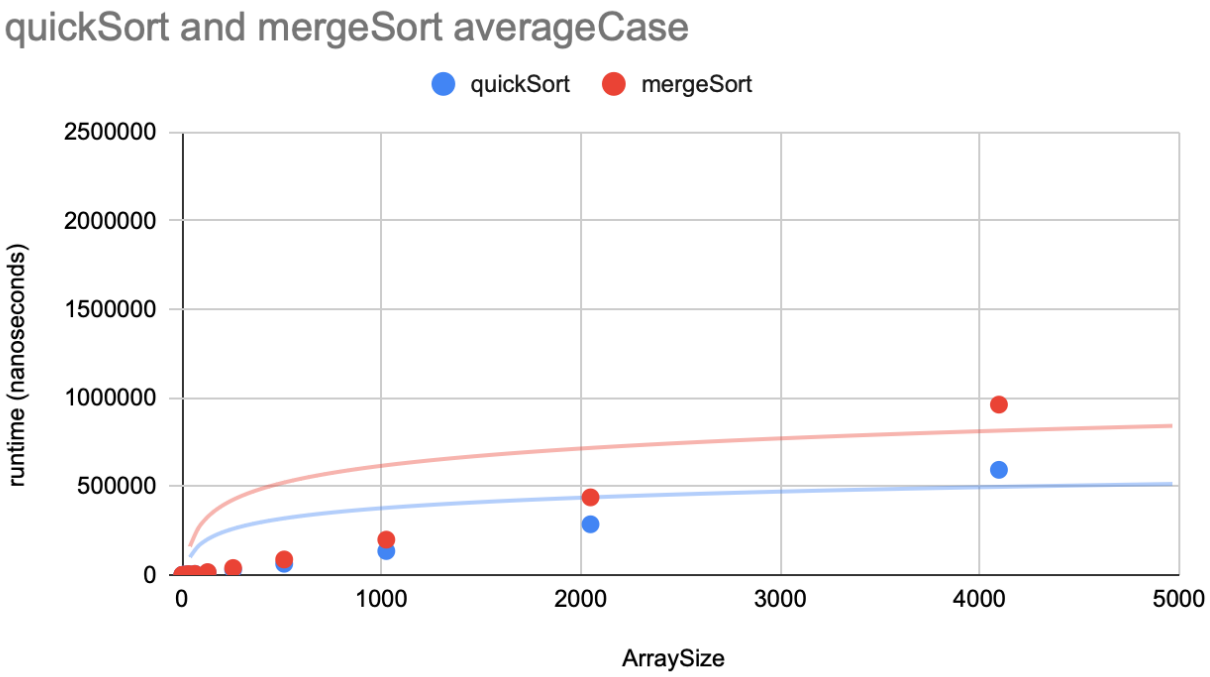
Mergesort vs. Quicksort Experiment

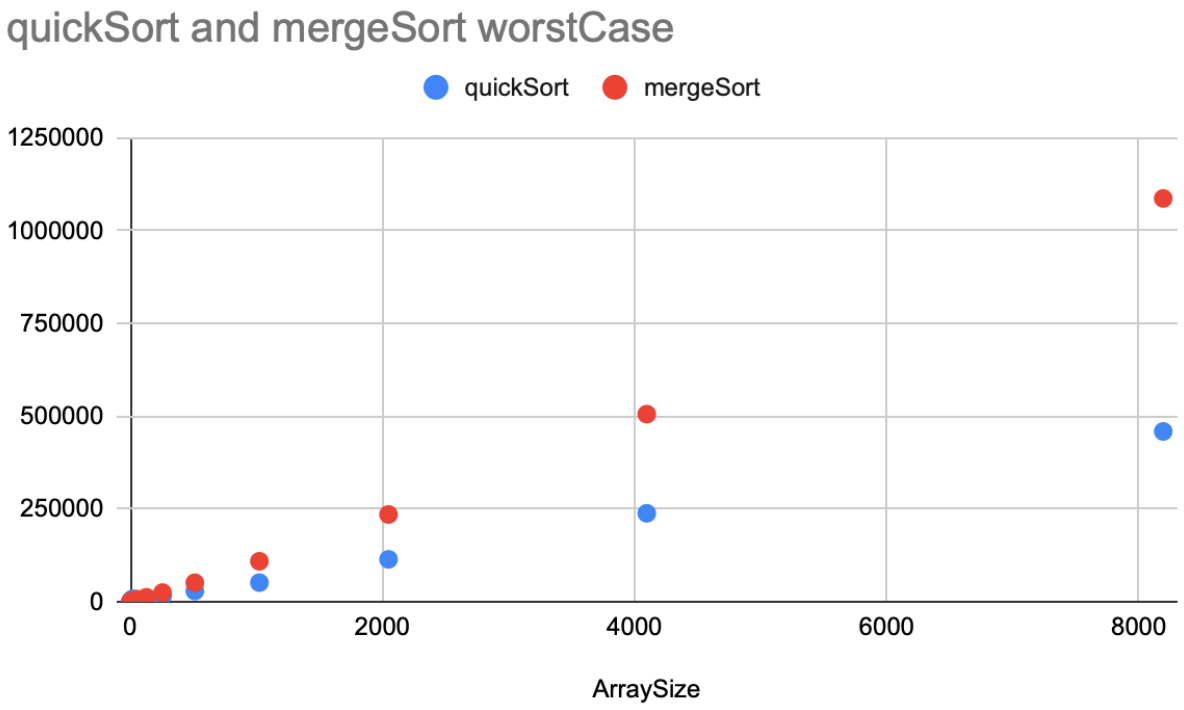
I used the random approximation for my pivot in all of the following quicksort vs mergesort experiments. I used a threshold of 20 in all of my mergesort functions, as that is where mergesort started to outperform insertion sort in my original analysis.

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. Mergesort is stable. The total runtime is for each loop of mergesort is O(N) because we have to make a copy of all of the elements. We have log(N) levels because we split the entire array until the pieces are smaller than the threshold, and then insertion sort is called in.

The growth rate for quicksort followed the trend that I expected, after we fixed our algorithm to correctly call the end index, instead of the list.size(). 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%.

Quicksort outperformed mergesort in all three analyses. In the best case scenario, mergesort and quicksort performed extremely similarly. In the average case scenario, quicksort outperformed mergesort more significantly, especially at larger array sizes. In the worst case scenario, mergesort grew much more quickly. The difference was the largest in the worst case scenario. I think this difference in runtimes can be attributed to the fact that mergesort copies elements into a different array, where our quicksort does not make another array, but alters the current array. I expected the difference in runtime to be the smallest in the best case, but I did not anticipate how poorly mergesort would perform in the worst case scenario. I wonder if this is because the merge time significantly increases.


quickSort and mergeSort averageCase
quickSort
mergeSort
2500000
2000000
1500000
1000000
500000
0
runtime (nanoseconds)
0
1000
2000
3000
4000
5000
ArraySize

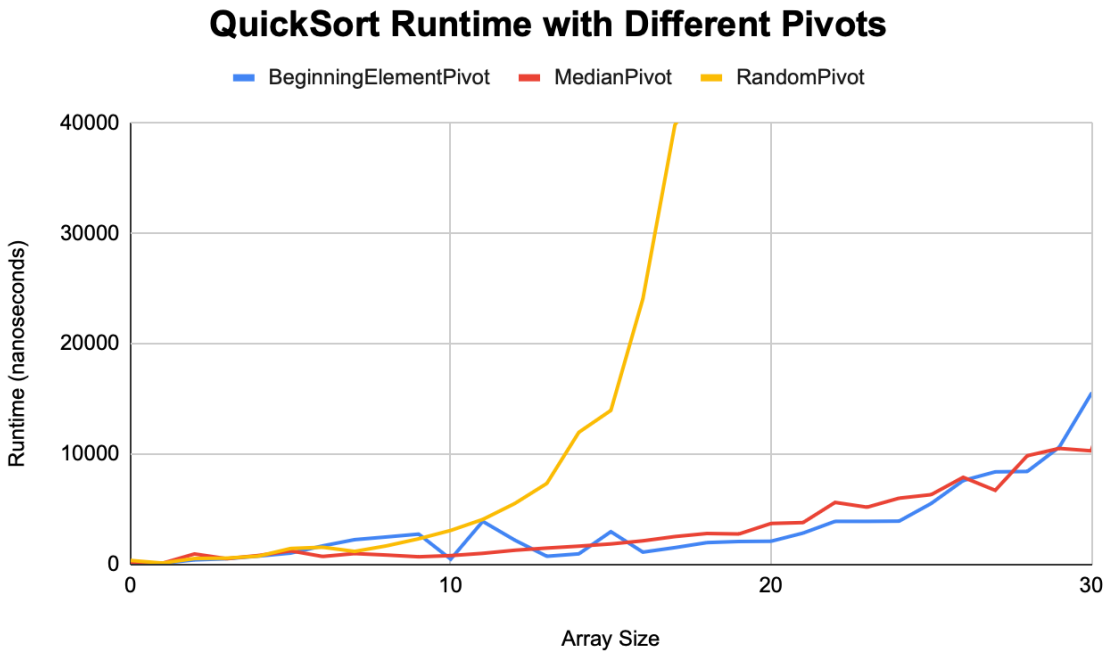

quickSort and mergeSort worstCase
quickSort
mergeSort
1250000
1000000
750000
500000
250000
0
0
2000
4000
6000
8000
ArraySize

Analysis before fixing our quicksort Algorithm:

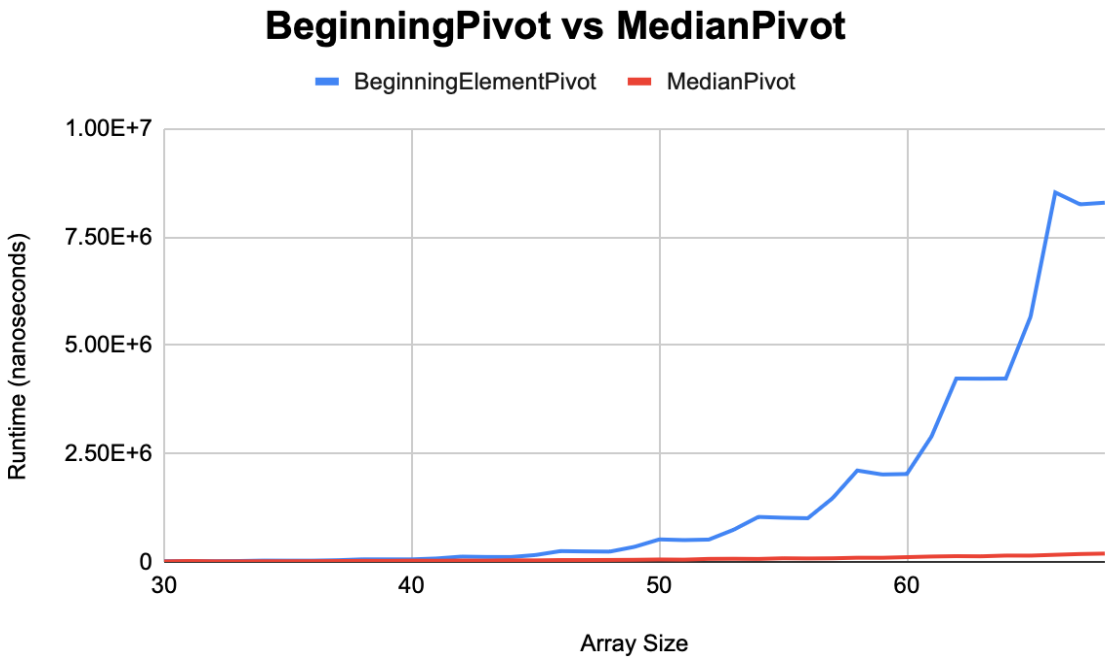
Quicksort Pivot Experiment

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)).



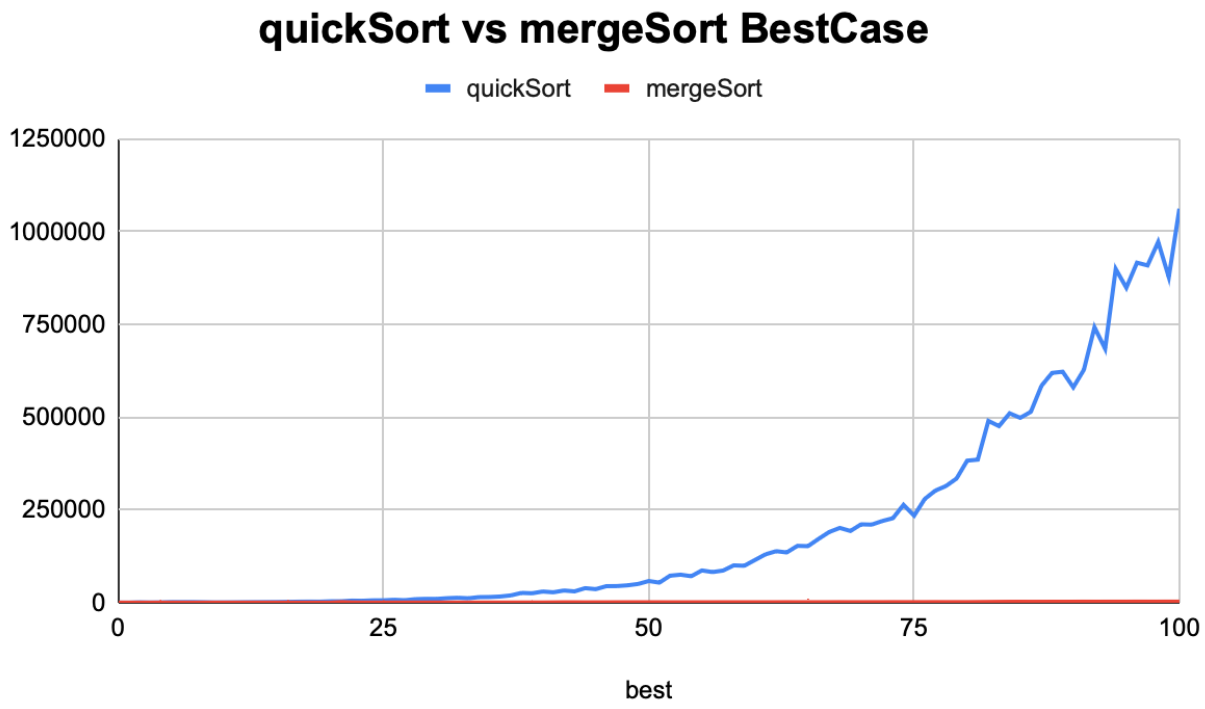
After we fixed our quicksort algorithm, I reran these tests to make sure that I was getting similar results. The error in our quicksort algorithm was in the recursive call to the “second” array that went from the pivot to the end element. I did not hypothesize that it would change, but wanted to verify that prior to rerunning my quicksort vs mergesort analysis.

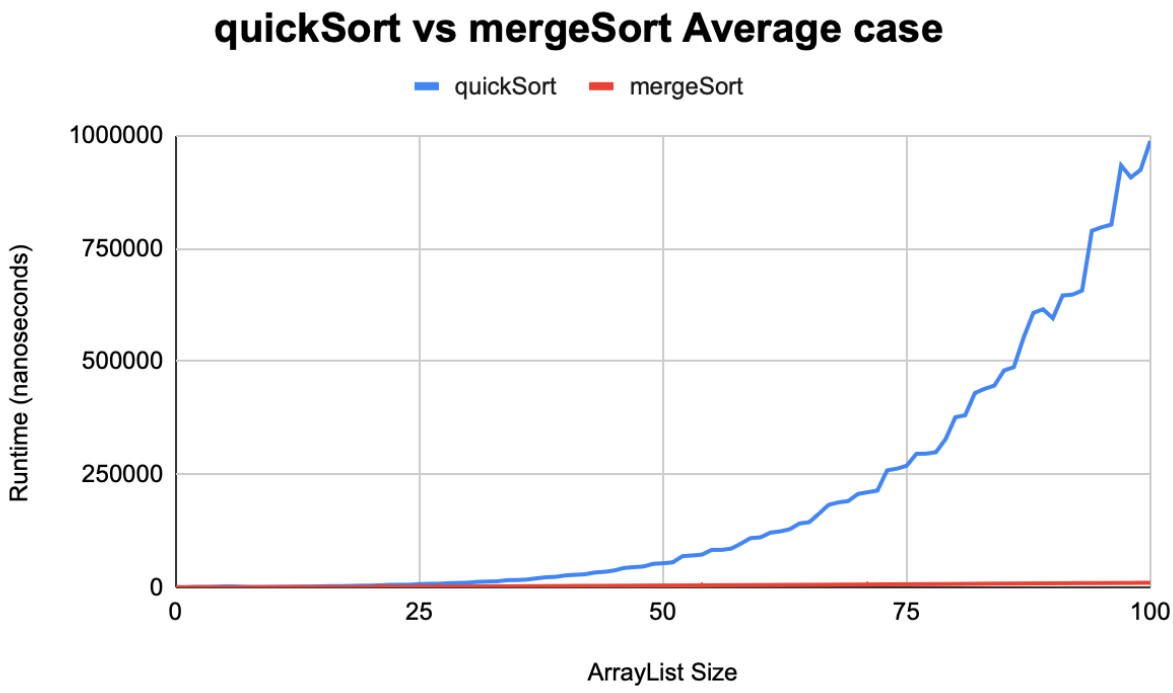
Mergesort vs. Quicksort Experiment

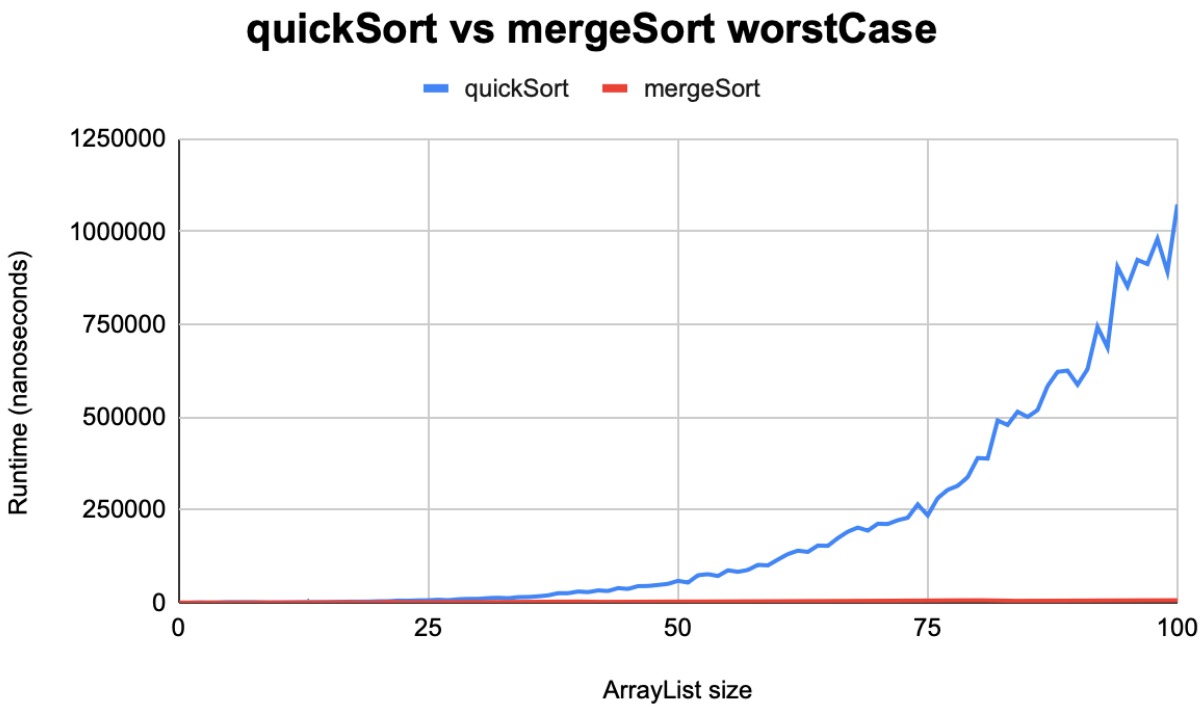
I used the median approximation for my pivot in all of the following quicksort vs mergesort experiments. I used a threshold of 20 in all of my mergesort functions, as that is where mergesort started to outperform insertion sort in my original analysis.

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.

For my quicksort runtime, I expected O(log(N)). Like mergesort, quicksort touches each element in an array at each level (N runtime). However, we do not copy the elements into a new array in our quicksort method. 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