**Figure A**

| Threshold | Time(Nanoseconds) |
|---|---|
| 0 | 2306101 |
| 8 | 1281663 |
| 64 | 1755109 |
| 1024 | 7042016 |
| 4096 | 18585992 |

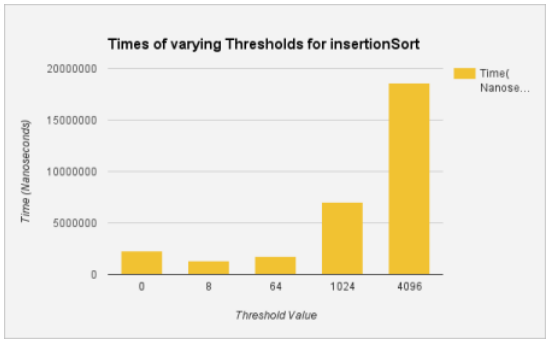
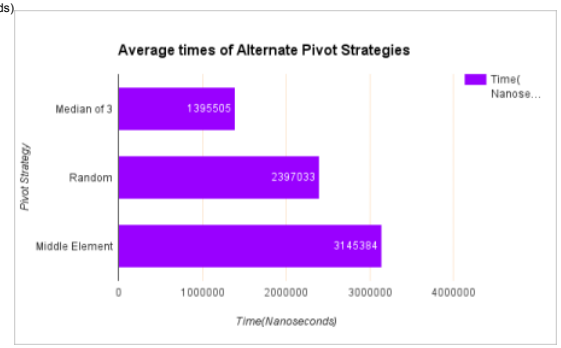



**Figure B**

| Pivot Strategy | Time(Nanoseconds) |
|---|---|
| Median of 3 | 1395505 |
| Random | 2397033 |
| Middle Element | 3145384 |



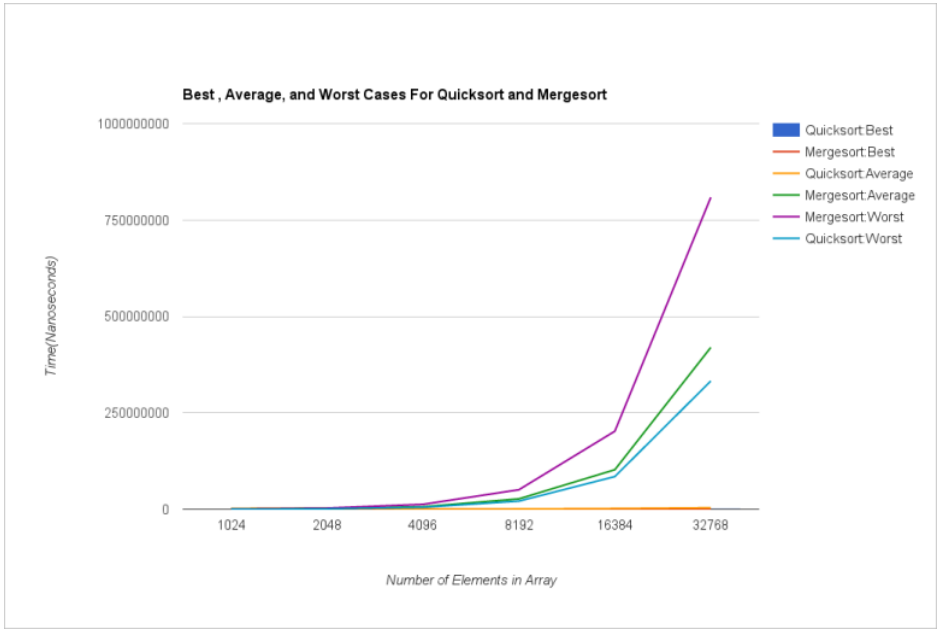

My Programming Partner was Joshua Shipley. He submitted the code. He was a good partner, we sometimes had scheduling issues , but he got a lot of tests made for the JUNIT class and I worked on debugging. Pair Programming is useful because you sometimes speed through assignments that normally take longer working alone. Some cons to having a partner is scheduling a time to meet up, and misunderstanding of concepts between the programmers.

To find a good threshold for mergeSort we sorted 1000 permuted arrays consisting of 10000 elements each. We found that running insertionSort past 8 - 16 elements in the mergeSort was starting to slow the program down. So we settled on setting insertionSort to activate at 8 elements. (NOTE: in the graph a threshold of 0 means running the experiment with mergeSort only.) See Figure A;

For finding the best pivot strategy. We tried three methods: ---Picking an element at random,
---Picking the first, last and the middle element and finding the median between the three and....
---Just Picking the middle element
After running it through the "Gauntlet"( same 1000 arrays with 10000 elements each) We discovered that picking the median between the 3 values was the best because, the chances of running into an extreme pivot point was lower. See Figure B

**Figure C**

| | Quicksort:Best | Mergesort:Best | Quicksort:Avera | Mergesort:Avera | Mergesort:Worst | Quicksort:Worst |
|---|---|---|---|---|---|---|
| 1024 | 1320746 | 6198 | 2389339 | 723370 | 821202 | 417190 |
| 2048 | 886108 | 9319 | 3559252 | 1833551 | 3200359 | 1310072 |
| 4096 | 2018252 | 17776 | 568268 | 6616622 | 12816546 | 5332618 |
| 8192 | 1082017 | 20116 | 894524 | 27058978 | 50700421 | 21200786 |
| 16384 | 1341930 | 135190 | 2102495 | 102627739 | 202507984 | 84891904 |
| 32768 | 2153032 | 158386 | 4065858 | 419689098 | 808823435 | 332665467 |




After testing the best case (an already sorted array) , the worst case(reverse ordered array) and the average case( shuffled array) between Quick and mergeSort ,we discovered that the mergeSort was quite a bit faster than the quickSort because it would sort elements as it traversed the array. The quicksort would take alot longer partitioning elements and partitioned every element even if it was already sorted. MergeSort using insertionSort also helped it get faster times.

When I was running the tests I was expecting around a O(Nlog(N)) for both mergeSort and QuickSort. What I got was O(Nlog(N)) for mergeSort and O(N^2) for QuickSort. I predicted MergeSort was a guaranteed O(Nlog(N)) due to the way it is implemented. Quicksort however was a lot different. I thought that the picking the median between 3 elements was a solid strategy for picking a good pivot, but the variance in the pivots was too great and caused a lot of the testing times to be longer, more frequently . MergeSort was much more consistent in its times and was faster because of it. (Figure C)