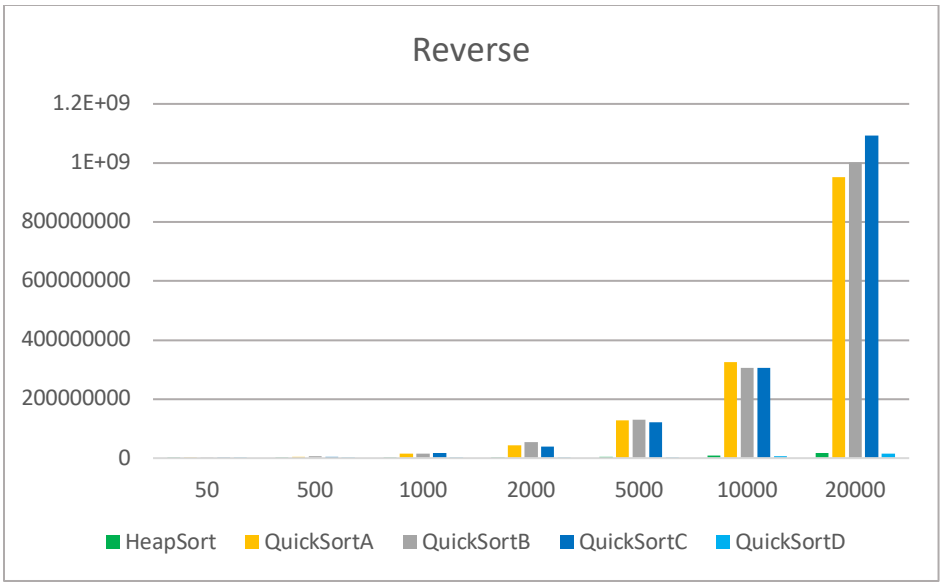
Sharise Dantzler
Lab 4 Analysis
Due Date: May 5, 2020

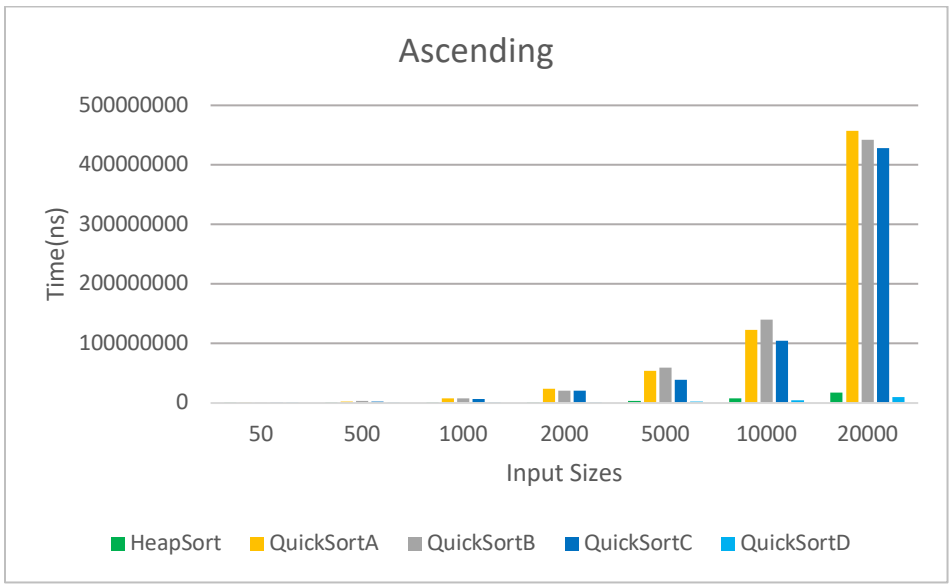
This lab consisted of five main components: user input and output through the file system, programming four variations of a Quicksort, programming a Heapsort, and gathering the time used to run each sort per input. Both sorts used were recursive programs given that there is less code to write and I preferred the recursive solutions as opposed to the iterative solutions. Also, all four variations of the Quicksort mentioned base cases or stopping cases, which naturally seemed recursive in my opinion. If the Heapsort were written iteratively it would build a max heap and maintain those properties when sorting without the recursive calls to the heapify method. The iterative solution of a Quicksort would have to implement the use of an auxiliary stack to maintain intermediate values of low and high.

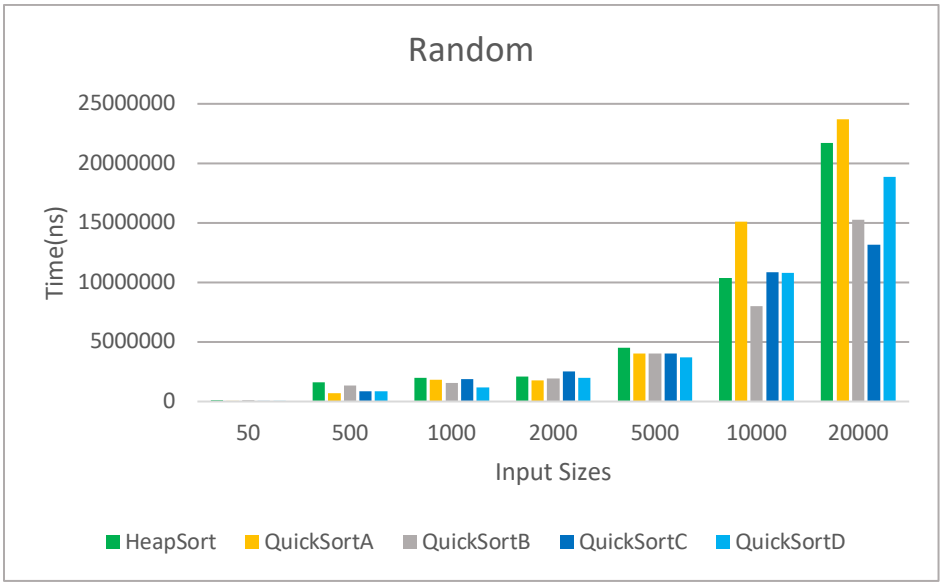
The input files were pre-generated in order to limit the duplicates to less than 1% and to ensure the correct number of data variables were included. The four variations of the Quicksort included four different ways to partition and select the pivot and alternatives to finish the sort. Quicksort A selects the first item of the partition as the pivot and uses a partition of size one and two as a stopping case. This stopping case uses a simple insertion sort to finish. Quicksort B selects the first item of the partition as the pivot and uses a partition of size 100 as a stopping case. Quicksort C selects the first item of the partition as the pivot and uses a partition of size 50 as a stopping case. Finally, Quicksort D selects a median of three as the pivot and uses a partition of size one and two as a stopping case. Heapsort uses the basic structure where a heap is built, and items are extracted in sorted order. Each program tracks how long each sort is in nanoseconds for the following input sizes: 50, 500, 1000, 2000, 5000, 10000, and 20000.

The average time complexity for both sorts, including variations, is $O(n\log_2 n)$. However, each Quicksort presents a different stopping case. Thus, before discussing please note that each Quicksort uses the optimized version of tail recursion to ensure maximum space complexity is $O(\log_2 n)$. Also, for simplicity and accuracy, each Quicksort uses a simple insertion sort for different stopping cases, which allows for less comparisons and swaps and a best-case time complexity of $O(n)$ on small data sets. After reviewing the data per each individual input size run, I noticed Quicksort A, B, and C continued to increase in run times especially for the ascending and reverse data input sets. This same conclusion is present in the ascending and reverse graphs for all the sorts compared for all input sizes. Please refer to 3 graphs and table below. In contrast, Quicksort D and Heapsort performed with a time complexity of $O(\log_2 n)$ across the board. However, the random graph tells a different story for all of the sorts! In this graph Quicksort B and C performed better than usual while Quicksort A was the worst following Heapsort and Quicksort D. In conclusion, for random data all the sorts are building a trend of $O(n\log_2 n)$, which is accurate with the average time complexity. However, for the reverse and ascending order Quicksort A and Heapsort are showing a trend of on average $O(\log_2 n)$ time complexity, which is awesome, while the other three sorts show an average time complexity going towards $O(n^2)$.

The inefficiencies of Quicksort A, B, and C may largely be related to the of pivot selection. Given that the first item is selected to partition the subarrays, this presents a huge inefficiency because there is a constant situation where there is 1 value on the right of the partition and n-2 values on the other side. Also, the stopping cases can slow down the sort depending on the value of k. Insertion sort takes over in all the stopping cases and is particularly sensitive to reverse data and has a worst-case time complexity of $O(n^2)$. This evidence is shown in the Reverse graph shown below. I think the factor that has the most effect on the data is the order of the data! According to the graphs below a different story is told per input per ordered representation.


Reverse
1.2E+09
1E+09
800000000
600000000
400000000
200000000
0
50
500
1000
2000
5000
10000
20000
HeapSort
QuickSortA
QuickSortB
QuickSortC
QuickSortD


Ascending
500000000
400000000
300000000
200000000
100000000
0
Time(ns)
50
500
1000
2000
5000
10000
20000
Input Sizes
HeapSort
QuickSortA
QuickSortB
QuickSortC
QuickSortD


Random
25000000
20000000
15000000
10000000
5000000
0
Time(ns)
50
500
1000
2000
5000
10000
20000
Input Sizes
HeapSort
QuickSortA
QuickSortB
QuickSortC
QuickSortD

	HeapSort	QuickSortA	QuickSortB	QuickSortC	QuickSortD
Reverse					
50	106862	178939	190404	171324	77601
500	1240840	5012533	6084610	5277196	620474
1000	1775176	15196682	15193688	16717827	1377094
2000	2298488	43520135	54228417	39043216	1947524
5000	4022768	127664828	130205989	122488537	2946734
10000	8684775	326345174	305388680	305589428	5768113
20000	18013104	951292585	1002401714	1092017757	14369397
Ascending					
50	122950	104699	104307	103740	48418
500	1527208	2753230	3532521	2698772	677224
1000	1673892	7928367	7479424	7059501	1236414
2000	1854466	23603320	20854507	20484819	1137687
5000	3484319	54514446	59643418	39385721	2642115
10000	8383153	122970817	140565920	104463616	4855145
20000	17035140	457017376	442345211	428213550	10449908
Random					
50	125284	67637	84989	52377	79780
500	1623591	672051	1349184	837855	853849
1000	1966489	1836650	1563213	1889514	1207065
2000	2110692	1758253	1957427	2544760	1963291
5000	4492583	4005357	4052697	4013176	3697696
10000	10380909	15098039	8011041	10858466	10789194
20000	21687666	23689372	15249332	13184667	18851184

What I Learned

I learned that the organization of data can have a large impact on the run time and complexity. Personally, from analyzing the 125 runs, I would choose Quicksort D almost every time. I think this sort has an excellent time complexity on average in every situation and a space complexity of $O(\log_2 n)$. The space complexity was the determining factor on whether or not I would choose Quicksort D over Heapsort whose space complexity is linear.

What I Might Do Differently

If I were to revisit this problem in the future, I would consider comparing Quicksort and Heapsort against other sorts such as Straight and Natural Merge. Also, the use of files with 15-20% duplicates.

Justification for Design Decisions

My source code consisted of 5 classes. One for the Heapsort, and four others for the variated Quicksort. Each main method reads in an input file of data and returns an output file of sorted data. The data is read into an ArrayList to accommodate various sizes of data and then converts the data into a regular array to be passed into the methods.

Issues of Efficiency

Discussed in analysis above.