## Choice of Implementation

For this project, I chose to implement the Heap sort and the Quick sort recursively. I chose this method because it made more intuitive sense to me. The quick sort was splitting into two parts and performing the same sorting techniques on each part. This continued until it reached a stopping point of the minimum partition size. The combination of performing the same steps on a smaller and smaller partition until a stopping point just screamed recursion to me.

Heap sort was the same way. Every time a value was removed, you would have to recursively go through all the affected subtrees and make sure that it was in a max heap. It seemed easier and more intuitive to implement in recursion than in iteration.

## Runtime Performance

It’s interesting to see how big of a difference in runtimes there were. Let’s first talk about Heapsort.

For Heapsort, it appears that it does not perform as well on smaller samples of ordered data compared to the rest, but it does perform better on larger numbers of ordered data. This is interesting, as I would not have suspected that. I am a little puzzled as to why it performs better on larger number of ordered data than the same number of reverse or random data. Perhaps it is peculiar to these files, although it seems to have similar performance on both the Rand and Reverse files, so that is probably not the case. A Google search does not reveal the answer either. My best guess is that heap sort does gain some efficiencies with the order of the file, even though it is mostly order insensitive.

For Quicksort, it seems that the order of the file does have a difference. It appears to perform the best when the data is random. For smaller files, reverse order seems to have the best performance, but this is due to the efficiency of insertion sort, not quicksort. For larger files, random seems to have the best performance. It really does not like ordered files however. It should be noted that performance for the partition sizes of 50 and 100 means that the smaller files are measuring mostly insertion sort performance. The larger files for ordered and unordered first pivot are understandably bad, since every partition is going to be a partition of 1.

In terms of partition size for Quicksort, it doesn’t really seem to make too much of a difference. It seems that the 3 partitions switch back and forth on who is the fastest and the slowest. It seems the size 50 partition might have a slight edge in larger files, while the size 100 has an edge on smaller files.

The biggest difference in runtimes, however, seems to be the pivot selection. Selecting the middle of the three has huge benefits in terms of runtime. It seems that selecting a pivot that is in the center, especially in the case of ordered and reverse ordered files, provides a large performance boost. This is probably because there is much less likelihood of seriously uneven partitions.

Between quicksort and heapsort, it seems that the heapsort is relatively agnostic about file order compared to quicksort. It also has better performance than the worst cases of quicksort, the ordered files. It seems to have worse performance than the best cases of quicksort, where the pivot selected is close to the middle.

## Costs

The complexity of the heap sort program is O(nlogn). The function heapify runs in O(logn) time. This will be run n times through sortHeap. If the data was really bad, it could reach closer to O(n^2), as heapify will need to be rerun for every node affected by it, and if somehow every node was affected it could need more runs. This is highly unlikely, however. The space complexity is O(n), as only the original array is ever used, and this is a linear function of file size. The memory complexity is probably the same as the runtime cost, O(nlogn), as it is a recursive program and all of the steps will need to be stored in memory.

The complexity of the quick sort program is dependent on the order of the data. In the worst case scenario, quickSort will run n times, and makePartition will run n times. This is in the case when every single partition is the greatest or the least, as in ordered or reverse ordered files. The best is when we pick the middle. This means that the quickSort will run n times as before, but makePartition will only run logn times, as it will become almost a perfect binary tree. The cost of that is O(logn). There is also a factor of partition size, where insertion sort cost becomes an issue. This is negligible at larger file sizes; however, at smaller file sizes, this could push it closer towards O(n) or O(n^2) depending on the order of the file (reverse is best).

## Lessons learned and improvements

I learned that the order of the file is important to the selection of a sort method. The times could vary vastly depending on the method of sort. I think I would want to test multiple files of the same size in the future over multiple runs and find a mean and standard deviation of those runs. That would be more accurate than just having one run. I would also want to try the method of selecting the mean instead of an actual number in the array.

# Appendix

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Run | Type | Heap | QuickSort par.2 First | QuickSort par.50 First | QuickSort par.100 First | QuickSort par.2 Median |
| 50 | Ordered | 39345 | 60040 | 2811 | 2299 | 19418 |
| 500 | Ordered | 103473 | 1134372 | 415681 | 378891 | 36791 |
| 1000 | Ordered | 126467 | 1629255 | 1457055 | 1925367 | 88910 |
| 2000 | Ordered | 161214 | 430755 | 715370 | 630037 | 32448 |
| 5000 | Ordered | 407250 | 4463653 | 2297871 | 2343603 | 73326 |
| 20000 | Ordered | 1795578 | 53329549 | 44172303 | 39099564 | 1844888 |
| 50 | Rand | 6643 | 32703 | 511 | 256 | 3066 |
| 500 | Rand | 57485 | 350531 | 335712 | 246803 | 11242 |
| 1000 | Rand | 93254 | 112415 | 109094 | 108072 | 19162 |
| 2000 | Rand | 285637 | 725334 | 563098 | 534739 | 35002 |
| 5000 | Rand | 554923 | 2417184 | 2284074 | 2518869 | 115481 |
| 20000 | Rand | 2482589 | 42623016 | 32711566 | 45886125 | 374037 |
| 50 | Reverse | 8687 | 2300 | 255 | 255 | 1022 |
| 500 | Reverse | 26571 | 38579 | 30403 | 28615 | 6132 |
| 1000 | Reverse | 53397 | 105773 | 105773 | 106539 | 10986 |
| 2000 | Reverse | 108328 | 386045 | 399840 | 390899 | 21716 |
| 5000 | Reverse | 539849 | 4073776 | 3139707 | 4233713 | 53653 |
| 20000 | Reverse | 2402365 | 47973216 | 42934713 | 43621724 | 374292 |