Anosh Abraham
3/3/23

Programming Assignment 1 Discussion

Output comparison and time results:

```
*******PART ONE*******

***MERGE SORT RESULTS***
Reversed Array -> n=32
        Original Array = 32
        Sorted Array = 1 2 3
        COMPARISONS = 80
Pre-sorted Array -> n=32
        Original Array = 1 2
        Sorted Array = 1 2 3
        COMPARISONS = 80
Random Array -> n=32
        Original Array = 32
        Sorted Array = 4 5 6
        COMPARISONS = 122

***HEAP SORT RESULTS***
Reversed Array -> n=32
        Original Array = 32
        Sorted Array = 1 2 3
        COMPARISONS = 202
Pre-sorted Array -> n=32
        Original Array = 1 2
        Sorted Array = 1 2 3
        COMPARISONS = 231
Random Array -> n=32
        Original Array = 32
        Sorted Array = 4 5 6
        COMPARISONS = 221

***QUICK SORT RESULTS***
Reversed Array -> n=32
        Original Array = 32
        Sorted Array = 1 2 3
        COMPARISONS = 496
Pre-sorted Array -> n=32
        Original Array = 1 2
        Sorted Array = 1 2 3
        COMPARISONS = 357
Random Array -> n=32
        Original Array = 32
        Sorted Array = 4 5 6
        COMPARISONS = 226
```

```
*******PART TWO*******

Random Array -> n=1024
***MERGE SORT RESULTS***
        COMPARISONS = 8953
        TIME TO SORT = 1.4654ms
***HEAP SORT RESULTS***
        COMPARISONS = 17305
        TIME TO SORT = 0.4996ms
***QUICK SORT RESULTS***
        COMPARISONS = 20593
        TIME TO SORT = 1.2862ms

Random Array -> n=32768
***MERGE SORT RESULTS***
        COMPARISONS = 449993
        TIME TO SORT = 16.1349ms
***HEAP SORT RESULTS***
        COMPARISONS = 882291
        TIME TO SORT = 10.7455ms
***QUICK SORT RESULTS***
        COMPARISONS = 1492183
        TIME TO SORT = 19.2678ms

Random Array -> n=1048576
***MERGE SORT RESULTS***
        COMPARISONS = 19645338
        TIME TO SORT = 213.5842ms
***HEAP SORT RESULTS***
        COMPARISONS = 38706431
        TIME TO SORT = 256.0685ms
***QUICK SORT RESULTS***
        COMPARISONS = 59434654
        TIME TO SORT = 244.0539ms
```

Tabulate the performance results, listing both the number of comparisons and the measured clock times. Use the tabulated number of comparisons to determine the time complexity. For example, does the data for quicksort verify O(n2) or O(nlogn) time, and what is the constant factor? Note that since you use a random sequence to sort, the running times should correspond to the average-case time complexities.

Observing the results in the output for part two we can clearly see that as n increases, the time to sort also increases. This trend does follow the average case of O(nlogn) for all three sort algorithms that were tested which were merge, heap, and quick sort. This is because of how random the arrays being generated are. With the constant factor however, it's difficult to say what it would be due to many contributing factors like the hardware/software being used and how efficient they are as well as how each sort is implemented. But we can say some general information about constant factors for each sort. With merge sort, it would have a relatively high constant factor due to merging two sorted subarrays that requires more memory and computation in order to compare/merge elements. Heap sort on the other hand has a relatively low constant factor due to its operations having to directly affect the input array without using more memory. And for quick sort, this has a more fluctuating constant factor all due to three things 1) the pivot element being used to sort the rest of the elements, 2) the partitioning scheme and 3) the input data. Specifically with the quicksort implementation that I used in this programming assignment, it used a three way partitioning scheme to group elements into three categories. That being elements less than pivot, equal to pivot, and more than pivot. Because of this implementation, it has less comparisons than a non three way partitioning scheme. So trying to determine a constant factor in this case for quicksort can be challenging due to the implementation of the sorting algorithm.