Assignment 4 on exploring various Sorting Algorithms Report

Prepared for
Dr. Sachin Meena
Senior Lecturer of Computer Science PH. D
University of Georgia

By
Danyal Khan
Computer Science student at the University of Georgia

Juan Hernandez
Computer Science student at the University of Georgia

November 17, 2023

Abstract

This report explores the number of comparisons made from various common sorting algorithms such as Selection Sort, Merge Sort, Quick Sort, and Heap Sort. Based on the number of comparisons, the identification and analysis on complexity for each of the sorting algorithms are concluded. Then various randomized sets of data, with different sizes, are explored for each sorting algorithm and a theoretical graph is made based on the average number of comparisons for a specific n size of data inputs is drawn.

Experiment 1: Finding number of comparisons based on the fixed input files for each of the above-mentioned sorting algorithms.

Algorithm	**Input Type**	**# Comparisons**	**Comments about time complexity and # comparisons**
Selection Sort	Ordered.txt	49995000	$\frac{1}{2}(n^2) = 0(n^2)$
Selection Sort	Random.txt	49995000	$\frac{1}{2}(n^2) = O(n^2)$
Selection Sort	Reverse.txt	49995000	$\frac{1}{2}(n^2) = O(n^2)$
Merge Sort	Ordered.txt	64608	O(nlogn)
Merge Sort	Random.txt	120294	O(nlogn)
Merge Sort	Reverse.txt	69008	O(nlogn)
Quick Sort (First Pivot)	Ordered.txt	50004999	$O(n^2)$ *This is our worst-case scenario since our input file is already sorted in ascending order*
Quick Sort (First Pivot)	Random.txt	162825	O(nlogn) * This is the best-case scenario since the input file is not pre-sorted*
Quick Sort (First Pivot)	Reverse.txt	50004999	$O(n^2)$ * This is the worst-case scenario since the input file is already pre-sorted in descending order*

Quick Sort (Random Pivot)	Ordered.txt	157214	O(nlogn)
Quick Sort (Random Pivot)	Random.txt	156672	O(nlogn)
Quick Sort (Random Pivot)	Reverse.txt	148884	O(nlogn)
Heap Sort	Ordered.txt	258913	O(nlogn)
Heap Sort	Random.txt	243469	O(nlogn)
Heap Sort	Reverse.txt	228393	O(nlogn)

*For a data inputted that is not sorted, any approach to quick sort (first pivot or random pivot) is fine to use. However, when the data is already sorted and entered as an input for the quick sort algorithm, the quick sort-fp is less efficient to that of the quick sort-rp. Quick Sort-fp is $O(n^2)$ complexity to Quick Sort-rp that is a O(nlogn) for sorted input. This is because picking a pivot as the first element of every sub array that is already sorted, will require unnecessary number of comparisons which will never move values to the left or right of the pivot value. Going through the data set would be meaningless. But for a randomized pivot under an already sorted data will at most go through half the data to check which values to move behind or forward of the pivot value in the subarrays. *

* Notice how merge sort number of comparisons is greatly increased when an unsorted input data is called for this function as opposed to a pre-sorted data as an input. This can be due to the merge sort algorithm recursively splits the data into equal subarrays until a length of 1. Because the data is already sorted, there will not be a need to reorder and swap the data in these min arrays before merge them. Hence the number of comparisons will be reduced if the data is already sorted. But is the data is not sorted, the extra number of comparisons to swap proper elements will be invoked leading to a greater number of comparisons. *

*Notice that the heap sort algorithm is fairly consistent number of comparisons across all 3 input file types. This is because regardless if

the data is already pre-sorted or not, the array will be turned into a heap and then back into an array. At this point we have to check the first element in the array with the last element of the array, which will be the max value, and swap it if unordered and not swap it if it is ordered. Regardless if the numbers are sorted or not, the comparison will be made leading to a similar overall number of comparisons for any input file. *

*Notice that both cases of quick sort require a greater number of comparisons than that of merge sort. This is because merge sort equally partitions a list into subsets of the same equal size and keeps on doing this recursively until length of 1. It swaps and reorders after the dividing of elements is done. The same general format applies to any data set. However, quick sort sorts the data based on the pivot element first before dividing into smaller sub arrays. If the data requires to be swapped too much, a bad pick for a pivot, the number of comparisons for quick sort will increase much more than merge sort. When quick sort divides an array, both subarrays may not be an equal size based upon a pivot that you pick. If there are much more smaller values than the pivot than bigger values than the pivot, the two subarrays length will be unbalanced. As a result, the subarray with the bigger size will require more comparisons than the other subarray. *

b. Did you use extra memory space or other data structures other than the input array? If so, explain where and why?

- Selection Sort: No other data structures and no extra memory space were used as we were swapping with new minimum values found within the same **in-place array.**

- Merge Sort: The overall algorithm after dividing into multiple sub arrays until length 1 array was recursive and hence used more stack memory than a normal iterative algorithm would have. Each stack frame on the call would have it's own local variables and repat the same algorithm multiple times. This is overall inefficient as they involve computation of similar terms again and again.

- Quick Sort (First Pivot): The overall algorithm uses a local pivot variable which will be assigned to the first array element. The pivot will check and be used to push all values less than it to the left of the pivot and all values greater than the pivot to the right of it. Each recursion call will increase the number of int array copies making it memory intensive.

- Quick Sort (Random Pivot): The overall algorithm uses a local pivot variable which will be assigned to a random array element (which is determined by the random class). The pivot will check and be used to push all values less than it to the left of the pivot and all values greater than the pivot to the right of it. Each recursion call will increase the number of int array copies similar to the first pivot quick sort making it memory intensive.

- Heap Sort: The overall algorithm converts an array of data into a heap. We are sorting in the same array whole time and no subarrays are created. This in-place sorting algorithm only requires data space when swapping and does not require any other additional data structure.

c. Explain what sorting algorithms work best in what situations based on your experimental results.

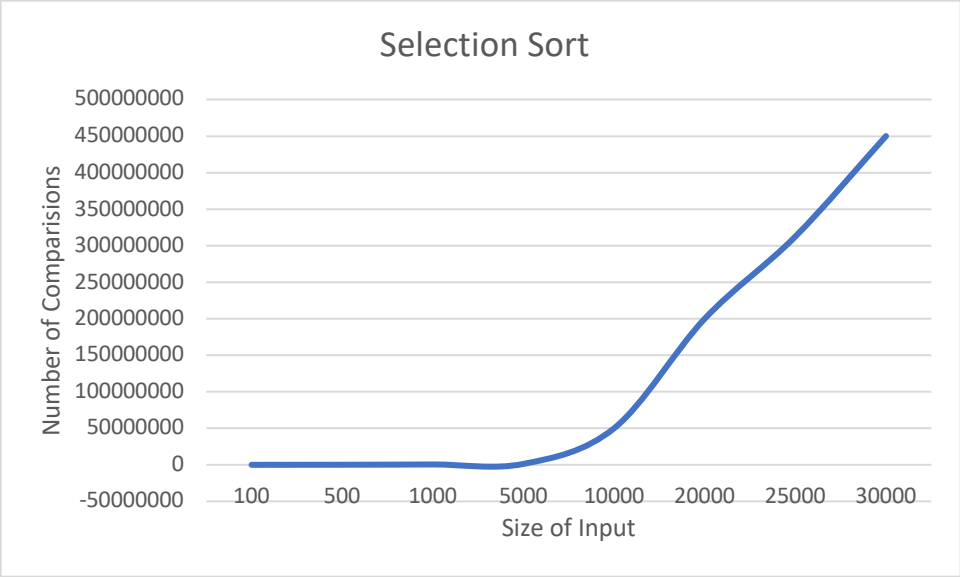
For both ordered and reverse data sets of any size, quick sort is not recommended as $O(n^2)$ number of comparisons will be made. Heap and merge sort are both O(nlogn) complexity for both of these files. However, merge sort significantly uses a smaller number of comparisons, about 70,000 comparisons compared to about 300,000 for heap sort. This can be attributed to the fact that merge sort is a dive and conquer algorithm that splits a big problem into smaller sub-problems that are easier to sort than in-place swapping. For the random input data set, merge sort would be the most preferred sorting algorithm of having a O(nlogn) complexity and efficient number of comparisons being made. It is a preference as to whether we

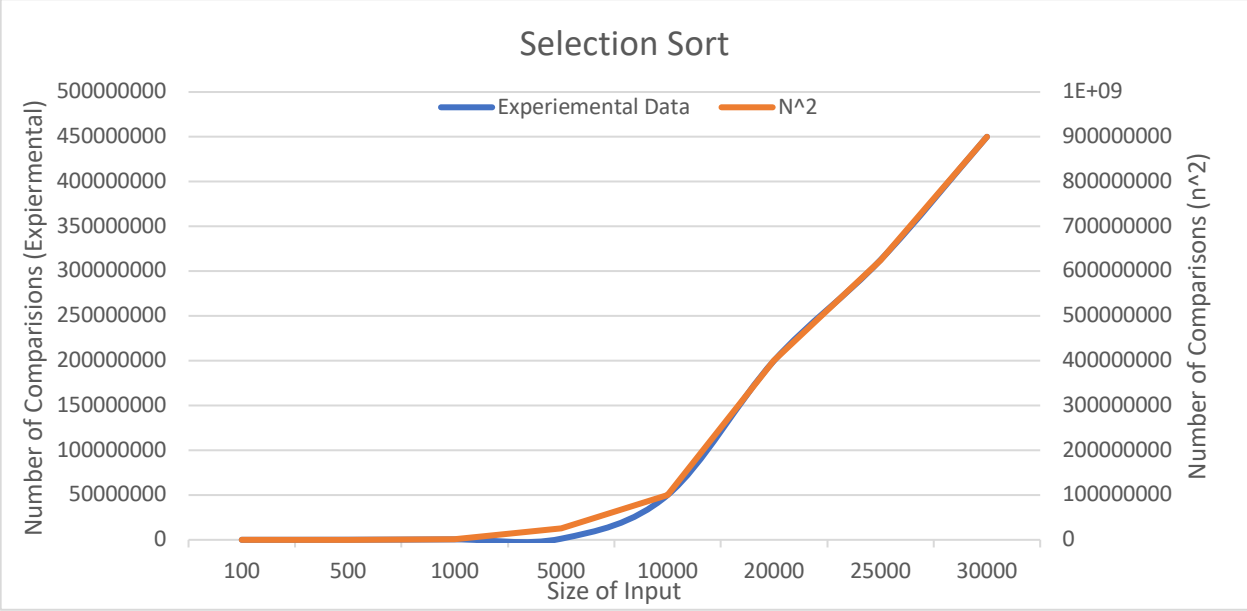
value time or resources of which sorting algorithm with the same complexity to choose.

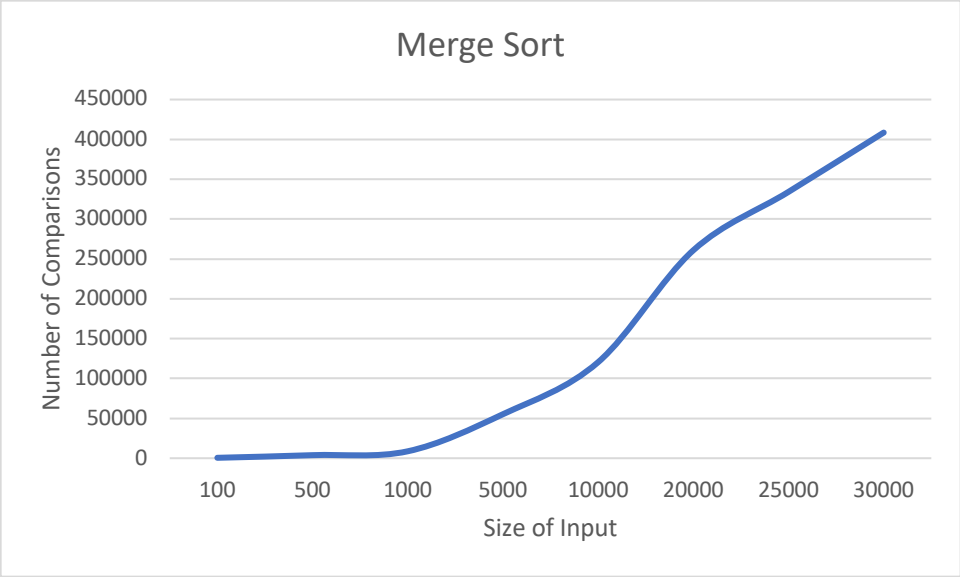
Experiment 2: Graphs of all 5 sorting algorithms with respect to the number of comparisons and input size.

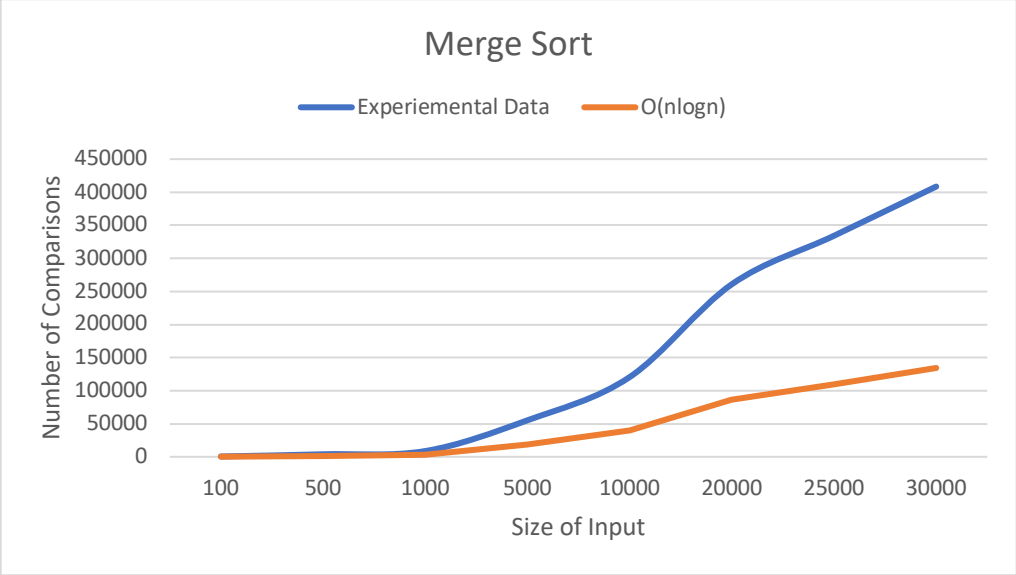
Sorting Alogrithm	100	500	1000	5000	10000	20000	25000	30000
Selection Sort	4950	124750	499500	1249750	49995000	199990000	312487500	449985000
Merge Sort	533	3853	8710	55214	120487	260873	334172	408556
Heap Sort	1126	7850	17707	111671	243471	526834	675085	824859
QuickSort-fp	750	5255	11223	75317	165493	398152	497863	621397
QuickSort-rp	624	4628	11101	71238	154494	339217	446496	543221

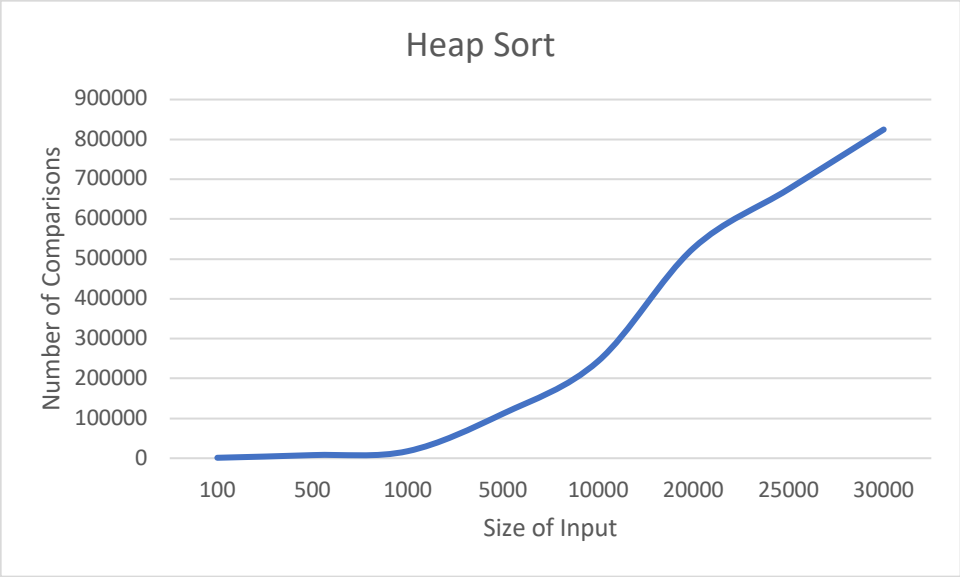
Note separate excel program with calculations for average number of comparisons for each of the sorting algorithms, will be submitted along with the report.

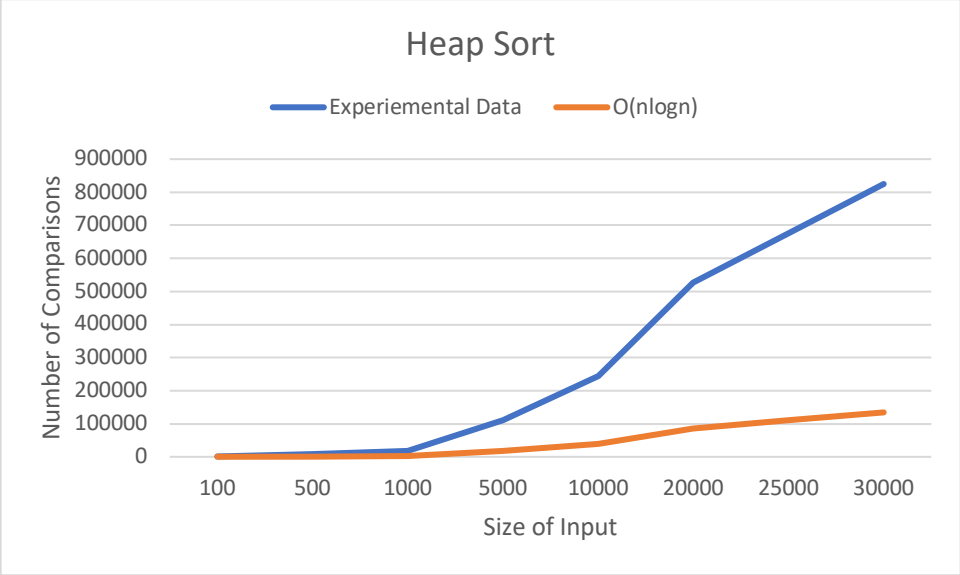

Selection Sort
500000000
450000000
400000000
350000000
300000000
250000000
200000000
150000000
100000000
50000000
0
-50000000
Number of Comparisons
100
500
1000
5000
10000
20000
25000
30000
Size of Input

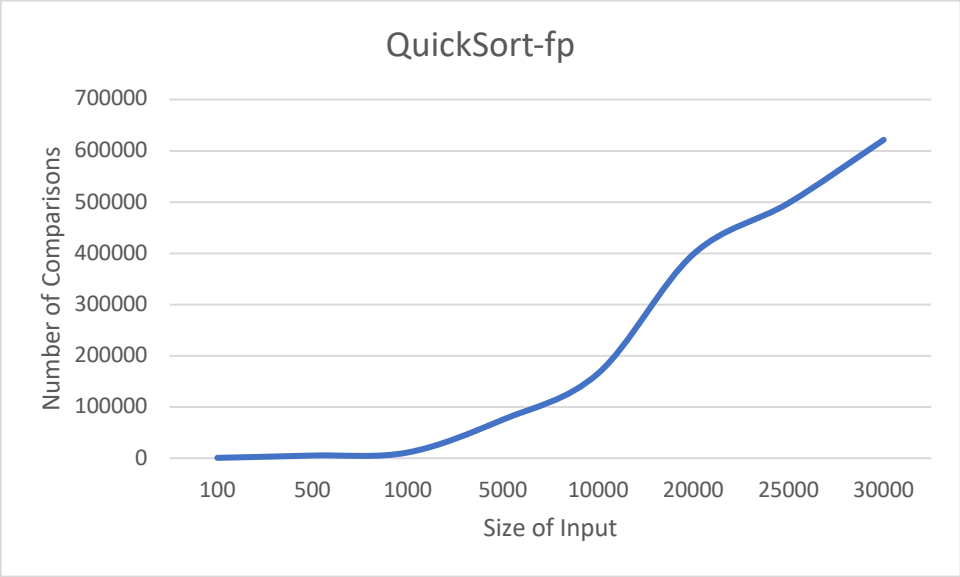

Selection Sort
Experiemental Data
N^2
500000000
450000000
400000000
350000000
300000000
250000000
200000000
150000000
100000000
50000000
0
Number of Comparisions (Expiermental)
1E+09
900000000
800000000
700000000
600000000
500000000
400000000
300000000
200000000
100000000
0
Number of Comparisons (n^2)
100
500
1000
5000
10000
20000
25000
30000
Size of Input

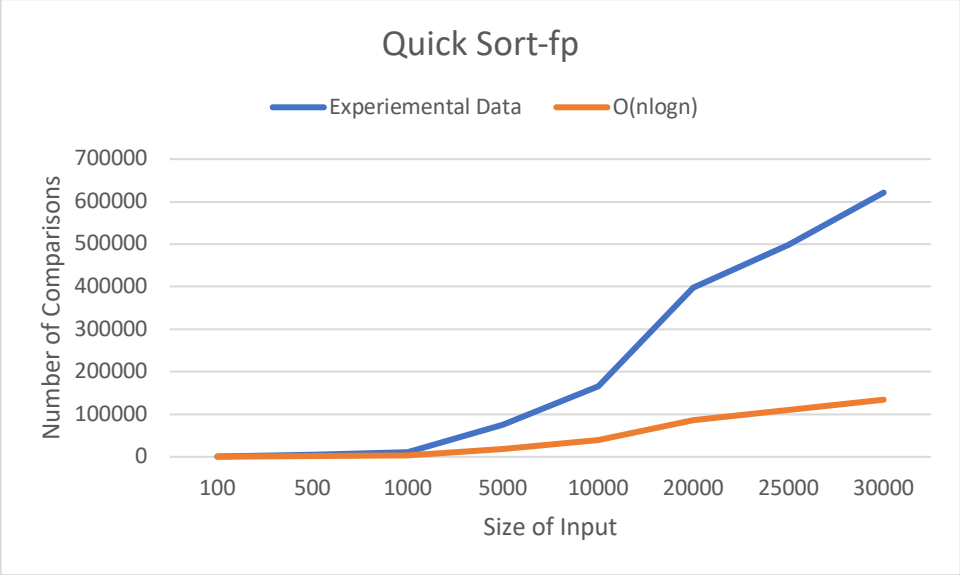

Merge Sort
Number of Comparisons
450000
400000
350000
300000
250000
200000
150000
100000
50000
0
100
500
1000
5000
10000
20000
25000
30000
Size of Input

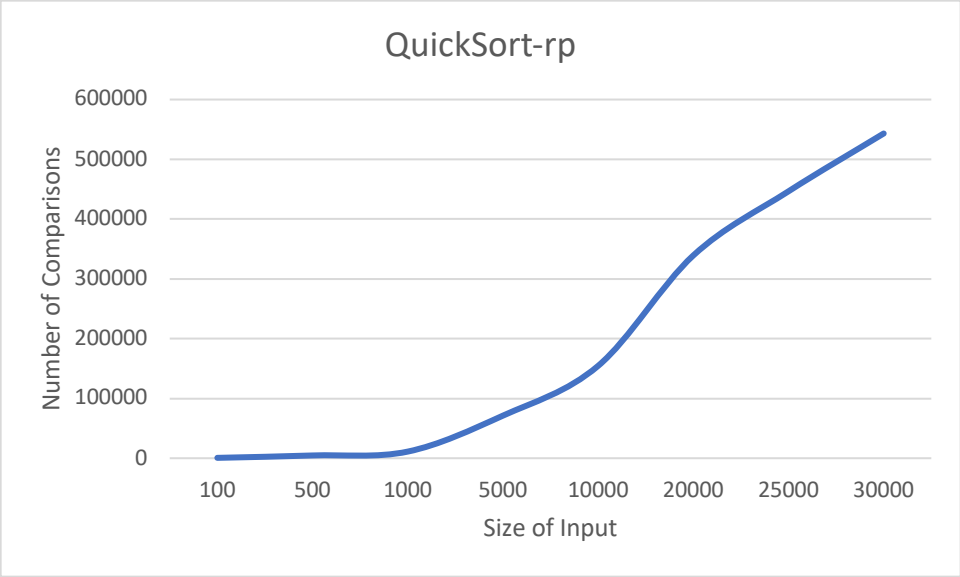

Merge Sort
Experiemental Data
O(nlogn)
Number of Comparisons
450000
400000
350000
300000
250000
200000
150000
100000
50000
0
100
500
1000
5000
10000
20000
25000
30000
Size of Input

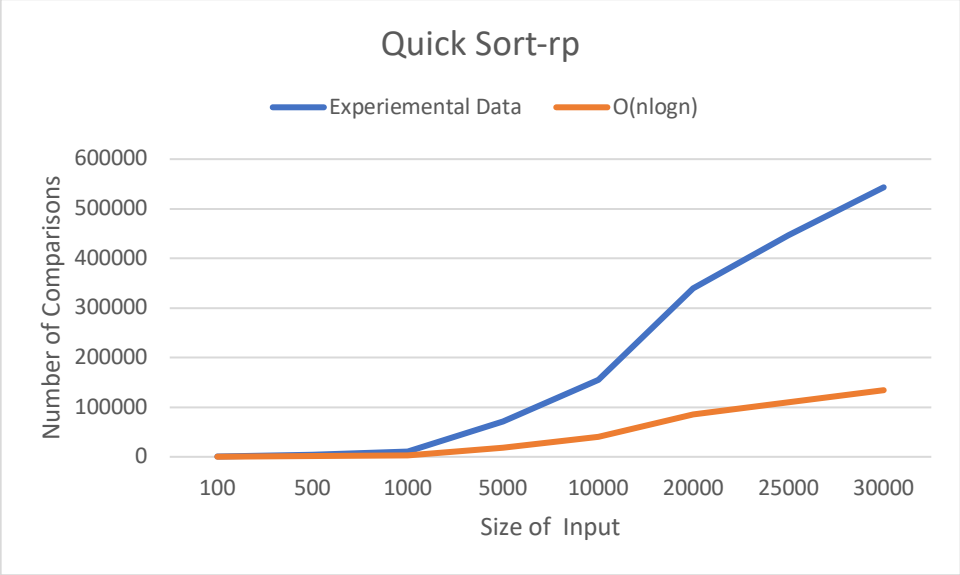

Heap Sort
900000
800000
700000
600000
500000
400000
300000
200000
100000
0
Number of Comparisons
100
500
1000
5000
10000
20000
25000
30000
Size of Input

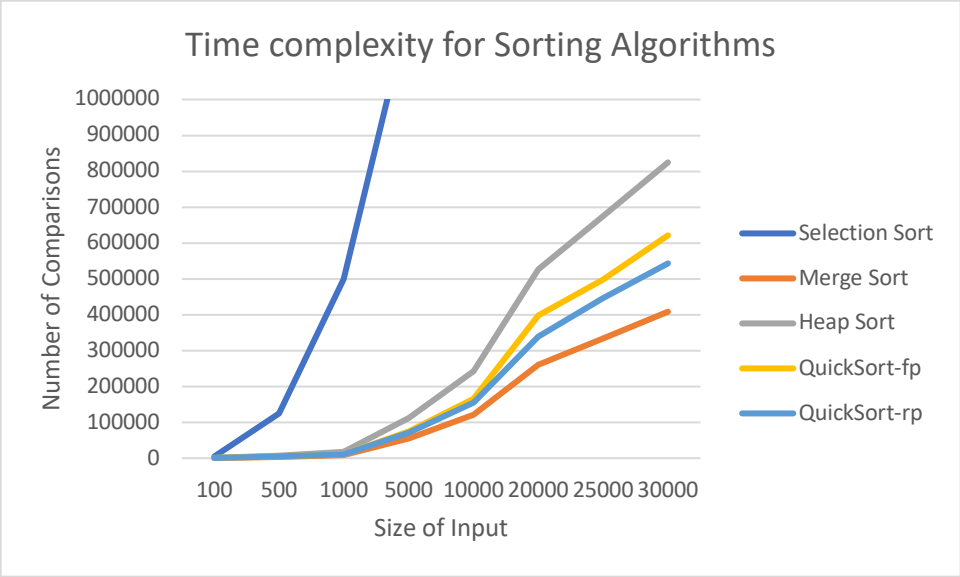

Heap Sort
Experiemental Data
O(nlogn)
900000
800000
700000
600000
500000
400000
300000
200000
100000
0
Number of Comparisons
100
500
1000
5000
10000
20000
25000
30000
Size of Input


QuickSort-fp
Number of Comparisons
700000
600000
500000
400000
300000
200000
100000
0
100
500
1000
5000
10000
20000
25000
30000
Size of Input


Quick Sort-fp
Experiemental Data
O(nlogn)
Number of Comparisons
700000
600000
500000
400000
300000
200000
100000
0
100
500
1000
5000
10000
20000
25000
30000
Size of Input


QuickSort-rp
600000
500000
400000
300000
200000
100000
0
Number of Comparisons
100
500
1000
5000
10000
20000
25000
30000
Size of Input


Quick Sort-rp
Experiemental Data
O(nlogn)
600000
500000
400000
300000
200000
100000
0
Number of Comparisons
100
500
1000
5000
10000
20000
25000
30000
Size of Input



b. You will then provide some discussion about your results. Compare the theoretical result
with your experimental result for each algorithm. Does your experimental result coincide
with the Big-O of that specific algorithm? Describe if there may be some inconsistencies
between your plot and what the theoretical plot looks like.

- **Selection Sort:** According to the theoretical result of the selection sort algorithm, the number of comparisons being made for an input size of 10,000 ordered, random, or reversed sorted numbers lead to a $O(n^2)$ complexity. The number of comparisons were almost squared of the initial 10,000 input size. This complexity seems to hold true with the experimental results with different input sizes of unsorted data. From 100 input size to 30,000 input size, the number of comparisons includes 4 - 8 digits until the maximum input size of 30,000 is reached. The selection sort plot maps to a very steep increasing curve which is common for x^2 functions in mathematics. From this point alone, the pattern of the selection sort algorithm, for any input size, the curve is in a shape of n^2 which verifies to the theoretical result that selection sort is a $O(n^2)$ algorithm.

- **Merge Sort:** According to the theoretical result of the merge sort algorithm, the number of comparisons being made for an input size of 10,000 ordered, random, or reversed sorted numbers lead to a

O(nlogn) complexity. Because nlogn where n = 10,000 would be 40,000, we are to expect that merge sort would have the total number of comparisons including at most 6 digits for any input size up to 30,000. This holds true from our experimental results where all of the merge sort averages, for the various input sizes, are within 5-6 digits. Also, the plot for merge sort is far less step than that of selection sort and increases greater than linear but smaller than a quadratic function. This is common behavior of a xlogx curve. The shape of the curve, using multiple input sizes, verifies our theoretical predication that merge sort is an O(nlogn) algorithm for any input size.

- **Heap Sort:** According to the theoretical result of the heap sort algorithm, the number of comparisons being made for an input size of 10,000 ordered, random, or reversed sorted numbers lead to a O(nlogn) complexity. Because nlogn where n = 10,000 would be 40,000, we are to expect that heap sort would have the total number of comparisons including at most 6 digits for any input size up to 30,000. This holds true from our experimental results where all of the heap sort averages, for the various input sizes, are within 5-6 digits. Also, the plot for heap sort is far less step than that of selection sort and increases greater than linear but smaller than a quadratic function. This is common behavior of a xlogx curve. The shape of the curve, using multiple input sizes, verifies our theoretical predication that heap sort is an O(nlogn) algorithm for any input size.

- **Quick Sort-fp:** According to the theoretical result of the quick sort-fp algorithm, the number of comparisons being made for an input size of 10,000 random sorted numbers lead to a O(nlogn) complexity. But when the data inputted into quick sort-fp is already pre-sorted, the number of comparisons will be quadratic function of $O(n^2)$. This means that the worst-case scenario for quick sort-fp is when we enter an already sorted array as an input to be sorted by this algorithm. But, since our experimental data will generate random numbers in the input array, this will very least likely be the case of the data being pre-sorted. As such our experimental data that we found followed random data that was not pre-sorted and followed nlogn number of comparisons. Because nlogn where n = 10,000 would be 40,000, we are to expect that quick sort-fp would have the total number of comparisons including at most 6 digits for any input size up to 30,000

(randomly unsorted data sets). This holds true from our experimental results where all of the quick sort-fp averages, for the various input sizes, are within 5-6 digits. Also, the plot for quick sort-fp is far less step than that of selection sort and increases greater than linear but smaller than a quadratic function. This is common behavior of a xlogx curve. The shape of the curve, using multiple input sizes, verifies our theoretical predication that quick sort-fp is an O(nlogn) algorithm for any input size (given unsorted inputs).

- **Quick Sort-rp:** According to the theoretical result of the quick sort-rp algorithm, the number of comparisons being made for an input size of 10,000 ordered, random, or reversed sorted numbers lead to a O(nlogn) complexity. Because nlogn where n = 10,000 would be 40,000, we are to expect that quick sort-rp would have the total number of comparisons including at most 6 digits for any input size up to 30,000. This holds true from our experimental results where all of the quick sort-rp averages, for the various input sizes, are within 5-6 digits. Also, the plot for quick sort-rp is far less step than that of selection sort and increases greater than linear but smaller than a quadratic function. This is common behavior of a xlogx curve. The shape of the curve, using multiple input sizes, verifies our theoretical predication that heap sort is an O(nlogn) algorithm for any input size.

- **Time Complexity of Sorting Algorithms**: Looking at the last graph, that shows all five of the sorting algorithms curves being plotted on the same graph of various random input sizes, there seems to be a clear efficient algorithm to use. We want to achieve a complexity that is either O(1) or close to O(1) complexity for an algorithm runtime. O(1) curve is a flat horizontal line across the x-axis. Selection sort is the most inefficient algorithm, as depicted by the graph, as being a very steep increasing curve that is far from being constant. Heap Sort, Merge Sort, Quick Sort-fp, and Quick Sort-rp, all have the same general curve behavior of following a O(nlogn) for our randomized unsorted data as an input. The reason as to why heap sort is amongst the most inefficient sorting algorithms **amongsts all the O(nlogn) algorithms** is because the extra time it takes to create a heap for the input array. Every swapping of nodes, after a comparison is made, has to call the heapify function to ensure the order property is maintained in the created heap. Because of this reheap up and down are called

accordingly which increases the amount of work that heap sort algorithm has to do, and hence the number of comparisons is increased, leading to a slightly less efficient algorithm. Based on the graph, the algorithm amongst O(nlogn), that is closest to O(1) horizontal curve would be the most efficient algorithm to use. Hence, merge sort is the most efficient sorting algorithm to use for any unsorted input size.