2022

Sorting Algorithm Analysis

IMPLEMENTED IN C++

KEESE PHILLIPS

Algorithm	***Input Type***	***Number of Comparisons***	***Comments***
Selection Sort	Ordered	Greater than 49995000	**$O(n^2)$** Total comparisons increase dramatically as input size increases
	Random	Greater than 49995000	
	Reversed	Greater than 49995000	
Merge Sort	Ordered	69008	**O(nlog(n))** Second fewest comparisons for all inputs.
	Random	120414	
	Reversed	64608	
Heap Sort	Ordered	244460	**O(nlog(n))** Greatest number of comparisons, except for selection sort, no matter input.
	Random	235430	
	Reversed	226682	
Quick Sort	Ordered	Greater than 49995000	**O(nlog(n))** Ineffective when the data is already sorted, comparisons approach $O(n^2)$.
	Random	161842	
	Reversed	Greater than 49995000	
Quick Sort (Randomized)	Ordered	143214	**O(nlog(n))** Improves the Quick Sort complexity when the data is sorted in some form.
	Random	150733	
	Reversed	156982	

Merge Sort uses a significant amount of memory as the input size increases, due to the use of temporary arrays. Merge Sort requires such use due to breaking the array down and the merging the two in a sorted fashion. Merge Sort was implemented by using heap memory, so if the input size is significantly high then allocation of memory might become an issue.

Selection Sort seems very inefficient because it offers the most comparisons, although it does not require the use of allocated memory. Merge Sort is very effective when the input size is low but as input size increases a concern with memory allocation might become more of an issue. Heap Sort does not perform as efficiently as the other sorting algorithms, expect for selection sort, but might be implemented in cases when one might need only the greatest value. Quick Sort is the most effective sorting algorithm when the data is unsorted but if the data is sorted, one might consider implementing a bubble sort with a flag to check if the data is sorted so the complexity is O(n). Moreover, a Randomized Quick Sort would also address the issue of the Quick Sort complexity approaching $O(n^2)$, although Randomized Quick Sort did perform slightly worse than the non-randomized partition.

Discussion

Input Size	***Selection***	n^2	***Merge***	***Heap***	***Quick***	***Random Quick***	*2nlog(n)*
10	45	100	23	34	20	23	66
50	1225	2500	227	417	195	233	564
100	4950	10000	542	1035	460	592	1329
250	31125	62500	1683	3232	1578	2120	3983
500	124750	250000	3816	7405	3036	4567	8966
750	280875	562500	6217	12063	6937	7594	14326
1000	499500	1000000	8708	16842	9401	10146	19932
2500	3123750	6250000	25134	48904	32928	32126	56439
5000	12497500	25000000	55284	107718	56135	73757	122877
7500	28121250	56250000	87188	170157	105340	121002	193090
10000	49995000	100000000	120381	235318	130885	164602	265754
15000	112492500	225000000	189411	370498	193024	251454	416180
20000	199990000	400000000	260857	510676	289654	324937	571508
25000	312487500	625000000	333943	654989	408938	420303	730482
50000	1249975000	2500000000	718315	1409407	834104	1009061	1560964

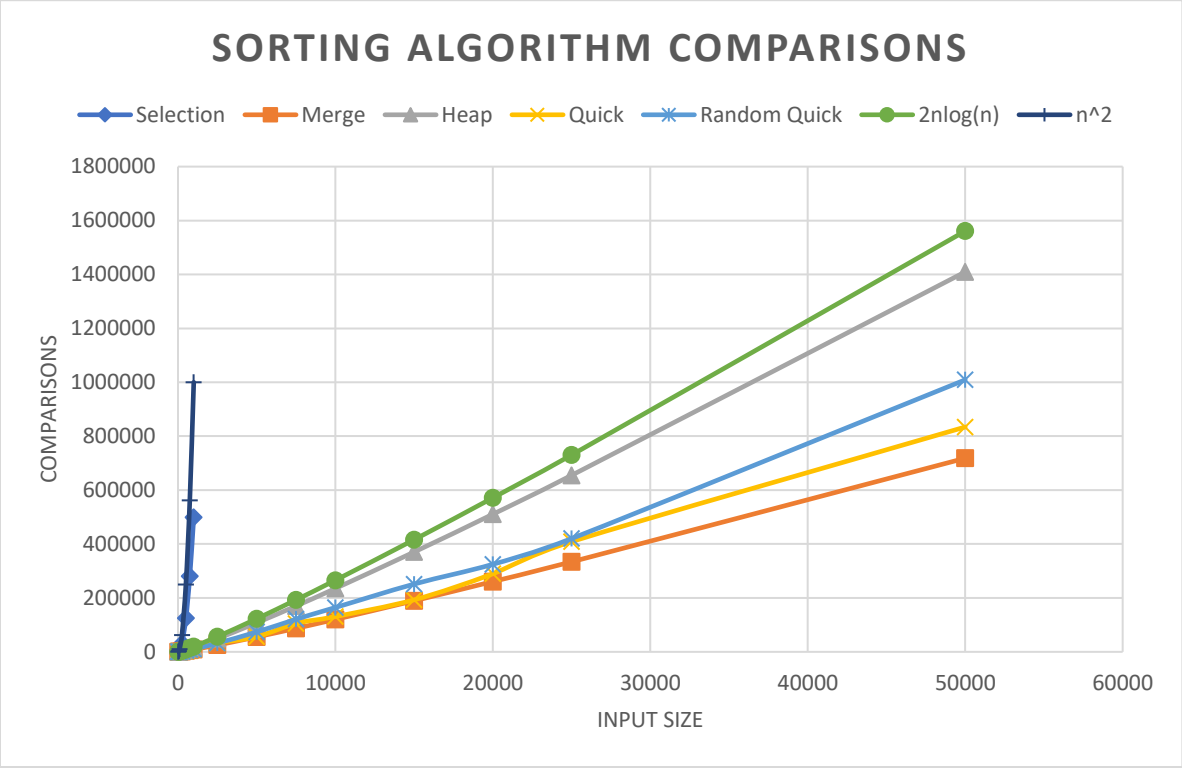
The theoretical results seem to coincide with the experimental results. Although the differing sorting algorithms have the same complexity of O(nlog(n)), except for Selection Sort, the sorting algorithms differ most significantly as the input size increases towards 50,000, see *Appendix I*. The *Selection Sort* algorithm seems to coincide with the theoretical $O(n^2)$ complexity, seemingly very inefficient as input size increases. Selection Sort does not seem to offer any benefit that the more complex algorithms with O(nlog(n)) does not already offer except for the lack of allocated memory Merge Sort requires. *Merge Sort* seems to follow along with the O(nlog(n)) complexity and seems to perform better than all other algorithms in terms of comparisons when input is larger than 750. *Heap Sort* seems to become less and less efficient as input size increases. While the other O(nlog(n)) sorting algorithms are clustered together, Heap Sort comparisons seem to increase at a faster rate. While *Quick Sort (Randomized)* algorithm improves upon *Quick Sort (Non-Randomized)* when the data is sorted, Quick Sort (Randomized) seems to always provide an upper bound for Quick Sort (Non-Randomized). Quick Sort (Randomized) seems to provide the best tradeoff in terms of both resources and efficiency. When resources are not a concern Merge Sort would provide the fewest comparisons, but if resources are scarce then Merge Sort might cause some memory allocation problems.

Some of the inconsistencies between the actual and the theoretical results occur due to the input data. Depending on how the data is arranged some algorithms will perform better than others. For example, Bubble Sort is a very inefficient sorting algorithm when the data is unsorted, but if the data is already sorted Bubble Sort can be implemented to achieve a complexity of O(n), which is better than any other complexities for comparison-based sorting algorithms. The Merge Sort algorithm seems to implement the fewest, or very close to the fewest, comparisons for any size of unsorted data. Heap Sort on-the-other-hand seems to

implement the greatest number of comparisons, except for Selection Sort. Quick Sort seems to implement the most efficient comparison-based sorting algorithm due to the limited size of memory required and when implemented with a random partition will provide very close to the number of comparisons Merge Sort offers, without the use of allocated memory.

Appendices

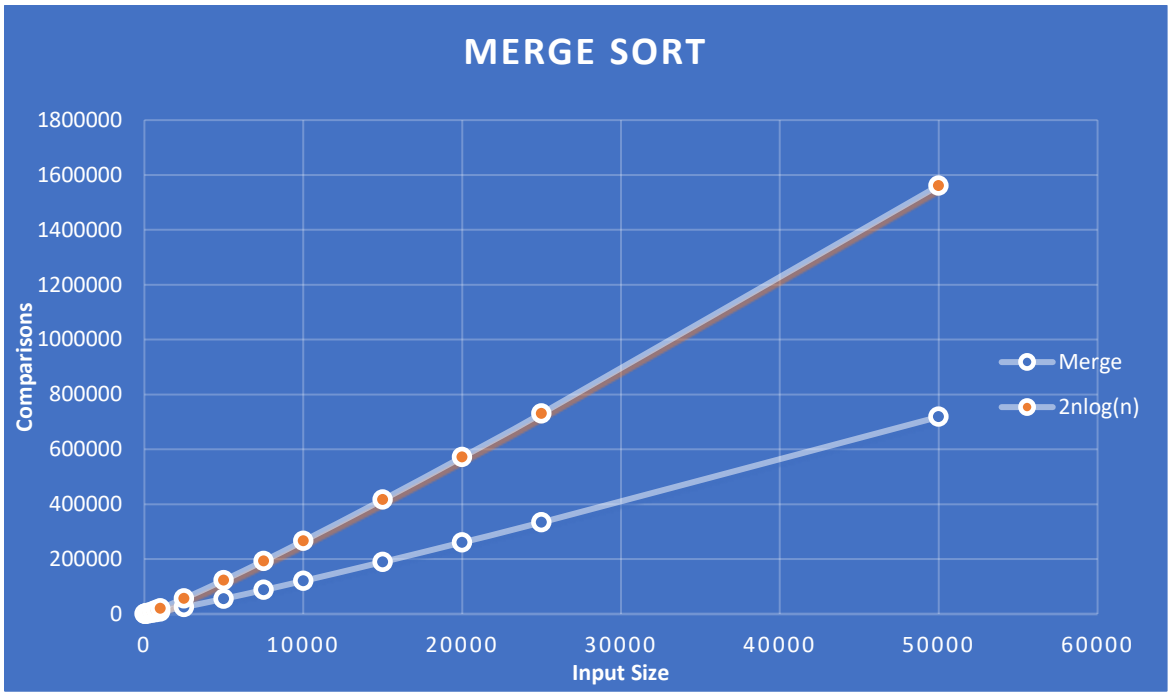
Appendix I: Sorting Algorithm Comparisons



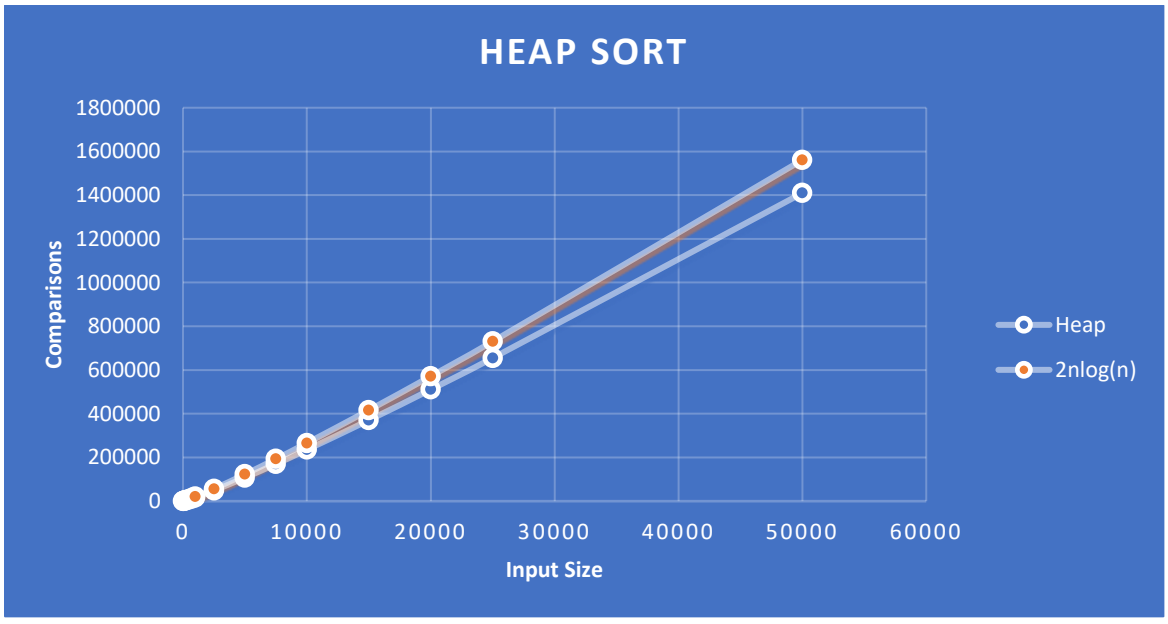
Selection Sort and n^2 distorted the graph when input size > 1,000 and were not included

Appendix II: Selection Sort

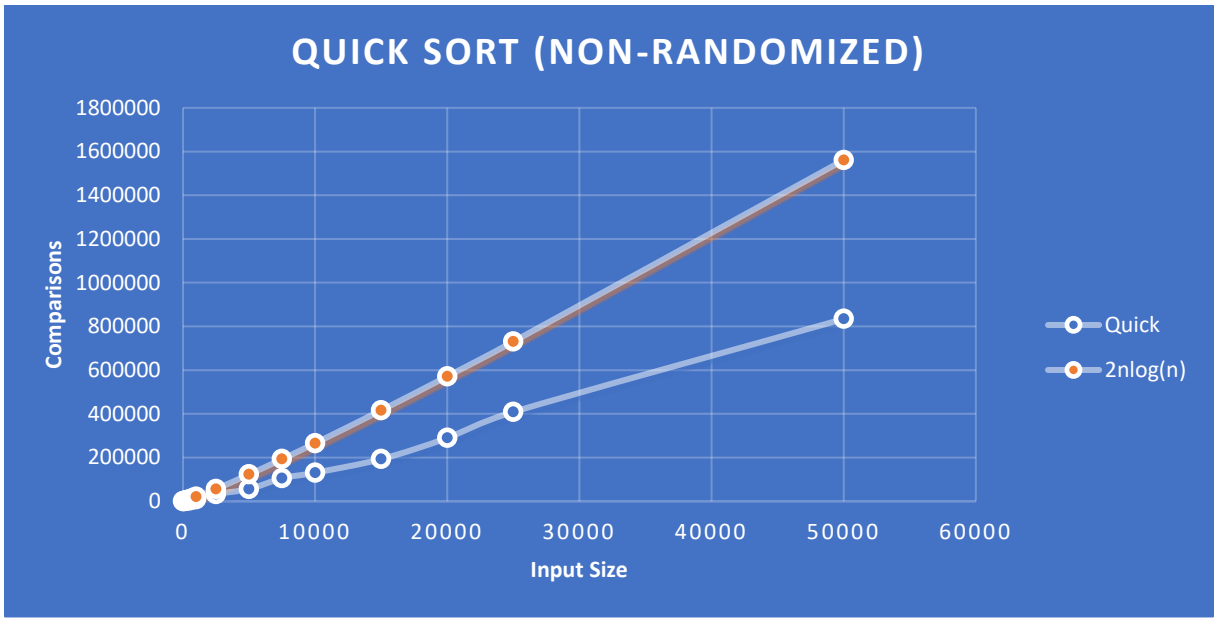
Appendix III: Merge Sort



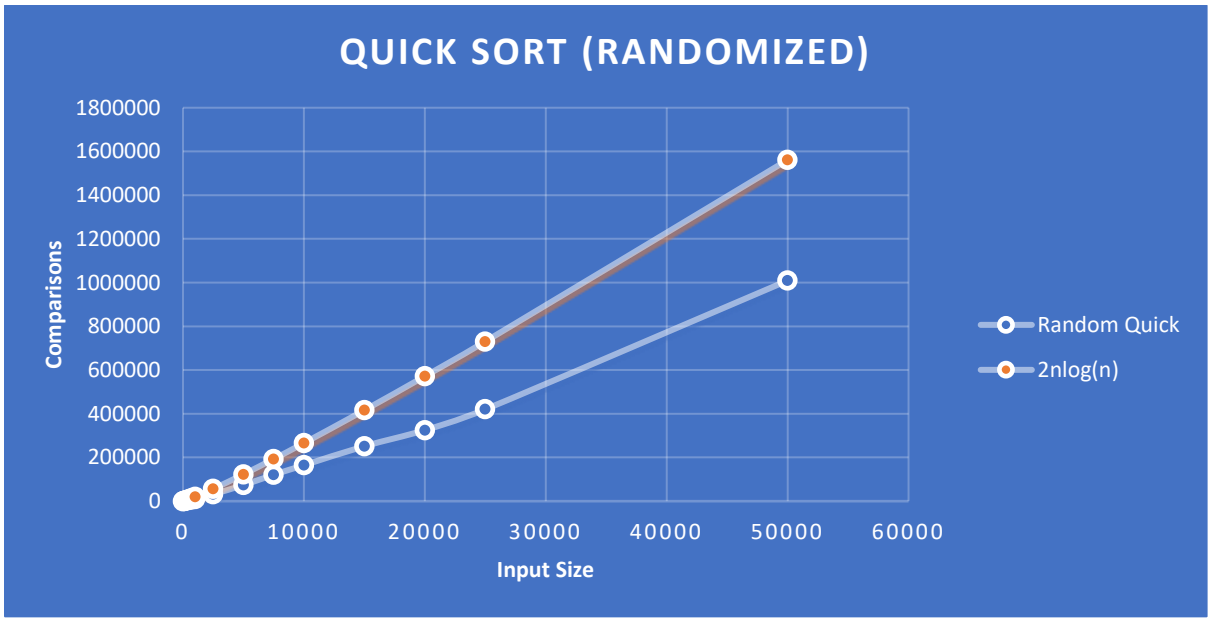
Appendix IV: Heap Sort



Appendix V: Quick Sort



Appendix VI: Randomized Quick Sort



References

C++ Plus Data Structures, Nell Dale, Jones & Bartlett Learning, 6th Ed., 2018.