CSE 2046

Analysis of Algorithms

Homework I: *Experimental Analysis*

Anılcan Erciyes, 150119520

Fatih Said Duran, 150119029

Anılcan Erciyes, 150 119 520

Fatih Said Duran, 150 119 029

INTRODUCTION

In this project, our task is to highlight the characteristics of various sorting algorithms, by analyzing their performances with respect to differently curated input sets. The eventual aim is to successfully verify that the theoretical/mathematical knowledge and the practical running complexities are in harmony.

In order to achieve this task, our painstakingly programmed source code can generate different kinds of input sets:

- Sorted
- Reverse sorted
- Almost sorted
- Duplicate
- Random.

The program takes three inputs from the user: The array size, the characteristic and the number of repeats. Then, it immediately generates the desired input set, traverse through different sorting algorithms by using it and eventually, note their performances to the console in a table-like manner.

As the name states, in random characteristic, the program generates all the numbers in a completely desultory way. This kind of input sets will be used to demonstrate how the sorting algorithms behave in the "Average Case". However, in order to make sure that the machine does not accidentally create a set that is in fact not that "average", this process is repeated fifty times.

Moreover, duplicate characteristic shows indicate an array where there is a vast number of same elements. Since swapping operations on duplicate members is the most crucial aspect of stable vs. non-stable algorithms, our aim was to display how the algorithms react to mostly duplicated inputs.

RUNNING THE PROGRAMME & SCREENSHOTS

In this step, the results for different characteristics and input sizes will be obtained and the console screenshots from the program will be used. Then, at the next step, all this information will be combined and demonstrated in a table to make comments.

Average complexity table of all unique inputs, based on time :

Random 1000 0-to-1000	:	110482	260142	174933	71456	45956	90372	76985
Random 2000 0-to-2000	:	377168	299092	359787	143983	93473	123314	22633
Random 3000 0-to-3000	:	849769	485638	590704	222598	146630	185578	27146
Random 4000 0-to-4000	:	1509465	793832	793721	299569	199201	257643	32215
Random 5000 0-to-5000	:	2338824	1165089	1003558	378217	249160	325879	39427
Random 6000 0-to-6000	:	3290706	1611772	1202607	458118	305144	396023	46002
Random 7000 0-to-7000	:	4458753	2117959	1282029	541604	361080	469631	41931
Random 8000 0-to-8000	:	5818573	2706498	1472148	622830	417616	548556	48014
Random 9000 0-to-9000	:	7357109	3360594	1667803	704877	478483	622649	54692
Random 10000 0-to-10000	:	9156619	4140524	1923445	793650	540311	704463	64593
Sorted 1000 0-to-1000	:	1289	9713	139744	612819	374632	32186	5851
Sorted 2000 0-to-2000	:	2551	19201	278754	2322475	1475213	86760	10800
Sorted 3000 0-to-3000	:	3819	30692	421164	5142440	3304855	133515	16294
Sorted 4000 0-to-4000	:	5097	42529	566766	9029276	5868484	180167	22253
Sorted 5000 0-to-5000	:	6374	58417	709499	15962398	9153586	229361	27424
Sorted 6000 0-to-6000	:	7621	71810	855904	20149756	13200728	277744	33458
Sorted 7000 0-to-7000	:	8971	86382	1000399	27378756	17969391	326848	37182
Sorted 8000 0-to-8000	:	10346	97196	1142870	37402310	23466014	375955	42364
Sorted 9000 0-to-9000	:	11507	99687	1286829	45150210	29623401	423007	47361
Sorted 10000 0-to-10000	:	13172	113932	1449748	56785103	36545432	474134	53609
Almost sorted 1000 0-to-1000	:	5704	14328	153159	399262	212667	39219	7112
Almost sorted 2000 0-to-2000	:	12053	28480	281456	1408247	978714	87716	10797
Almost sorted 3000 0-to-3000	:	18647	43035	462152	2916669	1861995	136771	19721
Almost sorted 4000 0-to-4000	:	25439	57852	648205	4981155	3235551	182894	28132
Almost sorted 5000 0-to-5000	:	33078	73126	815182	5912146	4839833	233502	35467
Almost sorted 6000 0-to-6000	:	39258	89956	983186	10337976	6958270	281029	42054
Almost sorted 7000 0-to-7000	:	45406	103244	1142643	14305140	10651589	331171	48575
Almost sorted 8000 0-to-8000	:	51795	121939	1302547	17201624	15676514	374989	56308
Almost sorted 9000 0-to-9000	:	64876	138401	1437713	16991179	17052639	429025	50455
Almost sorted 10000 0-to-10000	:	70827	153081	1447235	27857191	19347677	474503	53773
Reverse sorted 1000 0-to-1000	:	209754	107505	153744	485884	215712	41091	7013
Reverse sorted 2000 0-to-2000	:	746954	345423	281795	1475538	694263	85883	10995
Reverse sorted 3000 0-to-3000	:	1669110	756305	427763	3215116	1570782	137919	16739
Reverse sorted 4000 0-to-4000	:	2955524	1319945	574997	5747143	2754453	185615	22516
Reverse sorted 5000 0-to-5000	:	4605673	2041995	715912	8762135	4285257	235825	27978
Reverse sorted 6000 0-to-6000	:	6581585	2903505	859171	12732623	6270810	287210	32393
Reverse sorted 7000 0-to-7000	:	9017177	3962561	1011566	17096489	8720641	339360	40472
Reverse sorted 8000 0-to-8000	:	11864729	5189631	1160352	23607561	11416768	385137	43011
Reverse sorted 9000 0-to-9000	:	14957810	6561993	1306621	30767507	14694815	439716	48681
Reverse sorted 10000 0-to-10000	:	18459633	8117016	1451802	34519801	18286384	492120	52812
Duplicate 1000 0-to-6	:	83811	57189	160939	153764	150472	43642	5537
Duplicate 2000 0-to-6	:	333861	184303	321955	479487	609758	91135	10159
Duplicate 3000 0-to-6	:	717174	377865	485874	949058	1316714	137452	15371
Duplicate 4000 0-to-6	:	1255075	642887	652371	1589837	2217928	184487	20321
Duplicate 5000 0-to-6	:	1971852	970496	820212	2402050	3445831	232278	25446
Duplicate 6000 0-to-6	:	2823120	1397369	983300	3346370	5082200	278327	30328
Duplicate 7000 0-to-6	:	3842224	1872909	1146181	4492655	6754152	325162	35228
Duplicate 8000 0-to-6	:	5004288	2391251	1313578	5744683	8923791	373426	40175
Duplicate 9000 0-to-6	:	6290262	3017861	1480567	7239787	11145902	421615	45014
Duplicate 10000 0-to-6	:	7758945	3702320	1644594	8810534	13285554	470028	49894
Random 10000 0-to-1000	:	9220179	4059325	1888070	925559	527013	697831	47267
Random 10000 0-to-2000	:	9243229	4113457	1895018	872184	528664	702044	48175
Random 10000 0-to-3000	:	9263321	4135616	1902661	843621	536906	700842	49089
Random 10000 0-to-4000	:	9248740	4156677	1893150	822333	535610	704527	52273
Random 10000 0-to-5000	:	9295637	4170730	1997628	816395	538489	704119	59589
Random 10000 0-to-6000	:	9242534	4168480	2035488	808044	539179	705989	64517
Random 10000 0-to-7000	:	9247642	4190422	2092514	804843	536577	705394	71110

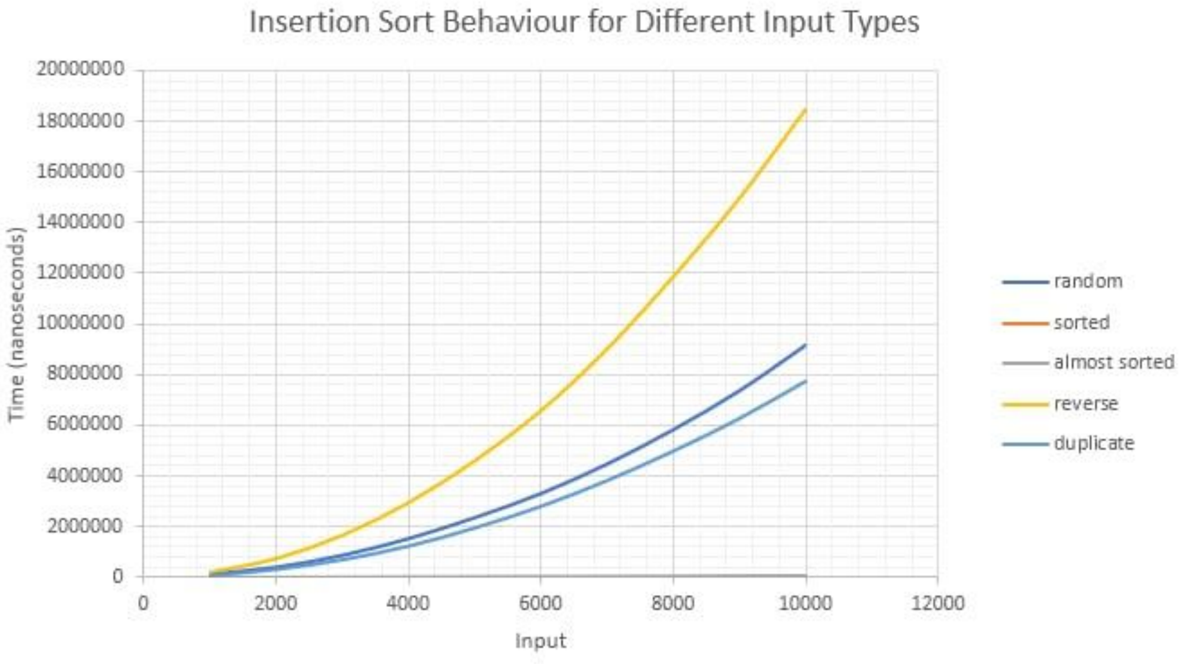
Console output for all experiments. All experiments are conducted for 100 times and the average values are taken into eventual consideration.

These data will be used for the upcoming graphs

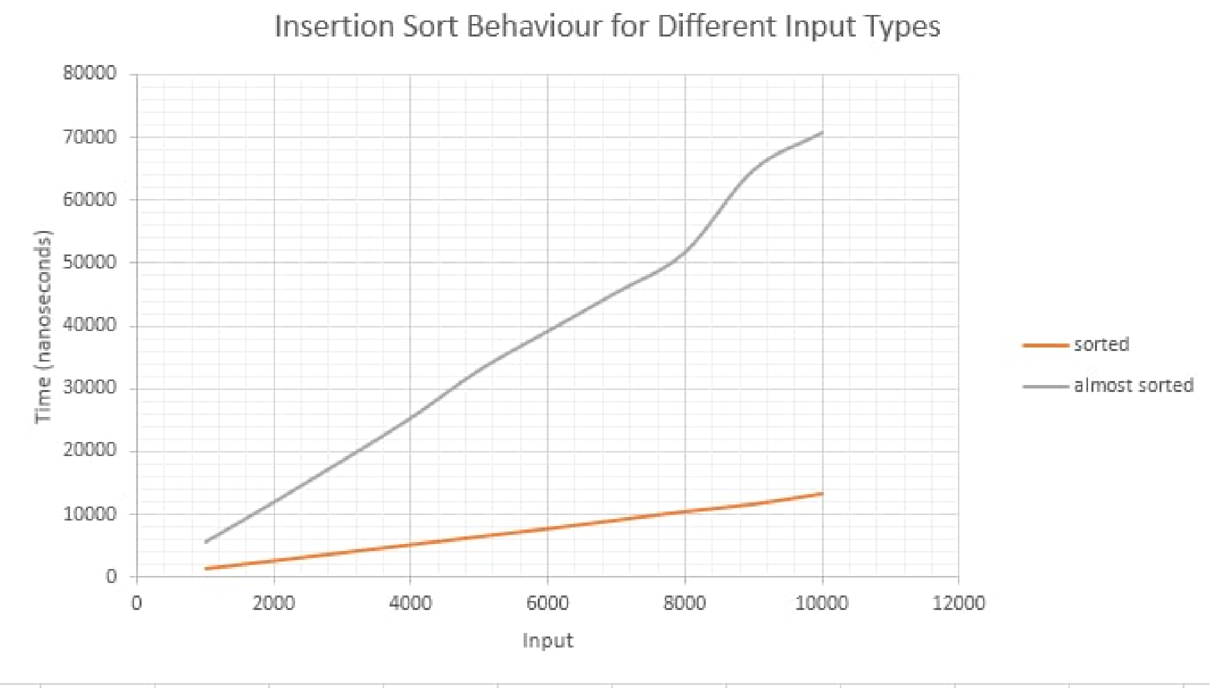
ILLUSTRATING THE RESULTS: TABLES AND GRAPHS

Sort Algorithms Behavior for Different Types of Inputs:

I. Insertion Sort, zoomed out and zoomed in

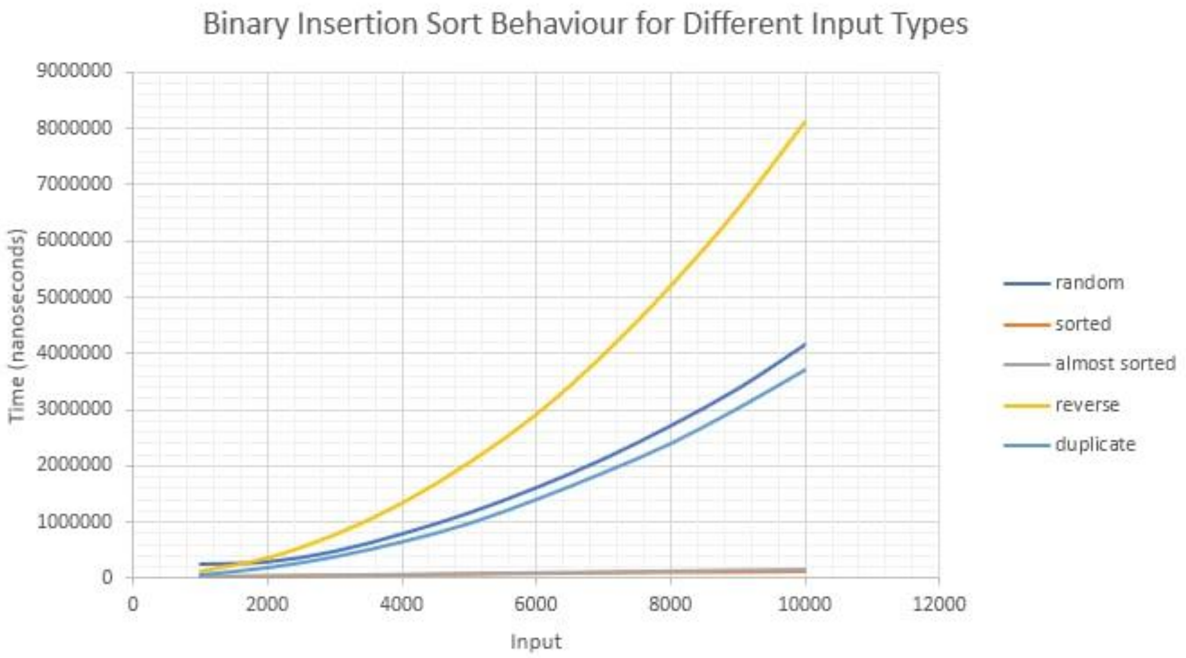


➔ We can see the quadratic time complexity obtained from reverse sorted, random and duplicate inputs. If we look at the complexity results of input with a size of 2000 and compare it with complexity results of input with a size of 4000, we can see that the complexity behaves quadratically for reverse sorted, random and duplicate inputs, which means O(n^2) complexity.



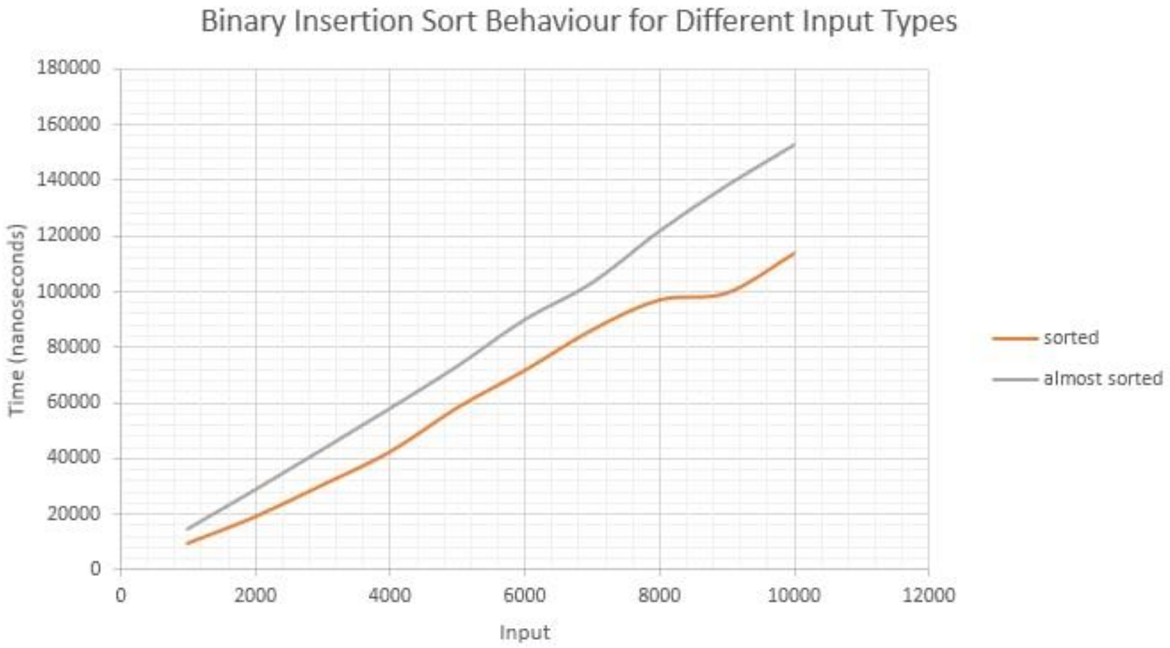
- ➔ When we zoom in and look at sorted and almost sorted inputs complexity results, we can see that there is linear behavior. Complexity results of 2000 sized input and 4000 sized input give us O(n) complexity. Almost sorted inputs give Ω (n).

II. Binary Insertion Sort, zoomed in and zoomed out



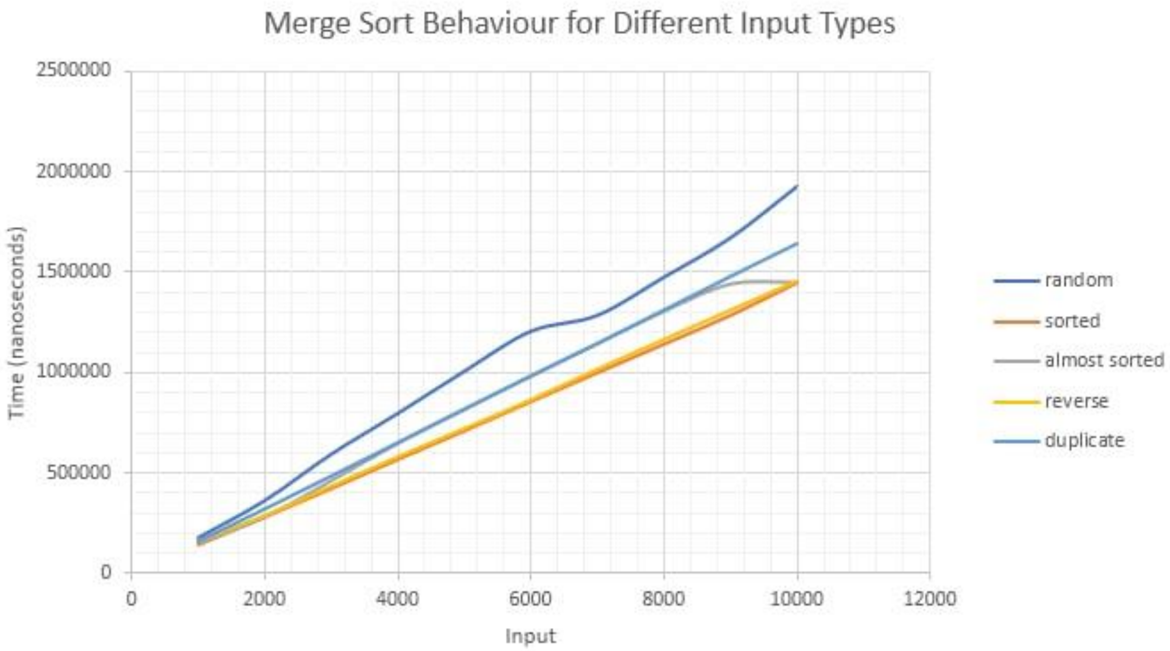
- ➔ We can see that reverse sorted input complexity results is quadratic just like in normal Insertion Sort, albeit smaller. Random and duplicate inputs give a much smaller

complexity than O(n^2) but give a bigger than O(nlogn) complexity.
Which is Ω (nlogn).



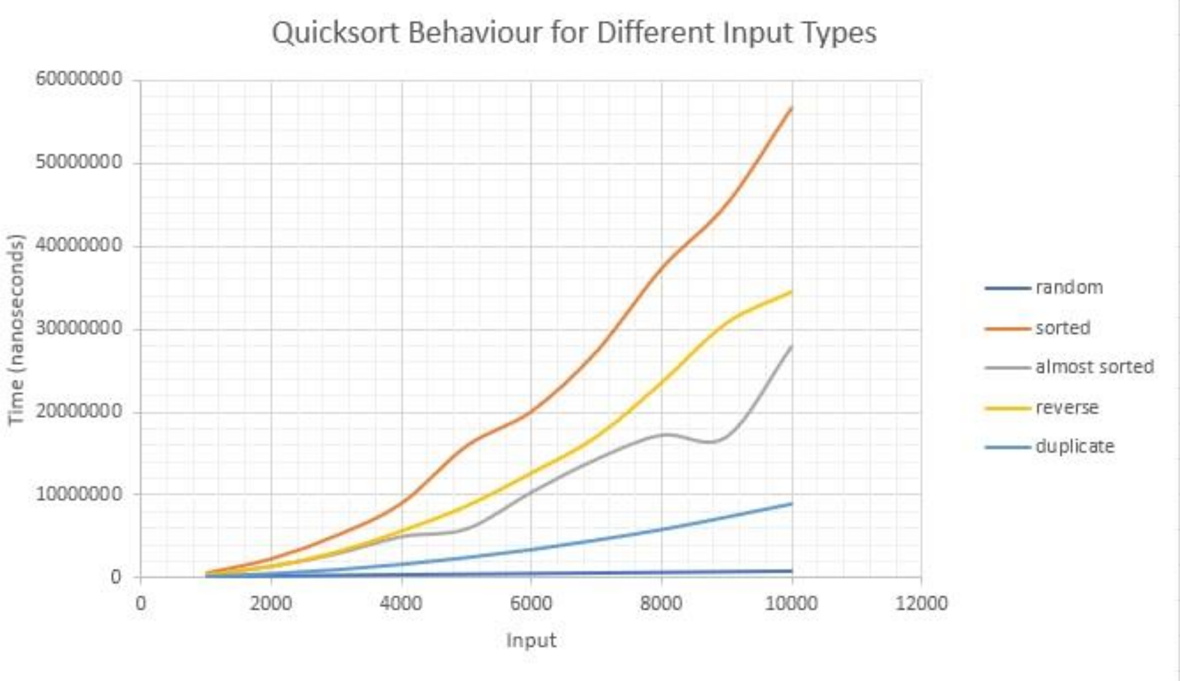
➔ Sorted and almost sorted inputs give linear complexity. We can see that almost sorted input shows a very close graph with sorted input complexity O(n).

III. Merge Sort



➔ For merge sort all the different inputs give the same complexity behaviour. For random and sorted inputs, the coefficient may be different but overall by comparing different size input complexities, we can safely say all types of inputs give O(nlogn) complexity.

IV. Quick Sort



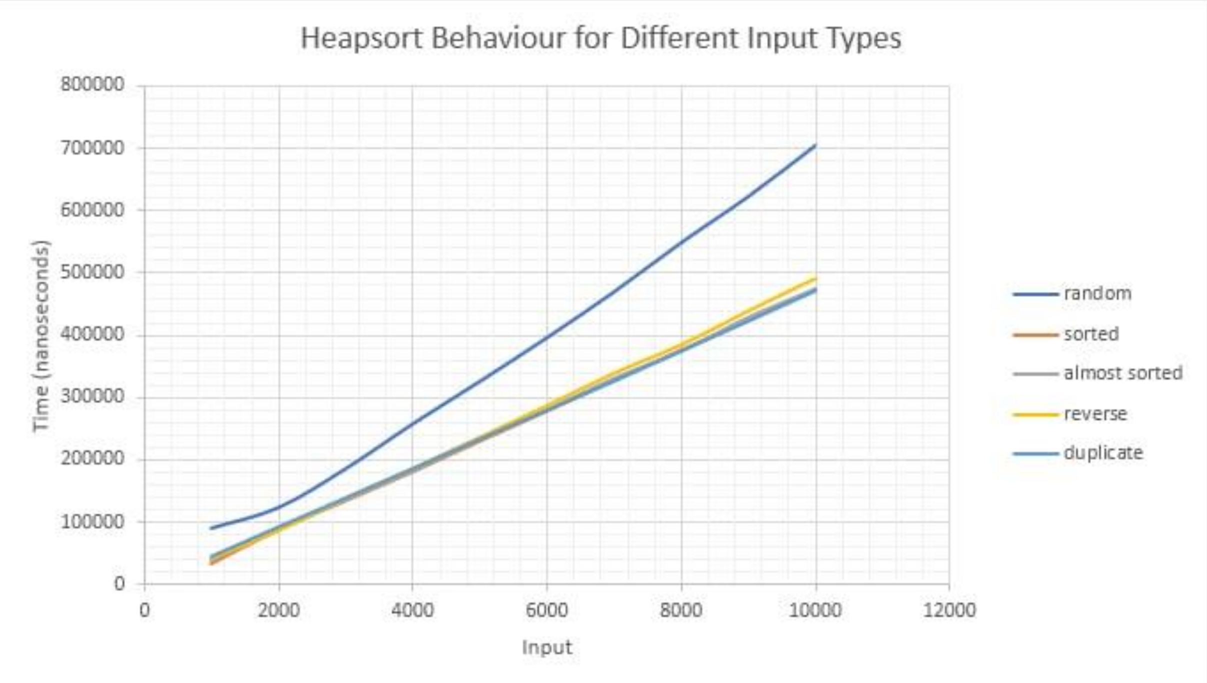
➔ Quicksort gives a fast result for random inputs, by comparing the different sized inputs complexity values with each other, we see O(nlogn) complexity. The rest of the inputs give quadratic graphs for us. Quicksort's best case may be its average case handling random inputs, but all our other inputs have given worst case complexity of O(n^2).

We witness oscillations for this experiments results. One guess we have is paging increasing the sorting time. For almost sorted inputs oscillations, perhaps some inputs were not that sorted and as a result gave a faster sorting time complexity.

V. Quick Sort using Median-of-Three

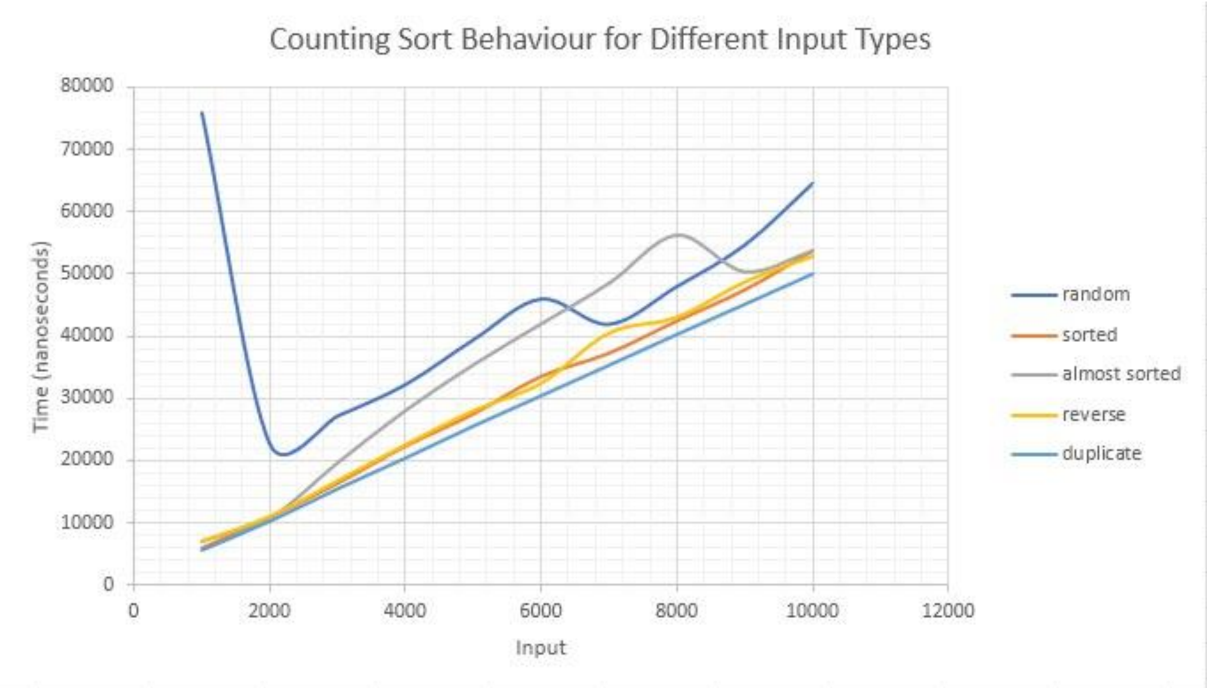
➔ Median of Three Quicksort is an optimization for Quicksort, and we can see that in our results. It has a goal of achieving O(nlogn) complexity for worst case as well. In our experiments (which have done hundreds of times) despite being faster, could not achieve O(nlogn) time complexity for its worst cases.

VI. Heap Sort

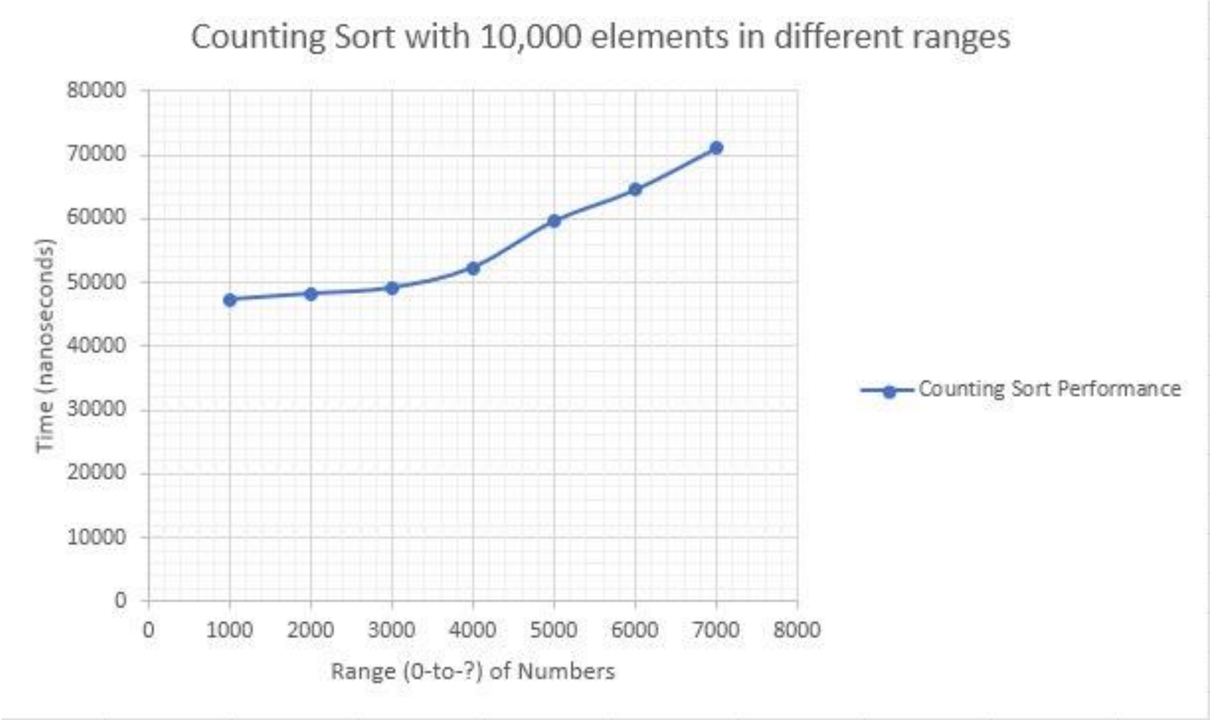


➔ Heap sort has the same time complexity behavior for all our different types of inputs. Other than random inputs, all of them had very close results. Comparing different sized inputs time complexities with each other, we see that all the inputs give O(nlogn) complexity.

VII. Counting Sort, understanding the role of 'n' and 'r'



- Counting sort’s complexity graphs have lots of oscillations, especially for random inputs and almost sorted inputs. I speculate that the reason for this was unforeseen added time while trying to use the memory, which in our experiment must have happened more when dealing with more random inputs. The overall trend of the counting sort time complexity is safely to say linear (in our experiments case where input size and maximum integer increased linearly). As a result, we can understand where the n comes from in O(n + r) complexity of counting sort.

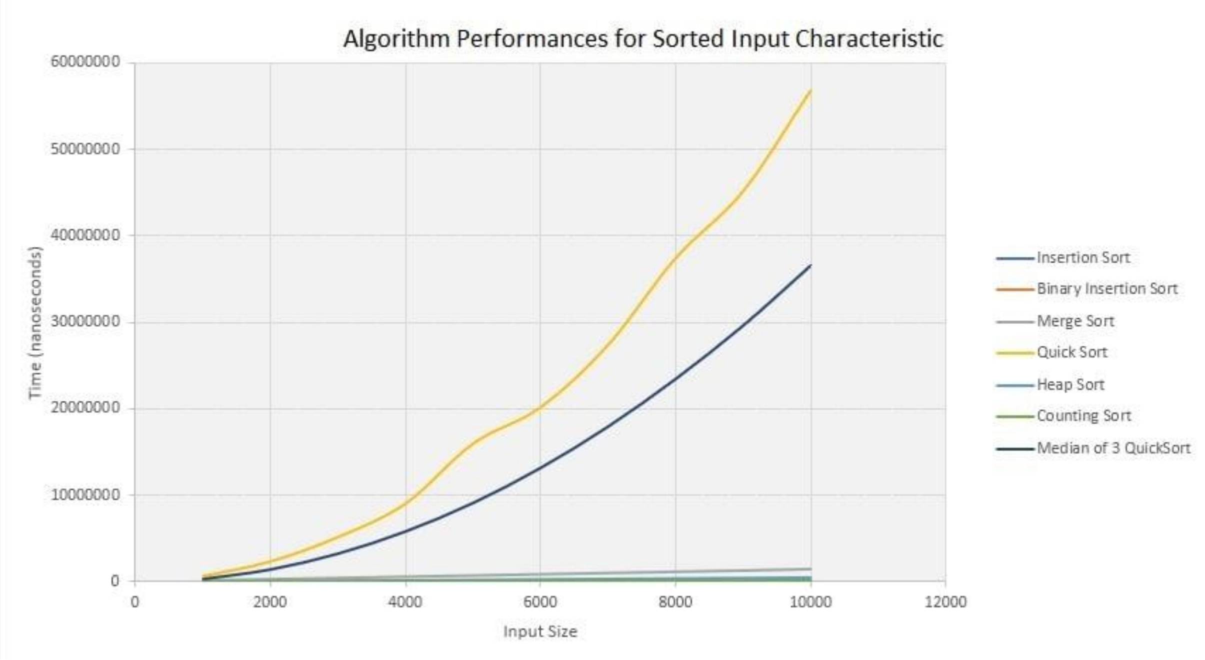


- To observe the effect of increasing the value the max integer in an input we made this graph. Although not exact, we witness a linear increase as our r (max integer value) increases 1000 by 1000, with a random input that has constant size of 10000.

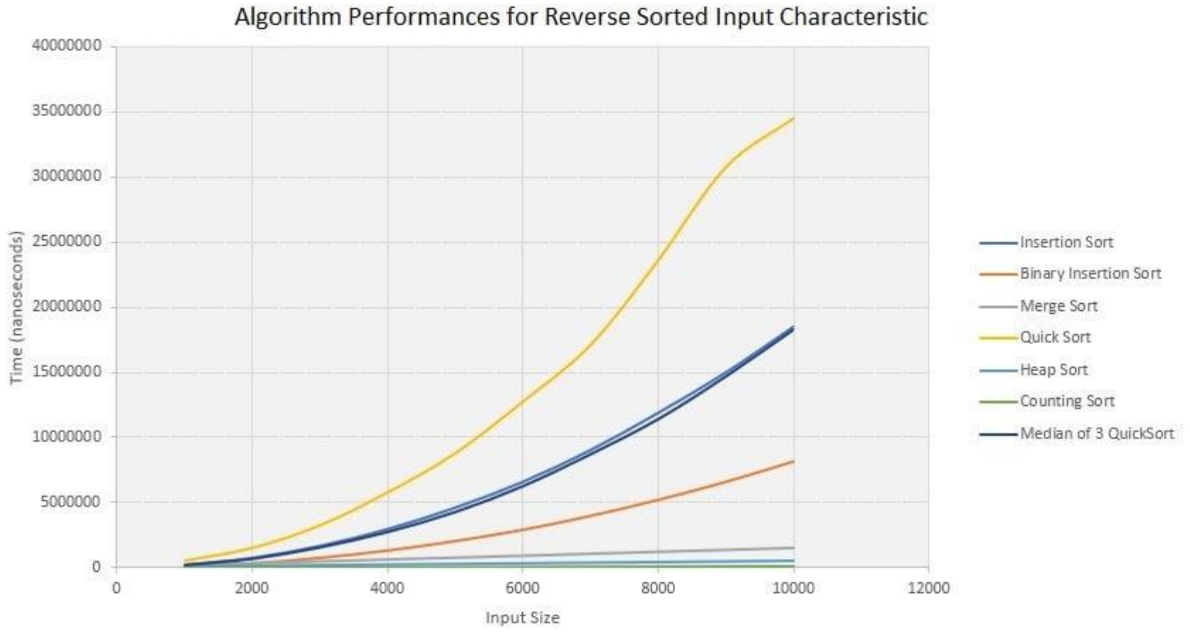
  Looking at these two graphs we can see that the type of the input doesn’t affect time complexity drastically for counting sort. What matters is the size of the input and what the max variable is: O(n+r).

Sort Algorithms Compared for Different Types of Inputs:

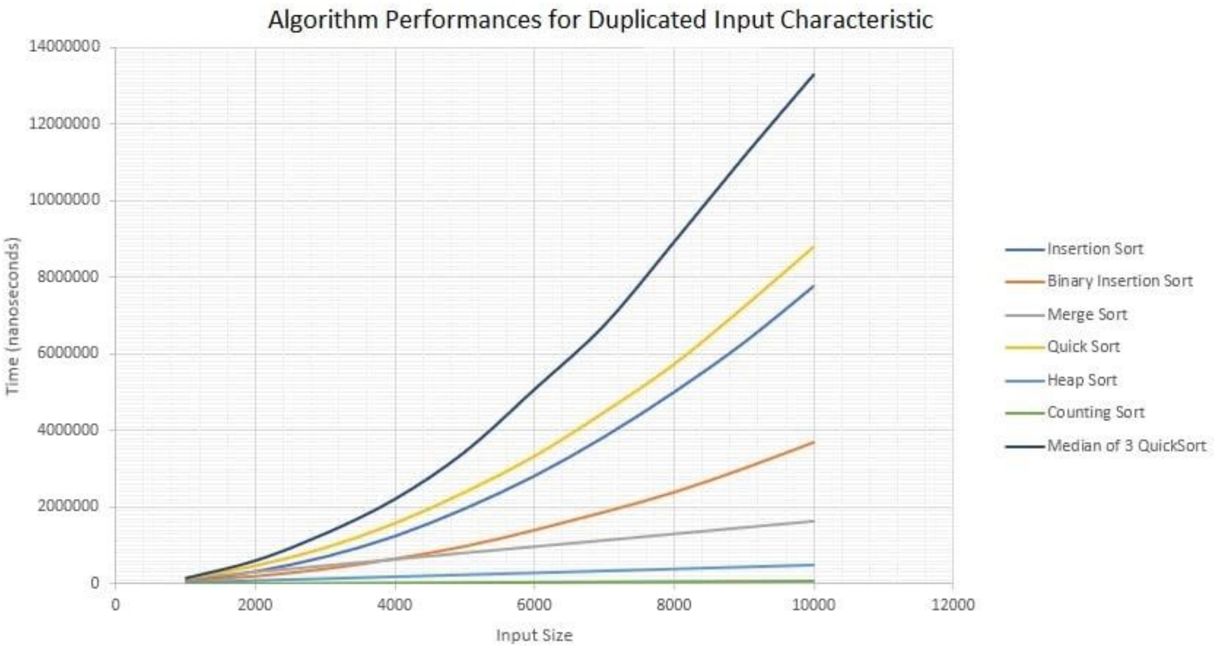
I. Sorted



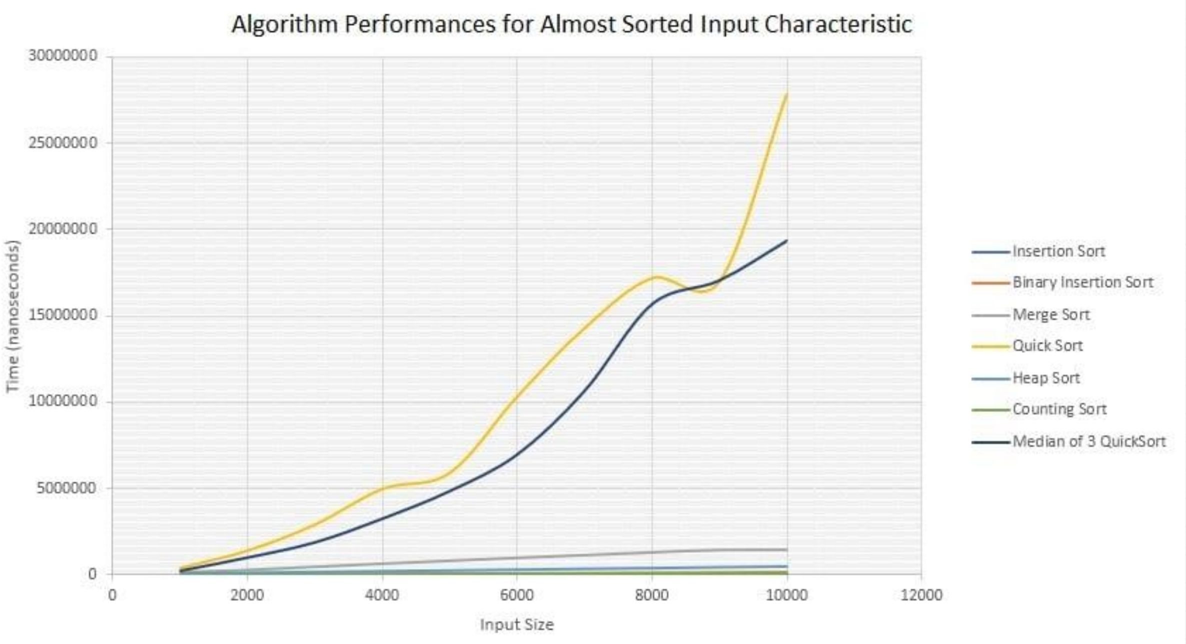
II. Reverse Sorted

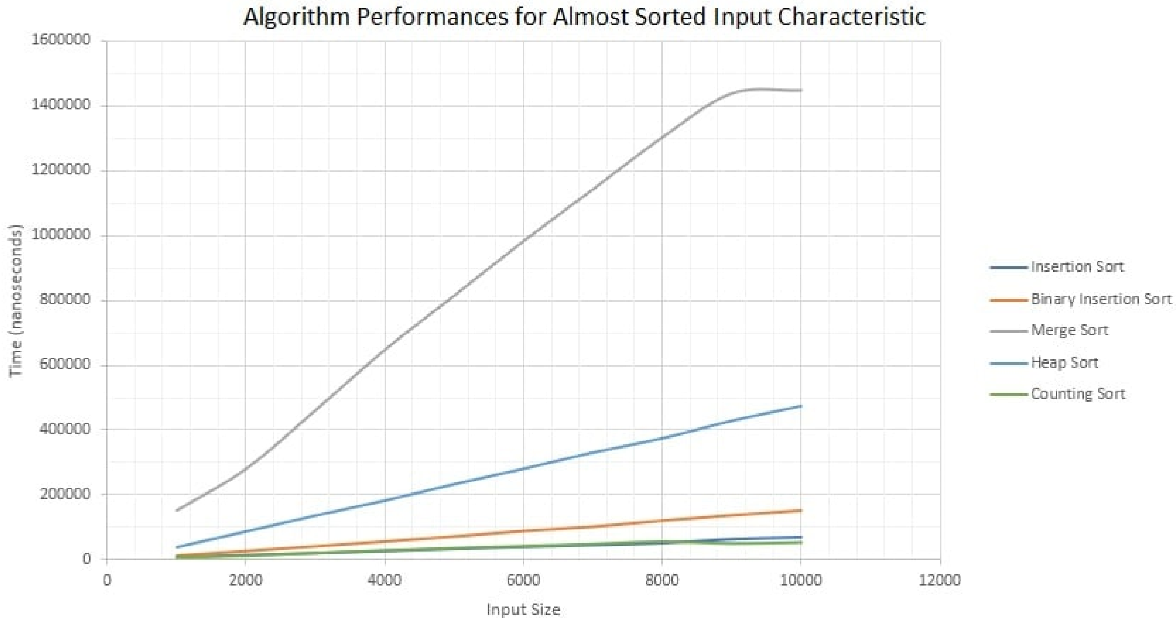


III. Duplicated

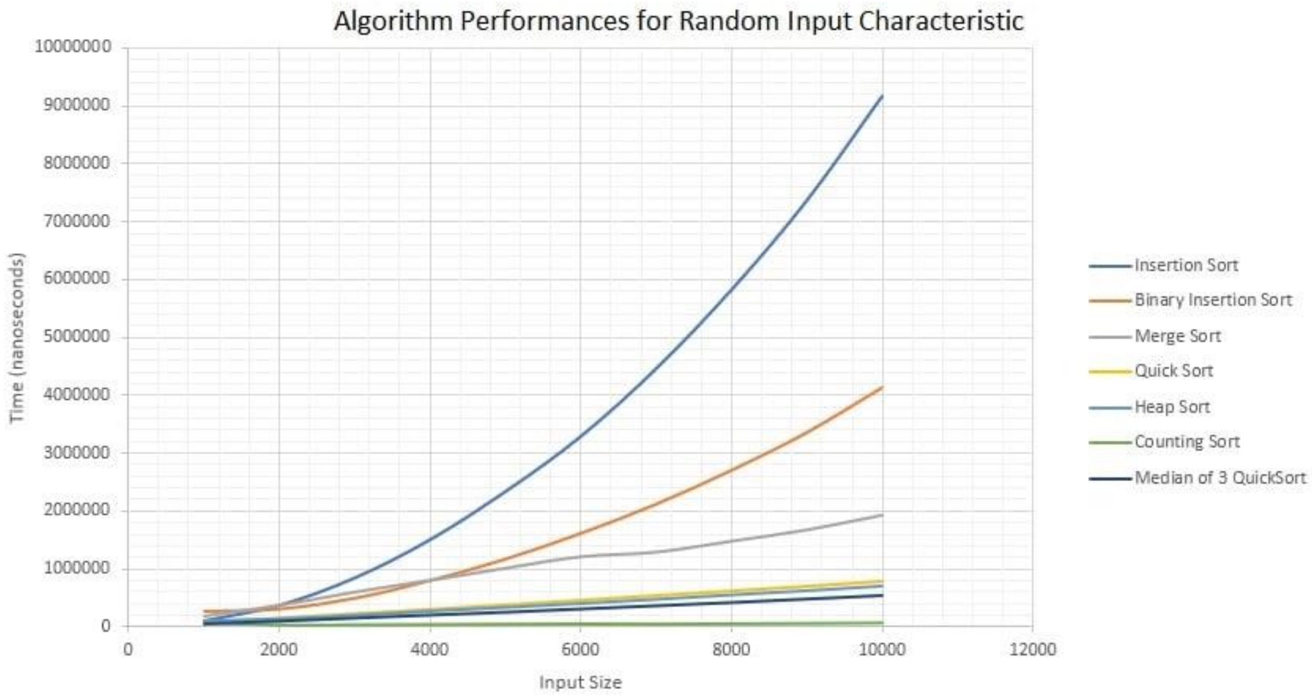


IV. Almost Sorted, zoomed out and zoomed in





V. Random {Average Case}



ANALYSIS OF EMPIRICAL DATA AND OBSERVATIONS

Before starting to analyze the empirical data, it would be beneficial to provide a theoretical table:

	Insertion Sort	Binary Insertion Sort	Merge Sort	Quick Sort	Quick Sort, Mo3	Heap Sort	Counting Sort
Best	n	n	n*logn	n*logn	n*logn	n*logn	n+r
Average	n^2	n*logn	n*logn	n*logn	n*logn	n*logn	n+r
Worst	n^2	n*logn	n*logn	n^2	n*logn	n*logn	n+r

I. Straight Insertion Sort / Binary Insertion Sort

As discussed in the lectures, insertion sort can be regarded as the best possible "basic" sorting algorithm, working in great speed especially for sorted/almost sorted inputs: O(n). Binary Insertion Sort, which uses the binary search to find the position of the key at the left array, can be seen as a noteworthy optimization for the worst case. This algorithm displayed its power and ability to speed up the process when the array was reverse sorted. For sorted/almost sorted inputs, Insertion Sort and Binary Insertion Sort were almost the same. Nevertheless, for reversely ordered input sets, BIS came out with a huge performance acceleration. Also, for the average case and the heavily duplicated input set, BIS managed to significantly expedite the process. All of these findings state these two things:

- Insertion Sort (both straight and binary versions) are the best "basic" sorting algorithms, performing very well for sorted/pre-sorted inputs.
- Binary Insertion Sort is a great way of optimization for this family of algorithms, accelerating the process dramatically for average and duplicated cases.

Both of these experimental findings are completely in harmony with the theoretical information we already had.

II. Quick Sort / Median of Three Quick Sort

Just like the insertion sort and the binary insertion sort, again we have a standard and an optimized version of the same sorting logic: the difference comes from the selection of the "better" pivot. For quicksort algorithms, the worst case happens when the elements are already ordered in a certain fashion: sorted or reverse-sorted. Theoretically, quick sort would reach O(n^2) for such a case, whereas median of three quicksort is still expected to yield a better performance [O(n*logn)].

Nevertheless, the power of this complex sorting algorithm should be clearly seen for a completely random/average input. At this stage, it is expected to see much better results than the insertion sort family, since the asymptotical upper bound is O(n*logn).

Again, our experiment verifies the theoretical knowledge. The quick sort family showed the worst performances (possibly due to their heavier computational complexity) for sorted and reverse sorted inputs. However, for random inputs, they were among the best ways to go: they were almost the same with merge sort and a little quicker than the heap sort.

There is last one aspect to report: in general, median of three quicksort always brought a good performance boost. However, for the heavily duplicated input, there was an exception. When all the inputs are almost the same, the algorithm lost some unnecessary energy for finding the median: Calculating the median of {1,1,1} resulted in such an outcome.

For final words, it can be claimed that quicksort is one of the best sorting algorithms for completely arbitrary and random input sets, whereas their weakness comes to the surface when the input is already ordered or duplicated. Also, the standard and the optimized versions differ for duplicated inputs.

III. Merge Sort

Just as expected (theoretically n*logn for all cases), merge sort always showed a great consistency throughout the experiment. Unlike quick sort, which can be wonderfully fast or unluckily slow, merge sort was the most reliable algorithm considering all different kinds of input sets. However, as it is discussed in the lectures, this successful algorithm has a disadvantage: space complexity. Merge Sort is not an in-place algorithm: even though it is able to provide a rapid time complexity, for gigantic input sets, the machine can suffer from all those created sub-arrays. Therefore, this aspect of merge sort should be known by all engineers, in order to not be deceived solely by its runtime speed.

IV. Heapsort

Heapsort, just like the Merge sort, provided a consistent and speedy outcome throughout the experiment (n*logn for all cases). Even more, it has an advantage over merge sort: heapsort is an in-place algorithm. All these combined makes it one of the best options to go with. Nevertheless, it should be noted that, heapsort was the slowest algorithm among the “fastest”, in terms of runtime performance. Therefore, although heapsort is a strong choice independent of the conditions, for huge input sets, it can be claimed that if the set is completely random, quicksort might be a slightly faster choice.

V. Counting Sort

Even though counting sort seemed as if it is the best option to go with O(n+r) complexity and a reliable speed throughout the experiment, it cannot be considered as a “real” sorting algorithm. Due to its nature (very similar to the hashing logic), it can only be used for sorting integer sets within a certain range. This behavior heavily limits its usage. Additionally, counting sort is not meaningful if the range is greater than the actual number of inputs.

Since the count array is calculated with index arithmetic, counting sort is not suitable to be used with negative numbers.

However, for small integer inputs, it could make a fast problem solver.

CONCLUSION

In this experiment, different sorting algorithms were scrutinized according to different characteristics. Later, numerical findings are compared and interpreted, by applying cross-checking with the theoretical knowledge. After obtaining and testing the best, worst and average cases for all algorithms, special conditions like duplicated inputs were also taken into consideration.

The results obtained from this experiment successfully verified the theoretical understanding.