Analysis of Sorting Algorithms

-BISHNU TIWARI

-GX-09

- EXAM ROLL:- 510518009
- PH.NO-9079939134
- EMAIL-BISHNUTIWARI963@GMAIL.COM
- WHATSAPP-9593048305

PREREQUISITES FOR THE ANALYSIS:

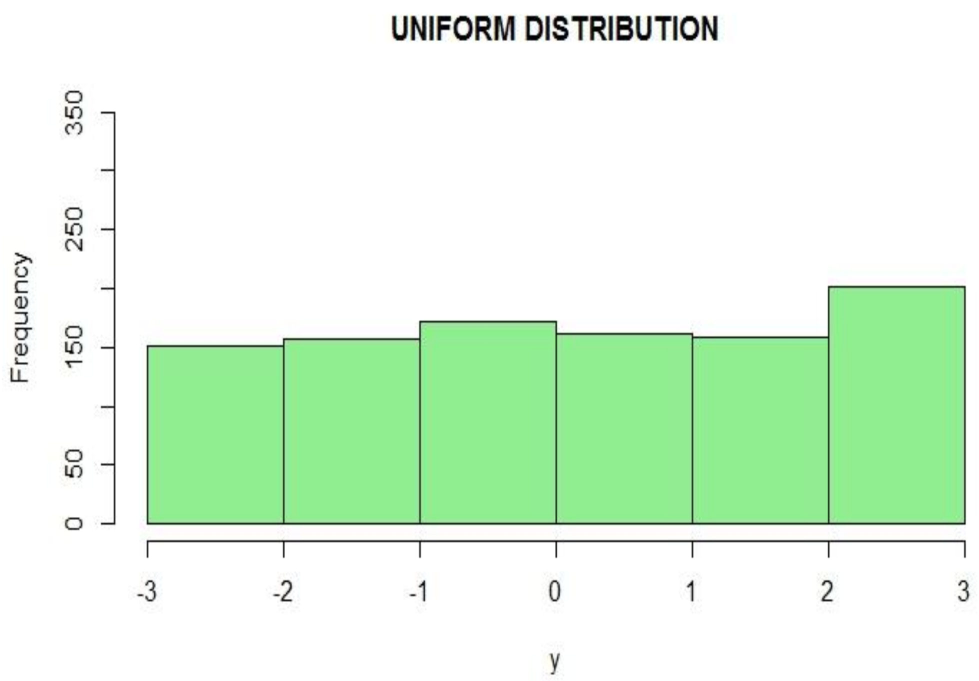
In order to study and analyze sorting algorithms, we need to apply them to sort different types of data in varying sizes.

In this assignment, we supply the algorithms with datasets containing elements derived from two kinds of distributions: Uniform Distribution and Normal distribution. The datasets contain elements derived from these distributions in a random sequence. Also, the number of data elements in these datasets increases to study the working of the algorithm with increase in input size.

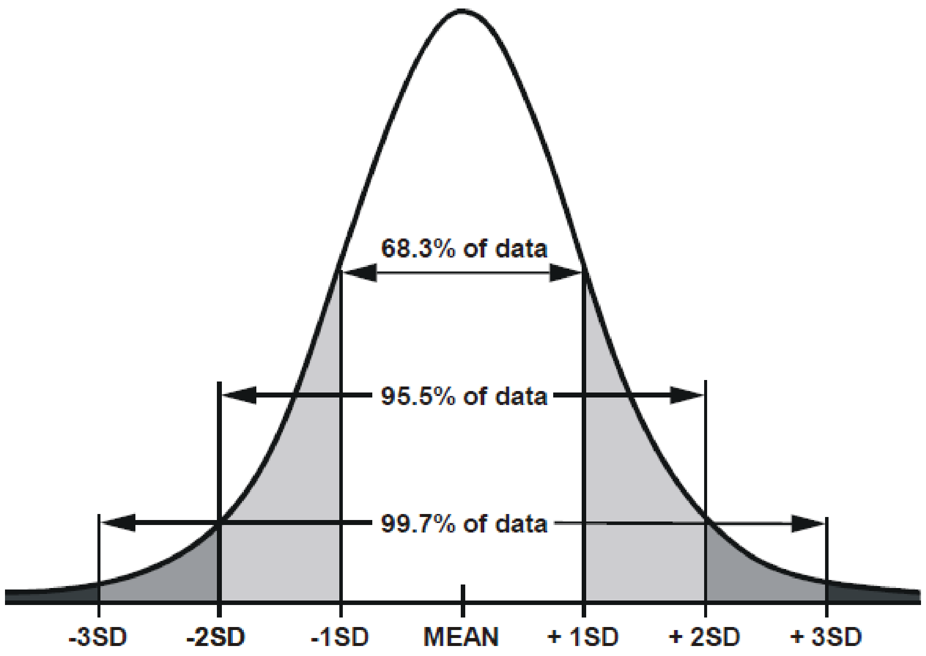
The datasets are also shuffled among themselves and the data is tested to improve the statistical basis of the analysis, as this decreases any bias that may arise due to a particular ordering of input dataset.

GRAPHICAL REPRESENTATION:

UNIFORM DISTRIBUTION



NORMAL DISTRIBUTION

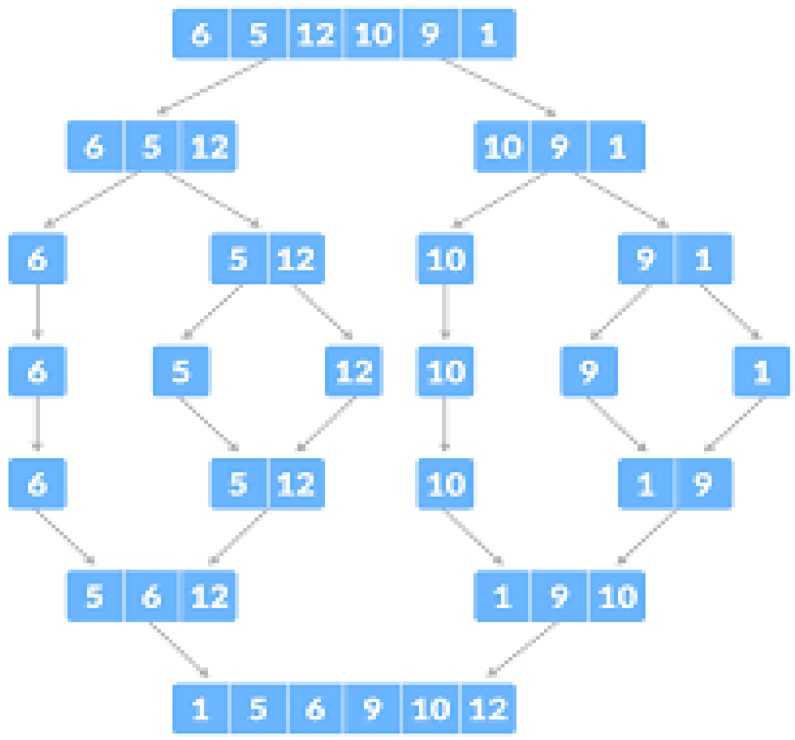


Merge Sort

THE ALGORITHM

- The basic idea behind merge sort is the divide and conquer approach
- The input data is split into two sub-datasets and these two are separately sorted
- The two sorted datasets are then merged into a single dataset in such a way that the resulting dataset is in sorted order
- The process is implemented by recursively calling the mergesort function, and after the recursive calls, a merge function is called to merge the sorted datasets
- The complexity of this algorithm is O(n lg(n))

Diagrammatic representation of the working of this algorithm:



Experimental Results:

APPLIED ON THE DISTRIBUTIONS:

No. of elements	Uniform distribution			Normal distribution		
	Comparisons (cmp)	Moves	CPU time	Comparisons (cmp)	Moves	CPU time
8	16.2	24	0.0000016	16.4	24	0.0000018
32	122.6	160	0.0000048	119.2	160	0.0000062
64	303.6	384	0.0000086	305.8	384	0.000012
128	728.6	896	0.000018	736	896	0.00002
512	3967	4608	0.0000802	3957.2	4608	0.0000774
2048	19943.4	22528	0.0003662	19951	22528	0.0003516
8192	96138.2	106496	0.0017068	96146.6	106496	0.0016548
16384	208695.6	229376	0.0035724	208629.2	229376	0.0036834
32768	450081.2	491520	0.0074592	450162	491520	0.0076212
65536	965708.2	1048576	0.0165738	965805	1048576	0.0159988
262144	4386919.8	4718592	0.0710362	4387181	4718592	0.0713818
524288	9298481	9961472	0.1486212	9298468.6	9961472	0.147747

Data obtained after averaging out the values obtained after shuffling dataset 5 times

Tabulating shuffled sets of data for Uniform Distribution:

Number of elements	Shuffle no.	n lg n (n1)	Comparisons(cmp)	cmp/n1	Moves	CPU time
32	1	160	122	0.7625	160	0.000006
32	2	160	122	0.7625	160	0.000004
32	3	160	115	0.71875	160	0.000005
32	4	160	126	0.7875	160	0.000005
32	5	160	128	0.8	160	0.000004
2048	1	22528	19903	0.883478	22528	0.00038
2048	2	22528	19944	0.885298	22528	0.00037
2048	3	22528	19971	0.886497	22528	0.000361
2048	4	22528	19960	0.886009	22528	0.000361
2048	5	22528	19939	0.885076	22528	0.000359

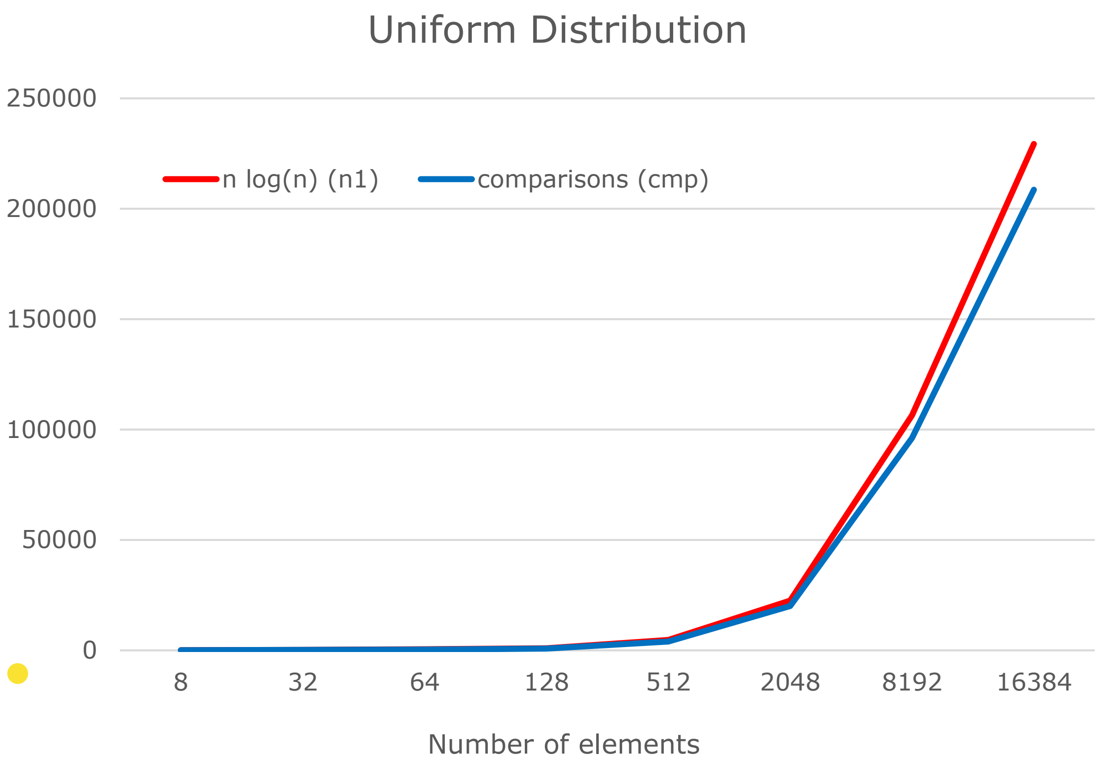
Tabulating shuffled sets of data for Normal Distribution:

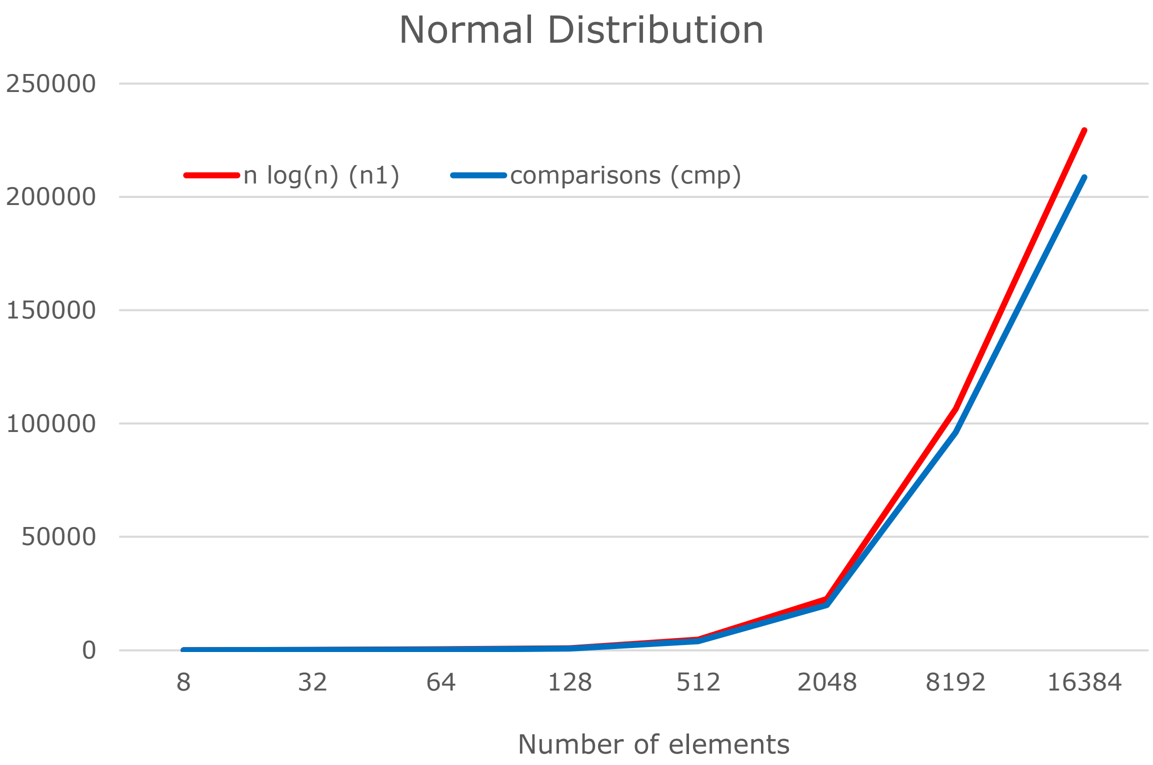
Number of elements	Shuffle no.	n lg n	Comparisons	cmp/n1	moves	CPU time
32	1	160	117	0.73125	160	0.000008
32	2	160	120	0.75	160	0.000006
32	3	160	121	0.75625	160	0.000005
32	4	160	118	0.7375	160	0.000006
32	5	160	120	0.75	160	0.000006
2048	1	22528	19980	0.886896	22528	0.000361
2048	2	22528	19963	0.886142	22528	0.000352
2048	3	22528	19958	0.88592	22528	0.00035
2048	4	22528	19938	0.885032	22528	0.000348
2048	5	22528	19916	0.884055	22528	0.000347

Thus we see that shuffling the dataset has no major impact on the operation of merge sort for both the uniform and normal distribution.

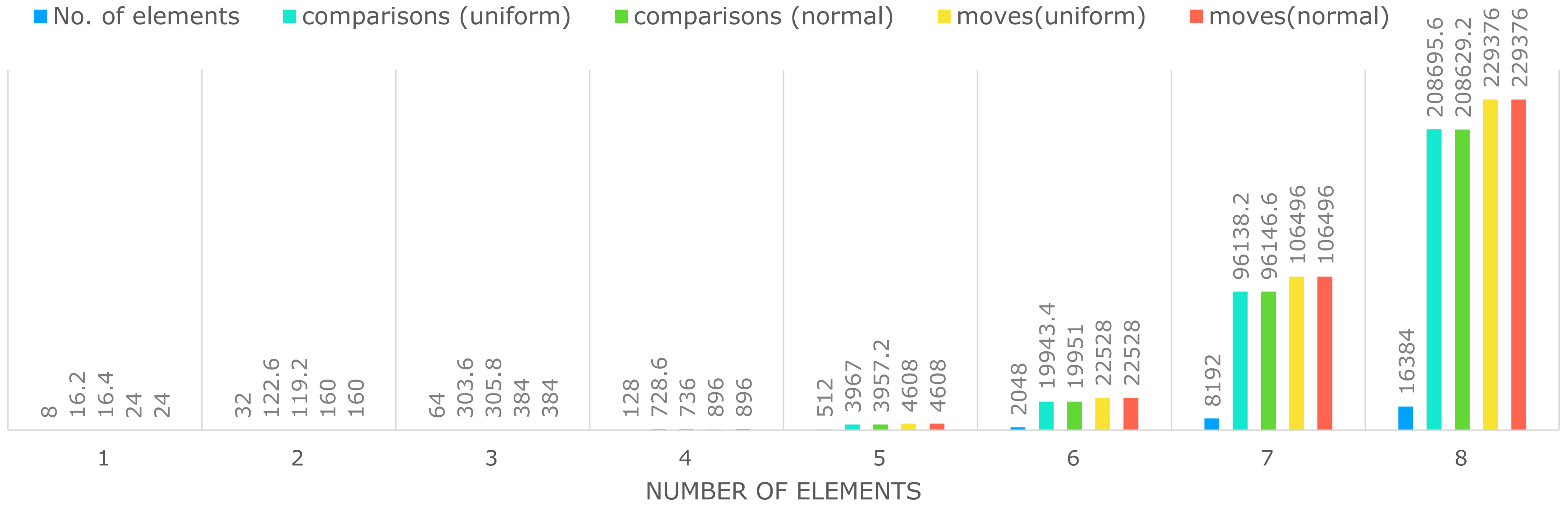
COMPLEXITY ANALYSIS:

Number of comparisons compared to n lg n:



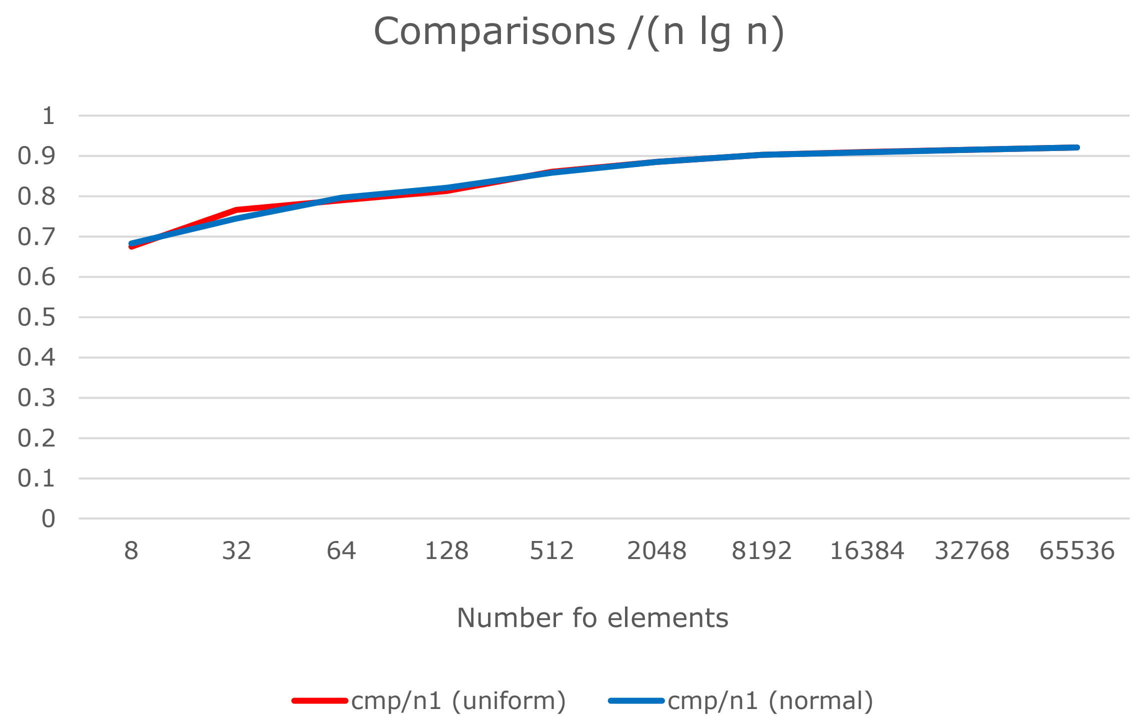


PERFORMANCE ON UNIFORM VS. NORMAL DISTRIBUTION



- Here, we can see how the algorithm has similar complexity for datasets belonging to the two distributions.

The number of comparisons is very close for both normal and uniform distribution, and the number of moves comes out to be identical.



Calculating constant factor

As we have obtained a seemingly constant value for the (number of comparisons)/(n lg n), we can safely conclude that the algorithm has a complexity of O(n lg n).

From our analysis so far, we can conclude that the merge sort algorithm is of the order of O(n lg n). The complexity of the algorithm does not seem to depend upon the type of distribution, as we obtain similar results for both uniform and normal distribution. We can see in the plots how the complexity of the algorithm increases as the input size increases.

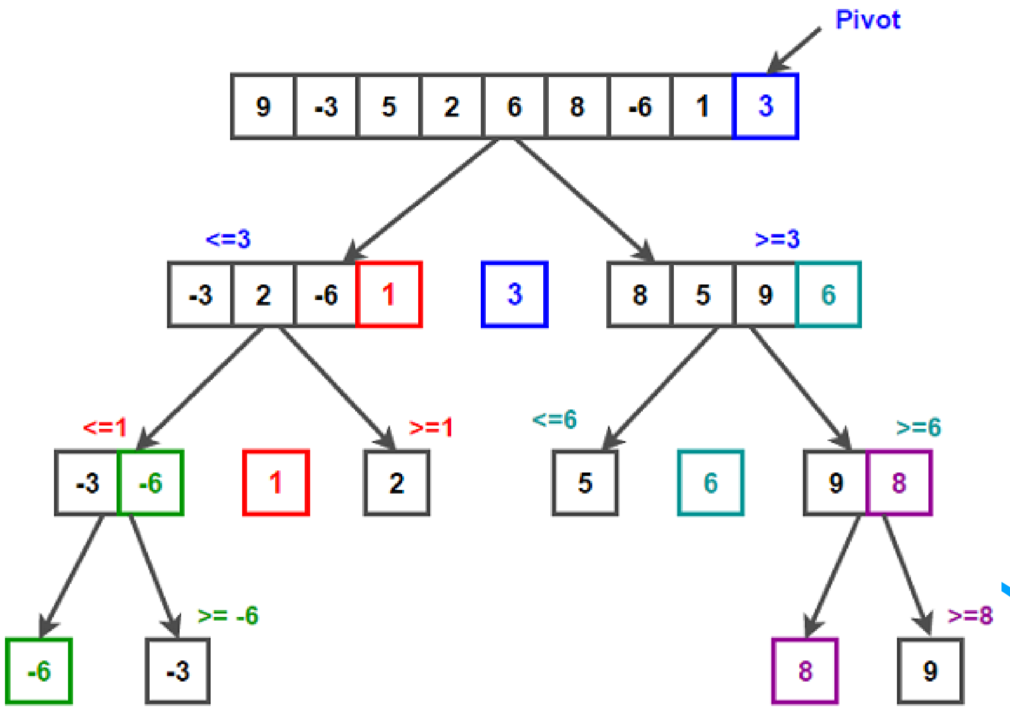
This completes our analysis of merge sort.

Quick Sort

THE ALGORITHM

- Quick sort is also an algorithm based on the divide and conquer method
- The input dataset is split into two sub-datasets based on a pivot element. Generally, the first or last element of the dataset is chosen as pivot
- The data is partitioned such that the elements to the left of the pivot are less than the pivot, and the elements to the right are greater, in no particular order
- The process is implemented by recursively calling the quicksort function for each of these partitions, and each recursive call invokes the partition function
- Repeating this procedure ensures the array is sorted, as after each partition, the pivot is placed at its appropriate position
- The best case and average case complexity of this procedure is O(n lg(n)), but the worst case complexity comes to $O(n^2)$

Diagrammatic representation of working of the algorithm:



Experimental Results:

APPLIED ON THE DISTRIBUTIONS:

No. of elements	Uniform distribution			Normal distribution		
	Comparisons (cmp)	Swaps	CPU time	Comparisons (cmp)	Swaps	CPU time
8	15.8	11.8	1.2E-06	17.6	13.6	0.0000014
32	139.2	83.4	3.6E-06	141.8	87.4	0.0000032
64	357.6	218.6	6.2E-06	338.6	215.2	0.0000056
128	878	488.4	1.3E-05	862	510.6	0.0000124
512	4812.4	2722.6	5.7E-05	4736.6	2626	0.0000518
2048	24747.6	13593.8	0.00025	25782.2	13931.4	0.0002628
8192	120590.8	67714.6	0.00125	126081	64964.2	0.001168
16384	270008.2	139953	0.00255	280639.2	146742.2	0.0025946
32768	584432.4	316985	0.00556	597095.8	323366.2	0.0055532
65536	1247749.2	669652	0.01248	1258355.2	682102.2	0.011585
262144	5749864.2	3040238	0.05272	5897053.6	3028571.2	0.0520454
524288	12508246.2	6674689	0.11303	12149271.2	6386450	0.1079722

Data obtained after averaging out the values obtained after shuffling dataset 5 times

EFFECT OF SHUFFLING: UNIFORM DISTRIBUTION

Number of elements	Shuffle number	n lg n (n1)	n^2 (n2)	Comparisons (cmp)	cmp/n1	cmp/n2	Swaps	CPU time
128	1	896	16384	896	1	0.054688	539	0.000014
128	2	896	16384	824	0.919643	0.050293	472	0.000012
128	3	896	16384	955	1.065848	0.058289	461	0.000013
128	4	896	16384	812	0.90625	0.049561	414	0.000012
128	5	896	16384	903	1.007812	0.055115	556	0.000014
16384	1	229376	268435456	270529	1.179413	0.001008	149246	0.002536
16384	2	229376	268435456	260055	1.13375	0.000969	138549	0.002445
16384	3	229376	268435456	261188	1.138689	0.000973	128370	0.002502
16384	4	229376	268435456	266932	1.163731	0.000994	145037	0.002575
16384	5	229376	268435456	291337	1.270129	0.001085	138562	0.002673

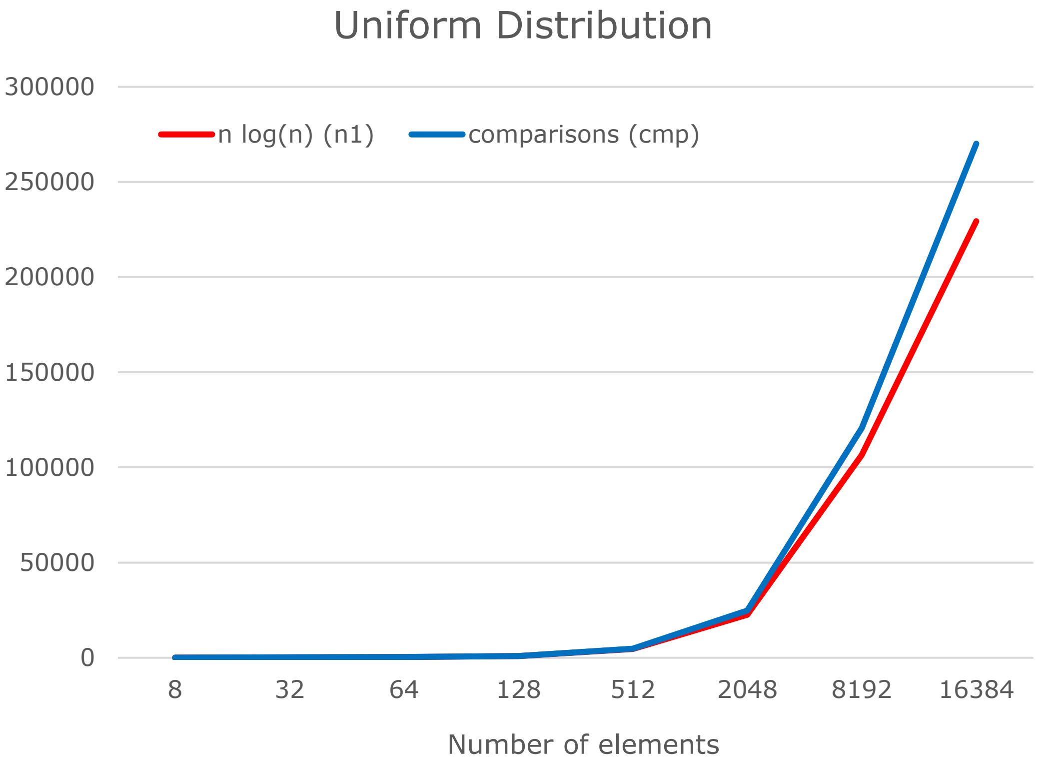
EFFECT OF SHUFFLING: NORMAL DISTRIBUTION

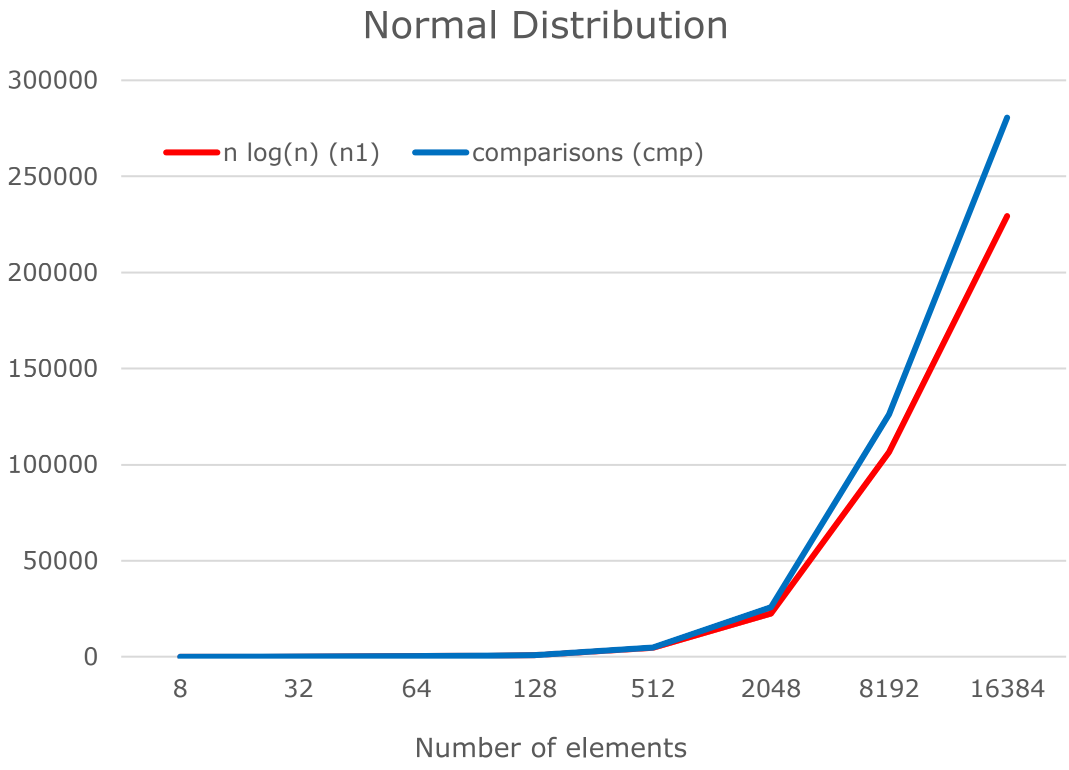
Number of elements	Shuffle number	n lg n (n1)	n^2 (n2)	Comparisons (cmp)	cmp/n1	cmp/n2	Swaps	CPU time
128	1	896	16384	895	0.998884	0.054626	505	0.000013
128	2	896	16384	800	0.892857	0.048828	437	0.000011
128	3	896	16384	806	0.899554	0.049194	518	0.000011
128	4	896	16384	902	1.006696	0.055054	465	0.000014
128	5	896	16384	907	1.012277	0.055359	628	0.000013
16384	1	229376	268435456	258576	1.127302	0.000963	141770	0.002542
16384	2	229376	268435456	295035	1.286251	0.001099	135114	0.002527
16384	3	229376	268435456	312562	1.362662	0.001164	169552	0.002665
16384	4	229376	268435456	270483	1.179212	0.001008	135391	0.002685
16384	5	229376	268435456	266540	1.162022	0.000993	151884	0.002554

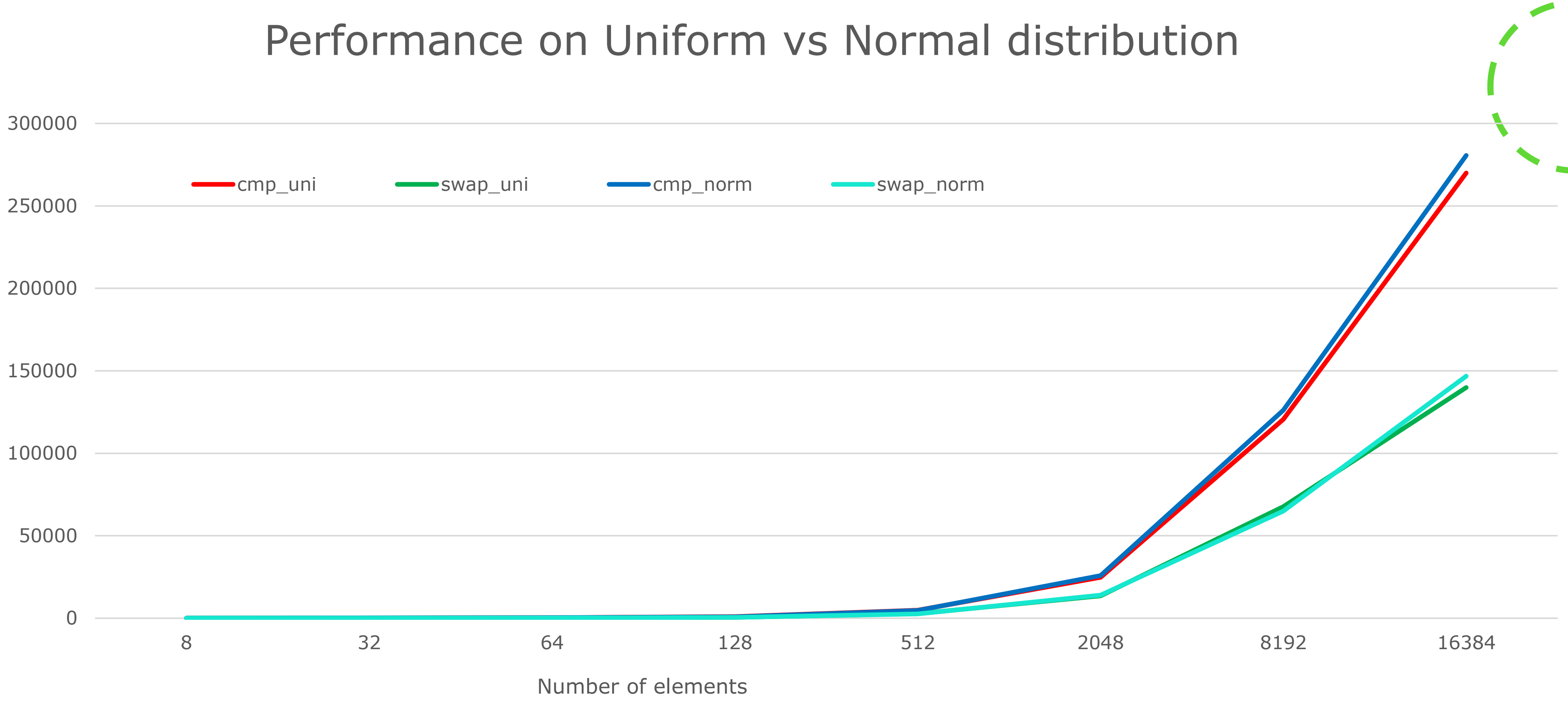
Here we can see shuffling has a major impact on the number of comparisons made, for both the datasets, thus the algorithm depends on the order of elements in input dataset

COMPLEXITY ANALYSIS:

Number of comparisons compared to n lg n:

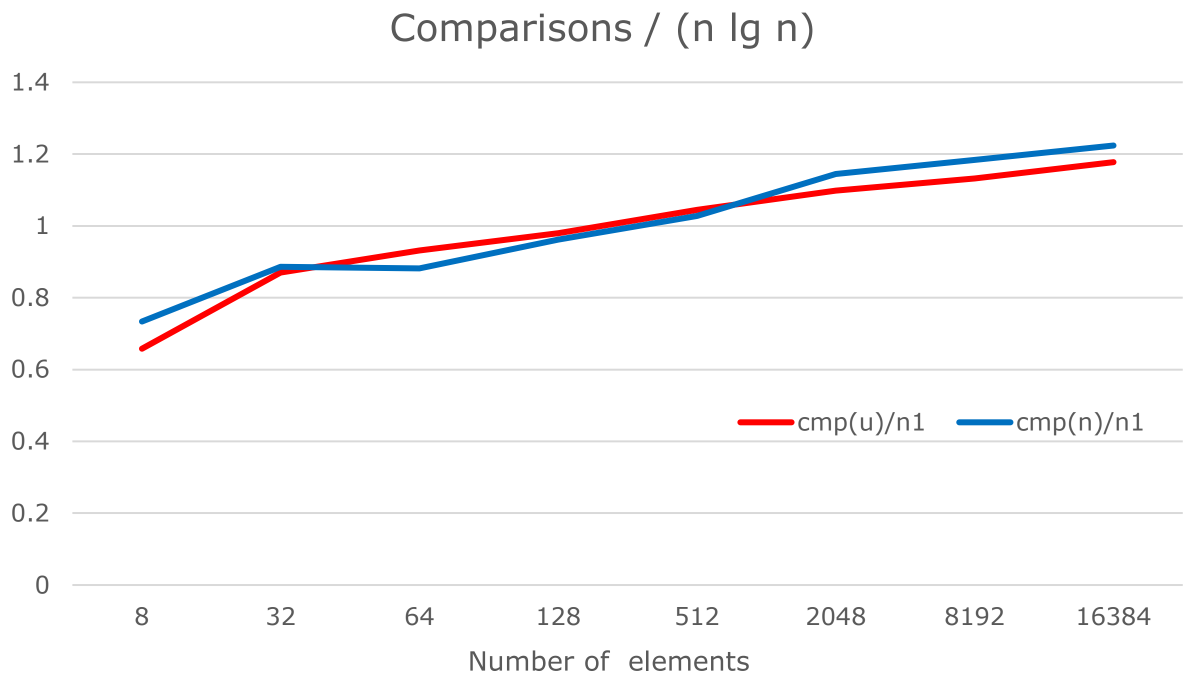






- Here, we can see how the algorithm has similar complexity for datasets belonging to the two distributions, though it seems to perform slightly better for the uniform distribution.

This is because as the size of the dataset increases, the partitions become increasingly worse for the normal distribution



Calculating constant factor

The value for number of comparisons/(n lg n) doesn't seem to attain a constant value, thus depicting that the complexity is not bound by O(n lg n), but is slightly greater

From our analysis so far, we can conclude that the quick sort algorithm performs close to a complexity of O(n lg n). The complexity of the algorithm seems to depend slightly upon the type of distribution, as we obtain minutely better results for uniform distribution as compared to normal distribution. We can see in the plots how the complexity of the algorithm increases as the input size increases.

This completes our analysis of quick sort.

Randomized Quick Sort

THE ALGORITHM

- The Randomized Quick sort algorithm is the same algorithm as the quick sort algorithm, with one difference, the pivot is chosen at random and can be any element of the dataset
- The random element chosen is swapped with the first or last element, and then that is chosen as the pivot, and this is done for every partition
- This randomization gives us a statistical advantage, as it makes the algorithm free of any predefined order of the dataset before input, and increases our chances of obtaining a good partition
- To put it simply, even if a sorted dataset is provided, the algorithm does not run in worst case complexity. The algorithm encounters worst case running time only when the random element chosen is at every point the worst possible choice, thus ensuring better partitions
- The best case and average case complexity of this procedure is $O(n \lg(n))$, but the worst case complexity comes to $O(n^2)$

Experimental Results:

APPLIED ON THE DISTRIBUTIONS:

No. of elements	Uniform distribution			Normal distribution		
	Comparisons (cmp)	Swaps	CPU time	Comparisons (cmp)	Swaps	CPU time
8	18.6	0.775	0.290625	17.2	0.7166666	0.26875
32	139.6	0.8725	0.1363282	137.8	0.86125	0.1345704
64	368.2	0.958854	0.0898928	384.6	1.0015624	0.0938964
128	933.8	1.0421876	0.0569944	897.6	1.0017858	0.054785
512	4876.4	1.0582464	0.0186022	5072.8	1.100868	0.0193512
2048	26009.8	1.1545542	0.0062014	25888.6	1.1491744	0.0061724
8192	121772	1.1434422	0.0018146	129339.4	1.2145	0.0019272
16384	271638.4	1.1842494	0.0010118	267081.2	1.1643814	0.0009952
32768	579911.2	1.1798322	0.0005402	593258.8	1.2069882	0.0005528
65536	1253610.2	1.195536	0.0002918	1260071	1.2016974	0.0002934
262144	5828405	1.2352	0.0000846	5729013.4	1.214136	0.0000834
524288	12298133	1.2345698	0.000045	12209453.2	1.2256674	0.0000444

Data obtained after averaging out the values obtained after shuffling dataset 5 times

EFFECT OF SHUFFLING: UNIFORM DISTRIBUTION

Shuffle number	Number of elements	n lg n (n1)	n^2 (n2)	Comparisons (cmp)	cmp/n1	cmp/n2	Swaps	CPU time
1	128	896	16384	857	0.956473	0.052307	651	0.000015
2	128	896	16384	919	1.02567	0.056091	557	0.000013
3	128	896	16384	861	0.960938	0.052551	633	0.000013
4	128	896	16384	1150	1.283482	0.07019	784	0.00002
5	128	896	16384	882	0.984375	0.053833	679	0.000016
1	16384	229376	268435456	287272	1.252407	0.00107	179325	0.003016
2	16384	229376	268435456	273046	1.190386	0.001017	154119	0.003098
3	16384	229376	268435456	264897	1.154859	0.000987	160815	0.002911
4	16384	229376	268435456	270371	1.178724	0.001007	141423	0.002779
5	16384	229376	268435456	262606	1.144871	0.000978	155612	0.002947

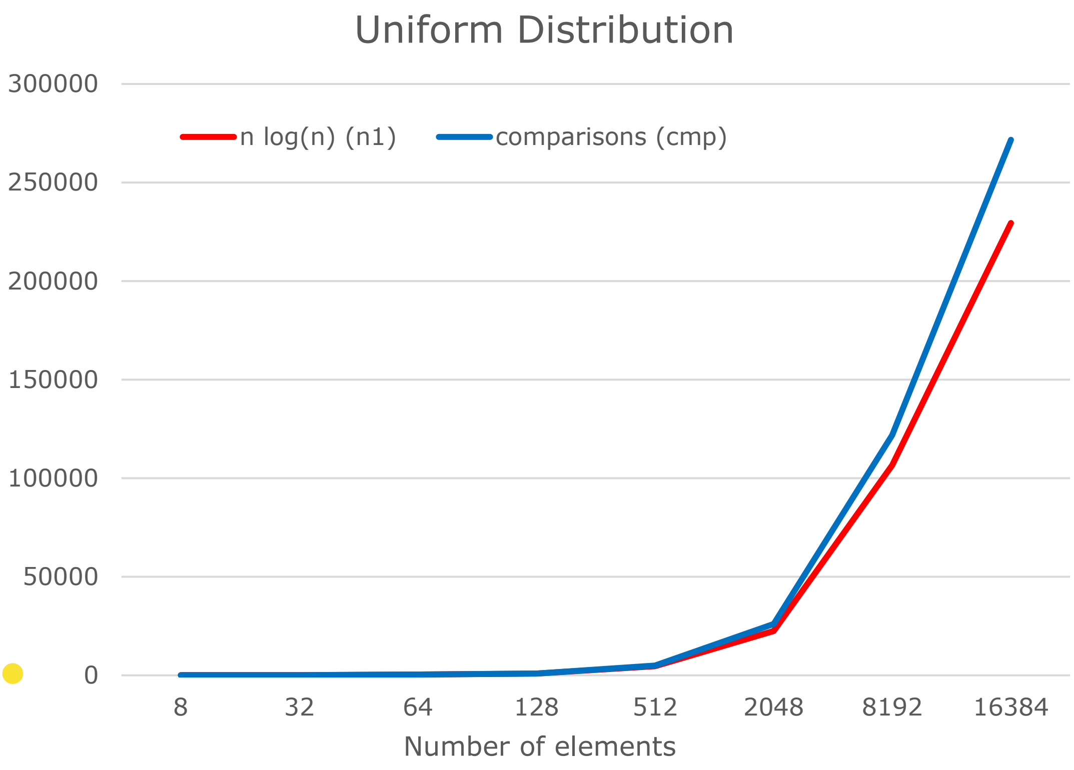
EFFECT OF SHUFFLING: NORMAL DISTRIBUTION

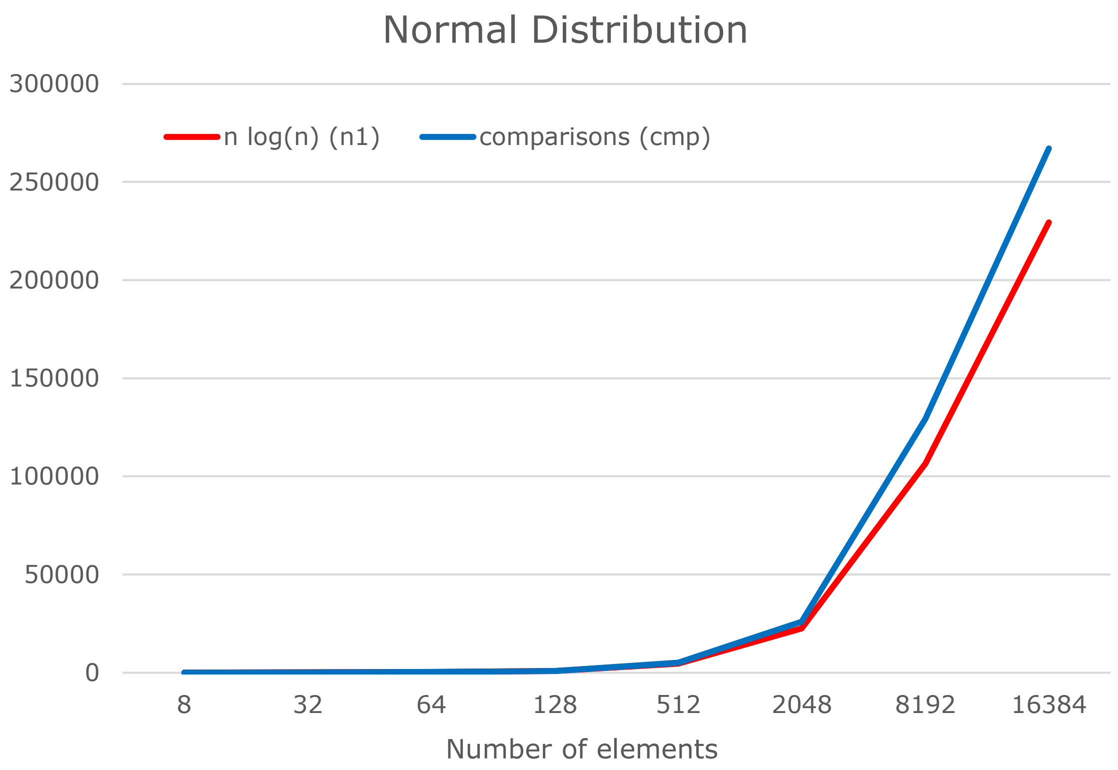
Shuffle number	Number of elements	n lg n (n1)	n^2 (n2)	Comparisons (cmp)	cmp/n1	cmp/n2	Swaps	CPU time
1	128	896	16384	839	0.936384	0.051208	622	0.000015
2	128	896	16384	852	0.950893	0.052002	577	0.000016
3	128	896	16384	1077	1.202009	0.065735	710	0.000029
4	128	896	16384	837	0.934152	0.051086	531	0.000017
5	128	896	16384	883	0.985491	0.053894	561	0.000018
1	16384	229376	268435456	261594	1.140459	0.000975	161213	0.002874
2	16384	229376	268435456	268854	1.17211	0.001002	146841	0.002838
3	16384	229376	268435456	269346	1.174255	0.001003	156854	0.002825
4	16384	229376	268435456	275865	1.202676	0.001028	160362	0.002888
5	16384	229376	268435456	259747	1.132407	0.000968	155127	0.002918

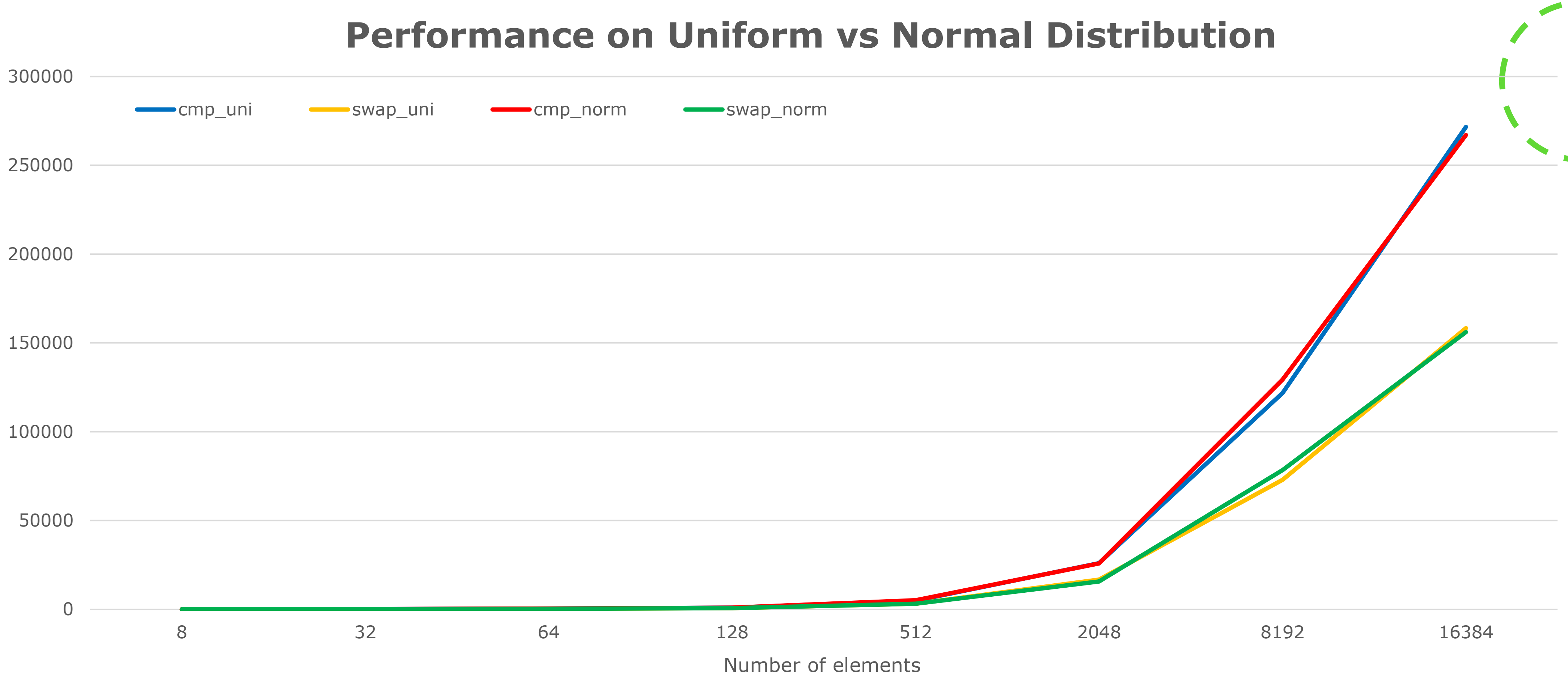
Here we can see shuffling has a significant impact on the number of comparisons made, for both the datasets, thus the algorithm depends on the order of elements in input dataset

COMPLEXITY ANALYSIS:

Number of comparisons compared to n lg n:

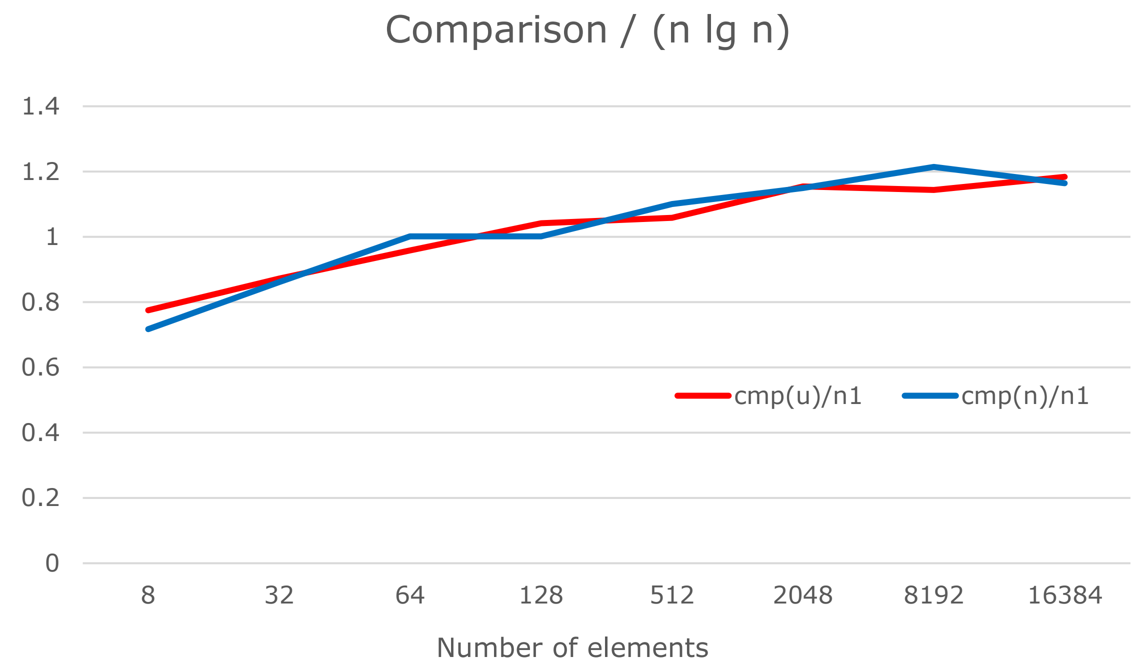






- Here, we can see how the algorithm has similar complexity for datasets belonging to the two distributions, and there is no clear distinction in performance on the two types of datasets

This is due to the randomness in choosing the pivot, which prevents the performance of the algorithm from being impacted by any ordering of input data



Calculating constant factor

The value for number of comparisons/(n lg n) doesn’t seem to attain a constant value, and seems to be highly fluctuating, thus depicting that the complexity is not bound by O(n lg n), but is slightly greater

From our analysis so far, we can conclude that the randomization of quick sort algorithm performs close to a complexity of O(n lg n). The complexity of the algorithm is independent of the type of distribution and also of any preordering of the input dataset. We can see in the plots how the complexity of the algorithm increases as the input size increases.

The randomization of pivot selection improves performance a little and gives us a statistical advantage.

This completes our analysis of quick sort with randomized pivot.

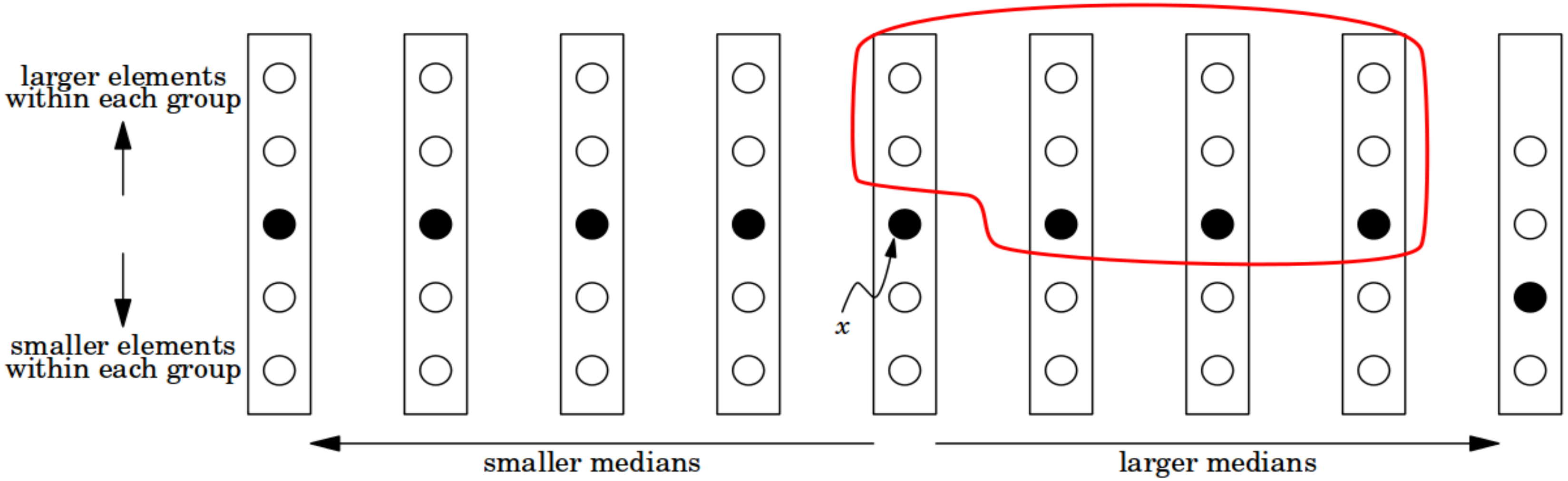
Median of Medians pivot Quick Sort

ORDER STATISTICS

- The kth order statistic of a dataset is its kth smallest value, thus the median will be the n/2th statistic of a dataset
- Determining the order statistics of a dataset can be found in two ways : Randomised select and Worst case linear time
- Randomised select randomly picks a pivot and keeps partitioning the dataset till it arrives at the required rank
- The worst case linear time procedure uses the median of medians of the dataset to arrive at the desired rank
- Both these approaches make use of the property that whenever an array is partitioned based on a pivot, the pivot is placed at its appropriate rank. Thus, checking the pivot ranks will get us to the desired order statistic
- The average case complexity of Random select is linear, but the complexity $\Theta(n)$ but its worst case is complexity may approach $\Theta(n^2)$
- Worst case linear time algorithm always gives a expected running time of O(n)

THE ALGORITHM

- The Median of Median Quick sort algorithm is the same algorithm as the quick sort algorithm, with one difference, the pivot chosen at every point is the median off medians of the dataset partition
- The median of medinas is swapped with the first or last element, and then that is chosen as the pivot, and this is done for every partition
- The median of medians is chosen using the worst case linear time selection procedure
- This selection of pivot gives us a statistical advantage, as it makes the algorithm free of any predefined order of the dataset before input, and increases our chances of obtaining a good partition
- To put it simply, even if a sorted dataset is provided, the algorithm does not run in worst case complexity, as it is always partitioned around its median



Experimental Results:

APPLIED ON THE DISTRIBUTIONS:

No. of elements	Uniform distribution			Normal distribution		
	Comparisons (cmp)	Swaps	CPU time	Comparisons (cmp)	Swaps	CPU time
8	15.4	20.8	0.000004	19	20.2	0.000005
32	114	111	0.000027	154	131.6	2.98E-05
64	289.6	252.4	0.0000686	351.4	249	7.28E-05
128	729.4	605.2	0.0001458	941.2	633.4	0.000191
512	3860	2871	0.0007468	4896.6	3204.8	0.000831
2048	19354.6	13142.6	0.0034886	25630.2	15089.6	0.004469
8192	94085.4	61815.6	0.016363	122705.8	71232.4	0.021043
16384	205390.6	133187	0.0353832	277066.8	159223.8	0.046662
32768	443018.8	281478.4	0.076357	588284.2	323394.8	0.098701
65536	950016.2	594490.8	0.1615446	1257676.8	715835	0.202826
262144	4324677	2608484.8	0.7105656	5624400.2	3170886	0.91181
524288	9184459.8	5541400.2	1.5041128	12277901	6838036	1.985491

Data obtained after averaging out the values obtained after shuffling dataset 5 times
The dataset is divided into groups of 3 to find median of medians

APPLIED ON THE DISTRIBUTIONS:

No. of elements	Uniform distribution			Normal distribution		
	Comparisons (cmp)	Swaps	CPU time	Comparisons (cmp)	Swaps	CPU time
8	17.4	24.2	0.0000042	18.2	17.8	4.4E-06
32	117.4	114	0.000024	135	101.8	2.36E-05
64	290.8	257.2	0.000051	347.4	243.6	5.48E-05
128	716.4	610.4	0.0001226	890	585.4	0.000125
512	3860.6	2918.2	0.0005512	4737.4	3107.8	0.000597
2048	19627	13755.2	0.0024218	24632.2	14721.8	0.003004
8192	94814.6	63773.4	0.0115662	128897.6	72876	0.015535
16384	206653.2	137194	0.0250092	269720.2	156540	0.032579
32768	447000	291082.4	0.0535456	578904.2	342942.8	0.068316
65536	957788.8	613284.6	0.11415	1240781.8	683616.8	0.141414
262144	4343514.2	2664520.8	0.518234	5649243.2	3109349	0.645713
524288	9238904	5678089.8	1.0569036	12064749.8	6771035	1.342637

Data obtained after averaging out the values obtained after shuffling dataset 5 times
The dataset is divided into groups of 5 to find median of medians

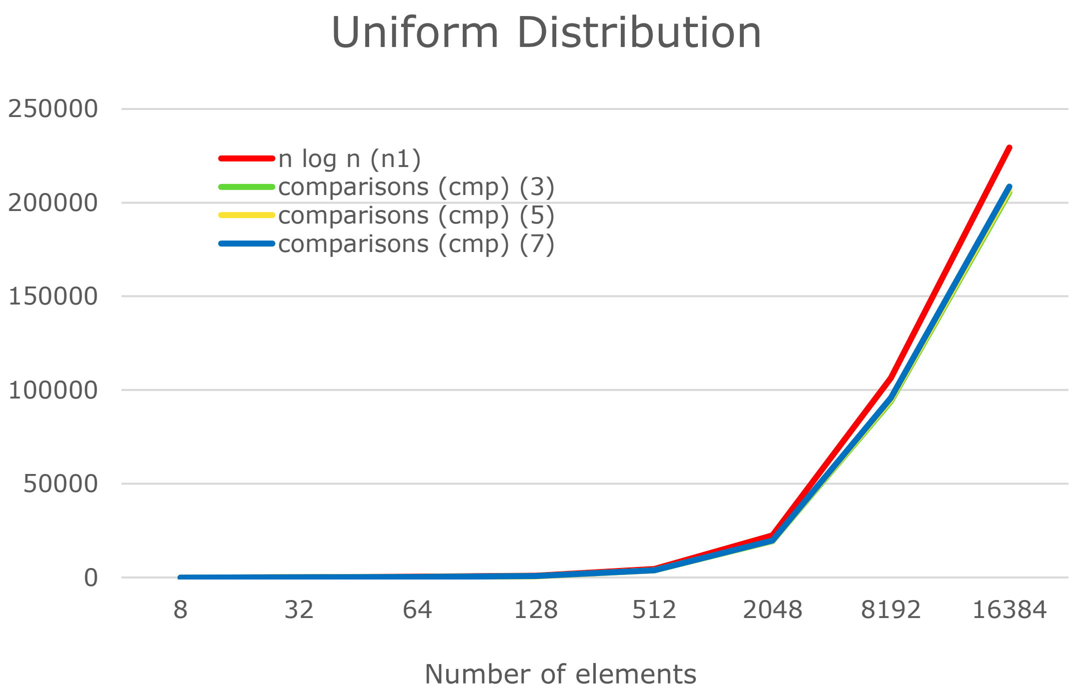
APPLIED ON THE DISTRIBUTIONS:

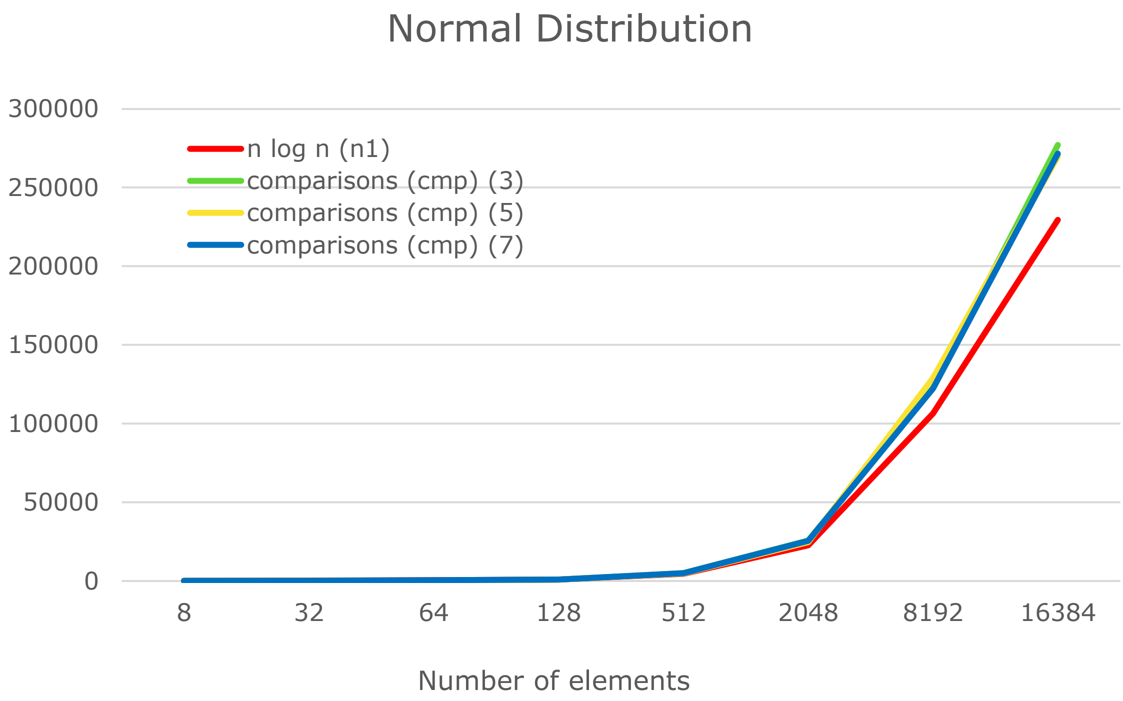
No. of elements	Uniform distribution			Normal distribution		
	Comparisons (cmp)	Swaps	CPU time	Comparisons (cmp)	Swaps	CPU time
8	18.4	28	0.000004	15	17.6	3.6E-06
32	115.2	115.2	0.0000208	142.2	108.6	2.08E-05
64	304.6	284.2	0.0000468	379.6	253.4	4.92E-05
128	730.8	639.4	0.000102	862.8	627	0.000121
512	3951.4	3091.2	0.0005024	5034	3184.8	0.000589
2048	19724.2	14046.2	0.0022356	25516.6	15947.8	0.002948
8192	95876	65843.8	0.0109744	122312	68758.2	0.013663
16384	208524.6	140976.6	0.0233328	271446.2	148480	0.030223
32768	449616	297381.6	0.0501508	605049	338606.4	0.066174
65536	964500.8	626901	0.105659	1267514.2	711557.6	0.135888
262144	4366302.2	2713601.4	0.4851888	5623189	3115322	0.600219
524288	9287007.6	5784325	0.9916212	11929130.4	6593882	1.234435

Data obtained after averaging out the values obtained after shuffling dataset 5 times
The dataset is divided into groups of 7 to find median of medians

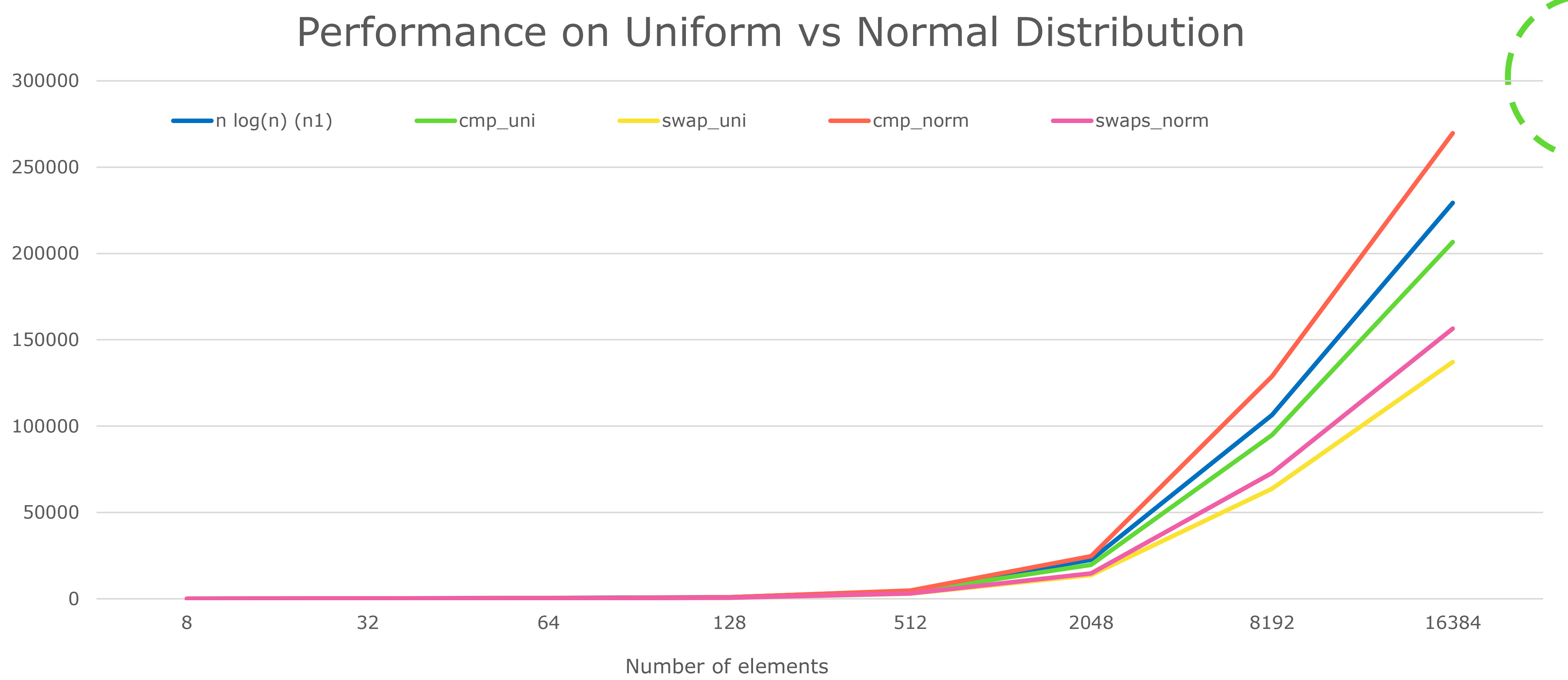
COMPLEXITY ANALYSIS:

Number of comparisons compared to n lg n:





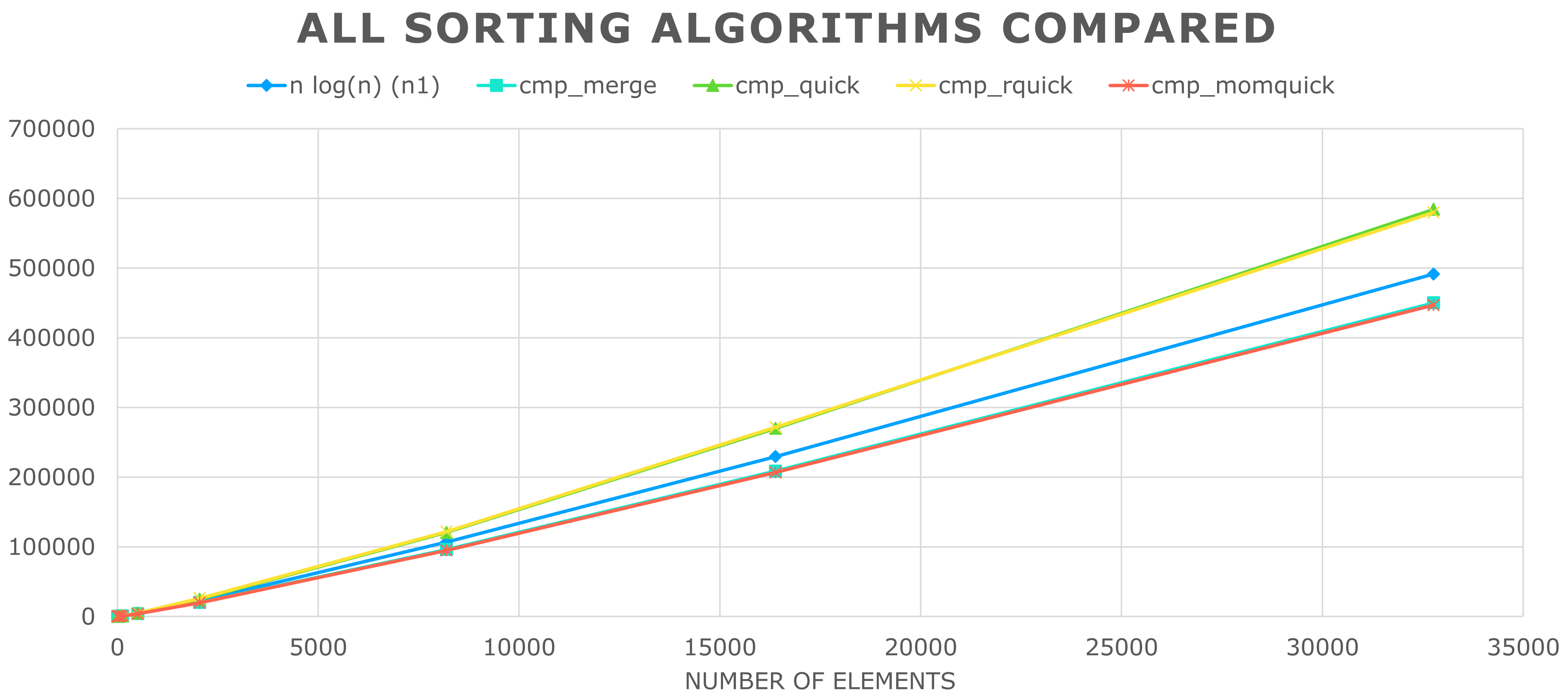
As we can see, the group size has no significant impact on the performance of the algorithm. Though, the grouping of 5 performs better than 3 and that of 7 performs slightly better than 5 for uniform distribution



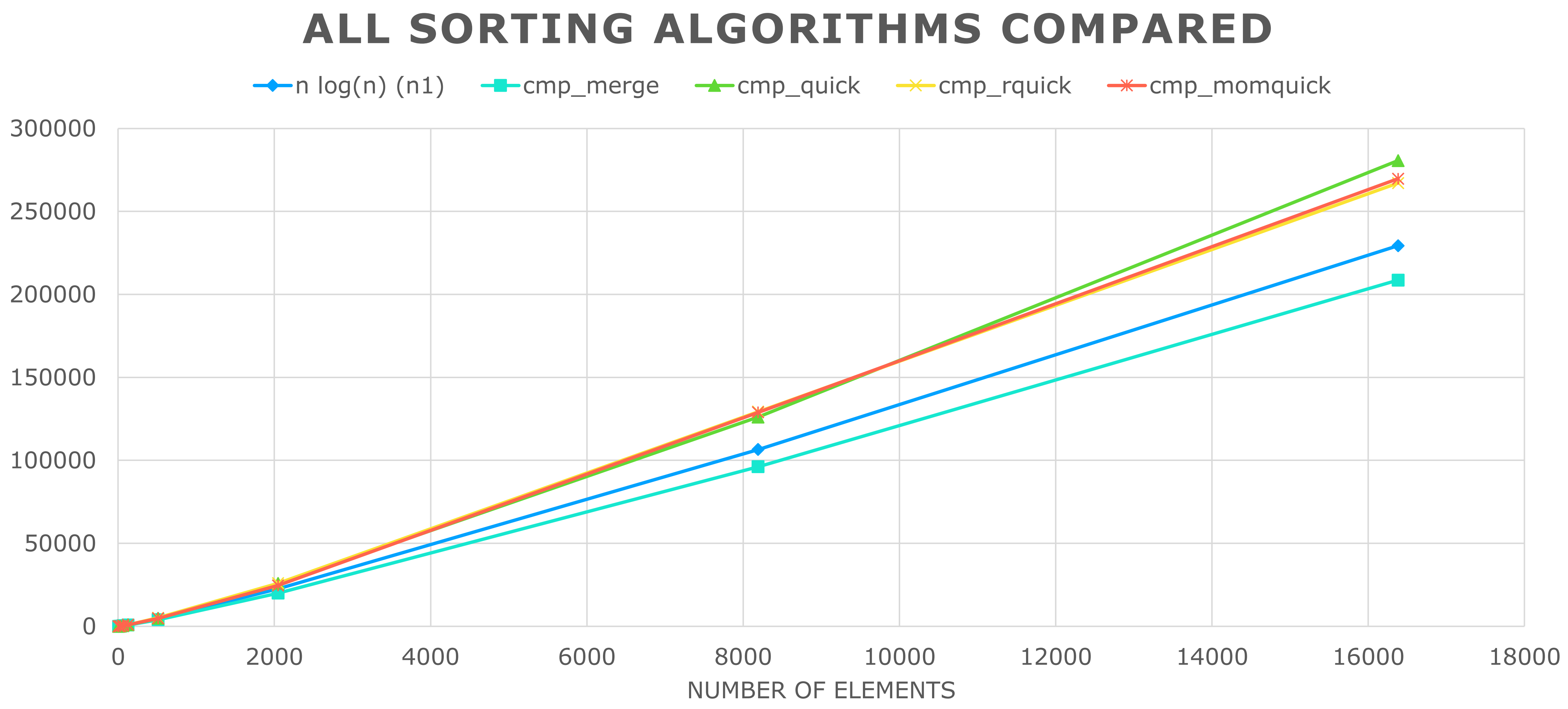
Here, we can see how the performance of the algorithm varies greatly with the type of dataset. For uniform distribution, it performs exceptionally well, performing in the order of O(n lg n), but for normal distribution, the performance is worse that O(n lg n).

This may be because the normal distribution has greater central tendency. Therefore, for uniform distribution, choosing the median of medians provides good partitions throughout, but for normal distribution, the partitioning worsens as the values are farther from any central tendency

COMPARISON FOR UNIFORM DISTRIBUTION:



COMPARISON FOR NORMAL DISTRIBUTION:

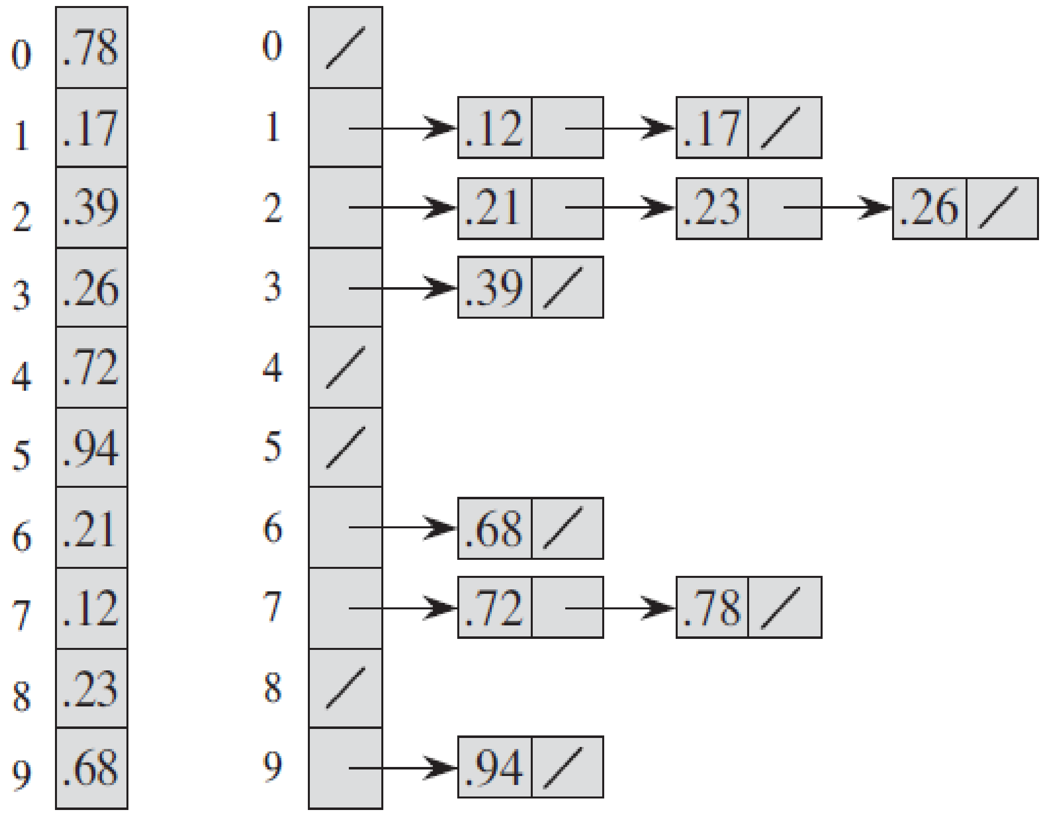


Bucket Sort

THE ALGORITHM

- ❖Bucket Sort algorithm is a way to sort elements in linear time, specially when the input in derived from a uniform distribution of data in range [0.1)
- ❖For a dataset containing n elements, the procedure creates n ‘buckets’ and puts the elements in the appropriate buckets to which they belong (using some hashing like: element * number of elements)
- ❖The elements in the individual buckets are then sorted in the desired order using any sorting algorithm, like insertion sort
- ❖The list of buckets is then traversed in order, and this will yield the desired sorted dataset, in linear time
- ❖With normal distribution, chances of creation of skewed buckets, i.e, extremely long buckets are possible, as the data is fairly close at one point
- ❖Bucket sort is supposed to give a complexity of O(n) as it sorts elements in linear time.

Pictorial representation of buckets:



Input array

Buckets created for input array

Experimental Results:

BUCKET SORT ON UNIFORM DISTRIBUTION:

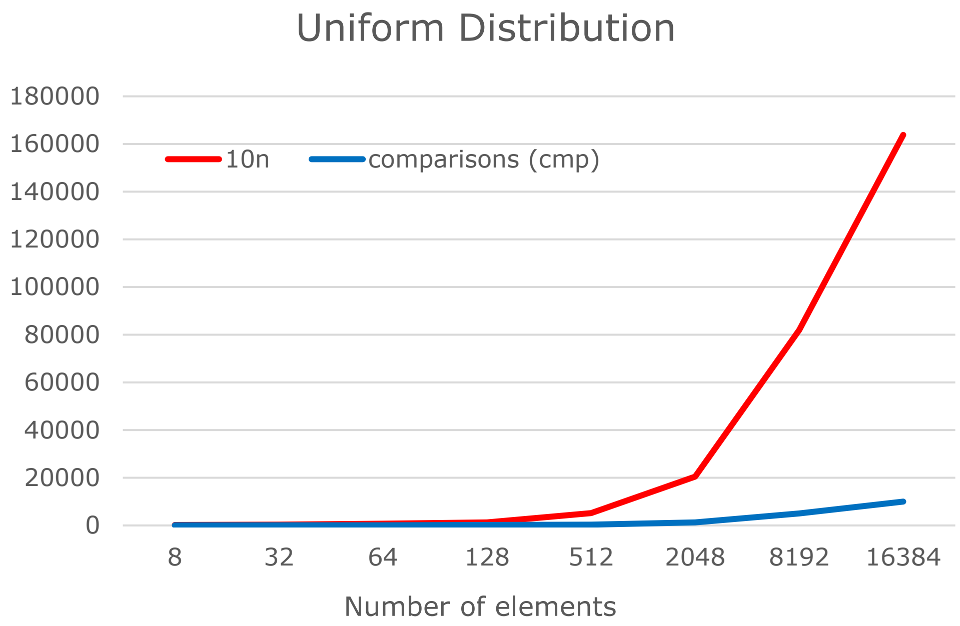
No. of elements	Comparisons (cmp)	Number of buckets	Max bucket size	CPU time
8	7	4	4	0.000002
32	31	18	5	0.000003
64	46	37	5	0.000005
128	94	78	4	0.000012
512	282	333	4	0.000037
2048	1206	1283	6	0.000149
8192	4994	5134	7	0.000617
16384	9922	10350	7	0.001431
32768	19770	20722	7	0.003938
65536	39590	41406	8	0.007629
262144	146061	165273	8	0.049889
524288	260091	329124	9	0.112542

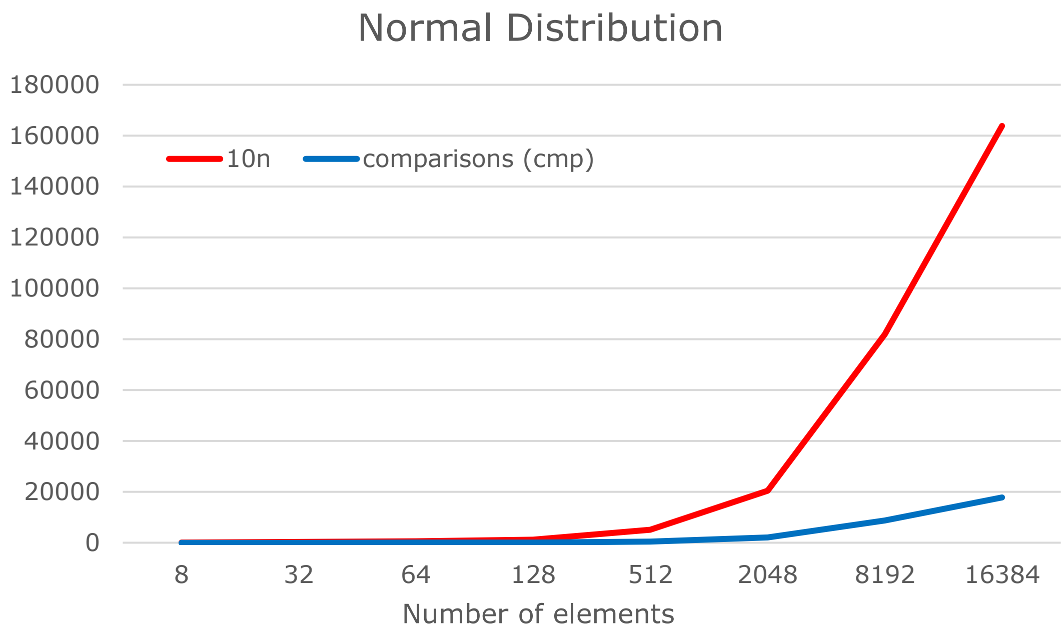
BUCKET SORT ON NORMAL DISTRIBUTION:

No. of elements	Comparisons (cmp)	Number of buckets	Max bucket size	CPU time
8	5	5	4	0.000003
32	20	18	3	0.000004
64	44	37	5	0.000007
128	101	67	5	0.000012
512	507	248	8	0.000042
2048	2075	960	7	0.000164
8192	8804	3659	9	0.000688
16384	17873	7197	11	0.001533
32768	35354	14446	11	0.003228
65536	77353	26585	14	0.009919
262144	281932	104317	13	0.060918
524288	506475	197545	15	0.151563

COMPLEXITY ANALYSIS:

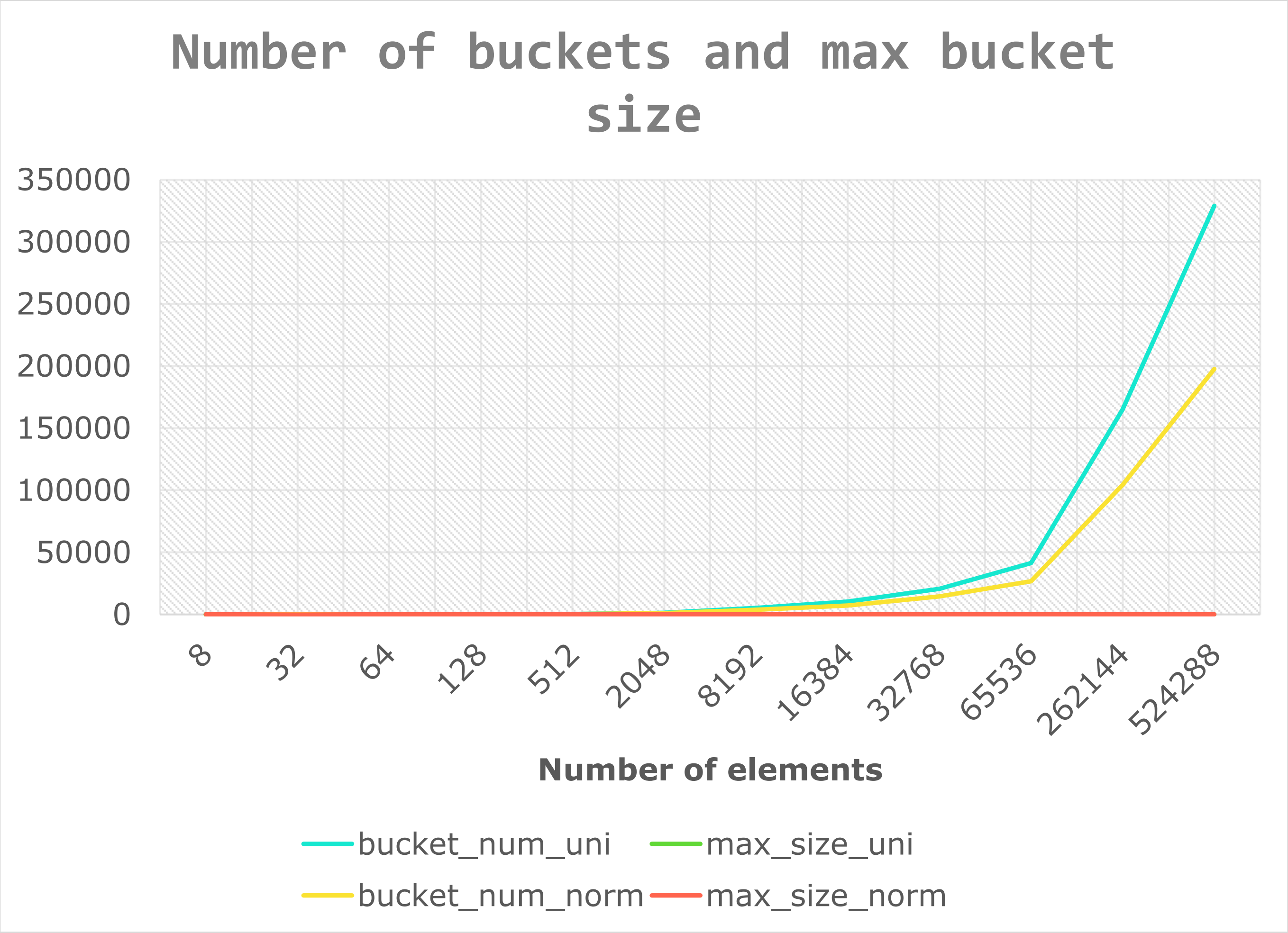
Number of comparisons compared to 10n:





We observe that the performance is similar for both the distributions, but it is slightly greater for the normal distribution as the number of elements increases.
Thus, we have verified that the bucket sort procedure has a complexity of O(n).

COMPARISON OF BUCKET USE:



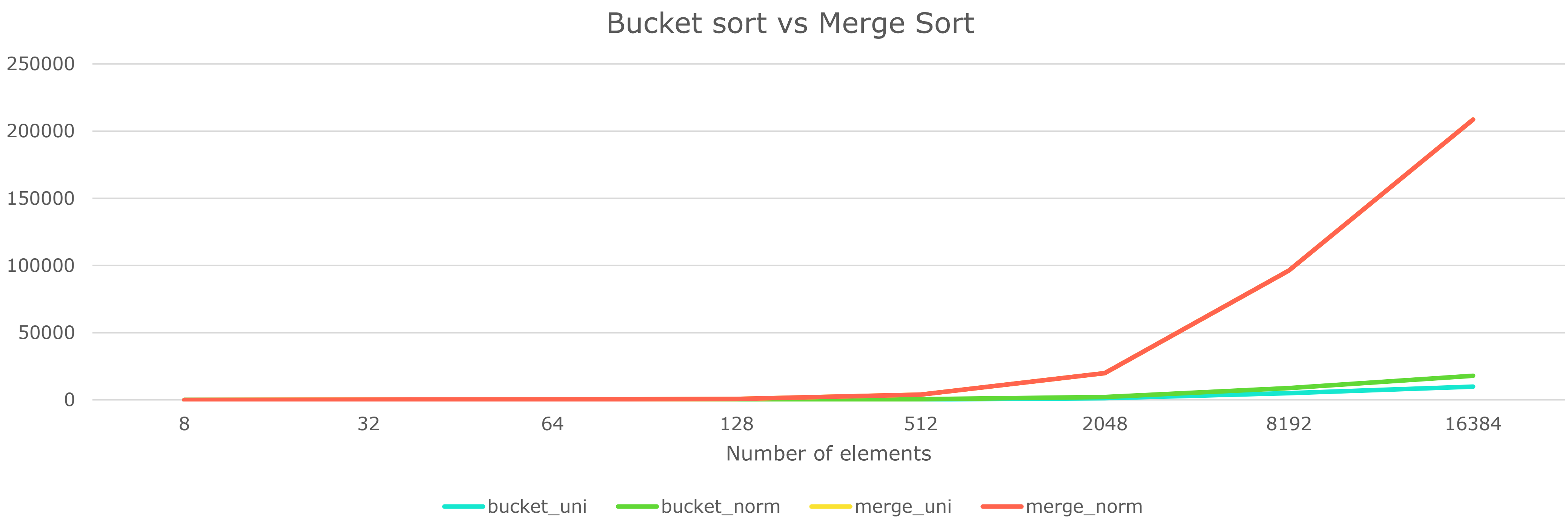
We can see that initially, the size of the largest bucket is quite similar for both the distributions, but as the number of elements increases, the size of largest bucket formed rises steeply for the normal distribution, indicating that the complexity will increase, as the sorting function used to sort the bucket will perform more number of operations.

For uniform distribution, we do not encounter such a problem as a uniform distribution is supposed to have uniformly distributed points, enabling the elements to be distributed in more bucket.

But in a normal distribution, more elements lie closer to the mean and then decrease further away from the mean. Therefore the bucket size increases as there are more number of closely spaced points.

BUCKET SORT VS. MERGE SORT

As we observed that merge sort has the best complexity of all the algorithms explored before, we will now compare it with bucket sort to see how different they are:



This gives us an idea of how drastically different these approaches are.

CONCLUSION:

From the above analysis, we can conclude that:

- For uniform distribution, merge sort and quick sort with median of medians as pivot performs relatively better, with a complexity of O(n lg n), whereas normal quick sort algorithm and randomized quick sort perform slightly worse that O(n lg n)

  bucket sort<merge sort<=mom quick sort<n lg n<randomized quick sort<=quick sort

- For normal distribution, only merge sort performs with a complexity of O(n lg n), while all the variants of the quick sort algorithm perform worse than a bound of O(n lg n)

  bucket sort<merge sort<n lg n<(quick sort, mom quick sort, randomized quick sort)

- Among the variants of quick sort, the randomized and median of median pivot selection work better as they make the algorithm independent of the order of elements in input dataset, thus ensuring that we obtain good partitions, and therefore bringing the complexity close to O(n lg n)
- Bucket sort provides a linear complexity of O(n) to sort elements, and this is vastly more efficient when compared to any of the other sorting algorithms
- The analysis has also shown how the algorithms stick to their bounds, and how the statistical measures to improve algorithms help, as seen in case of quick sort using randomized pivot selection

Thank you!