ASSIGNMENT 1

ALGORITHM LAB (CS2271)

Introduction

This Report is Made By:

- Abhiroop Mukherjee (510519109)
- Hritick Sharma (510519114)
- Sayak Rana (510519108)

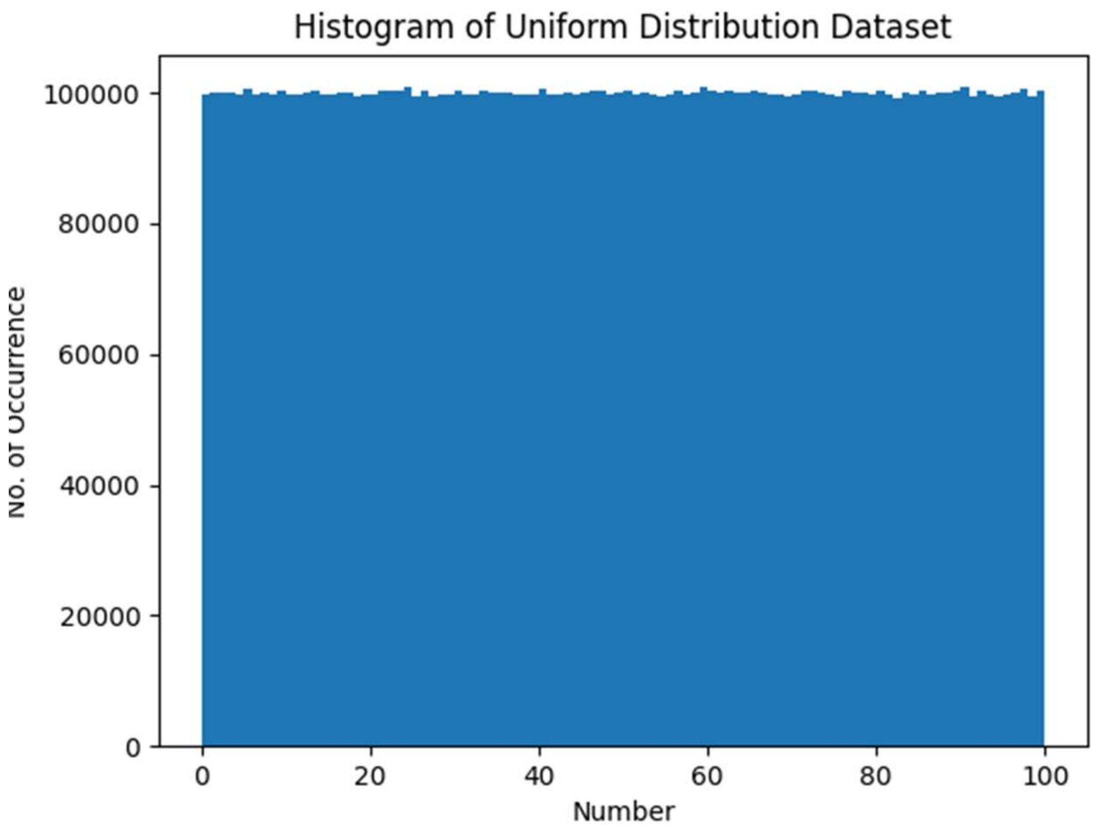
The Source codes for all the Assignments can be found here:
https://github.com/Abhiroop25902/CS-2271-Algo-Lab

Question 1A

CONSTRUCT LARGE DATA SETS TAKING RANDOM NUMBERS FROM UNIFORM DISTRIBUTION (UD)…

Procedure

- The inbuilt C function `rand()` generates random numbers distributed uniformly.
- Using `rand()` we generated 10^6 uniformly distributed random numbers between 0 to 100 and stored it in `uniform_distribution.csv` which will act as uniformly distributed dataset for further works.
- We also plotted a histogram of the dataset to make sure that the dataset generation worked well or not…



Observation

- We see from the histogram of the dataset that every numbers has almost equal number of occurrence, which shows that the numbers are uniformly distributed.

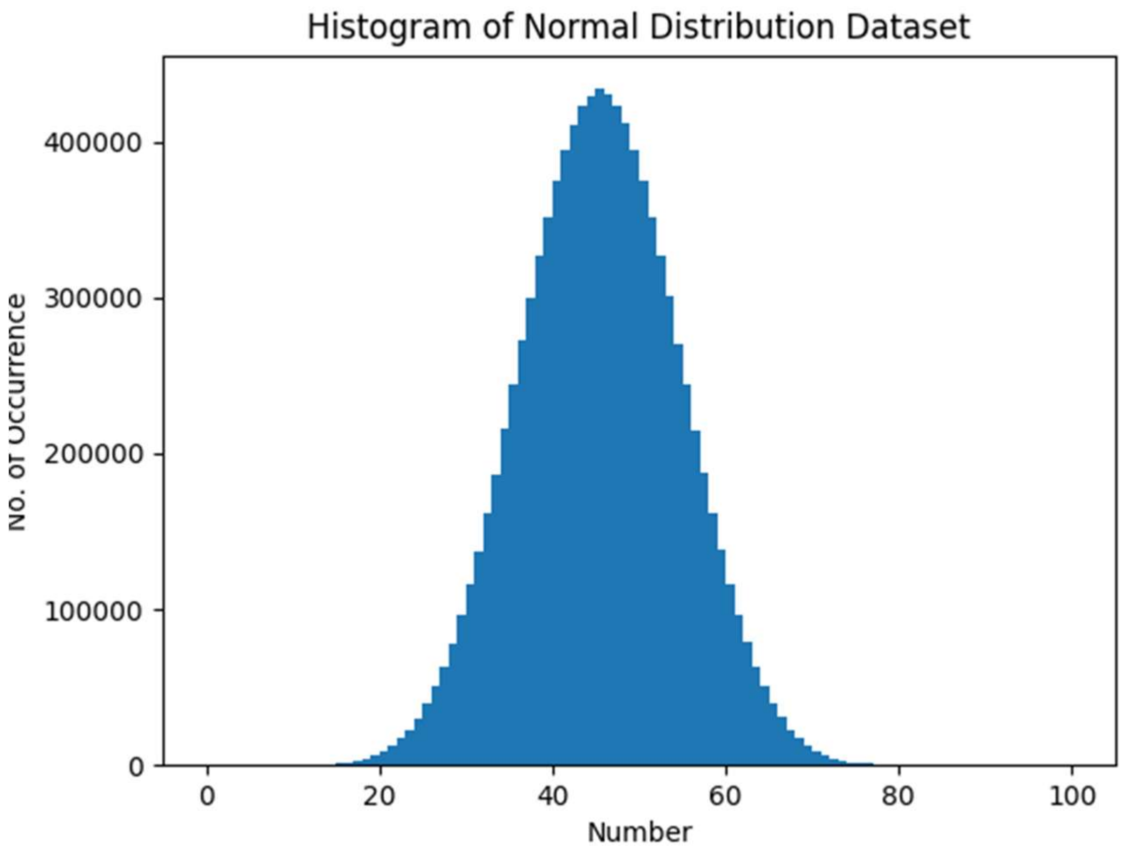
Question 1B

CONSTRUCT LARGE DATA SETS TAKING RANDOM NUMBERS FROM NORMAL DISTRIBUTION (ND)…

Procedure

.

- The idea is that sum of m uniform randomly generated number can't be too high and can't be too low, the sum will be more concentrated towards the mid, which resembles Normal Distribution.
- So we randomly generated 10 uniformly distributed numbers between 0 and 10, took their sum and saved it to the dataset.
- These value range was selected so as to make the min. sum 0 and max. sum 100, i.e. "Normally" Distributed dataset with values between 0-100.
- We did the same procedure 10^6 times to generate a dataset `normal_distribution.csv` consisting of 10^6 normally distributed numbers.
- Then we also plotted it's histogram to make sure if the dataset actually follows Normal Distribution or not…



Observation

- Here we see the recognizing shape of “bell” curve of Normally Distributed Random variable.
- This shows that the dataset values are uniformly distributed

Question 2A

IMPLEMENT MERGE SORT (MS) AND CHECK FOR CORRECTNESS.

What is **Merge Sort**

The Merge sort is a “Divide and Conquer” Algorithm which works as follows:

1. Divide n-element sequence into n/2 elements in two subsequences. [$O(1)$, divides into two parts]
2. CONQUER: sort the two subsequences recursively.
3. COMBINE: MERGE the two subsequences. [$O(n)$]

This method gives the following recurrence relation:

$$T(n) = 2\,T\left(\frac{n}{2}\right) + O(n)$$

Merge Sort Algorithm

```
MergeSort(arr[ ], l,  r){

If (r > l) {

     // Find the middle point to divide the array into two
halves

        middle m = (l+r)/2 ;

     // Call Merge Sort for first half

        MergeSort(arr, l, m) ;

     // Call merge Sort for second half

        MergeSort(arr, m+1, r) ;

     // Merge the two halves sorted

        Merge(arr, l, m, r) ;

        }

}
```

Question 2B

IMPLEMENT QUICK SORT (QS) AND CHECK FOR CORRECTNESS.

What is **Quick Sort**

Quicksort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of Quicksort that pick pivot in different ways.

1. Always pick first element as pivot. (implemented below)
2. Always pick last element as pivot
3. Pick a random element as pivot.
4. Pick median as pivot.

The key process in Quicksort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

Quick Sort Algorithm

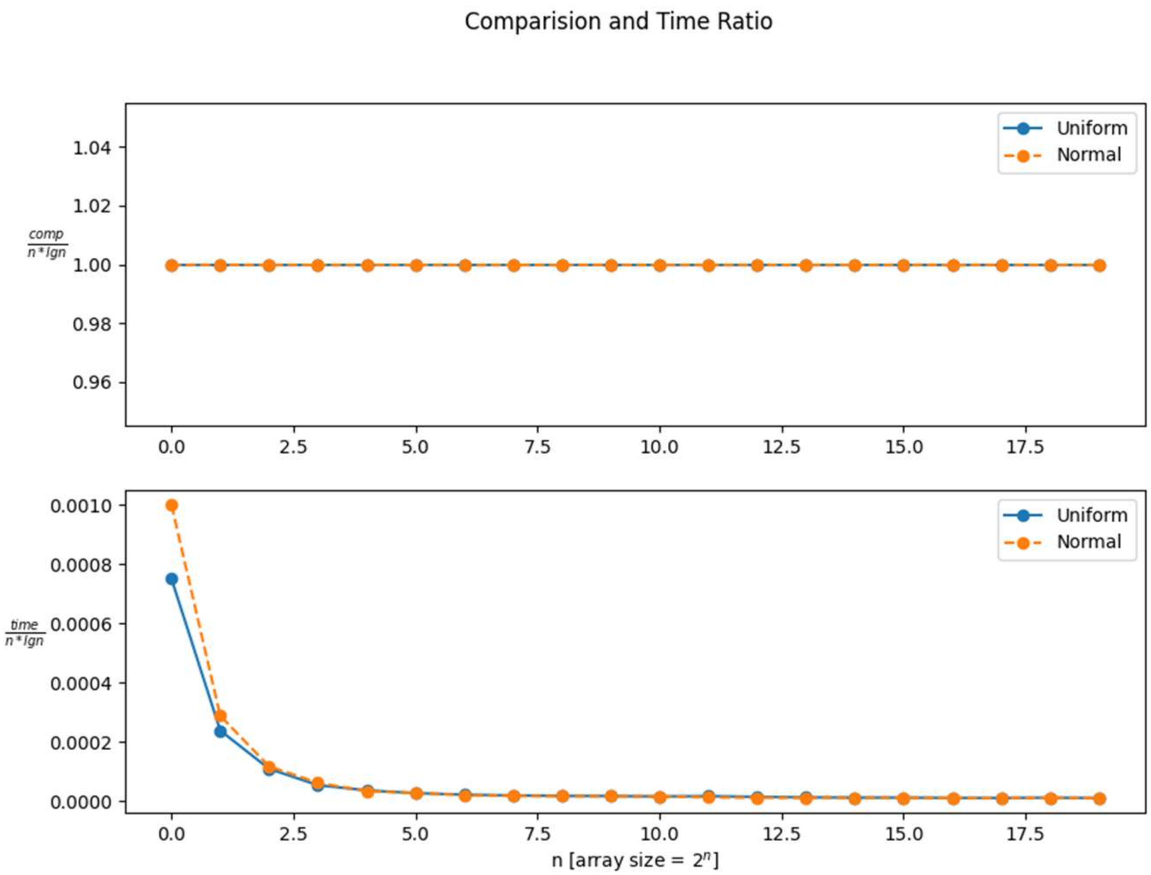
```
PARTITION(A,p,r){

        x=A[p] ;

        i = p-1; j = r+1 ;

        while (true){
                repeat j=j-1 until A[j}<=x;
                repeat i=i+1 until A[i]>=x;
        }

        if( i < j )

                exchange( A[i] , A[j] );

        else

                return (j);

}

QUICK_SORT(A,p,r) {

        if (p<r) {

                q=PARTITION(A,p,r) ;

                QUICK_SORT(A,p,q) ;

                QUICK_SORT(A,q+1,r) ;

        }

}
```

Question 3

COUNT THE OPERATIONS PERFORMED, LIKE COMPARISONS AND SWAPS WITH PROBLEM SIZE INCREASING IN POWERS OF 2, FOR BOTH MS AND QS WITH BOTH UD AND ND AS INPUT DATA.

Procedure

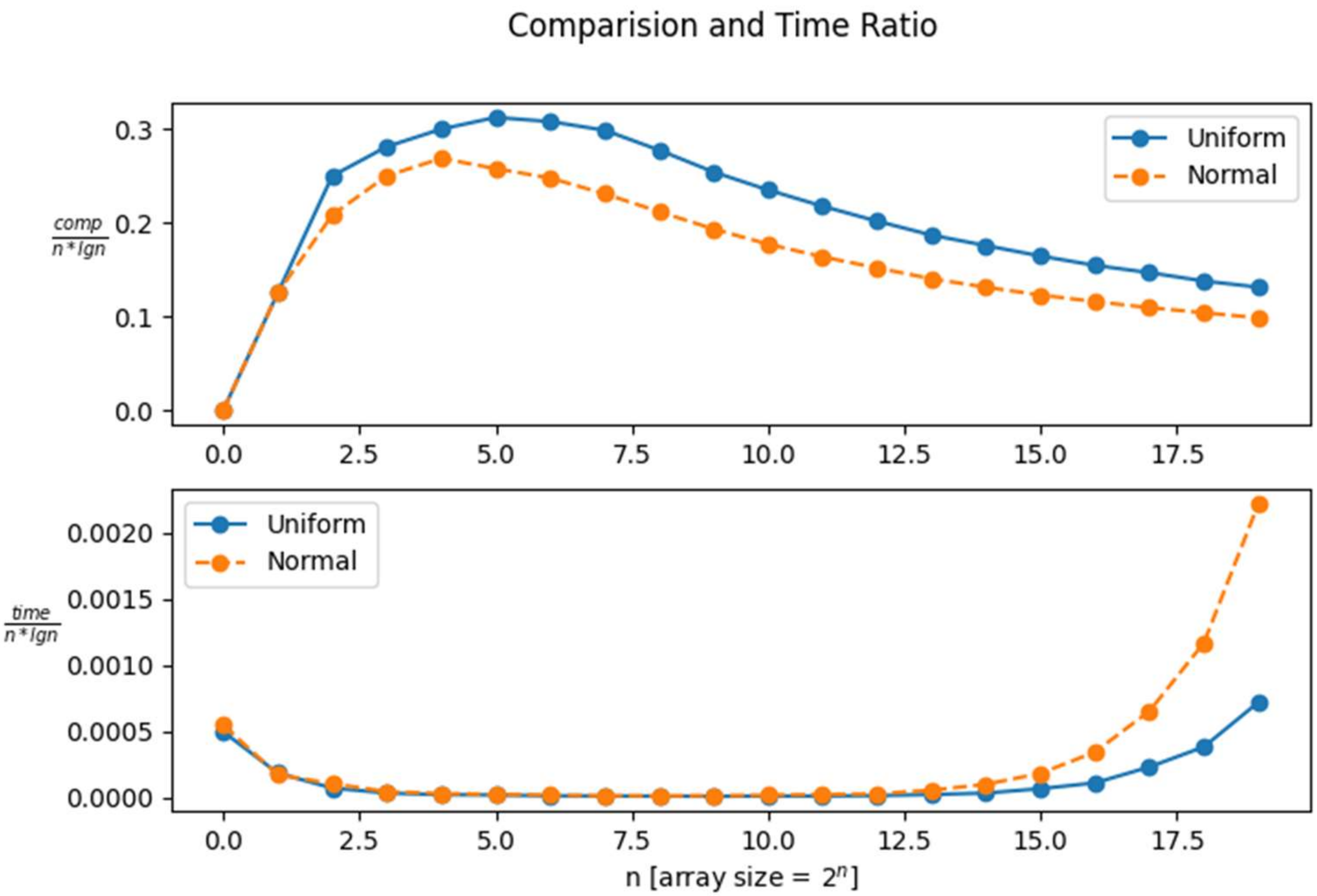
- We did both sorting algorithm for different array size, which starts from 2 and increments in powers of 2 till our system gives Error due to Huge Size
  - We were able to record observation from array size 2 to array size 2^{20}
- For each array size n, we sorted 100 different arrays of size n, taken from the datasets made in Question 1. Recorded the number of comparison and Time Taken per sort, then took their mean.
- We then Plotted the following graphs for both datasets
  - $comparision\ ratio = \frac{avg\_comp}{nlgn}$ vs n
  - $time\ ratio = \frac{avg\_time\_taken}{nlgn}$ vs n



Observation for Merge sort

Following Inference can be made by the Observed Graph:

- From the graphs, we see that both time ratio and comparison ratio converges to constant as n goes to very big sizes, it means that Merge Sort Algorithm has $O(nlgn)$ complexity

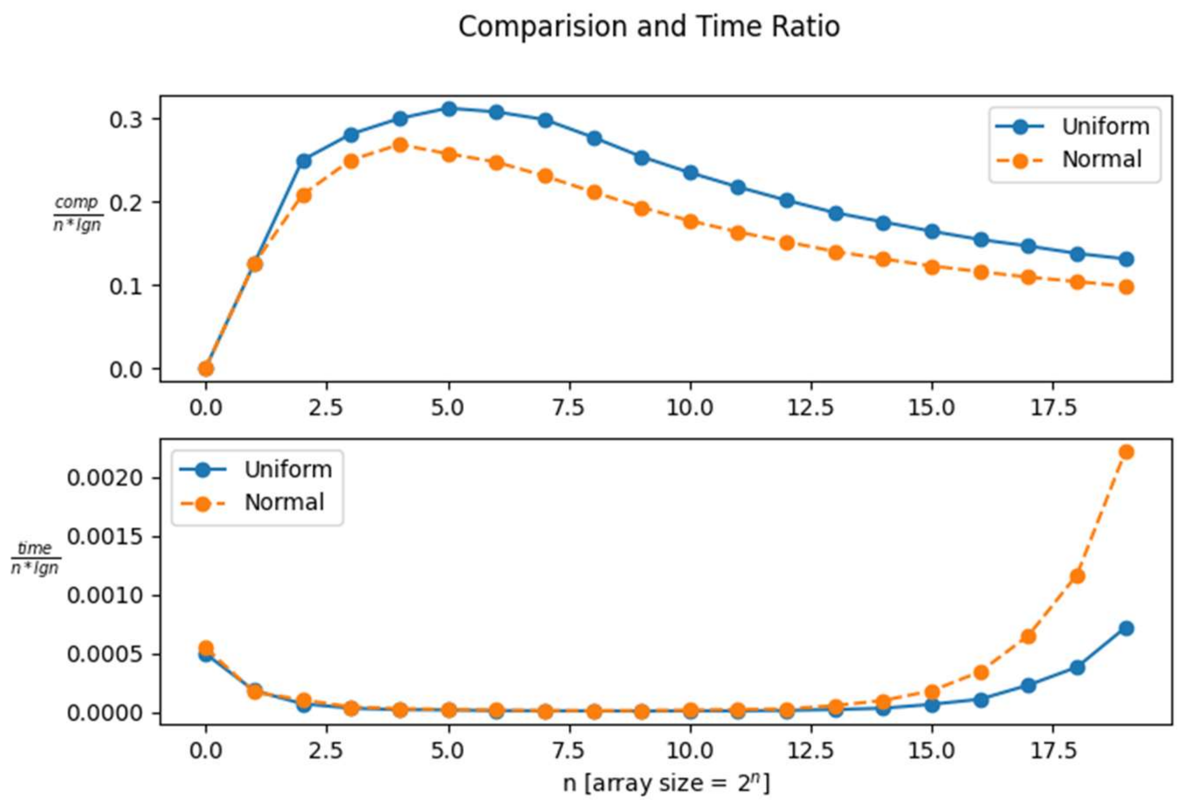


Observation for Quicksort

Following Inference can be made by the Observed Graph:

- From the graphs, we see that both comparison ratio converges to constant as n goes to very big sizes, it means that Merge Sort Algorithm has $O(nlgn)$ complexity
- As array size keep increasing, we see peculiar observations in Time ratio
  1. Time ratio diverges for very high array size
  2. Although Comparison Taken is lower for Normal, Time Taken for Normal Distribution is higher than Uniform Dataset

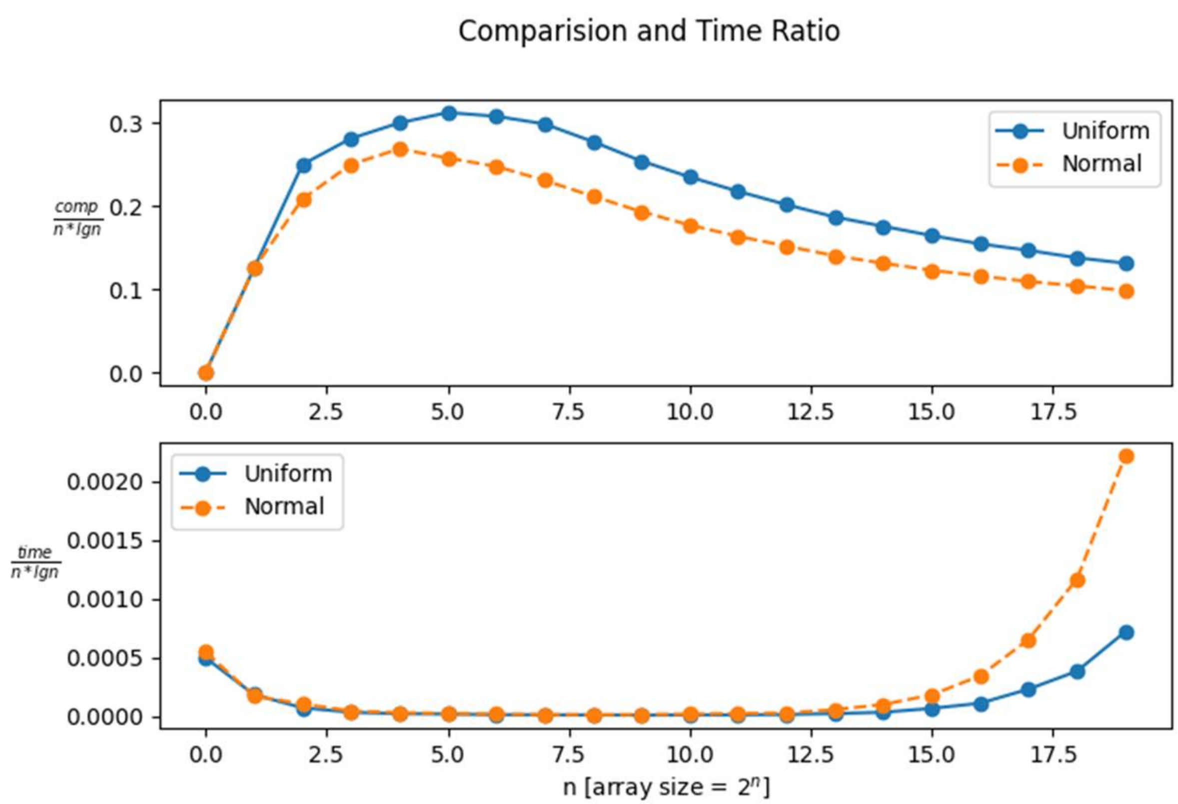
Comparison Analysis Quicksort



For Normal Distribution, there is higher chance for a pivot to be near median, which makes more chances for 1:1 Partitions.

And the more 1:1 partition we get, more the less sub partitions will be formed, resulting in less no. of comparisons.

Due to this property, we see a smaller number of comparisons for normal, when compared to uniform distribution



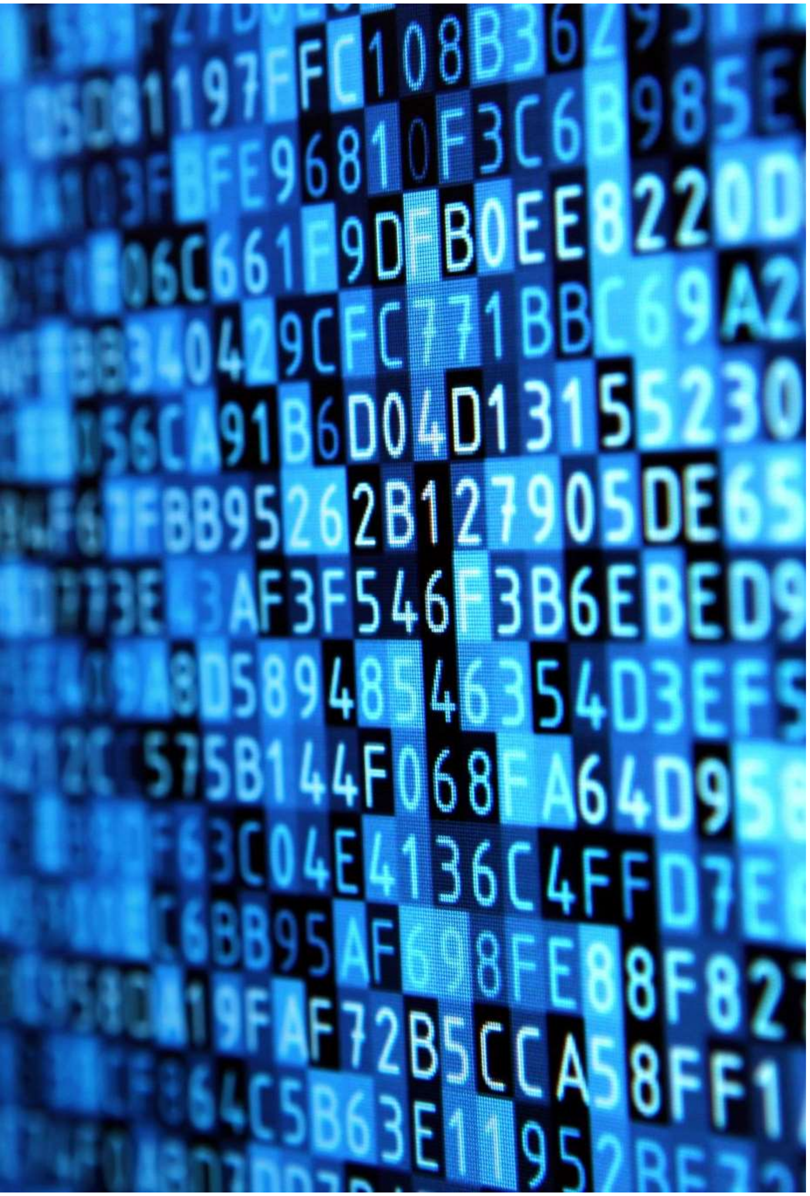
Time Analysis Quicksort

Here we see very strange observations:

1. Even though Normal Distribution takes less comparison, it takes more time than Uniform. This could be since Time taken for comparison in Normal is more than that of Uniform
2. As array size goes too high, we see time taken by algorithm breaks the $O(nlgn)$ complexity, we don't have any idea why this is happening

Question 4

EXPERIMENT WITH RANDOMIZED QUICK SORT WITH BOTH UNIFORM AND NORMAL DISTRIBUTION AS INPUT DATA TO ARRIVE AT THE AVERAGE COMPLEXITY (COUNT OF OPERATIONS PERFORMED) WITH BOTH INPUT DATASETS.



What is a Randomized Algorithm?

An algorithm that uses random numbers to decide what to do next anywhere in its logic is called a Randomized Algorithm. For example, in Randomized Quick Sort, we use a random number to pick the next pivot (or we randomly shuffle the array). And in Karger's algorithm, we randomly pick an edge.

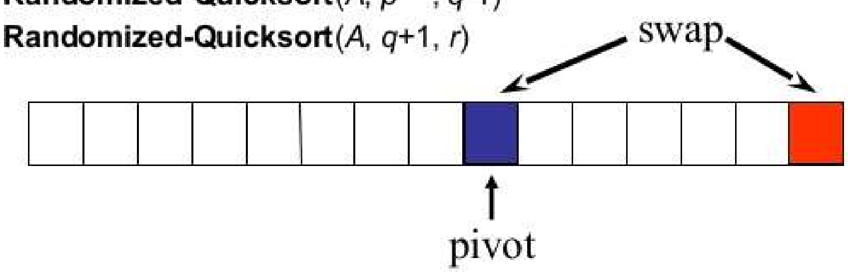
Randomized Quicksort

<u>Randomized-Partition(*A, p, r*)</u>

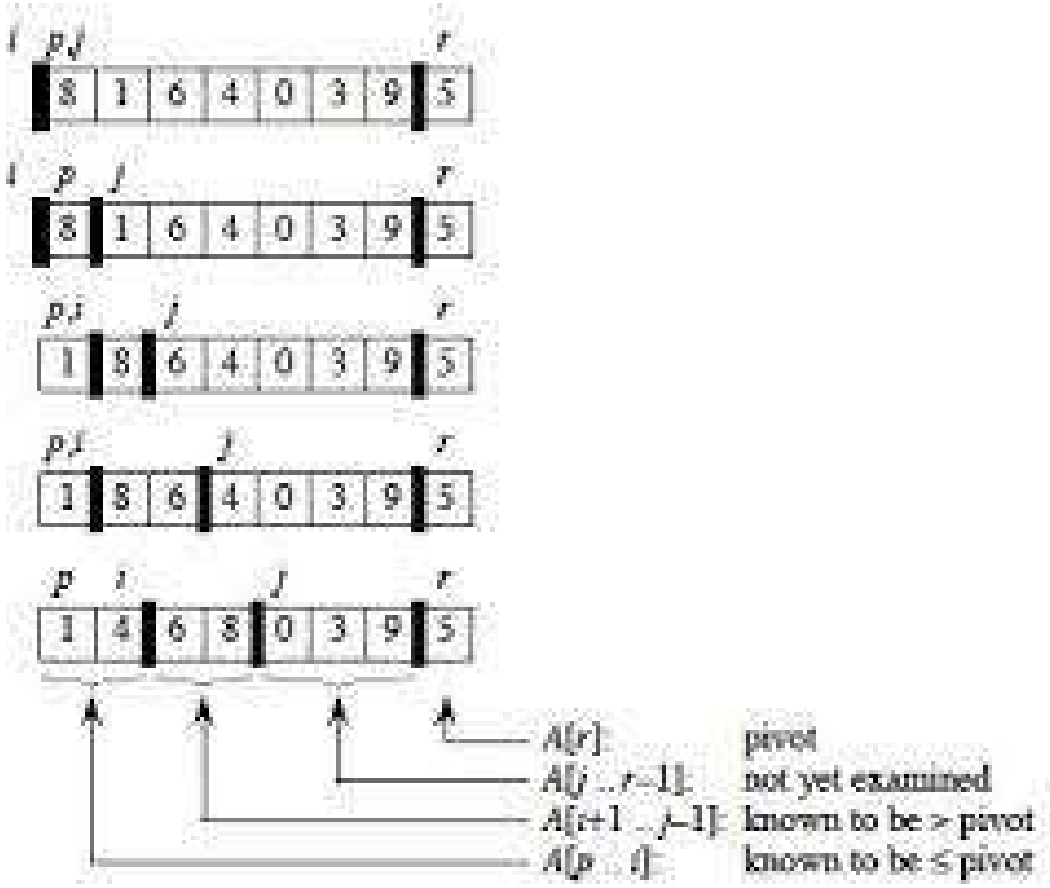
1. $i \leftarrow Random(p, r)$
2. exchange $A[r] \leftrightarrow A[i]$
3. **return Partition**(A, p, r)

<u>Randomized-Quicksort(*A, p, r*)</u>

1. **if** $p < r$
2.   **then** $q \leftarrow$ **Randomized-Partition**(A, p, r)
3.   **Randomized-Quicksort**$(A, p, q\text{-}1)$
4.   **Randomized-Quicksort**$(A, q\text{+}1, r)$



Randomized Quicksort Algorithm



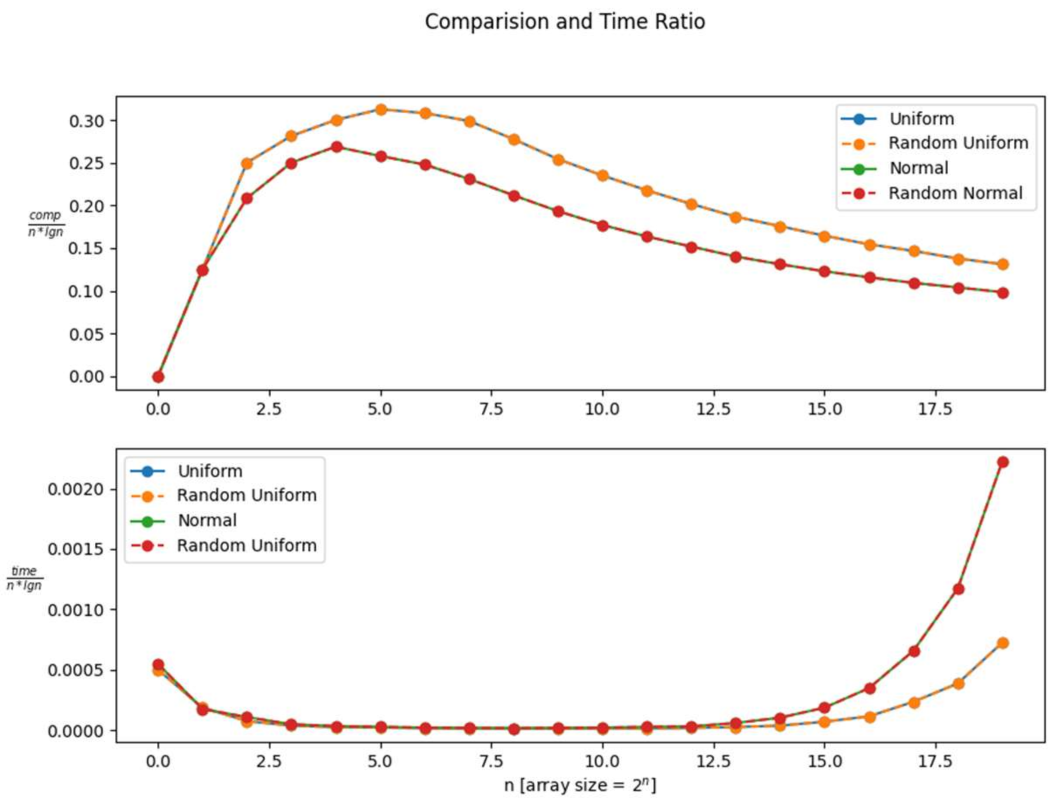
Working of Randomized Quicksort

Time Complexity of RQ..

Randomized quicksort has expected time complexity as $O(nlgn)$,but worst-case time complexity remains same. In worst case the randomized function can pick the index of corner element every time.

Procedure

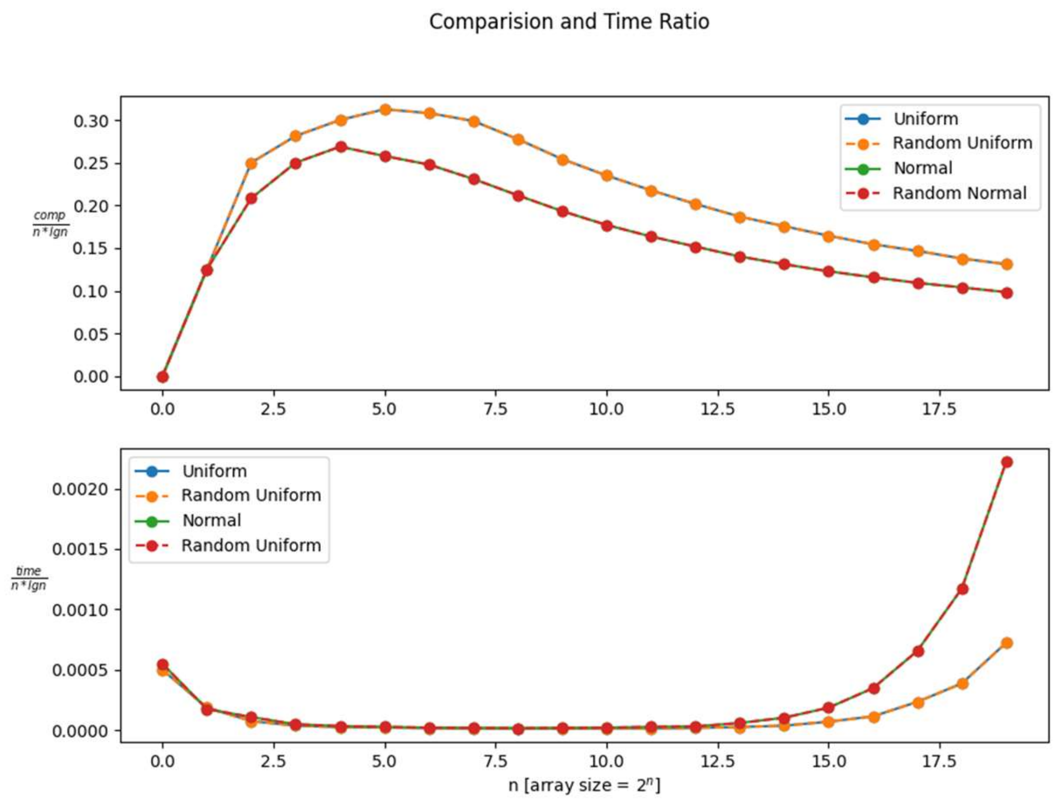
- We did Randomized Quicksort for different array size, which starts from 2 and increments in powers of 2 till our system gives Error due to Huge Size
  - We were able to record observation from array size 2 to array size 2^{20}
- For each array size n, we sorted 50 different arrays of size n, taken from the datasets made in Question 1. Recorded the number of comparison and Time Taken per sort, then took their mean.
- We then Plotted the following graphs for both datasets
  - $comparision\ ratio = \frac{avg\_comp}{nlgn}$ vs n
  - $time\ ratio = \frac{avg\_time\_taken}{nlgn}$ vs n
- We also plotted the usual Quick Sort Observations for comparisons too.



Observation

Following Inference can be made by the Observed Graph:

- As we did 50 sorts per n, and took its mean, the time complexity comes out to be same as Non-Random Quick Sort, i.e., $O(nlgn)$
- As array size keep increasing, we see similar observation, i.e., the comparison ratio and time ratio converges.
  - The Divergence of Time ratio for Big array size might be linked to low system RAM. which asks for more Secondary Memory Calls



Problem and Solution

As we can see in the graphs, the observation between Randomized and Non-Randomized Algorithms are near same as we averaged the observation

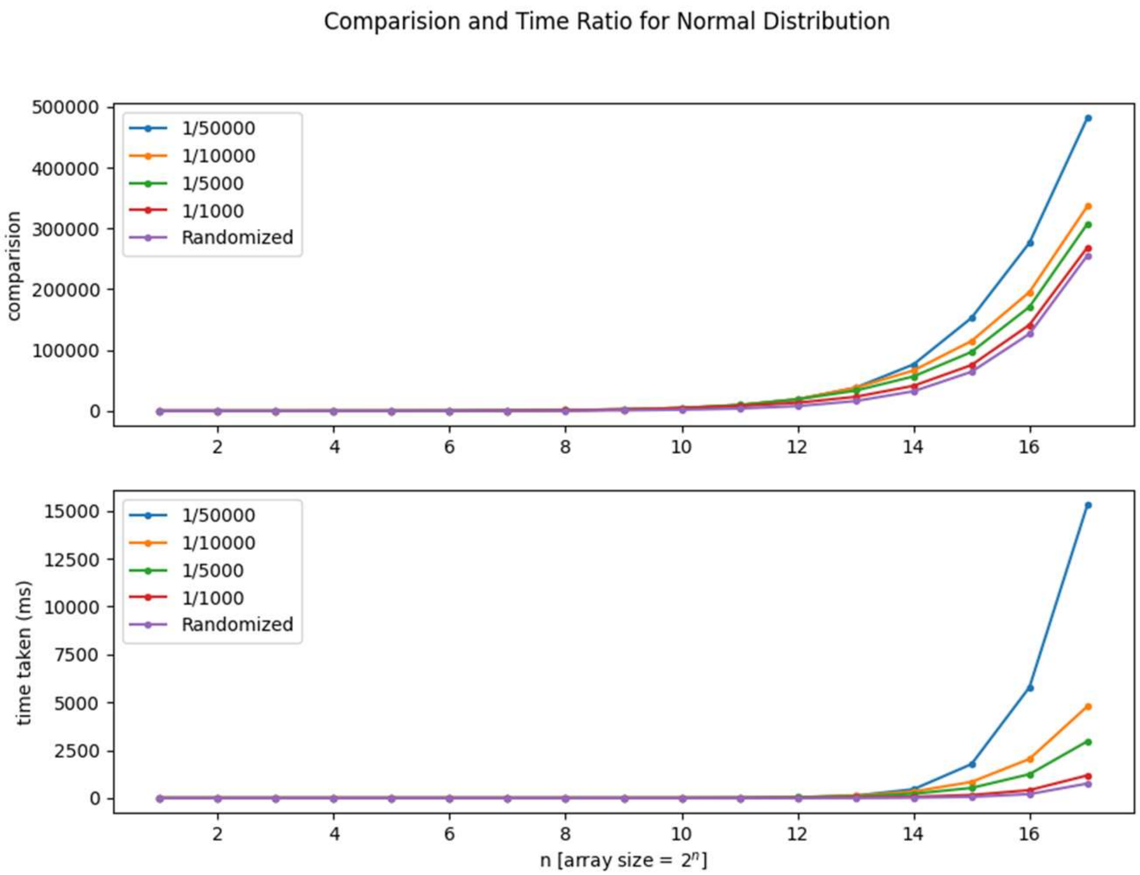
Solution: We Test the Randomized Quicksort with different deliberately made partition of size other than halves (as randomized tends to go towards 1:1 partition)

How do we make custom size partitions

- As we need to make our own size partitions, we need a way to make custom sized partitions
- For this we used `randomized_select()` algorithm
- This function uses same `randomized_partition()` to find k^{th} smallest element of array without sorting the array and does the same in $O(n)$ average time complexity
- We can then use the output of `randomized_select()` as pivot to make custom size partitions

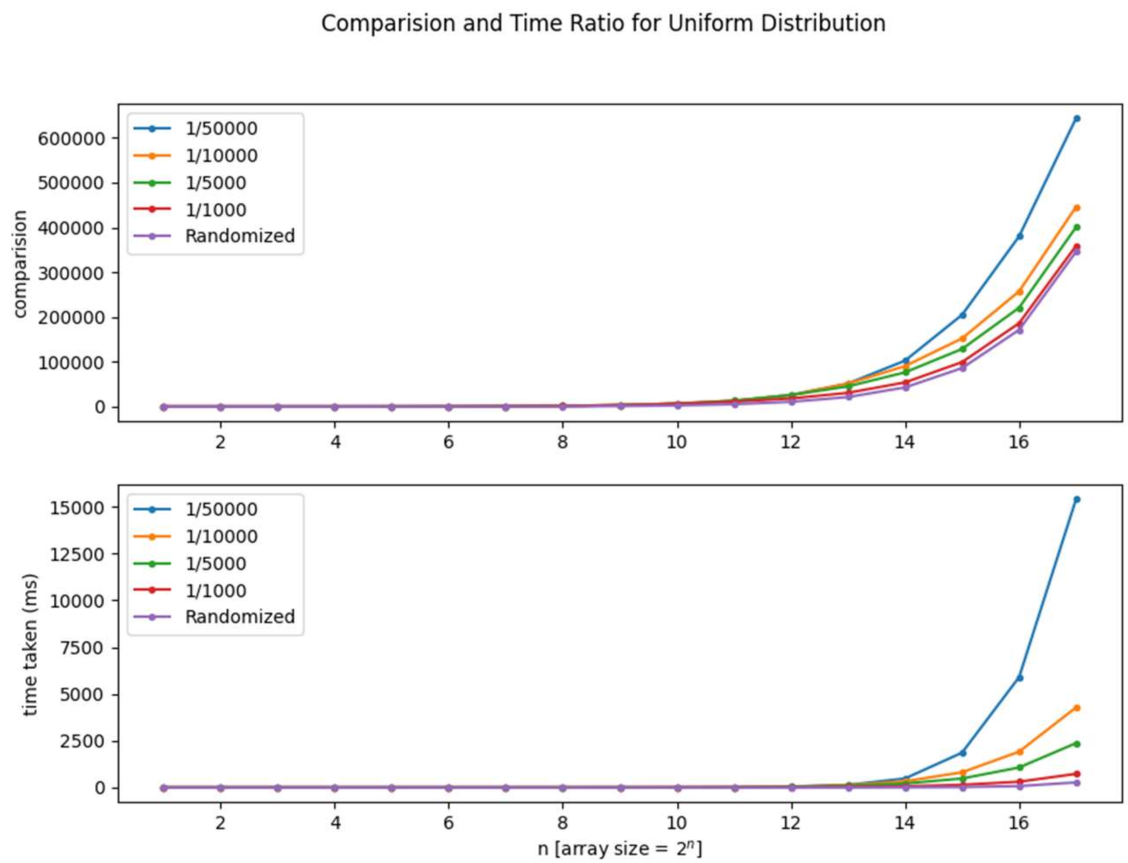
Procedure

- We did Randomized Quicksort for different array sizes as before, as well as different partition sizes
- For each array size n and partition size, we sorted 10 different arrays of size n, taken from the datasets made in Question 1. Recorded the number of comparison and Time Taken per sort, then took their mean.
- We then Plotted the following graphs for both datasets
  - $comparision\ ratio = avg\_comp$ vs n
  - $time\ ratio = avg\_time\_taken$ vs n
- Notation regarding the Upcoming Observation
  - Fraction $\frac{1}{\alpha}$ means that partitions are made as follows
    - One Partition has size α
    - The other Partition has size $(\mathrm{arr\_size} - \alpha)$
- We recorded partitions size so as to make the partitions as skew as possible



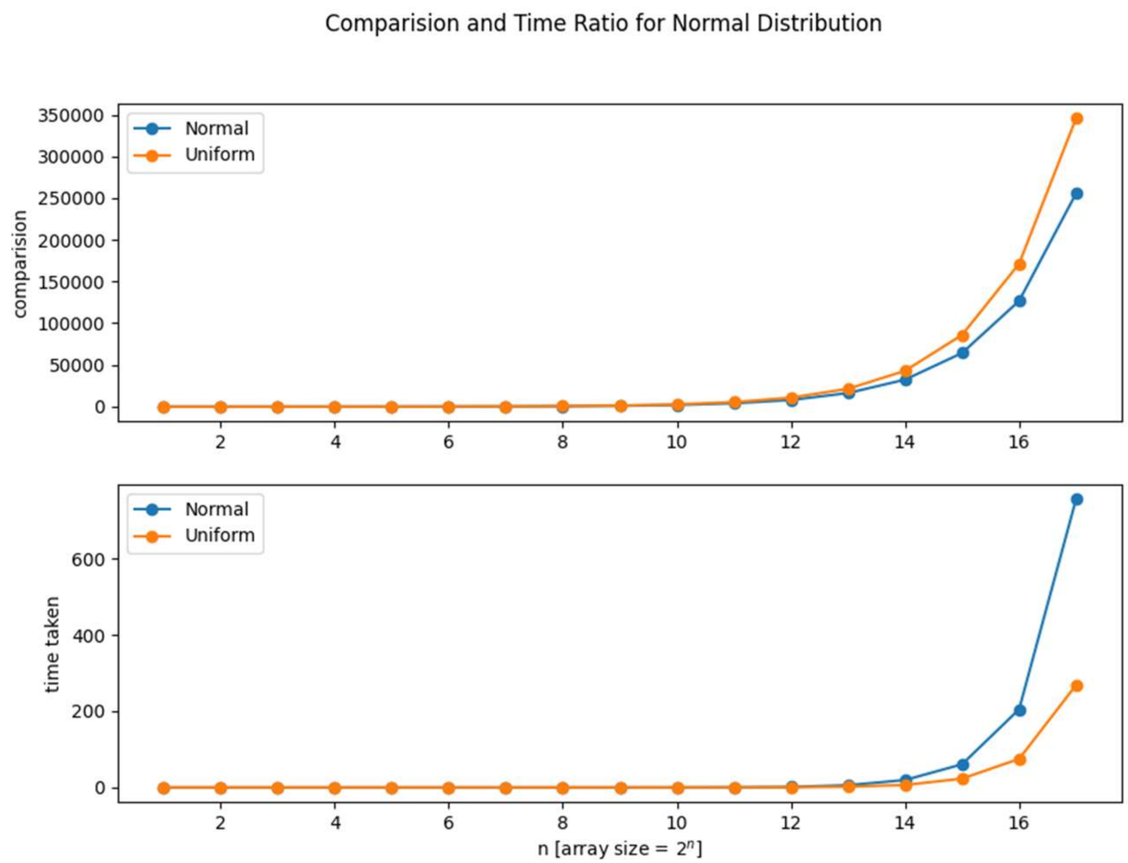
Normal Observation

Here we see that as we deliberately try to make partitions skew, both the time taken and number of comparisons increase drastically and goes towards $O(n^2)$ complexity.



Uniform Observation

Here we see same observations as that of Normal Distribution



Normal vs Uniform

Here we see same observations as the previous quick sort observations, i.e. Comparisons of Normal is lower than Uniform as in Normal Distribution, most of the elements are concentrated towards median.

And same as before, Time taken by Normal is more than Uniform although its Comparison are less than Uniform

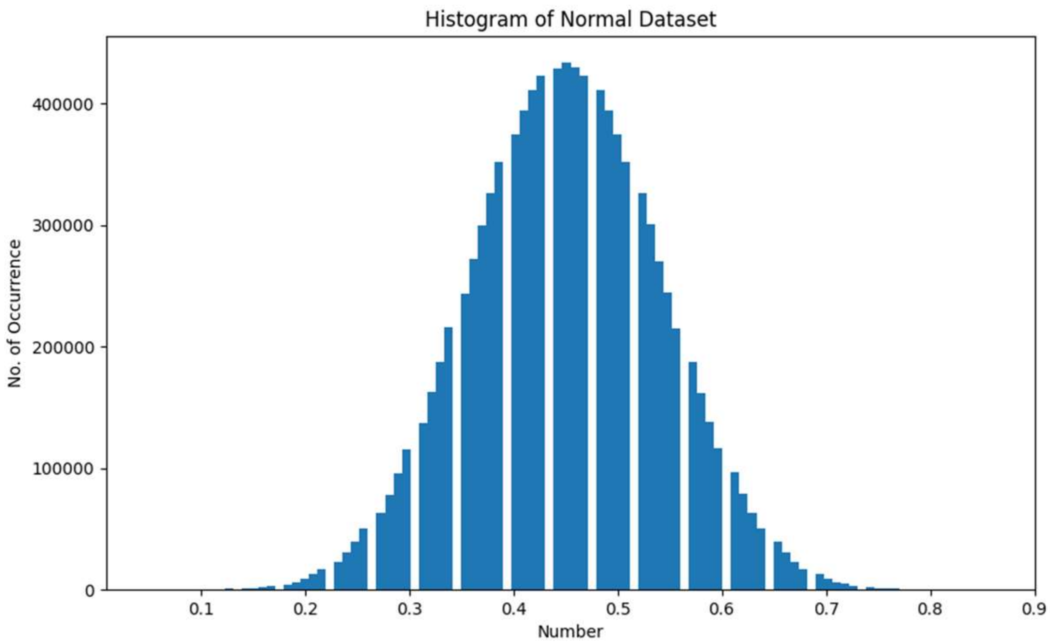
Question 5

NOW NORMALIZE BOTH THE DATASETS IN THE RANGE FROM 0 TO 1 AND IMPLEMENT BUCKET SORT (BS) ALGORITHM AND CHECK FOR CORRECTNESS

Procedure

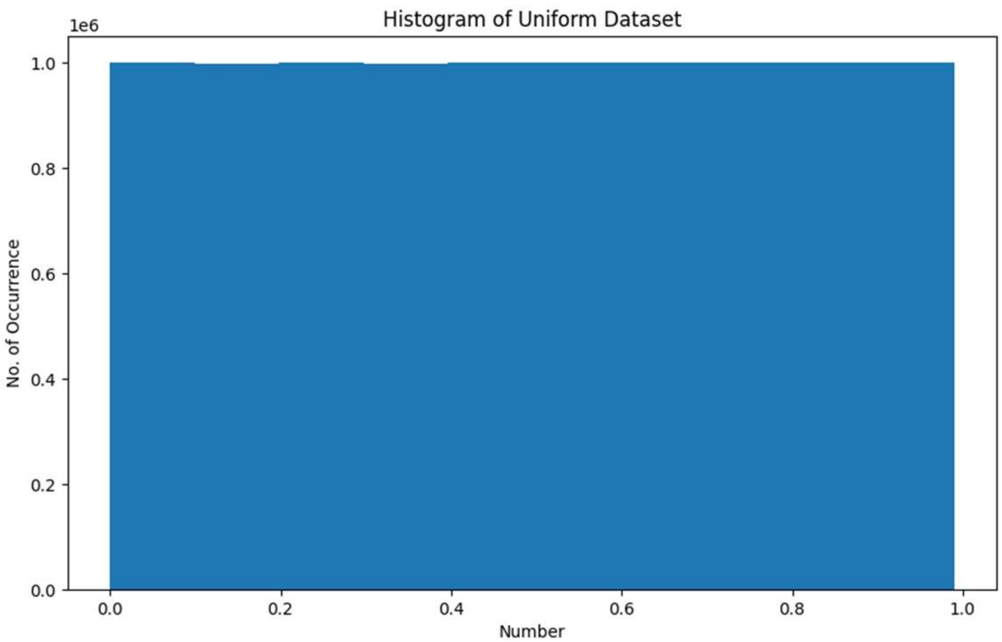
- As All Values in our dataset was between 0 and 100, we took each value one by one, divided each value by 100, and then saved it to a different files.
- The no. of data in the dataset was kept same, i.e. 10^6
- We Then Plotted the histogram of the datasets to make sure the conversion was conversion was correct.

Histogram for Normal Dataset



- As Expected, the histogram has the “bell” curve and max value of the dataset goes to 1.
- Hence, our conversion was successful

Histogram for Uniform Dataset



- As Expected, the histogram has the same number of occurrences for every numbers range and max value of the dataset goes to 1.
- Hence, our conversion was successful

Question 6

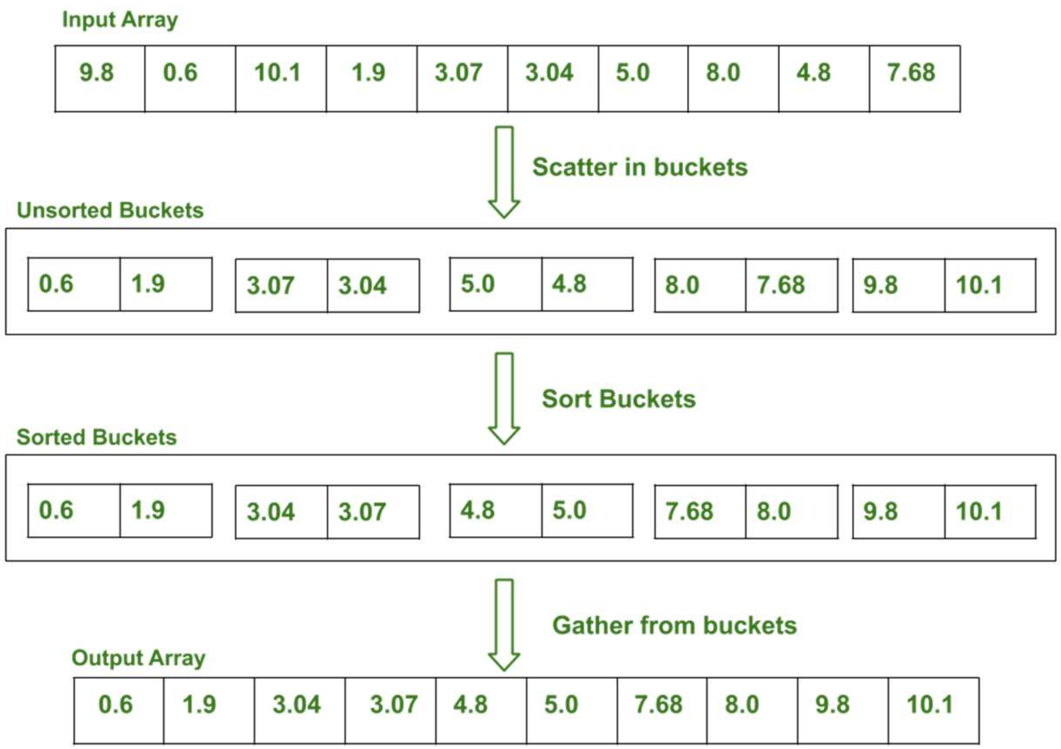
EXPERIMENT WITH BUCKET SORT TO ARRIVE AT ITS AVERAGE COMPLEXITY FOR BOTH UNIFORM AND NORMAL DATASETS AND INFER.

What is **Bucket Sort**

Bucket sort, or bin sort, is a sorting algorithm that works by distributing the elements of an array into several buckets. Each bucket is then sorted individually, either using a different sorting algorithm, or by recursively applying the bucket sorting algorithm. It is a distribution sort, a generalization of pigeonhole sort, and is a cousin of radix sort in the most-to-least significant digit flavor. Bucket sort can be implemented with comparisons and therefore can also be considered a comparison sort algorithm. The computational complexity depends on the algorithm used to sort each bucket, the number of buckets to use, and whether the input is uniformly distributed.

Bucket sort algorithm….

Bucket sort implementation on an array



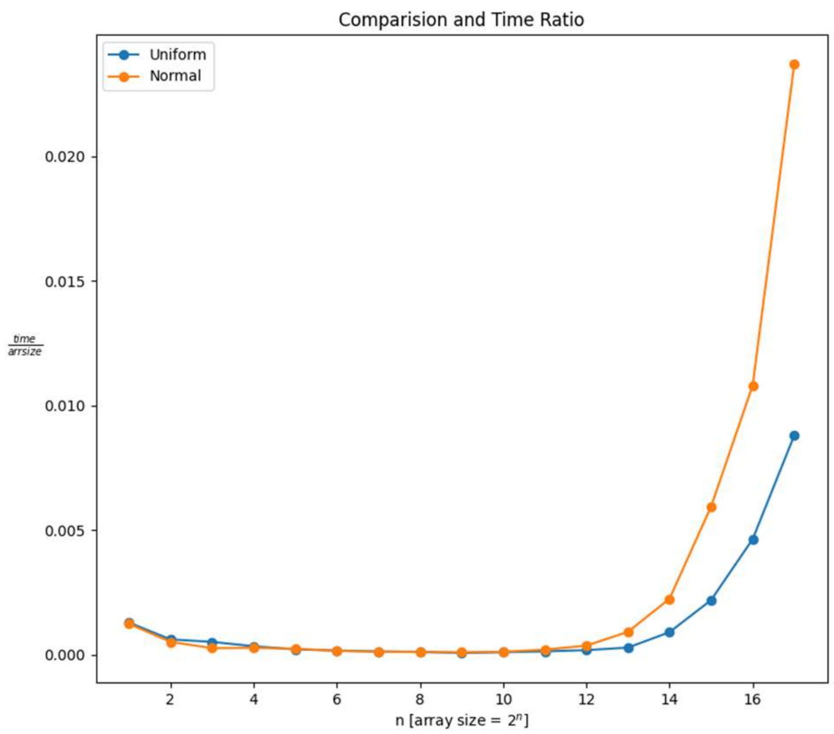
The complexity of the Bucket Sort Technique

Time Complexity: $O(n + k)$ for best case and average case and $O(n^2)$for the worst case.

Space Complexity: $O(nk)$ for worst case

Procedure

- We did Bucket Sort for different array size, which starts from 2 and increments in powers of 2 till 2^{15}
- For each array size n, we sorted 10 different arrays of size n, taken from the datasets made in Question 1. Recorded the number of comparison and Time Taken per sort, then took their mean.
- We then Plotted the following graph
  - $time\ ratio = \frac{avg\_time\_taken}{n}$ vs n



Observation

- Here we see that time taken by the Normal Distribution is way more than that of Uniform Distribution
- This is happening because in Normal Distribution more elements are concentrated around mean, due to which a lot of elements gets inserted on a same bin, and sorting that large linked list is very costly with respect to time.

Question 7

IMPLEMENT THE WORST CASE LINEAR MEDIAN SELECTION ALGORITHM BY TAKING THE MEDIAN OF MEDIANS (MOM) AS THE PIVOTAL ELEMENT AND CHECK FOR CORRECTNESS.

What is **Median Of Median** (MoM)

The Median of Medians is an approximate (median) selection algorithm, frequently used to supply a good pivot for an exact selection algorithm, mainly the quicksort, that selects the k^{th} largest element of an initially unsorted array.

Median of medians finds an approximate median in linear time only, which is limited but an additional overhead for quicksort.

When this approximate median is used as an improved pivot, the worst-case complexity of quicksort reduces significantly from $O(n^2)$ to $O(nlgn)$, which is also the asymptotically optimal worst-case complexity of any sorting algorithm.

Median Of Median Algorithm

1. Divide n elements into $\frac{n}{divide\_size}$ groups of $divide\_size$ element each, and $n \mathbin{\%} divide\_size$ elements in the last group.
2. Find median of each group using insertion sort and make an $\frac{n}{divide\_size}$ size array
3. Do steps 1 and 2 till we get a single value.

The complexity of the Median Of Median

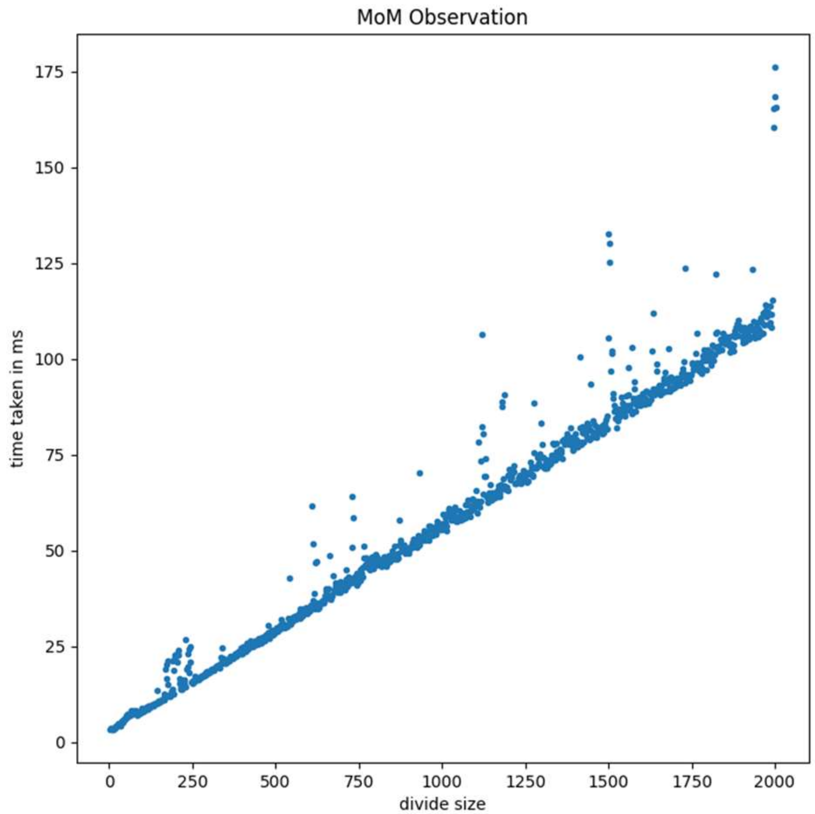
Time Complexity: $O(n)$ for worst case

Question 8

TAKE DIFFERENT SIZES FOR EACH TRIVIAL PARTITION (3/5/7 ...) AND SEE HOW THE TIME TAKEN IS CHANGING.

Procedure

- We did Median of Median for fixed array size 10^5, but varied the divide size in $\{3,5,7,9, \dots, 2003\}$
- For each divide size, we computed MoM for 10 different arrays, generated randomly (Uniform) with number in range $(0,100)$. Recorded Time Taken per Pass, then took their mean.
- We then Plotted the graph of time vs divide size



Observation

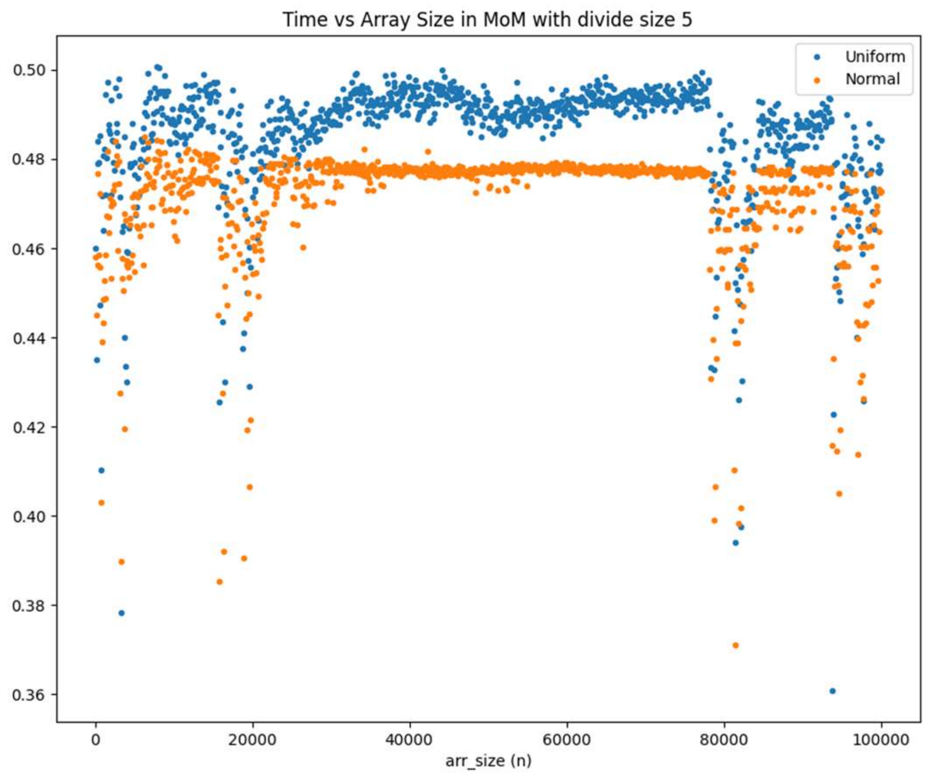
- From the graph we can infer that time complexity is Linear with Divide Size

Question 9

PERFORM EXPERIMENTS BY REARRANGING THE ELEMENTS OF THE DATASETS (BOTH UD AND ND) AND COMMENT ON THE PARTITION OR SPLIT OBTAINED USING THE PIVOTAL ELEMENT CHOSEN AS MOM.

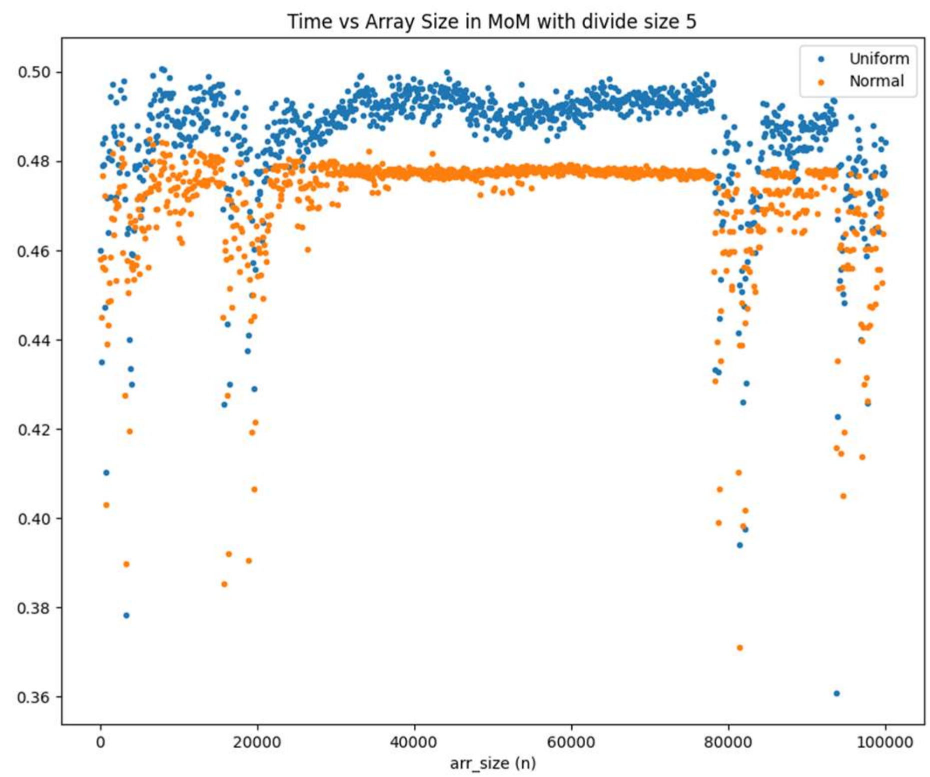
Procedure

- We did Median of Median for divide size 5, but varied the size of array from 100 to 10^5, with increments of 100
- For each array size, we computed MoM for 10 different arrays, chosen randomly form the datasets generated at question 1. used Partition() with MoM as pivot element, and recorded the output position, then took their mean.
- We then Plotted the graph of $\frac{partition}{arr\_size}$ vs arr_size for both datasets.



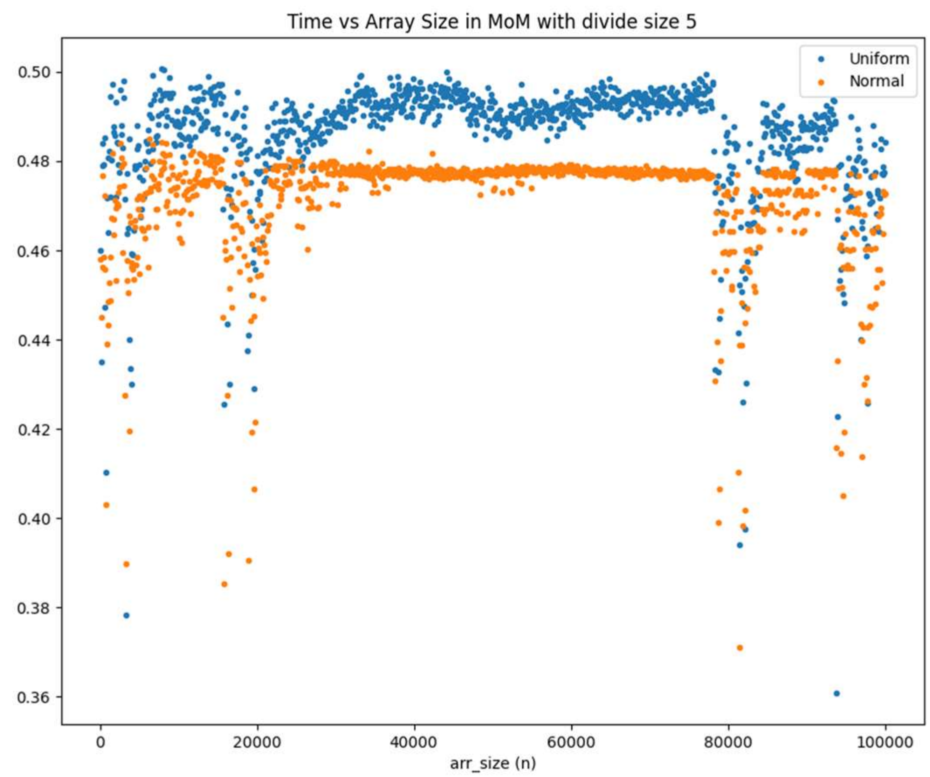
Observation

- As we can see the $\frac{partition}{arr\_size}$ varies in range $(0.36, 0.5)$ with majority of cases around 0.5.
- This shows that what MoM finds is very close to actual median, and hence MoM can mostly guarantee a good pivot of Quicksort Algorithm.



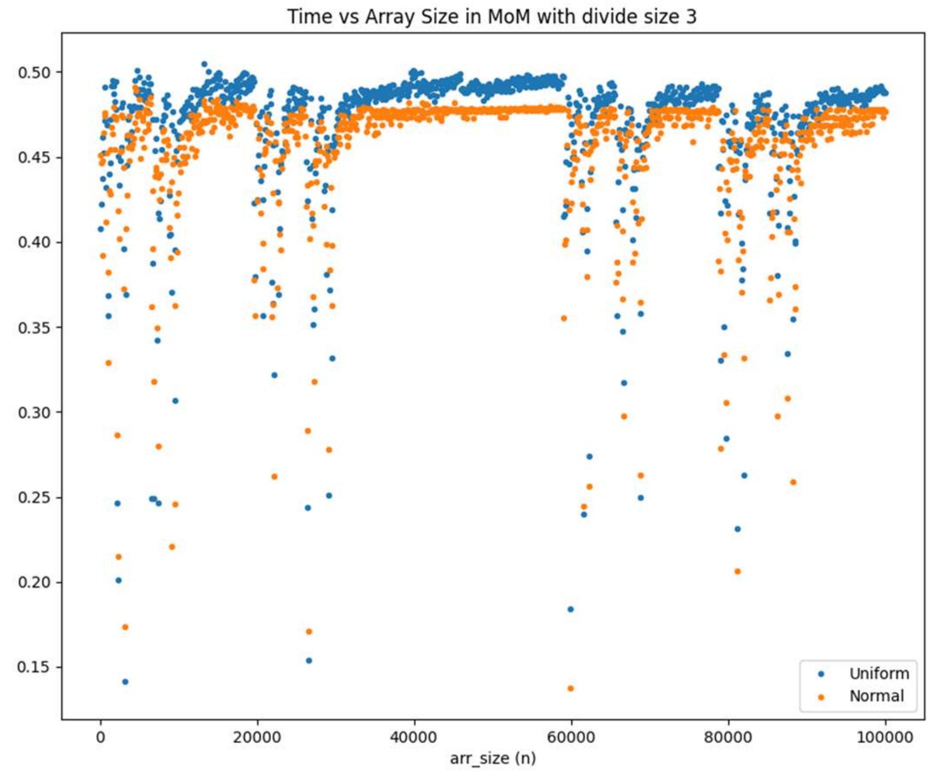
Peculiar Observation

- We observed two peculiar observations
  - There is a strange dip in the ratio in some places.
  - We also see a stark difference between Uniform and Normal Distribution



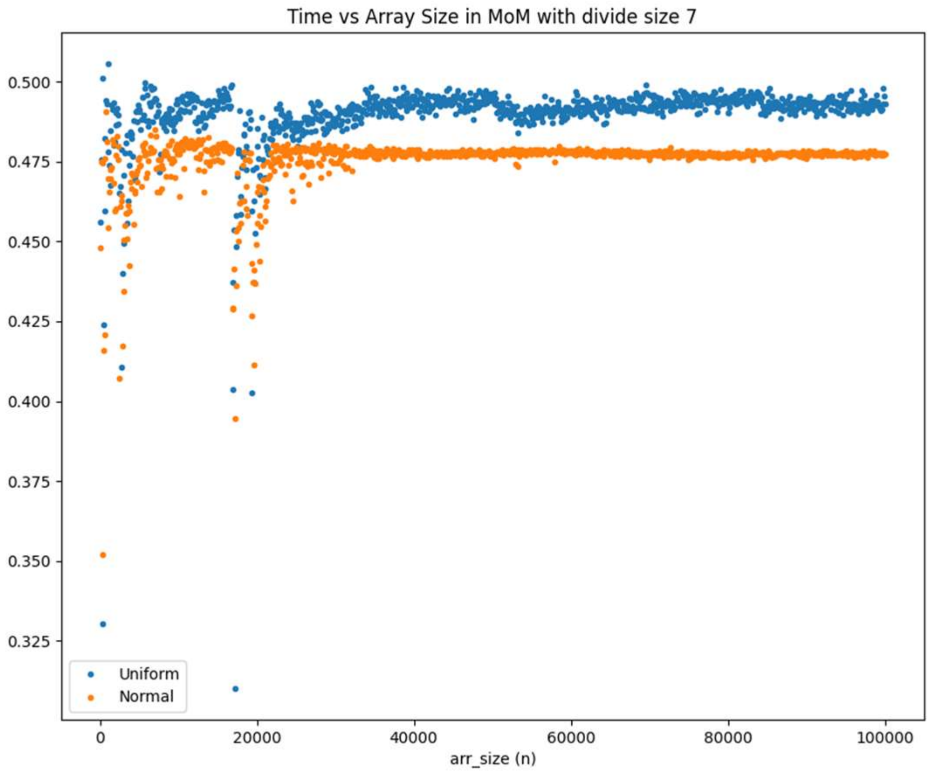
Uniform vs Normal

- The Stark difference between Uniform and Normal could be because MoM doesn't give exact median but approximates the median. And as Normal Distribution has more elements concentrated in the median; we will have decreased accuracy in case of normal
- To test this hypothesis, we did the same experiment with divide size 3 and 7
- If our hypothesis comes out to be true, the difference between uniform and normal should increase for divide size 7 and should decrease for divide size 3.



Divide Size 3

- Here we see that if we decrease the divide size to 3, the difference between Uniform and Normal Dataset decreases quite a lot.
- This is happening as the region of approx. median has decreased, giving better accuracy for Normal Distribution.



Divide Size 7

- Here we see that the difference between the Uniform and Normal is like divide size 5, but the drop region has gone away.
- By this observation, we could say that we might getting the “drops” as out divide size is very small and increasing the divide size might potentially remove the “drops”.

The End