Semester 04

Design and Analysis of Algorithm(CS2271)

Assignment 01:-

Group Members:-

1. Anish Banerjee (2021CSB001)
2. Joyabrata Acharya(2021CSB011)
3. Sk Fardeen Hossain(2021CSB023)

Source Codes and Documentations are available at Github

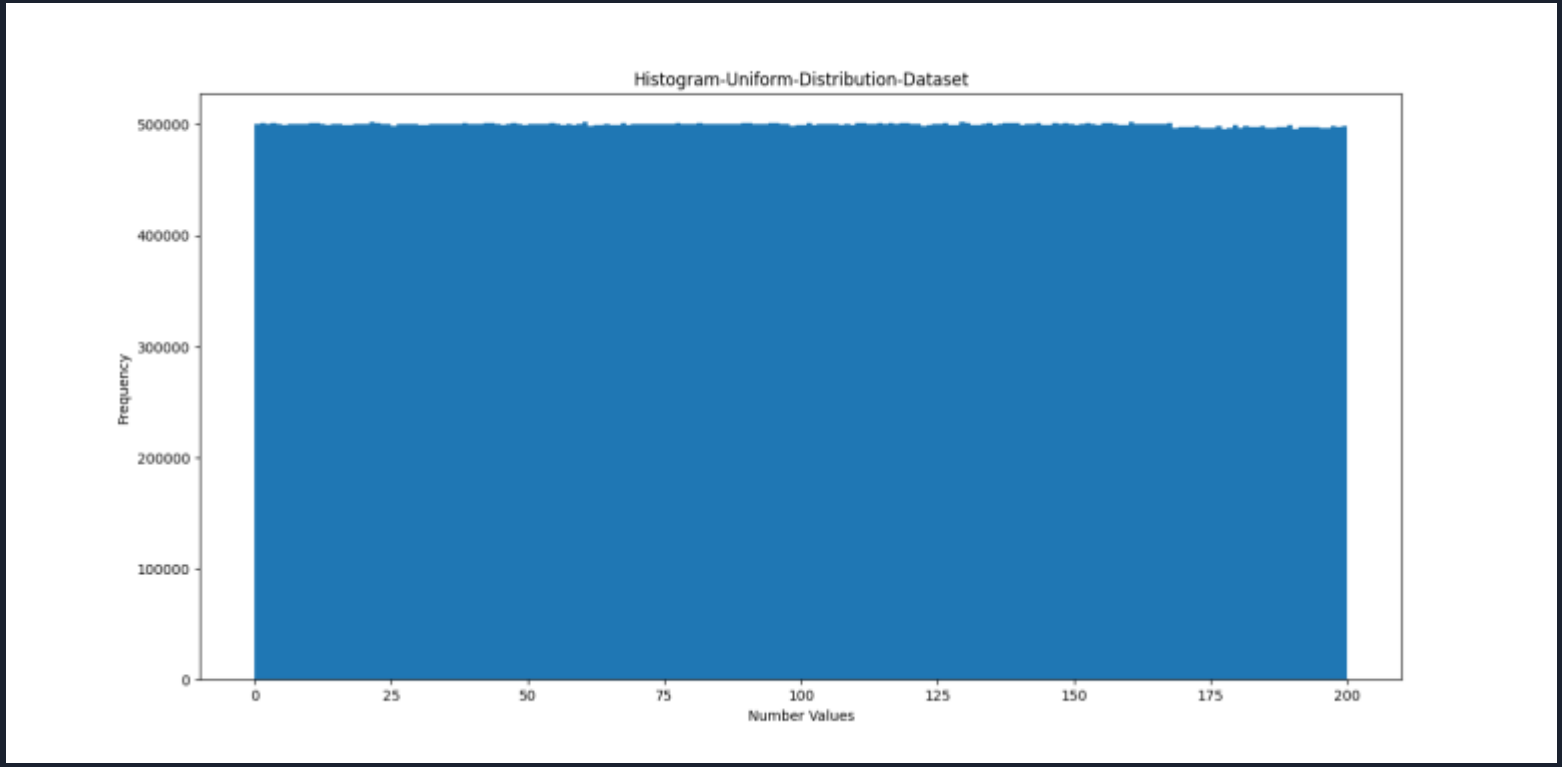
Indian Institute of Engineering Science and Technology, Shibpur

SL. NO.	TOPIC	SUBTOPICS
1	DATASET GENERATION	Uniform Distribution
		Normal Distribution
2	IMPLEMENTATION OF VARIOUS SORTING ALGORITHMS	Merge Sort
		Quick Sort
		Randomised Quick Sort
		Normalisation of Dataset, Bucket Sort
3	MASTER PROGRAM : GRAPHICAL ANALYSIS	Average Number of Comparisons
		Time Complexity
4	WORST CASE LINEAR MEDIAN SELECTION	Median of Medians
		Divide Size Analysis
		Partition Size Analysis

Introduction

- Sorting is a fundamental operation in Computer Science, and its extensive use is found in various applications such as database systems, search engines and scientific computing.
- In this study, several sorting algorithms and distributions were explored to analyse their performance and complexity.
- Large datasets were generated using two different distributions-uniform and normal, and then merge and quick sort were applied to them. Further quick sort was modified to make it randomised and datasets were normalised so as to apply bucket sort to them analysing the performance and correctness. The worst case linear median selection algorithm was implemented using median of medians approach and further partition size analysis was done taking median of medians as the pivotal element. A master program was also implemented to have an overview of all the sorting algorithms done in this study.

1-A: Construct large datasets taking random numbers from uniform distribution (UD)



Uniform Distribution of 10e7 numbers in the range of 0 to 200

Procedures to Generate the Dataset:-

- We will use the inbuilt C rand() function to generate random numbers uniformly. We setup the random seed using <time.h> module and print the random numbers obtained in "uniform_distribution.csv" file.
- The numbers can be generated via the formula:

  Number=rand()%MAX;// MAX is 200 in our case

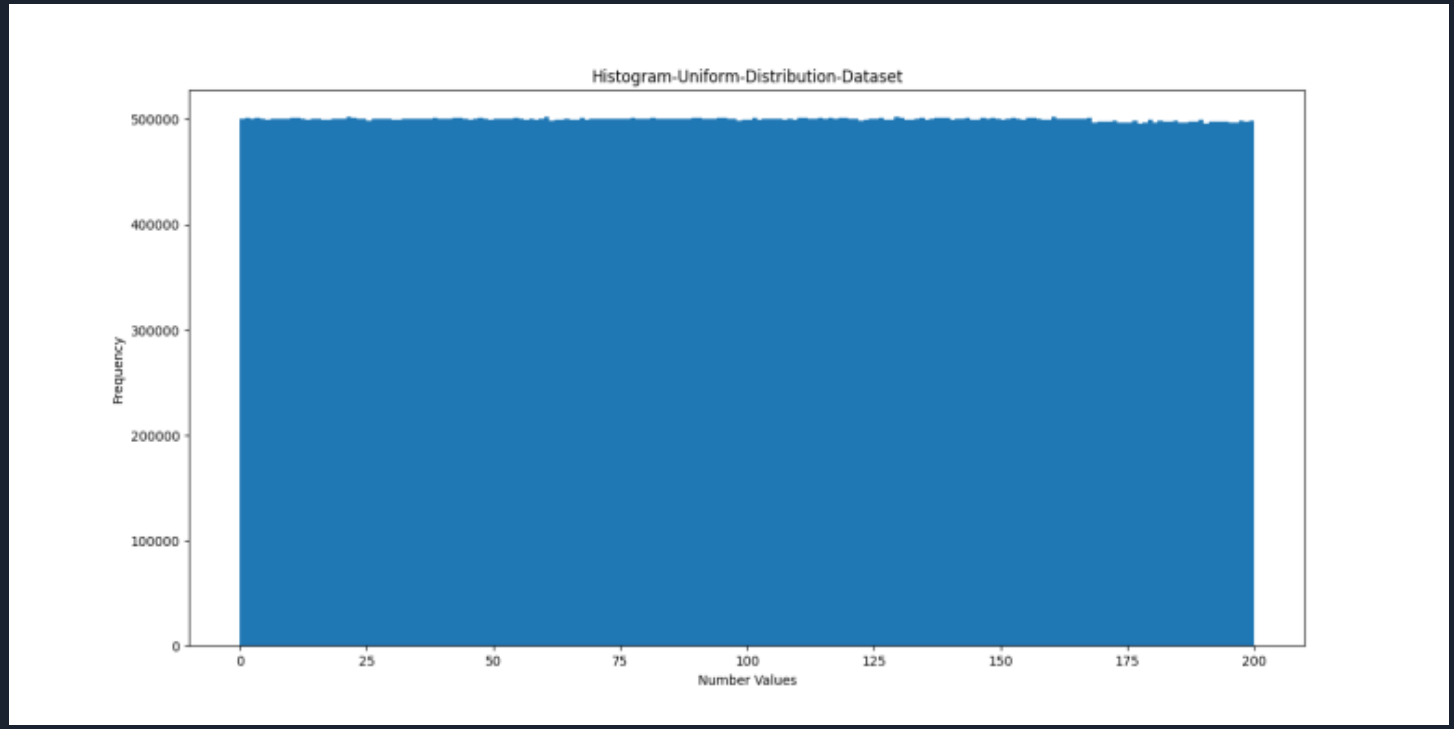
- We then plotted the histogram of the numbers generated and observe.

Code Snippet:

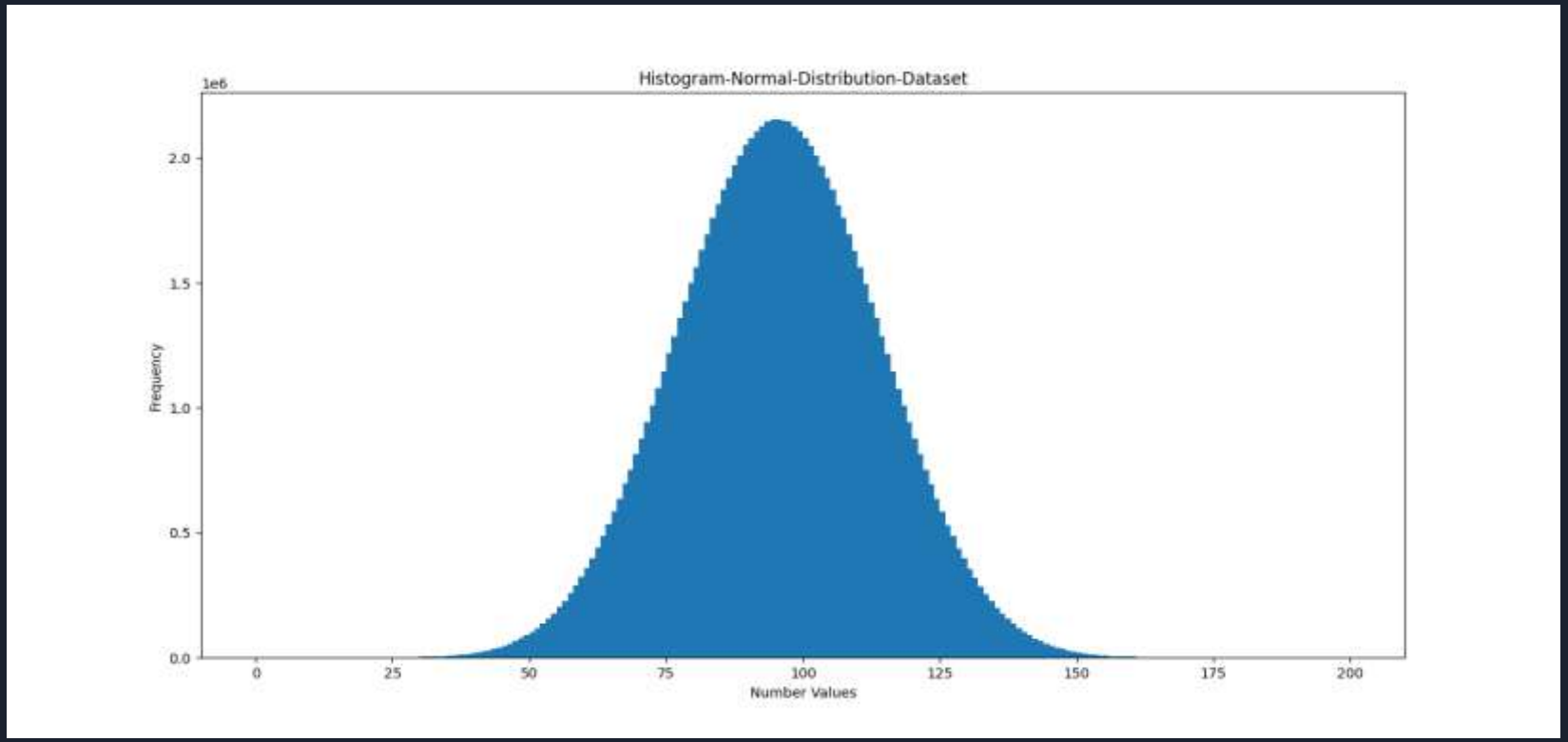
```c
const long long TOTAL = 10e7;
const int MAX = 200;

int main()
{
  /*For generating random numbers*/
  srand(time(0));

  /*Writing the dataset into a csv file*/
  FILE *fp = fopen("uniform_distribution.csv", "w");
  for (int i = 0; i < TOTAL; i++)
  {
    fprintf(fp, "%d\n", rand() % MAX);
  }
  printf("The Uniform Distribution Dataset has been successfully generated.\n");
  fclose(fp);
  return 0;
}
```

Observation:

We can see that the dataset has been generated successfully and every number in the range 0 to 200 has almost same number of occurrences(frequency) which was to be observed in a uniform dataset.



1-B: Construct large datasets taking random numbers from normal distribution (ND)



Normal Distribution of 10e7 numbers in the range of 0 to 200

Procedures to Generate the Dataset:

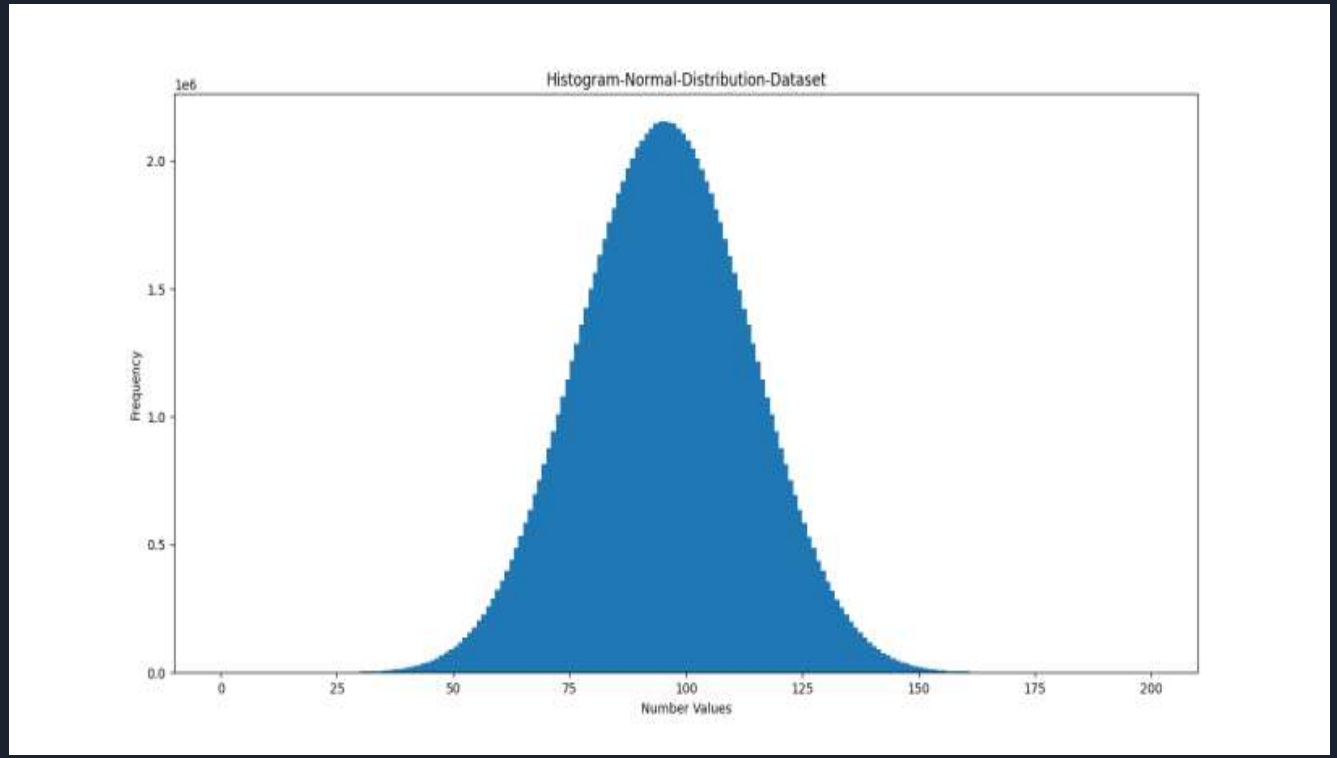
- The intuition to generate a normal distribution dataset is to know that the frequency of values at the extremes of the range must be low and those near the mean must be high.
- Now we just take the sum of 10 uniformly distributed each numbers between 0 and 10 for 20 rounds and print it onto the dataset such that the min. sum will be 0 and the max sum will be 200 but the frequency of sums near 100 will be appreciably higher.
- We did the same procedure 10^8 times to generate a dataset normal_distribution.csv consisting of 10^8 normally distributed numbers.
- Then we plot the values in the histogram and check for correctness of the generated dataset.

Code Snippet:

```c
int main()
{
  /*For generating random numbers*/
  srand(time(0));

  /*Writing the dataset into a csv file*/
  FILE *fp = fopen("normal_distribution.csv", "w");
  for (int i = 0; i < TOTAL; i++)
  {
    int sum = 0;
    for (int j = 0; j < SUB_ELEMENTS; j++)
    {
      sum += rand() % (MAX / SUB_ELEMENTS); // This ensures the frequency near
    }
    fprintf(fp, "%d\n", sum);
  }
  fclose(fp);
  printf("The Normal Distribution Dataset has been successfully generated.\n");
  return 0;
}
```

Observation

- We can see that the plot shows the natural bell curve a characteristic of normalised data thus ensuring that the dataset generated gives normal distribution.



2-A: Implement Merge Sort (MS) and check for correctness

- Merge Sort is a comparison-based sorting algorithm that works on the divide and conquer paradigm. There are 3 stages in a merge sort:
- Divide: n-element sequence into n/2 elements into 2 subsequences and this will take O(1) time
- Conquer: We sort the two subsequences recursively.
- Combine: We then combine the sorted subsequence into a sorted sequence; takes O(N) time

The method gives the following recurrence relation:

T(N)=2*T(N/2)+O(N); T(N) is the time taken to sort an array of size N.

The Time complexity obtained after solving the relation is O(NlogN) which is both the best case and average case time complexity.

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) ;

          }

}
```

You can check for correctness of code at :- MergeSort

2-B: Implement Quick Sort (QS) and check for correctness

- Quick Sort is also a Divide and Conquer algorithm but unlike merge sort in quick sort the conquer time is constant while the divide time is almost linear. Unlike Merge Sort which has a deterministic time complexity, quick sort algorithm has probabilistic time-complexity that is the choice of pivot element at each step in quick sort drives the time complexity.
- The pivot element can be chosen randomly, first element, last element or median of three and many more and each of the approach gives different tweaking in the time complexity of O(N*logN).
- The method in each step divides the array into 3 parts: 1st part contains all those elements less than pivot value, 2nd part contains only the pivot value and the last part contains all those elements more than pivot value. The pivot value in each step gets its position fixed as that will be its position in the sorted array

Quick Sort Algorithm:

```c
// Inplace Quick Sort
int partition(int *arr, int initial, int final)
{
  // Taking initial element as the pivot element
  int pivot_value = arr[initial];
  int left = initial;
  int right = final;

  while (1)
  {
    while (arr[left] < pivot_value)
      left++;

    while (pivot_value < arr[right])
      right--;

    if (arr[left] == arr[right])
    {
      if (left == right)
        return left;
      else
        right--;
    }
    else if (left < right)
      swap(&arr[left], &arr[right]);
    else
      return left;
  }
}
```

```c
void quickSort(int *a, int start, int end)
{
  if (start < end)
  {
    int pivot_index = partition(a, start, end);
    quickSort(a, start, pivot_index);
    quickSort(a, pivot_index + 1, end);
  }
}
```

You can check for correctness of code at :- QuickSort

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

- For each of the sorting algorithms we analyse two min areas:
  1. Average Number of Comparisons
  2. Average Time Taken
- The range of the size of the array was kept between 2 and 2^18 and are incremented in increasing powers of 2
- For each array size we take 50 rounds of the same array size and calculate the average time taken and the average number of comparisons taken
- We then plot the comparison and time ratio by dividing the average comparisons and time taken by N*logN N being the size of the array.

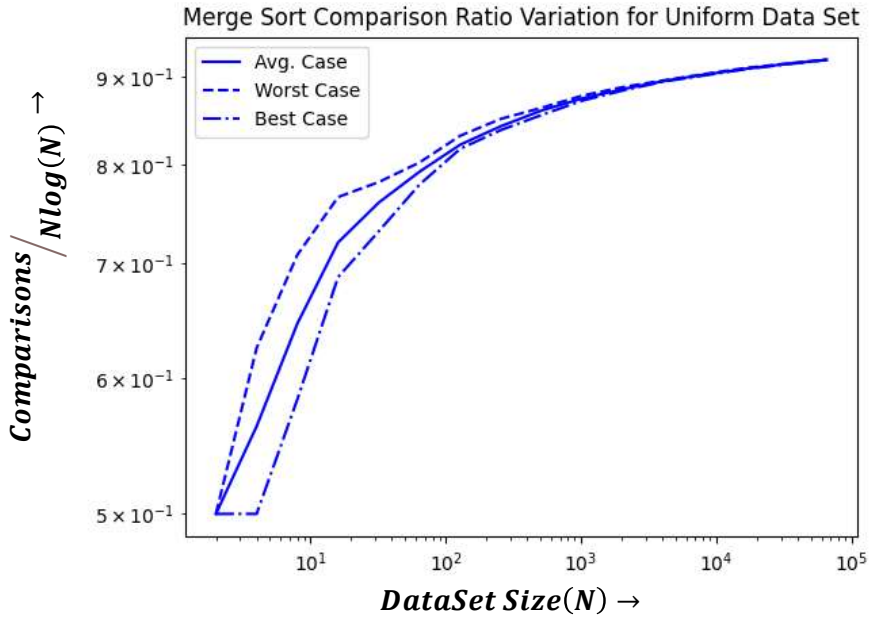
A. Merge Sort and Quick Sort

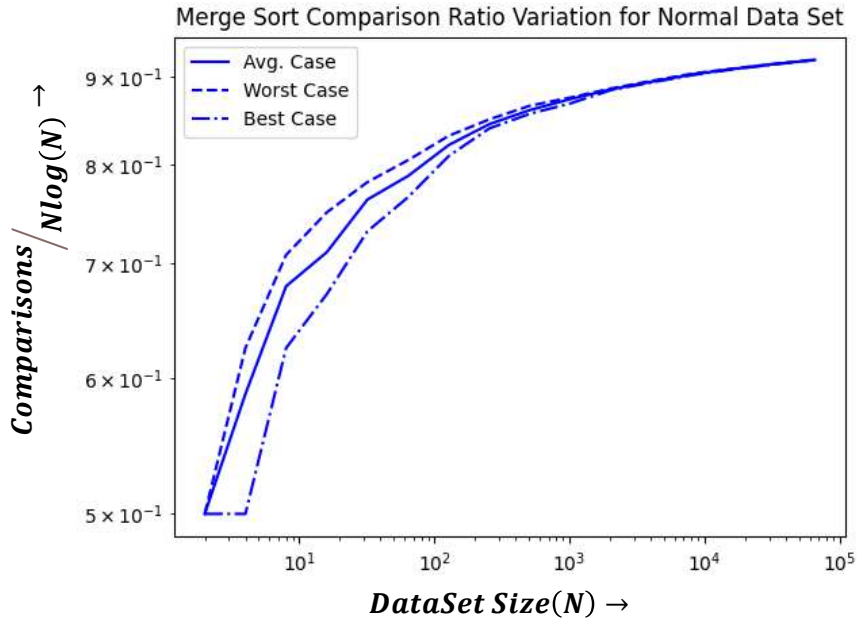
Code Snippets

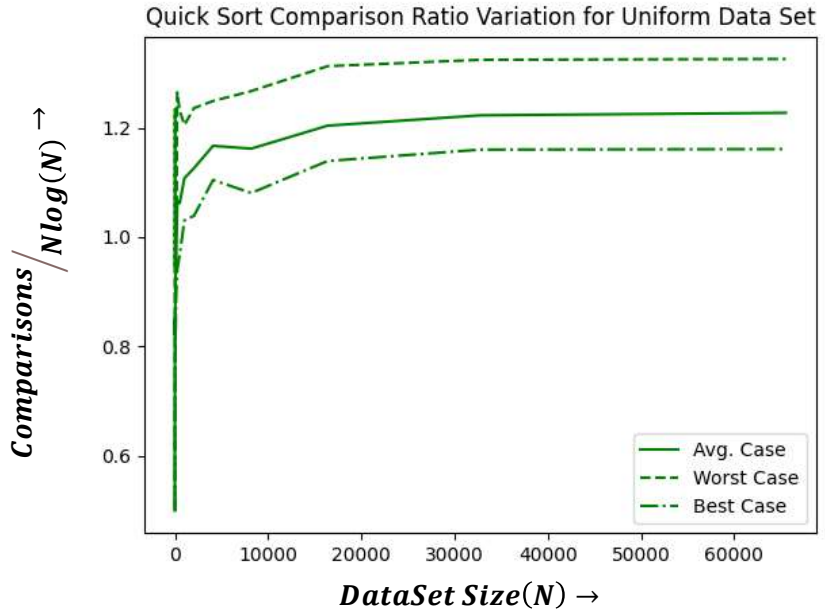
```
/*---Merge Sort---*/
void merge(double *merge, double *arr1, double *arr2, ui n1, ui n2)
{
    ui i=0,j=0,k=0;

    while (i < n1  && j < n2)
    {
        merge[k++] = ((arr1[i]<arr2[j]) ? arr1[i++] : arr2[j++]);
        (*comp)++;
    }

    while(i < n1)
          merge[k++] = arr1[i++];
    while (j < n2)
          merge[k++] = arr2[j++];
}
void mergeSort(double * arr,ui n,ui *comp)
{
    if(n < 2) return;
    ui mid = n >> 1;
    double larr[mid],rarr[n-mid];
    for (ui i = 0; i < mid; i++)
        larr[i] = arr[i];
    for (ui i = 0; i < n-mid; i++)
        rarr[i] = arr[i+mid];
    mergeSort(larr,mid,comp);
    mergeSort(rarr,n-mid,comp);

    merge(arr,larr,rarr,mid,n-mid,comp);
}
```

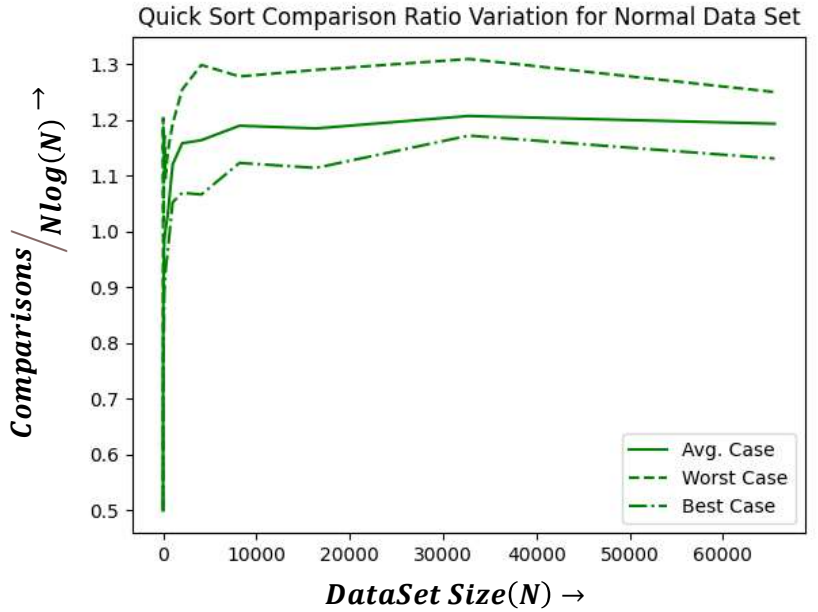
```
/*---Normal QS---*/
int partitionQS(double * arr, int low, int high,ui *comp,ui *swaps)
{
    double pivot = arr[high];
    int i = low;
    for (int j = low; j < high; j++)
    {
        if (arr[j] < pivot)
        {
            swap(&arr[i], &arr[j]);
            (*swaps)++;
            i++;
        }
        (*comp)++;
     }

    swap(&arr[i], &arr[high]);
    (*swaps)++;
    return i;
}
void quickSort(double * arr, int low, int high,ui *comp)
{
    if(low < high)
    {
        int pi = partitionQS(arr,low,high,comp,swaps);
        quickSort(arr,low,pi-1,comp,swaps);
        quickSort(arr,pi+1,high,comp,swaps);
    }
}
```

i. Merge Sort and Quick Sort Comparison Ratio Observations:

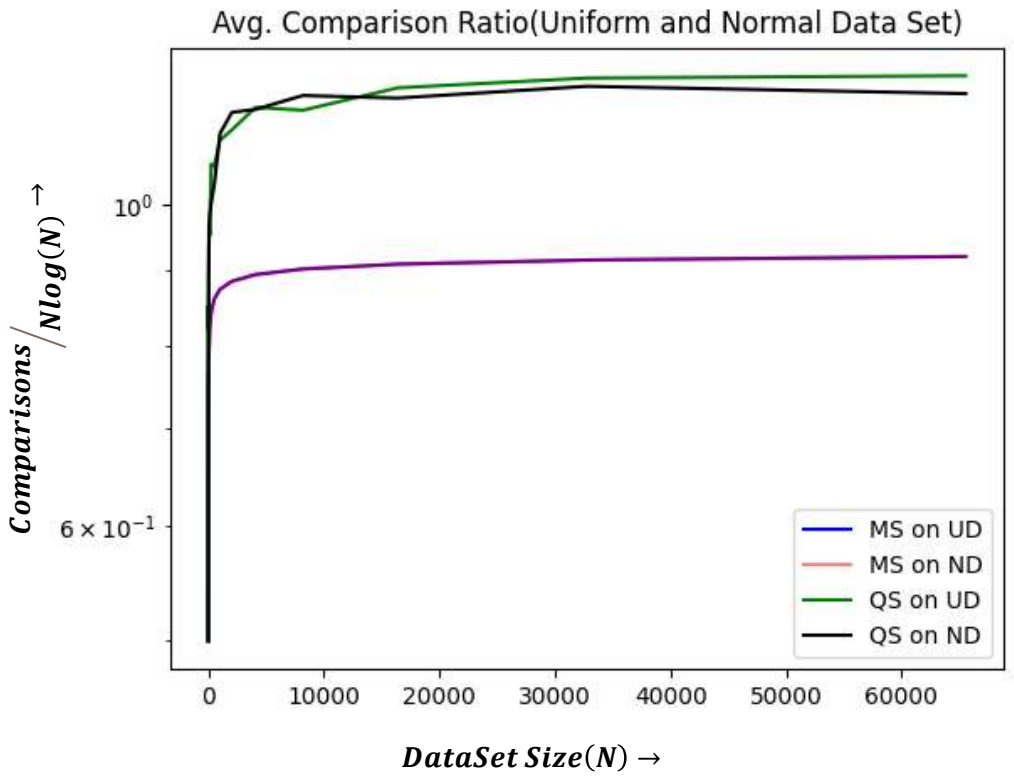








ii. Merge Sort vs Normal Quick Sort



Observation:

Around and beyond $N = 10000$,

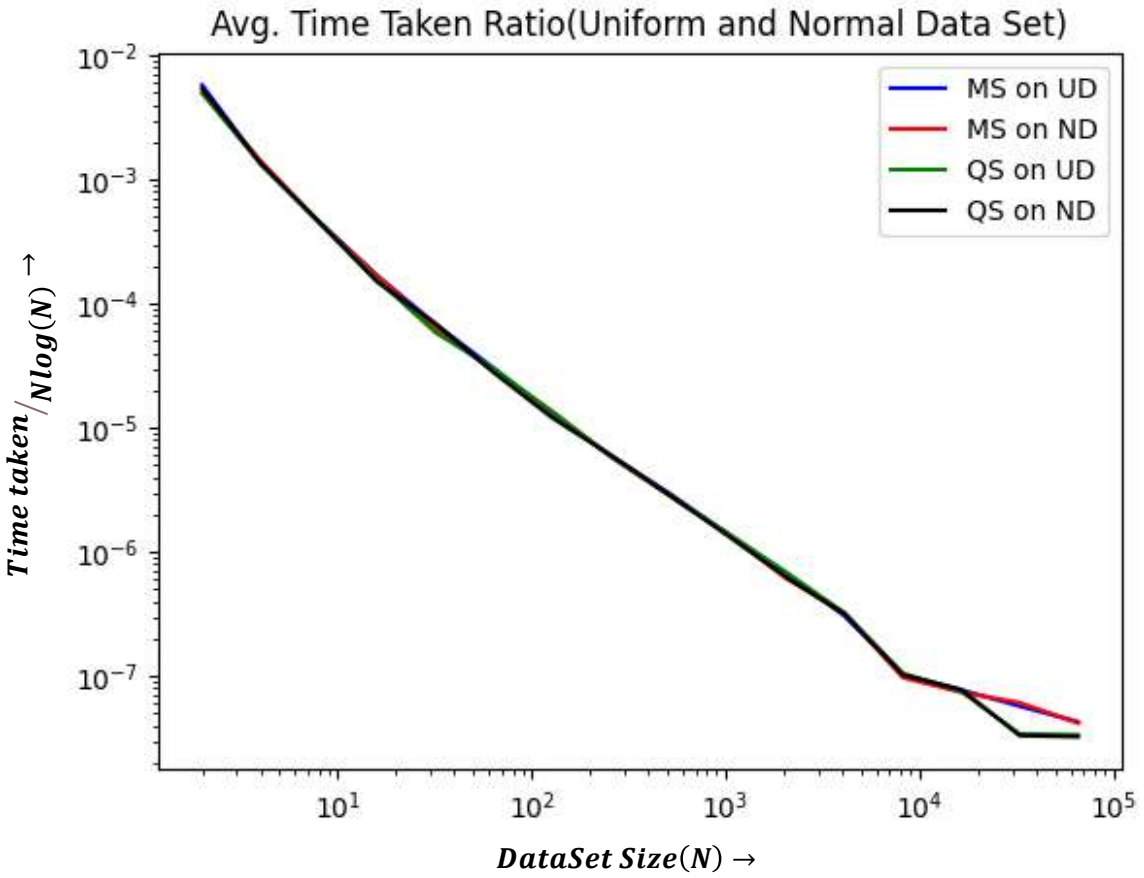
- $Avg\ comparison\ ratio$ for Merge Sort < 1
- $Avg\ comparison\ ratio$ for Quick Sort > 1 but < 1.5
- The graph flattens out in both cases implying that the ratio approaches a constant value.

Inference:

- The $Merge\ Sort$ has a tight upper-bound of N$logN$ that is it has an ***average case time complexity*** of $\boldsymbol{O(NlogN)}$

- The $Quick\ Sort$ has an ***averagecase time complexity*** of $\boldsymbol{O(NlogN)}$. The ratio, however is slightly above 1.

iii. Time Taken Ratio Analysis



Observation:

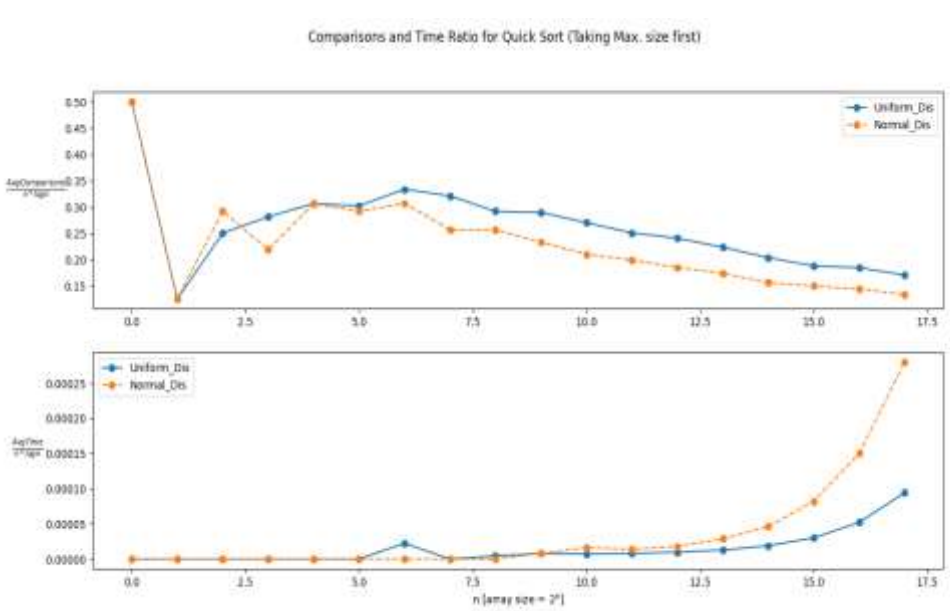
- The change in ratio is nearly identical for both Merge Sort and Quick Sort for both uniform and normal distributions.
- The ratio steadily decreases 10^{-2} to below 10^{-7} as the size increases.

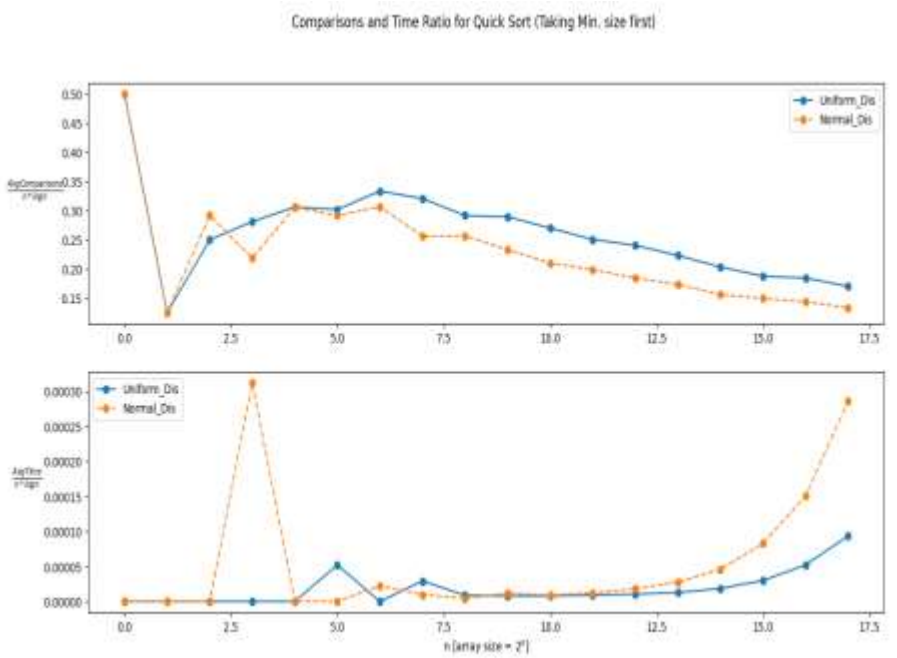
Inference:

- There is no appreciable difference in the behaviour of either sorting functions on either type of data sets.

Analysis for Quick Sort preferrential sorting

- Here we analyse the case when we sort the min or max size part first after partitioning the array in each turn. So what we do is to just tweak the quickSort part and quickSort the max or min size first depending on the choice.





As we can observe that sorting min size or max size first does not change the trend of the quick sort algorithm and it can be inferred that preferrentially sorting in this manner will give same average number of comparisons.

4. Experiment with randomized QS (RQS) with both UD and ND as input data to arrive at the average complexity (count of operations performed) with both input datasets.

- The idea of a randomised algorithm is the use of random numbers to decide what to do anywhere it's next logic. In a randomised Quick Sort Algorithm we use a random number in a given range as the pivot value.

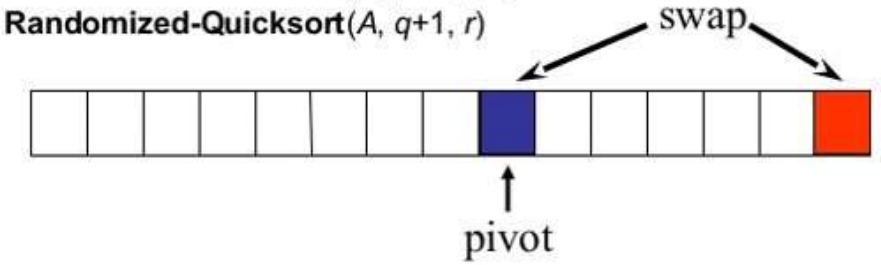
Randomized Quicksort

Randomized-Partition(*A, p, r*)

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

Randomized-Quicksort(*A, p, r*)

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



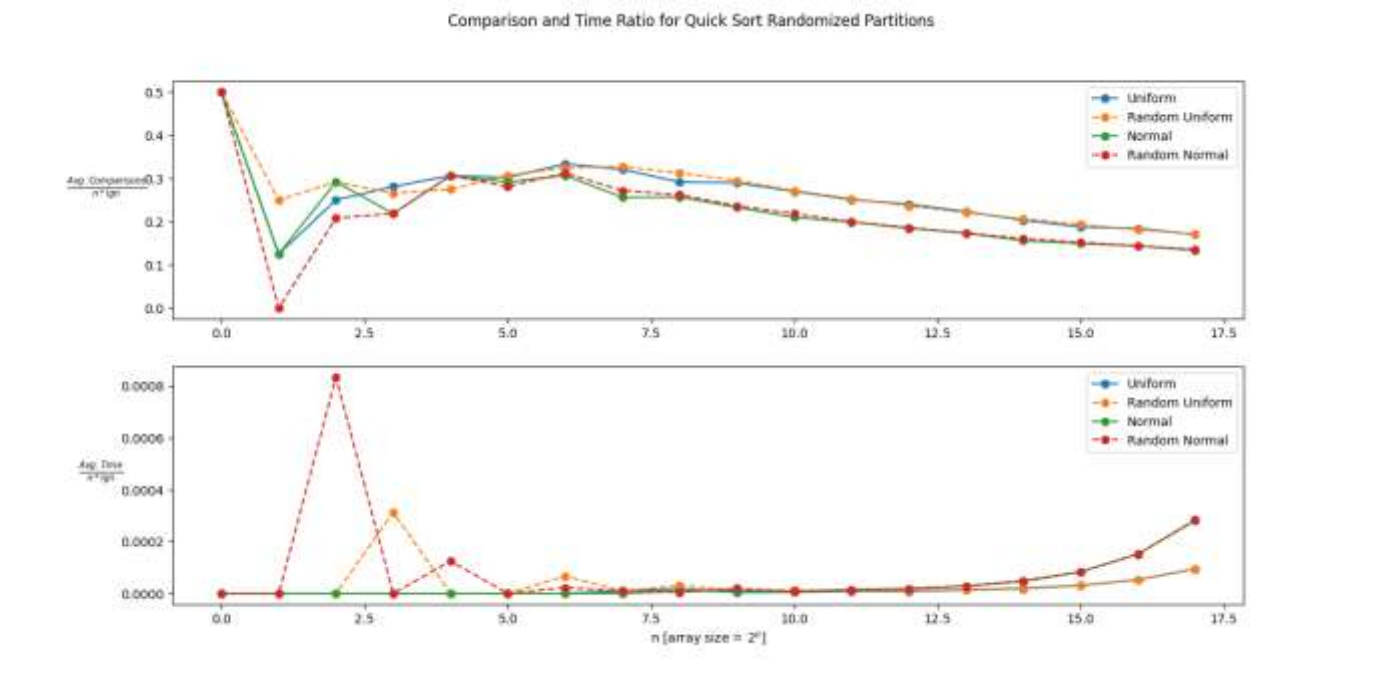
Time Complexity

- Being a randomised algorithm, the randomised quick sort algorithm has expected time complexity of O(nlogn) whereas the worst case time complexity is also same. In the worst case domain the pivot element to be picked is always the extreme points.

Procedure:

- We perform RQS for different size array starting from size 2 to 2^18 with size increasing in powers of 2 in each turn.
- For each size we perform 50 rounds of RQS and check for correctness keeping track of the average time taken and the number of comparisons.
- We then plotted the avg time taken and the number of comparisons by taking a ration with nlogn, n being the size of the array.

Observation



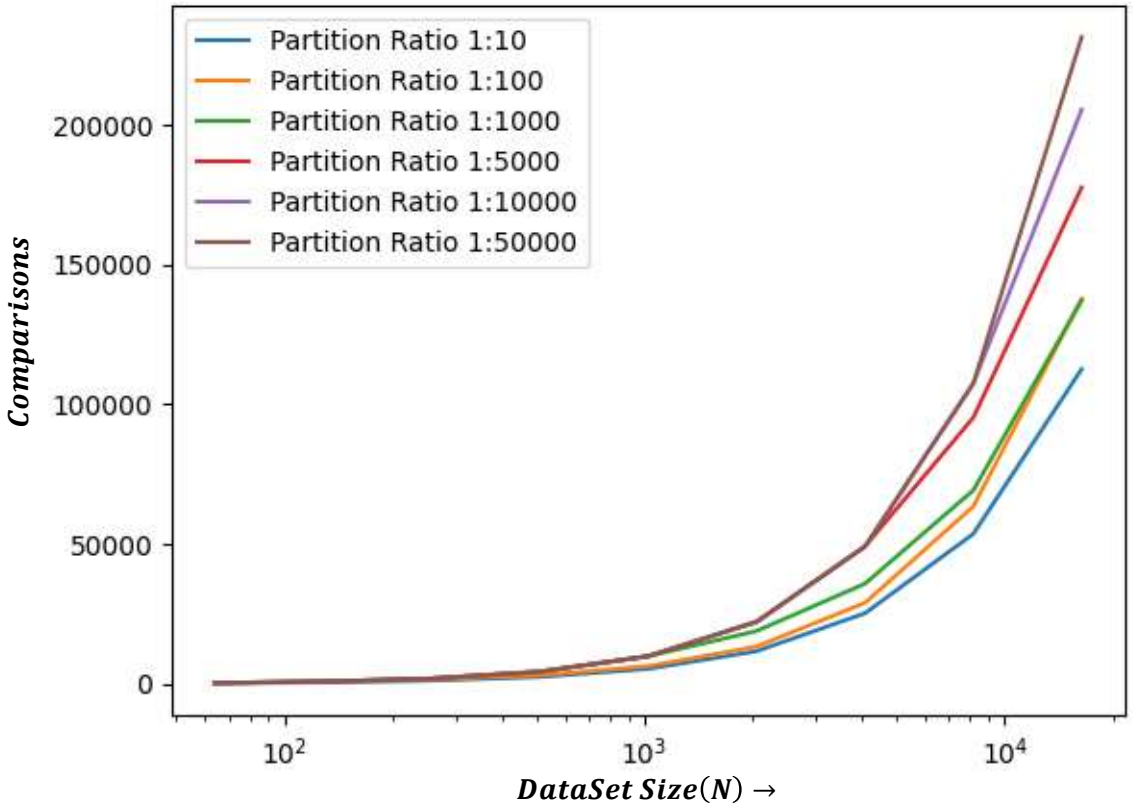
- As the array size keeps on increasing the comparison and time ratio settles to a stable value, thus making the average and worst case time complexity to be same in this case.
- The near same semblance to the normal quick sort may be due to the fact that the random pivot goes for a 1:1 partition. To get a perspective on the randomness we can do deliberate partitions and compare it with randomised partitions.

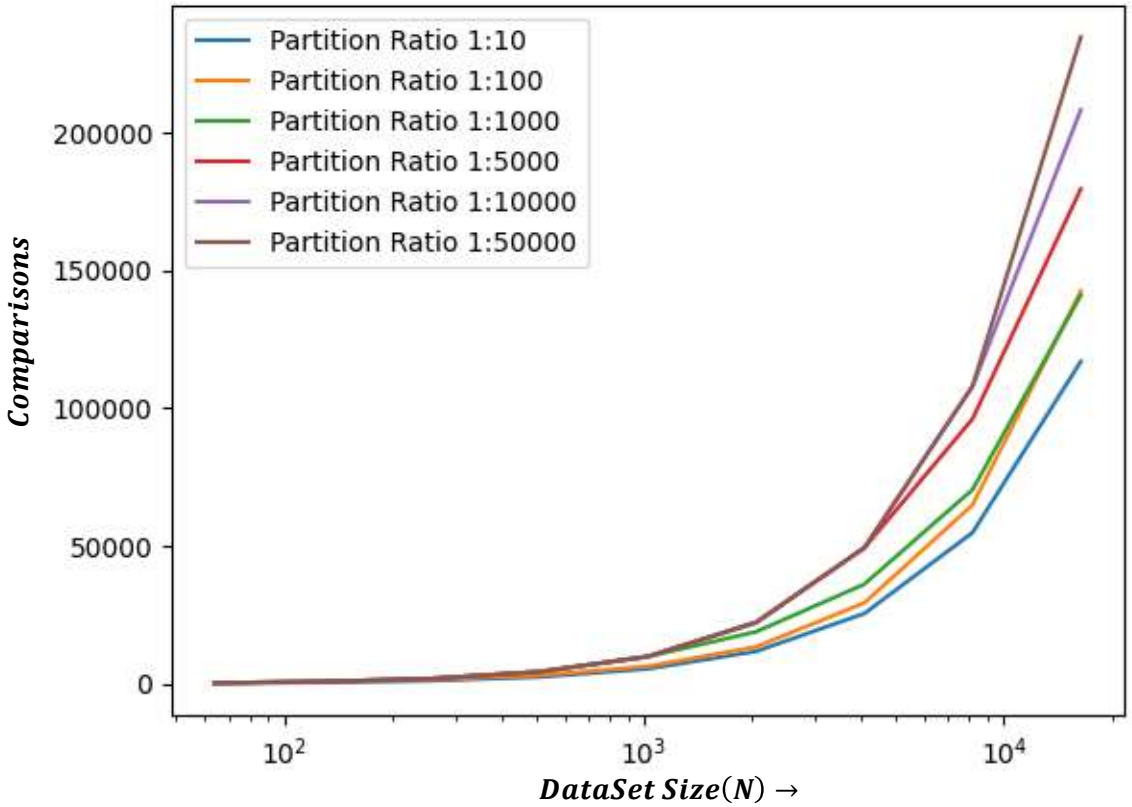
ii. Deliberate Parition Quick Sort

```
int partition(double *arr, int low, int high, int *comp)
{
    double pivot_value = arr[low];
    int i = low;
    int j = high;

    while (1)
    {
        while (arr[i] < pivot_value)
            i++;

        while (pivot_value < arr[j])
            j--;

        if (arr[i] == arr[j])
            if (i == j)     return i;
            else    j--;

        else if (i < j)
        {
            swap(&arr[i], &arr[j]);
            (*comp)++;
        }
        else   return i;
    }
}

int randomized_partition(double *arr, int low, int high, int
*comp)
{
    int i = rand() % (high - low) + low;
    swap(&arr[low], &arr[i]);
    return partition(arr, low, high, comp);
}
```

```
int randomized_select(double *arr,int low,int high,int find_pos, int *comp)
{
    if (low == high)    return low;

    if (find_pos < low  || find_pos > high)
    {
        printf("Error: Less\n");
        exit(0);
    }
    int q = randomized_partition(arr, low, high, comp);
    int k = q + 1;

    if (find_pos < k)
        return randomized_select(arr, low, q, find_pos, comp);
    else
        return randomized_select(arr, k, high, find_pos, comp);
}
void quick_sort_ratio(double *arr, int low, int high, int *comp, int ratio)
{
    if (low < high)
    {
        int pivot_pos = randomized_select(arr,low,high,low+(high-
low)/ratio,                         comp);
        quick_sort_ratio(arr, low, pivot_pos, comp, ratio);
        quick_sort_ratio(arr, pivot_pos + 1, high, comp, ratio);
    }
}
```

i. Effect of Growth in Comparison Quick Sort Comparison Ratio Observations

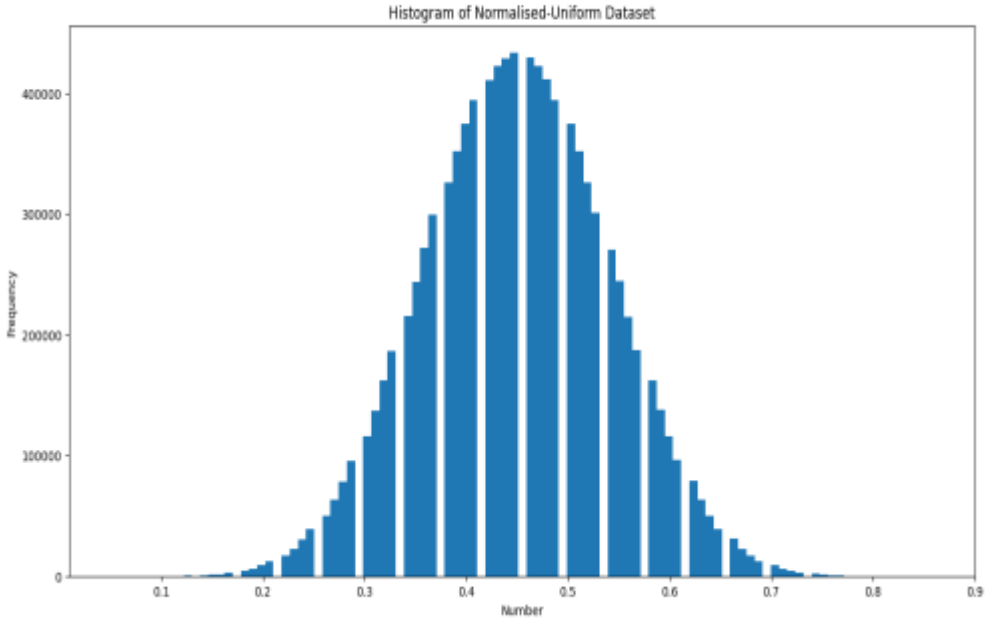


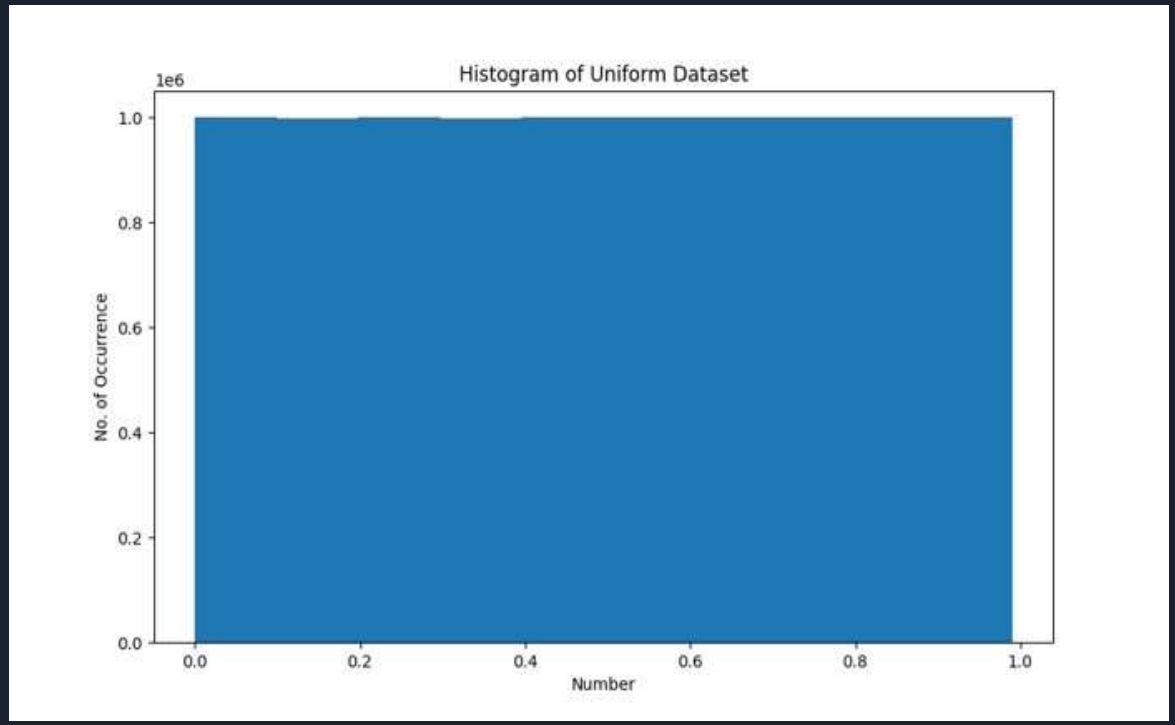


5. Now normalize both the datasets in the range from 0 to 1 and implement bucket sort (BS) algorithm and check for correctness.

- To normalise a dataset between 0 and 1 we just need to divide all the elements in the corresponding dataset by the max value of the range(200).
- We then save the normalised dataset into another .csv file and then plot the histograms of the dataset to check for correctness.

Observation

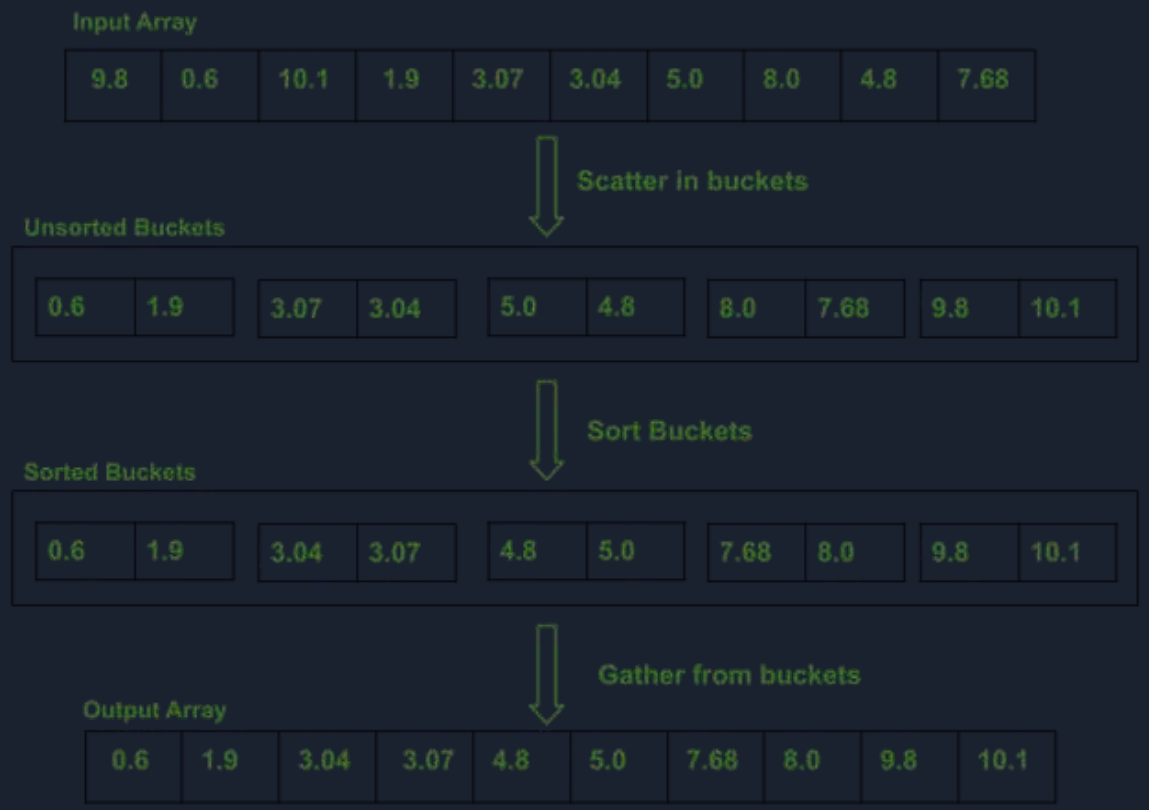




6. Experiment with BS to arrive at its average complexity for both UD and ND data sets and infer.

- 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.
- It is not a comparison based sort unlike QS or MS.

Bucket Sort in Action

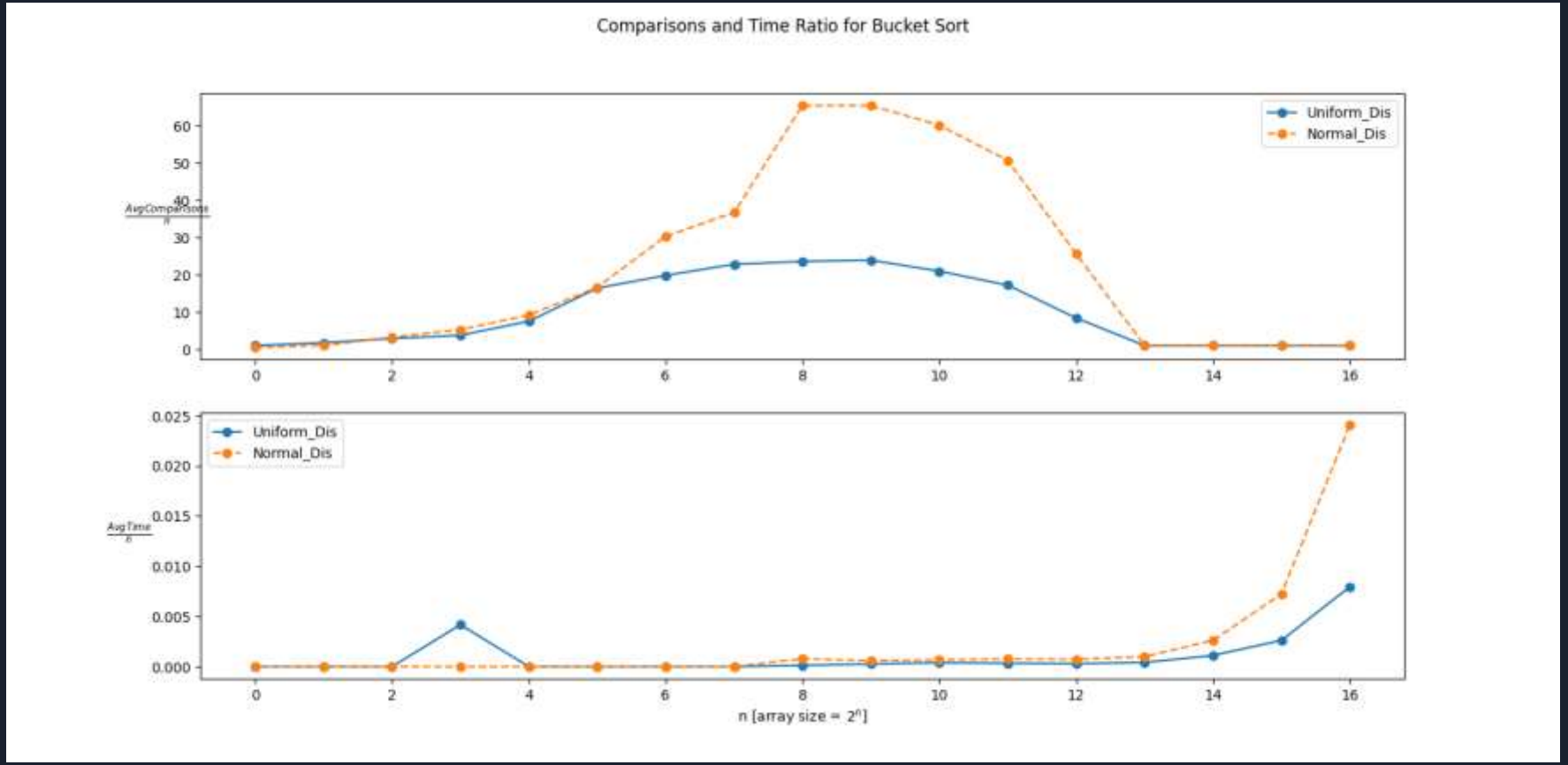


- The time complexity of bucket sort is O(n+k) for best case and average case while O(n^2) for worst case K being the average size of the bucket.
- The space complexity of bucket sort is O(nk) k being the average size of bucket

Procedure

- We implement the bucket sort algorithm using a array based linked list implementation where each element of the array is a bucket manifested in the form of a linked list.
- We apply the algorithm for the normalised dataset with the size of array increasing in powers of 2 and each size takes 15 rounds.
- We then compare the average time taken and average number of comparisons with n, n being the size of the array.

Observation



7. Master Program

- TO STUDY THE COMPARISON BETWEEN MERGE SORT, QUICK SORT, RANDOMISED QUICK SORT AND BUCKET SORT ON THE BASIS OF AVERAGE TIME TAKEN AND AVERAGE NUMBER OF COMPARISONS INVOLVED. GRAPHICAL INTERPRETATION HAS BEEN DEMONSTRATED

Procedure

- The previously mentioned implementations have been utilised to generate a master comparison program in C-language. Both Uniform and Normal Datasets of sizes varying 2 1 to 2 16 have been subjected to the 4 sorting algorithms separately. For the randomised Quick Sort, the same dataset was iterated 10 times while for the others, it was 10 various combinations of the datasets. Ultimately, the average time taken and number of comparisons have been plotted with respect the theoretically known complexities, i.e. ’n log2 n’ for Merge, Quick and Randomised Quick Sort, and ’n’ for Bucket Sort.
- The results have been generated in Master Text Files (both in detailed and easy viewable format). Respective graphs have been plotted.

Master File results

Time in mlliseconds

```
RESULTS FOR DATASET SIZE 2:

MERGE SORT: UNIFORM DATASET
Average Number of comparisons: 1
Average Time Taken: 0.0025
Constant (Comparisons): 0.5000
Constant (Time): 0.0013
MERGE SORT: NORMAL DATASET
Average Number of comparisons: 1
Average Time Taken: 0.0020
Constant (Comparisons): 0.5000
Constant (Time): 0.0010
RANDOMISED QUICK SORT: UNIFORM DATASET
Average Number of comparisons: 1
Average Time Taken: 0.0017
Constant (Comparisons): 0.5000
Constant (Time): 0.0009
RANDOMISED QUICK SORT: NORMAL DATASET
Average Number of comparisons: 1
Average Time Taken: 0.0010
Constant (Comparisons): 0.5000
Constant (Time): 0.0005
QUICK SORT: UNIFORM DATASET
Average Number of comparisons: 1
Average Time Taken: 0.0018
Constant (Comparisons): 0.5000
Constant (Time): 0.0009
QUICK SORT: NORMAL DATASET
Average Number of comparisons: 1
Average Time Taken: 0.0013
Constant (Comparisons): 0.5000
Constant (Time): 0.0006
BUCKET SORT: UNIFORM DATASET
Average Number of comparisons: 0
Average Time Taken: 0.0347
Constant (Comparisons): 0.0000
Constant (Time): 0.0174
BUCKET SORT: UNIFORM DATASET
Average Number of comparisons: 0
Average Time Taken: 0.0172
Constant (Comparisons): 0.0000
Constant (Time): 0.0086
```

```
RESULTS FOR DATASET SIZE 4:

MERGE SORT: UNIFORM DATASET
Average Number of comparisons: 5
Average Time Taken: 0.0017
Constant (Comparisons): 0.6250
Constant (Time): 0.0002
MERGE SORT: NORMAL DATASET
Average Number of comparisons: 5
Average Time Taken: 0.0013
Constant (Comparisons): 0.6250
Constant (Time): 0.0002
RANDOMISED QUICK SORT: UNIFORM DATASET
Average Number of comparisons: 4
Average Time Taken: 0.0008
Constant (Comparisons): 0.5000
Constant (Time): 0.0001
RANDOMISED QUICK SORT: NORMAL DATASET
Average Number of comparisons: 5
Average Time Taken: 0.0008
Constant (Comparisons): 0.6250
Constant (Time): 0.0001
QUICK SORT: UNIFORM DATASET
Average Number of comparisons: 5
Average Time Taken: 0.0015
Constant (Comparisons): 0.6250
Constant (Time): 0.0002
QUICK SORT: NORMAL DATASET
Average Number of comparisons: 6
Average Time Taken: 0.0010
Constant (Comparisons): 0.7500
Constant (Time): 0.0001
BUCKET SORT: UNIFORM DATASET
Average Number of comparisons: 0
Average Time Taken: 0.0155
Constant (Comparisons): 0.0000
Constant (Time): 0.0039
BUCKET SORT: UNIFORM DATASET
Average Number of comparisons: 0
Average Time Taken: 0.0148
Constant (Comparisons): 0.0000
Constant (Time): 0.0037
```

```
RESULTS FOR DATASET SIZE 8:

MERGE SORT: UNIFORM DATASET
Average Number of comparisons: 15
Average Time Taken: 0.0017
Constant (Comparisons): 0.6250
Constant (Time): 0.0001
MERGE SORT: NORMAL DATASET
Average Number of comparisons: 16
Average Time Taken: 0.0015
Constant (Comparisons): 0.6667
Constant (Time): 0.0001
RANDOMISED QUICK SORT: UNIFORM DATASET
Average Number of comparisons: 19
Average Time Taken: 0.0017
Constant (Comparisons): 0.7917
Constant (Time): 0.0001
RANDOMISED QUICK SORT: NORMAL DATASET
Average Number of comparisons: 18
Average Time Taken: 0.0015
Constant (Comparisons): 0.7500
Constant (Time): 0.0001
QUICK SORT: UNIFORM DATASET
Average Number of comparisons: 17
Average Time Taken: 0.0017
Constant (Comparisons): 0.7083
Constant (Time): 0.0001
QUICK SORT: NORMAL DATASET
Average Number of comparisons: 18
Average Time Taken: 0.0012
Constant (Comparisons): 0.7500
Constant (Time): 0.0001
BUCKET SORT: UNIFORM DATASET
Average Number of comparisons: 0
Average Time Taken: 0.0223
Constant (Comparisons): 0.0000
Constant (Time): 0.0028
BUCKET SORT: UNIFORM DATASET
Average Number of comparisons: 0
Average Time Taken: 0.0195
Constant (Comparisons): 0.0000
Constant (Time): 0.0024
```

```
RESULTS FOR DATASET SIZE 16:

MERGE SORT: UNIFORM DATASET
Average Number of comparisons: 43
Average Time Taken: 0.0020
Constant (Comparisons): 0.6719
Constant (Time): 0.0000
MERGE SORT: NORMAL DATASET
Average Number of comparisons: 47
Average Time Taken: 0.0013
Constant (Comparisons): 0.7344
Constant (Time): 0.0000
RANDOMISED QUICK SORT: UNIFORM DATASET
Average Number of comparisons: 55
Average Time Taken: 0.0018
Constant (Comparisons): 0.8594
Constant (Time): 0.0000
RANDOMISED QUICK SORT: NORMAL DATASET
Average Number of comparisons: 50
Average Time Taken: 0.0011
Constant (Comparisons): 0.7812
Constant (Time): 0.0000
QUICK SORT: UNIFORM DATASET
Average Number of comparisons: 61
Average Time Taken: 0.0017
Constant (Comparisons): 0.9531
Constant (Time): 0.0000
QUICK SORT: NORMAL DATASET
Average Number of comparisons: 62
Average Time Taken: 0.0017
Constant (Comparisons): 0.9688
Constant (Time): 0.0000
BUCKET SORT: UNIFORM DATASET
Average Number of comparisons: 1
Average Time Taken: 0.0158
Constant (Comparisons): 0.0625
Constant (Time): 0.0010
BUCKET SORT: UNIFORM DATASET
Average Number of comparisons: 0
Average Time Taken: 0.0156
Constant (Comparisons): 0.0000
Constant (Time): 0.0010
```

Master File results

Time in mlliseconds

```
RESULTS FOR DATASET SIZE 32:

MERGE SORT: UNIFORM DATASET
Average Number of comparisons: 121
Average Time Taken: 0.0034
Constant (Comparisons): 0.7562
Constant (Time): 0.0000
MERGE SORT: NORMAL DATASET
Average Number of comparisons: 121
Average Time Taken: 0.0026
Constant (Comparisons): 0.7562
Constant (Time): 0.0000
RANDOMISED QUICK SORT: UNIFORM DATASET
Average Number of comparisons: 142
Average Time Taken: 0.0028
Constant (Comparisons): 0.8875
Constant (Time): 0.0000
RANDOMISED QUICK SORT: NORMAL DATASET
Average Number of comparisons: 159
Average Time Taken: 0.0026
Constant (Comparisons): 0.9938
Constant (Time): 0.0000
QUICK SORT: UNIFORM DATASET
Average Number of comparisons: 156
Average Time Taken: 0.0029
Constant (Comparisons): 0.9750
Constant (Time): 0.0000
QUICK SORT: NORMAL DATASET
Average Number of comparisons: 164
Average Time Taken: 0.0027
Constant (Comparisons): 1.0250
Constant (Time): 0.0000
BUCKET SORT: UNIFORM DATASET
Average Number of comparisons: 2
Average Time Taken: 0.0152
Constant (Comparisons): 0.0625
Constant (Time): 0.0005
BUCKET SORT: UNIFORM DATASET
Average Number of comparisons: 0
Average Time Taken: 0.0139
Constant (Comparisons): 0.0000
Constant (Time): 0.0004
```

```
RESULTS FOR DATASET SIZE 64:

MERGE SORT: UNIFORM DATASET
Average Number of comparisons: 306
Average Time Taken: 0.0045
Constant (Comparisons): 0.7969
Constant (Time): 0.0000
MERGE SORT: NORMAL DATASET
Average Number of comparisons: 304
Average Time Taken: 0.0037
Constant (Comparisons): 0.7917
Constant (Time): 0.0000
RANDOMISED QUICK SORT: UNIFORM DATASET
Average Number of comparisons: 367
Average Time Taken: 0.0034
Constant (Comparisons): 0.9557
Constant (Time): 0.0000
RANDOMISED QUICK SORT: NORMAL DATASET
Average Number of comparisons: 340
Average Time Taken: 0.0030
Constant (Comparisons): 0.8854
Constant (Time): 0.0000
QUICK SORT: UNIFORM DATASET
Average Number of comparisons: 397
Average Time Taken: 0.0039
Constant (Comparisons): 1.0339
Constant (Time): 0.0000
QUICK SORT: NORMAL DATASET
Average Number of comparisons: 447
Average Time Taken: 0.0035
Constant (Comparisons): 1.1641
Constant (Time): 0.0000
BUCKET SORT: UNIFORM DATASET
Average Number of comparisons: 7
Average Time Taken: 0.0100
Constant (Comparisons): 0.1094
Constant (Time): 0.0002
BUCKET SORT: UNIFORM DATASET
Average Number of comparisons: 9
Average Time Taken: 0.0094
Constant (Comparisons): 0.1406
Constant (Time): 0.0001
```

```
RESULTS FOR DATASET SIZE 128:

MERGE SORT: UNIFORM DATASET
Average Number of comparisons: 727
Average Time Taken: 0.0132
Constant (Comparisons): 0.8114
Constant (Time): 0.0000
MERGE SORT: NORMAL DATASET
Average Number of comparisons: 742
Average Time Taken: 0.0107
Constant (Comparisons): 0.8281
Constant (Time): 0.0000
RANDOMISED QUICK SORT: UNIFORM DATASET
Average Number of comparisons: 830
Average Time Taken: 0.0113
Constant (Comparisons): 0.9263
Constant (Time): 0.0000
RANDOMISED QUICK SORT: NORMAL DATASET
Average Number of comparisons: 916
Average Time Taken: 0.0113
Constant (Comparisons): 1.0223
Constant (Time): 0.0000
QUICK SORT: UNIFORM DATASET
Average Number of comparisons: 985
Average Time Taken: 0.0126
Constant (Comparisons): 1.0993
Constant (Time): 0.0000
QUICK SORT: NORMAL DATASET
Average Number of comparisons: 975
Average Time Taken: 0.0108
Constant (Comparisons): 1.0882
Constant (Time): 0.0000
BUCKET SORT: UNIFORM DATASET
Average Number of comparisons: 33
Average Time Taken: 0.0153
Constant (Comparisons): 0.2578
Constant (Time): 0.0001
BUCKET SORT: UNIFORM DATASET
Average Number of comparisons: 36
Average Time Taken: 0.0129
Constant (Comparisons): 0.2812
Constant (Time): 0.0001
```

```
RESULTS FOR DATASET SIZE 256:

MERGE SORT: UNIFORM DATASET
Average Number of comparisons: 1730
Average Time Taken: 0.0262
Constant (Comparisons): 0.8447
Constant (Time): 0.0000
MERGE SORT: NORMAL DATASET
Average Number of comparisons: 1726
Average Time Taken: 0.0204
Constant (Comparisons): 0.8428
Constant (Time): 0.0000
RANDOMISED QUICK SORT: UNIFORM DATASET
Average Number of comparisons: 2043
Average Time Taken: 0.0230
Constant (Comparisons): 0.9976
Constant (Time): 0.0000
RANDOMISED QUICK SORT: NORMAL DATASET
Average Number of comparisons: 2087
Average Time Taken: 0.0228
Constant (Comparisons): 1.0190
Constant (Time): 0.0000
QUICK SORT: UNIFORM DATASET
Average Number of comparisons: 2461
Average Time Taken: 0.0255
Constant (Comparisons): 1.2017
Constant (Time): 0.0000
QUICK SORT: NORMAL DATASET
Average Number of comparisons: 2261
Average Time Taken: 0.0201
Constant (Comparisons): 1.1040
Constant (Time): 0.0000
BUCKET SORT: UNIFORM DATASET
Average Number of comparisons: 106
Average Time Taken: 0.0167
Constant (Comparisons): 0.4141
Constant (Time): 0.0001
BUCKET SORT: UNIFORM DATASET
Average Number of comparisons: 132
Average Time Taken: 0.0150
Constant (Comparisons): 0.5156
Constant (Time): 0.0001
```

Master File results

Time in mlliseconds

```
RESULTS FOR DATASET SIZE 512:

MERGE SORT: UNIFORM DATASET
Average Number of comparisons: 3964
Average Time Taken: 0.0497
Constant (Comparisons): 0.8602
Constant (Time): 0.0000
MERGE SORT: NORMAL DATASET
Average Number of comparisons: 3962
Average Time Taken: 0.0437
Constant (Comparisons): 0.8598
Constant (Time): 0.0000
RANDOMISED QUICK SORT: UNIFORM DATASET
Average Number of comparisons: 4851
Average Time Taken: 0.0425
Constant (Comparisons): 1.0527
Constant (Time): 0.0000
RANDOMISED QUICK SORT: NORMAL DATASET
Average Number of comparisons: 4939
Average Time Taken: 0.0419
Constant (Comparisons): 1.0718
Constant (Time): 0.0000
QUICK SORT: UNIFORM DATASET
Average Number of comparisons: 6231
Average Time Taken: 0.0468
Constant (Comparisons): 1.3522
Constant (Time): 0.0000
QUICK SORT: NORMAL DATASET
Average Number of comparisons: 5884
Average Time Taken: 0.0444
Constant (Comparisons): 1.2769
Constant (Time): 0.0000
BUCKET SORT: UNIFORM DATASET
Average Number of comparisons: 284
Average Time Taken: 0.0223
Constant (Comparisons): 0.5547
Constant (Time): 0.0000
BUCKET SORT: UNIFORM DATASET
Average Number of comparisons: 322
Average Time Taken: 0.0206
Constant (Comparisons): 0.6289
Constant (Time): 0.0000
```

```
RESULTS FOR DATASET SIZE 1024:

MERGE SORT: UNIFORM DATASET
Average Number of comparisons: 8963
Average Time Taken: 0.0894
Constant (Comparisons): 0.8753
Constant (Time): 0.0000
MERGE SORT: NORMAL DATASET
Average Number of comparisons: 8963
Average Time Taken: 0.0867
Constant (Comparisons): 0.8753
Constant (Time): 0.0000
RANDOMISED QUICK SORT: UNIFORM DATASET
Average Number of comparisons: 11292
Average Time Taken: 0.0749
Constant (Comparisons): 1.1027
Constant (Time): 0.0000
RANDOMISED QUICK SORT: NORMAL DATASET
Average Number of comparisons: 11684
Average Time Taken: 0.0732
Constant (Comparisons): 1.1410
Constant (Time): 0.0000
QUICK SORT: UNIFORM DATASET
Average Number of comparisons: 13057
Average Time Taken: 0.0814
Constant (Comparisons): 1.2751
Constant (Time): 0.0000
QUICK SORT: NORMAL DATASET
Average Number of comparisons: 14160
Average Time Taken: 0.0816
Constant (Comparisons): 1.3828
Constant (Time): 0.0000
BUCKET SORT: UNIFORM DATASET
Average Number of comparisons: 729
Average Time Taken: 0.0319
Constant (Comparisons): 0.7119
Constant (Time): 0.0000
BUCKET SORT: UNIFORM DATASET
Average Number of comparisons: 766
Average Time Taken: 0.0310
Constant (Comparisons): 0.7480
Constant (Time): 0.0000
```

```
RESULTS FOR DATASET SIZE 2048:

MERGE SORT: UNIFORM DATASET
Average Number of comparisons: 19901
Average Time Taken: 0.1850
Constant (Comparisons): 0.8834
Constant (Time): 0.0000
MERGE SORT: NORMAL DATASET
Average Number of comparisons: 19950
Average Time Taken: 0.1819
Constant (Comparisons): 0.8856
Constant (Time): 0.0000
RANDOMISED QUICK SORT: UNIFORM DATASET
Average Number of comparisons: 26246
Average Time Taken: 0.1558
Constant (Comparisons): 1.1650
Constant (Time): 0.0000
RANDOMISED QUICK SORT: NORMAL DATASET
Average Number of comparisons: 25458
Average Time Taken: 0.1538
Constant (Comparisons): 1.1301
Constant (Time): 0.0000
QUICK SORT: UNIFORM DATASET
Average Number of comparisons: 30363
Average Time Taken: 0.1715
Constant (Comparisons): 1.3478
Constant (Time): 0.0000
QUICK SORT: NORMAL DATASET
Average Number of comparisons: 30992
Average Time Taken: 0.1643
Constant (Comparisons): 1.3757
Constant (Time): 0.0000
BUCKET SORT: UNIFORM DATASET
Average Number of comparisons: 1678
Average Time Taken: 0.0592
Constant (Comparisons): 0.8193
Constant (Time): 0.0000
BUCKET SORT: UNIFORM DATASET
Average Number of comparisons: 1773
Average Time Taken: 0.0703
Constant (Comparisons): 0.8657
Constant (Time): 0.0000
```

```
RESULTS FOR DATASET SIZE 4096:

MERGE SORT: UNIFORM DATASET
Average Number of comparisons: 43988
Average Time Taken: 0.3976
Constant (Comparisons): 0.8949
Constant (Time): 0.0000
MERGE SORT: NORMAL DATASET
Average Number of comparisons: 43992
Average Time Taken: 0.3960
Constant (Comparisons): 0.8950
Constant (Time): 0.0000
RANDOMISED QUICK SORT: UNIFORM DATASET
Average Number of comparisons: 56478
Average Time Taken: 0.3376
Constant (Comparisons): 1.1490
Constant (Time): 0.0000
RANDOMISED QUICK SORT: NORMAL DATASET
Average Number of comparisons: 55304
Average Time Taken: 0.3445
Constant (Comparisons): 1.1252
Constant (Time): 0.0000
QUICK SORT: UNIFORM DATASET
Average Number of comparisons: 71764
Average Time Taken: 0.3714
Constant (Comparisons): 1.4600
Constant (Time): 0.0000
QUICK SORT: NORMAL DATASET
Average Number of comparisons: 68713
Average Time Taken: 0.3660
Constant (Comparisons): 1.3980
Constant (Time): 0.0000
BUCKET SORT: UNIFORM DATASET
Average Number of comparisons: 3654
Average Time Taken: 0.1511
Constant (Comparisons): 0.8921
Constant (Time): 0.0000
BUCKET SORT: UNIFORM DATASET
Average Number of comparisons: 3779
Average Time Taken: 0.2122
Constant (Comparisons): 0.9226
Constant (Time): 0.0001
```

Master File results

Time in mlliseconds

```
RESULTS FOR DATASET SIZE 8192:

MERGE SORT: UNIFORM DATASET
Average Number of comparisons: 96114
Average Time Taken: 0.8376
Constant (Comparisons): 0.9025
Constant (Time): 0.0000
MERGE SORT: NORMAL DATASET
Average Number of comparisons: 96201
Average Time Taken: 0.8415
Constant (Comparisons): 0.9033
Constant (Time): 0.0000
RANDOMISED QUICK SORT: UNIFORM DATASET
Average Number of comparisons: 123699
Average Time Taken: 0.7045
Constant (Comparisons): 1.1615
Constant (Time): 0.0000
RANDOMISED QUICK SORT: NORMAL DATASET
Average Number of comparisons: 128143
Average Time Taken: 0.7045
Constant (Comparisons): 1.2033
Constant (Time): 0.0000
QUICK SORT: UNIFORM DATASET
Average Number of comparisons: 150150
Average Time Taken: 0.7671
Constant (Comparisons): 1.4099
Constant (Time): 0.0000
QUICK SORT: NORMAL DATASET
Average Number of comparisons: 145839
Average Time Taken: 0.7556
Constant (Comparisons): 1.3694
Constant (Time): 0.0000
BUCKET SORT: UNIFORM DATASET
Average Number of comparisons: 7717
Average Time Taken: 0.4137
Constant (Comparisons): 0.9420
Constant (Time): 0.0001
BUCKET SORT: UNIFORM DATASET
Average Number of comparisons: 7820
Average Time Taken: 0.6708
Constant (Comparisons): 0.9546
Constant (Time): 0.0001
```

```
RESULTS FOR DATASET SIZE 16384:

MERGE SORT: UNIFORM DATASET
Average Number of comparisons: 208668
Average Time Taken: 1.7922
Constant (Comparisons): 0.9097
Constant (Time): 0.0000
MERGE SORT: NORMAL DATASET
Average Number of comparisons: 208679
Average Time Taken: 1.7864
Constant (Comparisons): 0.9098
Constant (Time): 0.0000
RANDOMISED QUICK SORT: UNIFORM DATASET
Average Number of comparisons: 272737
Average Time Taken: 1.4945
Constant (Comparisons): 1.1890
Constant (Time): 0.0000
RANDOMISED QUICK SORT: NORMAL DATASET
Average Number of comparisons: 274305
Average Time Taken: 1.4900
Constant (Comparisons): 1.1959
Constant (Time): 0.0000
QUICK SORT: UNIFORM DATASET
Average Number of comparisons: 321507
Average Time Taken: 1.6181
Constant (Comparisons): 1.4017
Constant (Time): 0.0000
QUICK SORT: NORMAL DATASET
Average Number of comparisons: 310596
Average Time Taken: 1.6038
Constant (Comparisons): 1.3541
Constant (Time): 0.0000
BUCKET SORT: UNIFORM DATASET
Average Number of comparisons: 15809
Average Time Taken: 1.3920
Constant (Comparisons): 0.9649
Constant (Time): 0.0001
BUCKET SORT: UNIFORM DATASET
Average Number of comparisons: 15935
Average Time Taken: 2.4906
Constant (Comparisons): 0.9726
Constant (Time): 0.0002
```

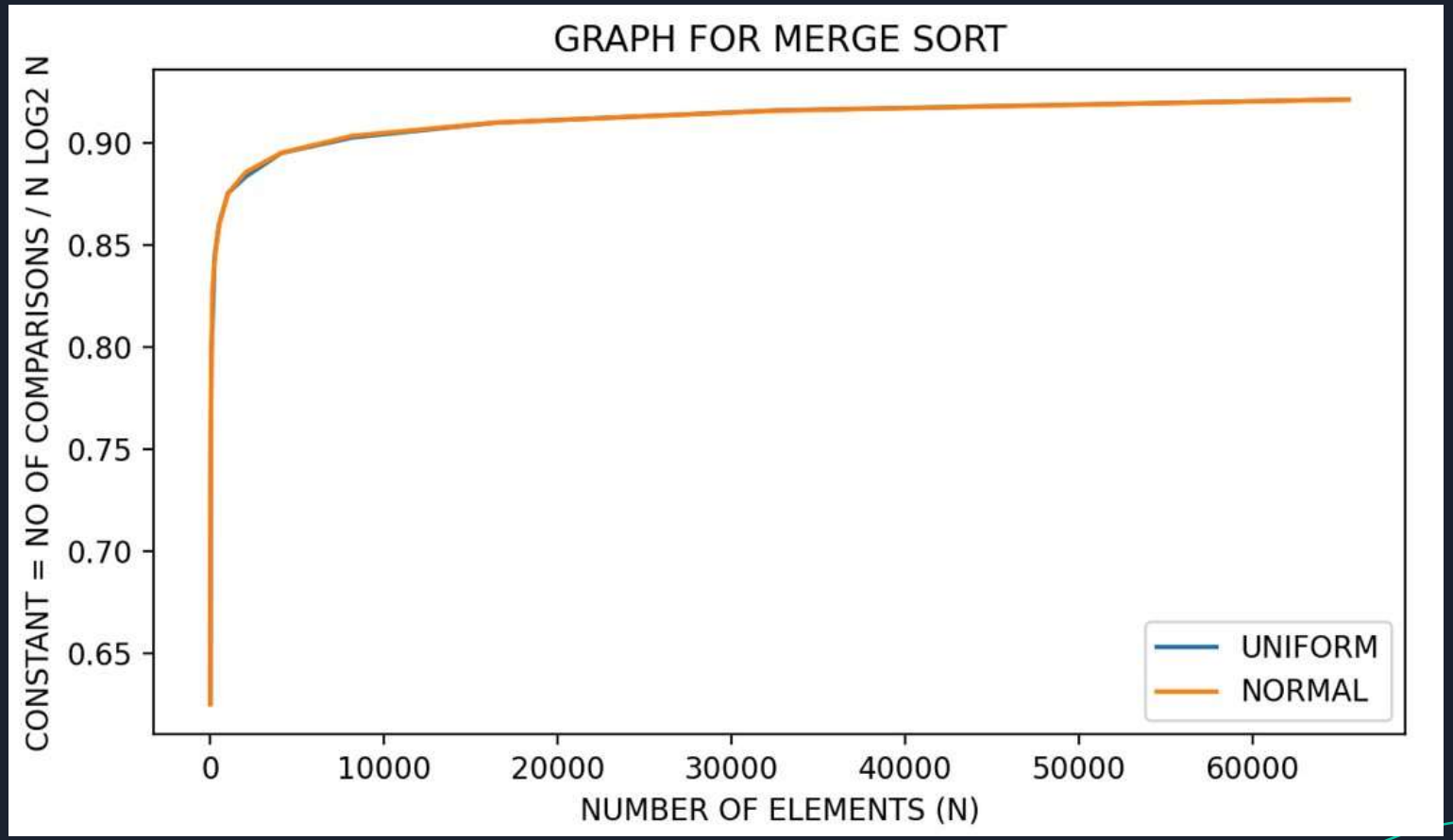
```
RESULTS FOR DATASET SIZE 32768:

MERGE SORT: UNIFORM DATASET
Average Number of comparisons: 450143
Average Time Taken: 3.8106
Constant (Comparisons): 0.9158
Constant (Time): 0.0000
MERGE SORT: NORMAL DATASET
Average Number of comparisons: 450064
Average Time Taken: 3.7856
Constant (Comparisons): 0.9157
Constant (Time): 0.0000
RANDOMISED QUICK SORT: UNIFORM DATASET
Average Number of comparisons: 592251
Average Time Taken: 3.1881
Constant (Comparisons): 1.2049
Constant (Time): 0.0000
RANDOMISED QUICK SORT: NORMAL DATASET
Average Number of comparisons: 594451
Average Time Taken: 3.1633
Constant (Comparisons): 1.2094
Constant (Time): 0.0000
QUICK SORT: UNIFORM DATASET
Average Number of comparisons: 728957
Average Time Taken: 3.4903
Constant (Comparisons): 1.4831
Constant (Time): 0.0000
QUICK SORT: NORMAL DATASET
Average Number of comparisons: 716355
Average Time Taken: 3.4340
Constant (Comparisons): 1.4574
Constant (Time): 0.0000
BUCKET SORT: UNIFORM DATASET
Average Number of comparisons: 32109
Average Time Taken: 4.8141
Constant (Comparisons): 0.9799
Constant (Time): 0.0001
BUCKET SORT: UNIFORM DATASET
Average Number of comparisons: 32279
Average Time Taken: 9.8628
Constant (Comparisons): 0.9851
Constant (Time): 0.0003
```

```
RESULTS FOR DATASET SIZE 65536:

MERGE SORT: UNIFORM DATASET
Average Number of comparisons: 965866
Average Time Taken: 8.0966
Constant (Comparisons): 0.9211
Constant (Time): 0.0000
MERGE SORT: NORMAL DATASET
Average Number of comparisons: 965833
Average Time Taken: 8.1034
Constant (Comparisons): 0.9211
Constant (Time): 0.0000
RANDOMISED QUICK SORT: UNIFORM DATASET
Average Number of comparisons: 1284711
Average Time Taken: 6.7230
Constant (Comparisons): 1.2252
Constant (Time): 0.0000
RANDOMISED QUICK SORT: NORMAL DATASET
Average Number of comparisons: 1257604
Average Time Taken: 6.7211
Constant (Comparisons): 1.1993
Constant (Time): 0.0000
QUICK SORT: UNIFORM DATASET
Average Number of comparisons: 1467839
Average Time Taken: 7.2208
Constant (Comparisons): 1.3998
Constant (Time): 0.0000
QUICK SORT: NORMAL DATASET
Average Number of comparisons: 1492357
Average Time Taken: 7.2092
Constant (Comparisons): 1.4232
Constant (Time): 0.0000
BUCKET SORT: UNIFORM DATASET
Average Number of comparisons: 64837
Average Time Taken: 17.7079
Constant (Comparisons): 0.9893
Constant (Time): 0.0003
BUCKET SORT: UNIFORM DATASET
Average Number of comparisons: 64982
Average Time Taken: 39.9228
Constant (Comparisons): 0.9915
Constant (Time): 0.0006
```

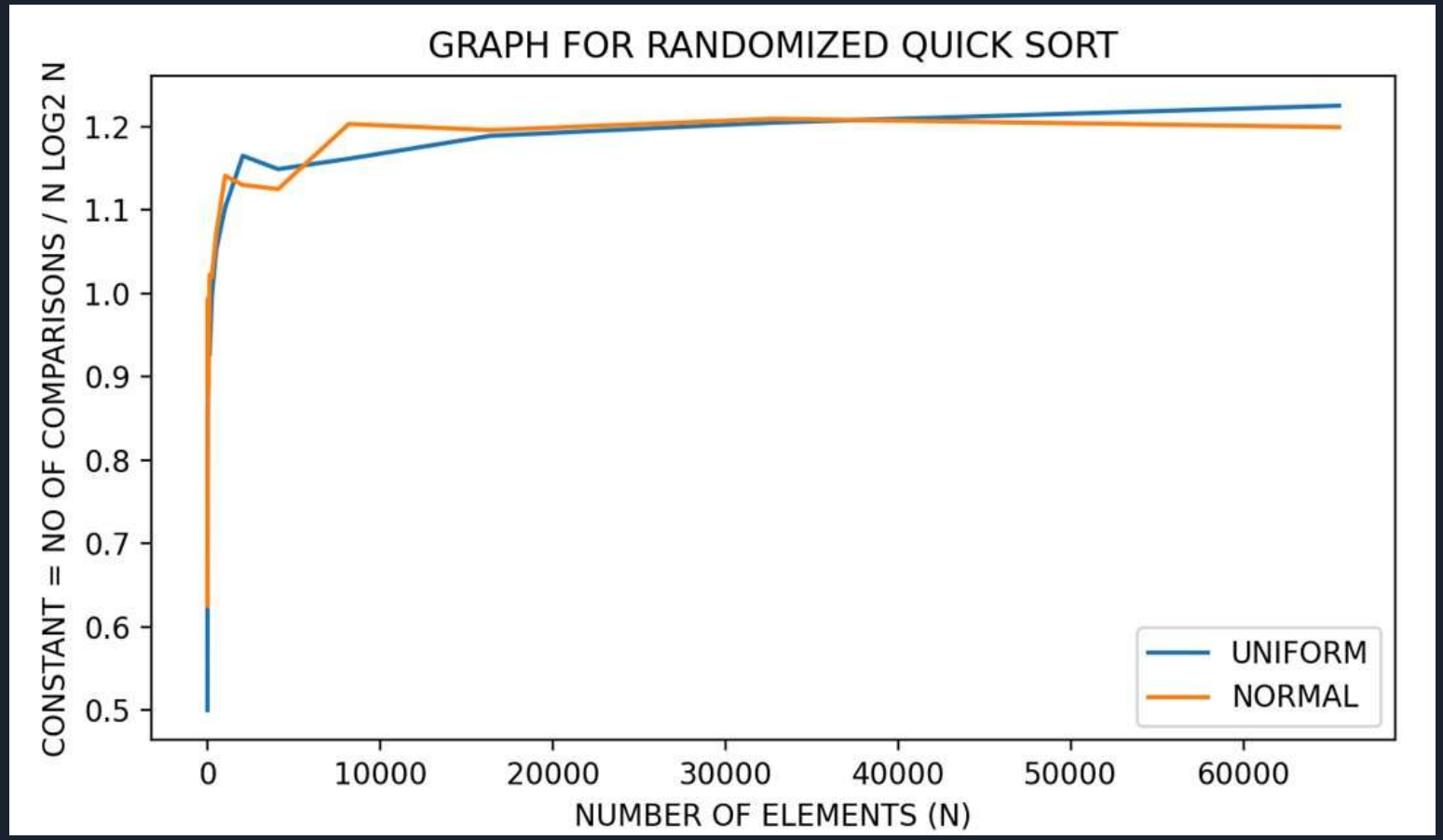
Summarised Results (Constant Ratio)

DATASET SIZE	MS_UD	MS_ND	RQS_UD	RQS_ND	QS_UD	QS_ND	BS_UD	BS_ND
2	0.500000	0.500000	0.500000	0.500000	0.500000	0.500000	0.000000	0.000000
4	0.625000	0.625000	0.500000	0.625000	0.625000	0.750000	0.000000	0.000000
8	0.625000	0.666667	0.791667	0.750000	0.708333	0.750000	0.000000	0.000000
16	0.671875	0.734375	0.859375	0.781250	0.953125	0.968750	0.062500	0.000000
32	0.756250	0.756250	0.887500	0.993750	0.975000	1.025000	0.062500	0.000000
64	0.796875	0.791667	0.955729	0.885417	1.033854	1.164062	0.109375	0.140625
128	0.811384	0.828125	1.052734	1.022321	1.099330	1.088170	0.257812	0.281250
256	0.844727	0.842773	1.102734	1.019043	1.201660	1.04004	0.414062	0.515625
512	0.860243	0.859809	1.165039	1.071832	1.352214	1.276910	0.554688	0.628906
1024	0.875293	0.875293	1.149048	1.141016	1.275098	1.382812	0.711914	0.748047
2048	0.883390	0.885565	1.165039	1.130060	1.347789	1.375710	0.819336	0.865723
4096	0.894938	0.895020	1.149048	1.125163	1.460042	1.397970	0.892090	0.922607
8192	0.902513	0.903330	1.161537	1.203266	1.409912	1.369432	0.942017	0.954590
16384	0.909720	0.909768	1.189039	1.195875	1.401659	1.354091	0.964905	0.972595
32768	0.915818	0.915658	1.204938	1.209414	1.483067	1.457428	0.979889	0.985077
65536	0.921122	0.921090	1.225196	1.199345	1.399840	1.423223	0.989334	0.991547

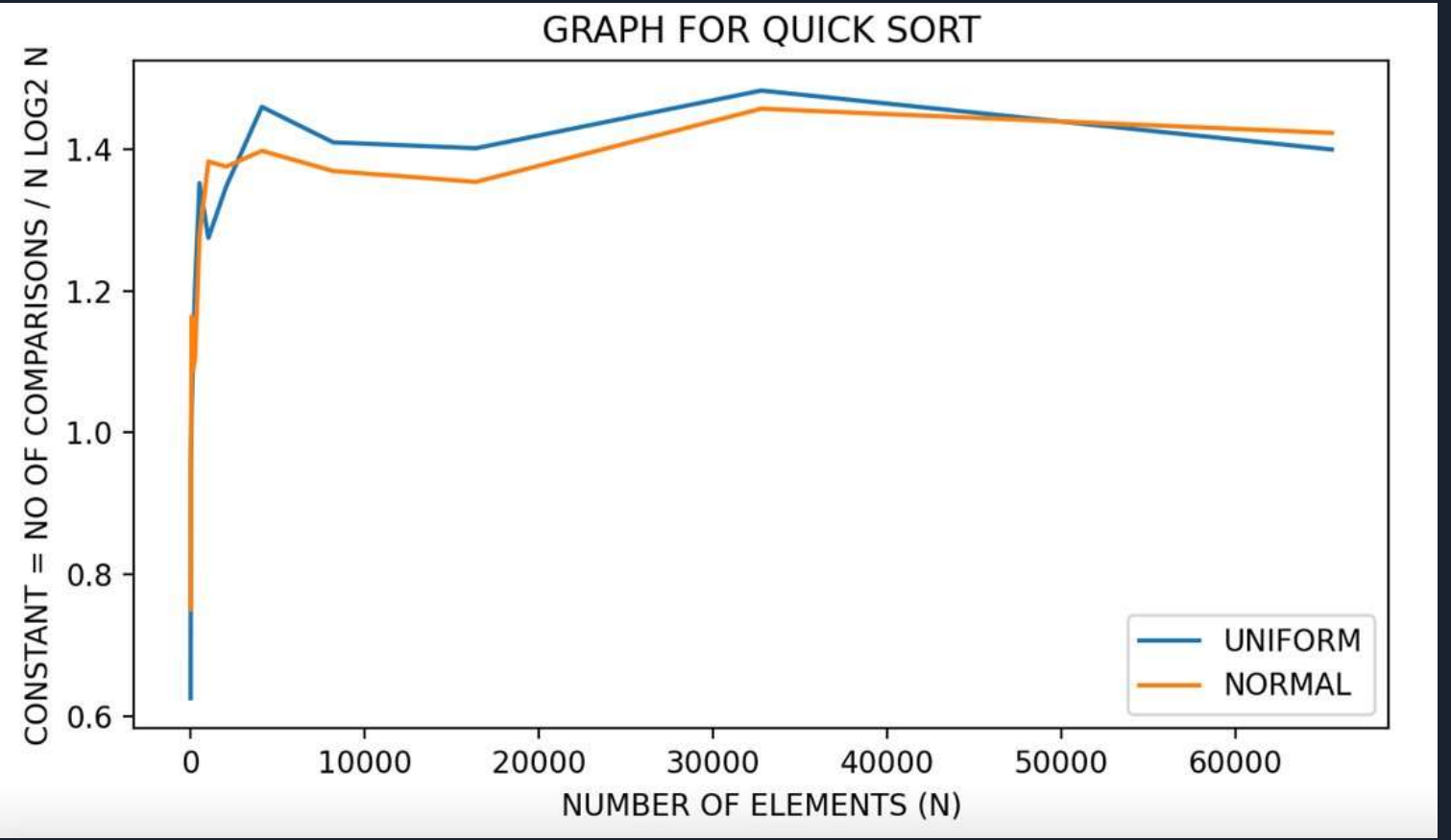
Comparative Graphs (Average number of Comparisons)



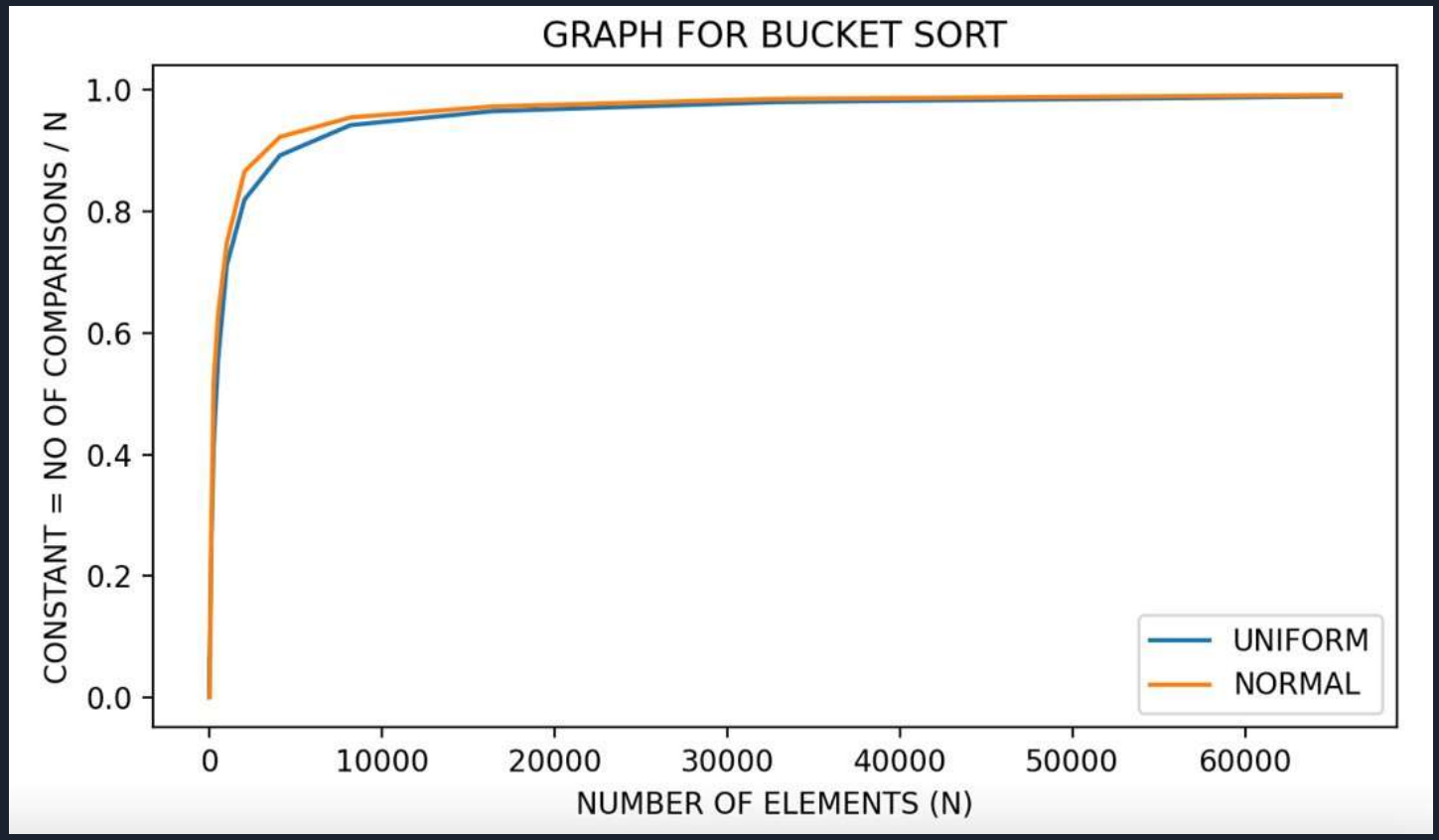
Comparative Graphs (Average number of Comparisons)



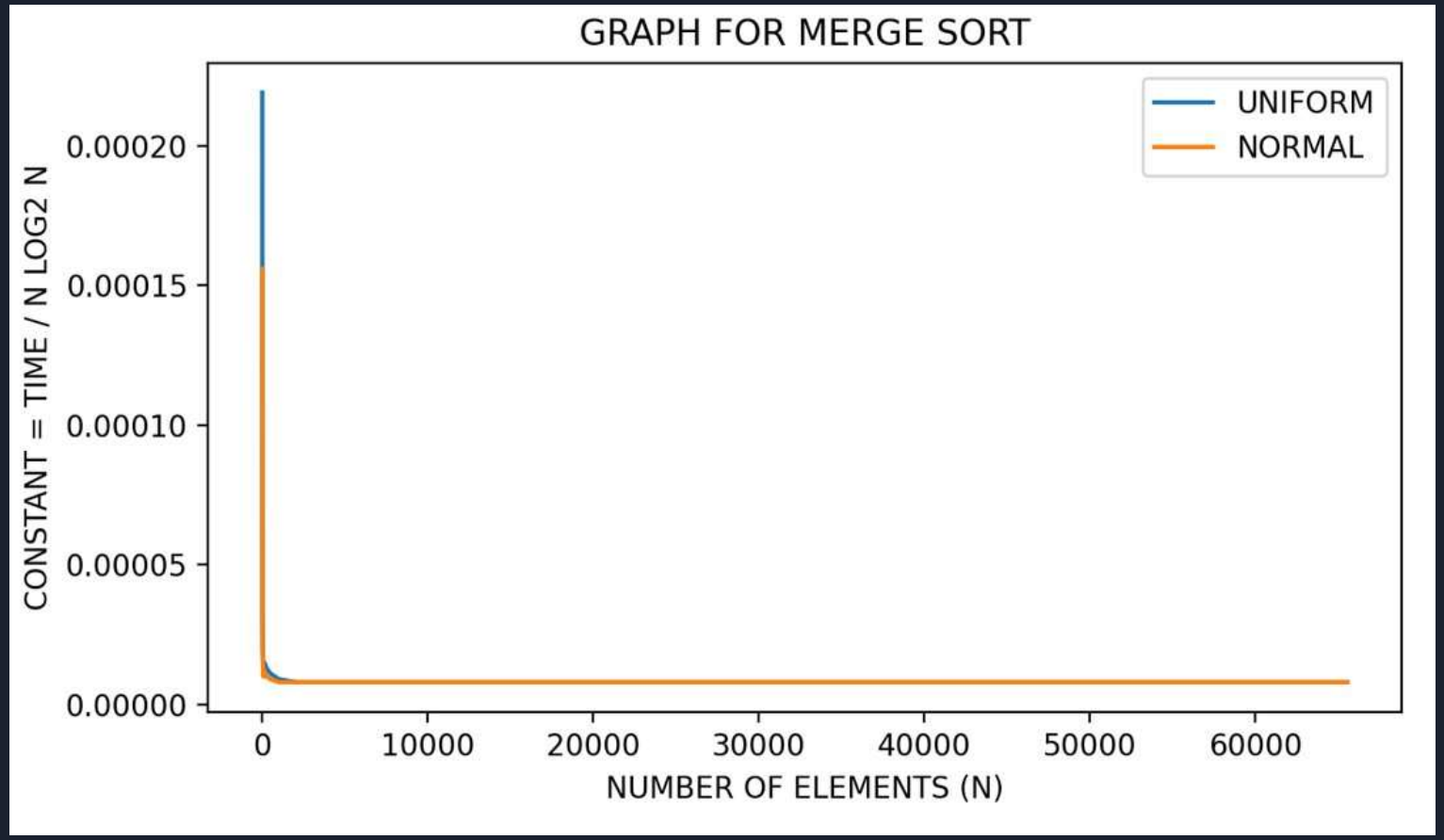
Comparative Graphs (Average number of Comparisons)



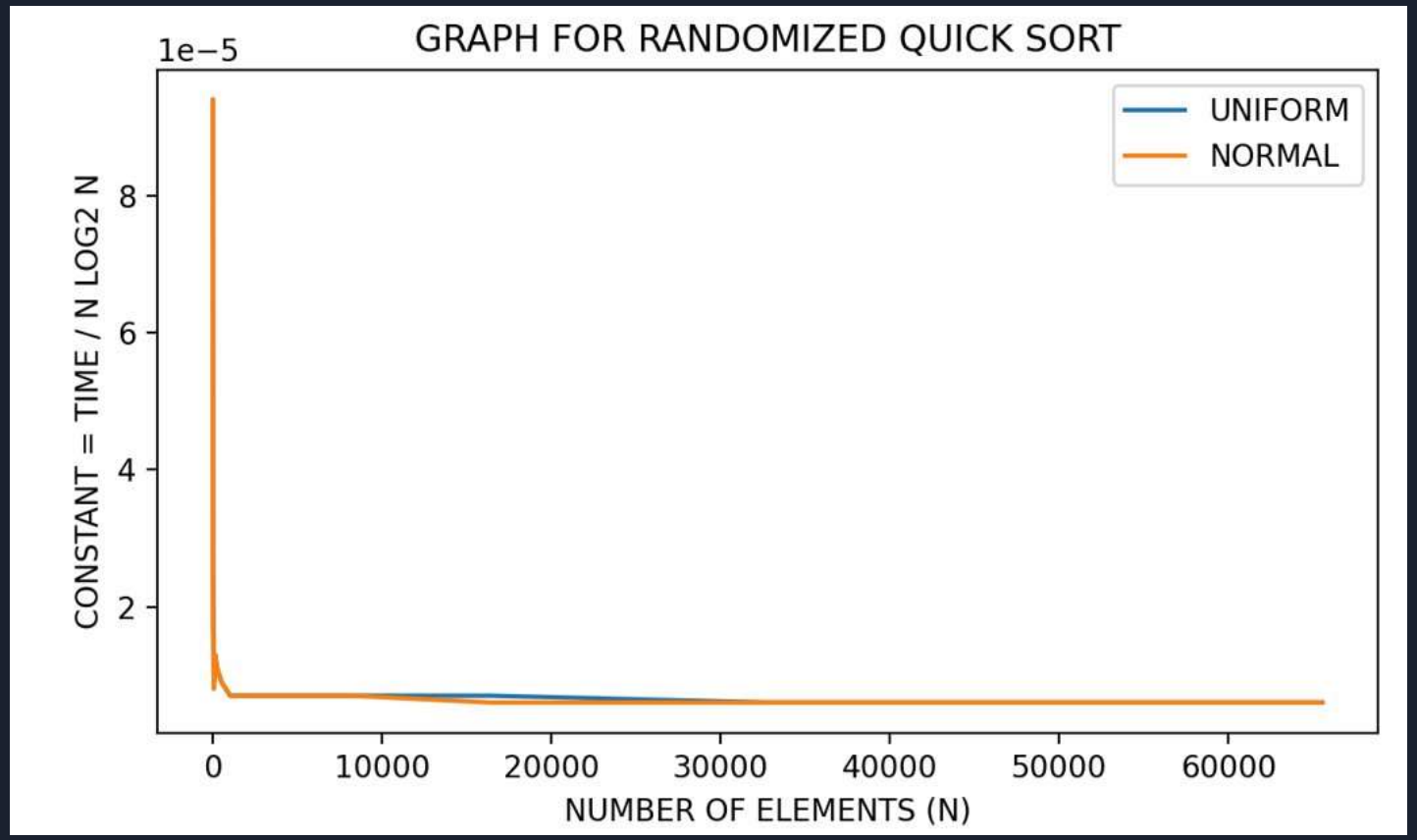
Comparative Graphs (Average number of Comparisons)



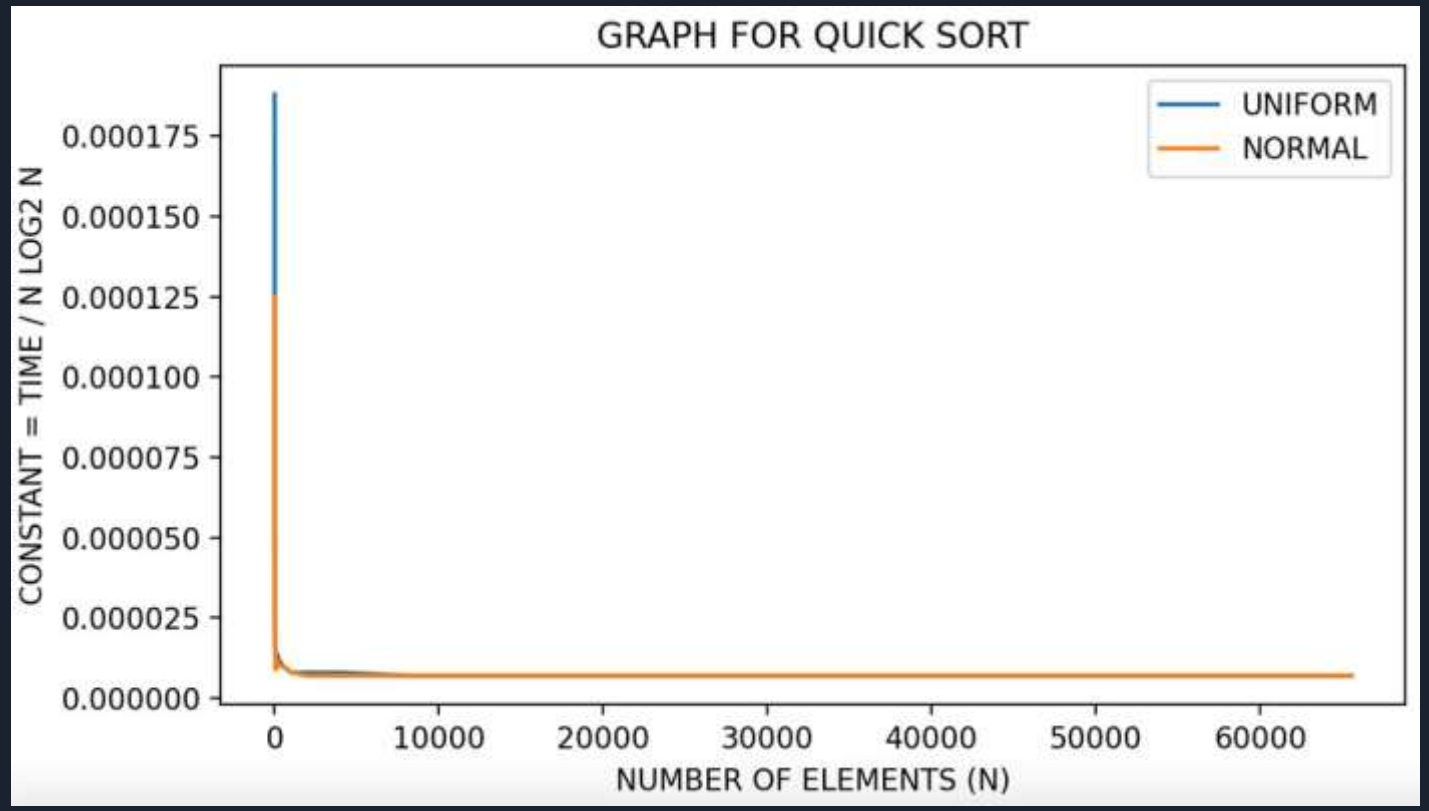
Comparative Graphs (Average Time Taken)



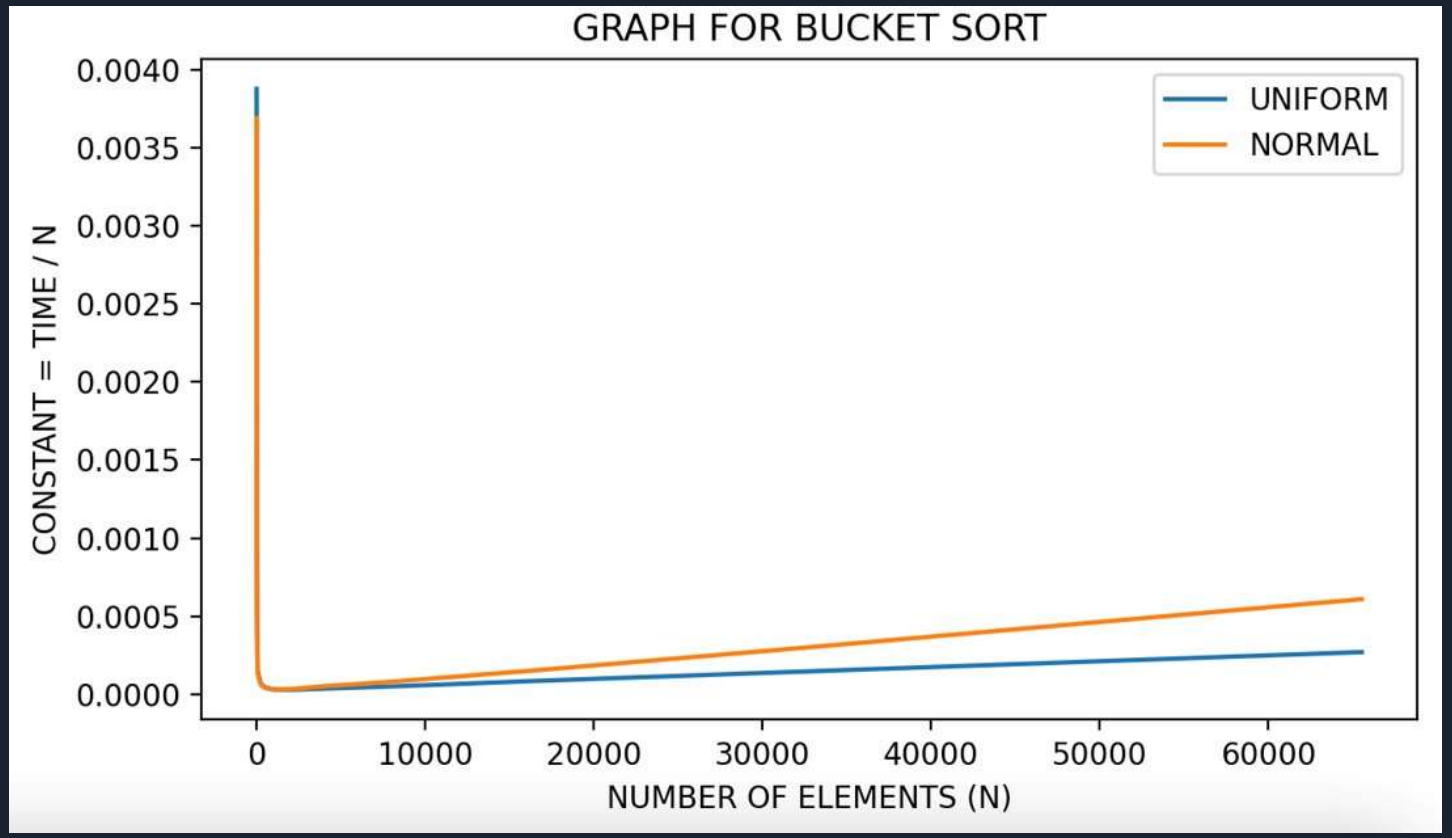
Comparative Graphs (Average Time Taken)



Comparative Graphs (Average Time Taken)



Comparative Graphs (Average Time Taken)



Inference

- **The graphs certainly give us an insight into the complexities of the various sorting algorithms. The results obtain indeed go hand-in-hand with our expected theoretical complexities.**

SORTING ALGORITHM	AVERAGE COMPLEXITY
MERGE	n log2 n
QUICK	n log2 n
RANDOMISED QUICK	n log2 n
BUCKET	n

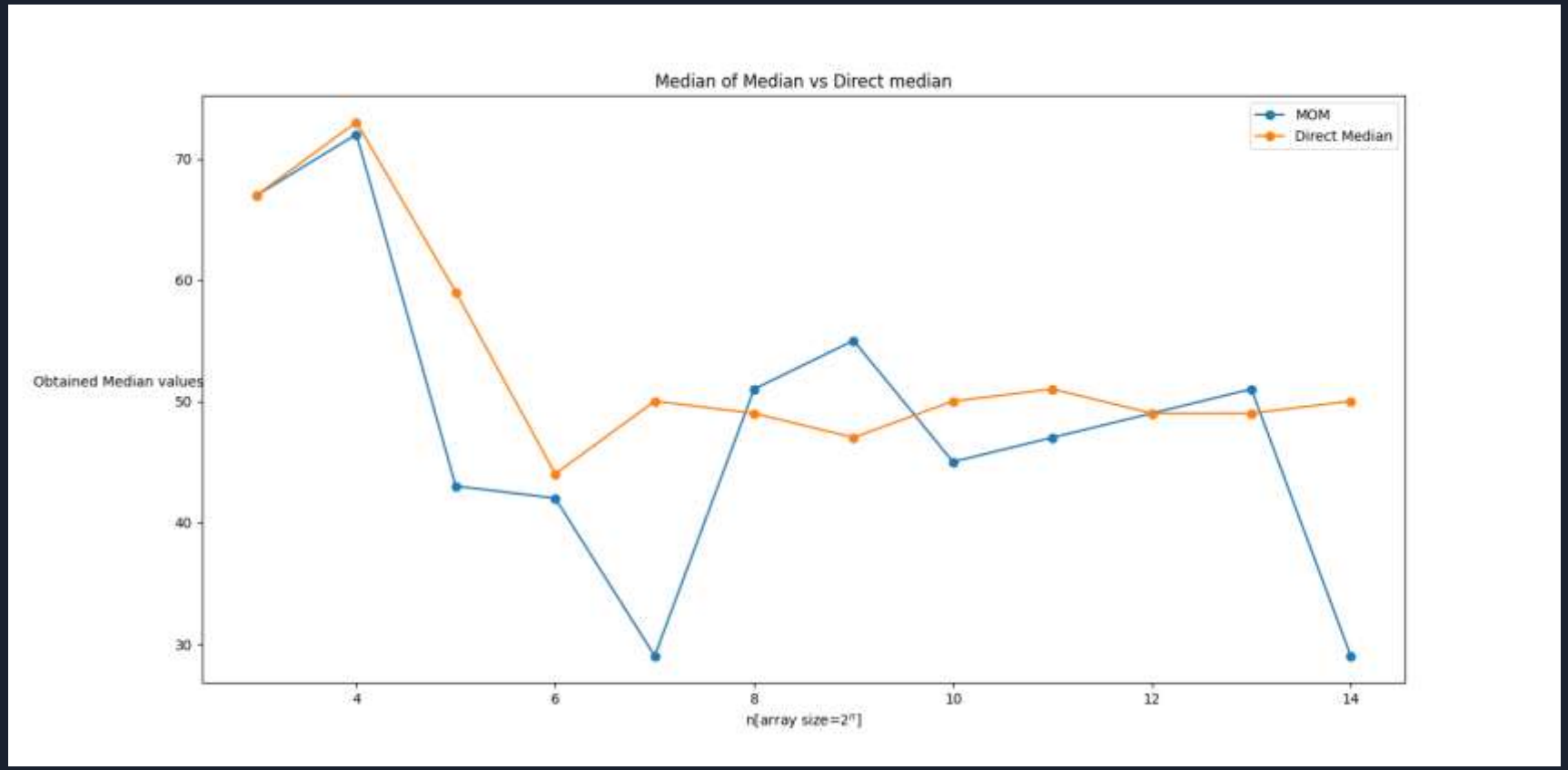
8. Implement the worst case linear median selection algorithm by taking the median of medians (MoM) as the pivotal element and check for correctness.

- In computer science, the **median of medians** is an approximate median selection algorithm, frequently used to supply a good pivot for an exact selection algorithm, most commonly quickselect, that selects the *k*th smallest element of an initially unsorted array. Median of medians finds an approximate median in linear time. Using this approximate median as an improved pivot, the worst-case complexity of quick-select reduces from quadratic to *linear*, which is also the asymptotically optimal worst-case complexity of any selection algorithm. In other words, the median of medians is an approximate median-selection algorithm that helps building an asymptotically optimal, exact general selection algorithm (especially in the sense of worst-case complexity), by producing good pivot elements.
- Median of medians can also be used as a pivot strategy in quicksort, yielding an optimal algorithm, with worst-case complexity O(NlogN). Although this approach optimizes the asymptotic worst-case complexity quite well, it is typically outperformed in practice by instead choosing random pivots for its average O(N) complexity for selection and average O(NlogN) complexity for sorting, without any overhead of computing the pivot.

Procedure

1. We n elements into n/divide_size groups of size same as divide_size and n%divide_size elements in the last group
2. We then find the direct median of each group and group them into an array of size n/divide_size(+1 if extra group exists)
3. We then repeat steps 1 and 2 till a single value is reached.

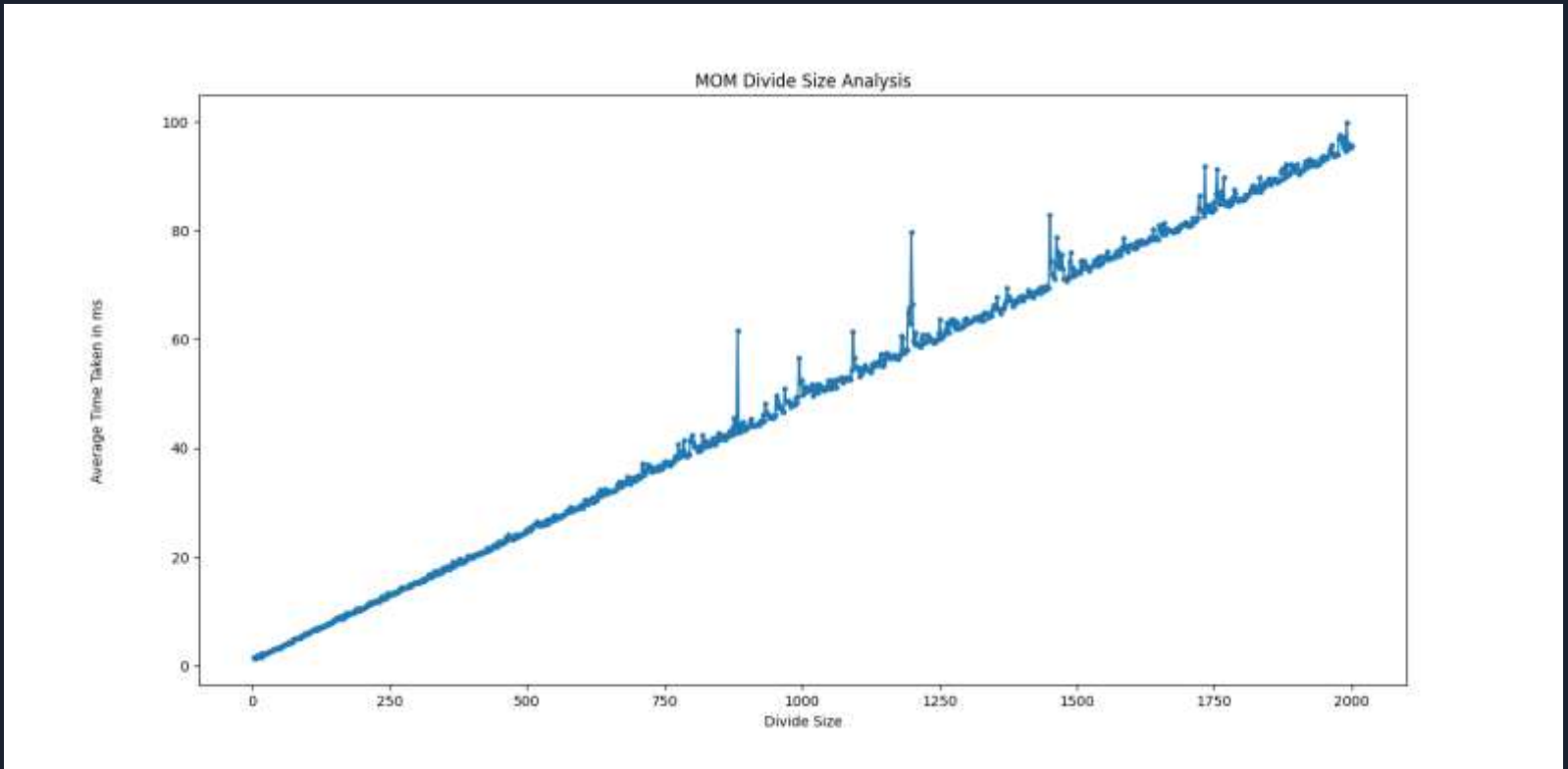
Observation



9. Take different sizes for each trivial partition (3/5/7 ...) and see how the time taken is changing.

- We performed Median of Medians algorithm on array of fixed size of 10^5 keeping the divide_size changing from 3 to 2001 consisting of odd numbers only.
- For each divide size we took 15 rounds and computed the average time taken to perform the median selection and then plotted it against the divide_size.

Observation:

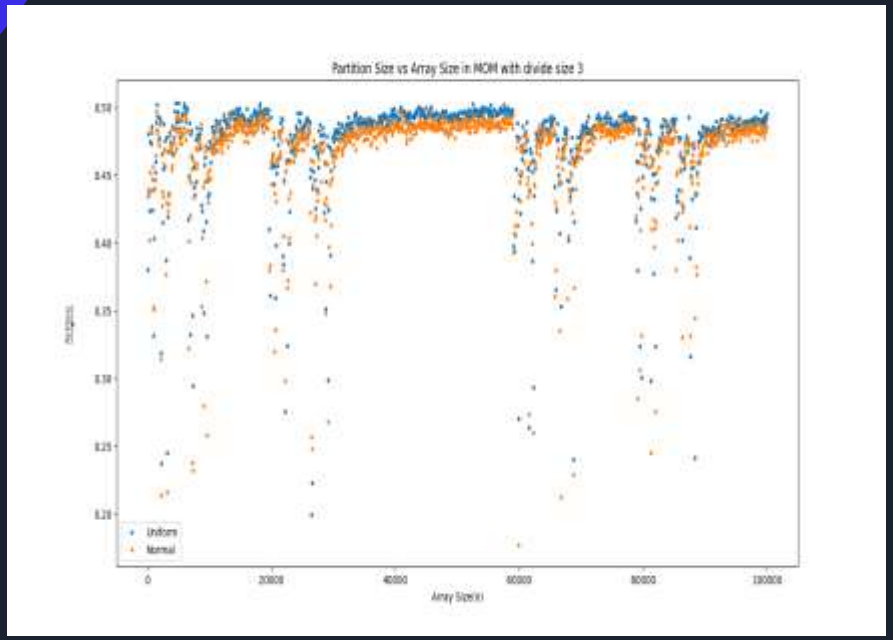


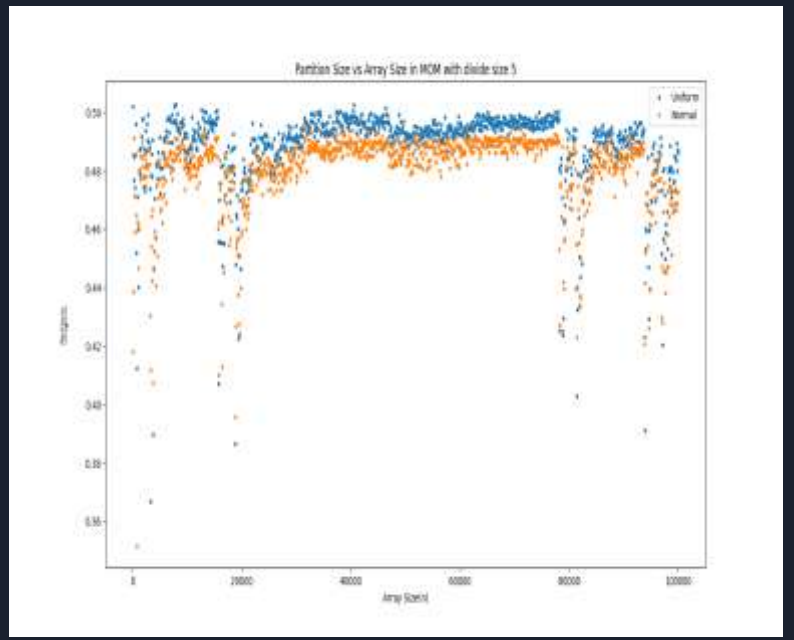
From the graph it can be inferred that the average system time taken is almost linear with the divide size.

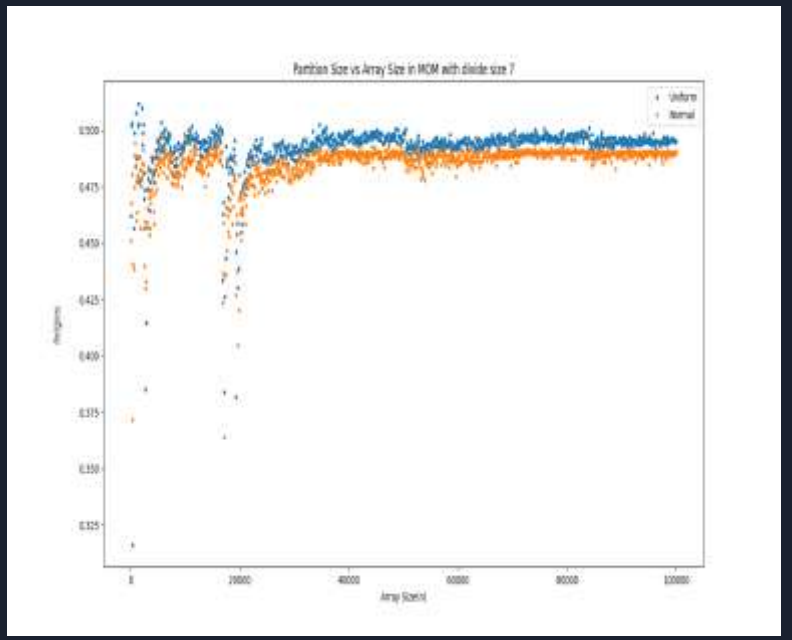
10. Perform experiments by rearranging the elements of the datasets (both UD and ND) and comment on the partition or split obtained using the pivotal element chosen as MoM.

- We perform the partitions on an array taken median of medians as the pivot element with the divide size being a constant in each observation.
- We increase the array size from 100 to 10^5 by incrementing the size by 100 in each turn. Each turn has 10 rounds, and we take the average partition size of each size and print it in a .csv file and observe.

Observation







Inferences

- There are some stark outliers at some regions in the graph for both the distributions and for all the divide sizes
- Another fact is the accuracy of MOM in case of normal distribution deprecates as compared to uniform which may be hinted due to its high density near the mean
- Divide Size 3 as compared to others gives a better accuracy for normal distribution as opposed to divide size 5 and 7.


THANK
YOU!