ALGORITHM ASSIGNMENT →1

ALGORITHM ASSIGNMENT → 1

ALGORITHM LAB

(CS2271)

ALGORITHM ASSIGNMENT → 1

Submitted by →

PRAYAS MAZUMDER

(2021CSB071)

DATE : 19/02/2023

QUESTION 1A

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

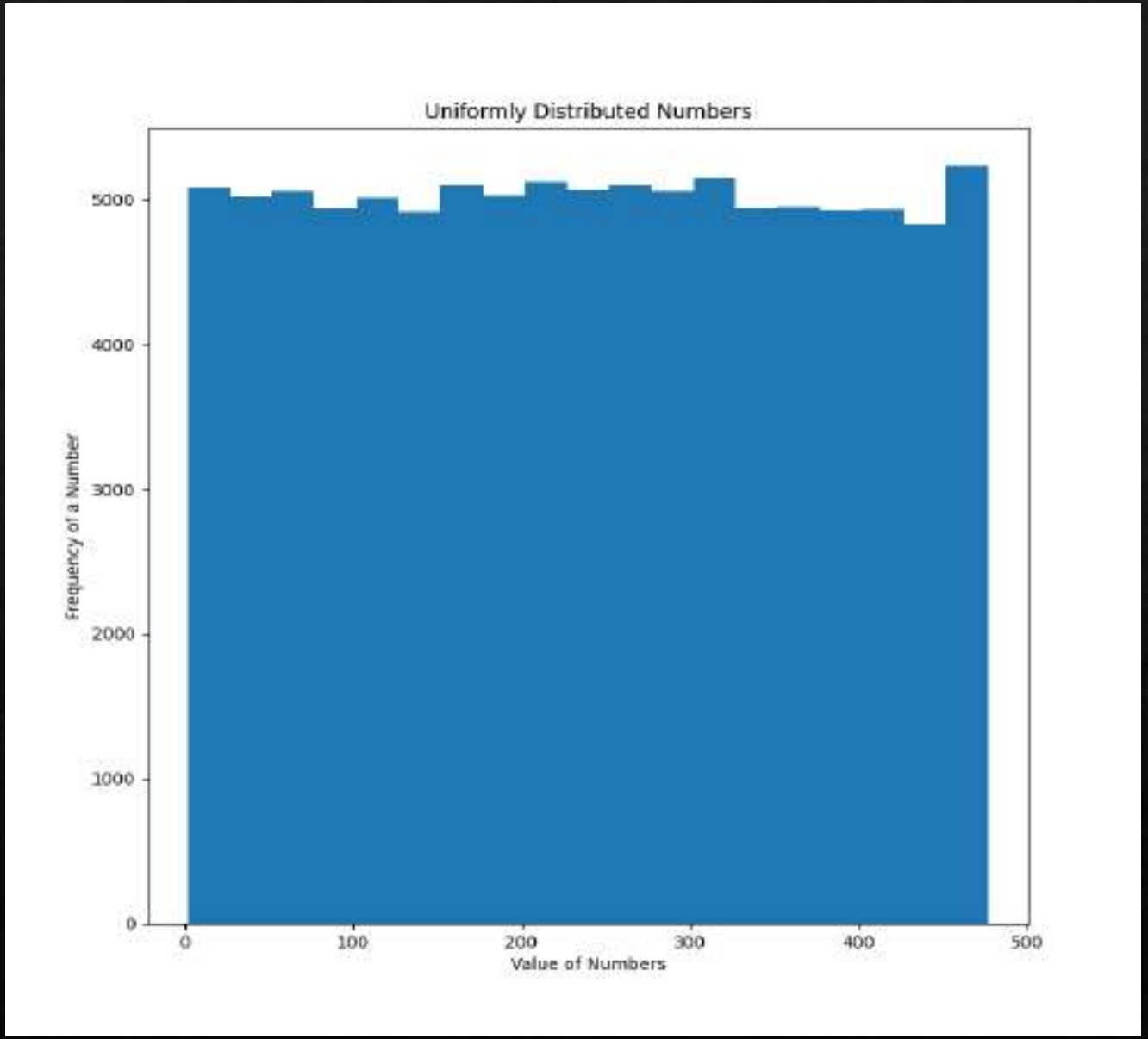
PROCEDURE TO GENERATE UNIFORM DISTRIBUTION (UD)

- The inbuilt C function `rand()` generates random numbers distributed uniformly.
- Using `rand()` we generated 10^6 uniformly distributed random numbers between 0 to (100-1)=99 and stored it in `uniform_distribution.csv` and `uniform_distribution.txt` 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 .

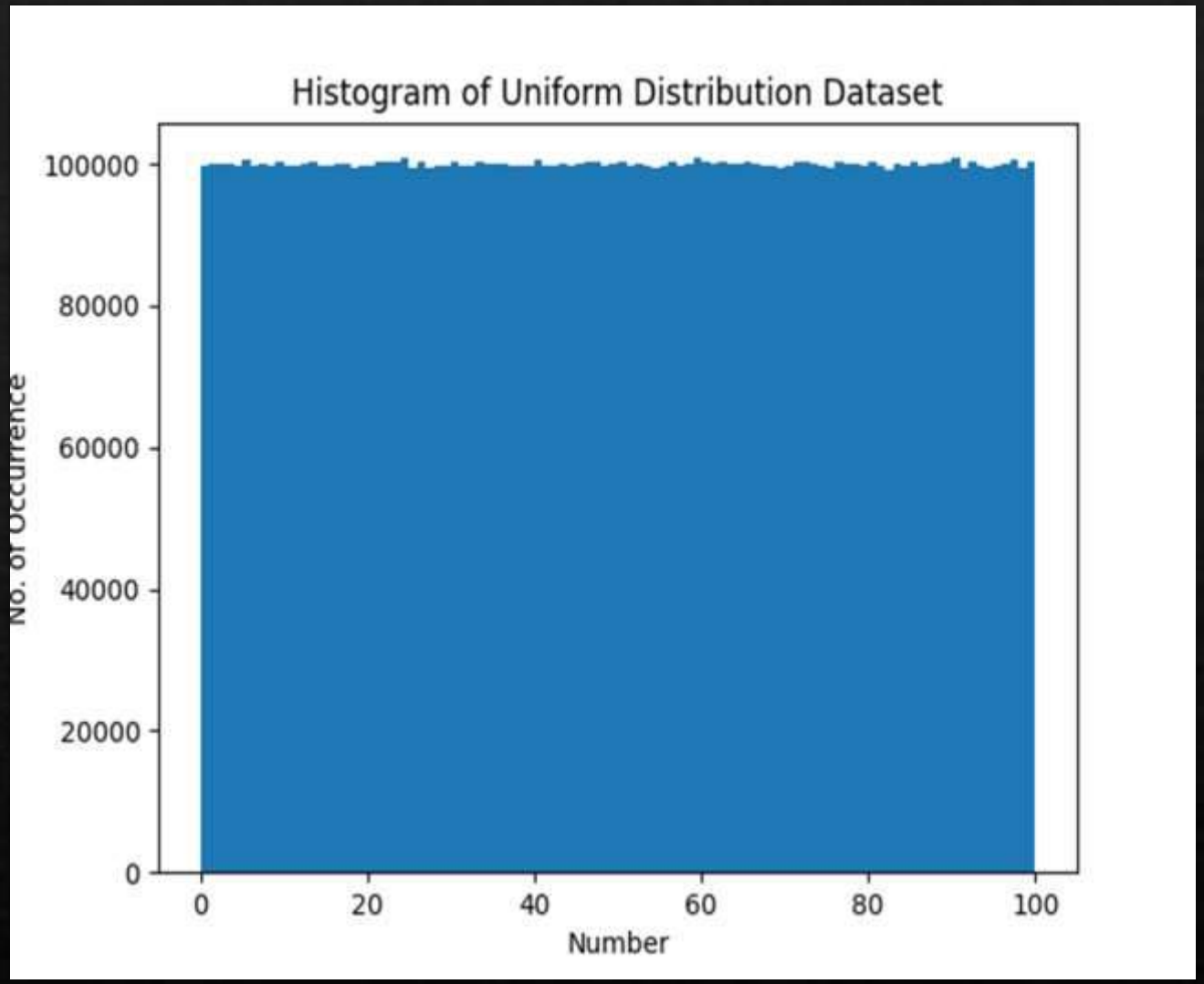
OBSERVATION

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

(NUMBERS ARE FROM 0-500)



UNIFORM DATASET DISTRIBUTION HISTOGRAM



QUESTION 1B

Construct large data sets taking random numbers from normal distribution (ND)...

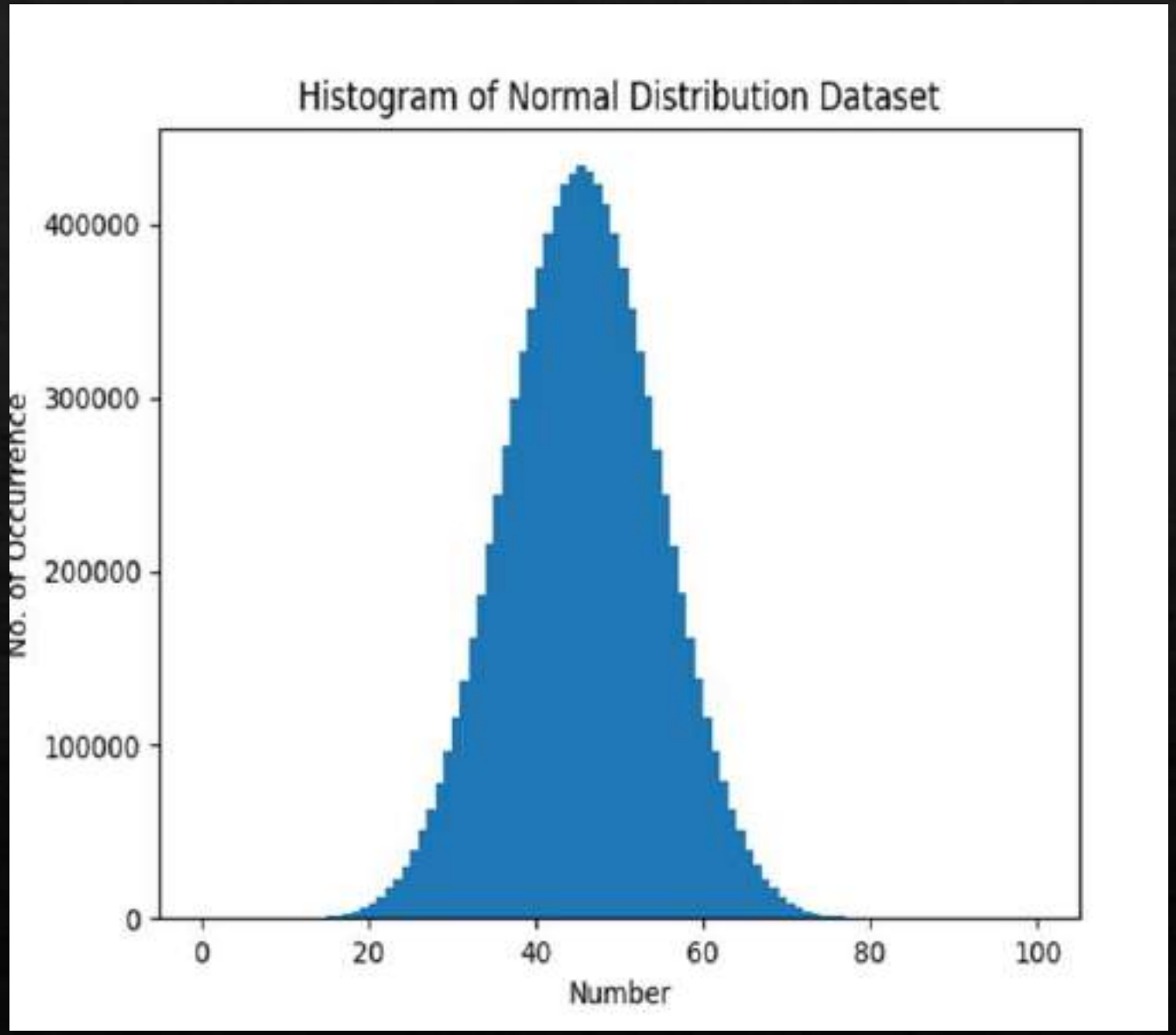
PROCEDURE TO GENERATE NORMAL DISTRIBUTION (ND)

- 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_dataset_file.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.

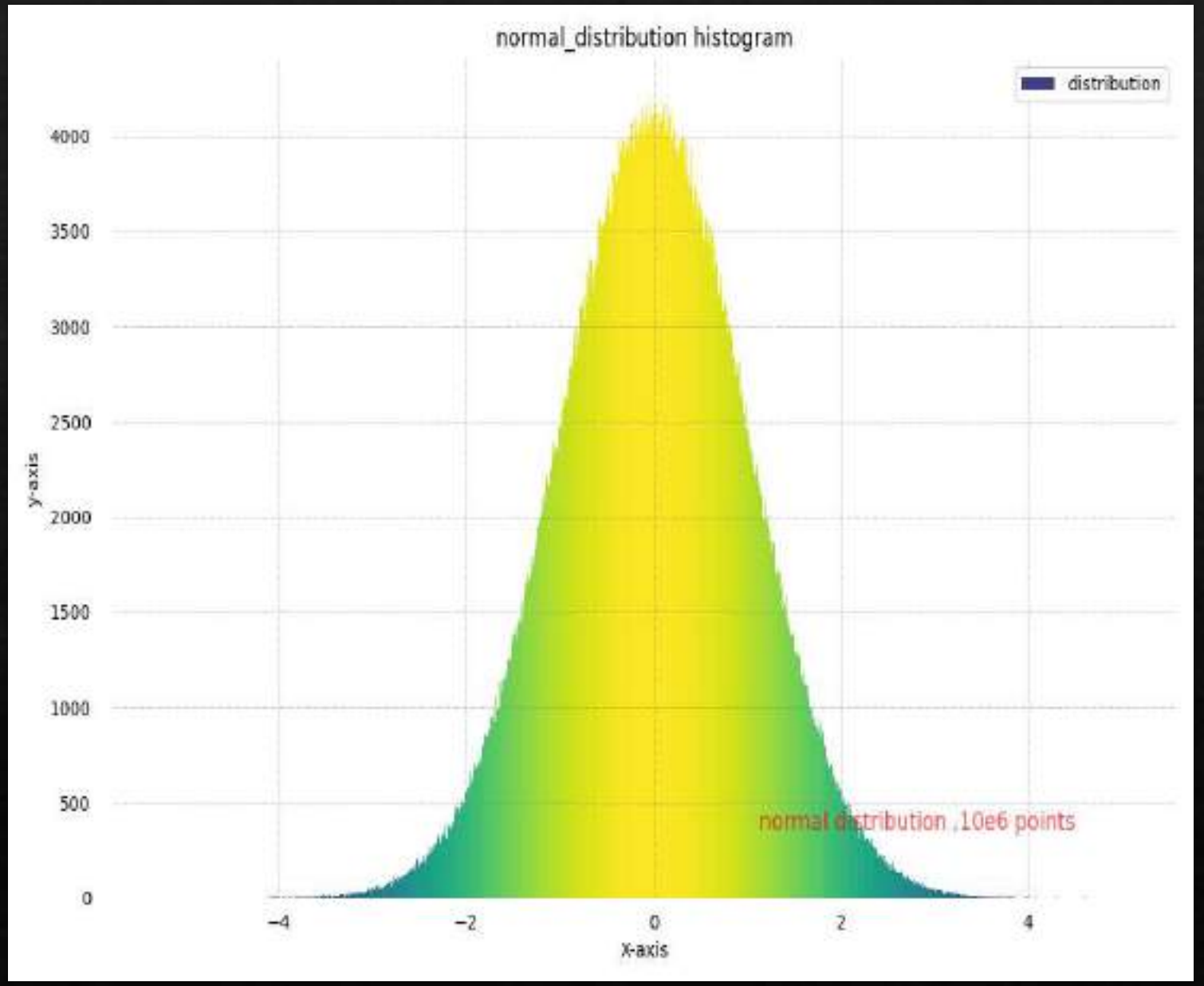
- This shows that the dataset values are normally distributed



HERE WE'VE TAKEN 10e6 POINTS AND TOTAL 1000 BINS ...

AND COLOURED THE NIRMAL DIST. GRAPH GRADIENTLY .....

THOUGH THIS IS NOT EXACTLY OUR DATASET , BUT IT LOOKS SOMEWHAT LIKE THIS ...



CODE SNIPPET PART → (1)

```
import matplotlib.pyplot as plt
import numpy as np
from matplotlib import colors
from matplotlib.ticker import PercentFormatter

# Creating dataset
np.random.seed(23752)
N_points = 1000000  # 10e6 points
n_bins = 1000  # 1k bins

# Creating distribution
x = np.random.randn(N_points)
y = .8 ** x + np.random.randn(1000000) + 25
legend = ['distribution']

# Creating histogram
fig, axs = plt.subplots(1, 1,
                        figsize=(10, 7),
                        tight_layout=True)

# Remove axes splines
for s in ['top', 'bottom', 'left', 'right']:
    axs.spines[s].set_visible(False)

# Remove x, y ticks
axs.xaxis.set_ticks_position('none')
axs.yaxis.set_ticks_position('none')

# Add padding between axes and labels
axs.xaxis.set_tick_params(pad=5)
axs.yaxis.set_tick_params(pad=10)

# Add x, y gridlines
axs.grid(b=True, color='grey',
         linestyle='-.', linewidth=0.5,
         alpha=0.6)
```

CODE SNIPPET PART →(2)

```
# Add Text watermark
fig.text(0.9, 0.15, 'normal distribution ,10e6 points',
         fontsize=12,
         color='red',
         ha='right',
         va='bottom',
         alpha=0.7)

# Creating histogram
N, bins, patches = axs.hist(x, bins=n_bins)

# Setting color
fracs = ((N**(1 / 5)) / N.max())
norm = colors.Normalize(fracs.min(), fracs.max())

for thisfrac, thispatch in zip(fracs, patches):
    color = plt.cm.viridis(norm(thisfrac))
    thispatch.set_facecolor(color)

# Adding extra features
plt.xlabel("X-axis")
plt.ylabel("y-axis")
plt.legend(legend)
plt.title('normal_distribution histogram')

# Show plot
plt.show()
```

Question 2A

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

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) ;
   }
}
```

MERGE-SORT CODE SNIPPET (1)

```
void merge(int *arr, int initial, int mid, int final)
{
    int size1 = mid - initial + 1;
    int arr1[size1];
    for (int i = 0; i < size1; i++)
    {
        arr1[i] = arr[initial + i];
    }

    int size2 = final - mid;
    int arr2[size2];
    for (int i = 0; i < size2; i++)
    {
        arr2[i] = arr[mid + 1 + i];
    }

    int arr1_pos = 0;
    int arr2_pos = 0;
    int arr_pos = initial;
    int count = 0;

    while (arr1_pos < size1 && arr2_pos < size2)
    {
        if (arr1[arr1_pos] < arr2[arr2_pos])
        {
            arr[arr_pos] = arr1[arr1_pos];
            count++;
            // return count;
            arr1_pos++;
            arr_pos++;
        }
        else
        {
            arr[arr_pos] = arr2[arr2_pos];
            count++;
            // return count;
            arr2_pos++;
            arr_pos++;
        }
    }
```

MERGE-SORT CODE SNIPPET (2)

```c
void merge_sort(int *arr, int initial, int final)
{
    if (initial < final)
    {
        int mid = (initial + final) / 2;
        merge_sort(arr, initial, mid);
        merge_sort(arr, mid + 1, final);
        merge(arr, initial, mid, final);
    }
}

// assuming 1 is true, 0 is false
int is_array_sorted(int *arr, int length)
{
    for (int i = 0; i < (length - 1); i++)
        if (arr[i] > arr[i + 1])
            return 0;

    return 1;
}
```

Question 2B

Implement Quick Sort (QS) and check for correctness

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.
- Always pick first element as pivot. (implemented below)
- Always pick last element as pivot
- Pick a random element as pivot.
- 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) ;
     }
}
```

QUICKSORT CODE SNIPPET IN C (1)

```c
void swap(int *a, int *b)
{
    int c = *a;
    *a = *b;
    *b = c;
}

int partition(int *arr, int initial, int final)
{
    int pivot_value = arr[initial];

    int left_iterator = initial;
    int right_iterator = final;

    while (true)
    {
        while (arr[left_iterator] < pivot_value)
            left_iterator++;

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

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

QUICKSORT CODE SNIPPET IN C (2)

```c
// sorts array using quick_sort algorithm, give initial as 0, and final as sizeofarray -1
void quick_sort(int *arr, int initial, int final)
{
    if (initial < final)
    {
        int pos_of_pivot = partition(arr, initial, final);
        quick_sort(arr, initial, pos_of_pivot);
        quick_sort(arr, pos_of_pivot + 1, final);
    }
}

// assuming 1 is true, 0 is false
int is_array_sorted(int *arr, int length)
{
    for (int i = 0; i < length - 1; i++)
        if (arr[i] > arr[i + 1])
            return 0;

    return 1;
}
```

QUESTION – 3

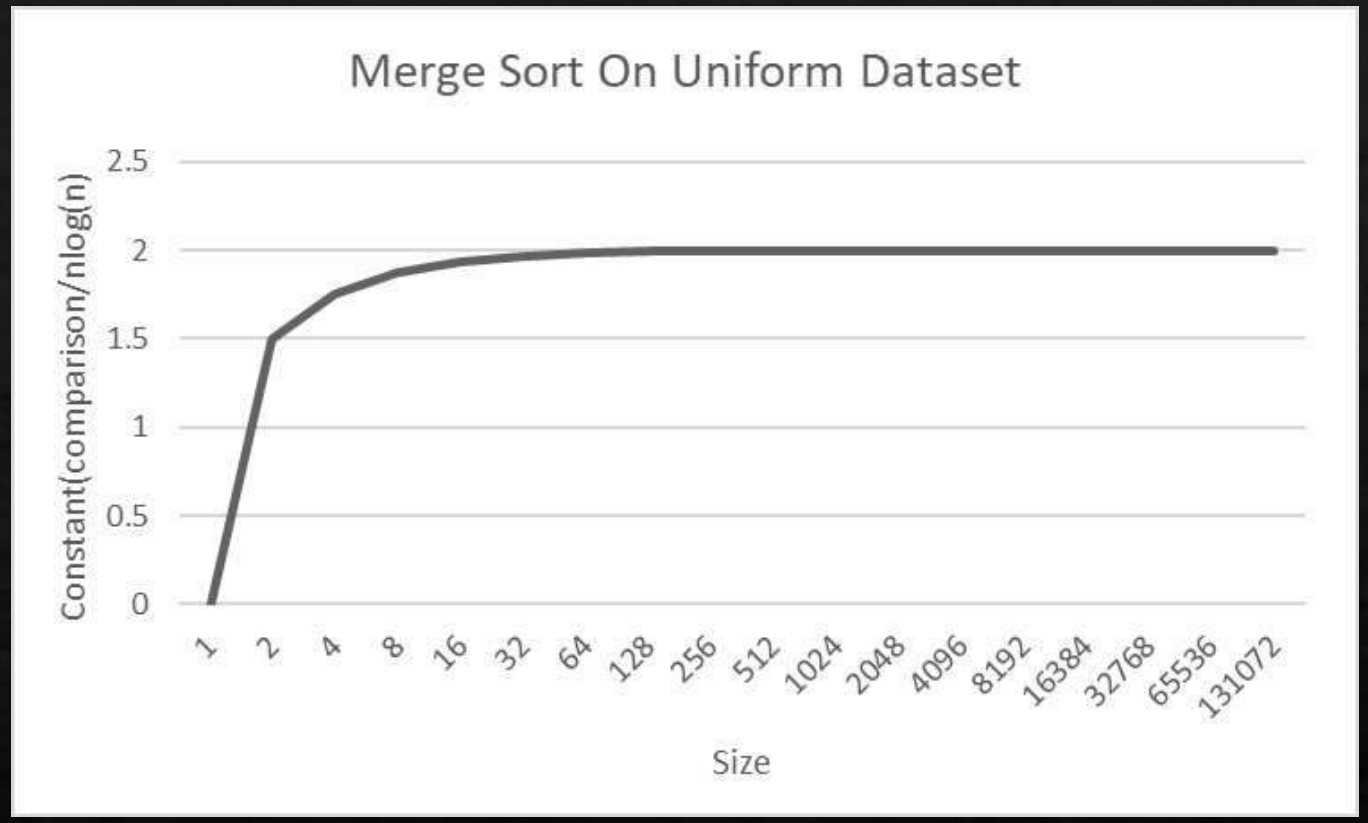
Count the operations performed, like comparisons and swaps with problem size increasing in powers of 2, for both MS and QS with both UD and ND as input data.

Procedure

- We did both sorting algorithm for different array size, which starts from 2 and increments in powers of 2 ...
  - We were able to record observation from array size 2 to array size 2^{16}
- For each array size n, we sorted ITER_NUM (variable) of 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

MERGE SORT OBSERVATION

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

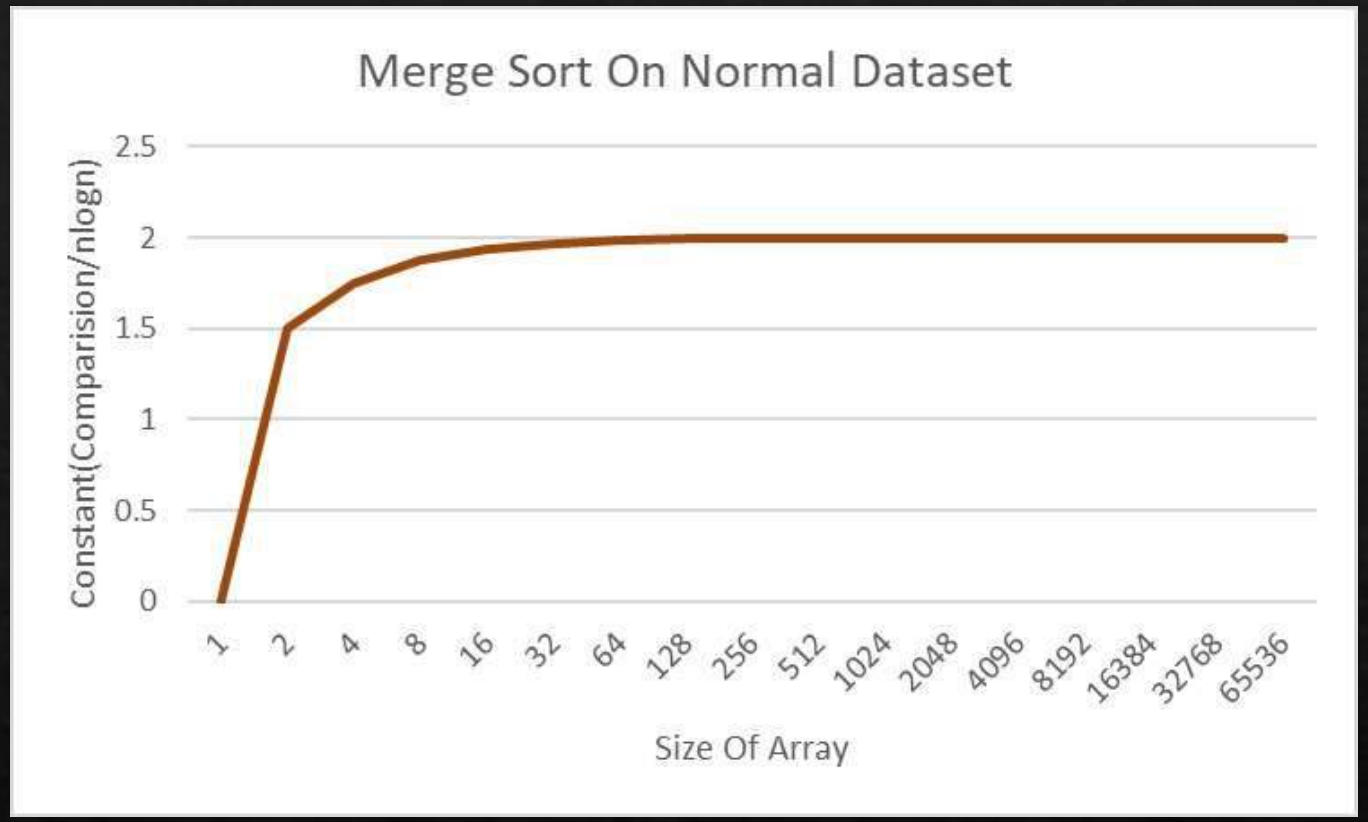


MERGE SORT OBSERVATION (2)

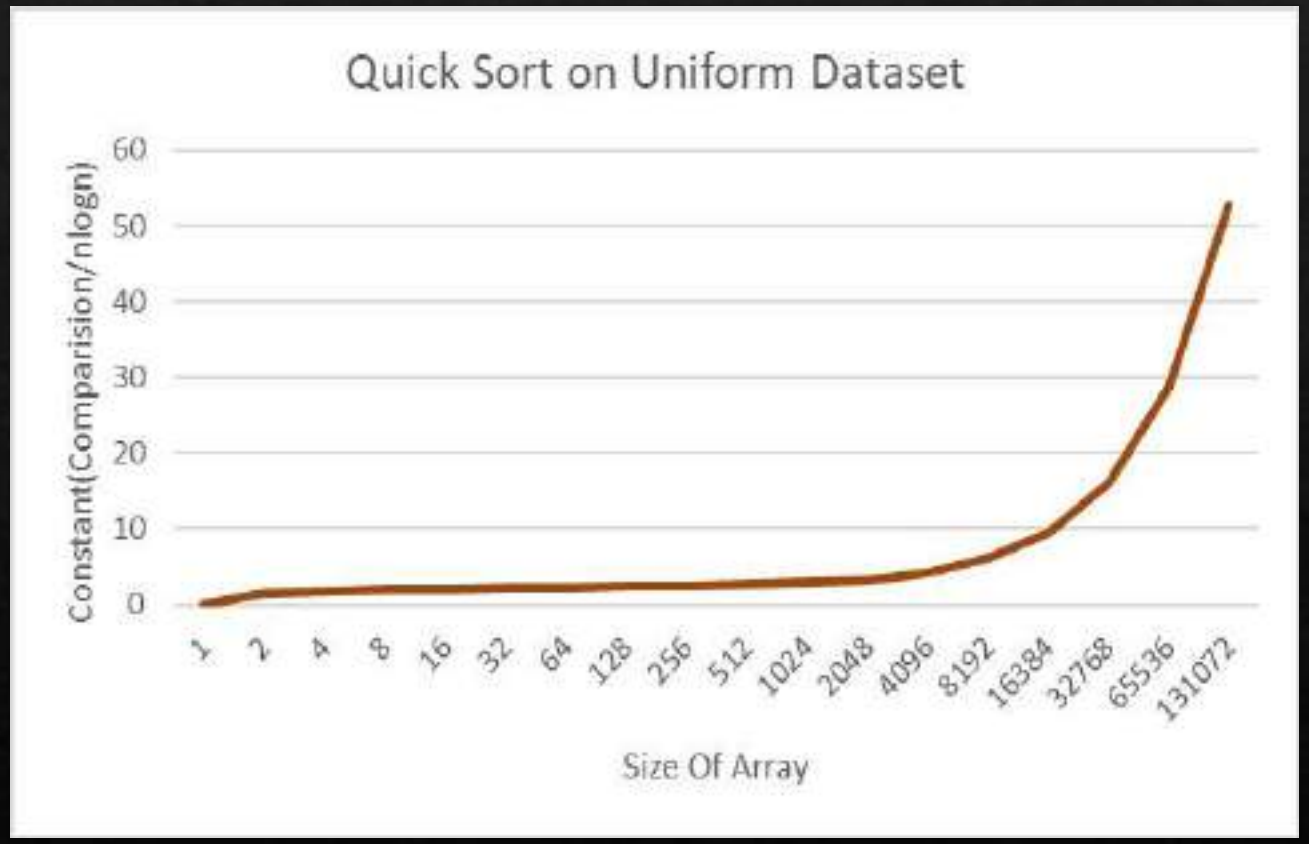
Here we see that in both ***merge sort*** normal and uniform dataset , for large dataset , i.e. for input array size in the power of two (2) , the comparison vs n(log n) ratio tends to be a constant value ….

That depicts that merge sort algorithm runs on the complexity of n(log n) ….

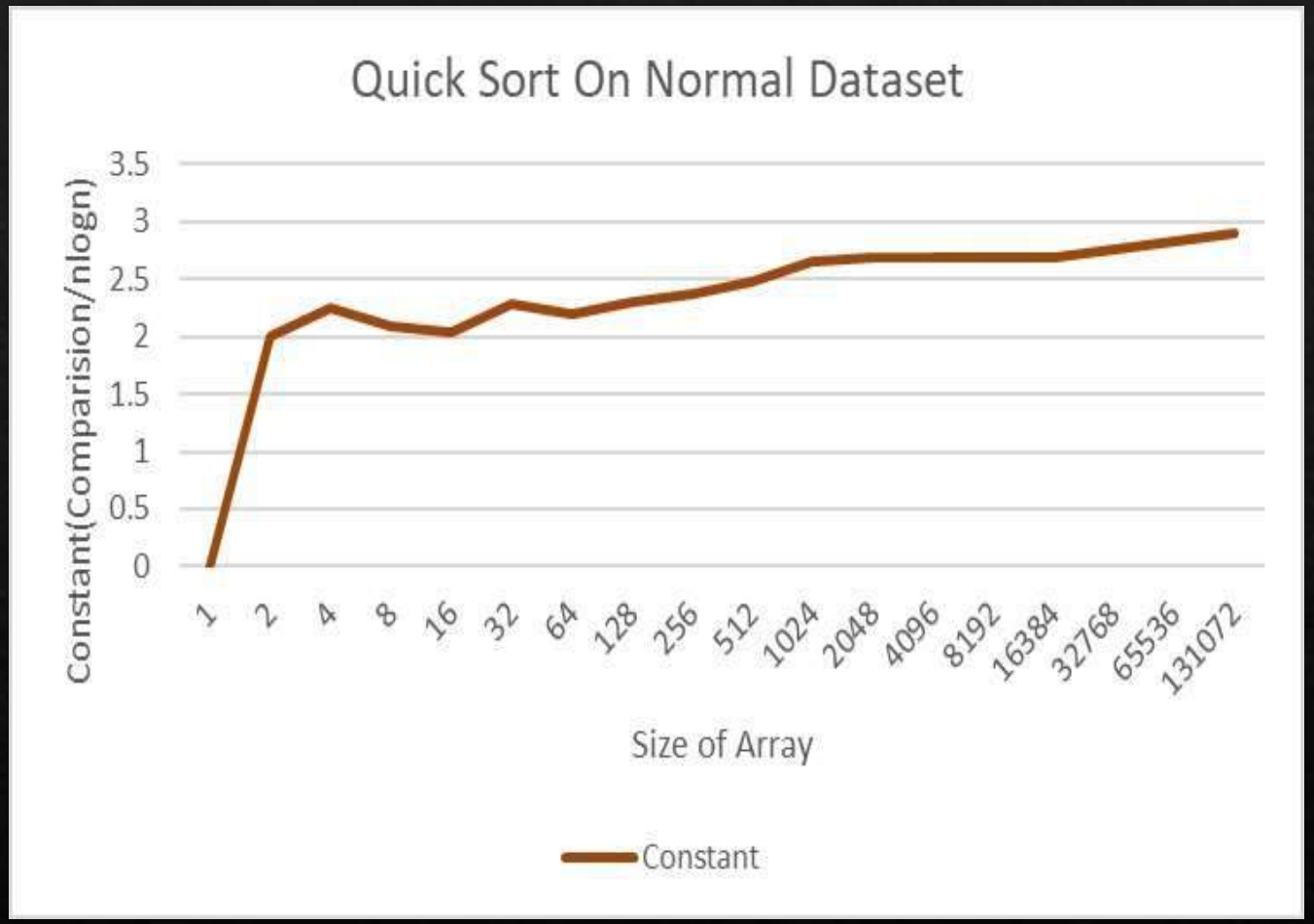
The value of the leading constant is almost here according to this graph …



QUICK SORT OBSERVATION (1)



QUICK SORT OBSERVATION (2)



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?

BEFORE WE START RANDOMIZED-

QUICKSORT , LET'S SEE WHAT RANDOMIZED ALGORITHM ACTUALLY MEAN ..

- 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.
- There are different types of randomized algorithms, including Monte Carlo algorithms, Las Vegas algorithms, and randomized approximation algorithms. Monte Carlo algorithms use random numbers to approximate the solution to a problem, while Las Vegas algorithms use randomization to speed up the computation and ensure that the algorithm always terminates. Randomized approximation algorithms use randomization to find an approximate solution to a problem that is close to the optimal solution.

Randomized Quicksort Algorithm

The new partitioning

procedure, RANDOMIZED-PARTITION, simply swaps before performing the

partitioning. The new quicksort procedure, RANDOMIZED-QUICKSORT, calls

RANDOMIZED-PARTITION instead of PARTITION.

- ***RANDOMIZED-PARTITION (A , p, r)***

```
1 i = RANDOM(p, r)
2 exchange A[r] with A[i]
3 return PARTITION(A, p, r)
```

- ***RANDOMIZED-QUICKSORT (A, p, r)***

```
1 if p < r
2 q = RANDOMIZED-PARTITION (A, p, r)
3 RANDOMIZED-QUICKSORT (A, p, q - 1)
4 RANDOMIZED-QUICKSORT (A, q + 1, r)
```

Time Complexity of Randomized Quicksort

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.

- ***Worst case scenario*** :

The worst-case time complexity of the randomized quicksort algorithm is O(n^2), where n is the number of elements to be sorted.

The worst-case scenario occurs when the pivot element chosen by the algorithm is the smallest or largest element in the array, and the partitioning routine always partitions the array in a way that leaves one partition with n-1 elements and the other partition with only one element. In this case, the algorithm must make n-1 recursive calls to sort the partition with n-1 elements, resulting in a total of n*(n-1) comparisons.

PROCEDURE

- We did Randomized Quicksort for different array size, which starts from 1 and increments in powers of 2 ...
- We recorded observation from array size 1(2^0) to array size 2^16 ...
- For each array size n, we sorted ITER_NUM 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

  $comparision\ ratio = (avg\_comp)/nlgn$ vs n

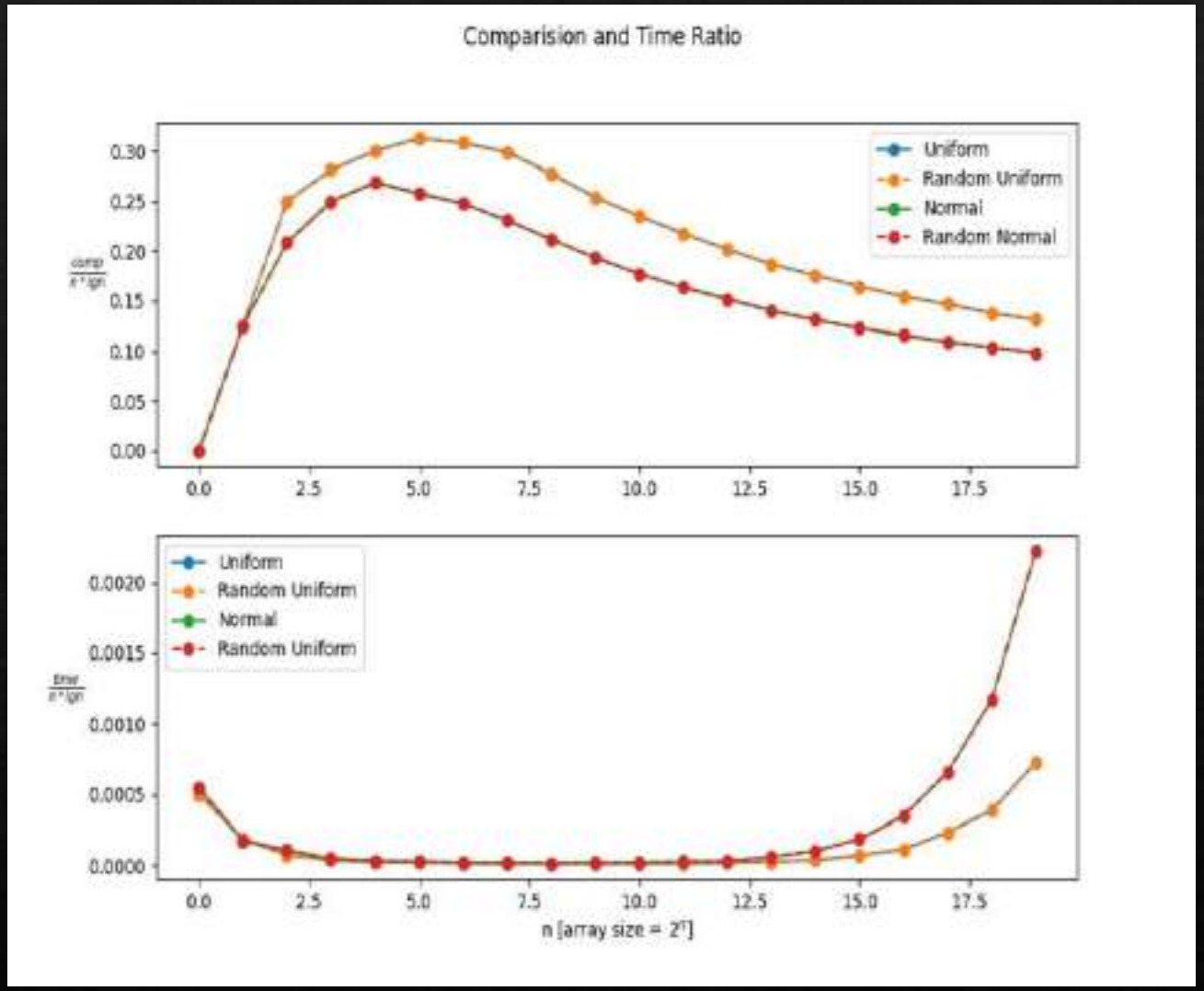
  $time\ ratio = (avg\_time\_taken)/nlgn$ vs n
- We also plotted the usual Quick Sort Observations for comparisons too.

Observation

Following Inference can be made by the Observed Graph:

As we did ITER_NUM sorts per n, and took its mean, the time complexity comes out to be same as Non-Random Quick Sort, i.e., $O(nlgn)$

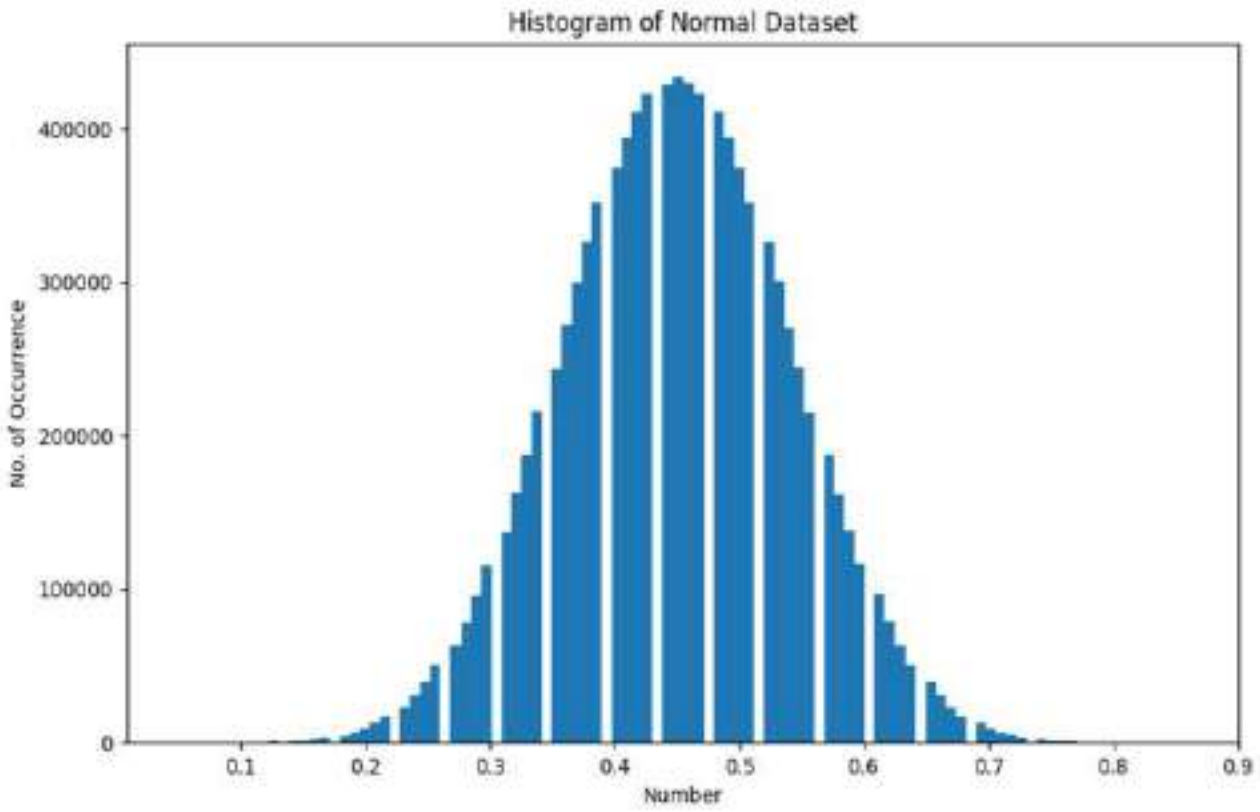
As array size keep increasing, we see similar observation, i.e., the comparison ratio and time ratio converges.



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

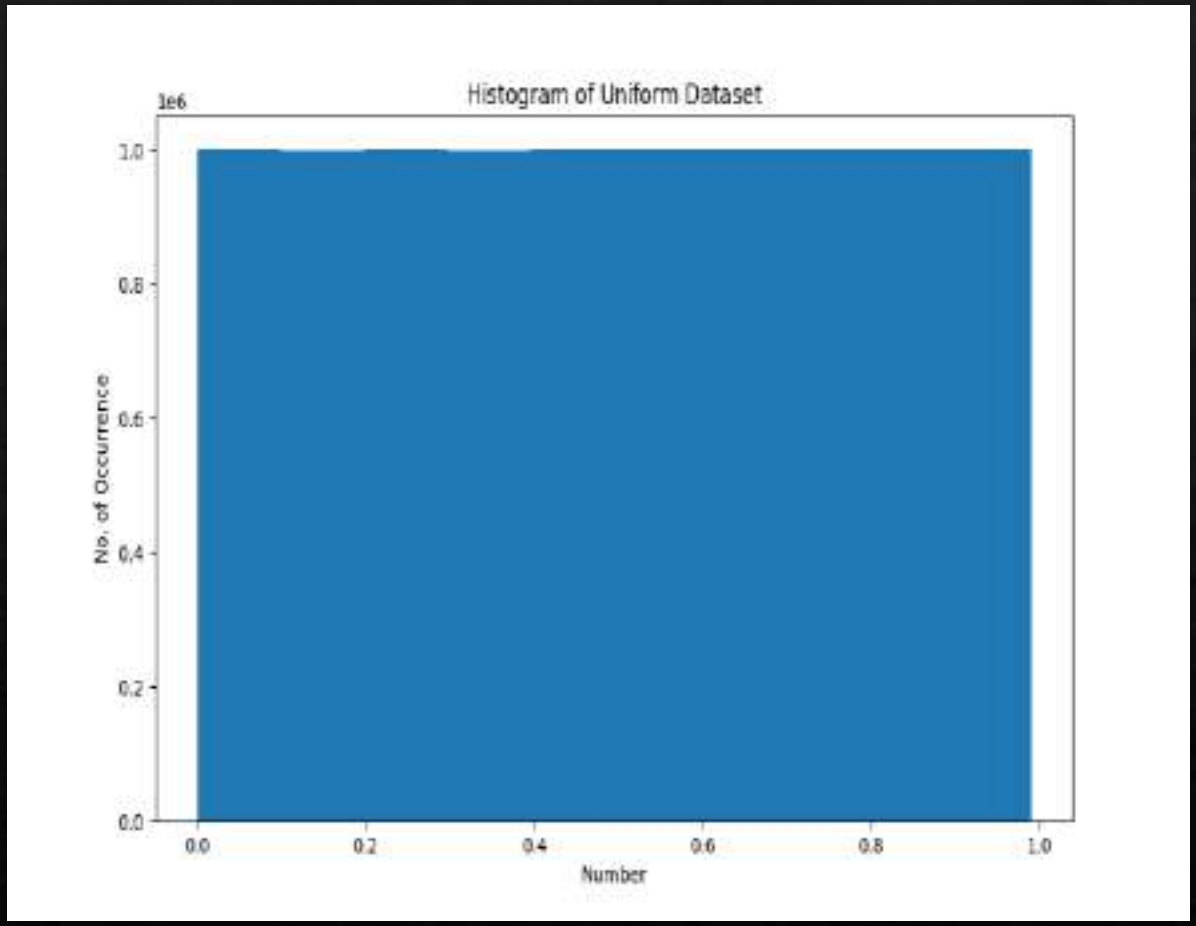


- As All Values in our dataset was between 0 and 100, we took each value one by one, divided each value by 100, and then saved it to a different files.
- The no. of data in the dataset was kept same, i.e. 10^6 = 1000000
- We Then Plotted the histogram of the datasets to make sure the conversion was correct.

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

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(A, n)
1  let B[0 : n − 1] be a new array
2  for i = 0 to n − 1
3      make B[i] an empty list
4  for i = 1 to n
5      insert A[i] into list B[⌊n · A[i]⌋]
6  for i = 0 to n − 1
7      sort list B[i] with insertion sort
8  concatenate the lists B[0], B[1], . . . , B[n − 1] together in order
9  return the concatenated lists
```

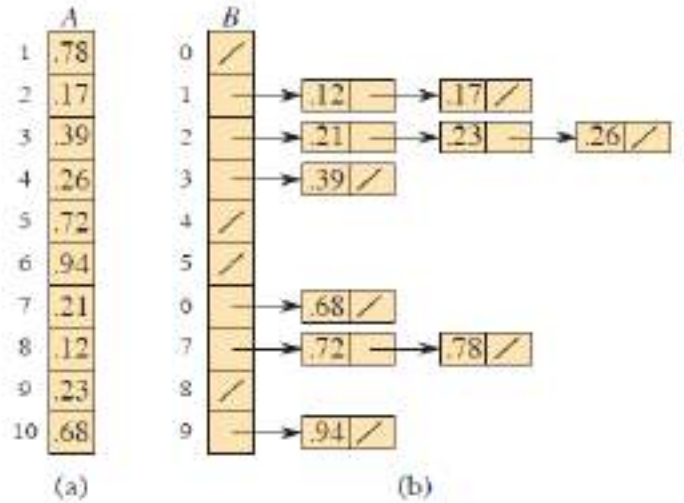


Figure 8.4 The operation of BUCKET-SORT for $n = 10$. **(a)** The input array $A[1 : 10]$. **(b)** The array $B[0 : 9]$ of sorted lists (buckets) after line 7 of the algorithm, with slashes indicating the end of each bucket. Bucket i holds values in the half-open interval $[i/10, (i + 1)/10)$. The sorted output consists of a concatenation of the lists $B[0], B[1], \ldots, B[9]$ in order.

Bucket Sort time complexity

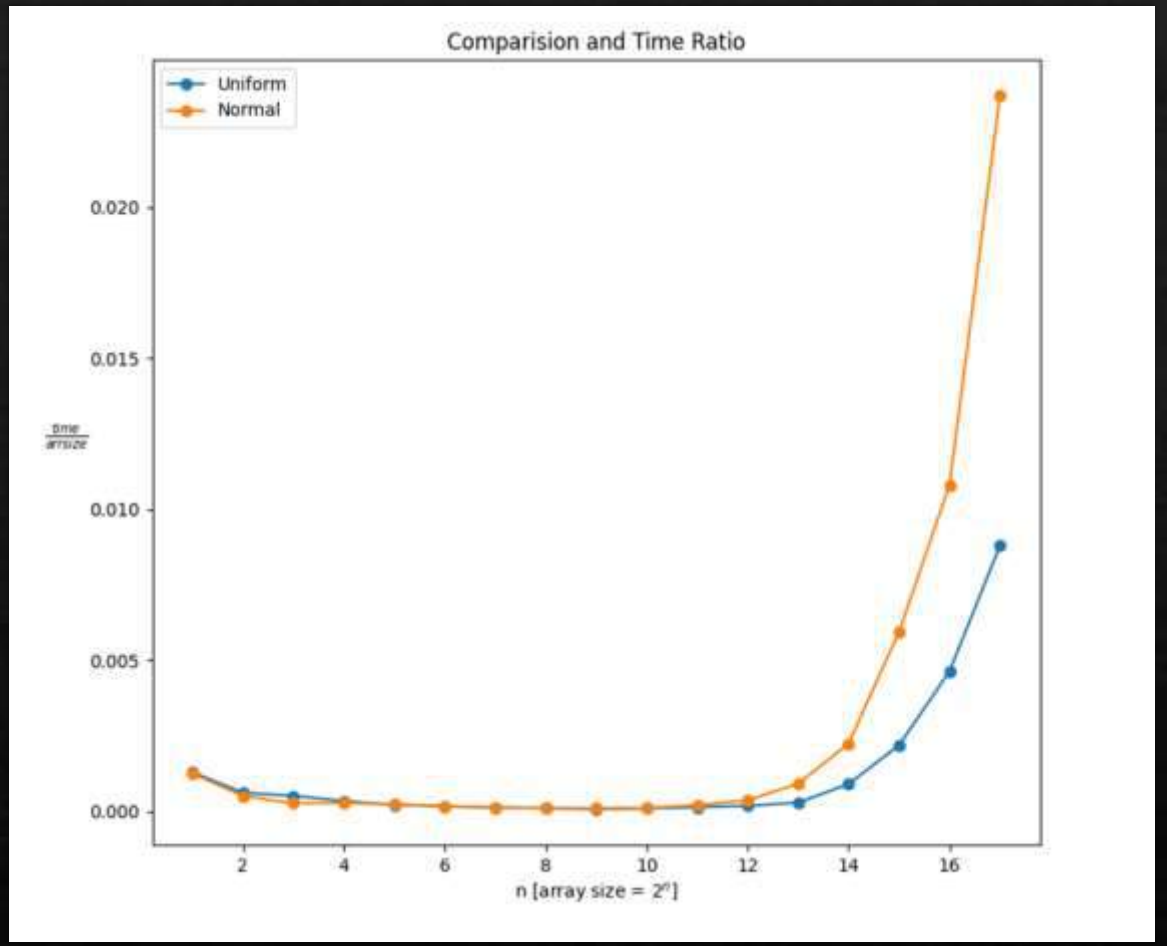
The time complexity of bucket sort depends on the sorting algorithm used to sort the individual buckets.

Let's assume that we use an efficient sorting algorithm like Quicksort or Merge sort to sort the individual buckets. The time complexity of bucket sort can then be expressed as follows →

- **Best case**: O(n+k), where n is the number of elements in the array to be sorted, and k is the number of buckets.
- **Average case**: O(n+k), assuming that the elements are uniformly distributed across the buckets.
- **Worst case**: O(n^2), which occurs when all the elements are placed in the same bucket, and the sorting algorithm used for that bucket has a worst-case time complexity of O(n^2).

Bucket Sort implementation 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


Comparision and Time Ratio
Uniform
Normal
0.020
0.015
0.010
0.005
0.000
2
4
6
8
10
12
14
16
n [array size = 2^n]

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.

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 Code Snippet (1)

```
//give arr_size as sizeof(arr)/sizeof(int)
int median_of_median(short arr[], int arr_size, int divide_size)
{
    if (arr_size < divide_size)
    {
        int median = give_median(arr, 0, arr_size - 1);
        return median;
    }

    int no_full_group = arr_size / divide_size;
    int elements_in_last = arr_size % divide_size;

    int next_arr_size;

    if (elements_in_last == 0)
        next_arr_size = no_full_group;
    else
        next_arr_size = no_full_group + 1;

    short next_arr[next_arr_size];

    for (int i = 0; i < next_arr_size; i++)
    {
        if (i == next_arr_size -1)
            next_arr[i] = give_median(arr, divide_size * i, arr_size - 1);
        else
            next_arr[i] = give_median(arr, divide_size * i, divide_size * (i + 1) - 1);
    }

    return median_of_median(next_arr, next_arr_size,divide_size);
}
```

Median Of Median Code Snippet (2)

```
// all position should be given according to zero indexed array
void insertion_sort(short arr[], int initial, int final)
{
    for (int i = initial; i <= final; i++)
    {
        int value = arr[i];
        int pos = i - 1;
        while (pos >= initial && arr[pos] > value)
        {
            arr[pos + 1] = arr[pos];
            pos--;
        }
        arr[pos + 1] = value;
    }
}

// all position should be given according to zero indexed array
int give_median(short arr[], int initial, int final)
{
    insertion_sort(arr, initial, final);
    int mid = (initial + final) / 2;
    return arr[mid];
}
```

MoM algorithm procedure

1. Divide n elements into $\frac{n}{divide\_size}$ groups of $divide\_size$ element each, and $n \% divide\_size$ 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.

Complexity Of Median Of Median algorithm ...

$O(n)$ for worst case (linear time) ...

Average case ***: O(n)***

Space complexity is : O(nlogn)

Question 8

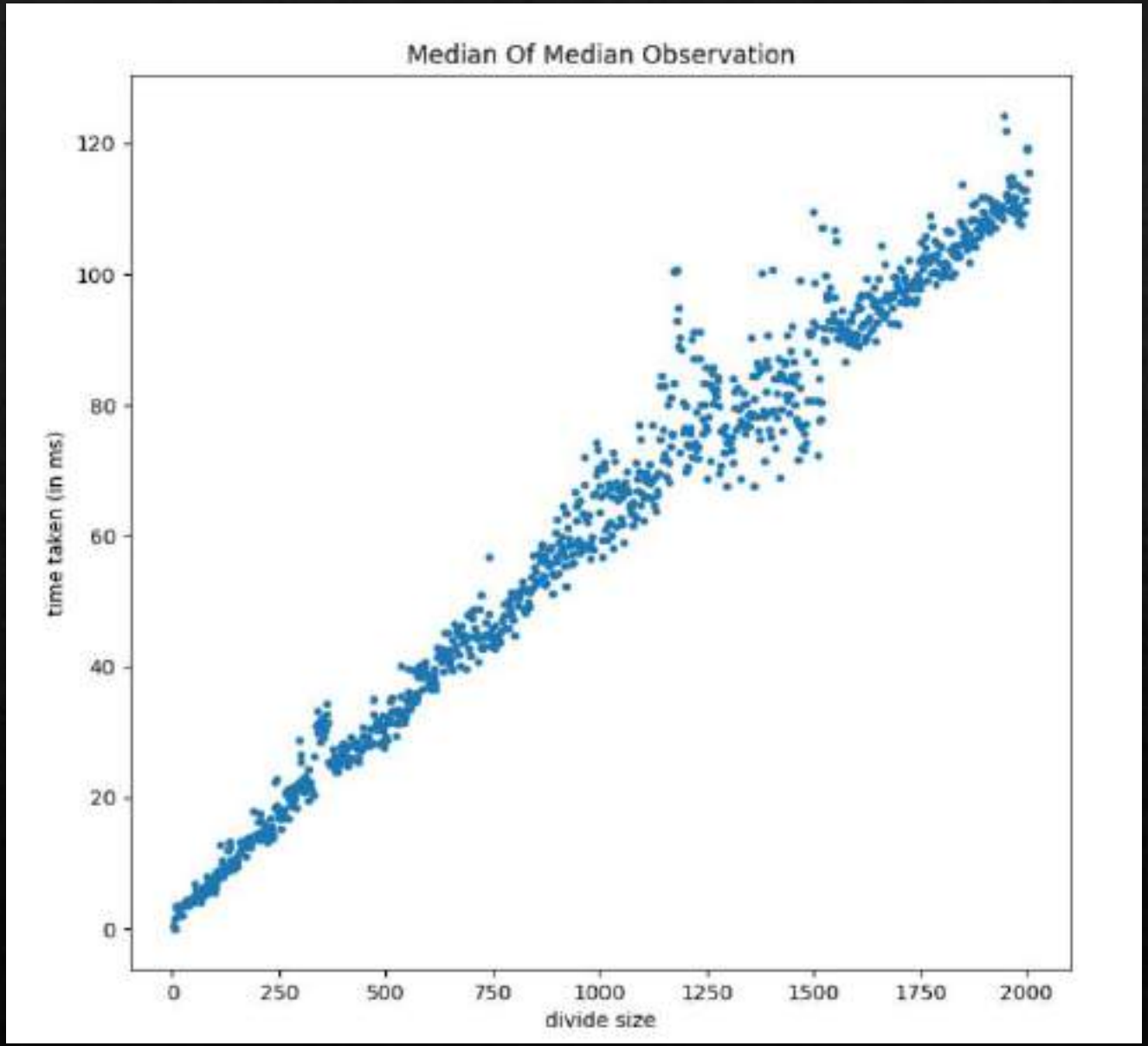
Take different sizes for each trivial partition (3/5/7 ...) and see how the time taken is changing.

Procedure for varying divide size ...

- We did Median of Median for fixed array size of 10^5 elements, but varied the divide size in $\{3,5,7,9,\ldots,2003\}$ (max_divide_size = 2003)
- For each divide size, we computed Median Of Medians 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

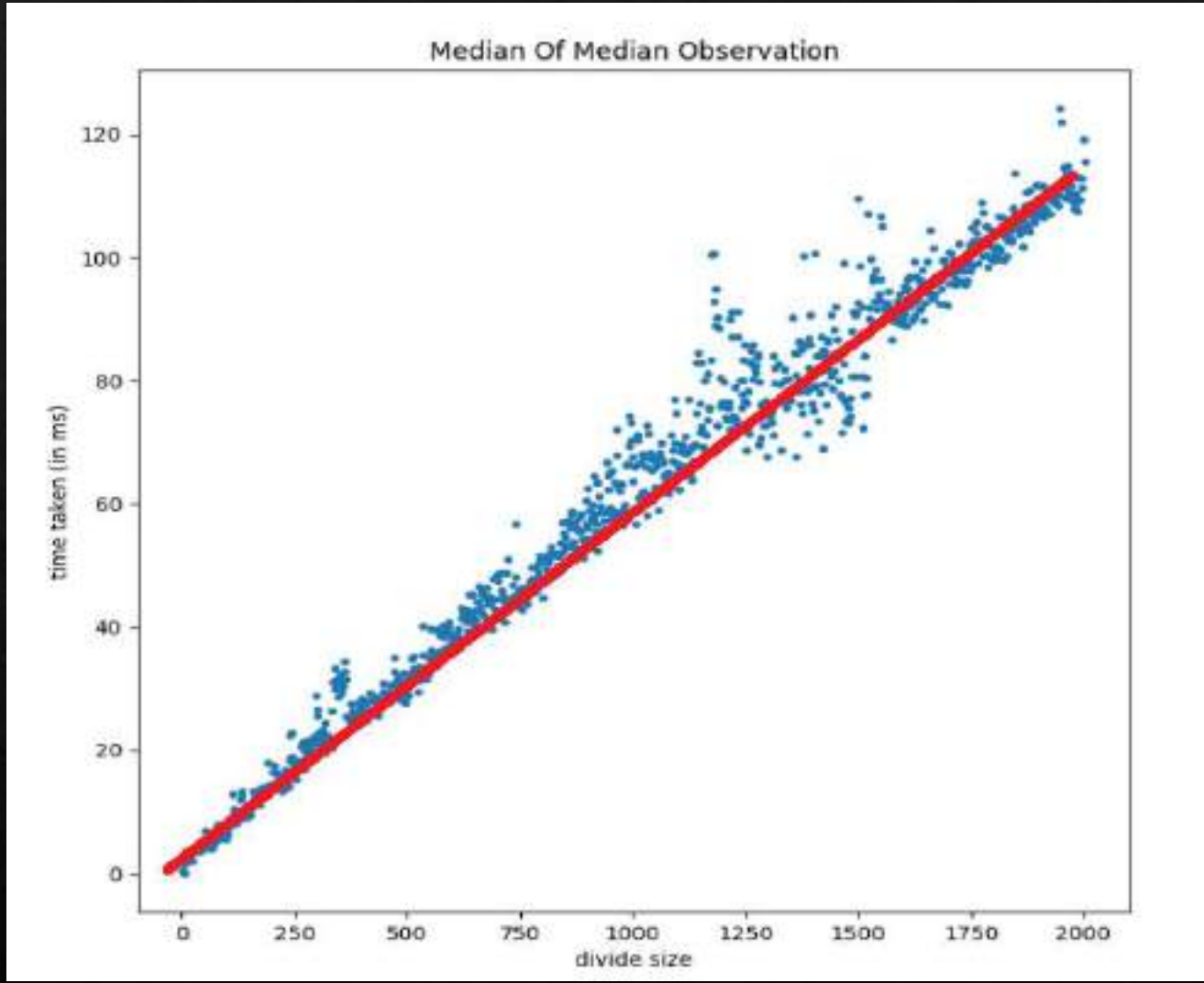
OBSERVATION

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



OBS . (2)

A STRAIGHT LINE IS DRAWN TO SHOW THAT LINEARITY



THANK YOU