ALGORITHMS LAB (CS-2271)

ARITRA BANDYOPADHYAY
(2021CSB107)

BIPRADEEP BERA
(2021CSB109)

1-A: CONSTRUCT LARGE DATASETS TAKING RANDOM NUMBERS FROM UNIFORM DISTRIBUTION (UD)

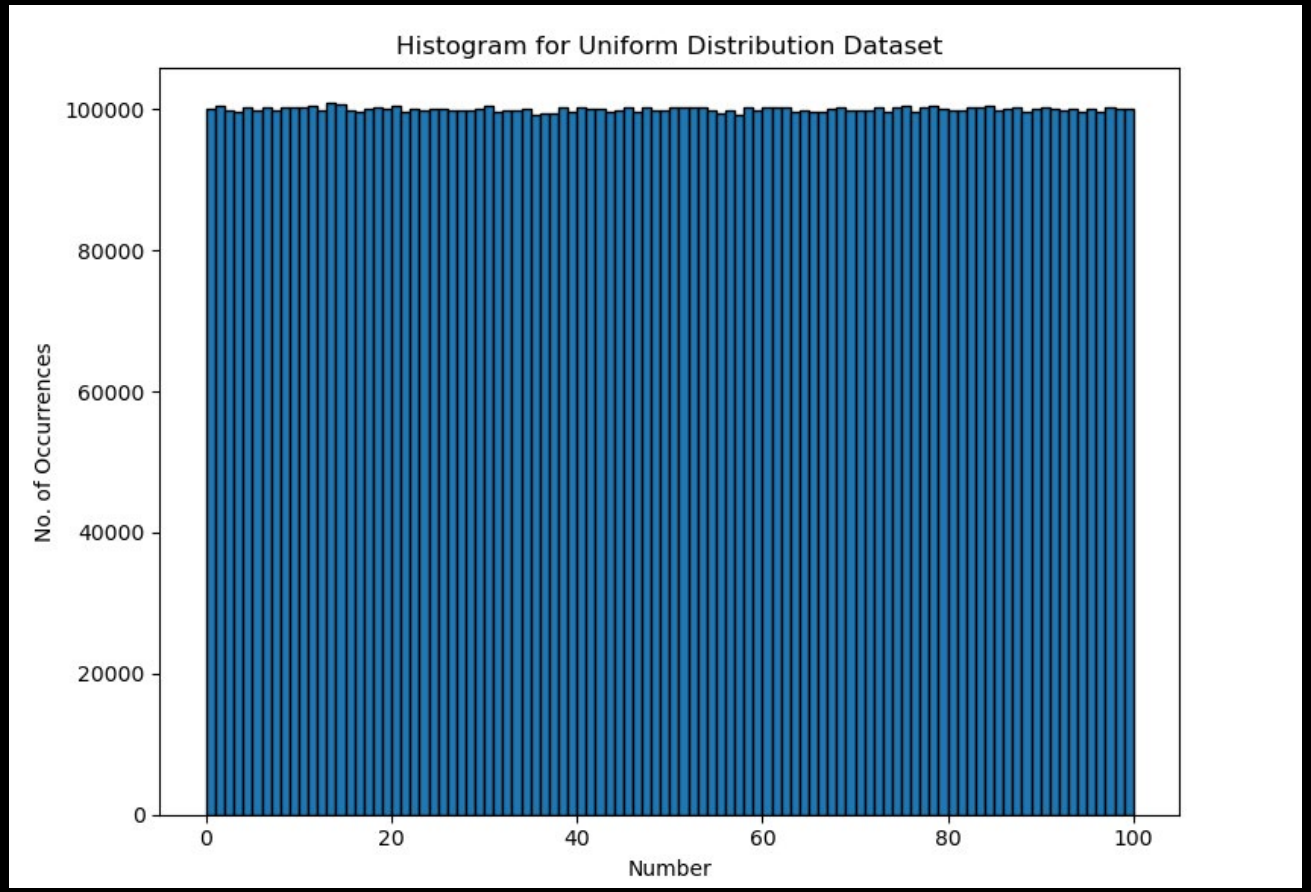
APPROACH AND CODE IN C

- The rand() function in C was used to generate 1 million pseudorandom numbers
- “seconds” value of the PC clock was used as the seed from the PRNG
- Dataset was saved as a .csv file

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

const int ELEMENTS = 10e6;

int main(void) {

    srand(time(0));
    int rangeHigh = 100;
    int rangeLow = 0;
    int range = rangeHigh - rangeLow;
    FILE *f = fopen("rand_uniform.csv", "w");

    for (int i = 0; i < ELEMENTS; i++) {
        int rand_int = rand() % range + rangeLow;
        fprintf(f, "%d\n", rand_int);
    }
    fclose(f);
}
```

GRAPH OF UD DATASET, PLOTTER CODE

```python
import numpy as np
import csv
import matplotlib.pyplot as plt


random_numbers = np.zeros(int(10e6))

with open('./rand_uniform.csv') as csv_file:
    csv_reader = csv.reader(csv_file, delimiter=',')
    i = 0
    for row in csv_reader:
        random_numbers[i] = row[0]
        i += 1

plt.figure("Histogram for Uniform Distribution Dataset")
plt.title("Histogram for Uniform Distribution Dataset")
plt.xlabel("Number")
plt.ylabel("No. of Occurrences")
plt.hist(random_numbers, bins=range(101), ec='black')
plt.show()
```



1-B: CONSTRUCT LARGE DATASETS TAKING RANDOM NUMBERS FROM NORMAL DISTRIBUTION (ND)

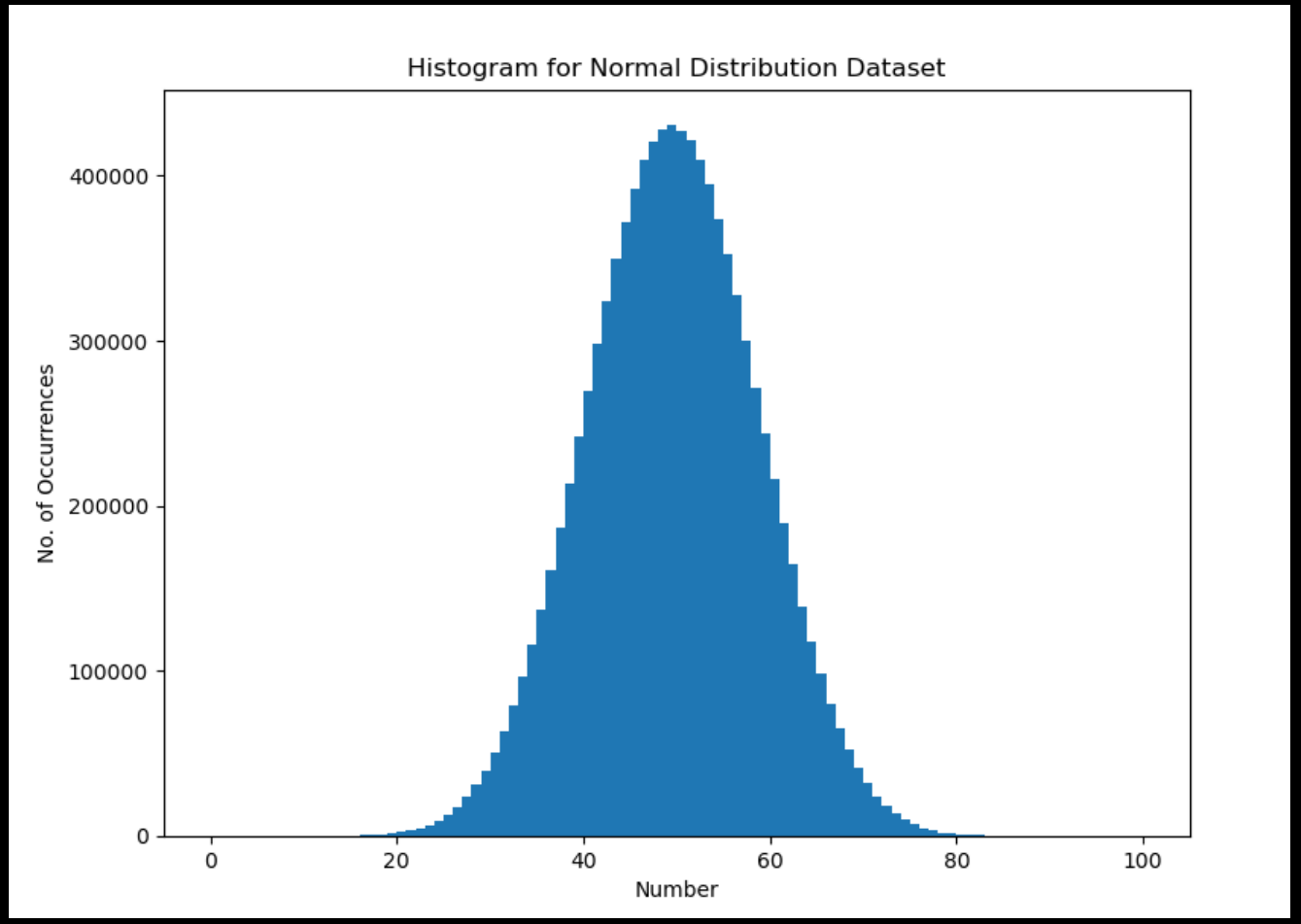
APPROACH AND CODE IN C

- “seconds” value of the PC clock was used as the seed from the PRNG
- To generate a single number from normal distribution, 10 random numbers were taken and averaged
- 1 million such averages were done to generate ND dataset
- Dataset was saved as a .csv file

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

const int ELEMENTS = 10e6;

int main(void) {

    srand(time(0));
    int rangeHigh = 100;
    int rangeLow = 0;
    int range = rangeHigh - rangeLow;
    FILE *f = fopen("rand_normal.csv", "w");

    for (int i = 0; i < ELEMENTS; i++) {

        int sum = 0;
        for (int j = 0; j < 10; j++) {
            int rand_int = rand() % range + rangeLow;
            sum += rand_int;
        }
        sum /= 10;
        fprintf(f, "%d\n", sum);
    }
    fclose(f);
}
```

GRAPH OF ND DATASET, PLOTTER CODE

```python
import numpy as np
import csv
import matplotlib.pyplot as plt


random_numbers = np.zeros(int(10e6))

with open('./rand_normal.csv') as csv_file:
    csv_reader = csv.reader(csv_file, delimiter=',')
    i = 0
    for row in csv_reader:
        random_numbers[i] = row[0]
        i += 1

plt.figure("Histogram for Normal Distribution Dataset")
plt.title("Histogram for Normal Distribution Dataset")
plt.xlabel("Number")
plt.ylabel("No. of Occurrences")
plt.hist(random_numbers, bins=range(101))
plt.show()
```



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

THE MERGE SORT ALGORITHM

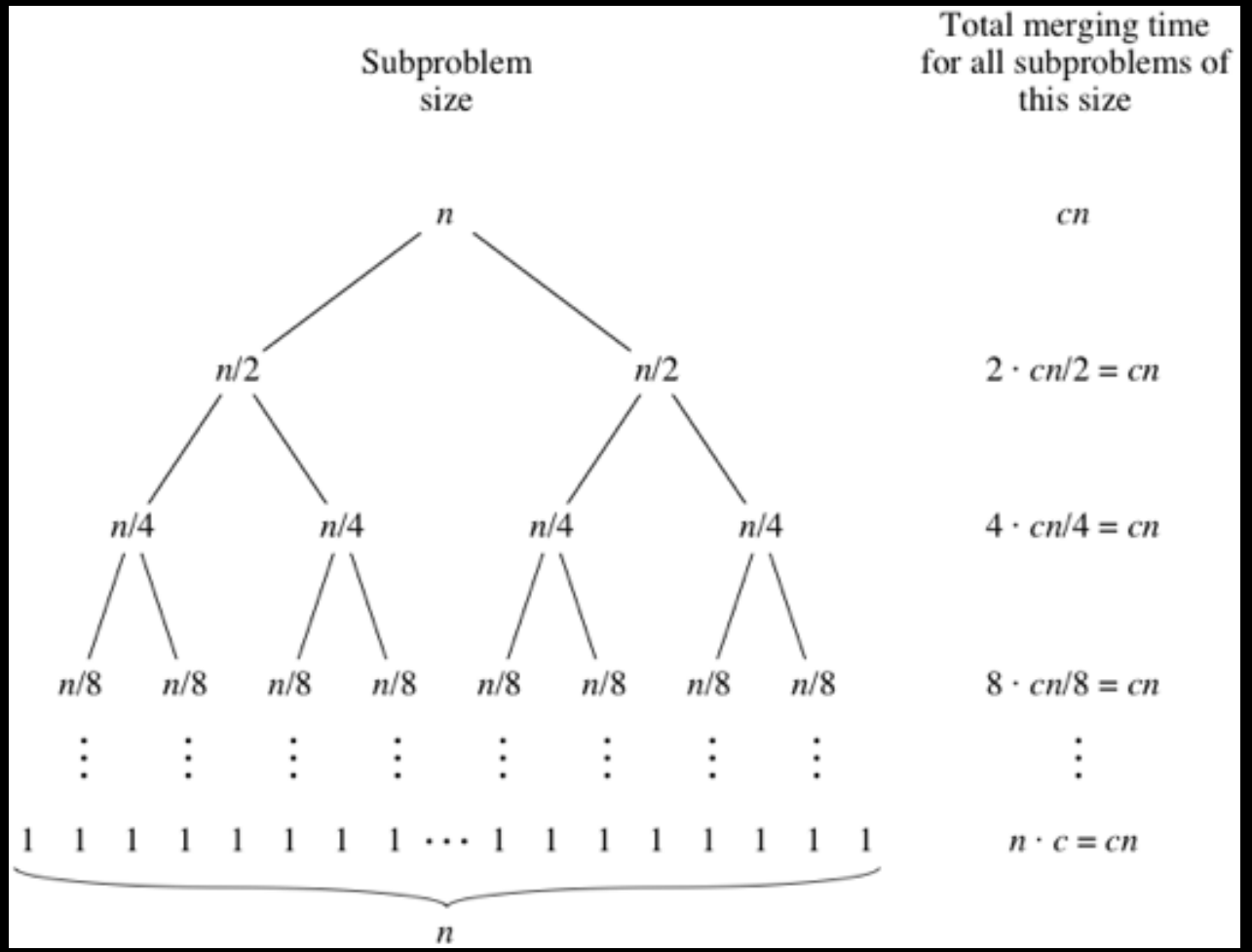
Merge sort is a Divide-and-Conquer Algorithm that works as follows:

1. **Divide**: Compute middle of the array, takes constant time O(1)
2. **Conquer:** Recursively solve two subproblems, each of size 2, which contributes 2T(n/2) to the running time
3. **Combine**: Merging the two subarrays, takes O(n)

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

RECURSION TREE FOR MERGE SORT

- At every level of recursion tree, cn amount of work needs to be done
- The height of the recursion tree is (lgn)
- So the work that needs to be done is O(nlgn) for sorting the whole array of size n



THE MERGE SORT ALGORITHM

- The pseudo-code for the Merge Sort algorithm and code in C is shown here
- The function divides an array into two subarrays
- Recursive call is then carried out on both subarrays [2T(n/2)]
- The merge method merges the sorted subarrays together O(n)

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

```c
void merge_sort(int *arr, int initial, int final, int *count)
{
    // condition for terminating
    if (initial >= final) return;

    int mid = (initial + final) / 2;
    merge_sort(arr, initial, mid, count);
    merge_sort(arr, mid + 1, final, count);
    merge(arr, initial, mid, final, count);

}
```

CHECKING FOR CORRECTNESS

- The function compares current element with next
- Returns false if any current element is greater than next
- Returns true otherwise

```c
bool is_array_sorted(int *arr, int length)
{
    for (int i = 0; i < length - 1; i++)
        if (arr[i] > arr[i + 1])
            return false;

    return true;
}
```

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

THE QUICK SORT ALGORITHM

- Quicksort is a Divide-and-Conquer algorithm that works as follows:

1. **Divide**: Partition (rearrange) the array into two (possibly empty) arrays
2. **Conquer:** Sort the two subarrays by recursive calls to quicksort
3. **Merge**: Because the subarrays are already sorted, no work is needed to combine them: the entire array is now sorted

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

```c
void quick_sort(int *arr, int initial, int final, int *count)
{
    if (initial < final) {
        int pos_of_pivot = partition(arr, initial, final, count);
        quick_sort(arr, initial, pos_of_pivot, count);
        quick_sort(arr, pos_of_pivot + 1, final, count);
    }
}
```

THE QUICK SORT ALGORITHM

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.

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

PARTITIONING

- The Hoare partitioning scheme is used here.
- The pivot is always set to the first element of the array in this scheme.

```
int partition(int *arr, int initial, int final, int *count)
{
    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]);
            (*count)++;
        }
        else
            return left_iterator;
    }
}
```

CHECKING FOR CORRECTNESS

- The function compares current element with next
- Returns false if any current element is greater than next
- Returns true otherwise

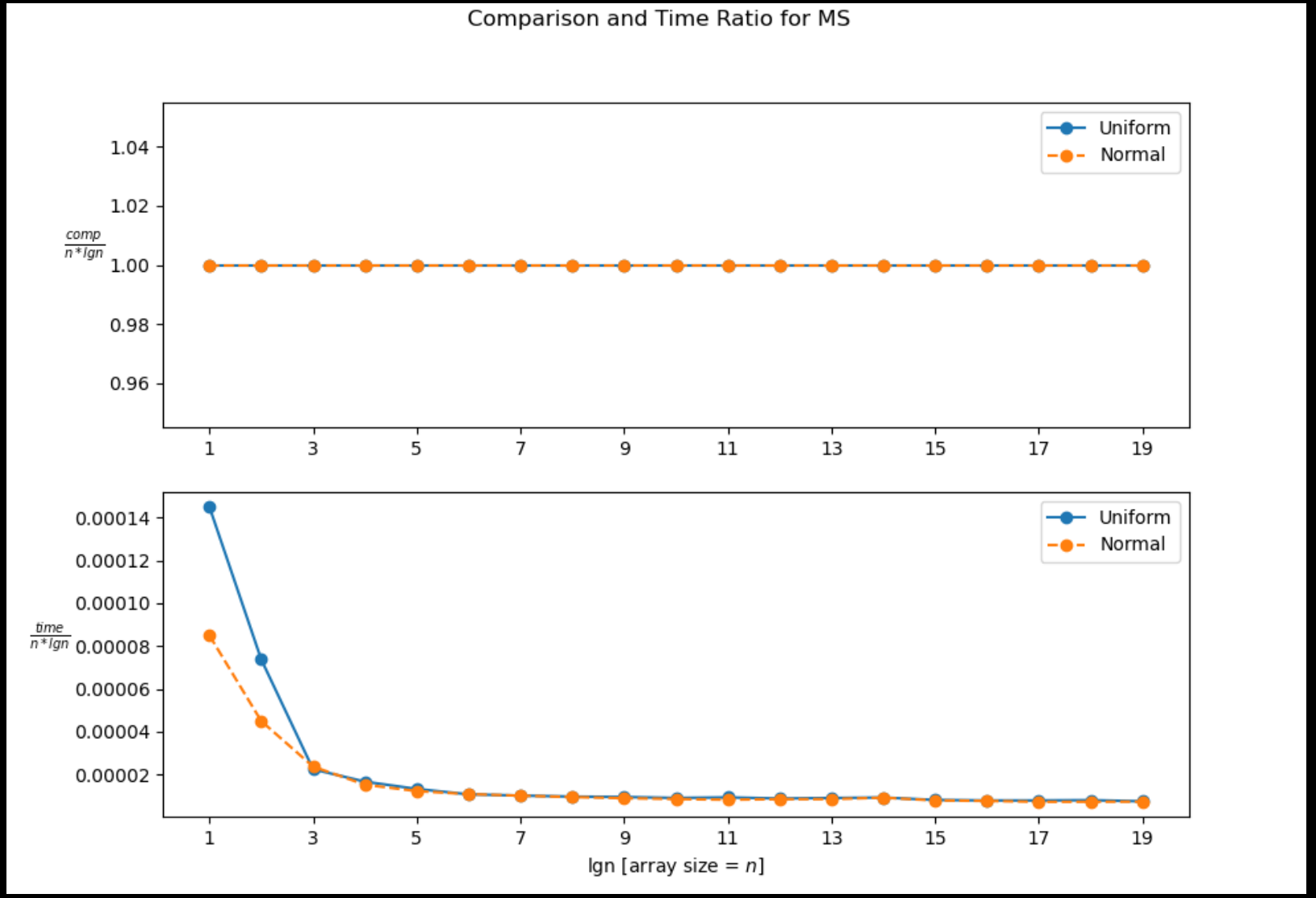
```
bool is_array_sorted(int *arr, int length)
{
    for (int i = 0; i < length - 1; i++)
        if (arr[i] > arr[i + 1])
            return false;

    return true;
}
```

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

COUNTING OPERATIONS

- We varied the array size (n) in powers of 2, recording array sizes from 2^1 to 2^{19}
- Operations like swaps, comparisons were counted. Also we counted writing to memory operation too as it is also costly
- We plotted $\frac{comparisons}{n \times \log_2 n}$ vs $\log_2 n$ and $\frac{time}{n \times \log_2 n}$ vs $\log_2 n$ for both MS and QS
- For QS, we permuted the array before sending it to QS so that we have 100 QS calls, each for a shuffled permutation of the array

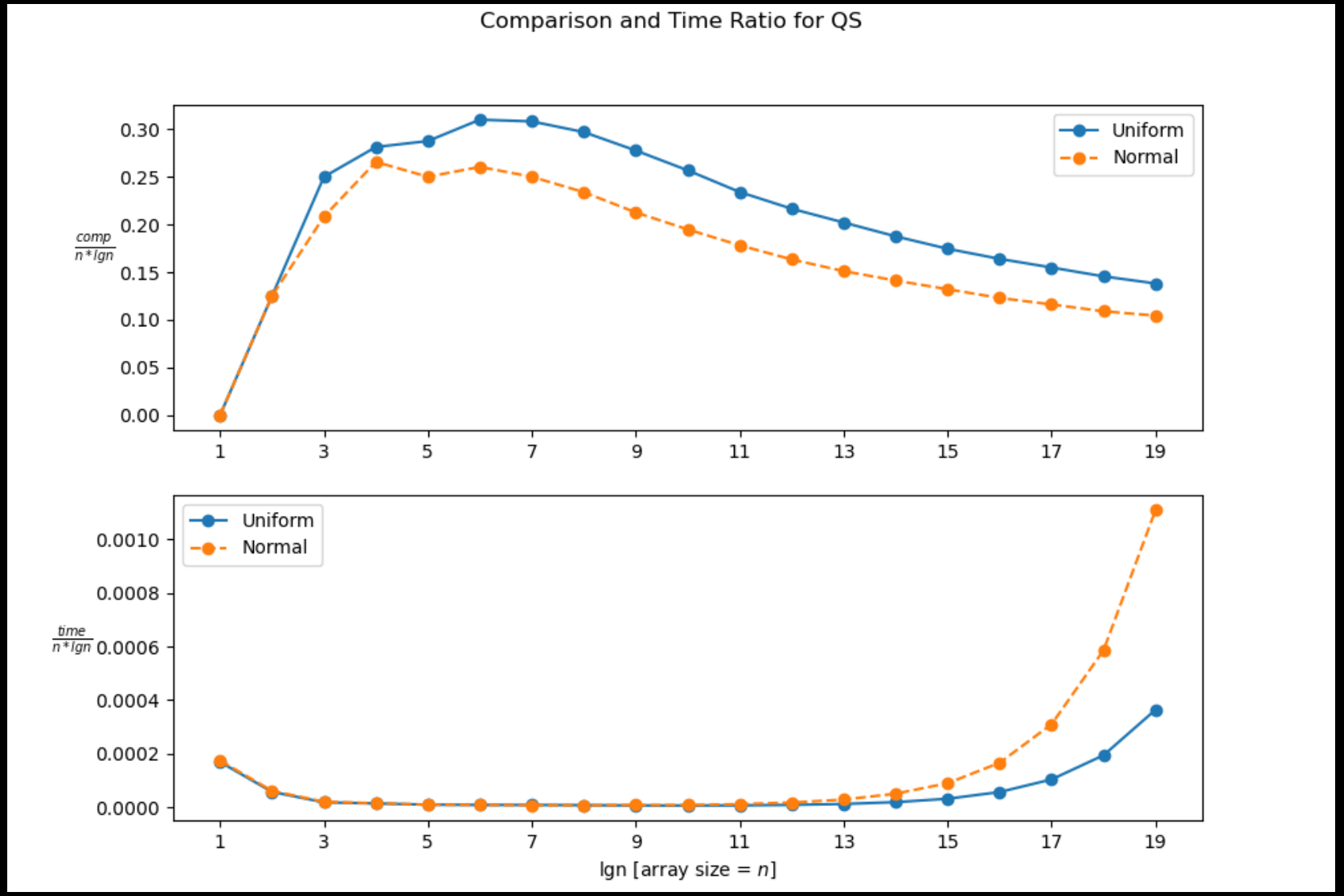
COUNTING OPERATIONS (MS)



OBSERVATIONS FOR MS

- $\frac{comparisons}{n \times \log_2 n}$ remains constant for all array sizes for both UD and ND
- $\frac{time}{n \times \log_2 n}$ also converges to a constant for arrays of appreciable sizes for both UD and ND data
- This shows the deterministic nature of Merge Sort where the number of operations is of the order O(nlgn)

COUNTING OPERATIONS (QS)



OBSERVATIONS FOR QS

- $\frac{comparisons}{n \times \log_2 n}$ ratio converges to a constant for higher array sizes
- $\frac{time}{n \times \log_2 n}$ ratio is constant but diverges for very large array sizes
- Operations done are lower for ND than UD. This might be because for the ND dataset the partition algorithm has a larger tendency to partition an element that is towards the middle in terms of value (frequency of median values is more in ND)

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.

RANDOMIZED QS ALGORITHM

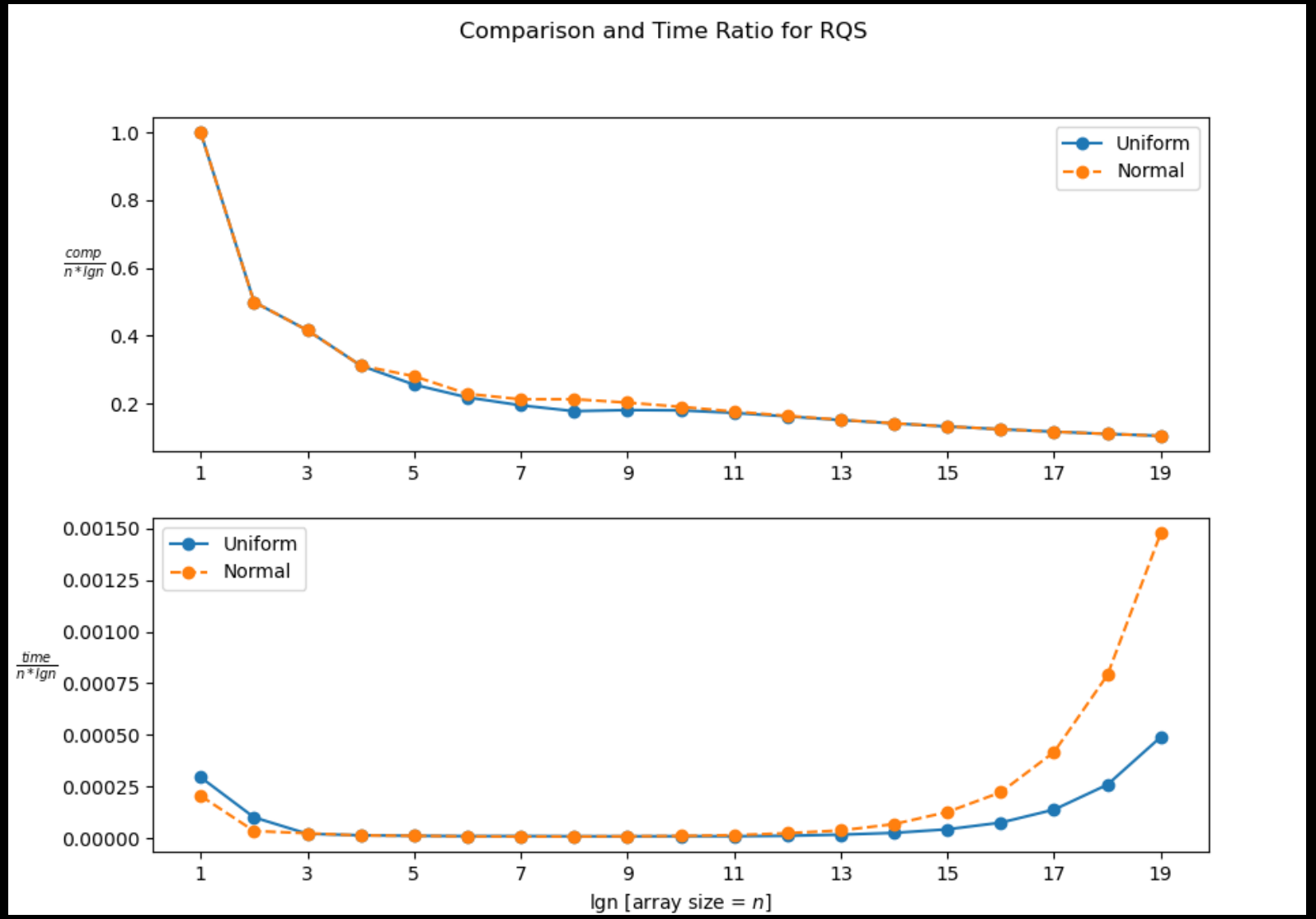
- Randomized Quicksort is a sorting algorithm that uses a randomized partitioning scheme to achieve an average-case time complexity of O(n log n) when sorting a sequence of n elements.

```
RANDOMIZED-SELECT(A, p, r, i)
1  if p == r
2       return A[p]
3  q = RANDOMIZED-PARTITION(A, p, r)
4  k = q − p + 1
5  if i == k            // the pivot value is the answer
6       return A[q]
7  elseif i < k
8       return RANDOMIZED-SELECT(A, p, q − 1, i)
9  else return RANDOMIZED-SELECT(A, q + 1, r, i − k)
```

COUNTING OPERATIONS

- We varied the array size (n) in powers of 2, recording array sizes from 2^1 to 2^{19}
- Operations like swaps, comparisons were counted. Also we counted writing to memory operation too as it is also costly
- We plotted $\frac{comparisons}{n \times \log_2 n}$ vs $\log_2 n$ and $\frac{time}{n \times \log_2 n}$ vs $\log_2 n$ for Randomized QS
- We also permuted the array before sending it to QS so that we have 100 RQS calls, each for a shuffled permutation of the array

COUNTING OPERATIONS (RQS)



OBSERVATIONS FOR QS

- $\frac{comparisons}{n \times \log_2 n}$ ratio converges to a faster than QS for both UD and ND
- $\frac{time}{n \times \log_2 n}$ ratio is constant but diverges for very large array sizes
- On average, the performance for Randomized QS is much better than QS implemented with Hoare partition
- This is because randomized partition tends to make the RQS O(n lgn) when averaged over a number of runs

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

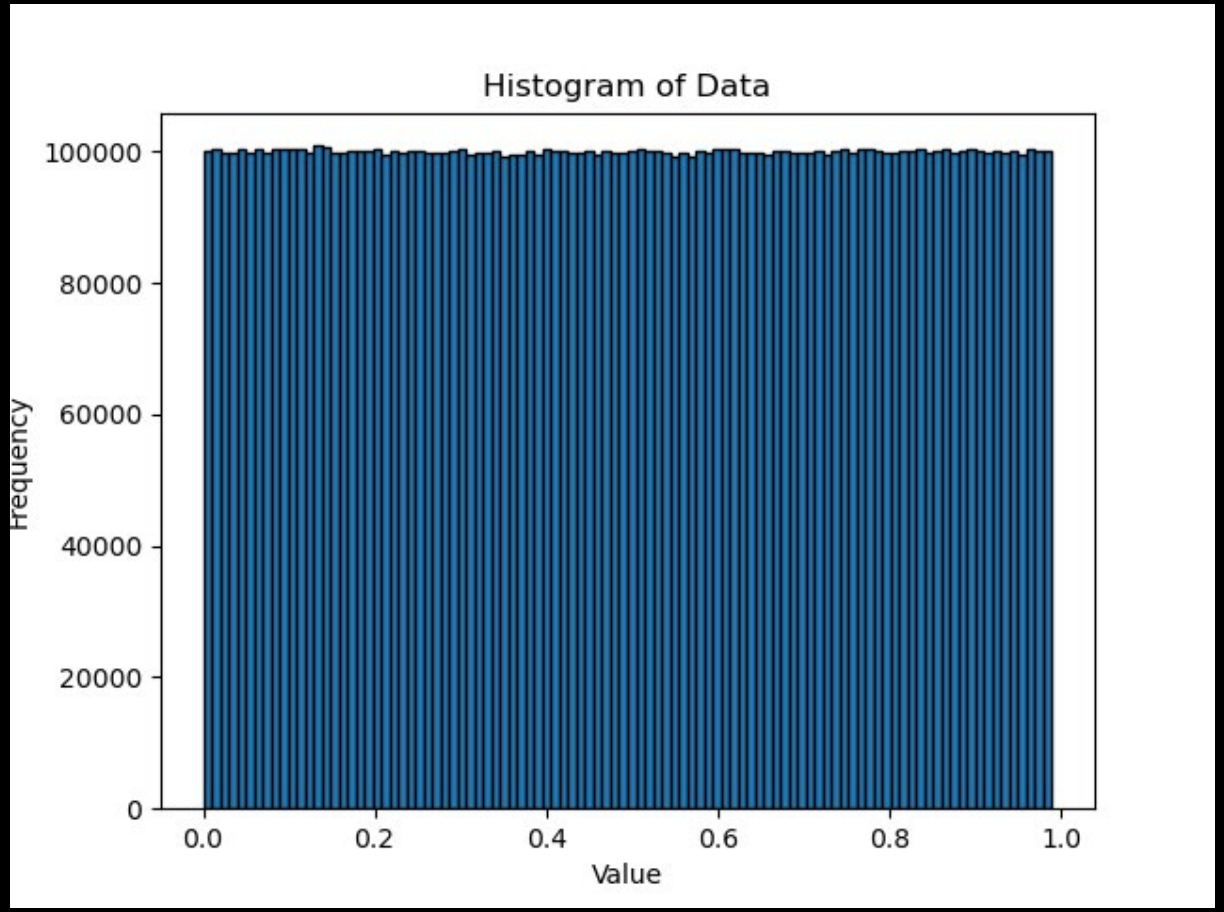
NORMALIZATION OF DATASETS

- Both the UD and ND datasets were normalized by dividing with the range of each value (range = 100-0 = 100)

```c
#include <stdio.h>

const int ELEMENTS = 10e6;
const int MAX_VALUE = 100;

int main()
{
    FILE *fin = fopen("./dataset/rand_uniform.csv", "r");
    // FILE *fin = fopen("./dataset/rand_normal.csv", "r");

    FILE *fout = fopen("./uniform_normalized.csv", "w");
    // FILE *fout = fopen("./normal_normalized.csv", "w");


    int temp;
    long long i = 0;
    while (!feof(fin))
    {
        i++;
        fscanf(fin, "%d\n", &temp);
        fprintf(fout, "%0.2f\n", ((float)temp) / MAX_VALUE);
        printf("\r%f %% done", ((float)i) / ELEMENTS * 100);
    }

    fclose(fin);
    fclose(fout);
    return 0;
}
```

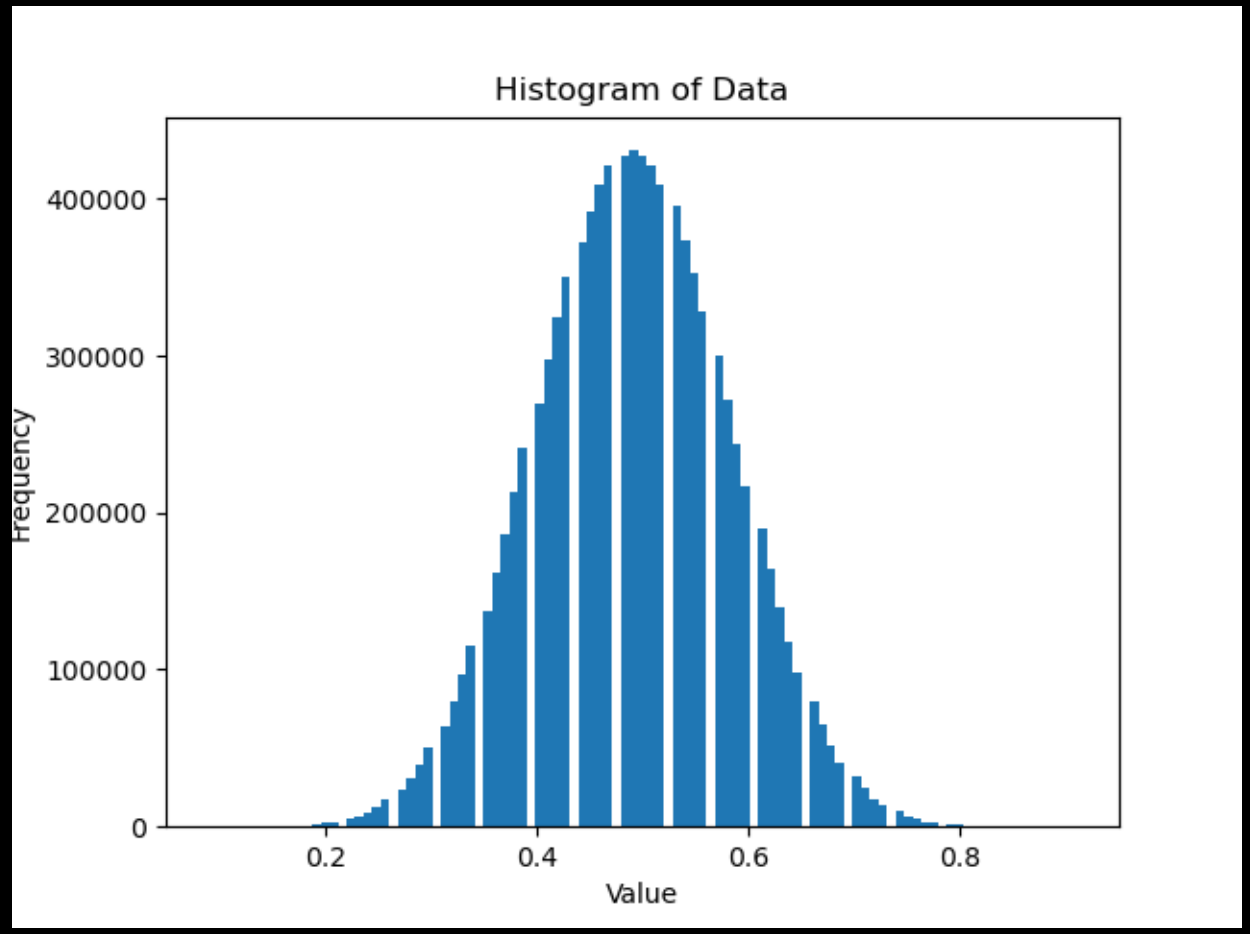
GRAPH OF UD DATASET (NORMALIZED)

```python
import pandas as pd
import matplotlib.pyplot as plt

# Load data from csv file into a pandas DataFrame
data = pd.read_csv('uniform_normalized.csv', header=None)

# Plot histogram of the data with 20 bins
plt.hist(data[0], bins=100, ec='black')

# Add x and y labels and a title to the plot
plt.xlabel('Value')
plt.ylabel('Frequency')
plt.title('Histogram of Data')

# Show the plot
plt.savefig('UD.png')
plt.show()
```



GRAPH OF UD DATASET (NORMALIZED)

```python
import pandas as pd
import matplotlib.pyplot as plt

# Load data from csv file into a pandas DataFrame
data = pd.read_csv('normal_normalized.csv', header=None)

# Plot histogram of the data with 20 bins
plt.hist(data[0], bins=101)

# Add x and y labels and a title to the plot
plt.xlabel('Value')
plt.ylabel('Frequency')
plt.title('Histogram of Data')

# Show the plot
plt.savefig('ND.png')
plt.show()
```



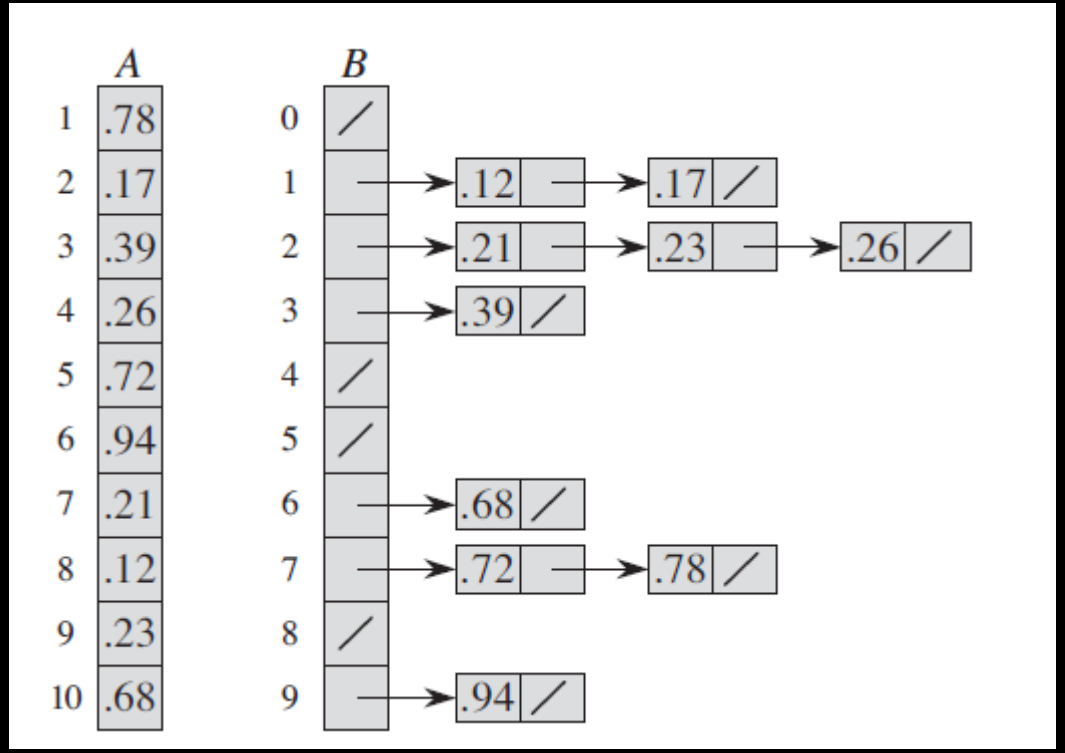
BUCKET SORT ALGORITHM

- Bucket sort, or bin sort is a sorting algorithm that works by partitioning an array into a number of buckets, each of which is then sorted individually using another sorting algorithm (like insertion sort)

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

BUCKET SORT ALGORITHM

- Bucket sort is most effective when the input values are uniformly distributed and the range of input values is not much larger than the number of elements to be sorted.
- Bucket sort's time complexity is O(n+k) where n is the number of elements to be sorted and k is the number of buckets used. If k is much larger than n, performance may decrease due to excessive overhead.



BUCKET SORT IMPLEMENTATION

- Insertion sort algorithm has been used to sort the LL
- The shown function inserts a node into its appropriate position in a linked list

```
node *LL_insert_sorted(node *list, float d, int *count) {
    node *new_node = (node *)malloc(sizeof(node));
    new_node->next = NULL;
    new_node->data = d;
    if (list == NULL) {
        list = new_node;
        (*count)++;
    }
    else if (new_node->data < list->data) {
        new_node->next = list;
        list = new_node;
        (*count)++;
    }
    else {
        node *temp = list;
        while ((temp->next != NULL) && (!((temp->data <= new_node->data)
                    && ((temp->next)->data >= new_node->data)))) {
            temp = temp->next;
            (*count) += 2;
        }

        new_node->next = temp->next;
        temp->next = new_node;
    }
    return list;
}
```

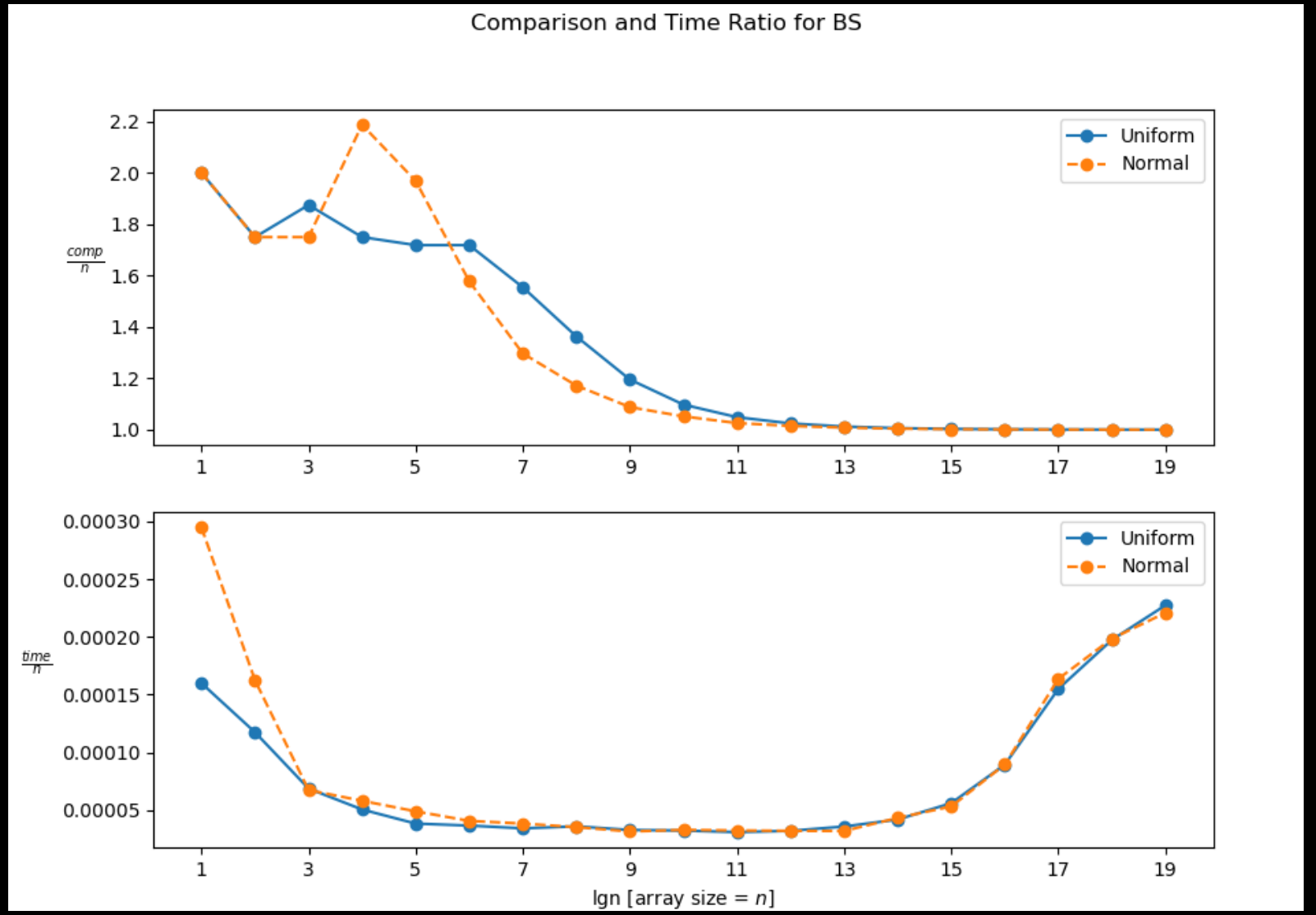
CHECKING FOR CORRECTNESS

- The function compares current element with next
- Returns false if any current element is greater than next
- Returns true otherwise

```
bool checkForCorrectness(float *arr, int length)
{
    for (int i = 0; i < length - 1; i++) {
        if (arr[i] > arr[i + 1]) {
            return false;
        }
    }

    return true;
}
```

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

COUNTING OPERATIONS (BS)



OBSERVATIONS FOR BS

- $\frac{comparisons}{n}$ ratio converges to a constant for both UD and ND datasets
- From this, we can infer that Bucket Sort is a linear time sorting algorithm, on an average

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-MEDIANS

- The Median of Medians is an approximate (median) selection algorithm, frequently used to supply a good pivot for an exact selection algorithm, mainly quicksort, that selects the k^{th} largest element of an initially unsorted array.
- When this approximate median is used as an improved pivot, the worst-case complexity of quicksort reduces significantly from O(n) to O(n lgn) , which is also the asymptotically optimal worst-case complexity of any sorting algorithm

IMPLEMENTATION IN C

give_median returns the median element of a given subarray, by first sorting it using insertion_sort

```c
void insertion_sort(int 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;
    }
}


int give_median(int arr[], int initial, int final)
{
    insertion_sort(arr, initial, final);
    int mid = (initial + final) / 2;
    return arr[mid];
}
```

IMPLEMENTATION IN C

median_of_medians recursively partitions an array into subarrays of size divide_size and calculates the median of each subarray using give_median. These medians are collected into a new array and median_of_medians is called on this new array until its size is less than divide_size. Finally, the median of the last subarray is returned.

```c
int median_of_median(int 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;

    int 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);
}
```

IMPLEMENTATION IN C

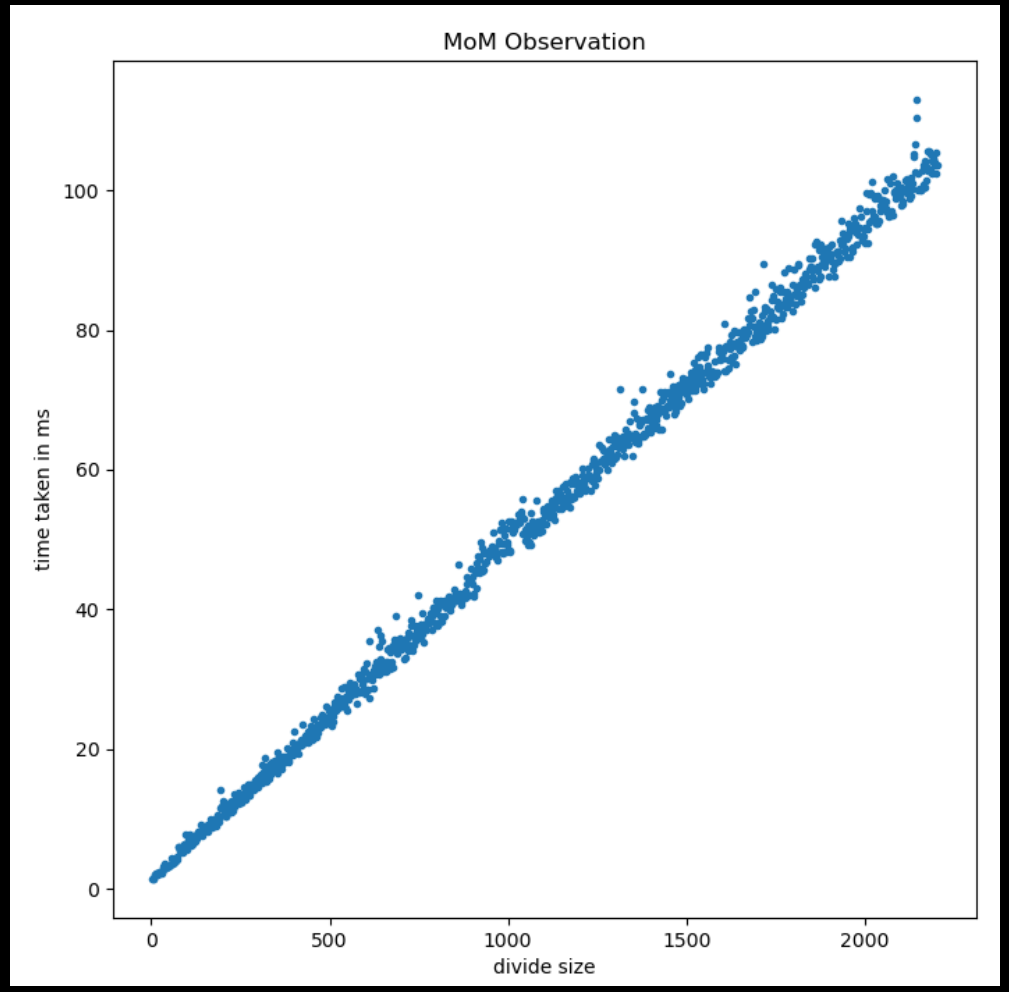
- In the main function, generate an array of random integers of length arr_size, and make a copy of it.
- Compute the direct calculation of the median of the original array using give_median.
- Permutation is applied to the array, and the results are written to a file for each iteration. The direct median of a randomly generated array is consistent across permutations. MOM analysis is only compared to the expected result.

```c
int main() {
    srand(time(0));
    FILE *fp;
    fp=fopen("MOM.txt","w");
    const int iter=10;
    for(int i=0;i<iter;i++) {
        fprintf(fp,"ITERATION NO:  %d",i+1);
        fprintf(fp,"-----------------------\n");
        int arr_size = 10;
        int arr[arr_size];
        for (int i = 0; i < arr_size; i++)
            arr[i] = rand() % 10;
        int copy[arr_size];
        for (int i = 0; i < arr_size; i++)
            copy[i] = arr[i];
        int direct=give_median(copy, 0, arr_size - 1);
        permutation(fp,arr, 0, arr_size-1,direct);
    }
    fclose(fp);
    return 0;
}
```

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

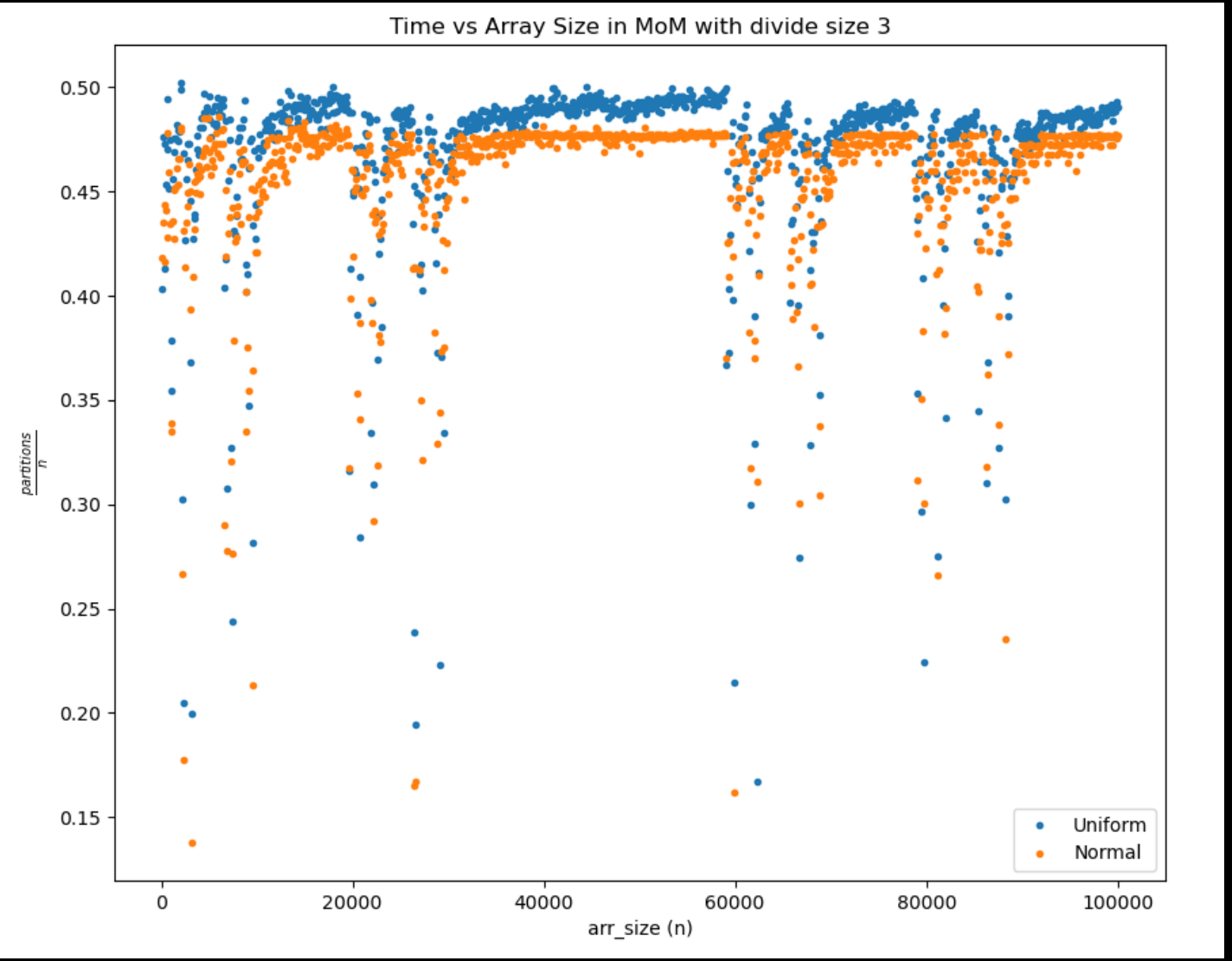
OBSERVATION OF TIME VS DIVIDE SIZE

- Median of medians was computed for fixes array size (10000) but varying the divide size every time (from 3, 5, 7, ..., 2203)
- Each divide size was run 10 times and avg_time was taken
- From the graph, we can observe that time taken varies linearly with divide size

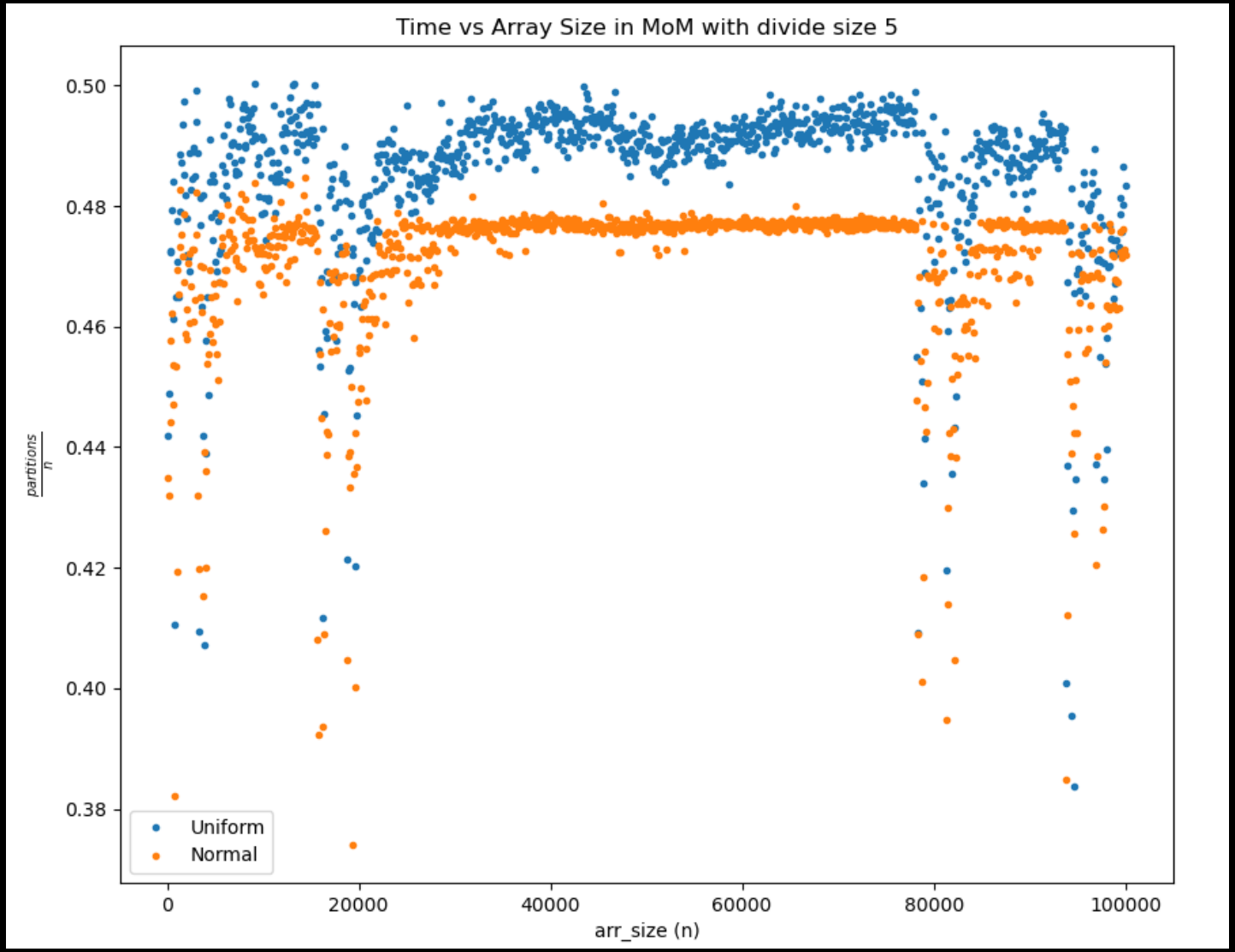


9. Perform experiments by rearranging the elements of the datasets (both UD and ND) and comment on the partition or split obtained using the pivotal element chosen as MoM

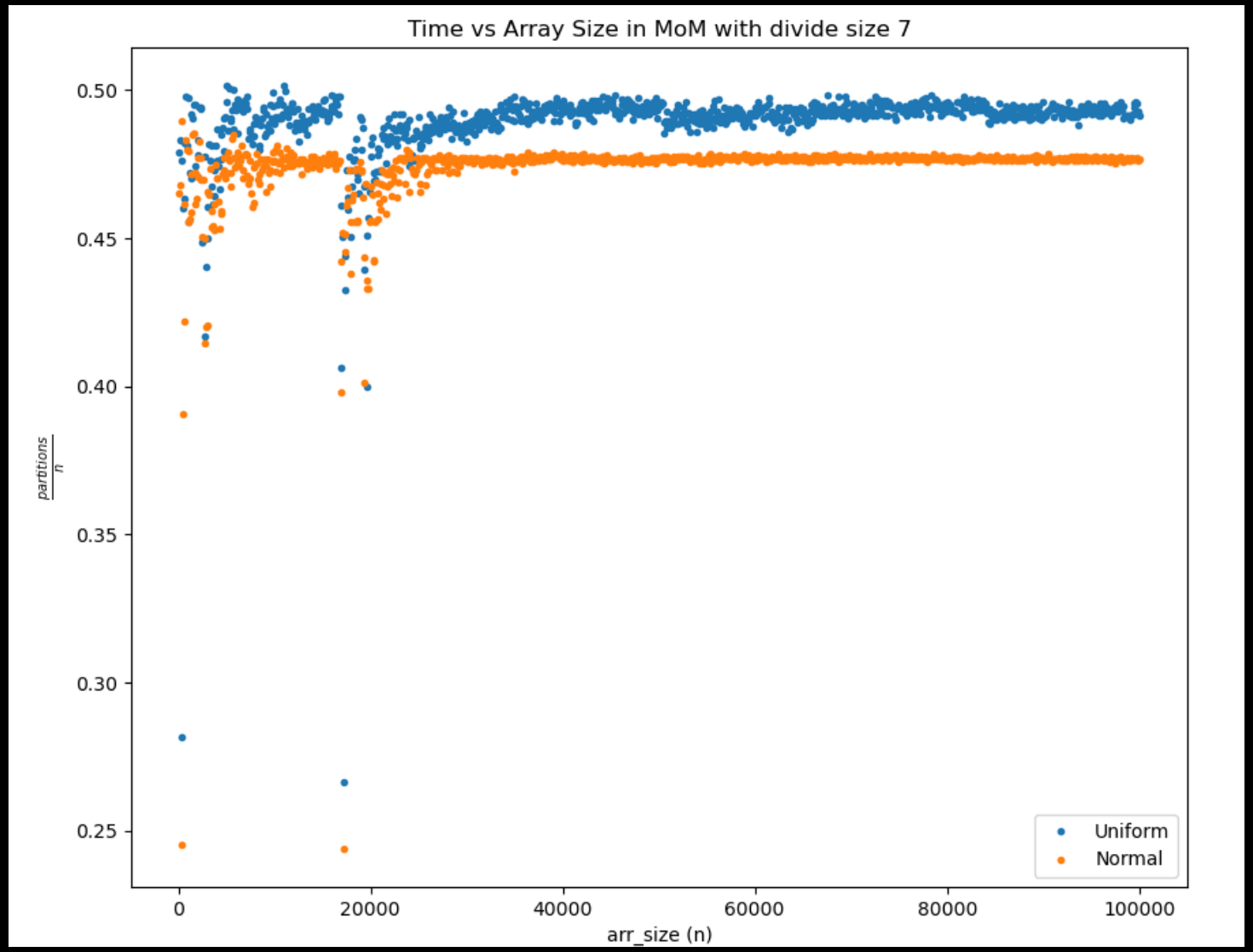
DIVIDE SIZE 3



DIVIDE SIZE 5



DIVIDE SIZE 7



INFERENCE

- $\frac{partition}{arr\_size}$ ratio is close to 0.5 for the various partition sizes shown (3/5/7) for both UD and ND dataset
- This shows that MoM finds a very good approximate median, thus proving to be a good pivot for the QS algorithm

THANK YOU