Report on Algorithm Lab Assignment 1

Ananya Sutradhar

2021CSB110

Algorithm Laboratory (CS2271)

Date: 19 February, 2023

Objective

In this assignment we tried to implement some algorithms and check correctness, distribution dependency, time complexity etcetera. The algorithms are Merge Sort, Quick Sort, Bucket Sort, and Median of Medians. All the codes used for the assignment can be found here.

The tasks assigned were as follows:

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

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

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

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

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.

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.

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

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

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

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

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.

Question no. 1

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

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

To construct large datasets taking random numbers rand() function available in the c library <stdlib.h> is used.

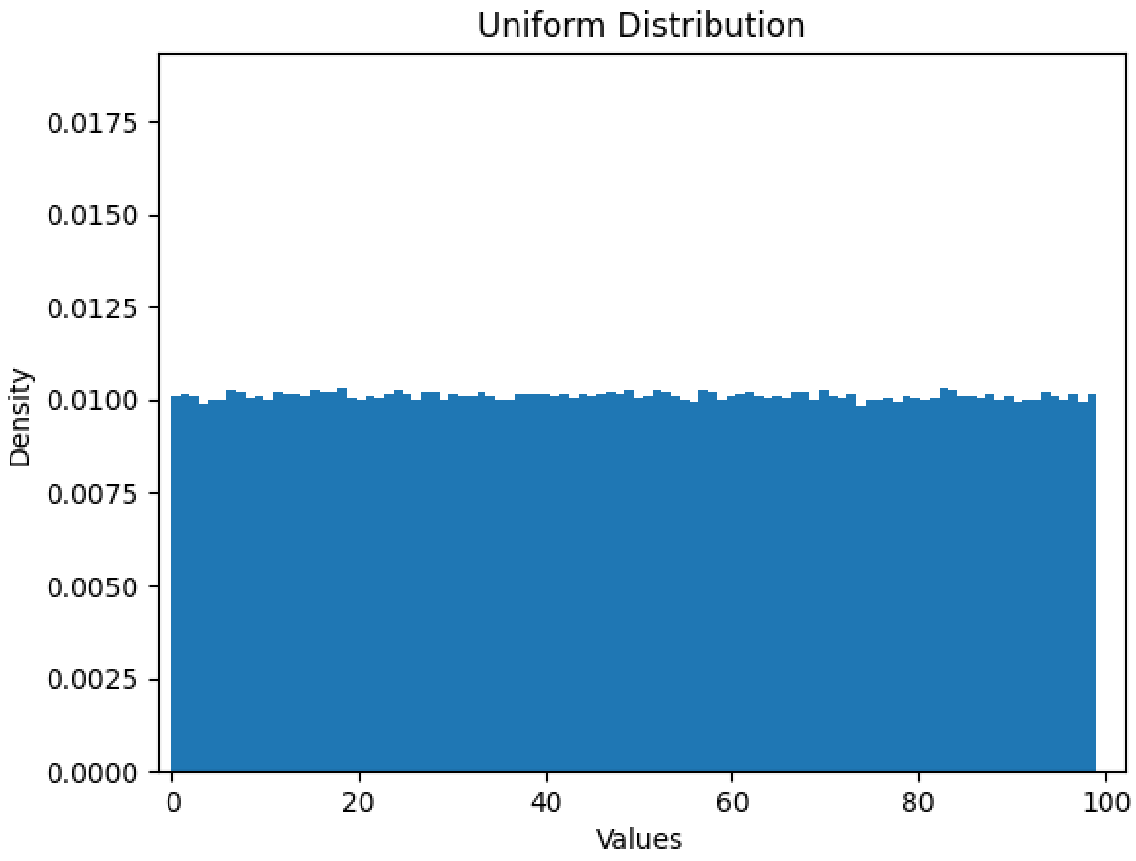
int rand(void) returns a pseudo-random number in the range of 0 to *RAND_MAX*.

1-A:

Uniform distributions are **probability distributions with equally likely outcomes**. In a discrete uniform distribution, outcomes are discrete and have the same probability. Random numbers generated by **int rand(void)** are distributed uniformly.

```
    int num;
            for(long long int i=0;i<1000000;i++){
                    num=rand()%100;
                    fprintf(file,"%d ",num);
            }
```

This piece of code helped to create a file containing 1000000 numbers ranging between 0 to 100 distributed uniformly.



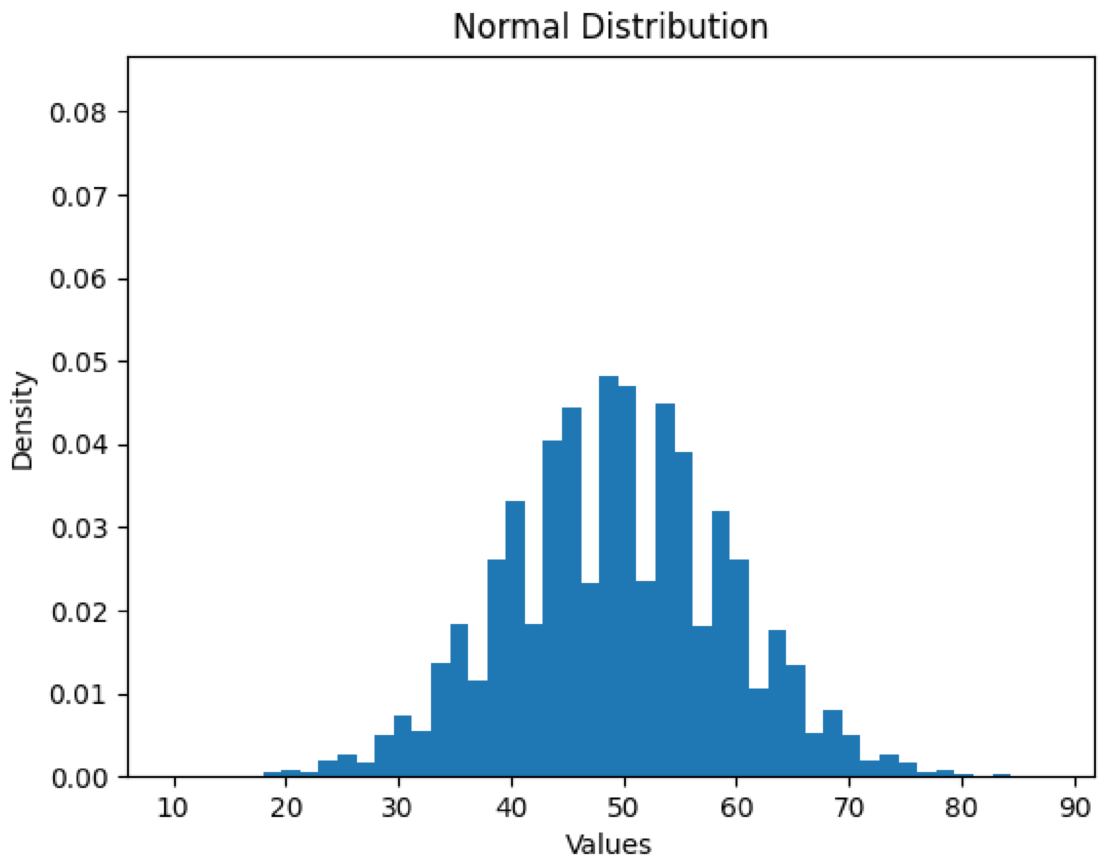
1-B:

Normal distribution, also known as the Gaussian distribution, is **a probability distribution that is symmetric about the mean, showing that data near the mean are more frequent in occurrence than data far from the mean**. In graphical form, the normal distribution appears as a "bell curve".

```
    double rand_gen() {
       // return a uniformly distributed random value
       return ((double)(rand())+1.)/((double)(RAND_MAX)+1.;
    }
    double normalRandom() {
       // return a normally distributed random value
       double v1=rand_gen();
       double v2=rand_gen();
       return cos(2*3.14*v2)*sqrt(-2.*log(v1));
    }
```

```
    double sigma = 10.0;
        double Mi = 50.0;
        for(int i=0;i<100000;i++) {
            double x = normalRandom()*sigma+Mi;
            cout << " x = " << x << endl;
            fprintf(file,"%d ",(int)x);
        }
```

This piece of code helped to create a file containing 1000000 numbers ranging between 0 to 100 distributed normally.



Question no. 2

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

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

2-A:

Merge sort is a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.

In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.

```
int  count=0; /* to count the number of comparisons */

 int merge( int arr [ ], int l, int m, int r)
{
 int i=l; /* left subarray*/
 int j=m+1; /* right  subarray*/
 int k=l; /* temporary array*/
 int temp[r+1];
 while( i<=m && j<=r)
 {
   if ( arr[i]<= arr[j])
  {
    temp[k]=arr[i];
    i++;
  }
 else
{
  temp[k]=arr[j];
  j++;
}
  k++;
  count++;

}
 while( i<=m)
{
  temp[k]=arr[i];
  i++;
  k++;
}
  while( j<=r)
{
  temp[k]=arr[j];
  j++;
  k++;
}
for( int p=l; p<=r; p++)
{
  arr[p]=temp[p];
}
 return count;
}
```

```
  int  mergesort( int arr[ ], int l, int r)
  {
    int comparisons;
    if(l<r)
  {
   int m= ( l+r)/2;
   mergesort(arr,l,m);
   mergesort(arr,m+1,r);
   comparisons = merge(arr,l,m,r);
  }
   return comparisons;
  }
```

2-B:

Like Merge Sort, QuickSort is a Divide and Conquer algorithm. It picks an element as a pivot and partitions the given array around the picked pivot.

The key process in quickSort is a partition(). The target of partitions is, given an array and an element x of an array as the pivot, put x at its correct position in a 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.

```
int comp=0;

//function for swapping two elements
void swap(int *i,int *j)
{
        int c = *i;
        *i = *j;
        *j = c;
}

//a function to make pivots
int partion(int array[], int left,int right)
{
        int ref,i,j;
    ref = array[right];// taking the last element as pivot
element.
    i = left-1;

    for(j=left; j<right; j++)//taking all the elements less
than pivot in one side before the elements greater than
pivot element.
    {
        if(array[j] <= ref)
                {
                        i++;
                        swap(&array[i],&array[j]);
                        s++;
        }
                comp++;
    }
        swap(&array[i+1],&array[right]);        // taking
the pivot element at its place or say between the
partition.
    s++;
        return (i+1);
```

```
}

//function to sort the elements.
void QuickSort(int array[],int lef,int rig)
{
        if(lef < rig)
        {
                int p = partion(array,lef,rig); //to find
the index of pivot element
                QuickSort(array,lef,p-1);
//recursively sort the second half (elements after the
pivot)
                QuickSort(array,p+1,rig);
//recursively sort the first half (elements before the
pivot)
        }
}
```

Question no. 3

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.

The time complexity of both the algorithms is as follows:

1. Merge sort(Average, Best, Worst) = O(n * log n)

2. Quick sort:

Best = O(n * log n)

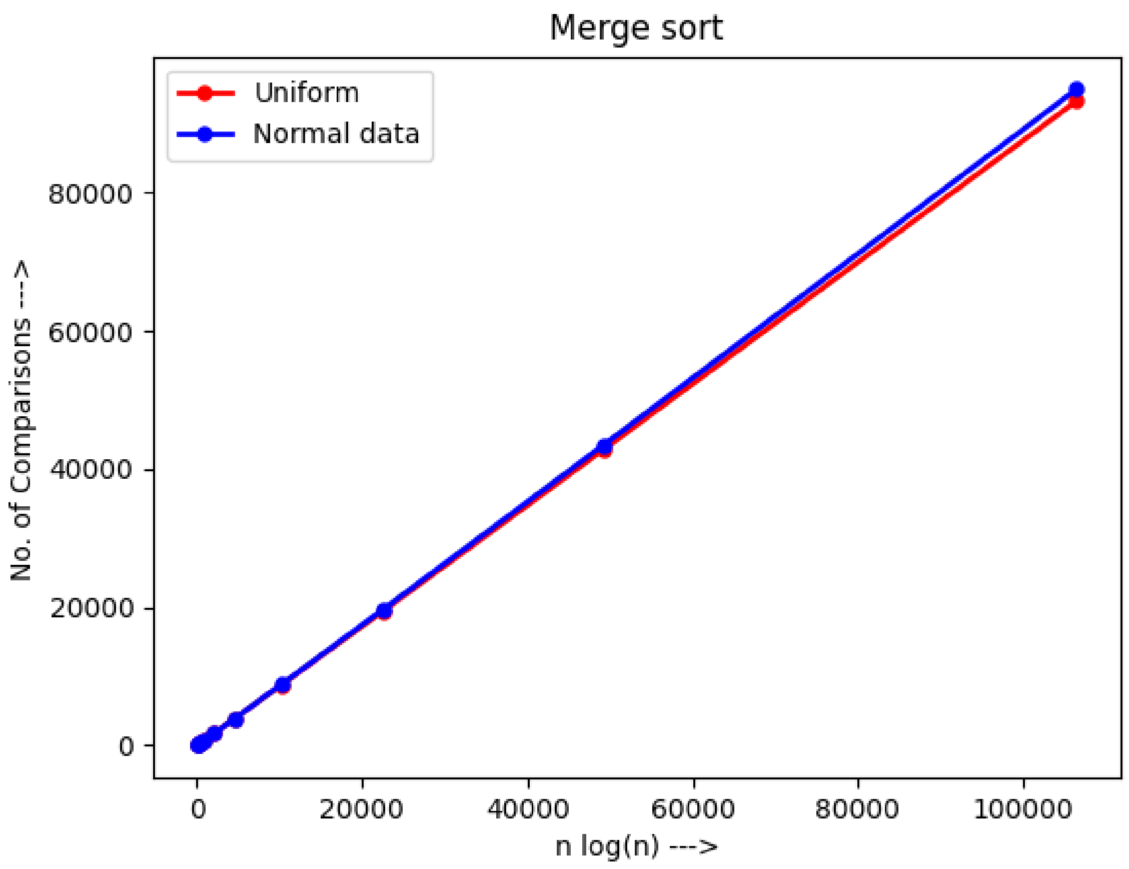
Average = O(n * log n)

Worst = O(n ^ 2), when the elements are already sorted in ascending or descending order.

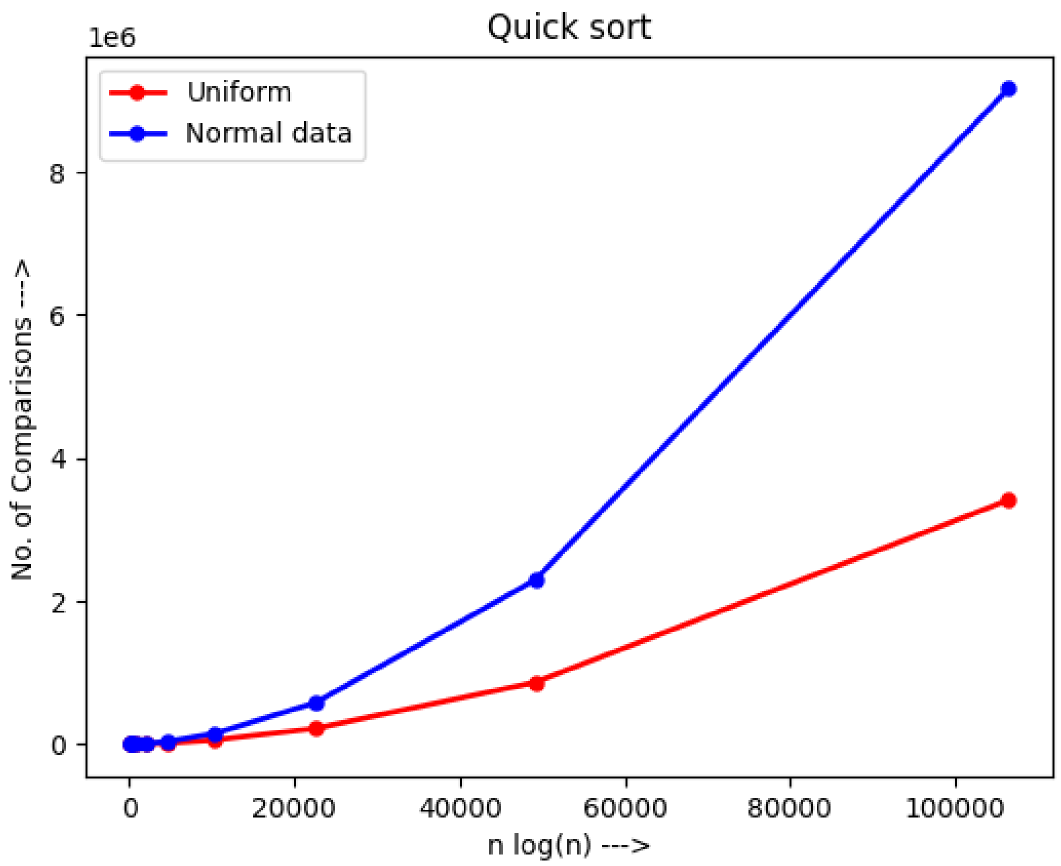
So, as we increase the number of elements in the array the running time will increase. We can also judge this by taking count of the comparisons needed for each case. Using different types of data distribution we can check for the distribution dependency of the algorithms.

We first take an algorithm and a distribution, then increase the number of elements in the array as the power of 2 (i.e. 4,8,16,32…), and take note of the number of comparisons. Then we compare the results. Here I have considered the problem size from 2^3 to 2^14.

Merge Sort on Uniform and Normal data:



Quick Sort on Uniform and Normal data:



We can easily comment that the time complexity we were expecting is actually true. Merge sort gives an almost straight line when comparisons vs n log(n) are plotted. Wheres for Quick sort we don't get a straight curve, so the complexity is slightly higher than n log(n) in some cases. Also, we notice that Merge sort gives similar results for both the data distributions but Quick sort performs better for Uniform data.

Question no. 4

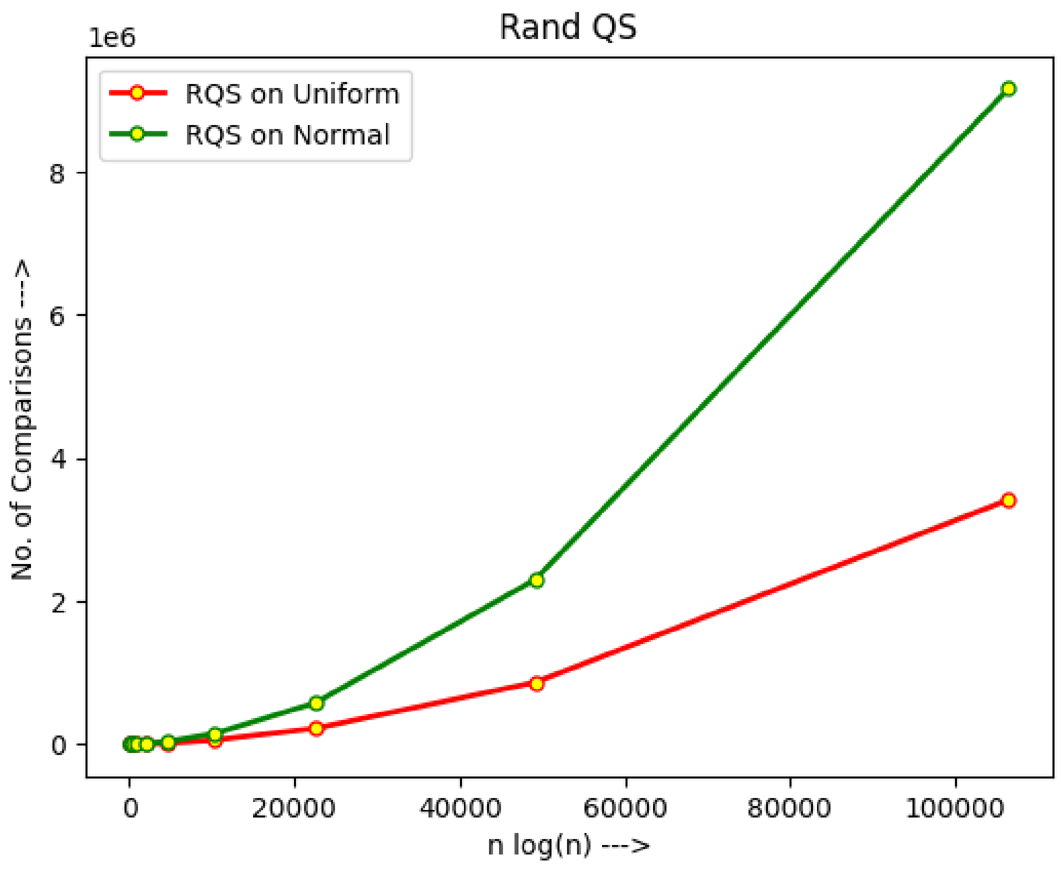
4. Experiment with randomized QS (RQS) with both UD and ND as input data to arrive at the average complexity (count of operations performed) with both input datasets.

In QuickSort we first partition the array in place such that all elements to the left of the pivot element are smaller, while all elements to the right of the pivot are greater than the pivot. Then we recursively call the same procedure for left and right subarrays. Thus Quicksort requires lesser auxiliary space than Merge Sort, which is why it is often preferred to Merge Sort. Using a randomly generated pivot we can further improve the time complexity of QuickSort.

```
int comp=0, s=0;


//function for swapping two elements

void swap(int *i,int *j)

{

        int c = *i;

        *i = *j;

        *j = c;

}


//a function to make pivots

int partition(int array[], int left,int right)

{

        int ref,i,j;
```

```
    ref = array[right];// taking the last element as pivot element.

    i = left-1;


    for(j=left; j<right; j++)//taking all the elements less than pivot in one side before the elements greater than pivot element.

    {

        if(array[j] <= ref)

                {

                        i++;

                        swap(&array[i],&array[j]);

                        s++;

        }

                comp++;

    }

        swap(&array[i+1],&array[right]);        // taking the pivot element at its place or say between the partition.

    s++;

        return (i+1);
```

```
}


int rand_partition(int arr[], int low, int high)
{
        srand(time(0));
        int random = low + (rand()+1) % (high - low);
        printf("\n rand = %d \n", random);
        int temp = arr[random];
        arr[random] = arr[low];
        arr[low] = temp;

        return partition(arr, low, high);
}


//function to sort the elements.
void QuickSort(int array[],int lef,int rig)
{
        if(lef < rig)
        {
                int p = rand_partition(array,lef,rig);
//to find the index of pivot element

                QuickSort(array,lef,p-1);
//recursively sort the second half (elements after the
pivot)

                QuickSort(array,p+1,rig);
//recursively sort the first half (elements before the
pivot)

        }

}
```



Question no. 5

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

Bucket Sort:

Bucket sort is a sorting algorithm that separate the elements into multiple groups said to be buckets. Elements in bucket sort are first uniformly divided into groups called buckets, and then they are sorted by any other sorting algorithm, like Insertion sort. After that, elements are gathered in a sorted manner by concatinating the buckets.

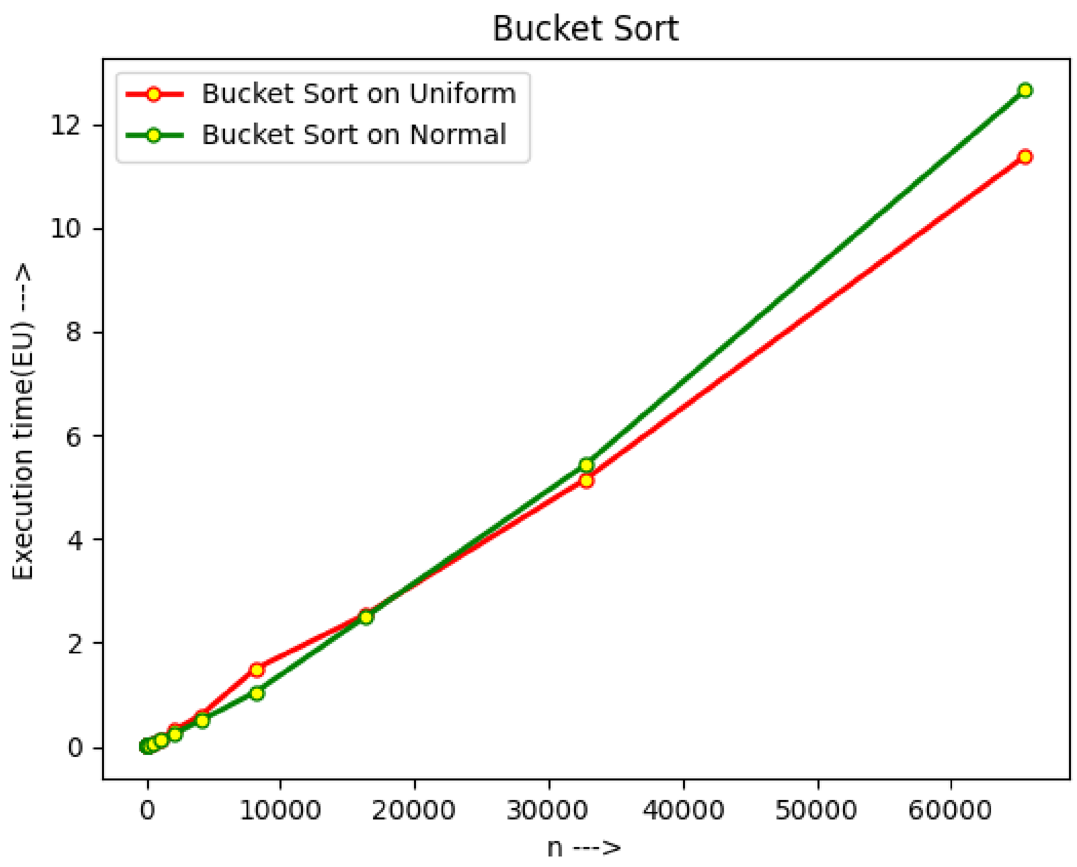
```
struct Node {
  float data;
  struct Node *next;
};

void BucketSort(float arr[],int n,int b,int u);
struct Node *InsertionSort(struct Node *list);
void print(float arr[], int n);
int printBuckets(struct Node *list);
int getBucketIndex(float value);

// Sorting function
void BucketSort(float arr[],int n,int b,int u) {
  int i, j;
  struct Node **buckets;

  // Create buckets and allocate memory size
  buckets = (struct Node **)malloc(sizeof(struct Node *) *
b);
```

```c
// Initialize empty buckets
for (i = 0; i < b; ++i) {
  buckets[i] = NULL;
}

// Fill the buckets with respective elements
for (i = 0; i < n; ++i) {
  struct Node *current;
  float pos = arr[i]*10;
  current = (struct Node *)malloc(sizeof(struct Node));
  current->data = arr[i];
  current->next = buckets[(int)pos];
  buckets[(int)pos] = current;
}

// Print the buckets along with their elements
for (i = 0; i < b; i++) {
  printf("Bucket[%d]: ", i);
  float d=printBuckets(buckets[i]);
  printf("\n");
}

// Sort the elements of each bucket
for (i = 0; i < b; ++i) {
  buckets[i] = InsertionSort(buckets[i]);
}

printf("-------------\n");
```

```c
  printf("Buckets after sorting\n");
  for (i = 0; i < b; i++) {
    printf("Bucket[%d]: ", i);
    printBuckets(buckets[i]);
    printf("\n");
  }

  // Put sorted elements on arr
  for (j = 0, i = 0; i < b; ++i) {
    struct Node *node;
    node = buckets[i];
    while (node) {
      arr[j++] = node->data;
      node = node->next;
    }
  }
    return ;
}

// Function to sort the elements of each bucket
struct Node *InsertionSort(struct Node *list) {
  struct Node *k, *nodeList;
  if (list == 0 || list->next == 0) {
    return list;
  }

  nodeList = list;
  k = list->next;
  nodeList->next = 0;
  while (k != 0) {
    struct Node *ptr;
    if (nodeList->data > k->data) {
      struct Node *tmp;
      tmp = k;
      k = k->next;
      tmp->next = nodeList;
      nodeList = tmp;
      continue;
    }

    for (ptr = nodeList; ptr->next != 0; ptr = ptr->next) {
      if (ptr->next->data > k->data)
        break;
    }

    if (ptr->next != 0) {
      struct Node *tmp;
      tmp = k;
      k = k->next;
      tmp->next = ptr->next;
      ptr->next = tmp;
      continue;
    } else {
      ptr->next = k;
      k = k->next;
      ptr->next->next = 0;
      continue;
    }
    }
    return nodeList;
}

int getBucketIndex(float value) {
     return value*10;
}
```

Question no. 6

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



From the graph we can say that average time complexity of Bucket Sort is **O(n)**. It is sightly more for Normally distributed data, as more and more element end up in the same bucket and the cost for Insertion sort increases. So, it is better to say that the average time complexity is **O(n+k),** where k is the number of buckets.

Question no. 7

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

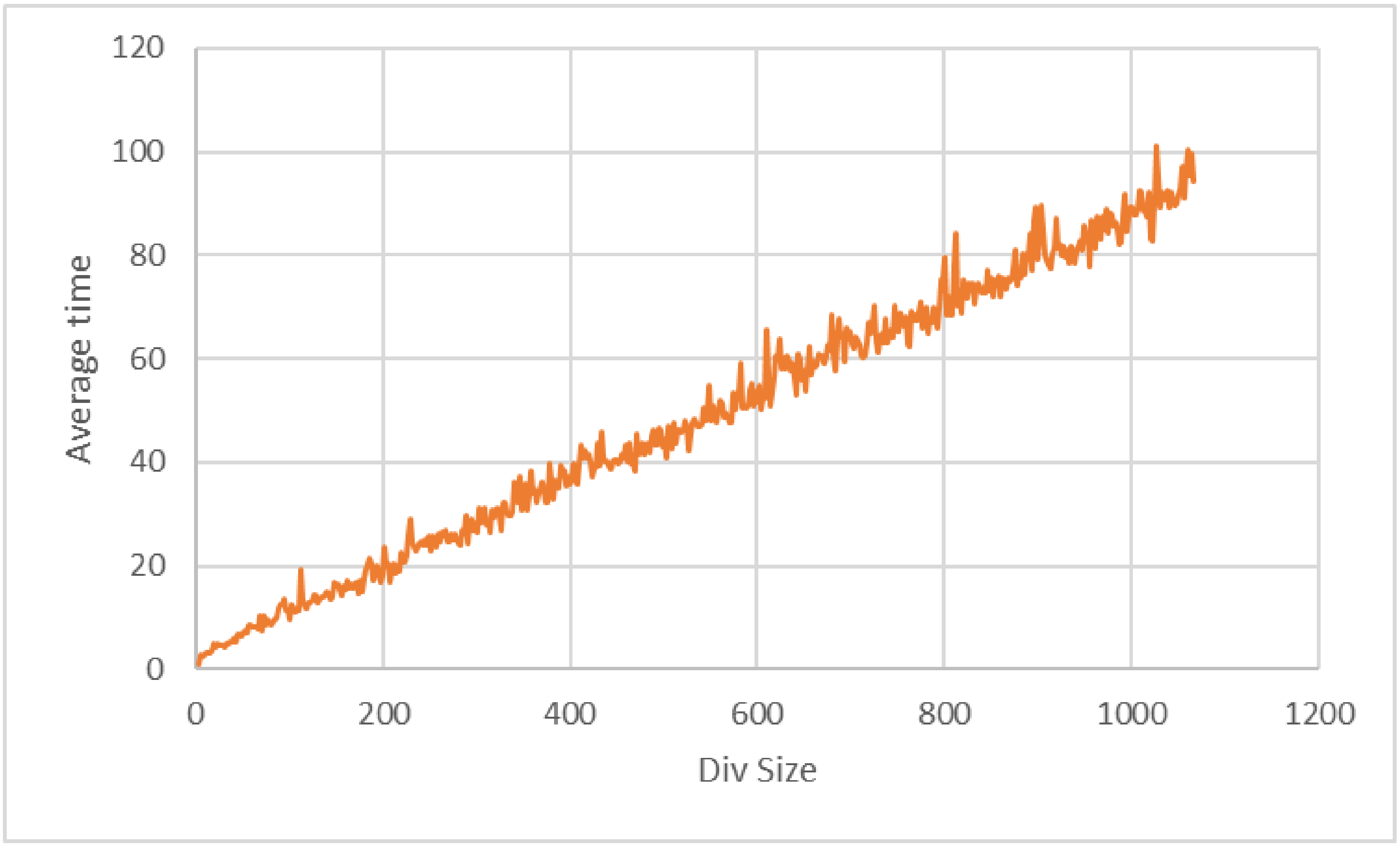
Median of medians:

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 kth 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 quickselect reduces from quadratic to linear, which is also the asymptotically optimal worst-case complexity of any selection algorithm.

```
int median(int *arr, int start, int final)
{
  insertion_sort(arr, start, final);
  int mid = (start + final) / 2;
  return arr[mid];
}

int median_of_medians(int *arr, int size, int div_size)
{
  if (size < div_size)
  {
    return median(arr, 0, size - 1);
  }
  int total = size / div_size;
  int last = size % div_size;

  int next;

  if (last == 0)
  {
    next = total;
  }
  else
  {
    next = total + 1;
  }

  int next_arr[next];

  for (int i = 0; i < next; i++)
  {
    if (i == next - 1)
    {
      next_arr[i]=median(arr,div_size* , size - 1);
    }
    else
    {
      next_arr[i] = median(arr, div_size * i, div_size * (i
+ 1) - 1);
    }
  }
  return median_of_medians(next_arr, next, div_size);
}
```

Question no. 8

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



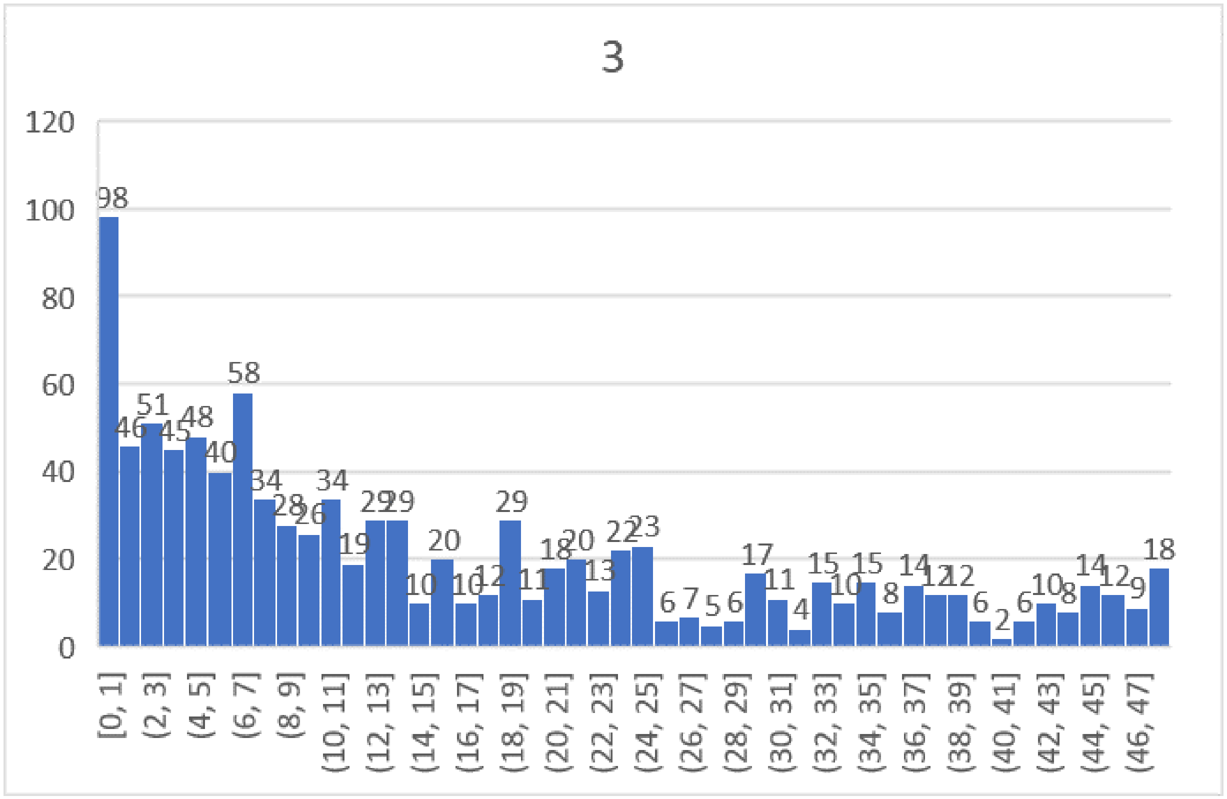
As we increase the divide(group) size the execution time increases proportionally.

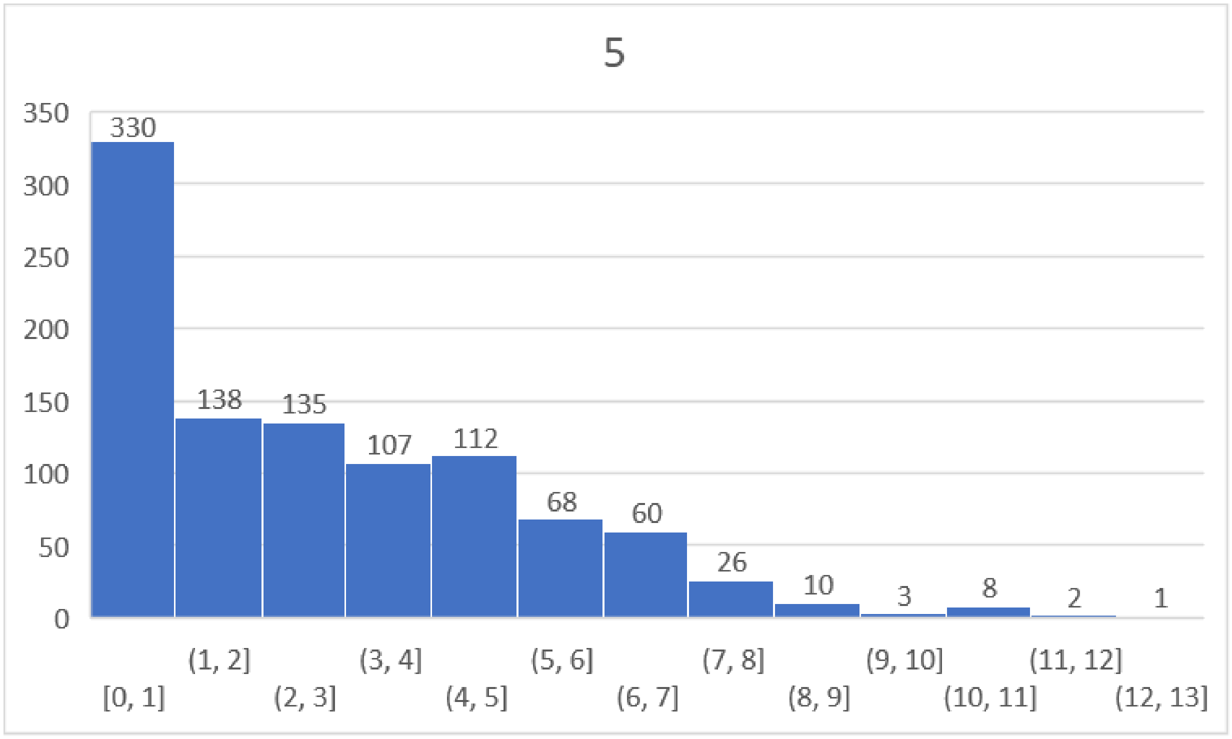
Also the difference between actual median and MoM follows a pattern for different devide size. The histograms bellow shows the frequency of variation between median and MoM.

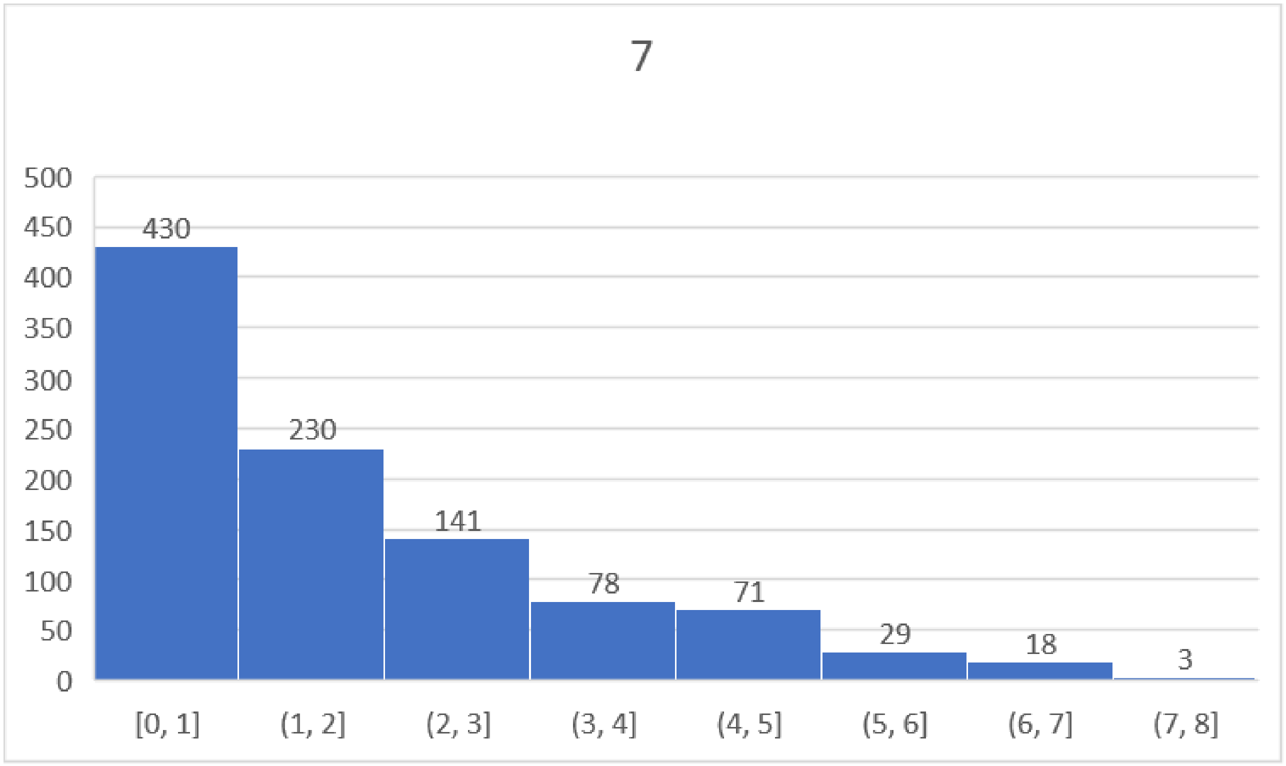
Now, changing the size of group :

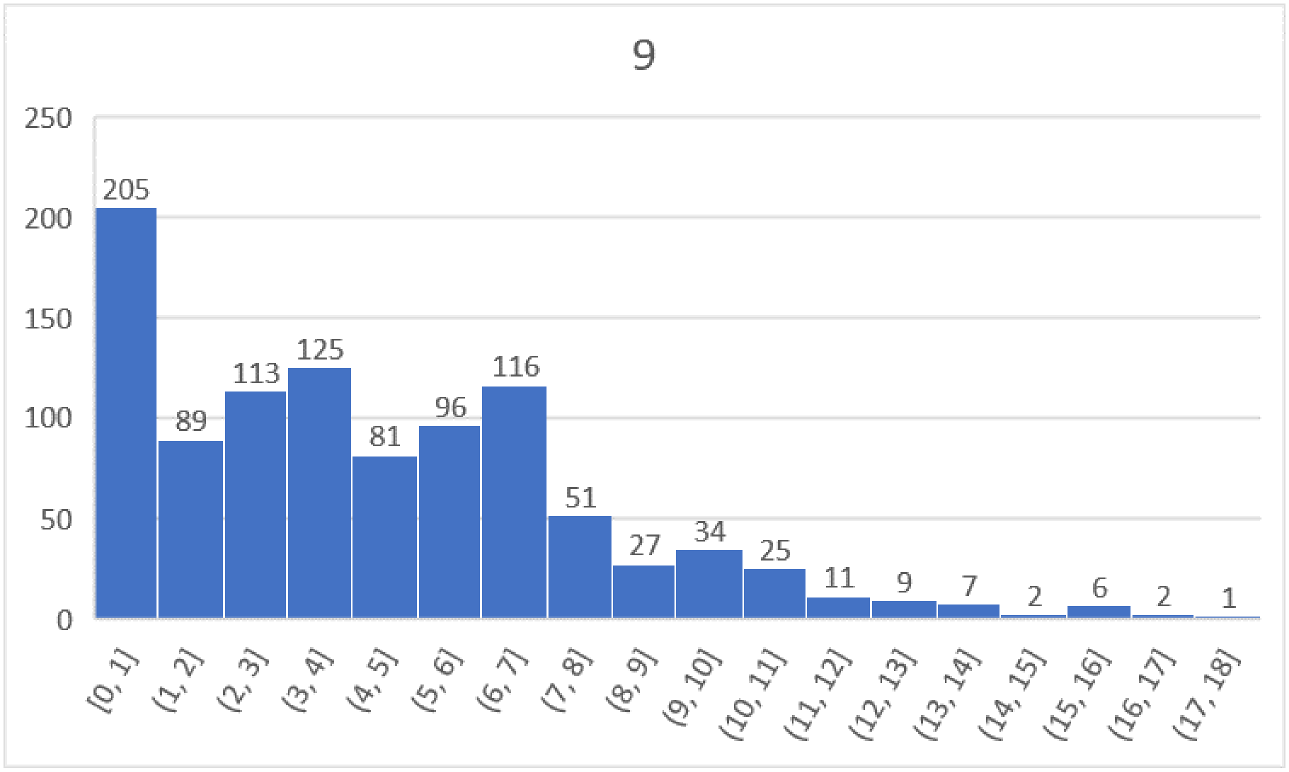
On X axis: Difference between actual median and MoM

On Y axis: Frequncy of the difference


3


5

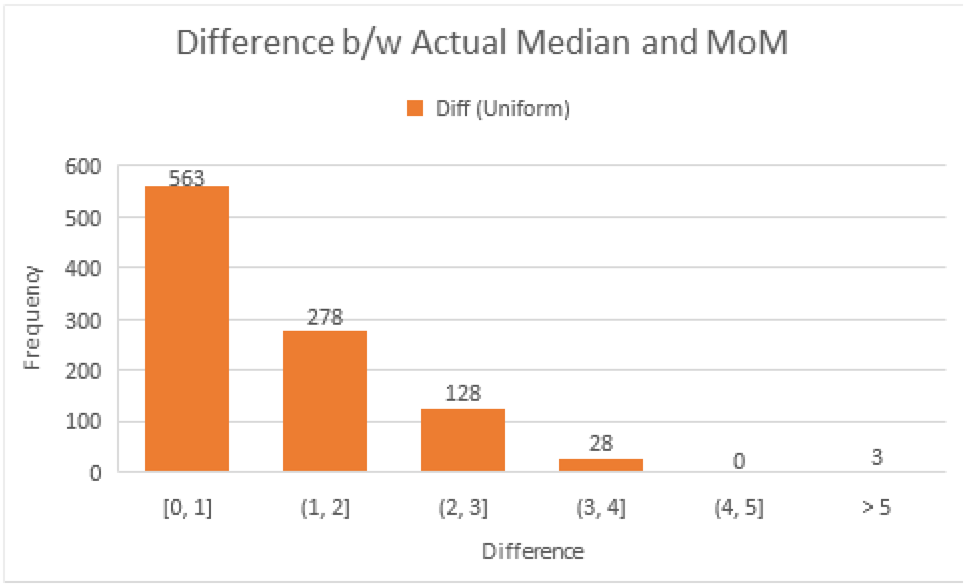


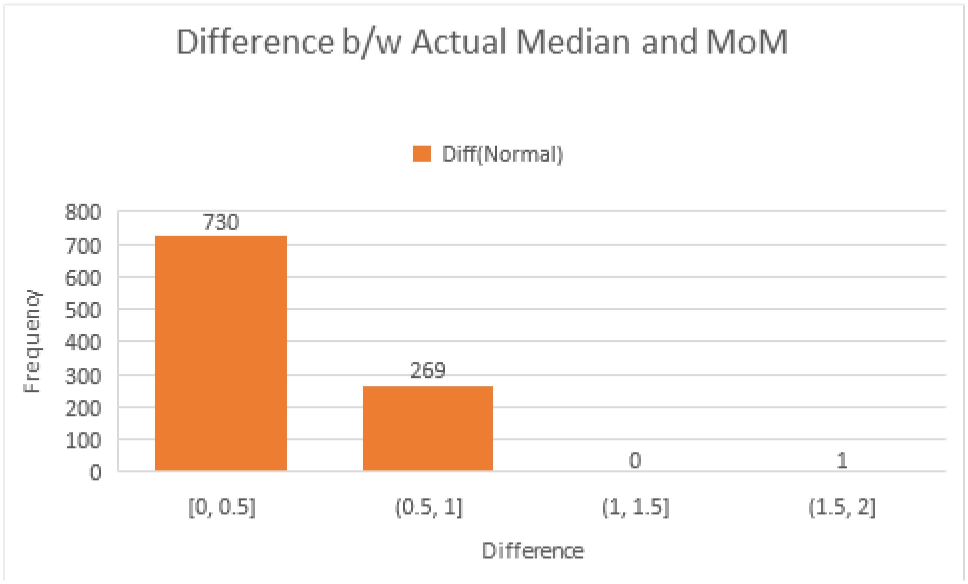


We see, group size of 5 or 7 gives us better results than group size of 3 or 9.

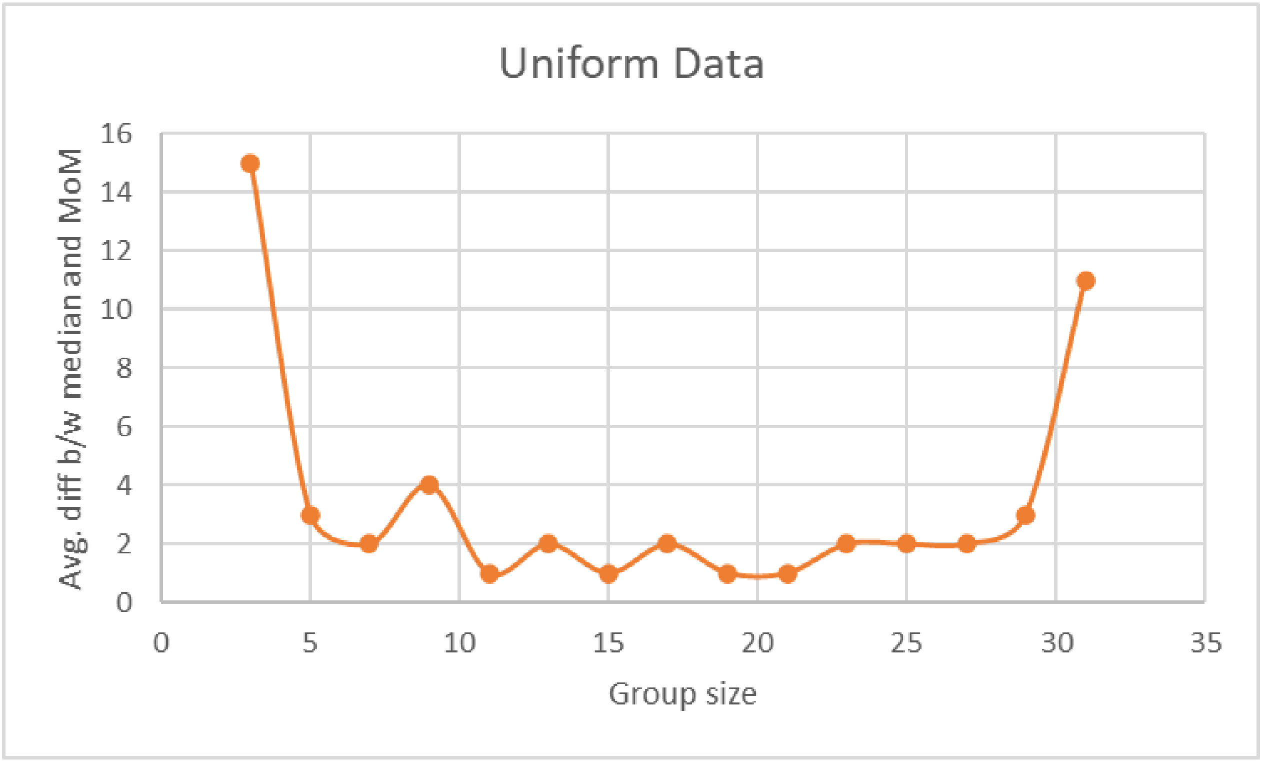
Question no. 9

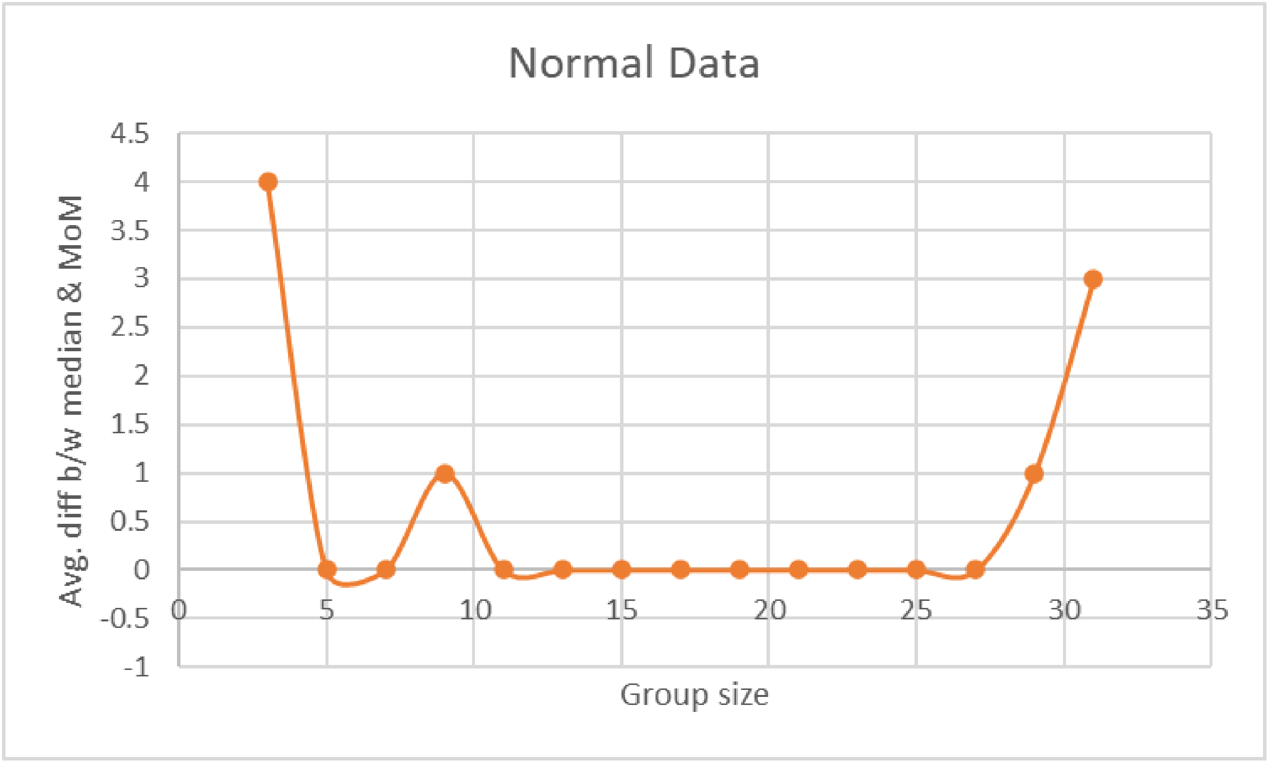
9. Perform experiments by rearranging the elements of the datasets (both UD and ND) and comment on the partition or split obtained using the pivotal element chosen as MoM.

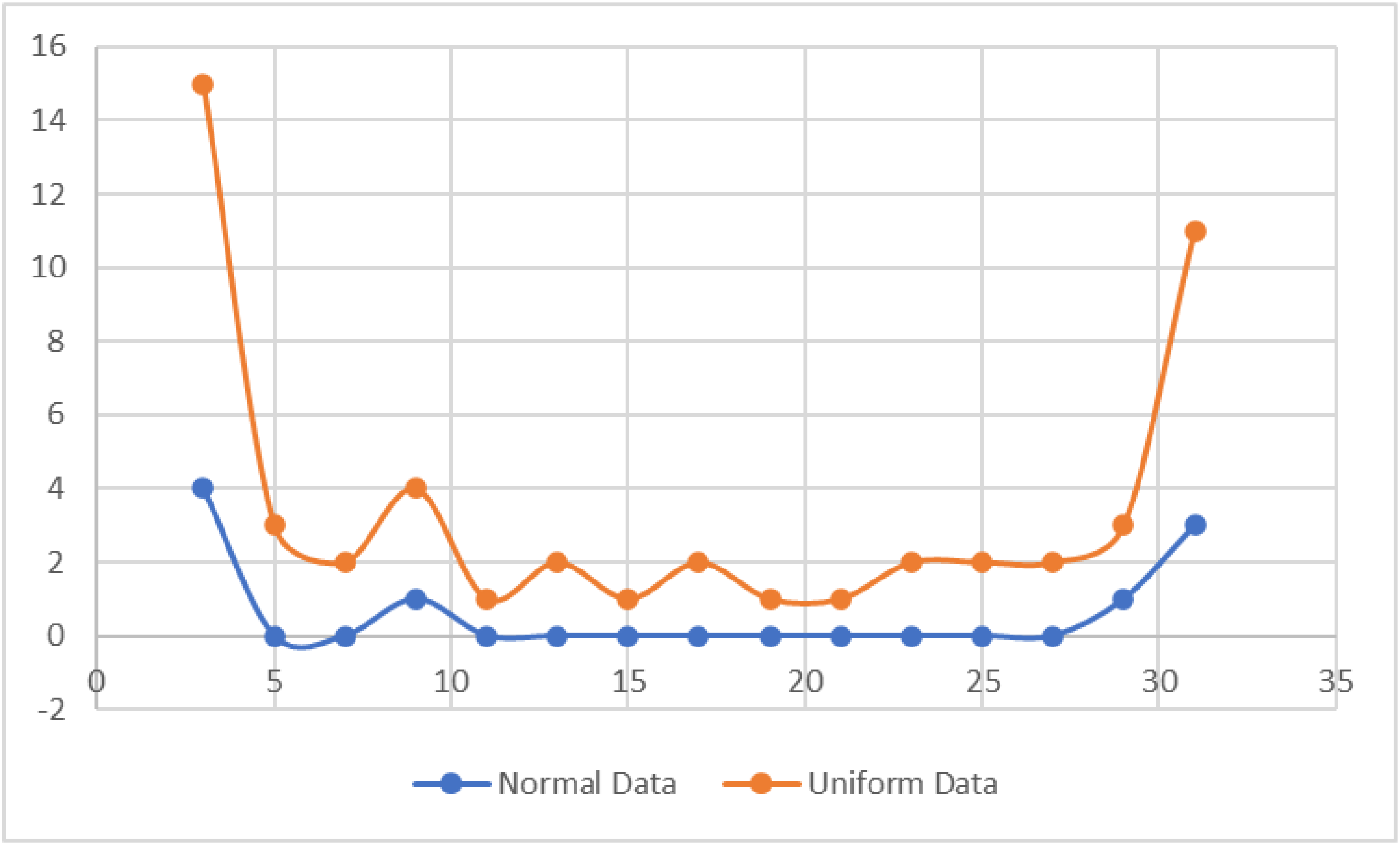




For a particular ordering of the elements, we get a deterministic MoM. Now if we rearrange (or permute) these elements, we get a different MoM. Now considering 1000 different permutations of the same dataset and taking note of the corresponding MoM. This gives you an idea about the range of variation of the MoM. Aanalysis from the graph suggests that this is within 30% when we take groups of five.







These graphs also show that variation is less for Normally distributed data than Uniformly distributed data.