Algorithm Lab (CS-2271)

Assignment-1

https://github.com/arnabsen1729/AlgoLab-1

Department Of Computer Science and Technology

indian Institute of Engineering Science and Technology, Shibpur

Group 10 Members

Meet our team

RAJDEEP GHOSH

510519002

KRISHNENDU BERA

510519004

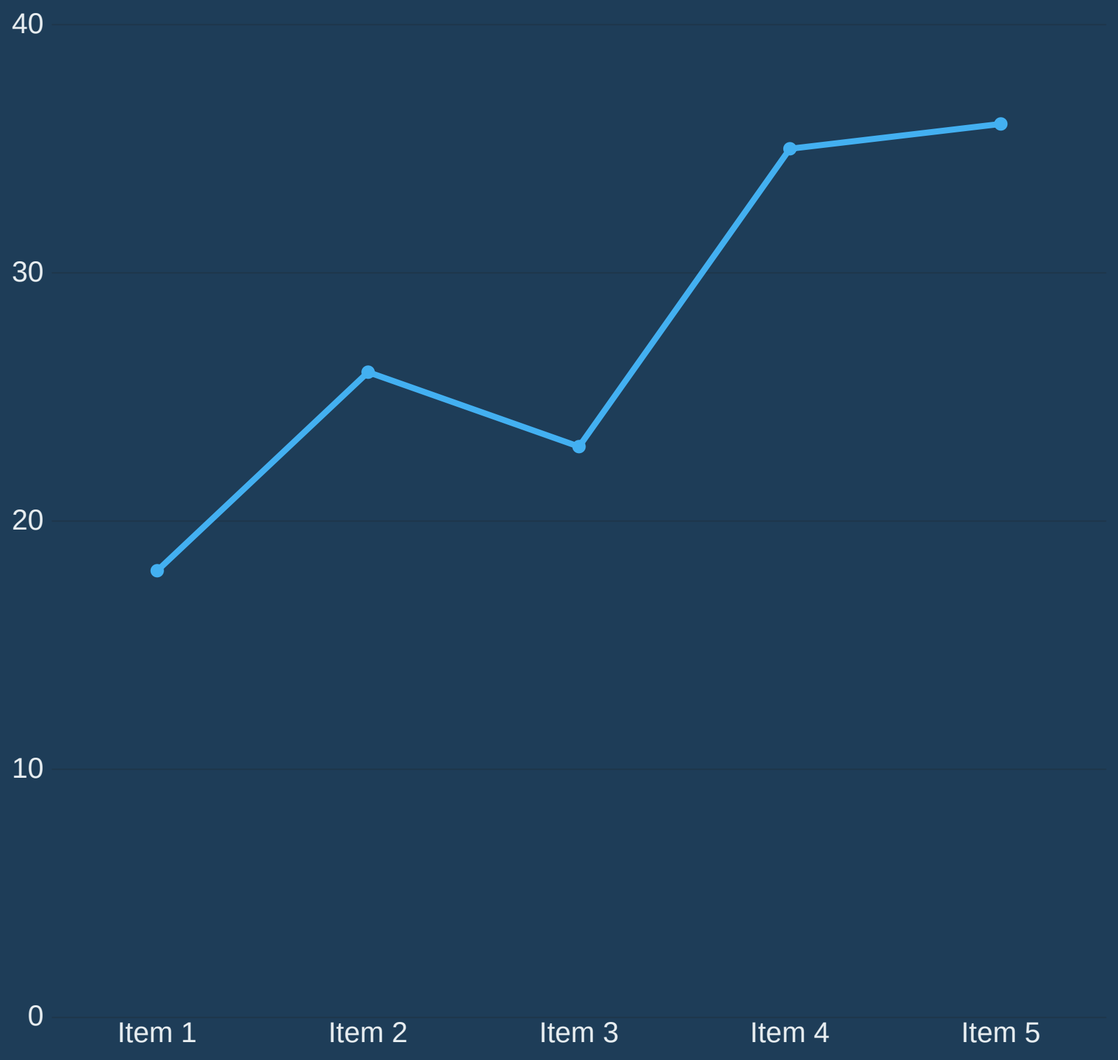
ARNAB SEN

510519006

1-A:
Construct large data sets taking random numbers from normal distribution (UD)

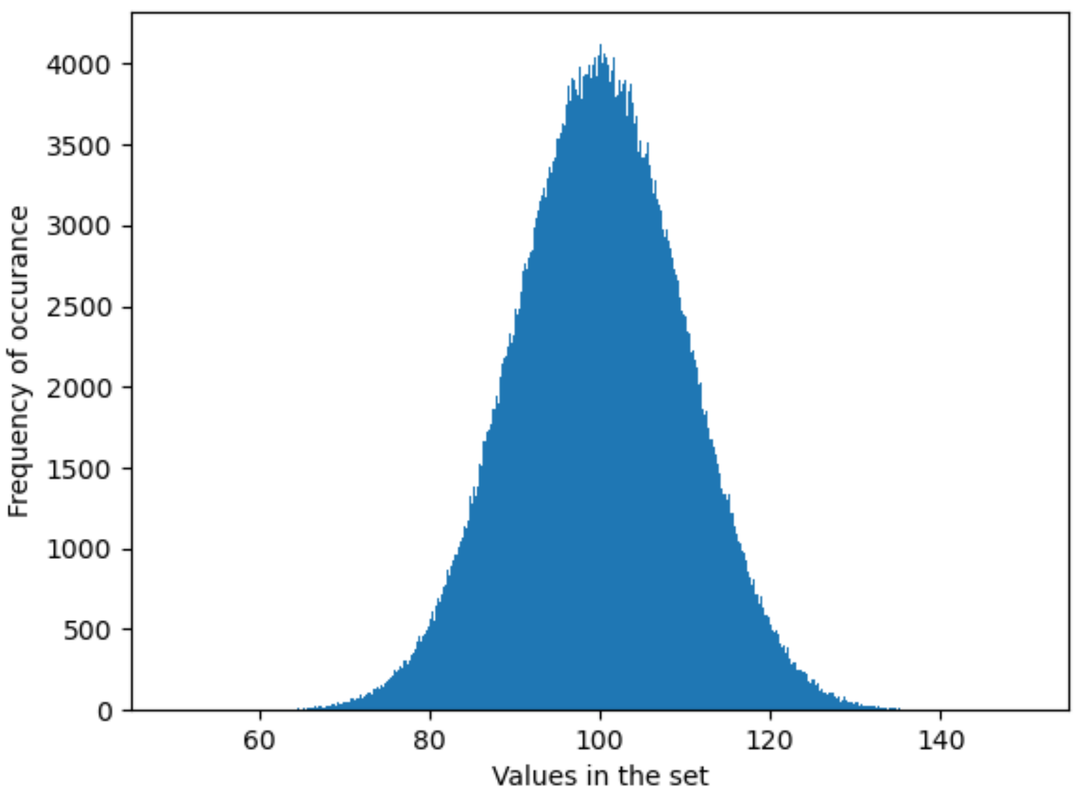
Our Approach

We have created a C program that generates a dataset which is normally distributed using mean and standard deviation. We plotted the histogram of the generated dataset and verified the plot with the standard Be



Visualization

Using Matplotlib (Python)



Plotting of normal dataset

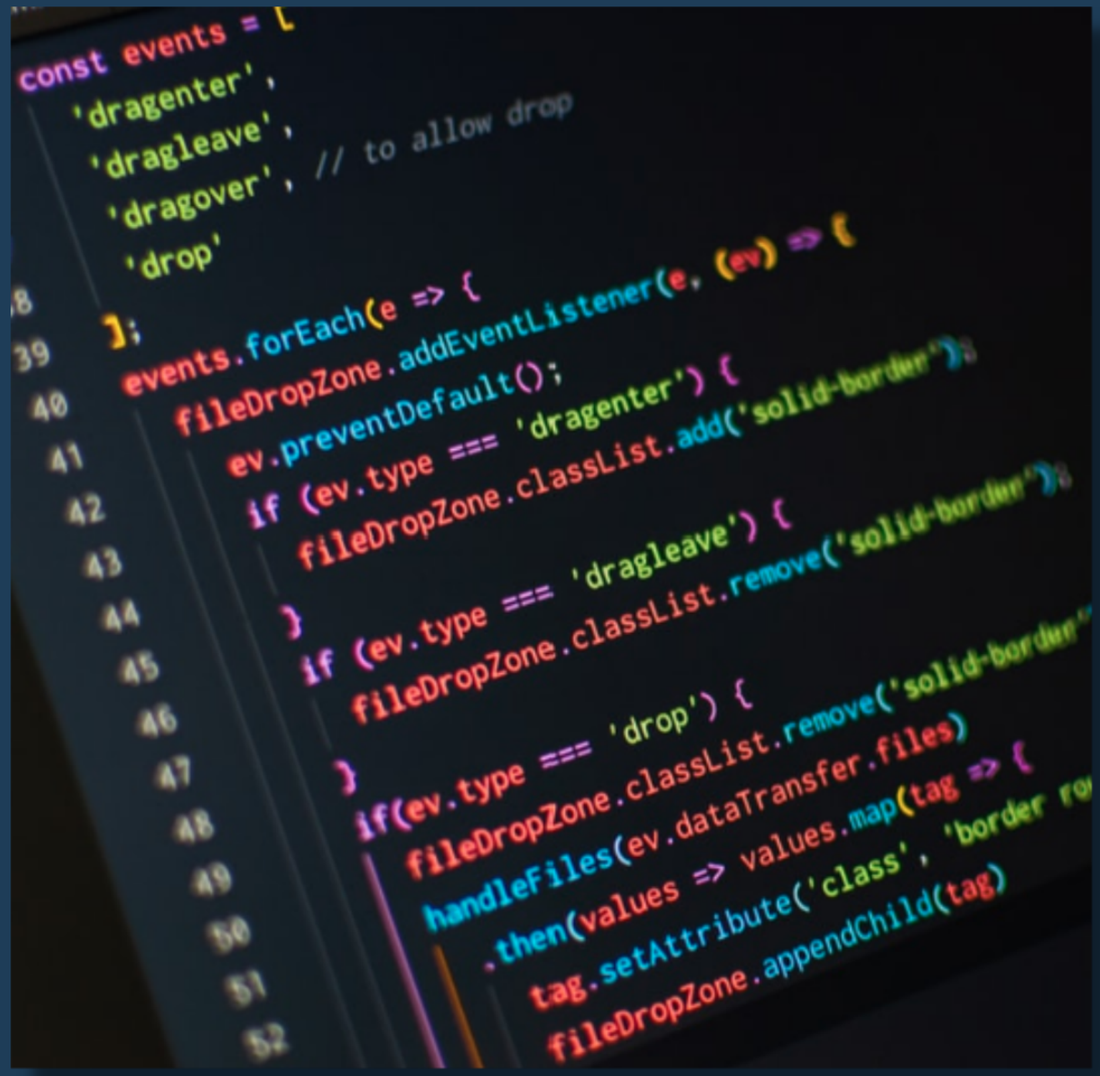
On Y-axis : Frequency of occurnce

On X-axis : Value in the set

Observation

The curve matches with the standard Bell Curve hence the generator function successfully generates a normally distributed dataset

Code in C


const events = [
'dragenter',
'dragleave',
'dragover', // to allow drop
'drop'
];
events.forEach(e => {
fileDropZone.addEventListener(e, (ev) => {
ev.preventDefault();
if (ev.type === 'dragenter') {
fileDropZone.classList.add('solid-border');
}
if (ev.type === 'dragleave') {
fileDropZone.classList.remove('solid-border');
}
if(ev.type === 'drop') {
fileDropZone.classList.remove('solid-border'
handleFiles(ev.dataTransfer.files)
.then(values => values.map(tag => {
tag.setAttribute('class', 'border ro
fileDropZone.appendChild(tag)

mean()

Takes the values and length of array as params and generates the mean of the array.

stddev()

Takes the values and length of array as params and generates the standard deviation

```c
#define NMAX 10000000

double mean(double *values, int n) {
    int i;
    double s = 0;

    for (i = 0; i < n; i++) {
        s += values[i];
    }
    return s / n;
}

double stddev(double *values, int n) {
    int i;
    double average = mean(values, n);
    double s = 0;

    for (i = 0; i < n; i++) {
        s += (values[i] - average) * (values[i] - average);
    }
    return sqrt(s / (n - 1));
}
```

generate()

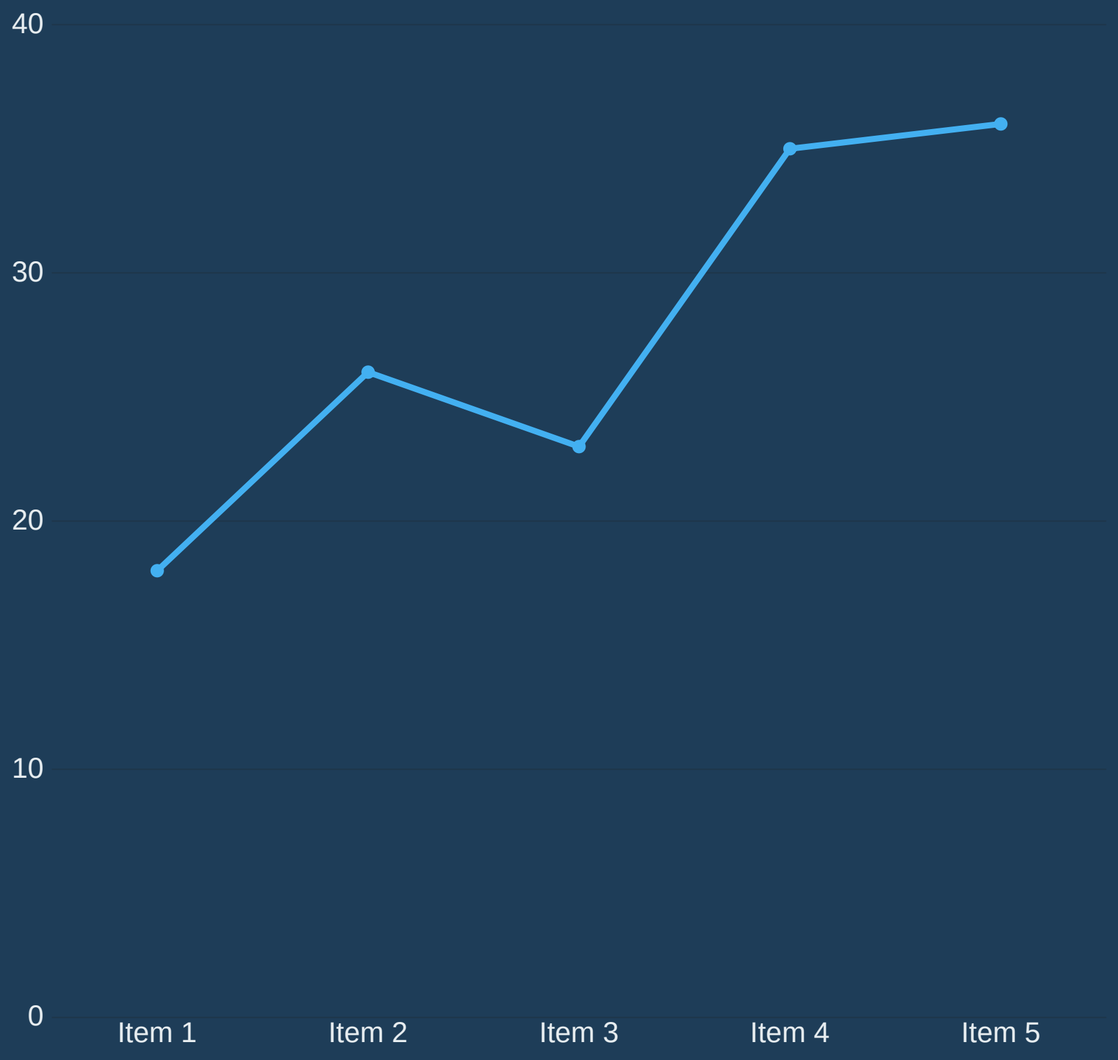
Takes length of array, mean and standard deviation as params and generates the Normal Distribution Data

```c
double *generate(int n, double miu, double sigma) {
    int i;
    int m = n + n % 2;
    double *values = (double *)calloc(m, sizeof(double));
    double average, deviation;

    if (values) {
        for (i = 0; i < m; i += 2) {
            double x, y, rsq, f;
            do {
                x = 2.0 * rand() / (double)RAND_MAX - 1.0;
                y = 2.0 * rand() / (double)RAND_MAX - 1.0;
                rsq = x * x + y * y;
            } while (rsq ≥ 1. || rsq == 0.);
            f = sqrt(-2.0 * log(rsq) / rsq);
            values[i] = miu + (sigma * x * f);
            values[i + 1] = miu + (sigma * y * f);
        }
    }
    return values;
}
```

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

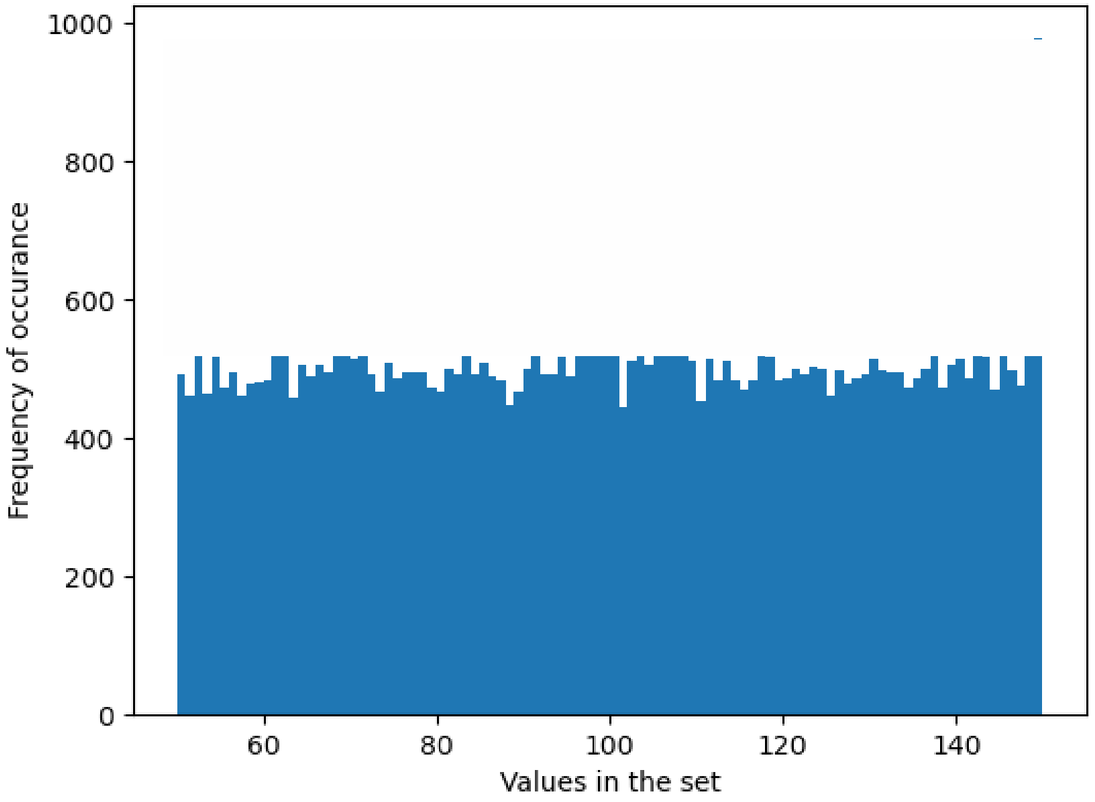
Our Approach

We have created a C program that generates a dataset which is uniformly distributed using the rand() function available in C. Although according to the documentation, the rand() might produce a bias, yet we were able to generate a fairly unfirom dataset by using the modulo operator



Visualization

Using Matplotlib (Python)



Plotting of normal dataset

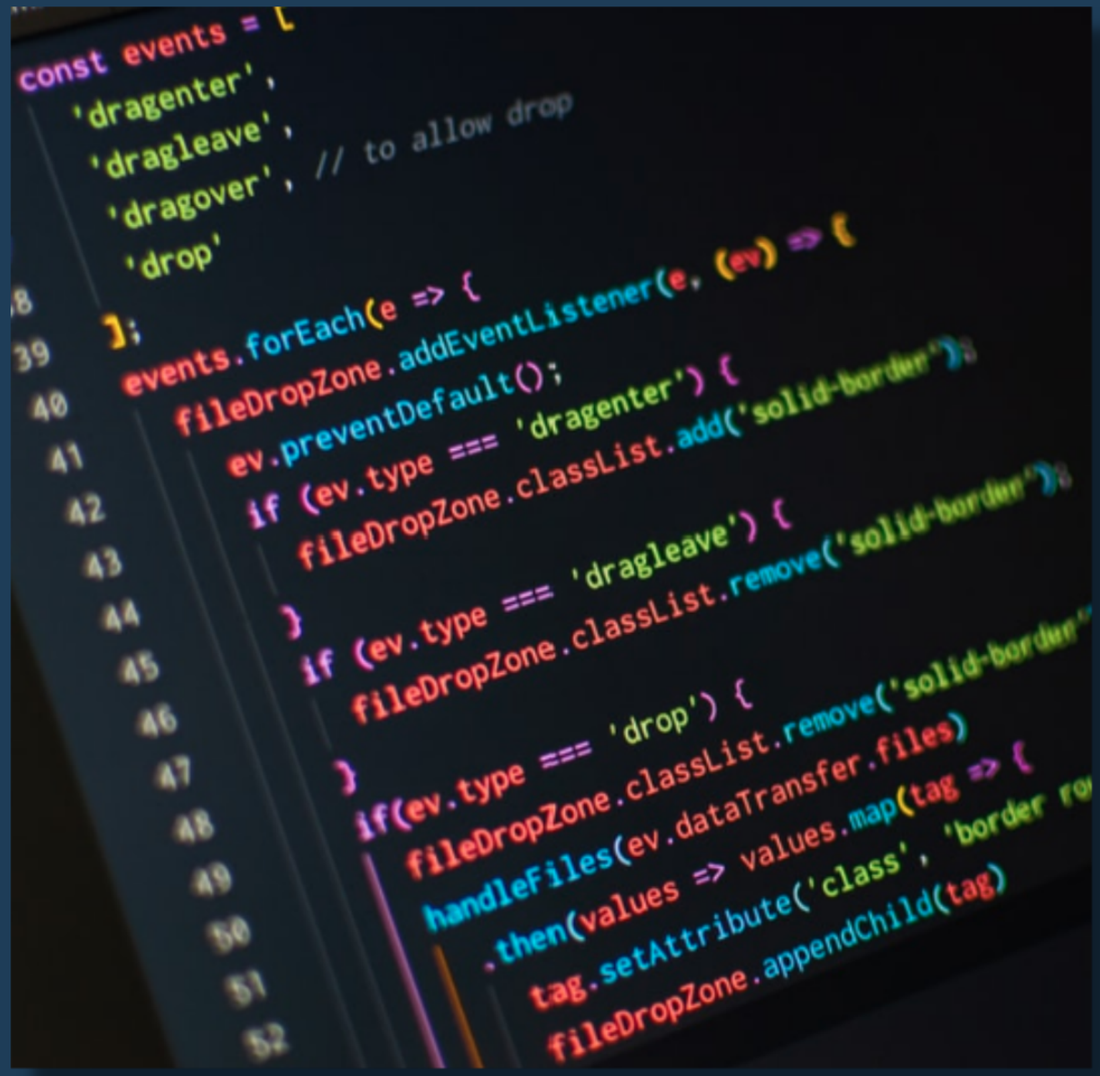
On Y-axis : Frequency of occurnce

On X-axis : Value in the set

Observation

The histogram is almost a straight graph (some irregularities, due to its internal implementation) hence our dataset can be considered as fairly uniform.

Code in C


const events = [
'dragenter',
'dragleave',
'dragover', // to allow drop
'drop'
];
events.forEach(e => {
fileDropZone.addEventListener(e, (ev) => {
ev.preventDefault();
if (ev.type === 'dragenter') {
fileDropZone.classList.add('solid-border');
}
if (ev.type === 'dragleave') {
fileDropZone.classList.remove('solid-border');
}
if(ev.type === 'drop') {
fileDropZone.classList.remove('solid-border'
handleFiles(ev.dataTransfer.files)
.then(values => values.map(tag => {
tag.setAttribute('class', 'border ro
fileDropZone.appendChild(tag)

uniformGenerator()

Takes the lower bound and upper bounds as params and calls the rand() function to generate the random value, which is supposed to be uniform.

```
int uniformGenerator(int hi, int lo)
{
    int range = hi - lo + 1;
    double frac = rand() / (1.0 + RAND_MAX);
    return (int)(frac * range + lo);
}
```

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

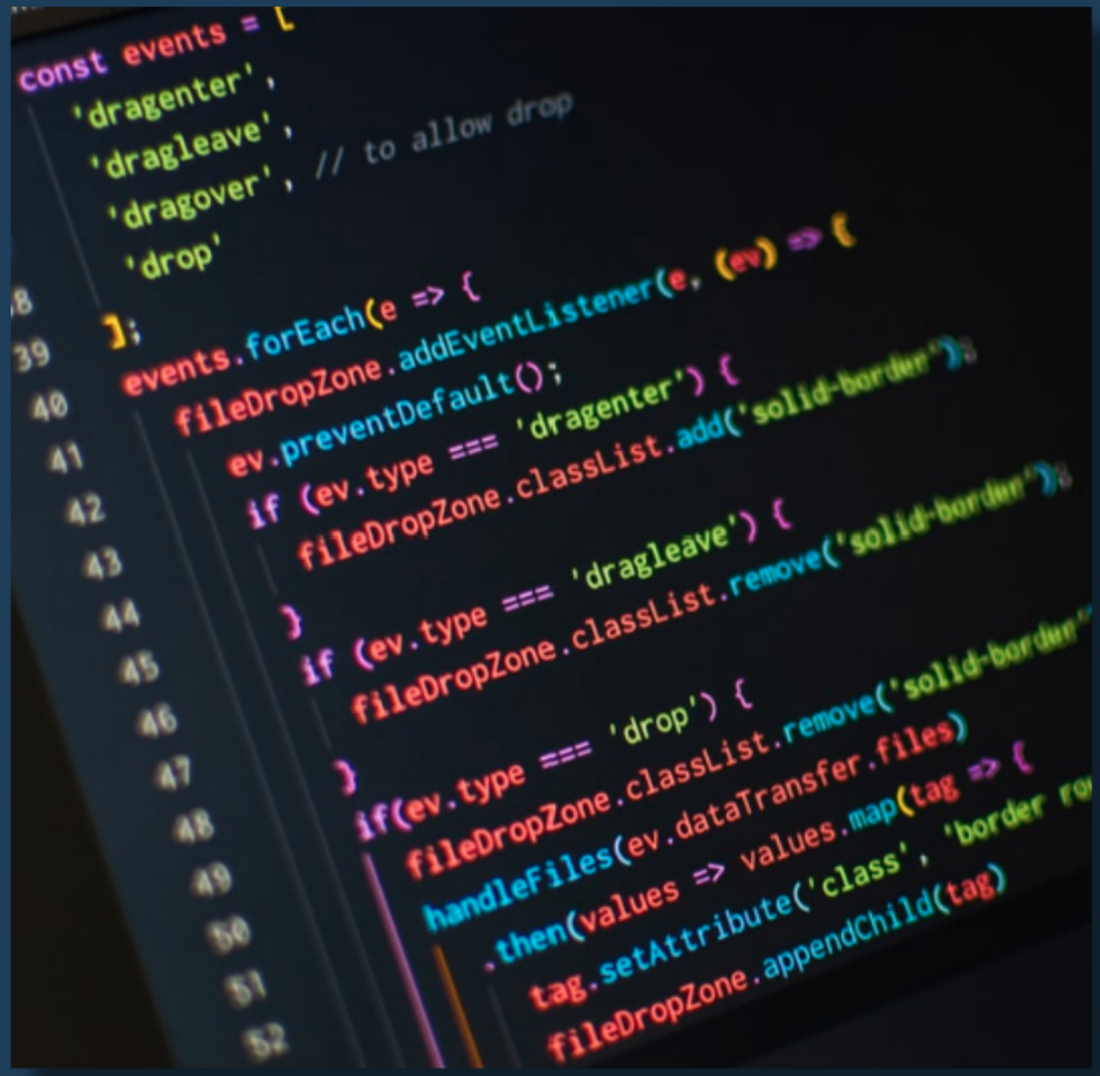
Our Approach

We implemented the recursive Merge sort is a “Divide and Conquer” Algorithm which works as follows:

1. **Divide** n-element sequence into two subarrays
2. **Conquer** sort the two subarrays recursively.
3. **Merge** the two subsequences.

$$T(n) \leq \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{T(\lceil n/2 \rceil)}_{\text{solve left half}} + \underbrace{T(\lfloor n/2 \rfloor)}_{\text{solve right half}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

Code in C


const events = [
'dragenter',
'dragleave',
'dragover', // to allow drop
'drop'
];
events.forEach(e => {
fileDropZone.addEventListener(e, (ev) => {
ev.preventDefault();
if (ev.type === 'dragenter') {
fileDropZone.classList.add('solid-border');
}
if (ev.type === 'dragleave') {
fileDropZone.classList.remove('solid-border');
}
if(ev.type === 'drop') {
fileDropZone.classList.remove('solid-border'
handleFiles(ev.dataTransfer.files)
.then(values => values.map(tag => {
tag.setAttribute('class', 'border ro
fileDropZone.appendChild(tag)

merge()

This is the **Conquer** functionality where we have created a new array which is sorted from the two sorted subsequences.

```
void merge(int arr[], int l, int m, int r)
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;
    int L[n1], R[n2];
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    i = 0;
    j = 0;
    k = l;

    while (i < n1 && j < n2)
    {
        if (L[i] ⩽ R[j])
        {
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
```

```
while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }


    while (j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }
}
```

mergeSort()

This is the recursive calling function, which divides the array into two subarrays

checkForCorrectness()

Simply check where we are comparing the current value with the previous value to check the correctness of our algo.

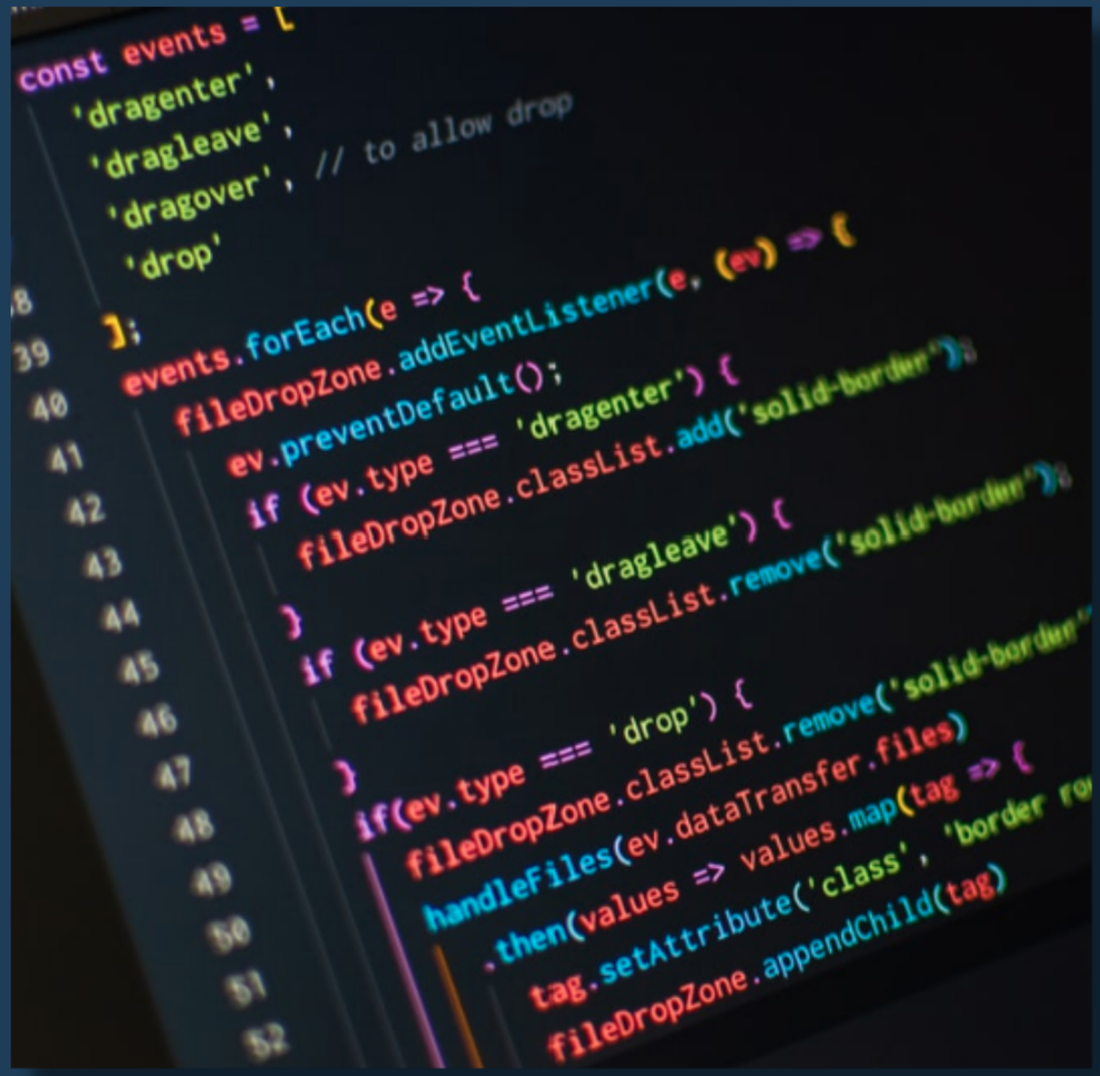
```
void mergeSort(int arr[], int l, int r)
{
    if (l < r)
    {

        int m = l + (r - l) / 2;

        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}

int checkForCorrectness(int *arr, int n)
{
    int correct = 1;
    for (int i = 1; i < n; i++)
    {
        if (arr[i] < arr[i - 1])
        {
            correct = 0;
            break;
        }
    }
    return correct;
}
```

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

Our Approach

Quicksort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked 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.

Code in C


const events = [
'dragenter',
'dragleave',
'dragover', // to allow drop
'drop'
];
events.forEach(e => {
fileDropZone.addEventListener(e, (ev) => {
ev.preventDefault();
if (ev.type === 'dragenter') {
fileDropZone.classList.add('solid-border');
}
if (ev.type === 'dragleave') {
fileDropZone.classList.remove('solid-border');
}
if(ev.type === 'drop') {
fileDropZone.classList.remove('solid-border'
handleFiles(ev.dataTransfer.files)
.then(values => values.map(tag => {
tag.setAttribute('class', 'border ro
fileDropZone.appendChild(tag)

mySwap()

Custom swap function since C doesn't have its own.

partition()

divides the array into two sections. One where all the elements are less than the pivot, other where all the elements are greater.

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

int partition(int arr[], int low, int high)
{
    int pivot = arr[high];
    int i = low - 1;

    for (int curr = low; curr ⩽ high; curr++)
    {
        if (arr[curr] < pivot)
        {
            i++;
            mySwap(&arr[i], &arr[curr]);
        }
    }
    mySwap(&arr[i + 1], &arr[high]);
    return (i + 1);
}
```

quickSort()

recursive call to the partition and itself to eventually sort the unsorted array. Here the pivot is the last element.

checkForCorrectness()

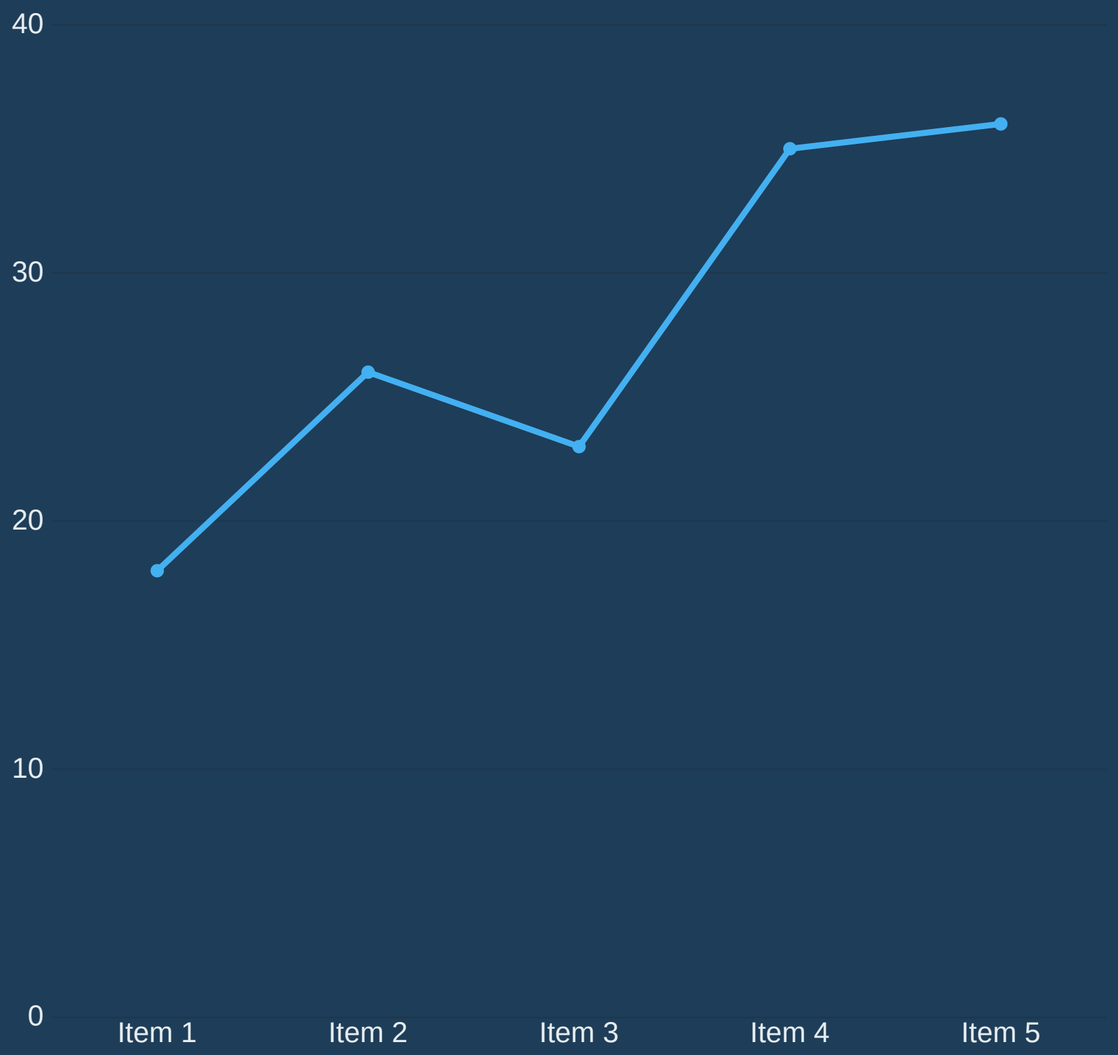
Simply check where we are comparing the current value with the previous value to check the correctness of our algo.

```c
void quickSort(int arr[], int low, int high)
{
    if (low ⩾ high)
    {
        return;
    }
    int pi = partition(arr, low, high);
    quickSort(arr, low, pi - 1);
    quickSort(arr, pi + 1, high);
}

int checkForCorrectness(int *arr, int n)
{
    int correct = 1;
    for (int i = 1; i < n; i++)
    {
        if (arr[i] < arr[i - 1])
        {
            correct = 0;
            break;
        }
    }
    return correct;
}
```

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

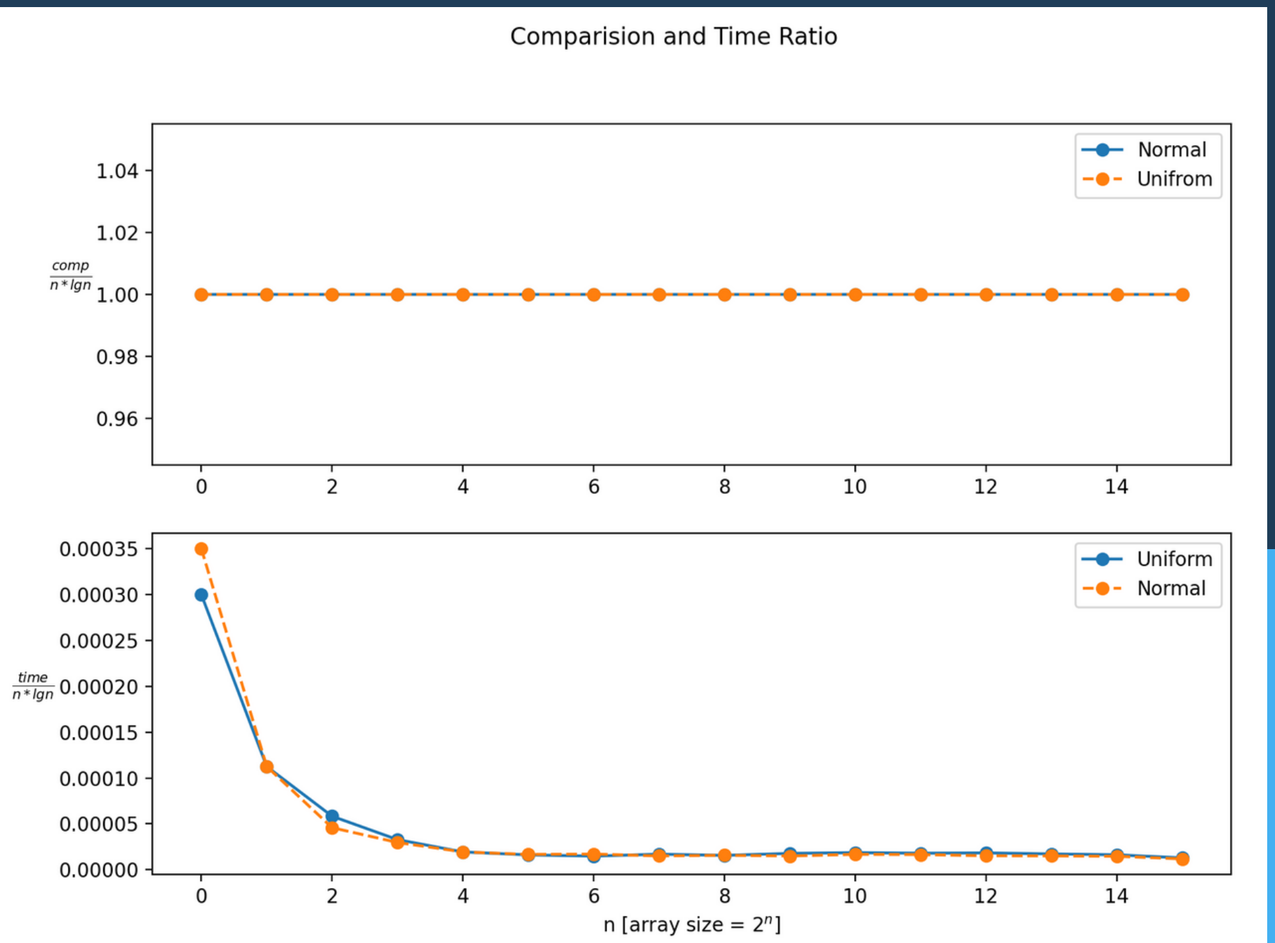
Our Approach for QS

We took the previously generated datasets and passed it through our sorting function, also tracked the comparisons and the total time taken. Using these datas, we have further plotted the graph.



Visualization for MS

Using Matplotlib (Python)



Plotting of normal dataset

On Y-axis : Comparison and Time ratio

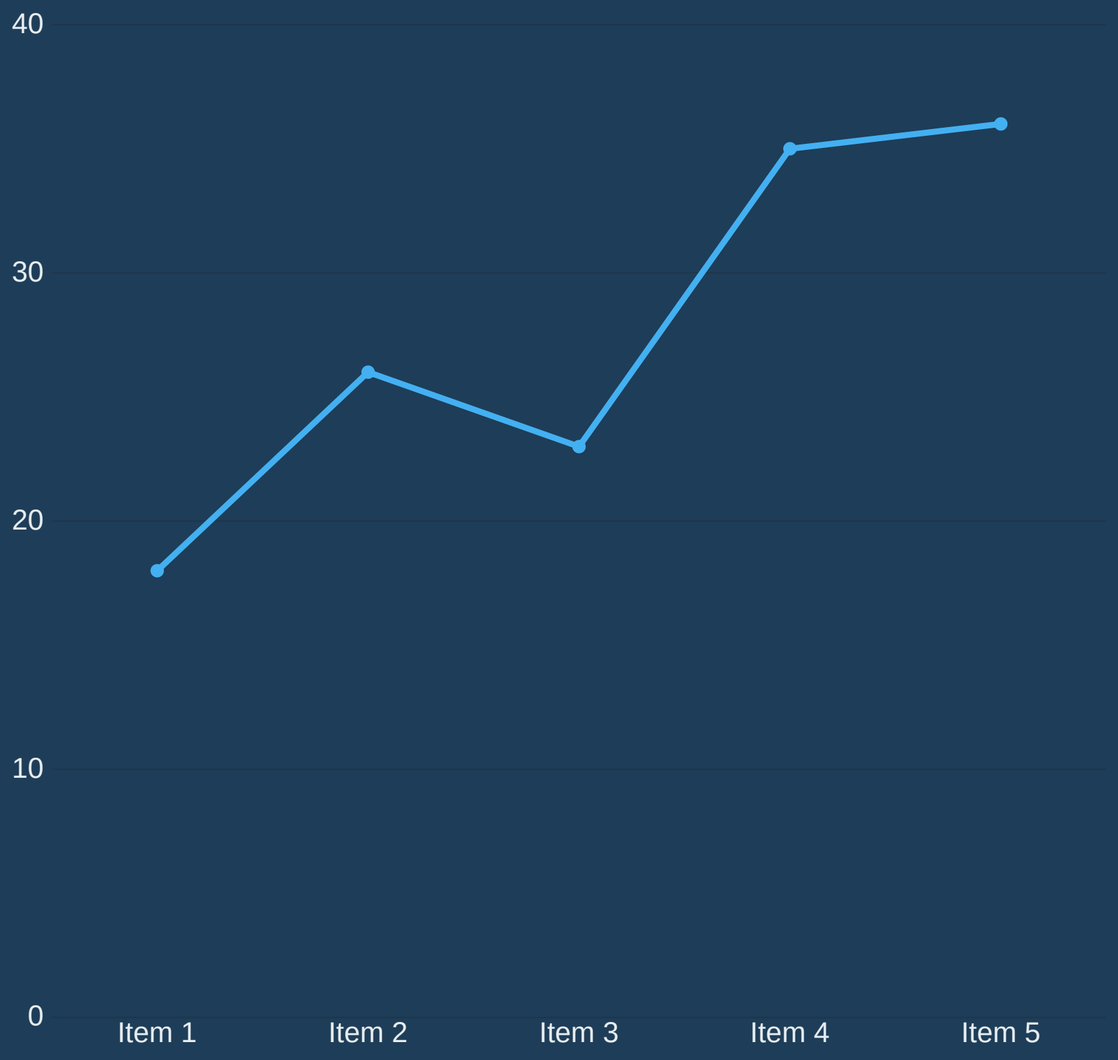
On X-axis : Size of the array in log2 scale

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

Our Approach for QS

We took the previously generated datasets and passed it through our sorting function, also tracked the comparisons and the total time taken. Using these datas, we have further plotted the graph.



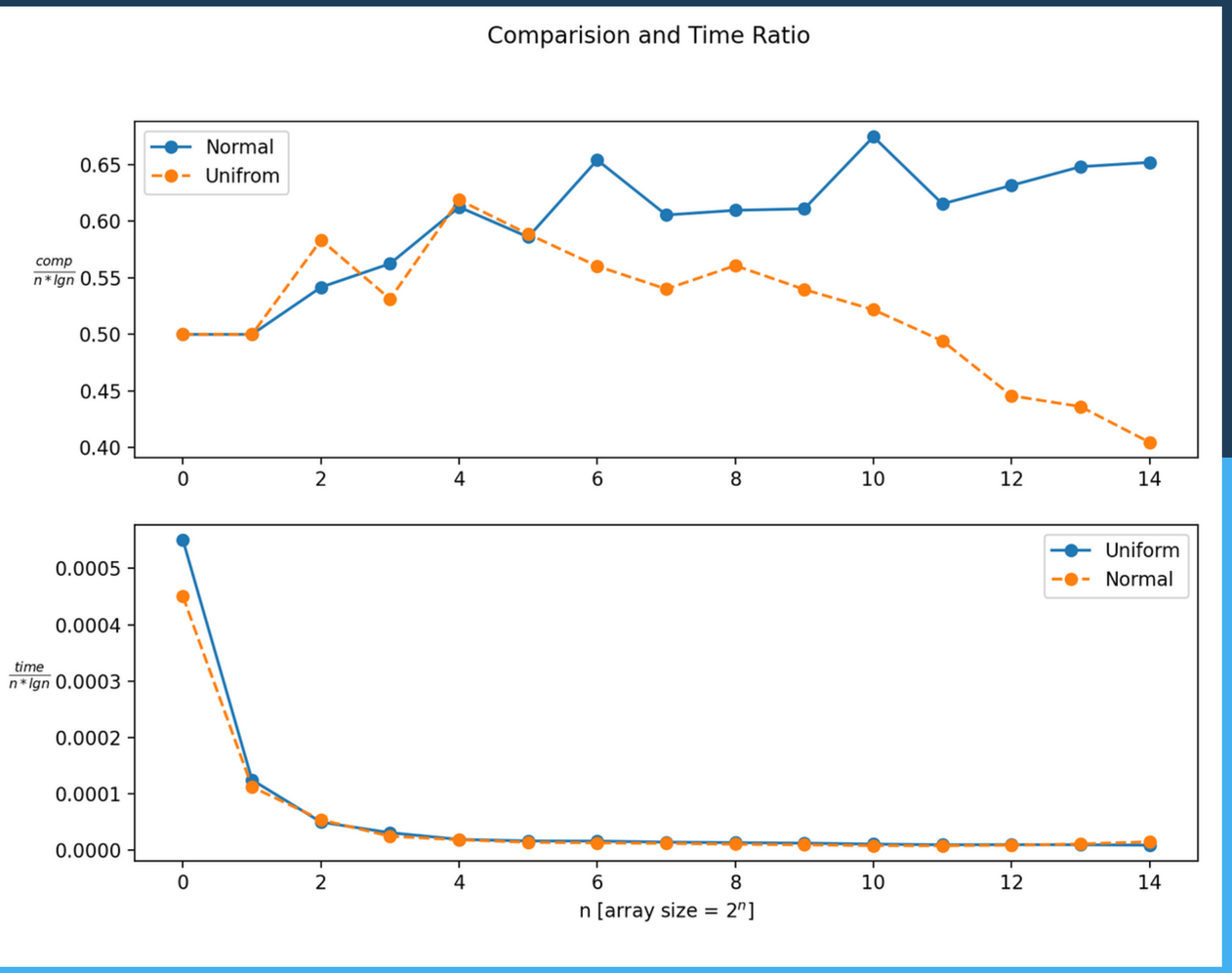
Visualization of QS

Using Matplotlib (Python)

Plotting of normal dataset

On Y-axis : Comparison Ratio and Time Ratio

On X-axis : Size of the array in log2 scale



Observation

As array size keep increasing, we see peculiar observations in Time ratio

1. Time ratio diverges for very high array size
2. Although Comparison Taken is lower for Normal, Time Taken for **Normal Distribution** is higher than **Uniform Dataset**

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.

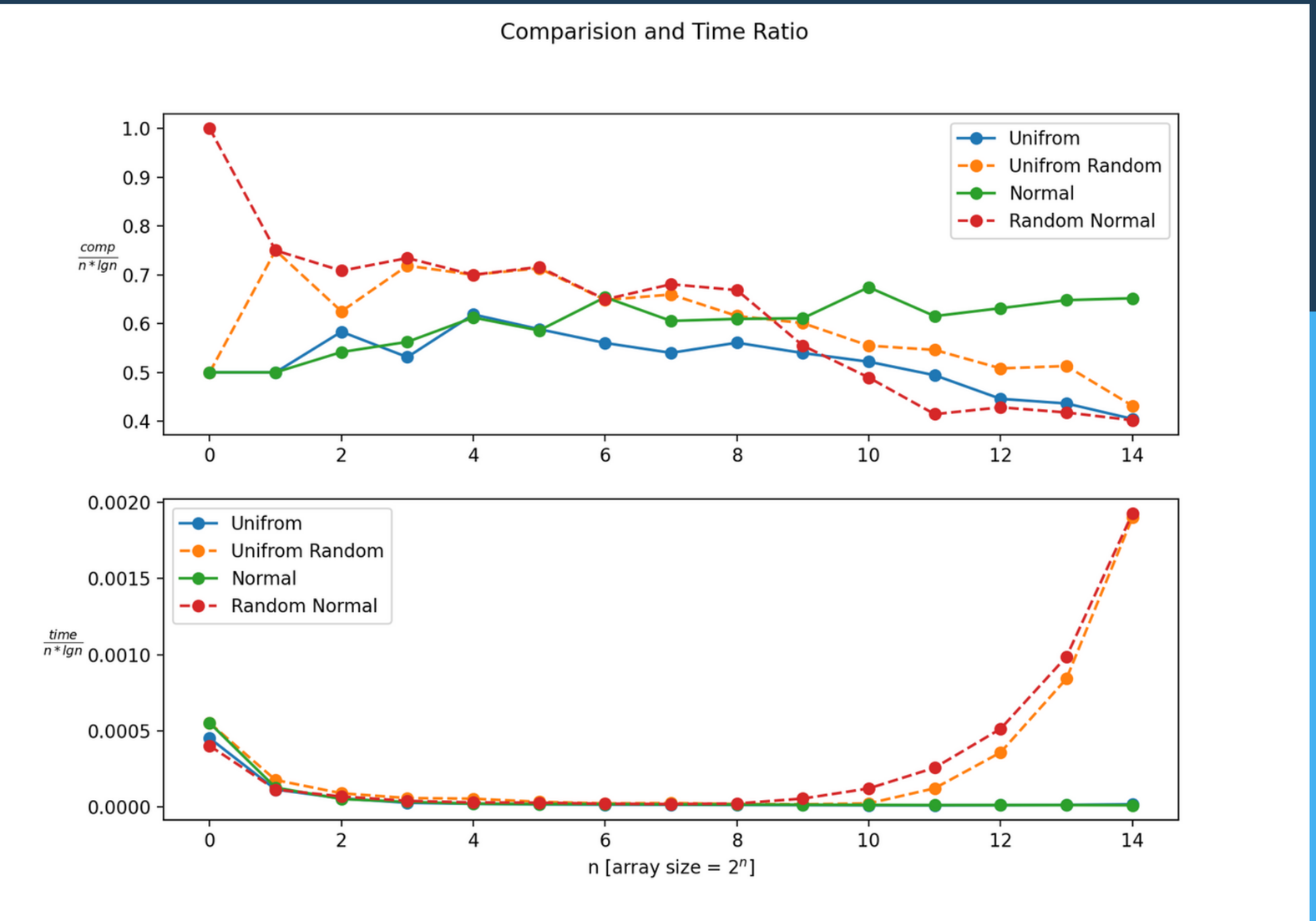
Our Approach

We 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 that the pivot. Then we recursively call the same procedure for left and right subarrays. Here we consider the pivot randomly

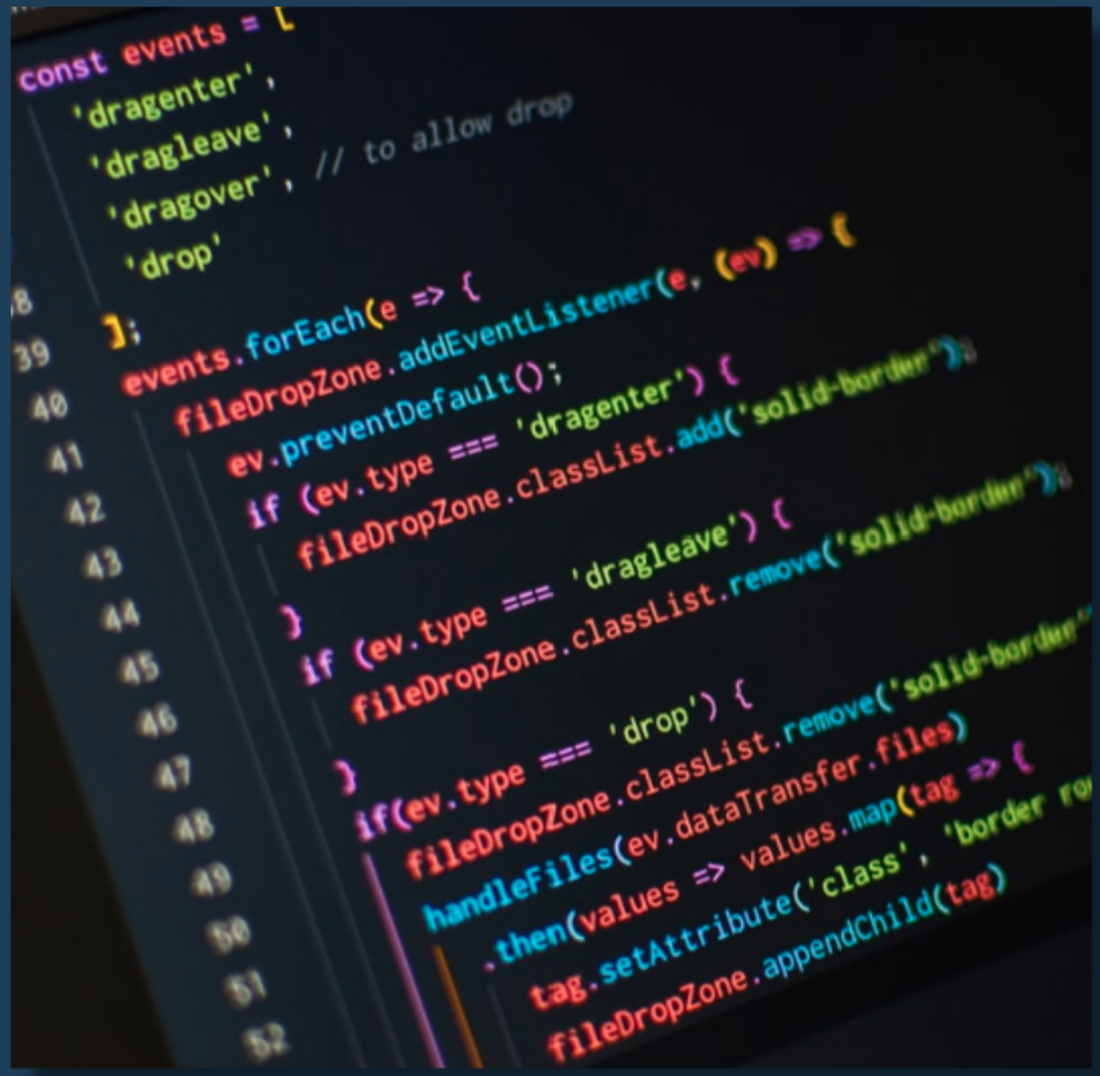
Plotting of normal dataset

On Y-axis : Comparison Ratio

On X-axis : Size of the array in log2 scale



Code in C


const events = [
'dragenter',
'dragleave',
'dragover', // to allow drop
'drop'
];
events.forEach(e => {
fileDropZone.addEventListener(e, (ev) => {
ev.preventDefault();
if (ev.type === 'dragenter') {
fileDropZone.classList.add('solid-border');
}
if (ev.type === 'dragleave') {
fileDropZone.classList.remove('solid-border');
}
if(ev.type === 'drop') {
fileDropZone.classList.remove('solid-border'
handleFiles(ev.dataTransfer.files)
.then(values => values.map(tag => {
tag.setAttribute('class', 'border ro
fileDropZone.appendChild(tag)

mySwap()

Custom swap function since C doesn't have its own.

partition()

divides the array into two sections. One where all the elements are less than the pivot, other where all the elements are greater.

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

int partition(int arr[], int low, int high, int *cnt)
{
    int pivot = arr[high];
    int i = low - 1;

    for (int curr = low; curr ⩽ high; curr++)
    {
        if (arr[curr] < pivot)
        {
            i++;
            mySwap(&arr[i], &arr[curr]);
            (*cnt)++;
        }
    }
    mySwap(&arr[i + 1], &arr[high]);
    (*cnt)++;
    return (i + 1);
}
```

randomized_partition()

Calculates the random index, and then call the partition array.

quickSort()

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 that the pivot. Then we recursively call the same procedure for left and right subarrays.

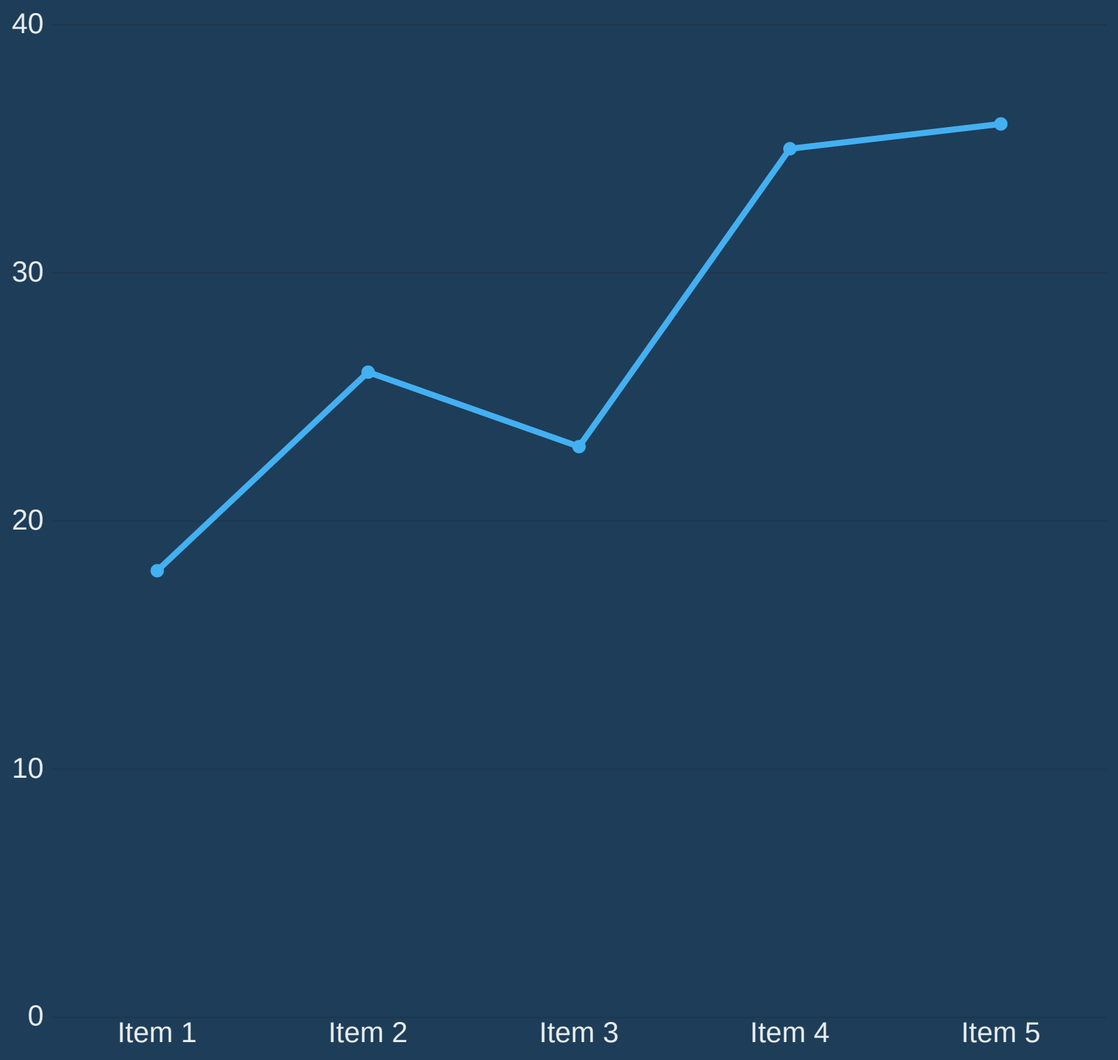
```c
int randomized_partition(int *arr, int initial, int final, int *count)
{
    int i = rand() % (final - initial) + initial;
    mySwap(&arr[final], &arr[i]);
    return partition(arr, initial, final, count);
}

void quickSort(int *arr, int initial, int final, int *count)
{
    if (initial < final)
    {
        int pos_of_pivot = randomized_partition(arr, initial, final,
count);
        quickSort(arr, initial, pos_of_pivot, count);
        quickSort(arr, pos_of_pivot + 1, final, count);
    }
}
                int checkForCorrectness(int *arr, int n)
{
    int correct = 1;
    for (int i = 1; i < n; i++)
    {
        if (arr[i] < arr[i - 1])
        {
            correct = 0;
            break;
        }
    }
    return correct;
}
```

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.

Our Approach

Bucket sort, or bin sort, is a sorting algorithm that works by distributing the elements of an array into a number of buckets. Each bucket is then sorted individually, either using a different sorting algorithm, or by recursively applying the bucket sorting algorithm.



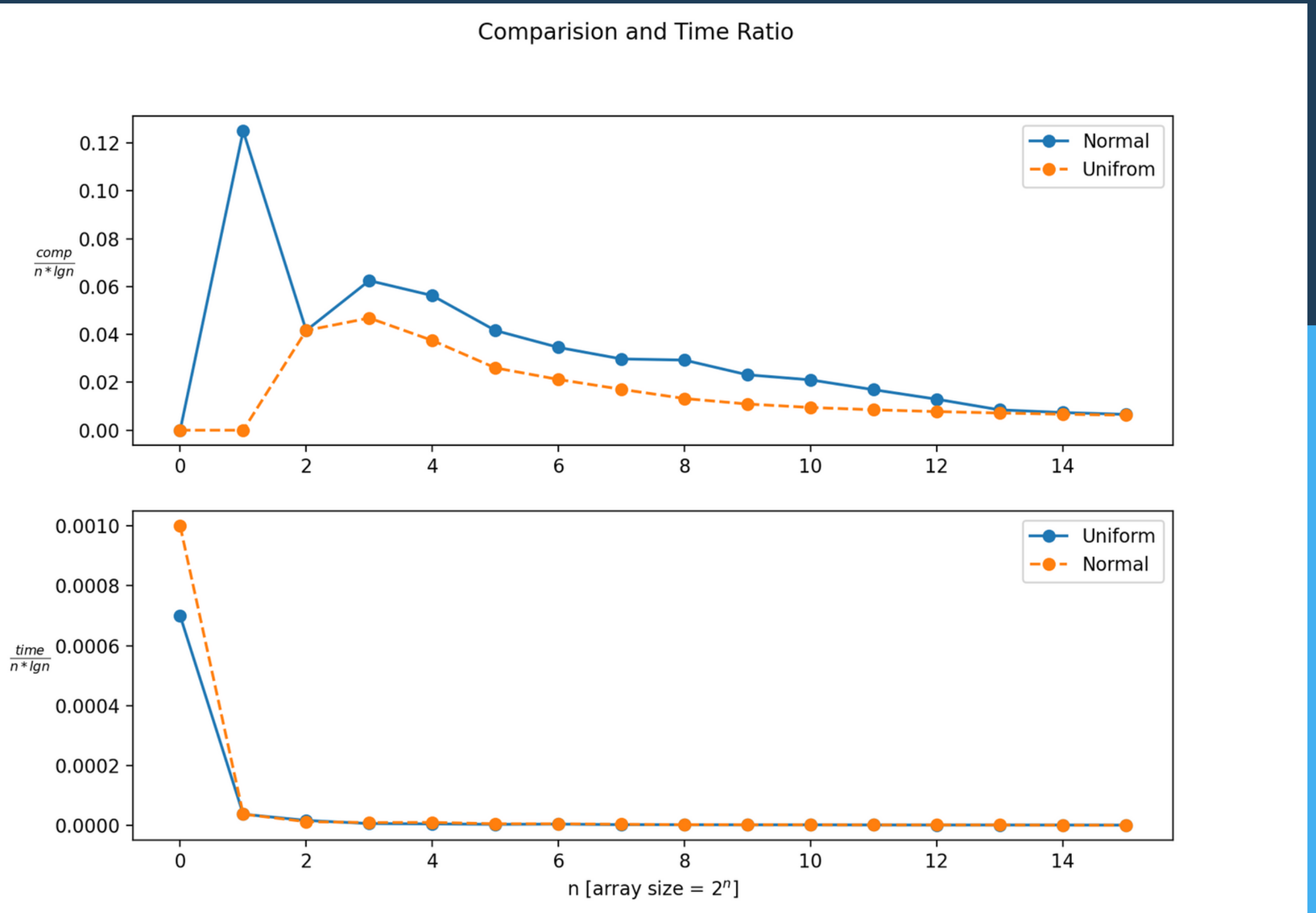
Visualization

Using Matplotlib (Python)

Plotting of normal dataset

On X-axis : Frequency of occurnce

On Y-axis : Value in the set

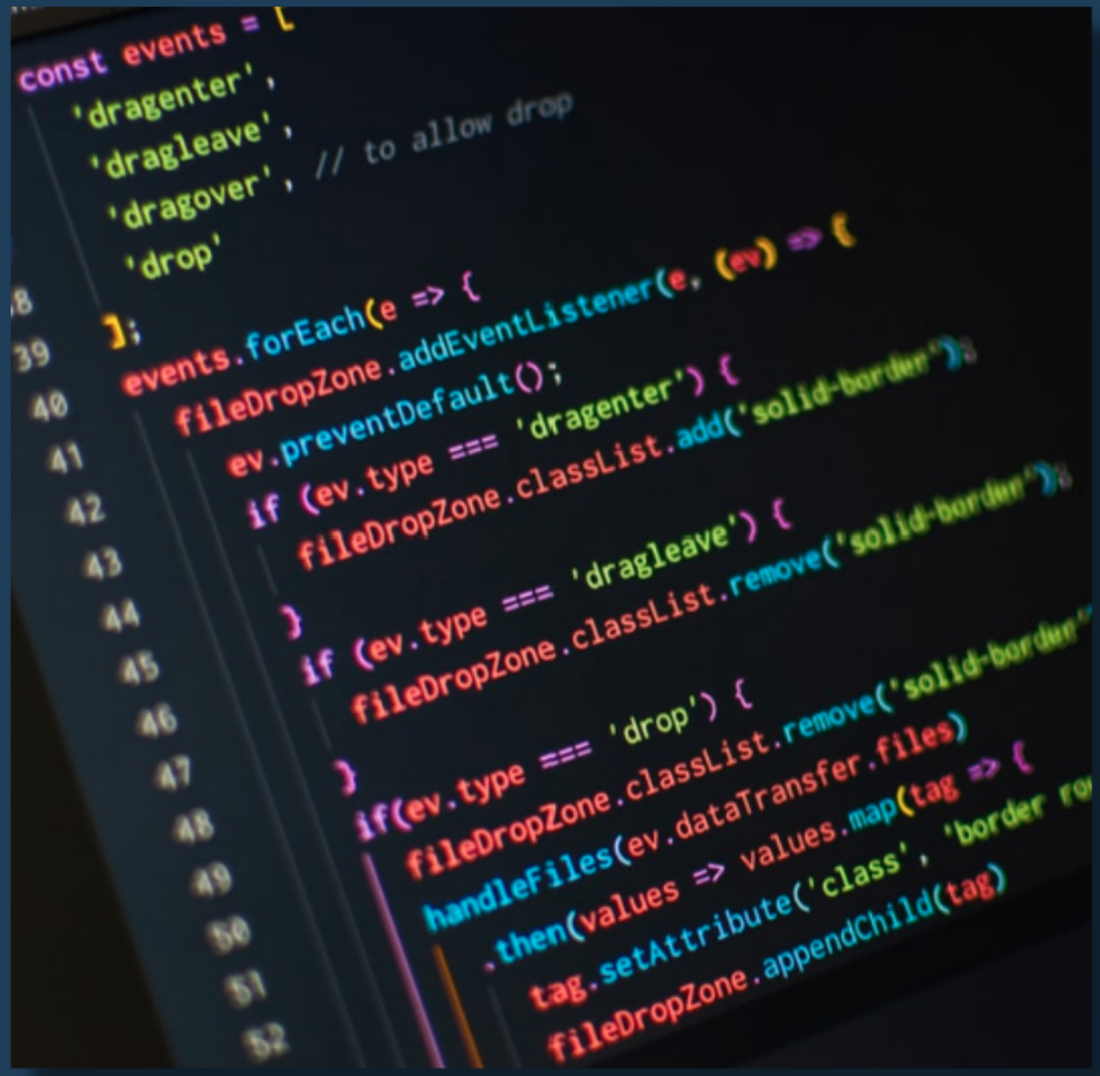


Observations:

For big array size, the both plots for both datasets converges to a
constant, which means that the algorithm implemented is **O(n).**

There are deviations for small n, this could be because of the file reading, and writing is making a big difference, later on its contributions becomes negligible

Code in C


const events = [
'dragenter',
'dragleave',
'dragover', // to allow drop
'drop'
];
events.forEach(e => {
fileDropZone.addEventListener(e, (ev) => {
ev.preventDefault();
if (ev.type === 'dragenter') {
fileDropZone.classList.add('solid-border');
}
if (ev.type === 'dragleave') {
fileDropZone.classList.remove('solid-border');
}
if(ev.type === 'drop') {
fileDropZone.classList.remove('solid-border'
handleFiles(ev.dataTransfer.files)
.then(values => values.map(tag => {
tag.setAttribute('class', 'border ro
fileDropZone.appendChild(tag)

LL_insert_sort()

Inserting the elements inside "buckets" (Linked List) using insertion sort algo.

```
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;
}
```

delete_start()

Delete a start element in a LL

bucket_sort()

Distributing the elements of an array into several buckets. Each bucket is then sorted individually.

```c
node *delete_start(node *list, float *d)
{
    node *temp = list;
    list = list->next;

    *d = temp->data;
    free(temp);

    return list;
}
void bucket_sort(float *arr, int n, int *count)
{
    node *bins[n];

    for (int i = 0; i < n; i++)
        bins[i] = NULL;

    int pos = 0;

    for (int i = 0; i < n; i++)
    {
        pos = floor(n * arr[i]);

        bins[pos] = LL_insert_sorted(bins[pos], arr[i], count);
        (*count)++;
    }

    pos = 0;
    float temp;

    for (int i = 0; i < n; i++)
    {
        while (bins[i] != NULL)
        {
            bins[i] = delete_start(bins[i], &temp);
            arr[pos] = temp;
            pos++;
        }
    }
}
```

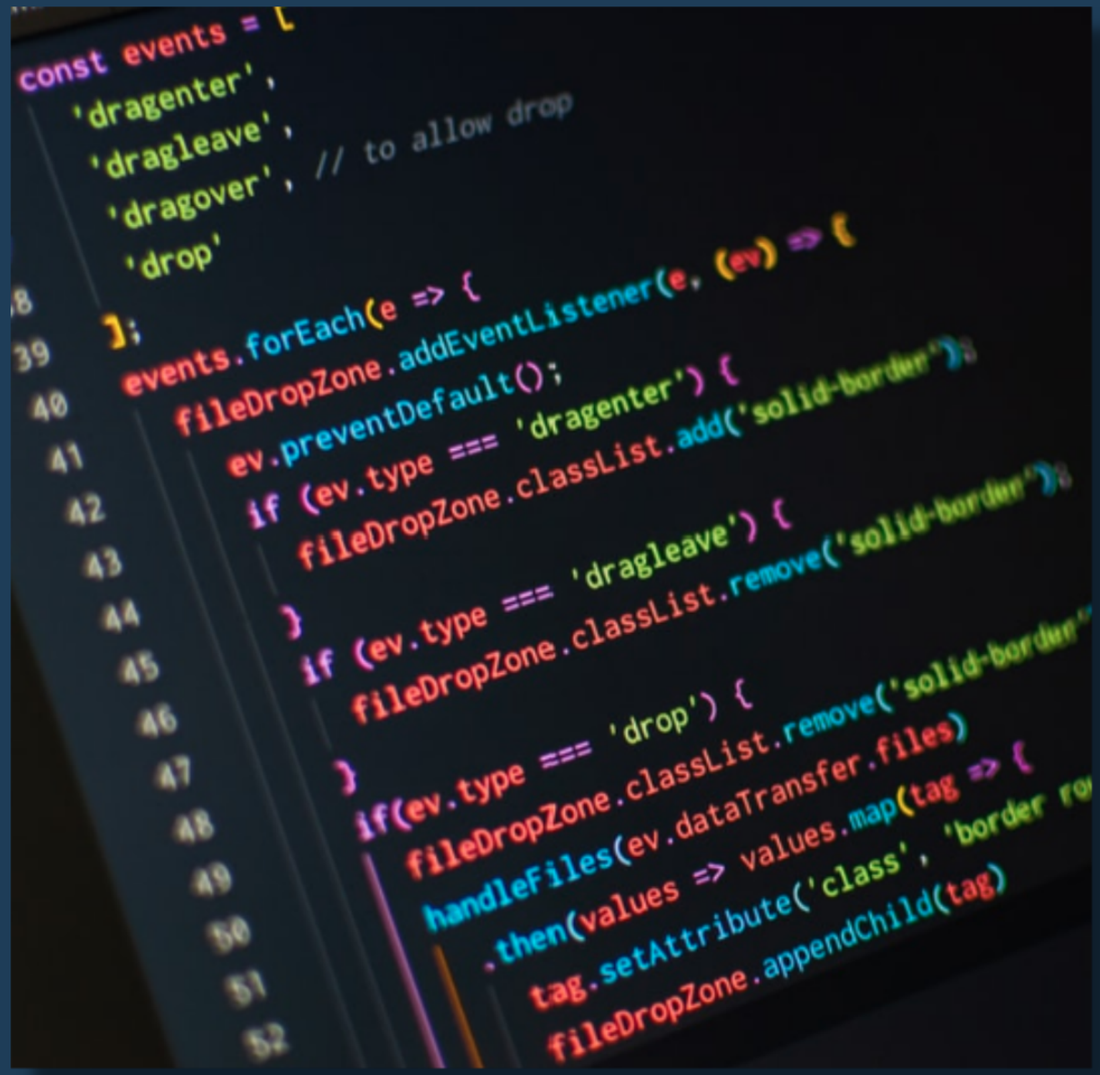
7. Implement the worst case linear median selection algorithm by taking the median of medians (MoM) as the pivotal element and check for correctness.

Our Approach

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 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.

Code in C


const events = [
'dragenter',
'dragleave',
'dragover', // to allow drop
'drop'
];
events.forEach(e => {
fileDropZone.addEventListener(e, (ev) => {
ev.preventDefault();
if (ev.type === 'dragenter') {
fileDropZone.classList.add('solid-border');
}
if (ev.type === 'dragleave') {
fileDropZone.classList.remove('solid-border');
}
if(ev.type === 'drop') {
fileDropZone.classList.remove('solid-border'
handleFiles(ev.dataTransfer.files)
.then(values => values.map(tag => {
tag.setAttribute('class', 'border
fileDropZone.appendChild(tag)

insertion_sort()

SImple implementation of Insertion sort in C

give_median()

returns the median, in the array in the range [initial , final]

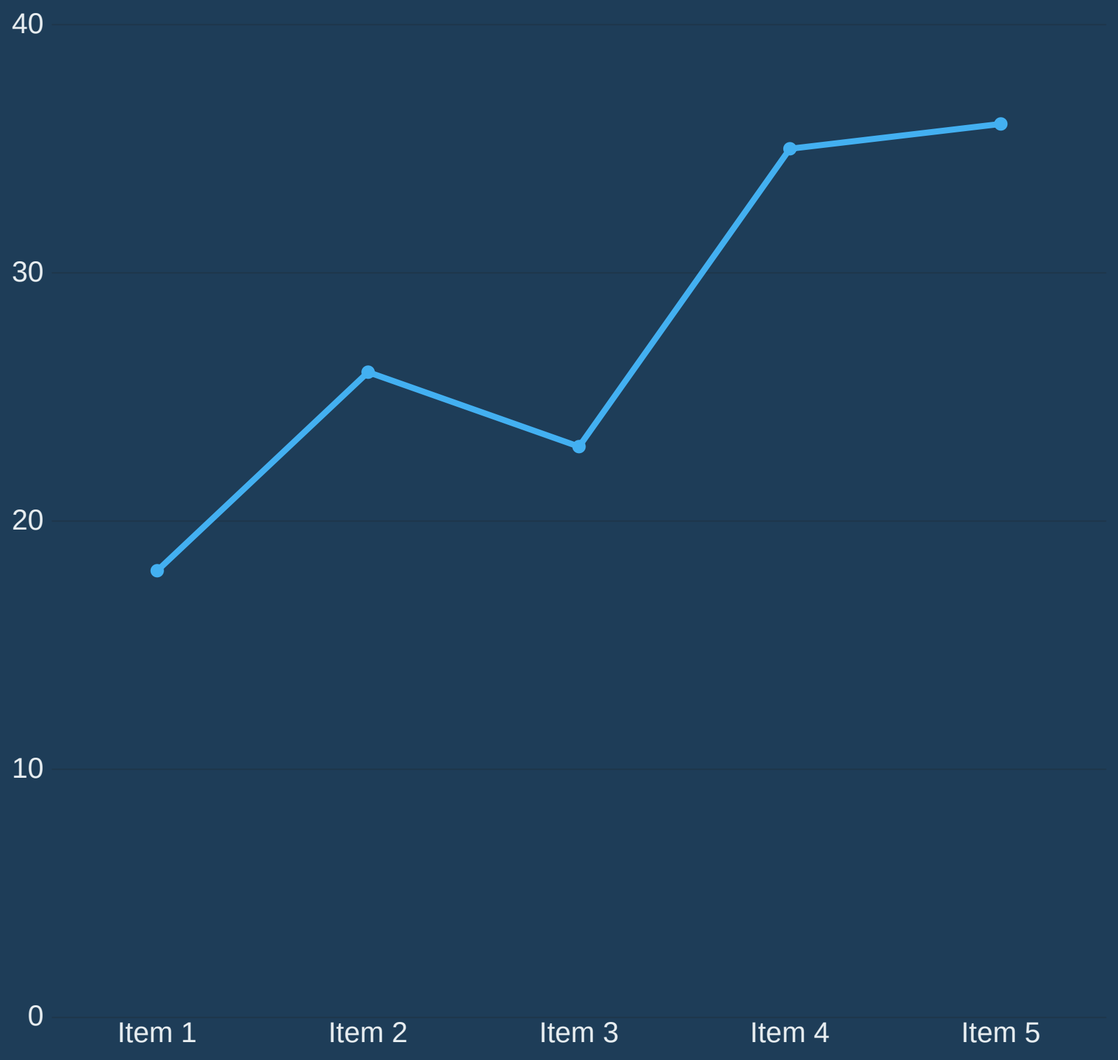
```c
void insertion_sort(short arr[], int initial, int final)
{
    for (int i = initial; i <= final; i++)
    {
        int value = arr[i];
        int pos = i - 1;
        while (pos >= initial && arr[pos] > value)
        {
            arr[pos + 1] = arr[pos];
            pos--;
        }
        arr[pos + 1] = value;
    }
}
int give_median(short arr[], int initial, int final)
{
    insertion_sort(arr, initial, final);
    int mid = (initial + final) / 2;
    return arr[mid];
}
```

median_of_median()

an array is taken and devided recusrsively and the median of the medians of each part is calculated.

```
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 == arr_size)
            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);
}
```

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



Visualization

Using Matplotlib (Python)

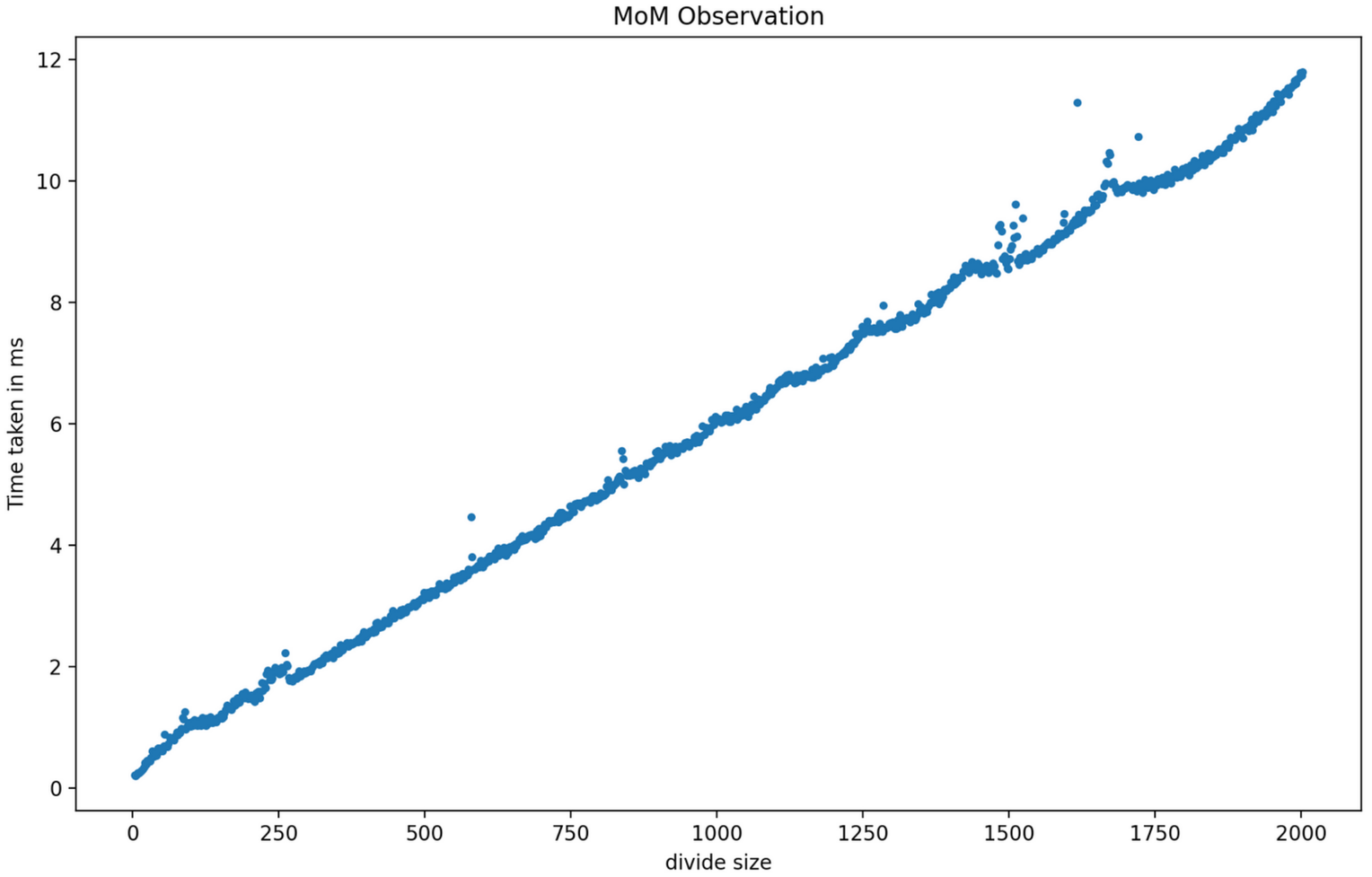
Plotting of normal dataset

On Y-axis : Time Taken

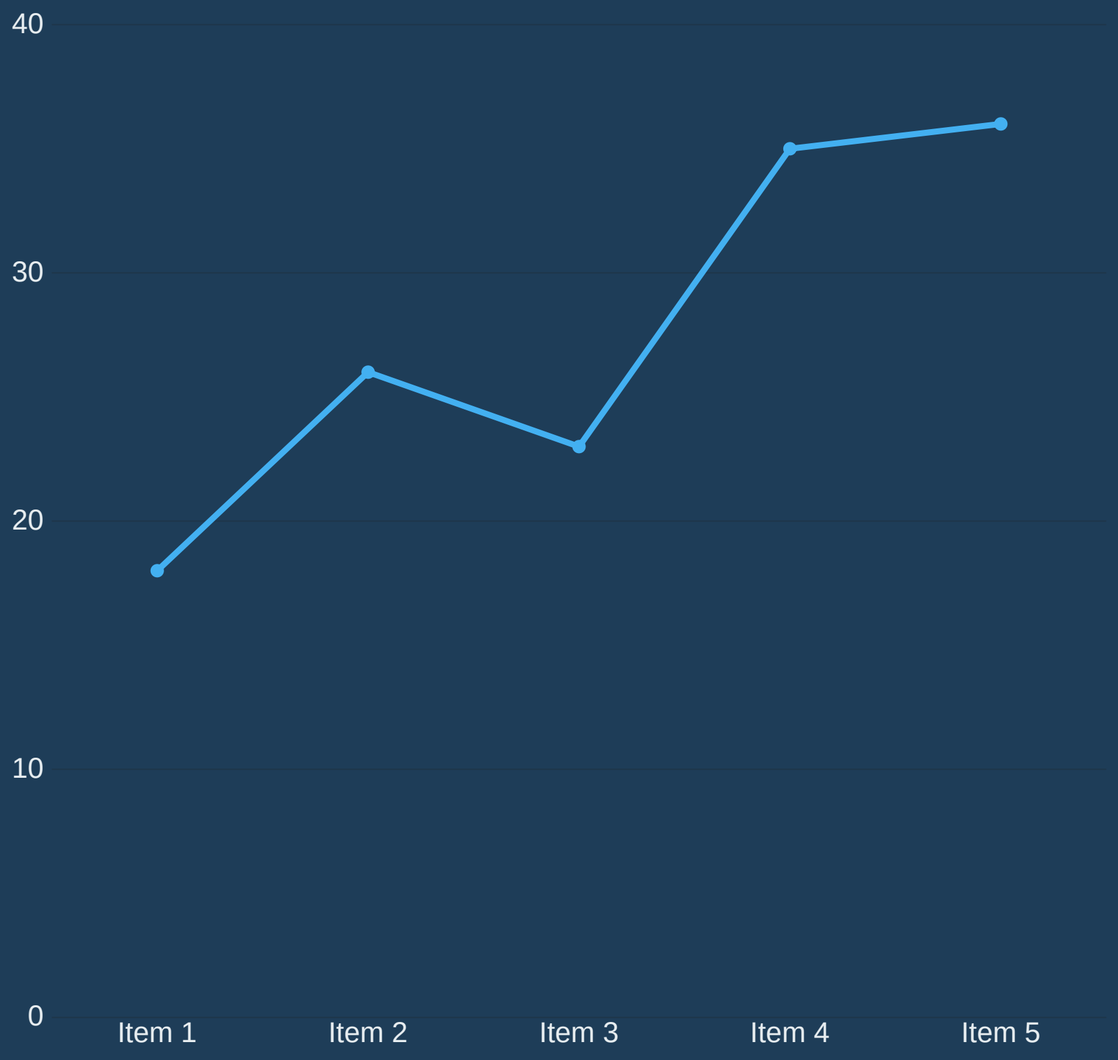
On X-axis : Divide Size

Observation

From the graph we can infer that time complexityis Linear 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.



Visualization

Using Matplotlib (Python)

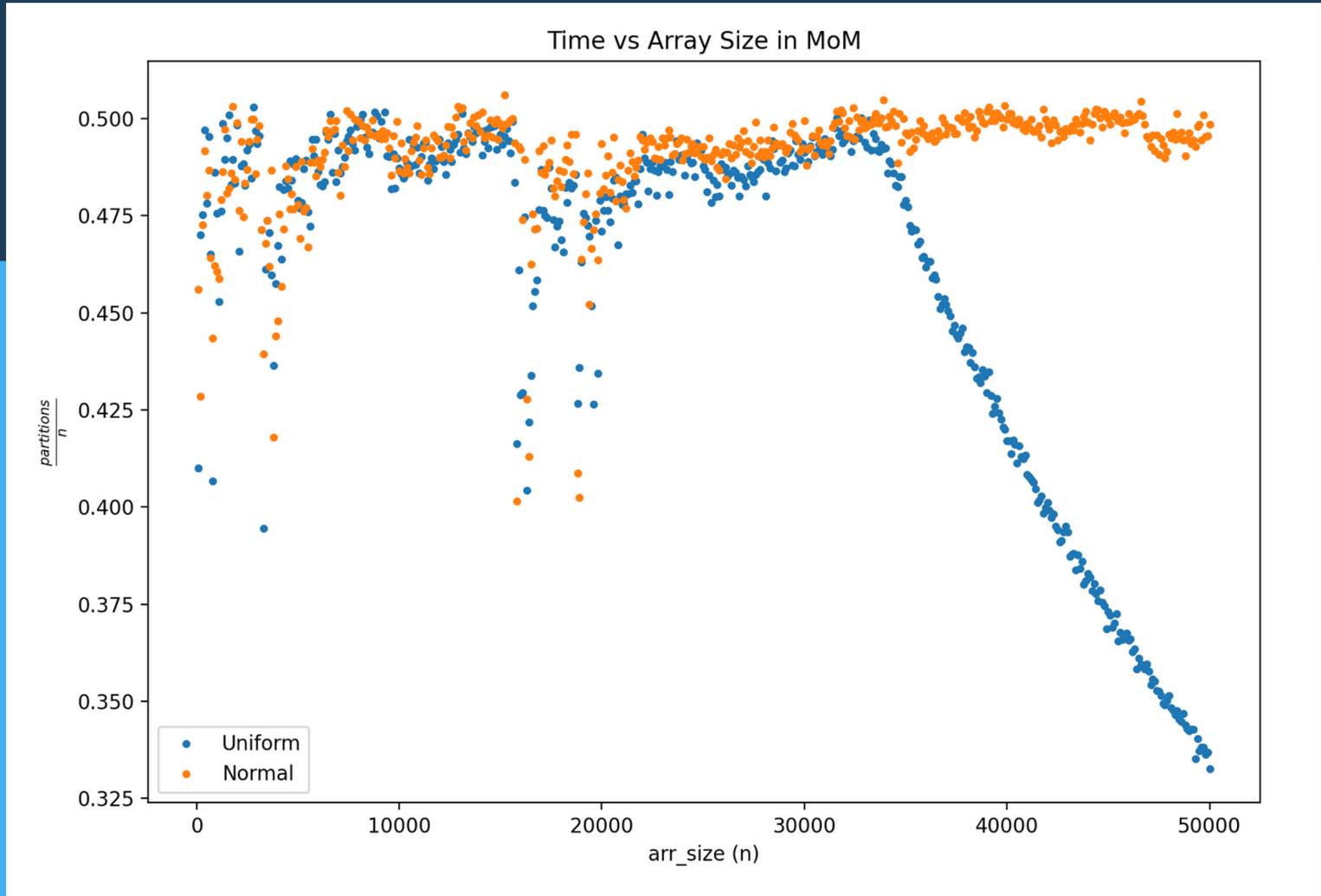
Plotting of normal dataset

On X-axis : arr size ()

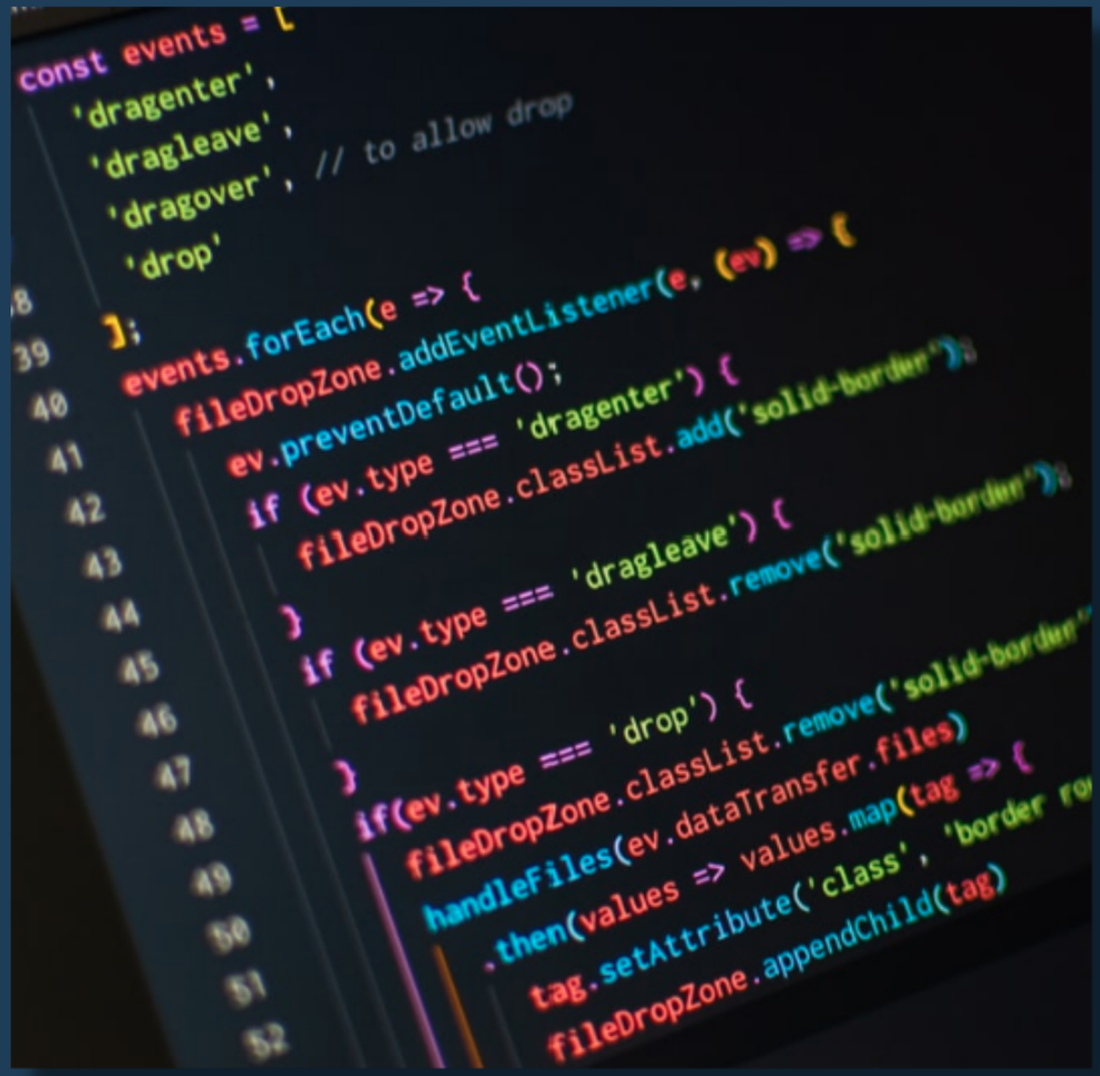
On Y-axis : partitions/n

Observation

This shows that what MoM finds is very close to actual median, and hence MoM can mostly guarantee a good pivot of Quicksort Algorithm.



Code in C


const events = [
'dragenter',
'dragleave',
'dragover', // to allow drop
'drop'
];
events.forEach(e => {
fileDropZone.addEventListener(e, (ev) => {
ev.preventDefault();
if (ev.type === 'dragenter') {
fileDropZone.classList.add('solid-border');
}
if (ev.type === 'dragleave') {
fileDropZone.classList.remove('solid-border');
}
if(ev.type === 'drop') {
fileDropZone.classList.remove('solid-border'
handleFiles(ev.dataTransfer.files)
.then(values => values.map(tag => {
tag.setAttribute('class', 'border
fileDropZone.appendChild(tag)

insertion_sort()

SImple implementation of Insertion sort in C

give_median()

Evaluates the median of the array
after sorting

```c
void insertion_sort(double arr[], int initial, int final)
{
    for (int i = initial; i ⩽ final; i++)
    {
        double value = arr[i];
        int pos = i - 1;
        while (pos ⩾ initial && arr[pos] > value)
        {
            arr[pos + 1] = arr[pos];
            pos--;
        }
        arr[pos + 1] = value;
    }
}

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

median_of_median()

an array is taken and devided recusrsively and the median of the medians of each part is calculated.

```
double median_of_median(double arr[], int arr_size, int divide_size)
{
    if (arr_size < divide_size)
    {
        double 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;

    double 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);
}
```

partition()

divides the array into two sections. One where all the elements are less than the pivot, other where all the elements are greater.

findPartition()

find the partition about the median

```
int partition(double arr[], int low, int high)
{
    double pivot = arr[high];
    int i = low - 1;

    for (int curr = low; curr ⩽ high; curr++)
    {
        if (arr[curr] < pivot)
        {
            i++;
            swap(&arr[i], &arr[curr]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

int findPartition(double *arr, int arr_size, int divide_size)
{
    double val = median_of_median(arr, arr_size, divide_size);

    for (int i = 0; i < arr_size; i++)
        if (arr[i] == val)
        {
            swap(&arr[arr_size - 1], &arr[i]);
            return partition(arr, 0, arr_size - 1);
        }
}
```

THANK YOU