Assignment 1

Made By- Ramesh Chandra Soren (2022CSB086)
Santosh Methre (2022CSB084)
Deep Saha (2022CSB078)

Algorithm Lab(CS2271)

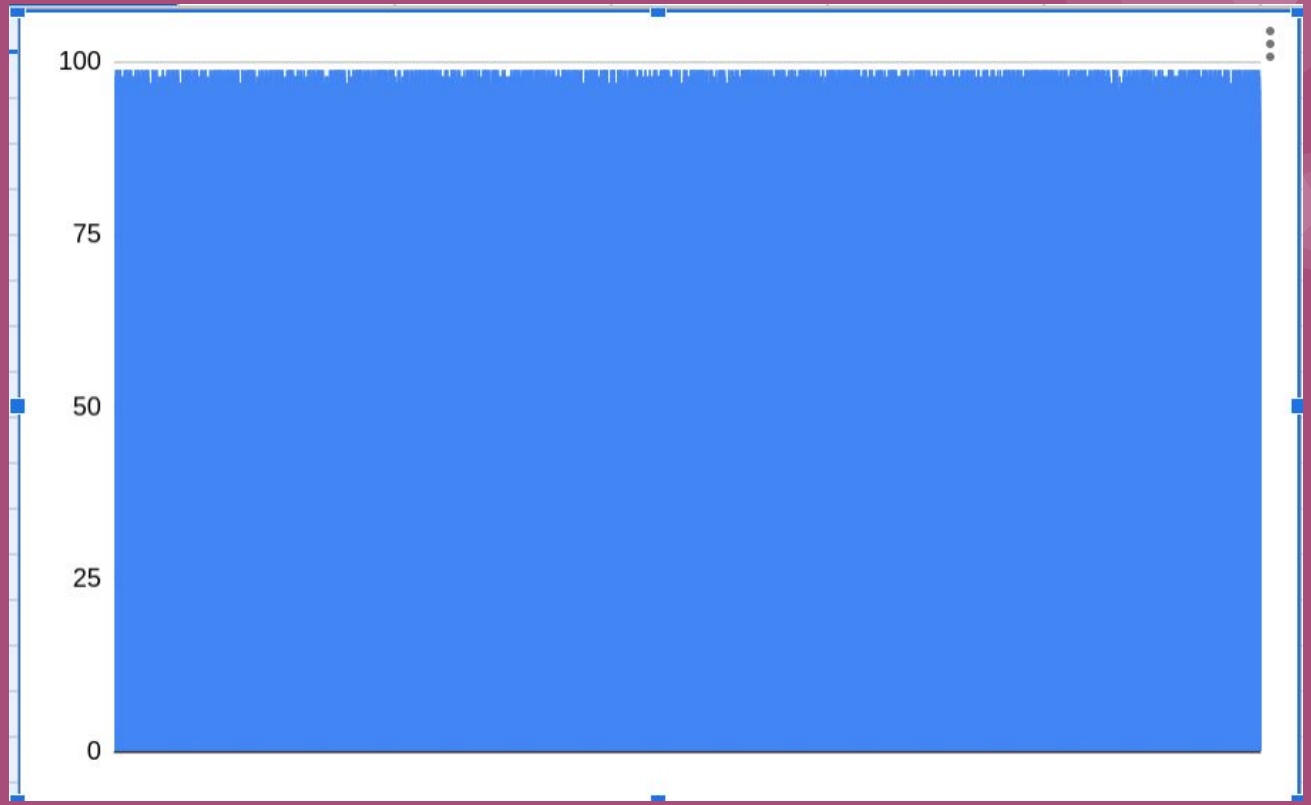
1-A: Construct large datasets taking random numbers from uniform distribution (UD)

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

1. **Uniform Distribution (UD)**: In statistics, a uniform distribution refers to a probability distribution where every value within a specified range is equally likely to occur. In other words, the probability of observing any particular outcome is constant.
2. **Generating Random Numbers**: To construct datasets with random numbers from a uniform distribution, we utilize random number generation techniques provided by programming languages or libraries. These techniques typically produce pseudorandom numbers, which appear to be random but are generated by deterministic algorithms.
3. **Range Specification**: When generating datasets, we define the range within which the random numbers should fall. This range is determined based on the requirements of the application or analysis. For example, if we're simulating test scores, the range might be from 0 to 100.
4. **Equal Probability**: In a uniform distribution, each value within the specified range has the same probability of being selected. This ensures that the generated dataset accurately represents the characteristics of the uniform distribution.
5. **Large Datasets**: The term "large datasets" typically implies a substantial number of data points. The size of the dataset depends on various factors, including the computational resources available and the specific requirements of the analysis or application.
6. **Implementation**: Generating large datasets with random numbers from a uniform distribution can be implemented using programming languages such as C, Python, or MATLAB. These languages provide built-in functions or libraries for random number generation, allowing developers to efficiently create datasets of any size.

Observation

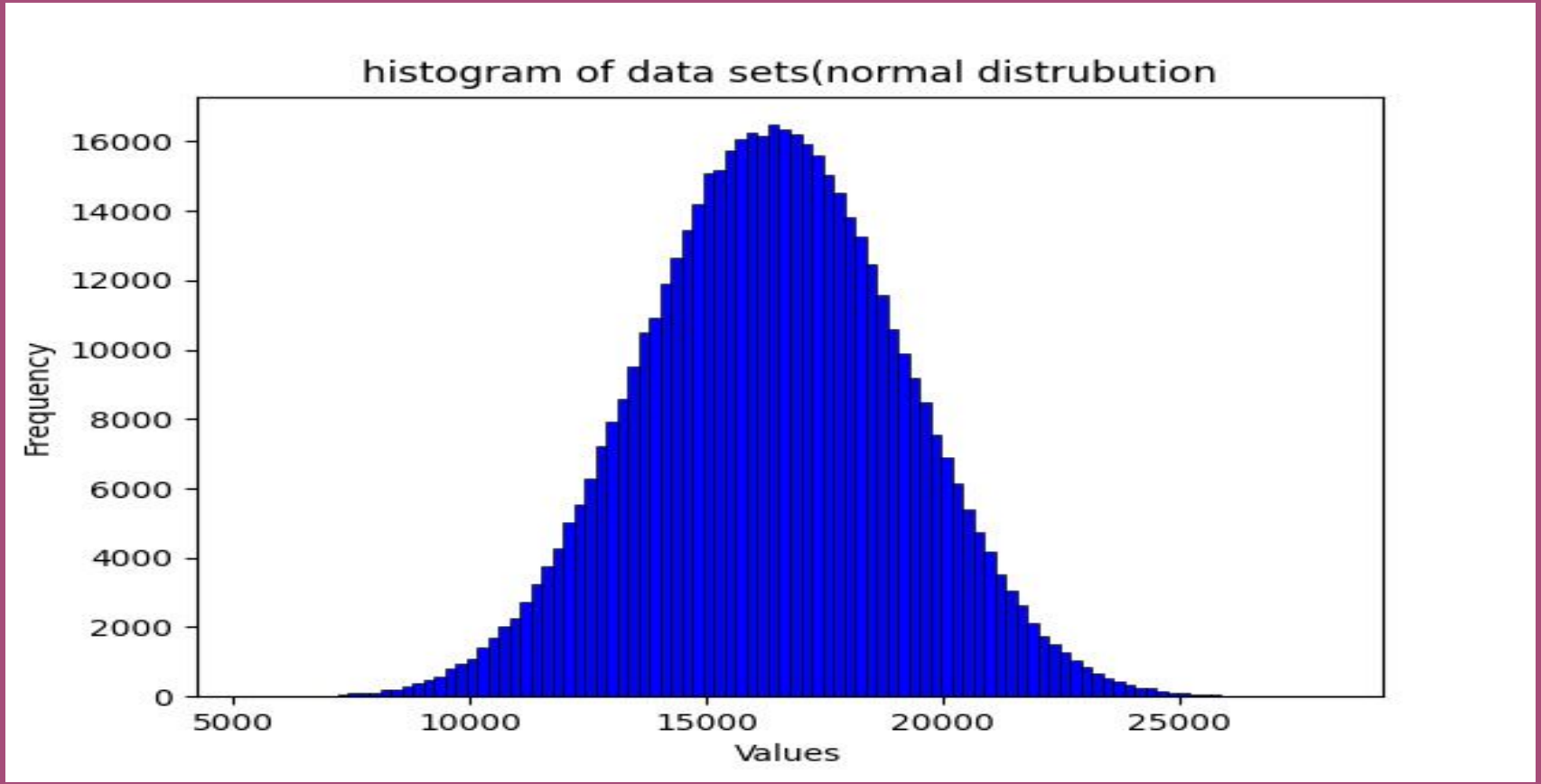
Uniform Distribution



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

1. **Normal Distribution (ND)**: The normal distribution is a fundamental probability distribution in statistics. It is characterized by a bell-shaped curve where the majority of the data points cluster around the mean, with symmetrical tails extending infinitely in both directions. The distribution is defined by two parameters: the mean (μ) and the standard deviation (σ).
2. **Bell-shaped Curve**: In a normal distribution, the probability density function (PDF) describes the likelihood of observing a particular value. The curve is symmetric around the mean, with the highest point at the mean. As you move away from the mean, the probability decreases, forming the characteristic bell shape.
3. **Generating Random Numbers**: Generating random numbers from a normal distribution typically involves using algorithms that rely on pseudorandom number generators. These algorithms produce numbers that mimic the statistical properties of true random numbers but are generated deterministically based on a seed value.
4. **Mean and Standard Deviation**: To construct datasets from a normal distribution, we specify the mean (μ) and standard deviation (σ) of the distribution. The mean determines the center of the distribution, while the standard deviation controls the spread or variability of the data around the mean.
5. **Central Limit Theorem**: The central limit theorem states that the sum or average of a large number of independent and identically distributed random variables tends to follow a normal distribution, regardless of the original distribution of the variables. This theorem justifies the widespread use of the normal distribution in modeling real-world phenomena.
6. **Large Datasets**: Generating large datasets with random numbers from a normal distribution involves generating a substantial number of data points that follow the specified distribution. The size of the dataset depends on factors such as the intended application, the complexity of the analysis, and available computational resources.
7. **Implementation**: Various programming languages and libraries offer functions for generating random numbers from a normal distribution. These functions typically allow users to specify the mean, standard deviation, and the size of the dataset to be generated.

Observation



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

Merge Sort (MS):

Merge Sort is a popular comparison-based sorting algorithm known for its efficiency, stability, and simplicity in implementation. It follows the divide-and-conquer paradigm, breaking down the sorting problem into smaller subproblems, solving them recursively, and then combining the solutions to produce the final sorted array.

Algorithm Steps:

1. **Divide:** Divide the unsorted array into two halves recursively until each subarray contains only one element.
2. **Conquer:** Sort the individual subarrays (single elements) by definition. At this stage, each subarray is already sorted.
3. **Combine:** Merge the sorted subarrays to produce a single sorted array.

Merge Operation:

The key operation in Merge Sort is the merging of two sorted arrays into a single sorted array. This operation is performed efficiently using a temporary array and two pointers to traverse the two input arrays. The elements from both arrays are compared, and the smaller element is placed in the temporary array. The pointers are then advanced accordingly until all elements are merged.

Code in C

```c
void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        // Same as (left+right)/2, but avoids overflow for large left and right
        int middle = left + (right - left) / 2;

        // Sort first and second halves
        mergeSort(arr, left, middle);
        mergeSort(arr, middle + 1, right);

        // Merge the sorted halves
        merge(arr, left, middle, right);
    }
}
```

Complexity Analysis:

- **Time Complexity:** Merge Sort has a time complexity of O(n log n) in all cases, where n is the number of elements in the array. This makes it efficient for large datasets.
- **Space Complexity:** Merge Sort has a space complexity of O(n) due to the auxiliary space required for the temporary arrays during the merge operation.

Stability:

Merge Sort is a stable sorting algorithm, meaning that it preserves the relative order of equal elements in the sorted output.

```
void merge(int arr[], int left, int middle, int right) {
    int i, j, k;
    int n1 = middle - left + 1;
    int n2 = right - middle;
    int leftArr[n1], rightArr[n2];
    // Copy data to temporary arrays leftArr[] and rightArr[]
    for (i = 0; i < n1; i++)
        leftArr[i] = arr[left + i];
    for (j = 0; j < n2; j++)
        rightArr[j] = arr[middle + 1 + j];
    // Merge the temporary arrays back into arr[left..right]
    i = 0;
    j = 0;
    k = left;
    while (i < n1 && j < n2) {
        if (leftArr[i] <= rightArr[j]) {
            arr[k] = leftArr[i];
            i++;
        } else {
            arr[k] = rightArr[j];
            j++;
        }
        k++;
    }
    // Copy the remaining elements of leftArr[], if there are any
    while (i < n1) {
        arr[k] = leftArr[i];
        i++;
        k++;
    }
    // Copy the remaining elements of rightArr[], if there are any
    while (j < n2) {
        arr[k] = rightArr[j];
        j++;
        k++;
    }
}
```

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

Quick Sort (QS):

Quick Sort is a widely-used sorting algorithm known for its efficiency and effectiveness in practice. It belongs to the category of comparison-based sorting algorithms and follows the divide-and-conquer approach. Quick Sort selects a pivot element and partitions the array into two subarrays, such that all elements smaller than the pivot are placed before it, and all elements larger than the pivot are placed after it. The subarrays are then recursively sorted.

Algorithm Steps:

1. **Partitioning:**
    - Choose a pivot element from the array. This pivot can be selected randomly, as the first, last, or middle element.
    - Rearrange the elements of the array so that all elements smaller than the pivot are placed before it, and all elements larger than the pivot are placed after it. The pivot itself is placed in its correct position in the final sorted array.
2. **Recursion:**
    - Recursively apply the same process to the subarrays on the left and right of the pivot. This step sorts the elements within each subarray.
3. **Combination:**
    - No combination step is needed in Quick Sort, as the elements are sorted in place during the partitioning step.

Complexity Analysis:

- **Time Complexity:** Quick Sort typically has an average-case time complexity of O(n log n). However, in the worst-case scenario (e.g., if the pivot is always the smallest or largest element), Quick Sort can degrade to O(n^2). Nevertheless, Quick Sort is usually faster than other O(n log n) algorithms in practice due to its efficient partitioning and cache efficiency.
- **Space Complexity:** Quick Sort has a space complexity of O(log n) due to the recursion stack space required for the recursive calls.

```c
// Quick Sort
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            // Swap arr[i] and arr[j]
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }
    // Swap arr[i+1] and arr[high] (pivot)
    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;

    return i + 1;
}

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);

        // Recursively sort elements before and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

- There are different versions of Quicksort that pick the pivot in different ways:
    - **Always pick the first element as the pivot:** This is the simplest implementation of Quicksort.
    - **Always pick the last element as the pivot:** Another straightforward approach to selecting the pivot.
    - **Pick a random element as the pivot:** This approach helps to mitigate the risk of worst-case scenarios and improves the average-case performance.
    - **Pick the median as the pivot:** Selecting the median as the pivot minimizes the number of comparisons required to partition the array optimally.
- The key process in Quicksort is the `partition()` function. The goal of the partitioning step is to rearrange the elements in the array such that:
    - Elements smaller than the pivot are placed before it.
    - Elements larger than the pivot are placed after it.
    - The pivot itself is placed in its correct position in the final sorted array.

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.

Theoretical Analysis for Merge Sort (MS):

Merge Sort has a well-defined number of comparisons and swaps based on the size of the input array. Here's a theoretical analysis:

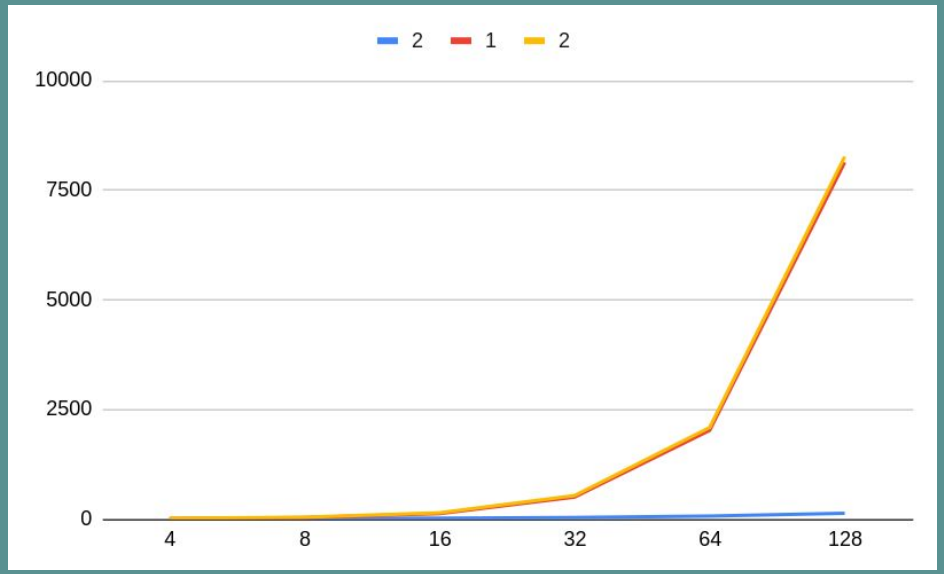
- **Comparisons:**
  - In Merge Sort, every element is compared at least once during the merge phase.
  - For an array of size n, the number of comparisons is O(n log n).
  - This is because the merge operation divides the array into halves recursively until single-element arrays are obtained, and then merges them back together in sorted order.
- **Swaps:**
  - Merge Sort typically does not involve any swaps, as it creates new arrays during the merge phase rather than modifying the original array in place.
  - Therefore, the number of swaps made by Merge Sort is generally minimal or zero.

Theoretical Analysis for Quick Sort (QS):

Quicksort performance depends heavily on the choice of pivot and the input data. Here's a general theoretical analysis:

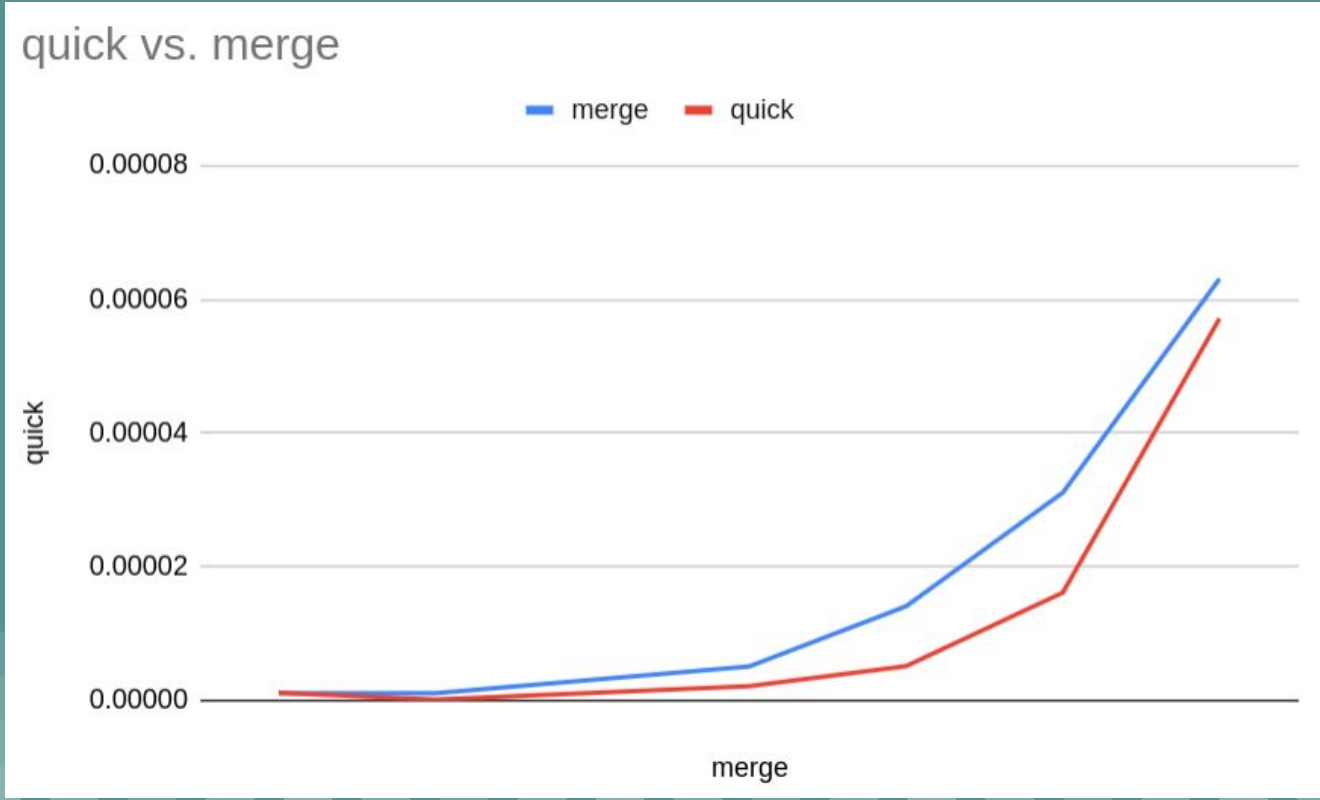
- **Comparisons:**
  - In the best-case scenario, where the pivot always divides the array into two roughly equal halves, Quick Sort performs O(n log n) comparisons, similar to Merge Sort.
  - However, in the worst-case scenario, where the pivot is always the smallest or largest element, Quick Sort may degrade to O(n^2) comparisons.
  - On average, Quick Sort performs O(n log n) comparisons.
- **Swaps:**
  - The number of swaps in Quick Sort depends on the number of inversions in the input array.
  - In the best-case scenario, where the array is already sorted or contains minimal inversions, Quick Sort requires minimal swaps.
  - In the worst-case scenario, where the array is in reverse order, Quick Sort may require O(n^2) swaps.
  - On average, Quick Sort requires O(n log n) swaps.

This is the analysis for quick sort.

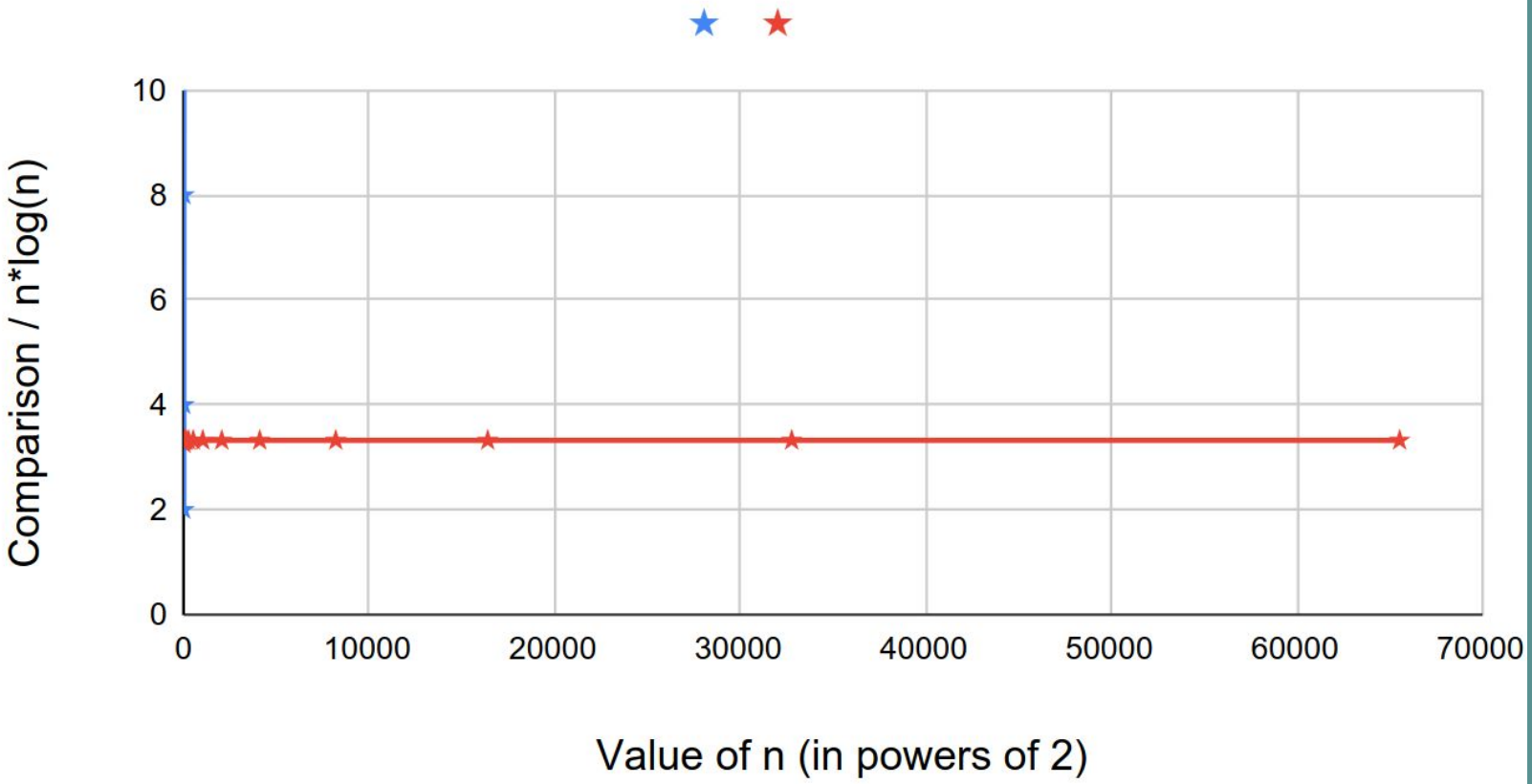


Size	Comparisons	Swaps
2	1	2
4	6	9
8	28	35
16	120	135
32	496	527
64	2016	2079
128	8128	8255

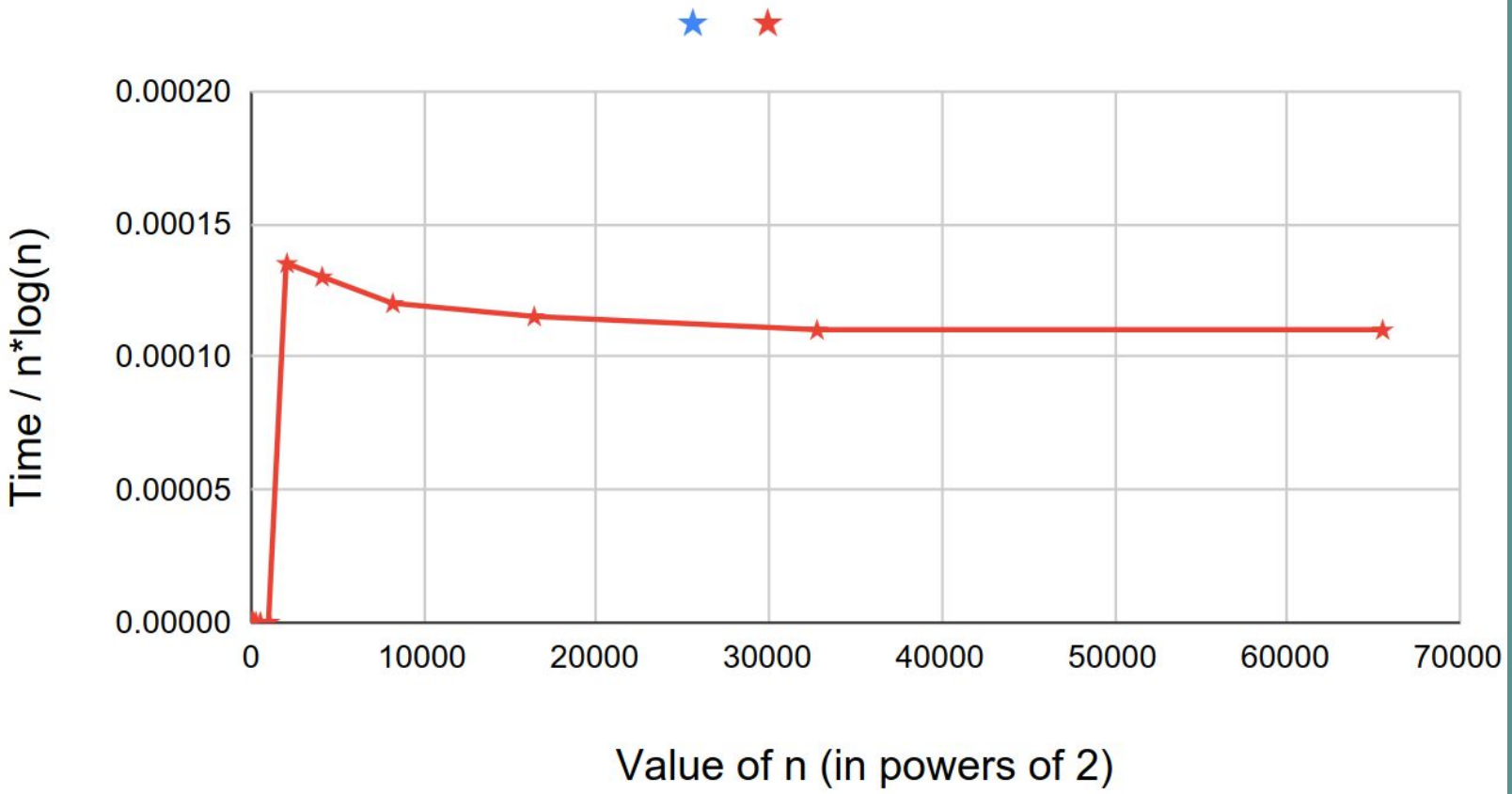
Observation: From the graph, we see that both time ratio and comparison ratio converges to constant as n goes to very big sizes. It means that MERGE SORT ALGORITHM has O(nlgn) complexity.



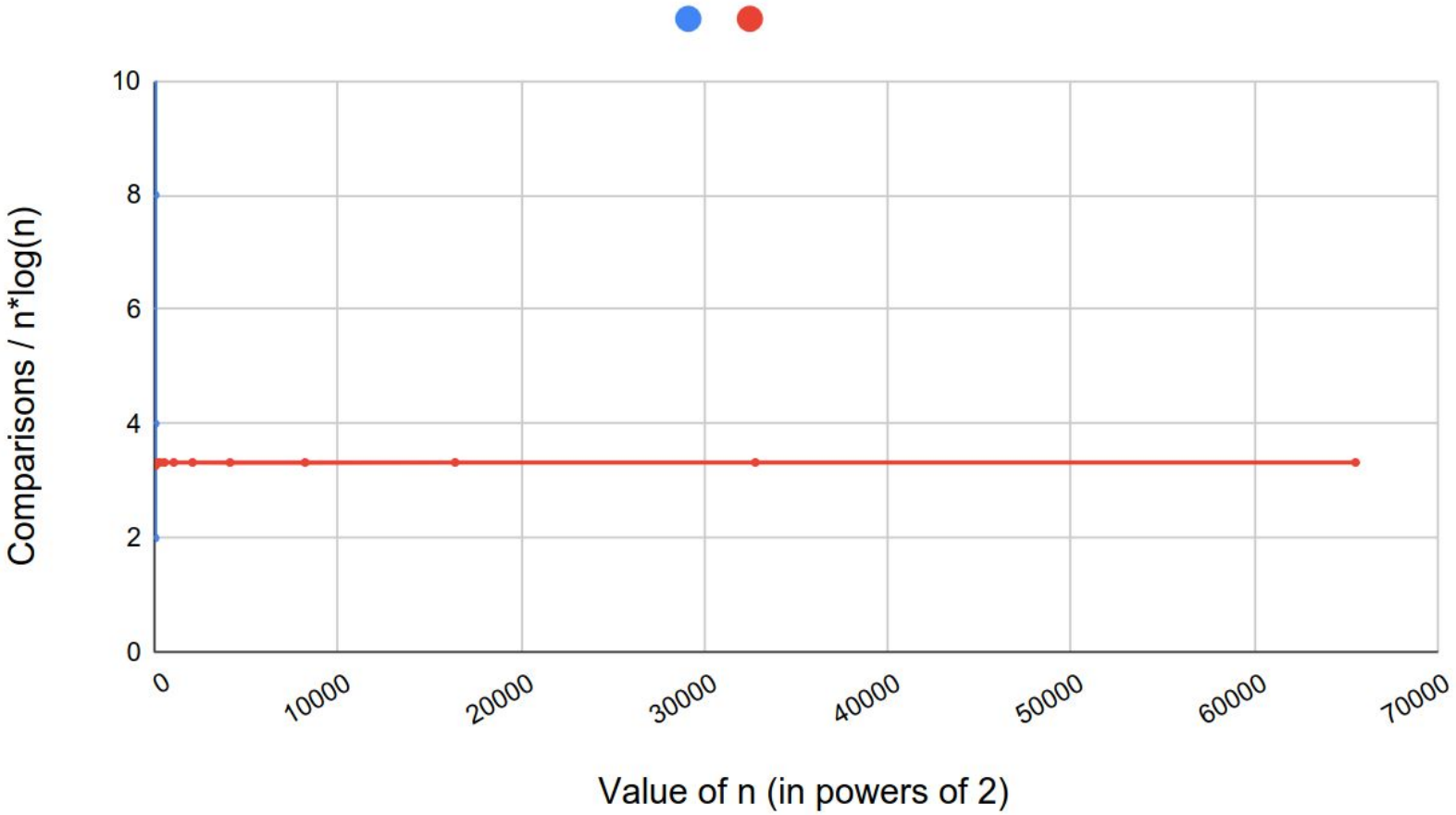
Merge Sort (for Uniform Distribution)



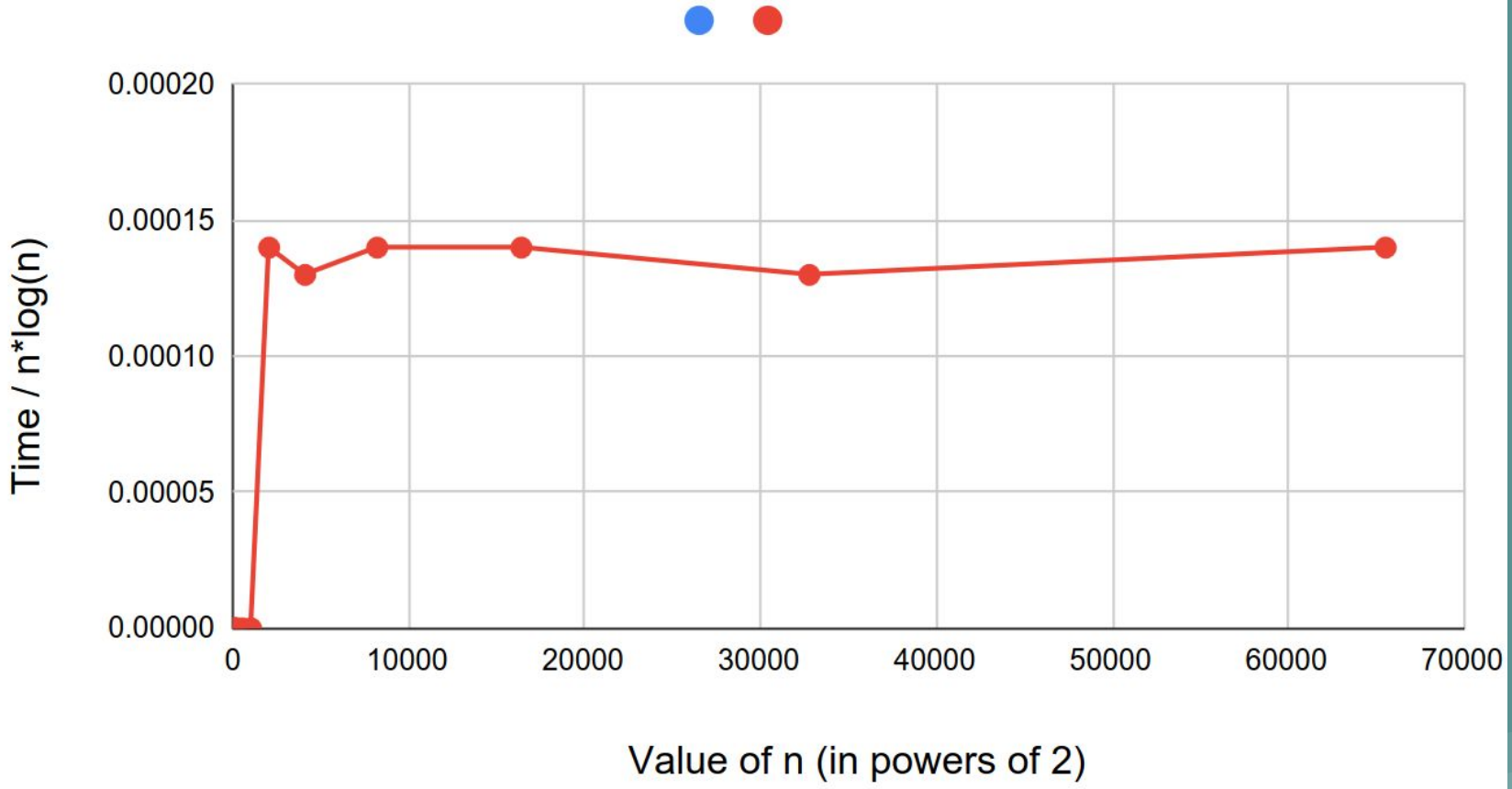
Merge Sort (for Uniform Distribution)



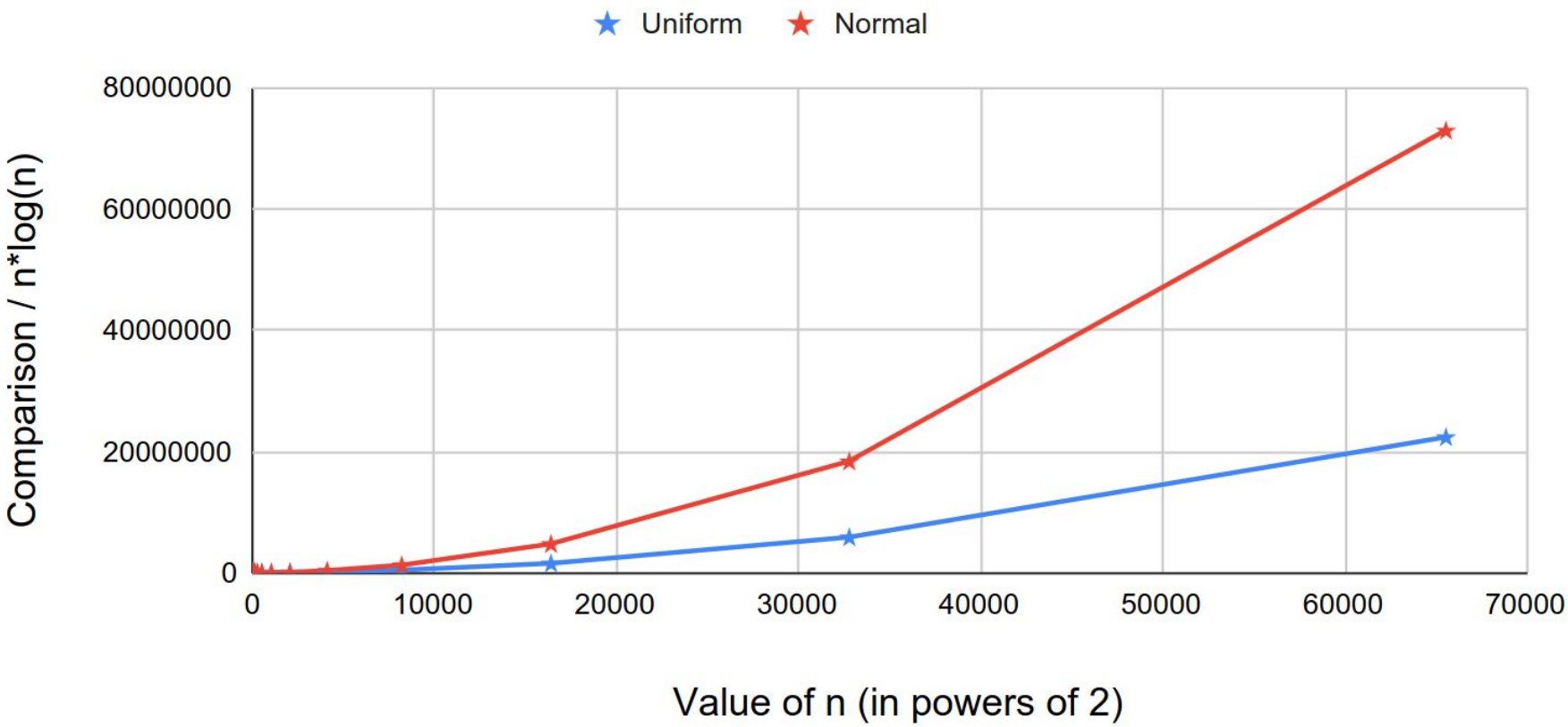
Merge Sort (for Normal Distribution)



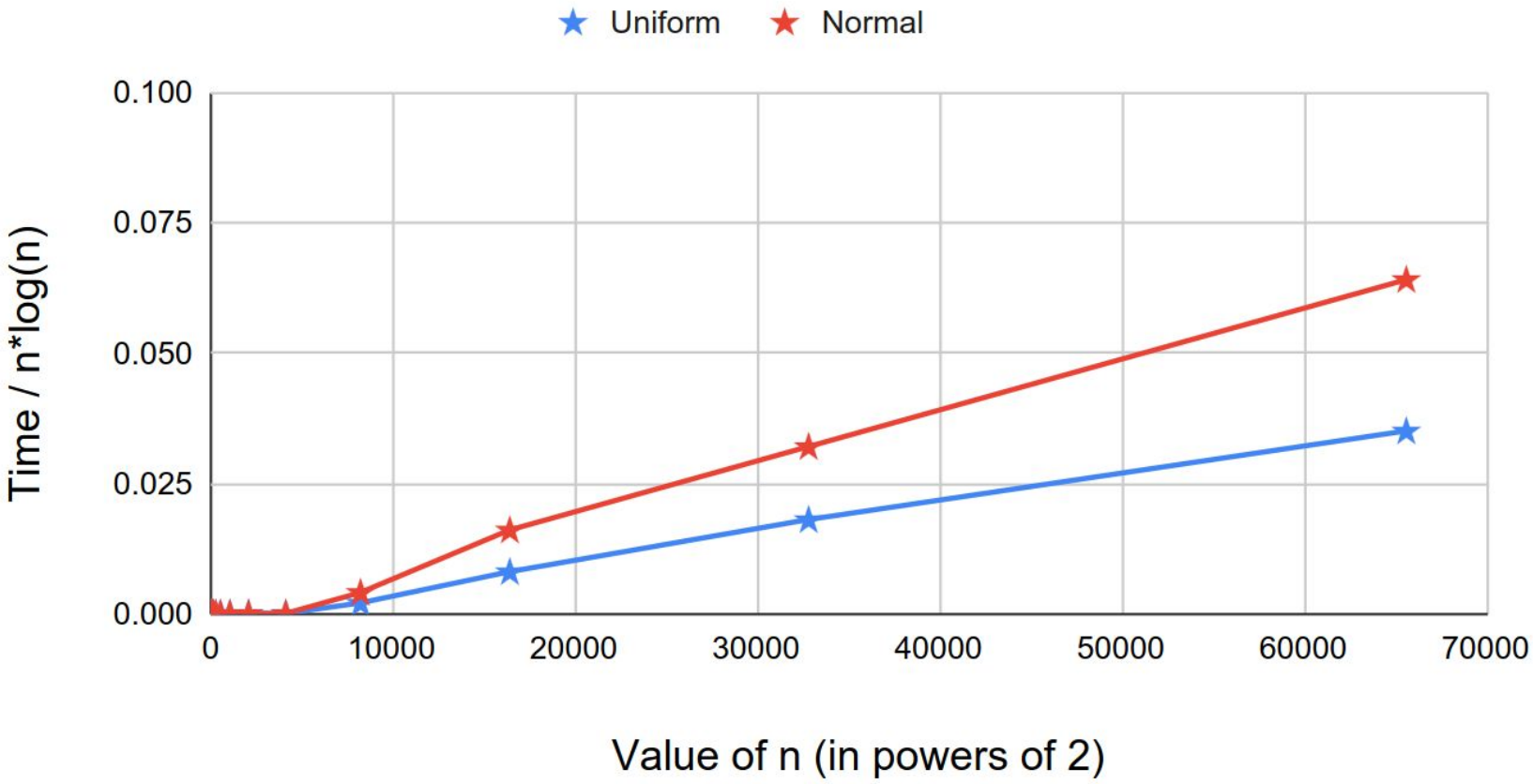
Merge Sort (for Normal Distribution)



Quick Sort (for Normal and Uniform Distribution)

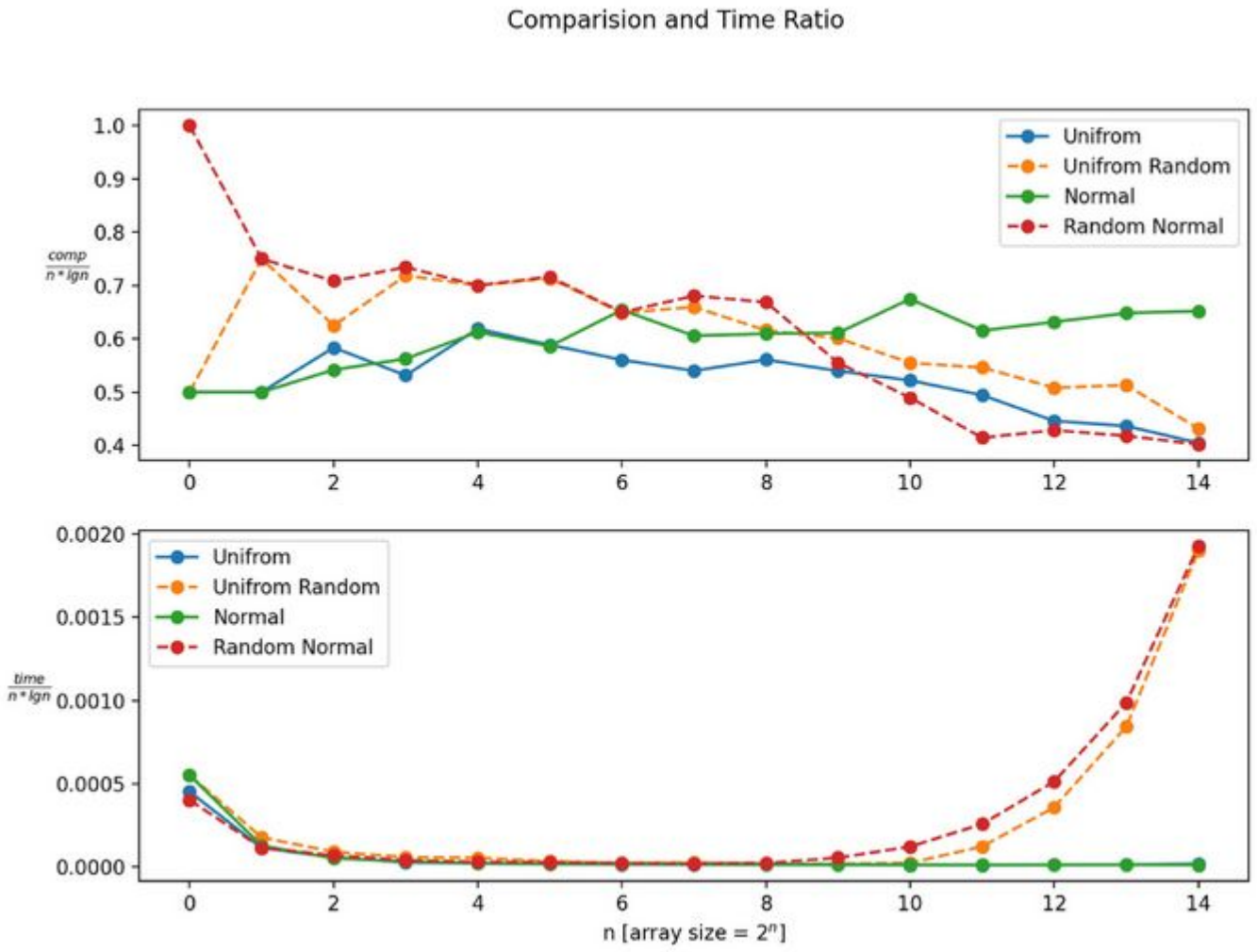


Quick Sort (for both Uniform and Normal)



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.

1. **Randomized QuickSort (RQS)**:
    - Randomized QuickSort addresses the issue of pivot selection bias in standard QuickSort by randomly selecting the pivot element.
    - Instead of choosing a fixed pivot position (such as the first, last, or middle element), RQS randomly selects a pivot element from within the array.
    - By randomizing the pivot selection, RQS aims to mitigate the likelihood of encountering worst-case scenarios and improve the average-case performance of the algorithm.
    - The random selection of the pivot element helps to distribute the elements more evenly during partitioning, leading to better average-case performance and reducing the likelihood of quadratic time complexity.
    - The choice of pivot can significantly impact the performance of QuickSort, especially in scenarios where the dataset is already sorted or nearly sorted. In the worst-case scenario, QuickSort can degrade to quadratic time complexity (O(n^2)), particularly if the pivot is consistently chosen as the smallest or largest element.


Comparision and Time Ratio
Unifrom
Unifrom Random
Normal
Random Normal
comp / n * lgn
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0
2
4
6
8
10
12
14
Unifrom
Unifrom Random
Normal
Random Normal
time / n * lgn
0.0020
0.0015
0.0010
0.0005
0.0000
n [array size = 2^n]

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.

- **Normalization:** Normalization is the process of scaling data to fit within a specific range. In this case, we aim to normalize both the Uniform Distribution (UD) and Normal Distribution (ND) datasets to the range [0, 1].
- **Bucket Sort (BS):** Bucket 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, typically with another sorting algorithm or recursively with the same bucket sort algorithm. Finally, the elements from the sorted buckets are combined to produce the sorted array.

Normalization Process:

- For each dataset, find the minimum and maximum values.
- Subtract the minimum value from each data point and divide the result by the range (maximum - minimum) to scale the data to the range [0, 1].

Bucket Sort Algorithm:

- Create an array of buckets, each capable of holding a range of values.
- Distribute the normalized data points into the appropriate buckets.
- Sort each bucket, either using another sorting algorithm or recursively applying bucket sort.
- Concatenate the sorted buckets to obtain the final sorted array.

Experimental Procedure:

2. **Data Generation:** Generate UD and ND datasets.
3. **Normalization:** Normalize both datasets to the range [0, 1].
4. **Bucket Sort:** Implement Bucket Sort on both normalized datasets.
5. **Performance Measurement:** Measure the execution time and count the number of operations (comparisons, swaps) performed during the sorting process.
6. **Average Complexity Analysis:** Calculate the average number of operations and execution time over multiple runs for each dataset size.
7. **Inference:** Compare the average complexities obtained for UD and ND datasets. Analyze how the distribution of data affects the performance of Bucket Sort.

```c
typedef struct node{
    int val;
    struct node *nxt;
}node;
node *InsertionSort(node *head,int *c,int *s){
    if(head == NULL || head->nxt == NULL){
        return head;
    }
    node *sorted = NULL;
    node *current = head;
    while(current != NULL){
        node *nextnode = current->nxt;
        (*c)++;
        if(sorted == NULL || current->val <= sorted->val){
            current->nxt = sorted;
            sorted = current;
        }
        else{
            node *temp = sorted;
            (*c)++;
            while(temp->nxt != NULL && temp->nxt->val < current->val){
                temp = temp->nxt; (*c)++;
            }
            current->nxt = temp->nxt;
            temp->nxt = current;
            (*s)++;
        }
        current = nextnode;
    }
    return sorted;
}
```

int index = floor((a[i] / max) * n);

This part is doing the normalization of dataset

```
void BucketSort(int *a,int n,int *c,int *s){
    int max = -1 , m;
    for(int i = 0; i < n; i++){
        m = a[i];
        if(m > max) {max = m;}
    }
    // printf("\nThe max element is %d\n.",max);
    node **buckets = (node **)malloc((n + 1) * sizeof(node *));
    for(int j = 0; j < n + 1; j++){
        buckets[j] = NULL;
    }
    for(int i = 0; i < n; i++){
        node *current = (node *)malloc(sizeof(node));
        int index = floor((a[i] / max) * n);
        current->val = a[i];
        current->nxt = buckets[index];
        buckets[index] = current;
    }
    for(int i = 0; i < n + 1; i++){
        buckets[i] = InsertionSort(buckets[i],c,s);
    }
    int k = 0;
    for(int i = 0; i < n + 1; i++){
        node *ptr = buckets[i];
        while(ptr != NULL){
            a[k] = ptr->val; k++;
            ptr = ptr->nxt;
        }
    }
}
```

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 (MoM):** The Median of Medians algorithm is used to find an approximate median of an array by dividing it into smaller subarrays, finding the median of each subarray, and then recursively applying the process until a small enough subarray is obtained. The median of these medians is chosen as the pivot for partitioning the array, ensuring a balanced partition and improving worst-case performance.
- **Linear Median Selection:** The linear median selection algorithm utilizes the MoM approach to efficiently find the median of an unsorted array in linear time complexity, even in the worst-case scenario.

Experimental Procedure:

1. **MoM Partitioning:**
   - Divide the array into subarrays of size 5 (or any fixed size).
   - Find the median of each subarray using a sorting algorithm or another median selection algorithm.
   - Recursively find the median of medians until a single median value is obtained.
2. **Choose MoM as Pivot:**
   - Select the median of medians as the pivot element for partitioning the array.
3. **Partitioning and Recursion:**
   - Partition the array around the MoM pivot element.
   - Recursively apply the algorithm to the appropriate partition until the median is found.
4. **Correctness Verification:**
   - After sorting, verify that the element at the middle position is indeed the median of the array.
   - Compare the result with a known correct median value to ensure correctness.

Conclusion:

Implementing the worst-case linear median selection algorithm using the MoM approach allows us to efficiently find the median of an unsorted array, even in scenarios where other methods might exhibit poor performance. By verifying the correctness of the algorithm's output, we can ensure its reliability and effectiveness for median selection tasks.

```c
int select(int *,int ,int ,int );

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

int MedianOfMedians(int *a,int p,int r){
    int num = r - p + 1;
    int Total_Groups = (num + 4) / 5;
    int *medians = (int *)malloc(Total_Groups * sizeof(int));
    int index = 0;
    for(int j = p; j <= r; j += 5){
        int grp_end = j + 4 < r ? j + 4 : r;
        // int grp_size = j + 4 <= r ? 5 : r - (j+4);
        int grp_size = grp_end - j + 1;
        for(int k = j; k <= grp_end; k++){
            for(int m = k+1; m <= grp_end; m++){
                if(a[m] < a[k]){
                    swap(&a[m],&a[k]);
                }
            }
        }
        // printf("\nAfter partial sorting : elements are : ");
        // for(int k = j; k <= grp_end; k++){printf("\t%d\t",a[k]);}
        medians[index++] = a[j + (grp_size / 2)];
    }
    return select(medians,0,Total_Groups-1,Total_Groups/2);
}
```

```
int partition(int *a,int p,int r,int v){
    int index = p - 1;
    for(int j = p; j < r; j++){
        if(a[j] == v) {swap(&a[r],&a[j]);}
    }
    for(int j = p; j < r ; j++){
        if(a[j] <= v){
            index++; swap(&a[j],&a[index]);
        }
    }
    swap(&a[index + 1],&a[r]);
    return (index + 1);
}

int select(int *a,int p,int r,int k){
    if(p == r){
        return a[p];
    }
    int m = MedianOfMedians(a,p,r);
    // printf("\nThe median now is : %d.",m);
    int index = partition(a,p,r,m);
    int x = index - p + 1;
    if(k == x){
        return a[index];
    }
    else if(k < x){
        return select(a,p,index-1,k);
    }
    else{
        return select(a,index+1,r,k - x);
    }
}
```

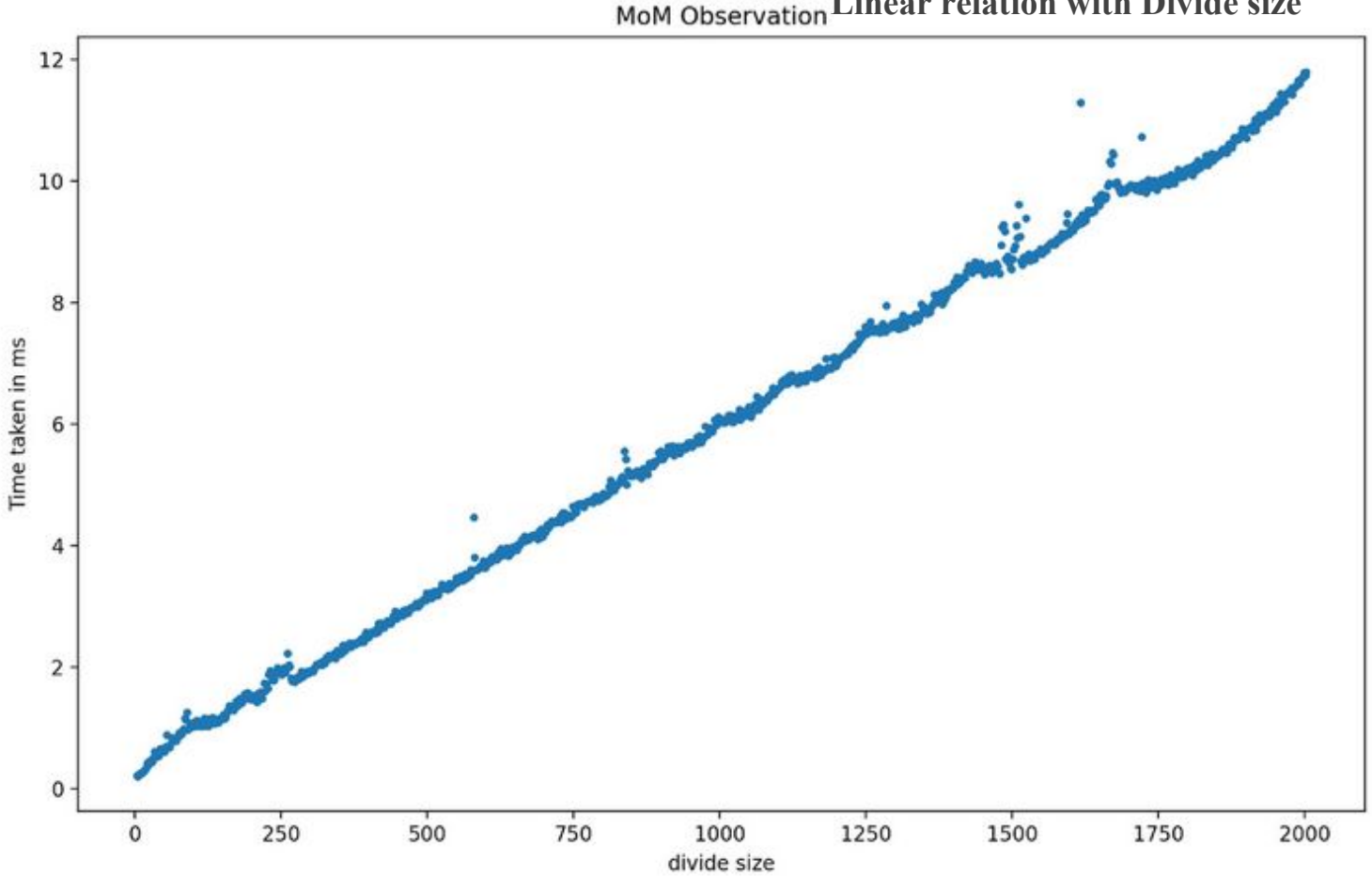
```c
int checkCorrectness(int *a,int n,int res,int k){
    for(int i = 1; i < n; i++){
        int j = i - 1; int key = a[i];
        while(j >= 0 && a[j] > key){
            a[j+1] = a[j]; j--;
        }
        a[j+1] = key;
    }
    // printf("\nAfter sorting the elements of the array are : \n");
    // for(int i = 0; i < n; i++){
    //     printf("%d\t",a[i]);
    // }
    printf("\nThe actual value is %d and res comes out to be %d.",a[n/2],res);
    if(a[k] == res){
        return 1;
    }
    return 0;
}

int check(int *a,int n){
    int sorted = 1;
    for(int i = 0; i < n-1; i++){
        if(a[i] <= a[i+1]) {}
        else{
            sorted = 0;
        }
    }
    printf("\n");
    if(sorted == 1){
        printf("The dataset entered is sorted.");
    }
    else{
        printf("The dataset entered is not sorted.");
    }
}
```

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

- The worst-case linear time selection algorithm aims to find the median in linear time.
- Adjusting the size of the trivial partitions, such as 3, 5, 7, can impact the time complexity of the algorithm.
- The time complexity of the algorithm using the MoM approach is O(n), where n is the size of the input array.
- Varying the partition size can influence the number of comparisons and recursive steps, affecting the overall time taken for the selection process.
- The choice of partition size can impact the algorithm's performance and time taken for selection.

Linear relation with Divide size



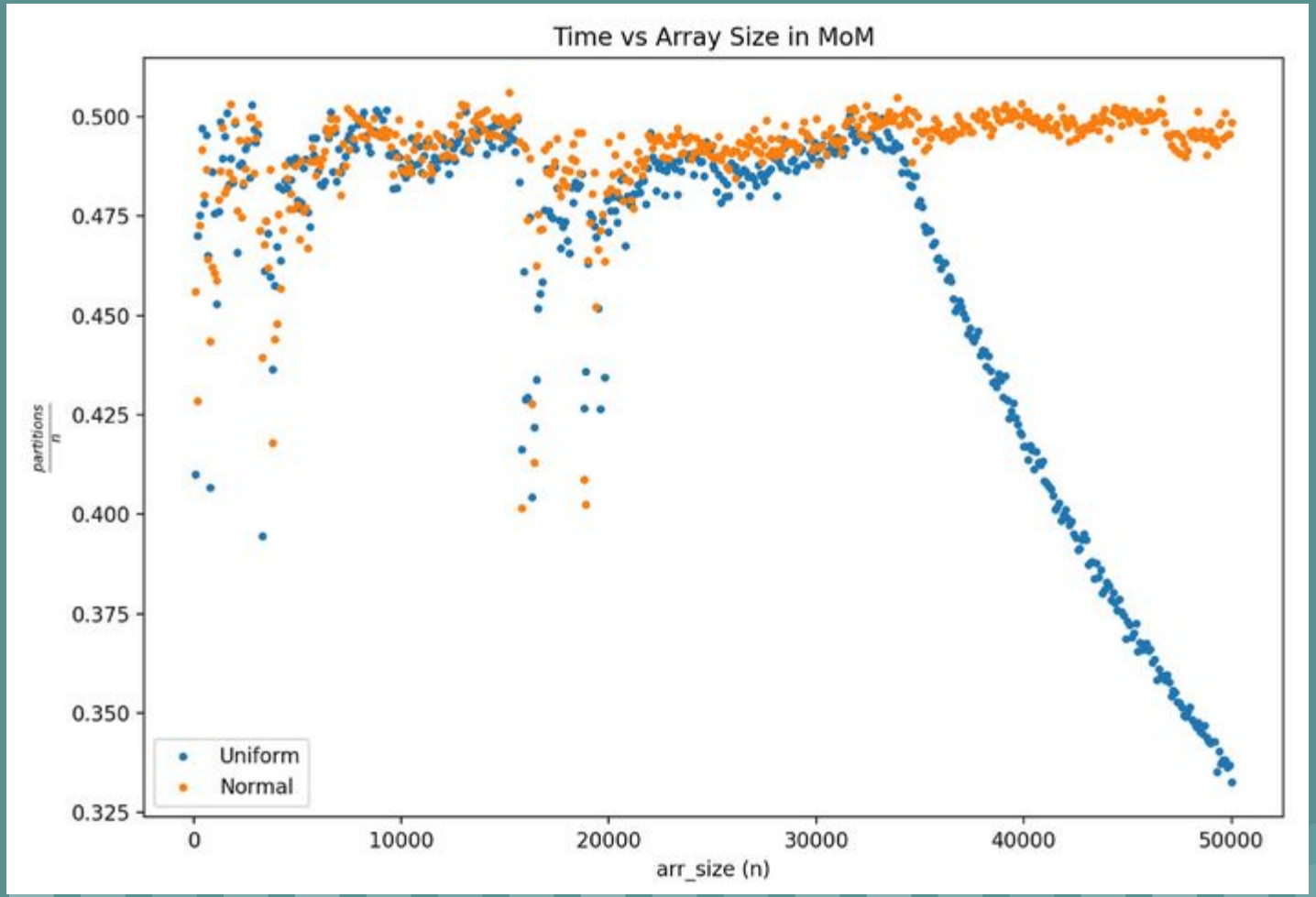
Code in C

```
int MedianOfMedians(int *a,int p,int r,int s){
    int num = r - p + 1;
    int Total_Groups = (num + s - 1) / s;
    int *medians = (int *)malloc(Total_Groups * sizeof(int));
    int index = 0;
    for(int j = p; j <= r; j += 5){
        int grp_end = j + s - 1 < r ? j + s - 1 : r;
        // int grp_size = j + 4 <= r ? 5 : r - (j+4);
        int grp_size = grp_end - j + 1;
        for(int k = j; k <= grp_end; k++){
            for(int m = k+1; m <= grp_end; m++){
                if(a[m] < a[k]){
                    swap(&a[m],&a[k]);
                }
            }
        }
        printf("\nAfter partial sorting, elements are : ");
        for(int k = j; k <= grp_end; k++){printf("%d\t",a[k]);}
        medians[index++] = a[j + (grp_size / 2)];
        int c = 0;
        printf("\nThe elements in medians list are : \n");
        for(int k = 0; k < index; k++){
            printf("%d ; ",medians[k]); c++;
        }
        printf("\n\n------------------------------\nThe no of elements in median list is : %d.\n------------------------------",c);
    }
    return select(medians,0,Total_Groups-1,Total_Groups/2,s);
}
```

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.

- **The worst-case linear time selection algorithm, using the MoM approach, aims to find the median in linear time.**
- **Adjusting the size of the trivial partitions, such as 3, 5, 7, can impact the time complexity of the algorithm.**
- **The time complexity of the algorithm using the MoM approach is O(n), where n is the size of the input array.**
- **Varying the partition size can influence the number of comparisons and recursive steps, affecting the overall time taken for the selection process.**

The experimental observation of rearranging the elements of the datasets (both UD and ND) and analyzing the resulting partitions can provide insights into how the choice of the pivotal element (MoM) and the partition size influence the algorithm's behavior and the obtained splits. The provided search results offer insights into the analysis of linear-time selection algorithms and the impact of different partition sizes on the algorithm's performance, which can be referenced to further explore the theoretical and practical implications of these experiments.



Result of MOM is very close to actual median. So MOM can guarantee a good pivot element for Quick Sort algorithm

Thank You